Design and Analysis of Algorithms

Sohail Aslam

January 2004

Contents

Chapter 1

Introduction

1.1 Origin of word: *Algorithm*

The word *Algorithm* comes from the name of the muslim author *Abu Ja'far Mohammad ibn Musa al-Khowarizmi*. He was born in the eighth century at Khwarizm (Kheva), a town south of river Oxus in present Uzbekistan. Uzbekistan, a Muslim country for over a thousand years, was taken over by the Russians in 1873.

His year of birth is not known exactly. Al-Khwarizmi parents migrated to a place south of Baghdad when he was a child. It has been established from his contributions that he flourished under Khalifah Al-Mamun at Baghdad during 813 to 833 C.E. Al-Khwarizmi died around 840 C.E.

Much of al-Khwarizmi's work was written in a book titled *al Kitab al-mukhatasar fi hisab al-jabr wa'l-muqabalah* (The Compendious Book on Calculation by Completion and Balancing). It is from the titles of these writings and his name that the words *algebra* and *algorithm* are derived. As a result of his work, al-Khwarizmi is regarded as the most outstanding mathematician of his time

1.2 Algorithm: Informal Definition

An algorithm is any well-defined computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into output.

1.3 Algorithms, Programming

A good understanding of algorithms is essential for a good understanding of the most basic element of computer science: programming. Unlike a program, an algorithm is a mathematical entity, which is independent of a specific programming language, machine, or compiler. Thus, in some sense, algorithm design is all about the mathematical theory behind the design of good programs.

Why study algorithm design? There are many facets to good program design. Good algorithm design is one of them (and an important one). To be really complete algorithm designer, it is important to be aware of programming and machine issues as well. In any important programming project there are two major types of issues, macro issues and micro issues.

Macro issues involve elements such as how does one coordinate the efforts of many programmers working on a single piece of software, and how does one establish that a complex programming system satisfies its various requirements. These macro issues are the primary subject of courses on software engineering.

A great deal of the programming effort on most complex software systems consists of elements whose programming is fairly mundane (input and output, data conversion, error checking, report generation). However, there is often a small critical portion of the software, which may involve only tens to hundreds of lines of code, but where the great majority of computational time is spent. (Or as the old adage goes: 80% of the execution time takes place in 20% of the code.) The micro issues in programming involve how best to deal with these small critical sections.

It may be very important for the success of the overall project that these sections of code be written in the most efficient manner possible. An unfortunately common approach to this problem is to first design an inefficient algorithm and data structure to solve the problem, and then take this poor design and attempt to fine-tune its performance by applying clever coding tricks or by implementing it on the most expensive and fastest machines around to boost performance as much as possible. The problem is that if the underlying design is bad, then often no amount of fine-tuning is going to make a substantial difference.

Before you implement, first be sure you have a good design. This course is all about how to design good algorithms. Because the lesson cannot be taught in just one course, there are a number of companion courses that are important as well. CS301 deals with how to design good data structures. This is not really an independent issue, because most of the fastest algorithms are fast because they use fast data structures, and vice versa. In fact, many of the courses in the computer science program deal with efficient algorithms and data structures, but just as they apply to various applications: compilers, operating systems, databases, artificial intelligence, computer graphics and vision, etc. Thus, a good understanding of algorithm design is a central element to a good understanding of computer science and good programming.

1.4 Implementation Issues

One of the elements that we will focus on in this course is to try to study algorithms as pure mathematical objects, and so ignore issues such as programming language, machine, and operating system. This has the advantage of clearing away the messy details that affect implementation. But these details may be very important.

For example, an important fact of current processor technology is that of locality of reference. Frequently accessed data can be stored in registers or cache memory. Our mathematical analysis will usually ignore these issues. But a good algorithm designer can work within the realm of mathematics, but still keep an open eye to implementation issues down the line that will be important for final implementation. For example, we will study three fast sorting algorithms this semester, heap-sort, merge-sort, and quick-sort. From our mathematical analysis, all have equal running times. However, among the three (barring any extra considerations) quick sort is the fastest on virtually all modern machines. Why? It is the best from the perspective of locality of reference. However, the difference is typically small (perhaps 10-20% difference in running time).

Thus this course is not the last word in good program design, and in fact it is perhaps more accurately just the first word in good program design. The overall strategy that I would suggest to any programming would be to first come up with a few good designs from a mathematical and algorithmic perspective. Next prune this selection by consideration of practical matters (like locality of reference). Finally prototype (that is, do test implementations) a few of the best designs and run them on data sets that will arise in your application for the final fine-tuning. Also, be sure to use whatever development tools that you have, such as profilers (programs which pin-point the sections of the code that are responsible for most of the running time).

1.5 Course in Review

This course will consist of four major sections. The first is on the mathematical tools necessary for the analysis of algorithms. This will focus on asymptotics, summations, recurrences. The second element will deal with one particularly important algorithmic problem: sorting a list of numbers. We will show a number of different strategies for sorting, and use this problem as a case-study in different techniques for designing and analyzing algorithms.

The final third of the course will deal with a collection of various algorithmic problems and solution techniques. Finally we will close this last third with a very brief introduction to the theory of NP-completeness. NP-complete problem are those for which no efficient algorithms are known, but no one knows for sure whether efficient solutions might exist.

1.6 Analyzing Algorithms

In order to design good algorithms, we must first agree the criteria for measuring algorithms. The emphasis in this course will be on the design of efficient algorithm, and hence we will measure algorithms in terms of the amount of computational resources that the algorithm requires. These resources include mostly running time and memory. Depending on the application, there may be other elements that are taken into account, such as the number disk accesses in a database program or the communication bandwidth in a networking application.

In practice there are many issues that need to be considered in the design algorithms. These include issues such as the ease of debugging and maintaining the final software through its life-cycle. Also, one of the luxuries we will have in this course is to be able to assume that we are given a clean, fully-specified mathematical description of the computational problem. In practice, this is often not the case, and the algorithm must be designed subject to only partial knowledge of the final specifications. Thus, in practice it is often necessary to design algorithms that are simple, and easily modified if problem parameters and specifications are slightly modified. Fortunately, most of the algorithms that we will discuss in this class are quite simple, and are easy to modify subject to small problem variations.

1.7 Model of Computation

Another goal that we will have in this course is that our analysis be as independent as possible of the variations in machine, operating system, compiler, or programming language. Unlike programs, algorithms to be understood primarily by people (i.e. programmers) and not machines. Thus gives us quite a bit of flexibility in how we present our algorithms, and many low-level details may be omitted (since it will be the job of the programmer who implements the algorithm to fill them in).

But, in order to say anything meaningful about our algorithms, it will be important for us to settle on a mathematical model of computation. Ideally this model should be a reasonable abstraction of a standard generic single-processor machine. We call this model a *random access machine* or RAM.

A RAM is an idealized machine with an infinitely large random-access memory. Instructions are executed one-by-one (there is no parallelism). Each instruction involves performing some basic operation on two values in the machines memory (which might be characters or integers; let's avoid floating point for now). Basic operations include things like assigning a value to a variable, computing any basic arithmetic operation (+, - , $\times$, integer division) on integer values of any size, performing any comparison (e.g. $x \leq 5$) or boolean operations, accessing an element of an array (e.g. $A[10]$). We assume that each basic operation takes the same constant time to execute.

This model seems to go a good job of describing the computational power of most modern (nonparallel) machines. It does not model some elements, such as efficiency due to locality of reference, as described in the previous lecture. There are some "loop-holes" (or hidden ways of subverting the rules) to beware of. For example, the model would allow you to add two numbers that contain a billion digits in constant time. Thus, it is theoretically possible to derive nonsensical results in the form of efficient RAM programs that cannot be implemented efficiently on any machine. Nonetheless, the RAM model seems to be fairly sound, and has done a good job of modelling typical machine technology since the early 60's.

1.8 Example: 2-dimension maxima

Let us do an example that illustrates how we analyze algorithms. Suppose you want to buy a car. You want the pick the fastest car. But fast cars are expensive; you want the cheapest. You cannot decide which is more important: speed or price. Definitely do not want a car if there is another that is both faster and cheaper. We say that the fast, cheap car *dominates* the slow, expensive car relative to your selection criteria. So, given a collection of cars, we want to list those cars that are not dominated by any other.

Here is how we might model this as a formal problem.

- Let a point p in 2-dimensional space be given by its integer coordinates, $p = (p.x, p.y)$.

- A point p is said to be dominated by point q if $p.x \leq q.x$ and $p.y \leq q.y$.
- Given a set of n points, $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space a point is said to be maximal if it is not dominated by any other point in P.

The car selection problem can be modelled this way: For each car we associate (x, y) pair where x is the speed of the car and y is the negation of the price. High y value means a cheap car and low y means expensive car. Think of y as the money left in your pocket after you have paid for the car. Maximal points correspond to the fastest and cheapest cars.

The *2-dimensional Maxima* is thus defined as

- Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space, output the set of maximal points of P, i.e., those points p_i such that p_i is not dominated by any other point of P.

Here is set of maximal points for a given set of points in 2-d.

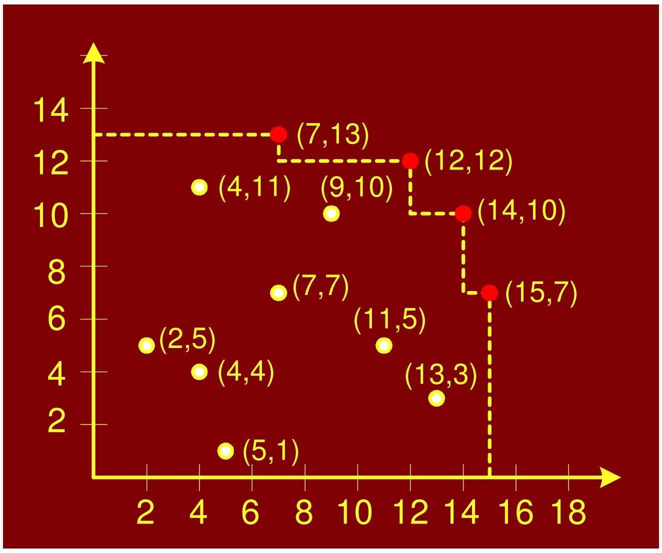


Figure 1.1: Maximal points in 2-d

Our description of the problem is at a fairly mathematical level. We have intentionally not discussed how the points are represented. We have not discussed any input or output formats. We have avoided programming and other software issues.

1.9 Brute-Force Algorithm

To get the ball rolling, let's just consider a simple brute-force algorithm, with no thought to efficiency. Let $P = \{p_1, p_2, \ldots, p_n\}$ be the initial set of points. For each point p_i, test it against all other points p_j. If p_i is not dominated by any other point, then output it.

This English description is clear enough that any (competent) programmer should be able to implement it. However, if you want to be a bit more formal, it could be written in pseudocode as follows:

```
MAXIMA(int $n$, Point $P[1 \ldots n]$)
for $i \leftarrow 1$ to $n$
do maximal $\leftarrow$ true
    for $j \leftarrow 1$ to $n$
    do
        if $(i \neq j)$ and $(P[i].x \le P[j].x)$ and $(P[i].y \le P[j].y)$
          then maximal $\leftarrow$ false;  break
    if (maximal = true)
      then output $P[i]$
```

There are no formal rules to the syntax of this pseudo code. In particular, do not assume that more detail is better. For example, I omitted type specifications for the procedure Maxima and the variable maximal, and I never defined what a Point data type is, since I felt that these are pretty clear from context or just unimportant details. Of course, the appropriate level of detail is a judgement call. Remember, algorithms are to be read by people, and so the level of detail depends on your intended audience. When writing pseudo code, you should omit details that detract from the main ideas of the algorithm, and just go with the essentials.

You might also notice that I did not insert any checking for consistency. For example, I assumed that the points in P are all distinct. If there is a duplicate point then the algorithm may fail to output even a single point. (Can you see why?) Again, these are important considerations for implementation, but we will often omit error checking because we want to see the algorithm in its simplest form.

Here are a series of figures that illustrate point domination.

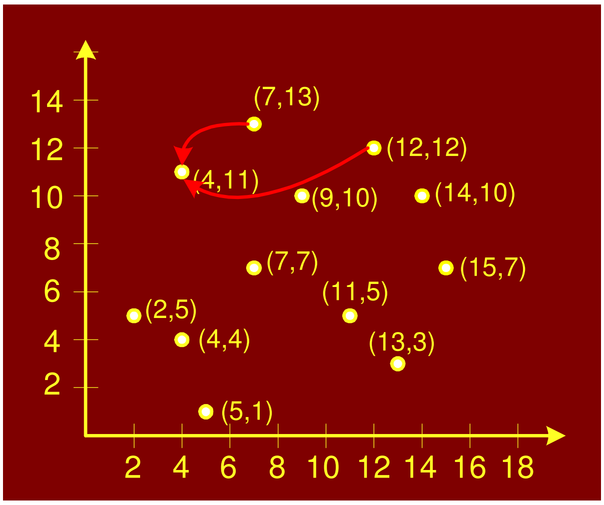


Figure 1.2: Points that dominate $(4, 11)$

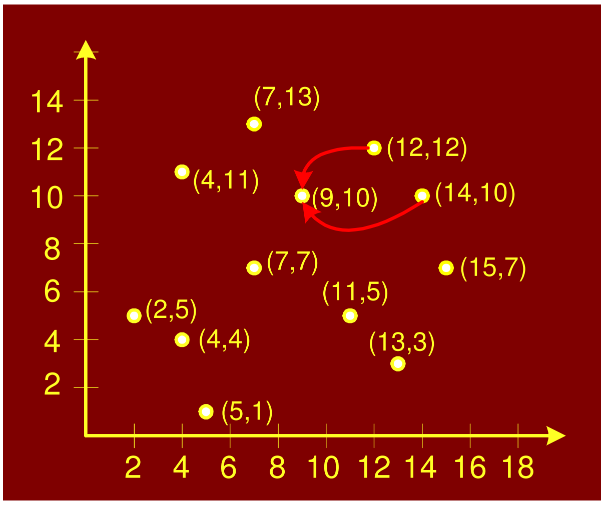


Figure 1.3: Points that dominate $(9, 10)$

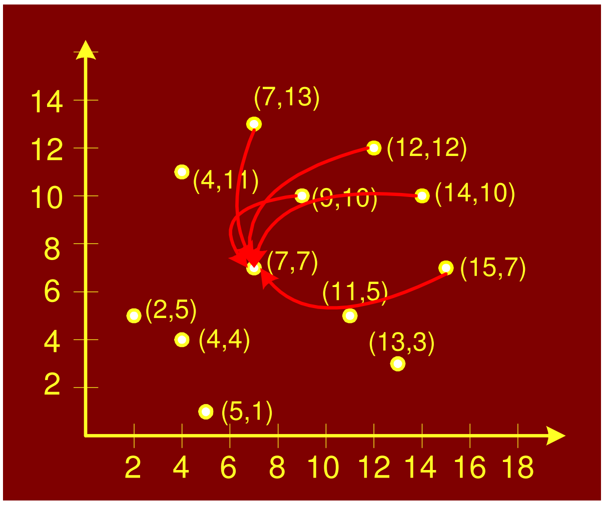


Figure 1.4: Points that dominate $(7, 7)$

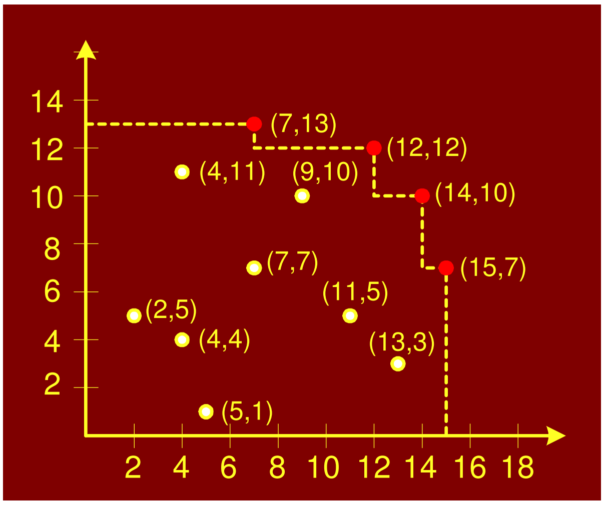


Figure 1.5: The maximal points

1.10 Running Time Analysis

The main purpose of our mathematical analysis will be measuring the execution time. We will also be concerned about the space (memory) required by the algorithm.

The running time of an implementation of the algorithm would depend upon the speed of the computer, programming language, optimization by the compiler etc. Although important, we will ignore these technological issues in our analysis.

To measure the running time of the brute-force 2-d maxima algorithm, we could count the number of steps of the pseudo code that are executed or, count the number of times an element of P is accessed or, the number of comparisons that are performed.

The running time depends upon the input size, e.g. n Different inputs of the same size may result in different running time. For example, breaking out of the inner loop in the brute-force algorithm depends not only on the input size of P but also the structure of the input.

Two criteria for measuring running time are *worst-case time* and *average-case time*.

Worst-case time is the maximum running time over all (legal) inputs of size n. Let I denote an input instance, let $|I|$ denote its length, and let $T(I)$ denote the running time of the algorithm on input I. Then

$$T_{worst}(n) = \max_{|I|=n} T(I)$$

Average-case time is the average running time over all inputs of size n. Let $p(I)$ denote the probability of seeing this input. The average-case time is the weighted sum of running times with weights being the probabilities:

$$T_{avg}(n) = \sum_{|I|=n} p(I)T(I)$$

We will almost always work with worst-case time. Average-case time is more difficult to compute; it is difficult to specify probability distribution on inputs. Worst-case time will specify an upper limit on the running time.

1.10.1 Analysis of the brute-force maxima algorithm.

Assume that the input size is n, and for the running time we will count the number of time that any element of P is accessed. Clearly we go through the outer loop n times, and for each time through this loop, we go through the inner loop n times as well. The condition in the if-statement makes four accesses to P. The output statement makes two accesses for each point that is output. In the worst case every point is maximal (can you see how to generate such an example?) so these two access are made for each time through the outer loop.

```
MAXIMA($\text{int } n, \text{Point } P[1 \ldots n]$)
for $i \leftarrow 1$ to $n$   n times
do $maximal \leftarrow true$
    for $j \leftarrow 1$ to $n$   n times
    do
        if $(i \neq j) \& (P[i].x \leq P[j].x) \& (P[i].y \leq P[j].y)$   4 accesses
          then $maximal \leftarrow false$ break
    if $maximal$
      then output $P[i].x, P[i].y$   2 accesses
```

Thus we might express the worst-case running time as a pair of nested summations, one for the i-loop and the other for the j-loop:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} \left(2 + \sum_{j=1}^{n} 4 \right) \\ &= \sum_{i=1}^{n} (4n + 2) \\ &= (4n + 2)n = 4n^2 + 2n \end{aligned}$$

For small values of n, any algorithm is fast enough. What happens when n gets large? Running time does become an issue. When n is large, n^2 term will be much larger than the n term and will *dominate* the running time.

We will say that the worst-case running time is $\Theta(n^2)$. This is called the asymptotic growth rate of the function. We will discuss this Θ-notation more formally later.

The analysis involved computing a summation. Summation should be familiar but let us review a bit here. Given a finite sequence of values $a_1, a_2, \ldots, a_n$, their sum $a_1 + a_2 + \ldots + a_n$ is expressed in summation notation as

$$\sum_{i=1}^{n} a_i$$

If $n = 0$, then the sum is additive identity, 0.

Some facts about summation: If c is a constant

$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i$$

and

$$\sum_{i=1}^{n}(a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

Some important summations that should be committed to memory.

Arithmetic series

$$\begin{aligned}\sum_{i=1}^{n} i &= 1 + 2 + \ldots + n \\ &= \frac{n(n+1)}{2} = \Theta(n^2)\end{aligned}$$

Quadratic series

$$\begin{aligned}\sum_{i=1}^{n} i^2 &= 1 + 4 + 9 + \ldots + n^2 \\ &= \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3)\end{aligned}$$

Geometric series

$$\begin{aligned}\sum_{i=1}^{n} x^i &= 1 + x + x^2 + \ldots + x^n \\ &= \frac{x^{(n+1)} - 1}{x - 1} = \Theta(n^2)\end{aligned}$$

If $0 < x < 1$ then this is $\Theta(1)$, and if $x > 1$, then this is $\Theta(x^n)$.

Harmonic series For $n \geq 0$

$$\begin{aligned}H_n &= \sum_{i=1}^{n} \frac{1}{i} \\ &= 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n \\ &= \Theta(\ln n)\end{aligned}$$

1.11 Analysis: A Harder Example

Let us consider a harder example.

```
NESTED-LOOPS()
for $i \leftarrow 1$ to $n$
do
    for $j \leftarrow 1$ to $2i$
    do $k = j$ ...
        while $(k \geq 0)$
        do $k = k - 1$ ...
```

How do we analyze the running time of an algorithm that has complex nested loop? The answer is we write out the loops as summations and then solve the summations. To convert loops into summations, we work from inside-out.

Consider the *inner most while* loop.

```
NESTED-LOOPS()
for $i \leftarrow 1$ to $n$
do for $j \leftarrow 1$ to $2i$
    do $k = j$
        while $(k \geq 0)$ ◀
        do $k = k - 1$
```

It is executed for $k = j, j-1, j-2, \ldots, 0$. Time spent inside the while loop is constant. Let $I()$ be the time spent in the while loop. Thus

$$I(j) = \sum_{k=0}^{j} 1 = j + 1$$

Consider the *middle for* loop.

```
NESTED-LOOPS()
for $i \leftarrow 1$ to $n$
do for $j \leftarrow 1$ to $2i$ ◀
    do $k = j$
        while $(k \geq 0)$
        do $k = k - 1$
```

Its running time is determined by i. Let $M()$ be the time spent in the for loop:

$$\begin{aligned}
M(i) &= \sum_{j=1}^{2i} I(j) \\
&= \sum_{j=1}^{2i} (j+1) \\
&= \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1 \\
&= \frac{2i(2i+1)}{2} + 2i \\
&= 2i^2 + 3i
\end{aligned}$$

Finally the *outer-most for* loop.

```
NESTED-LOOPS()
for $i \leftarrow 1$ to $n$
do for $j \leftarrow 1$ to $2i$
    do $k = j$
        while $(k \geq 0)$
        do $k = k - 1$
```

Let $T()$ be running time of the entire algorithm:

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n} M(i) \\
&= \sum_{i=1}^{n} (2i^2 + 3i) \\
&= \sum_{i=1}^{n} 2i^2 + \sum_{i=1}^{n} 3i \\
&= 2\frac{2n^3 + 3n^2 + n}{6} + 3\frac{n(n+1)}{2} \\
&= \frac{4n^3 + 15n^2 + 11n}{6} \\
&= \Theta(n^3)
\end{aligned}$$

1.11.1 2-dimension Maxima Revisited

Recall the 2-d maxima problem: Let a point p in 2-dimensional space be given by its integer coordinates, $p = (p.x, p.y)$. A point p is said to *dominated* by point q if $p.x \leq q.x$ and $p.y \leq q.y$. Given a set of n

points, $P = \{p_1, p_2, \ldots, p_n\}$ in 2-space a point is said to be *maximal* if it is not dominated by any other point in P. The problem is to output all the maximal points of P. We introduced a brute-force algorithm that ran in $\Theta(n^2)$ time. It operated by comparing *all pairs* of points. Is there an approach that is significantly better?

The problem with the brute-force algorithm is that it uses no intelligence in *pruning* out decisions. For example, once we know that a point p_i is dominated by another point p_j, we do not need to use p_i for eliminating other points. This follows from the fact that dominance relation is *transitive*. If p_j dominates p_i and p_i dominates p_h then p_j also dominates p_h; p_i is not needed.

1.11.2 Plane-sweep Algorithm

The question is whether we can make an significant improvement in the running time? Here is an idea for how we might do it. We will sweep a vertical line across the plane from left to right. As we sweep this line, we will build a structure holding the maximal points lying to the left of the sweep line. When the sweep line reaches the rightmost point of P , then we will have constructed the complete set of maxima. This approach of solving geometric problems by sweeping a line across the plane is called *plane sweep*.

Although we would like to think of this as a continuous process, we need some way to perform the plane sweep in discrete steps. To do this, we will begin by sorting the points in increasing order of their x-coordinates. For simplicity, let us assume that no two points have the same y-coordinate. (This limiting assumption is actually easy to overcome, but it is good to work with the simpler version, and save the messy details for the actual implementation.) Then we will advance the sweep-line from point to point in n discrete steps. As we encounter each new point, we will update the current list of maximal points.

We will sweep a vertical line across the 2-d plane from left to right. As we sweep, we will build a structure holding the maximal points lying to the left of the sweep line. When the sweep line reaches the rightmost point of P, we will have the complete set of maximal points. We will store the existing maximal points in a list The points that p_i dominates will appear at the end of the list because points are sorted by x-coordinate. We will scan the list left to right. Every maximal point with y-coordinate less than p_i will be eliminated from computation. We will add maximal points onto the end of a list and delete from the end of the list. We can thus use a stack to store the maximal points. The point at the top of the stack will have the highest x-coordinate.

Here are a series of figures that illustrate the plane sweep. The figure also show the content of the stack.

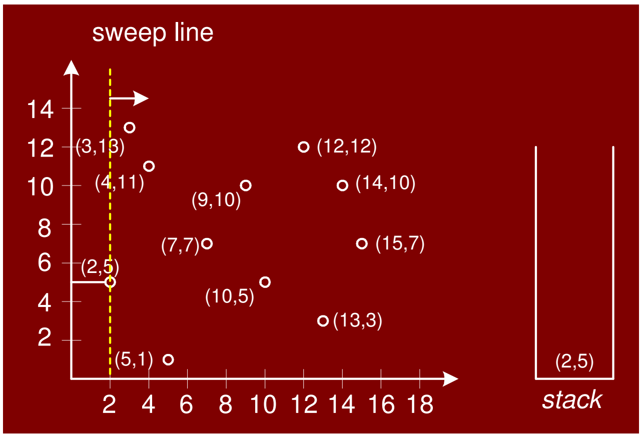


Figure 1.6: Sweep line at $(2,5)$

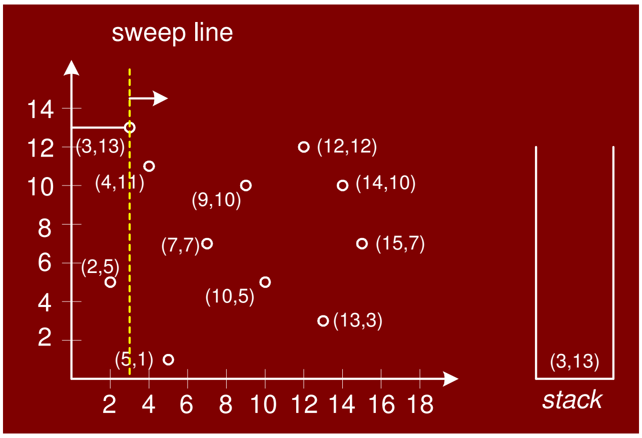


Figure 1.7: Sweep line at $(3,13)$

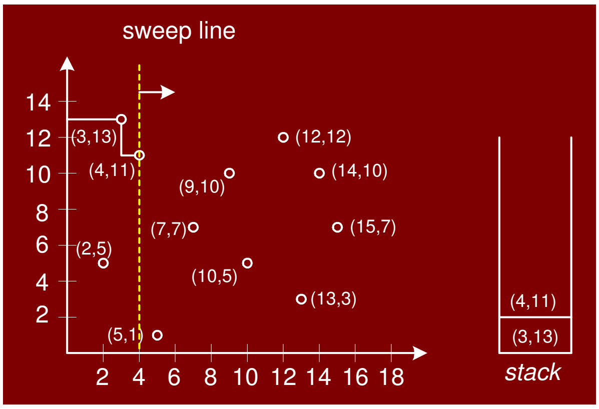


Figure 1.8: Sweep line at $(4,11)$

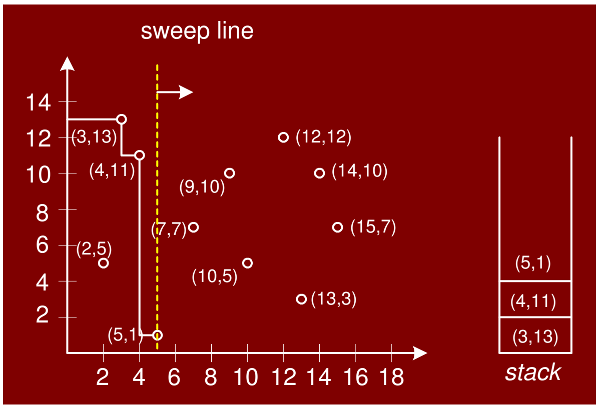


Figure 1.9: Sweep line at $(5,1)$

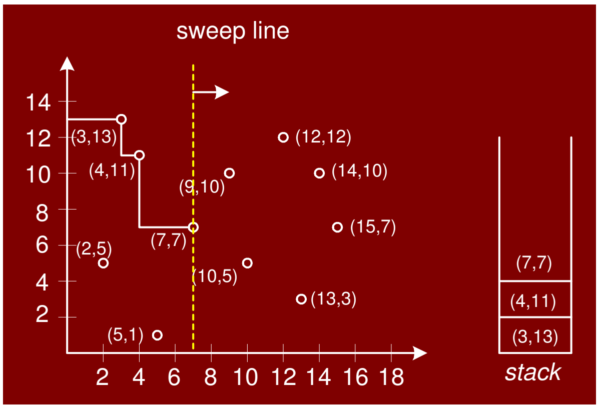


Figure 1.10: Sweep line at $(7,7)$

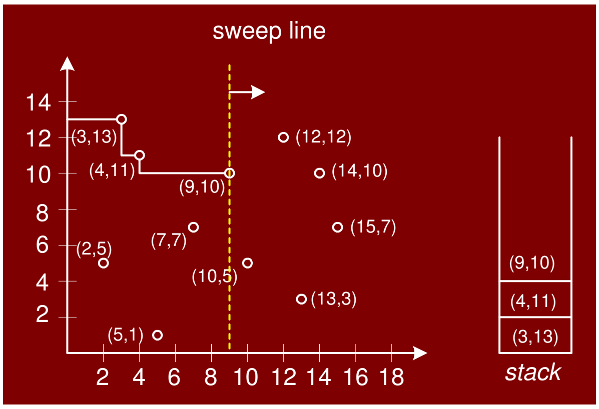


Figure 1.11: Sweep line at $(9,10)$

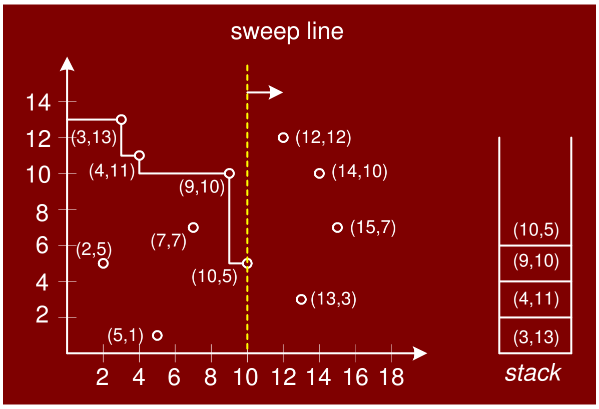


Figure 1.12: Sweep line at $(10, 5)$

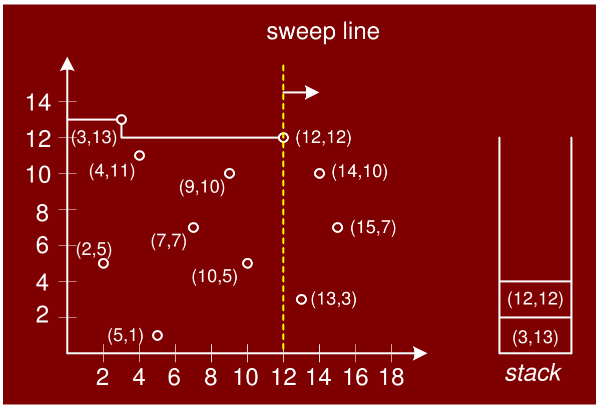


Figure 1.13: Sweep line at $(12, 12)$

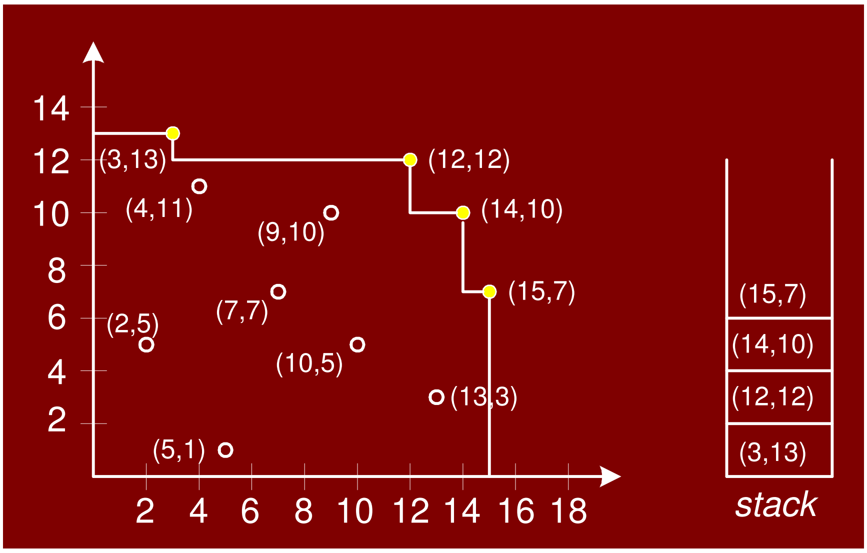


Figure 1.14: The final maximal set

Here is the algorithm.

```
PLANE-SWEEP-MAXIMA$(n, P[1..n])$
sort P in increasing order by x;
stack s;
for $i \leftarrow 1$ to $n$
do
    while $(s.notEmpty() \,\&\, s.top().y \le P[i].y)$
    do s.pop();
    s.push(P[i]);
output the contents of stack s;
```

1.11.3 Analysis of Plane-sweep Algorithm

Sorting takes $\Theta(n \log n)$; we will show this later when we discuss sorting. The for loop executes n times. The inner loop (seemingly) could be iterated $(n-1)$ times. It seems we still have an $n(n-1)$ or $\Theta(n^2)$ algorithm. Got fooled by simple minded loop-counting. The while loop will not execute more n times over the entire course of the algorithm. Why is this? Observe that the total number of elements that can be pushed on the stack is n since we execute exactly one push each time during the outer for-loop.

We pop an element off the stack each time we go through the inner while-loop. It is impossible to pop more elements than are ever pushed on the stack. Therefore, the inner while-loop cannot execute more than n times over the entire course of the algorithm. (*Make sure that you understand this*).

The for-loop iterates n times and the inner while-loop also iterates n time for a total of $\Theta(n)$. Combined with the sorting, the runtime of entire plane-sweep algorithm is $\Theta(n \log n)$.

1.11.4 Comparison of Brute-force and Plane sweep algorithms

How much of an improvement is plane-sweep over brute-force? Consider the ratio of running times:

$$\frac{n^2}{n \log n} = \frac{n}{\log n}$$

n	$\log n$	$\frac{n}{\log n}$
100	7	15
1000	10	100
10000	13	752
100000	17	6021
1000000	20	50171

For $n = 1,000,000$, if plane-sweep takes 1 second, the brute-force will take about 14 hours!. From this we get an idea about the importance of asymptotic analysis. It tells us which algorithm is better for large values of n. As we mentioned before, if n is not very large, then almost any algorithm will be fast. But efficient algorithm design is most important for large inputs, and the general rule of computing is that input sizes continue to grow until people can no longer tolerate the running times. Thus, by designing algorithms efficiently, you make it possible for the user to run large inputs in a reasonable amount of time.

Chapter 2

Asymptotic Notation

You may be asking that we continue to use the notation $\Theta()$ but have never defined it. Let's remedy this now. Given any function $g(n)$, we define $\Theta(g(n))$ to be a set of functions that *asymptotically equivalent* to $g(n)$. Formally:

$$\begin{aligned}\Theta(g(n)) = \{f(n) \mid & \text{ there exist positive} \\ & \text{constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ & 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \\ & \text{for all } n \ge n_0\}\end{aligned}$$

This is written as "$f(n) \in \Theta(g(n))$" That is, $f(n)$ and $g(n)$ are *asymptotically equivalent.* This means that they have essentially the same growth rates for large n. For example, functions like

- $4n^2$,
- $(8n^2 + 2n - 3)$,
- $(n^2/5 + \sqrt{n} - 10 \log n)$
- $n(n-3)$

are all asymptotically equivalent. As n becomes large, the *dominant* (fastest growing) term is some constant times n^2.

Consider the function

$$f(n) = 8n^2 + 2n - 3$$

Our informal rule of keeping the largest term and ignoring the constant suggests that $f(n) \in \Theta(n^2)$. Let's see why this bears out formally. We need to show two things for

$$f(n) = 8n^2 + 2n - 3$$

Lower bound $f(n) = 8n^2 + 2n - 3$ grows asymptotically at least as fast as n^2,

Upper bound $f(n)$ grows no faster asymptotically than n^2,

Lower bound: $f(n)$ grows asymptotically at least as fast as n^2. For this, need to show that there exist positive constants c_1 and n_0, such that $f(n) \geq c_1 n^2$ for all $n \geq n_0$. Consider the reasoning

$$f(n) = 8n^2 + 2n - 3 \geq 8n^2 - 3 = 7n^2 + (n^2 - 3) \geq 7n^2$$

Thus $c_1 = 7$. We implicitly assumed that $2n \geq 0$ and $n^2 - 3 \geq 0$ These are not true for all n but if $n \geq \sqrt{3}$, then both are true. So select $n_0 \geq \sqrt{3}$. We then have $f(n) \geq c_1 n^2$ for all $n \geq n_0$.

Upper bound: $f(n)$ grows asymptotically no faster than n^2. For this, we need to show that there exist positive constants c_2 and n_0, such that $f(n) \leq c_2 n^2$ for all $n \geq n_0$. Consider the reasoning

$$f(n) = 8n^2 + 2n - 3 \leq 8n^2 + 2n \leq 8n^2 + 2n^2 = 10n^2$$

Thus $c_2 = 10$. We implicitly made the assumption that $2n \leq 2n^2$. This is not true for all n but it is true for all $n \geq 1$ So select $n_0 \geq 1$. We thus have $f(n) \leq c_2 n^2$ for all $n \geq n_0$.

From lower bound we have $n_0 \geq \sqrt{3}$ From upper bound we have $n_0 \geq 1$. Combining the two, we let n_0 be the larger of the two: $n_0 \geq \sqrt{3}$. In conclusion, if we let $c_1 = 7$, $c_2 = 10$ and $n_0 \geq \sqrt{3}$, we have

$$7n^2 \leq 8n^2 + 2n - 3 \leq 10n^2 \quad \text{for all } n \geq \sqrt{3}$$

We have thus established

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \text{for all } n \geq n_0$$

Here are plots of the three functions. Notice the bounds.

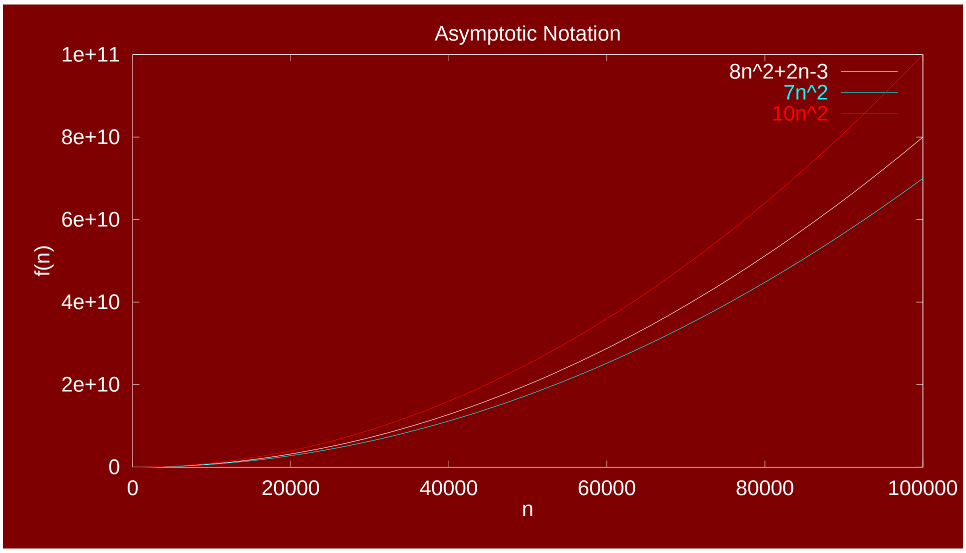


Figure 2.1: Asymptotic Notation Example

We have established that $f(n) \in n^2$. Let's show why $f(n)$ is not in some other asymptotic class. First, let's show that $f(n) \notin \Theta(n)$. Show that $f(n) \notin \Theta(n)$. If this were true, we would have had to satisfy both the upper and lower bounds. The lower bound is satisfied because $f(n) = 8n^2 + 2n - 3$ does grow at least as fast asymptotically as n. But the upper bound is false. Upper bounds requires that there exist positive constants c_2 and n_0 such that $f(n) \leq c_2 n$ for all $n \geq n_0$.

Informally we know that $f(n) = 8n^2 + 2n - 3$ will eventually exceed $c_2 n$ no matter how large we make c_2. To see this, suppose we assume that constants c_2 and n_0 did exist such that $8n^2 + 2n - 3 \leq c_2 n$ for all $n \geq n_0$ Since this is true for all sufficiently large n then it must be true in the limit as n tends to infinity. If we divide both sides by n, we have

$$\lim_{n \to \infty} \left(8n + 2 - \frac{3}{n} \right) \leq c_2.$$

It is easy to see that in the limit, the left side tends to ∞. So, no matter how large c_2 is, the statement is violated. Thus $f(n) \notin \Theta(n)$.

Let's show that $f(n) \notin \Theta(n^3)$. The idea would be to show that the lower bound $f(n) \geq c_1 n^3$ for all $n \geq n_0$ is violated. (c_1 and n_0 are positive constants). Informally we know this to be true because any cubic function will overtake a quadratic.

If we divide both sides by n^3:

$$\lim_{n \to \infty} \left(\frac{8}{n} + \frac{2}{n^2} - \frac{3}{n^3} \right) \geq c_1$$

The left side tends to 0. The only way to satisfy this is to set $c_1 = 0$. But by hypothesis, c_1 is positive. This means that $f(n) \notin \Theta(n^3)$.

The definition of Θ-notation relies on proving both a lower and upper asymptotic bound. Sometimes we only interested in proving one bound or the other. The O-notation is used to state only the asymptotic *upper* bounds.

$$\begin{aligned} O(g(n)) = \{f(n) \mid\ & \text{there exist positive} \\ & \text{constants } c \text{ and } n_0 \text{ such that} \\ & 0 \leq f(n) \leq cg(n) \\ & \text{for all } n \geq n_0\} \end{aligned}$$

The Ω-notation allows us to state only the asymptotic *lower* bounds.

$$\begin{aligned} \Omega(g(n)) = \{f(n) \mid\ & \text{there exist positive} \\ & \text{constants } c \text{ and } n_0 \text{ such that} \\ & 0 \leq cg(n) \leq f(n) \\ & \text{for all } n \geq n_0\} \end{aligned}$$

The three notations:

$$\begin{aligned}
\Theta(g(n)):&\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\\
O(g(n)):&\ 0 \le f(n) \le cg(n)\\
\Omega(g(n)):&\ 0 \le cg(n) \le f(n)\\
&\text{for all } n \ge n_0
\end{aligned}$$

These definitions suggest an alternative way of showing the asymptotic behavior. We can use limits for define the asymptotic behavior. Limit rule for Θ-notation:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,$$

for some constant $c > 0$ (strictly positive but not infinity) then $f(n) \in \Theta(g(n))$. Similarly, the limit rule for O-notation is

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,$$

for some constant $c \ge 0$ (nonnegative but not infinite) then $f(n) \in O(g(n))$ and limit rule for Ω-notation:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \ne 0,$$

(either a strictly positive constant or infinity) then $f(n) \in \Omega(g(n))$.

Here is a list of common asymptotic running times:

- $\Theta(1)$: Constant time; can't beat it!
- $\Theta(\log n)$: Inserting into a balanced binary tree; time to find an item in a sorted array of length n using binary search.
- $\Theta(n)$: About the fastest that an algorithm can run.
- $\Theta(n \log n)$: Best sorting algorithms.
- $\Theta(n^2),\ \Theta(n^3)$: Polynomial time. These running times are acceptable when the exponent of n is small or n is not to large, e.g., $n \le 1000$.
- $\Theta(2^n),\ \Theta(3^n)$: Exponential time. Acceptable only if n is small, e.g., $n \le 50$.
- $\Theta(n!),\ \Theta(n^n)$: Acceptable only for really small n, e.g. $n \le 20$.

Chapter 3

Divide and Conquer Strategy

The ancient Roman politicians understood an important principle of good algorithm design (although they were probably not thinking about algorithms at the time). You divide your enemies (by getting them to distrust each other) and then conquer them piece by piece. This is called *divide-and-conquer*. In algorithm design, the idea is to take a problem on a large input, break the input into smaller pieces, solve the problem on each of the small pieces, and then combine the piecewise solutions into a global solution. But once you have broken the problem into pieces, how do you solve these pieces? The answer is to apply divide-and-conquer to them, thus further breaking them down. The process ends when you are left with such tiny pieces remaining (e.g. one or two items) that it is trivial to solve them. Summarizing, the main elements to a divide-and-conquer solution are

Divide: the problem into a small number of pieces

Conquer: solve each piece by applying divide and conquer to it *recursively*

Combine: the pieces together into a global solution.

3.1 Merge Sort

Divide and conquer strategy is applicable in a huge number of computational problems. The first example of divide and conquer algorithm we will discuss is a simple and efficient sorting procedure called We are given a sequence of n numbers A, which we will assume are stored in an array $A[1..n]$. The objective is to output a permutation of this sequence sorted in increasing order. This is normally done by permuting the elements within the array A. Here is how the merge sort algorithm works:

- (Divide:) split A down the middle into two subsequences, each of size roughly $n/2$
- (Conquer:) sort each subsequence by calling merge sort recursively on each.
- (Combine:) merge the two sorted subsequences into a single sorted list.

Let's design the algorithm top-down. We'll assume that the procedure that merges two sorted list is available to us. We'll implement it later. Because the algorithm is called recursively on sublists, in addition to passing in the array itself, we will pass in two indices, which indicate the first and last indices of the sub-array that we are to sort. The call $MergeSort(A, p, r)$ will sort the sub-array $A[p : r]$ and return the sorted result in the same sub-array.Here is the overview. If $r = p$, then this means that there is only one element to sort, and we may return immediately. Otherwise if $(p \neq r)$ there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely $q = (p + r)/2$ (rounded down to the nearest integer). Then we split the array into sub-arrays $A[p : q]$ and $A[q + 1 : r]$. Call MergeSort recursively to sort each sub-array. Finally, we invoke a procedure (which we have yet to write) which merges these two sub-arrays into a single sorted array.

Here is the MergeSort algorithm.

```
MERGE-SORT( array $A$, int $p$, int $r$)
if $(p < r)$
  then
        $q \leftarrow (p + r)/2$
        MERGE-SORT$(A, p, q)$      // sort $A[p..q]$
        MERGE-SORT$(A, q + 1, r)$ // sort $A[q + 1..r]$
        MERGE$(A, p, q, r)$   // merge the two pieces
```

The following figure illustrates the dividing (splitting) procedure.

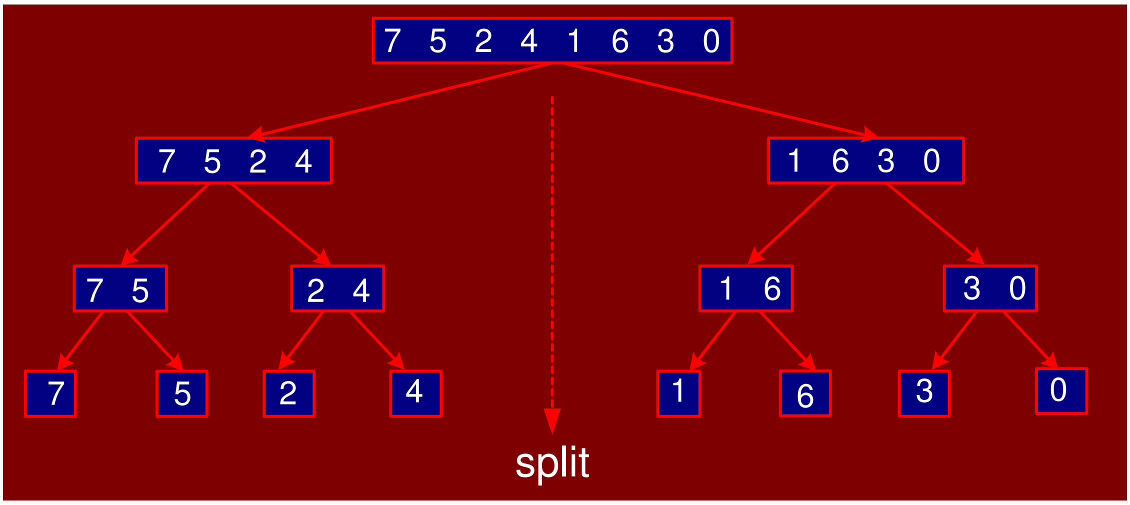


Figure 3.1: Merge sort divide phase

Merging: All that is left is to describe the procedure that merges two sorted lists. $Merge(A, p, q, r)$ assumes that the left sub-array, $A[p : q]$, and the right sub-array, $A[q + 1 : r]$, have already been sorted. We merge these two sub-arrays by copying the elements to a temporary working array called B. For convenience, we will assume that the array B has the same index range A, that is, $B[p : r]$. (One nice thing about pseudo-code, is that we can make these assumptions, and leave them up to the programmer to figure out how to implement it.) We have to indices i and j, that point to the current elements of each sub-array. We move the smaller element into the next position of B (indicated by index k) and then increment the corresponding index (either i or j). When we run out of elements in one array, then we just copy the rest of the other array into B. Finally, we copy the entire contents of B back into A. (The use of the temporary array is a bit unpleasant, but this is impossible to overcome entirely. It is one of the shortcomings of MergeSort, compared to some of the other efficient sorting algorithms.)

Here is the merge algorithm:

```
MERGE( array $A$, int $p$, int $q$ int $r$)
  int $B[p..r]$; int $i \leftarrow k \leftarrow p$; int $j \leftarrow q+1$
  while $(i \leq q)$ and $(j \leq r)$
  do if $(A[i] \leq A[j])$
          then $B[k{+}{+}] \leftarrow A[i{+}{+}]$
          else $B[k{+}{+}] \leftarrow A[j{+}{+}]$
  while $(i \leq q)$
  do $B[k{+}{+}] \leftarrow A[i{+}{+}]$
  while $(j \leq r)$
  do $B[k{+}{+}] \leftarrow A[j{+}{+}]$
  for $i \leftarrow p$ to $r$
  do $A[i] \leftarrow B[i]$
```

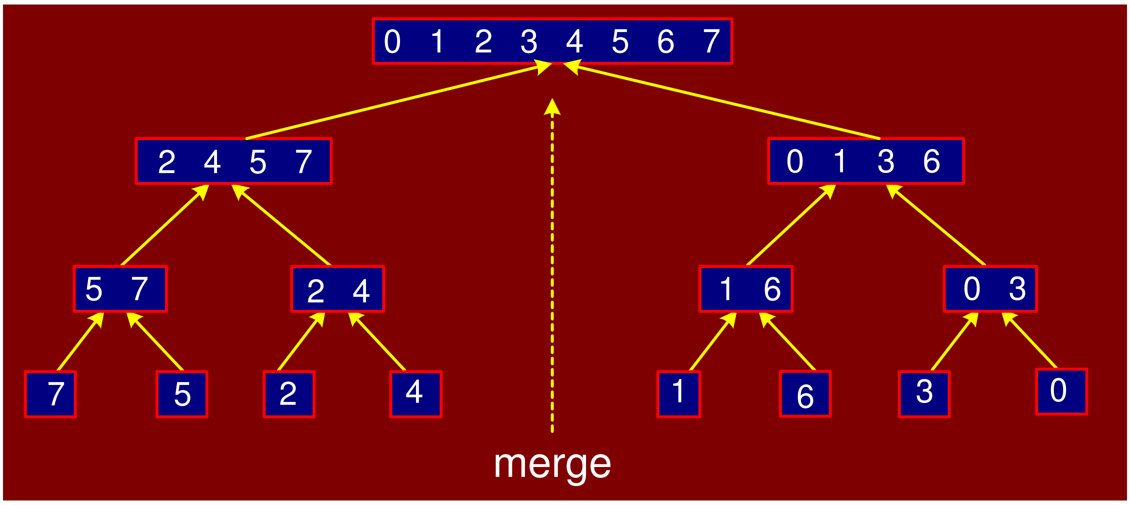


Figure 3.2: Merge sort: combine phase

3.1.1 Analysis of Merge Sort

First let us consider the running time of the procedure $\mathrm{Merge}(A, p, q, r)$. Let $n = r - p + 1$ denote the total length of both the left and right sub-arrays. What is the running time of Merge as a function of n? The algorithm contains four loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the while-loops together can only be executed n times in total, because each execution copies one new element to the array B, and B only has space for n elements.) Thus the running time to Merge n items is $\Theta(n)$. Let us write this without the asymptotic notation, simply as n. (We'll see later why we do this.)

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this through the use of a recurrence, that is, a function that is defined recursively in terms of itself. To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n. Finally, a recurrence has some basis values (e.g. for $n = 1$), which are defined explicitly.

Let $T(n)$ denote the worst case running time of MergeSort on an array of length n. If we call MergeSort with an array containing a single item ($n = 1$) then the running time is constant. We can just write $T(n) = 1$, ignoring all constants. For $n > 1$, MergeSort splits into two halves, sorts the two and then merges them together. The left half is of size $\lceil n/2 \rceil$ and the right half is $\lfloor n/2 \rfloor$. How long does it take to sort elements in sub array of size $\lceil n/2 \rceil$? We do not know this but because $\lceil n/2 \rceil < n$ for $n > 1$, we can express this as $T(\lceil n/2 \rceil)$. Similarly the time taken to sort right sub array is expressed as $T(\lfloor n/2 \rfloor)$. In conclusion we have

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise} \end{cases}$$

This is called *recurrence relation*, i.e., a recursively defined function. Divide-and-conquer is an important design technique, and it naturally gives rise to recursive algorithms. It is thus important to develop mathematical techniques for solving recurrences, either exactly or asymptotically.

Let's expand the terms.

$$\begin{aligned}
T(1) &= 1 \\
T(2) &= T(1) + T(1) + 2 = 1 + 1 + 2 = 4 \\
T(3) &= T(2) + T(1) + 3 = 4 + 1 + 3 = 8 \\
T(4) &= T(2) + T(2) + 4 = 4 + 4 + 4 = 12 \\
T(5) &= T(3) + T(2) + 5 = 8 + 4 + 5 = 17 \\
&\dots \\
T(8) &= T(4) + T(4) + 8 = 12 + 12 + 8 = 32 \\
&\dots \\
T(16) &= T(8) + T(8) + 16 = 32 + 32 + 16 = 80 \\
&\dots \\
T(32) &= T(16) + T(16) + 32 = 80 + 80 + 32 = 192
\end{aligned}$$

What is the pattern here? Let's consider the ratios $T(n)/n$ for powers of 2:

$$\begin{aligned}
T(1)/1 &= 1 & T(8)/8 &= 4 \\
T(2)/2 &= 2 & T(16)/16 &= 5 \\
T(4)/4 &= 3 & T(32)/32 &= 6
\end{aligned}$$

This suggests $T(n)/n = \log n + 1$ Or, $T(n) = n \log n + n$ which is $\Theta(n \log n)$ (using the limit rule).

3.1.2 The Iteration Method for Solving Recurrence Relations

Floor and ceilings are a pain to deal with. If n is assumed to be a power of 2 ($2^k = n$), this will simplify the recurrence to

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

The iteration method turns the recurrence into a summation. Let's see how it works. Let's expand the recurrence:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + n + n \\ &= 4(2T(n/8) + n/4) + n + n \\ &= 8T(n/8) + n + n + n \\ &= 8(2T(n/16) + n/8) + n + n + n \\ &= 16T(n/16) + n + n + n + n \end{aligned}$$

If n is a power of 2 then let $n = 2^k$ or $k = \log n$.

$$\begin{aligned} T(n) &= 2^k T(n/(2^k)) + \underbrace{(n + n + n + \cdots + n)}_{k\ times} \\ &= 2^k T(n/(2^k)) + kn \\ &= 2^{(\log n)} T(n/(2^{(\log n)})) + (\log n)n \\ &= 2^{(\log n)} T(n/n) + (\log n)n \\ &= nT(1) + n \log n = n + n \log n \end{aligned}$$

3.1.3 Visualizing Recurrences Using the Recursion Tree

Iteration is a very powerful technique for solving recurrences. But, it is easy to get lost in all the symbolic manipulations and lose sight of what is going on. Here is a nice way to visualize what is going on in iteration. We can describe any recurrence in terms of a tree, where each expansion of the recurrence takes us one level deeper in the tree.

Recall that the recurrence for MergeSort (which we simplified by assuming that n is a power of 2, and hence could drop the floors and ceilings)

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

Suppose that we draw the recursion tree for MergeSort, but each time we merge two lists, we label that node of the tree with the time it takes to perform the associated (nonrecursive) merge. Recall that to merge two lists of size $m/2$ to a list of size m takes $\Theta(m)$ time, which we will just write as m. Below is an illustration of the resulting recursion tree.

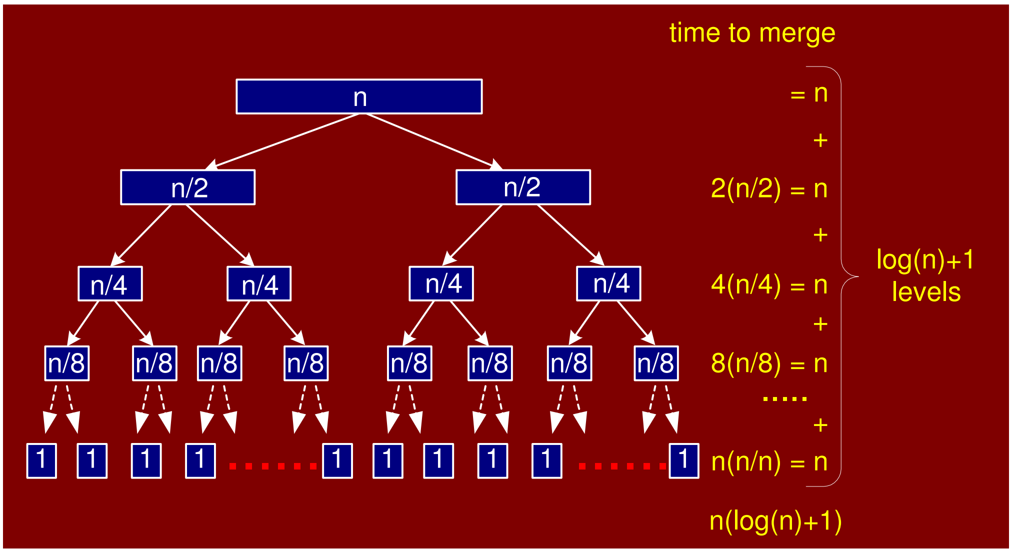


Figure 3.3: Merge sort Recurrence Tree

3.1.4 A Messier Example

The merge sort recurrence was too easy. Let's try a messier recurrence.

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 3T(n/4) + n & \text{otherwise} \end{cases}$$

Assume n to be a power of 4, i.e., $n = 4^k$ and $k = \log_4 n$

$$\begin{aligned} T(n) &= 3T(n/4) + n \\ &= 3(3T(n/16) + (n/4) + n \\ &= 9T(n/16) + 3(n/4) + n \\ &= 27T(n/64) + 9(n/16) + 3(n/4) + n \\ &= \dots \\ &= 3^k T(\frac{n}{4^k}) + 3^{k-1}(n/4^{k-1}) \\ &\qquad + \dots + 9(n/16) + 3(n/4) + n \\ &= 3^k T(\frac{n}{4^k}) + \sum_{i=0}^{k-1} \frac{3^i}{4^i} n \end{aligned}$$

With $n = 4^k$ and $T(1) = 1$

$$\begin{aligned}
T(n) &= 3^k T(\frac{n}{4^k}) + \sum_{i=0}^{k-1} \frac{3^i}{4^i} n \\
&= 3^{\log_4 n} T(1) + \sum_{i=0}^{(\log_4 n)-1} \frac{3^i}{4^i} n \\
&= n^{\log_4 3} + \sum_{i=0}^{(\log_4 n)-1} \frac{3^i}{4^i} n
\end{aligned}$$

We used the formula $a^{\log_b n} = n^{\log_b a}$. n remains constant throughout the sum and $3^i/4^i = (3/4)^i$; we thus have

$$T(n) = n^{\log_4 3} + n \sum_{i=0}^{(\log_4 n)-1} \left(\frac{3}{4}\right)^i$$

The sum is a geometric series; recall that for $x \neq 1$

$$\sum_{i=0}^{m} x_i = \frac{x^{m+1} - 1}{x - 1}$$

In this case $x = 3/4$ and $m = \log_4 n - 1$. We get

$$T(n) = n^{\log_4 3} + n \frac{(3/4)^{\log_4 n+1} - 1}{(3/4) - 1}$$

Applying the log identity once more

$$\begin{aligned}
(3/4)^{\log_4 n} &= n^{\log_4 (3/4)} = n^{\log_4 3 - \log_4 4} \\
&= n^{\log_4 3 - 1} = \frac{n^{\log_4 3}}{n}
\end{aligned}$$

If we plug this back, we get

$$\begin{aligned}
T(n) &= n^{\log_4 3} + n \frac{\frac{n^{\log_4 3}}{n} - 1}{(3/4) - 1} \\
&= n^{\log_4 3} + \frac{n^{\log_4 3} - n}{-1/4} \\
&= n^{\log_4 3} + 4(n - n^{\log_4 3}) \\
&= 4n - 3n^{\log_4 3}
\end{aligned}$$

With $\log_4 3 \approx 0.79$, we finally have the result!

$$T(n) = 4n - 3n^{\log_4 3} \approx 4n - 3n^{0.79} \in \Theta(n)$$

3.2 Selection Problem

Suppose we are given a set of n numbers. Define the *rank* of an element to be one plus the number of elements that are smaller. Thus, the rank of an element is its final position if the set is sorted. The minimum is of rank 1 and the maximum is of rank n.

Consider the set: $\{5, 7, 2, 10, 8, 15, 21, 37, 41\}$. The rank of each number is its position in the sorted order.

position	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Number	2	5	7	8	10	15	21	37	41

For example, rank of 8 is 4, one plus the number of elements smaller than 8 which is 3.

The selection problem is stated as follows:

> Given a set A of n distinct numbers and an integer k, $1 \le k \le n$, output the element of A of rank k.

Of particular interest in statistics is the *median*. If n is odd then the median is defined to be element of rank $(n+1)/2$. When n is even, there are two choices: $n/2$ and $(n+1)/2$. In statistics, it is common to return the average of the two elements.

Medians are useful as a measures of the *central tendency* of a set especially when the distribution of values is highly skewed. For example, the median income in a community is a more meaningful measure than average. Suppose 7 households have monthly incomes 5000, 7000, 2000, 10000, 8000, 15000 and 16000. In sorted order, the incomes are 2000, 5000, 7000, 8000, 10000, 15000, 16000. The median income is 8000; median is element with rank 4: $(7+1)/2 = 4$. The average income is 9000. Suppose the income 16000 goes up to 450,000. The median is still 8000 but the average goes up to 71,000. Clearly, the average is not a good representative of the majority income levels.

The selection problem can be easily solved by simply sorting the numbers of A and returning $A[k]$. Sorting, however, requires $\Theta(n \log n)$ time. The question is: can we do better than that? In particular, is it possible to solve the selections problem in $\Theta(n)$ time? The answer is yes. However, the solution is far from obvious.

3.2.1 Sieve Technique

The reason for introducing this algorithm is that it illustrates a very important special case of divide-and-conquer, which I call the *sieve technique*. We think of divide-and-conquer as breaking the problem into a small number of smaller subproblems, which are then solved recursively. The sieve technique is a special case, where the number of subproblems is just 1.

The sieve technique works in phases as follows. It applies to problems where we are interested in finding a single item from a larger set of n items. We do not know which item is of interest, however after doing some amount of analysis of the data, taking say $\Theta(nk)$ time, for some constant k, we find that we do not know what the desired the item is, but we can identify a large enough number of elements that cannot be the desired value, and can be eliminated from further consideration. In particular “large enough” means that the number of items is at least some fixed constant fraction of n (e.g. $n/2$, $n/3$). Then we solve the problem recursively on whatever items remain. Each of the resulting recursive solutions then do the same thing, eliminating a constant fraction of the remaining set.

3.2.2 Applying the Sieve to Selection

To see more concretely how the sieve technique works, let us apply it to the selection problem. We will begin with the given array $A[1..n]$. We will pick an item from the array, called the *pivot element* which we will denote by x. We will talk about how an item is chosen as the pivot later; for now just think of it as a random element of A.

We then *partition* A into three parts.

1. $A[q]$ contains the pivot element x,
2. $A[1..q-1]$ will contain all the elements that are less than x and
3. $A[q+1..n]$ will contains all elements that are greater than x.

Within each sub array, the items may appear in any order. The following figure shows a partitioned array:

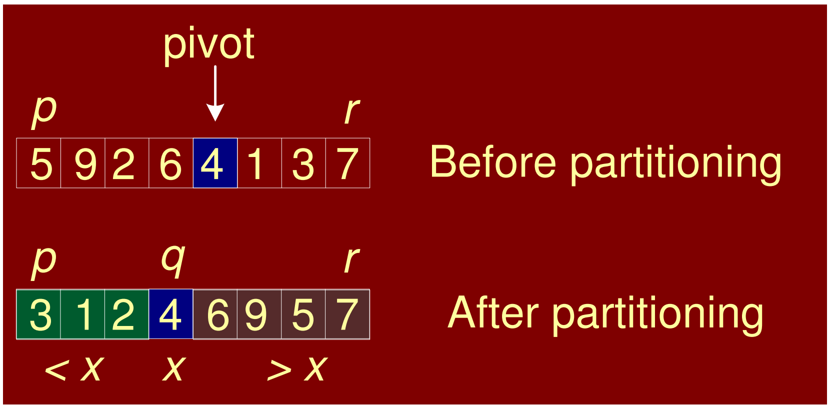


Figure 3.4: $A[p..r]$ partitioned about the pivot x

3.2.3 Selection Algorithm

It is easy to see that the rank of the The rank of the pivot x is $q-p+1$ in $A[p..r]$. Let $rank\_x = q-p+1$. If $k = rank\_x$ then the pivot is k^{th} smallest. If $k < rank\_x$ then search $A[p..q-1]$ recursively. If $k > rank\_x$ then search $A[q+1..r]$ recursively. Find element of rank $(k-q)$ because we eliminated q smaller elements in A.

```
SELECT( array A, int p, int r, int k)
```

```
if $(p = r)$
  then return $A[p]$
  else  $x \leftarrow$ CHOOSE_PIVOT$(A, p, r)$
        $q \leftarrow$ PARTITION$(A, p, r, x)$
        $rank\_x \leftarrow q - p + 1$
        if $k = rank\_x$
          then return $x$
        if $k < rank\_x$
          then return SELECT$(A, p, q - 1, k)$
          else  return SELECT$(A, q + 1, r, k - q)$
```

Example: select the 6^{th} smallest element of the set $\{5, 9, 2, 6, 4, 1, 3, 7\}$

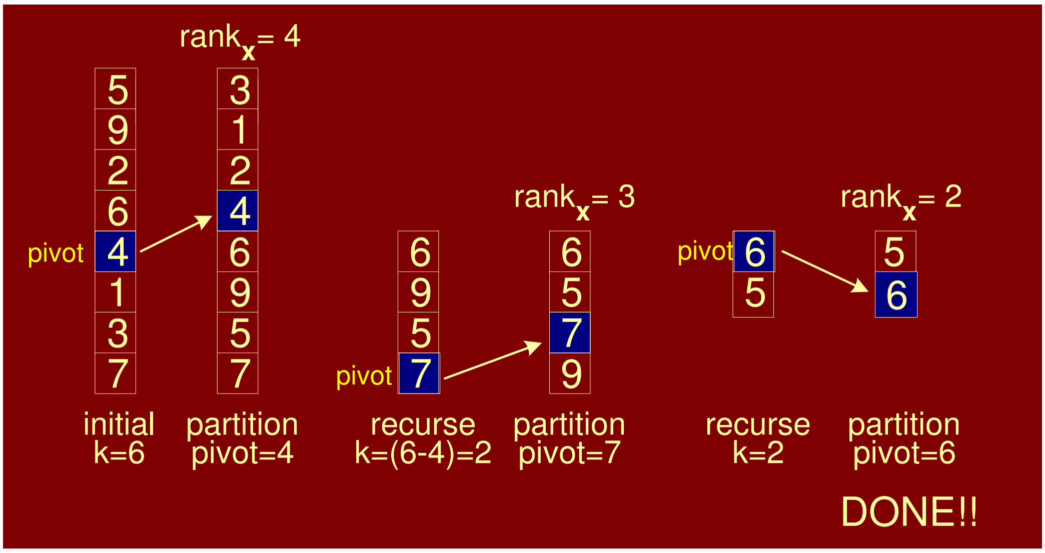


Figure 3.5: Sieve example: select 6^{th} smallest element

3.2.4 Analysis of Selection

We will discuss how to choose a pivot and the partitioning later. For the moment, we will assume that they both take $\Theta(n)$ time. How many elements do we eliminate in each time? If x is the largest or the smallest then we may only succeed in eliminating one element.

$$\begin{array}{rl} 5, 9, 2, 6, 4, \boxed{1}, 3, 7 & \quad \text{pivot is } 1 \\ \boxed{1}, 5, 9, 2, 6, 4, 3, 7 & \quad \text{after partition} \end{array}$$

Ideally, x should have a rank that is neither too large or too small.

Suppose we are able to choose a pivot that causes exactly half of the array to be eliminated in each phase.

This means that we recurse on the remaining $n/2$ elements. This leads to the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n/2) + n & \text{otherwise} \end{cases}$$

If we expand this recurrence, we get

$$\begin{aligned} T(n) &= n + \frac{n}{2} + \frac{n}{4} + \dots \\ &\leq \sum_{i=0}^{\infty} \frac{n}{2^i} \\ &= n \sum_{i=0}^{\infty} \frac{1}{2^i} \end{aligned}$$

Recall the formula for infinite geometric series; for any $|c| < 1$,

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$$

Using this we have

$$T(n) \leq 2n \in \Theta(n)$$

Let's think about how we ended up with a $\Theta(n)$ algorithm for selection. Normally, a $\Theta(n)$ time algorithm would make a single or perhaps a constant number of passes of the data set. In this algorithm, we make a number of passes. In fact it could be as many as $\log n$.

However, because we eliminate a constant fraction of the array with each phase, we get the convergent geometric series in the analysis. This shows that the total running time is indeed *linear* in n. This lesson is well worth remembering. It is often possible to achieve linear running times in ways that you would not expect.

Chapter 4

Sorting

For the next series of lectures, we will focus on sorting. There a number of reasons for sorting. Here are a few important ones. Procedures for sorting are parts of many large software systems. Design of efficient sorting algorithms is necessary to achieve overall efficiency of these systems.

Sorting is well studied problem from the analysis point of view. Sorting is one of the few problems where provable lower bounds exist on how fast we can sort. In sorting, we are given an array $A[1..n]$ of n numbers We are to reorder these elements into increasing (or decreasing) order.

More generally, A is an array of objects and we sort them based on one of the attributes - the *key value*. The key value need not be a number. It can be any object from a *totally ordered domain*. Totally ordered domain means that for any two elements of the domain, x and y, either $x < y$, $x = y$ or $x > y$.

4.1 Slow Sorting Algorithms

There are a number of well-known slow $O(n^2)$ sorting algorithms. These include the following:

Bubble sort: Scan the array. Whenever two consecutive items are found that are out of order, swap them. Repeat until all consecutive items are in order.

Insertion sort: Assume that $A[1..i-1]$ have already been sorted. Insert $A[i]$ into its proper position in this sub array. Create this position by shifting all larger elements to the right.

Selection sort: Assume that $A[1..i-1]$ contain the $i-1$ smallest elements in sorted order. Find the smallest element in $A[i..n]$ Swap it with $A[i]$.

These algorithms are easy to implement. But they run in $\Theta(n^2)$ time in the worst case.

4.2 Sorting in $O(n \log n)$ time

We have already seen that Mergesort sorts an array of numbers in $\Theta(n \log n)$ time. We will study two others: *Heapsort* and *Quicksort*.

4.2.1 Heaps

A *heap* is a left-complete binary tree that conforms to the *heap order*. The heap order property: in a (min) heap, for every node X, the key in the parent is smaller than or equal to the key in X. In other words, the parent node has key smaller than or equal to both of its children nodes. Similarly, in a max heap, the parent has a key larger than or equal both of its children Thus the smallest key is in the root in a min heap; in the max heap, the largest is in the root.

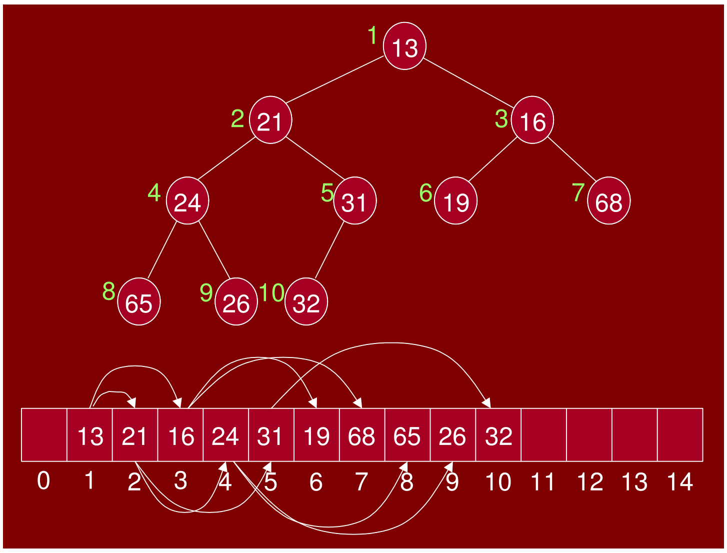


Figure 4.1: A min-heap

The number of nodes in a complete binary tree of height h is

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

h in terms of n is

$$h = (\log(n+1)) - 1 \approx \log n \in \Theta(\log n)$$

One of the clever aspects of heaps is that they can be stored in arrays without using any pointers. This is due to the left-complete nature of the binary tree. We store the tree nodes in level-order traversal. Access to nodes involves simple arithmetic operations:

$\mathit{left}(i)$: returns $2i$, index of left child of node i.
$\mathit{right}(i)$: returns $2i+1$, the right child.
$\mathit{parent}(i)$: returns $\lfloor i/2 \rfloor$, the parent of i.

The root is at position 1 of the array.

4.2.2 Heapsort Algorithm

We build a max heap out of the given array of numbers $A[1..n]$. We repeatedly extract the the maximum item from the heap. Once the max item is removed, we are left with a hole at the root. To fix this, we will replace it with the last leaf in tree. But now the heap order will very likely be destroyed. We will apply a heapify procedure to the root to restore the heap. Figure 4.2 shows an array being sorted.

```
HEAPSORT( array $A$, int $n$)
BUILD-HEAP$(A, n)$
$m \leftarrow n$
while $(m \geq 2)$
do SWAP$(A[1], A[m])$
   $m \leftarrow m - 1$
   HEAPIFY$(A, 1, m)$
```

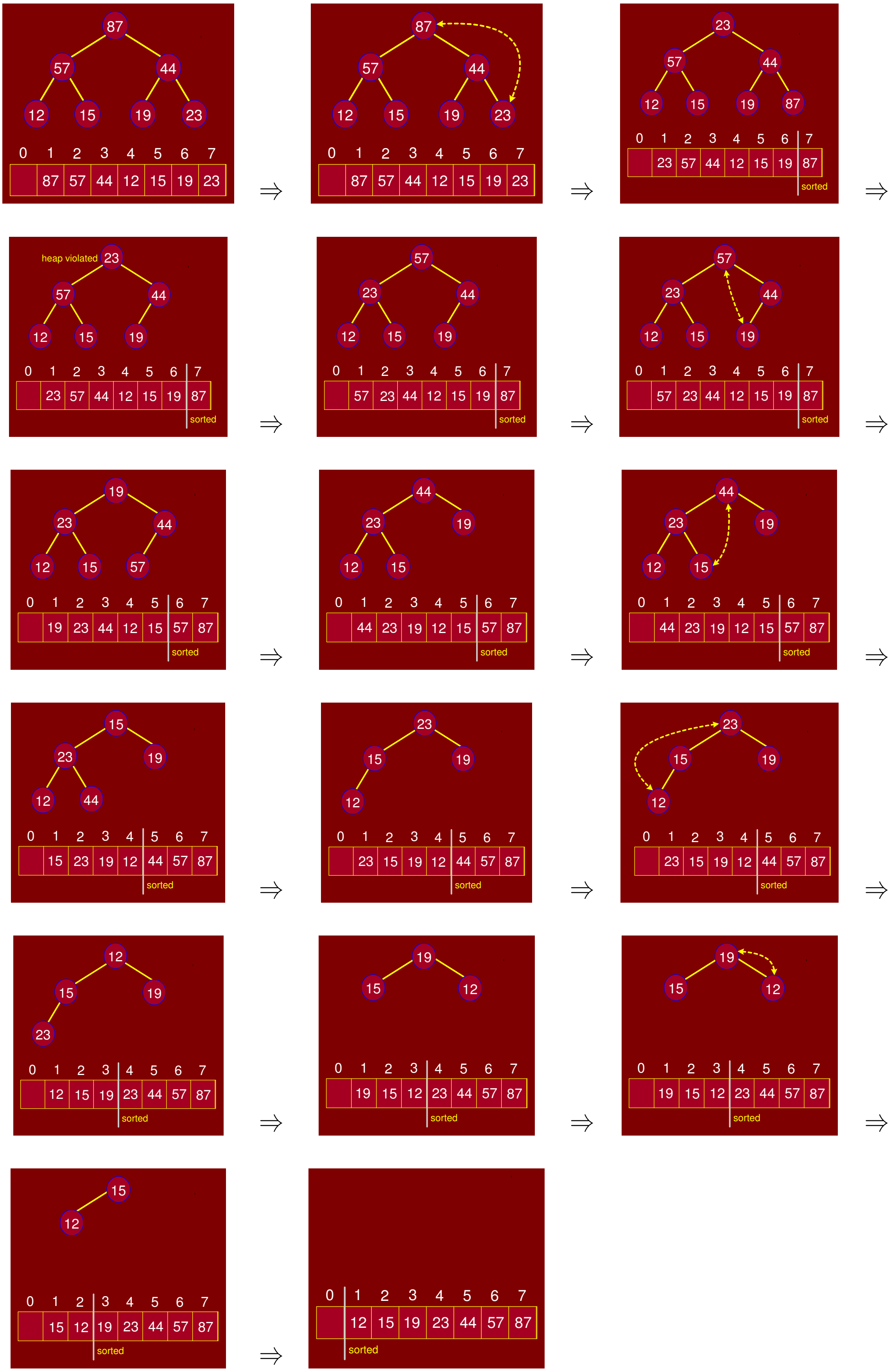


Figure 4.2: Example of heap sort

4.2.3 Heapify Procedure

There is one principal operation for maintaining the heap property. It is called Heapify. (In other books it is sometimes called sifting down.) The idea is that we are given an element of the heap which we suspect may not be in valid heap order, but we assume that all of other the elements in the subtree rooted at this element are in heap order. In particular this root element may be too small. To fix this we "sift" it down the tree by swapping it with one of its children. Which child? We should take the larger of the two children to satisfy the heap ordering property. This continues recursively until the element is either larger than both its children or until its falls all the way to the leaf level. Here is the algorithm. It is given the heap in the array A, and the index i of the suspected element, and m the current active size of the heap. The element $A[max]$ is set to the maximum of $A[i]$ and it two children. If $max \neq i$ then we swap $A[i]$ and $A[max]$ and then recurse on $A[max]$.

```
HEAPIFY( array $A$, int $i$, int $m$)
  $l \leftarrow$ LEFT$(i)$
  $r \leftarrow$ RIGHT$(i)$
  $max \leftarrow i$
  **if** $(l \leq m) and (A[l] > A[max])$
     **then** $max \leftarrow l$
  **if** $(r \leq m) and (A[r] > A[max])$
     **then** $max \leftarrow r$
  **if** $(max \neq i)$
     **then** SWAP$(A[i], A[max])$
             HEAPIFY$(A, max, m)$
```

4.2.4 Analysis of Heapify

We call heapify on the root of the tree. The maximum levels an element could move up is $\Theta(\log n)$ levels. At each level, we do simple comparison which $O(1)$. The total time for heapify is thus $O(\log n)$. Notice that it is not $\Theta(\log n)$ since, for example, if we call heapify on a leaf, it will terminate in $\Theta(1)$ time.

4.2.5 BuildHeap

We can use Heapify to build a heap as follows. First we start with a heap in which the elements are not in heap order. They are just in the same order that they were given to us in the array A. We build the heap by starting at the leaf level and then invoke Heapify on each node. (Note: We cannot start at the top of the tree. Why not? Because the precondition which Heapify assumes is that the entire tree rooted at node i is already in heap order, except for i.) Actually, we can be a bit more efficient. Since we know that each leaf is already in heap order, we may as well skip the leaves and start with the first non-leaf node. This will be in position $n/2$. (Can you see why?)

Here is the code. Since we will work with the entire array, the parameter m for Heapify, which indicates the current heap size will be equal to n, the size of array A, in all the calls.

```
BUILDHEAP( array $A$, int $n$)
for $i \leftarrow n/2$ downto 1
do
    HEAPIFY($A, i, n$)
```

4.2.6 Analysis of BuildHeap

For convenience, we will assume $n = 2^{h+1} - 1$ where h is the height of tree. The heap is a left-complete binary tree. Thus at each level j, $j < h$, there are 2^j nodes in the tree. At level h, there will be 2^h or less nodes. How much work does buildHeap carry out? Consider the heap in Figure 4.3:

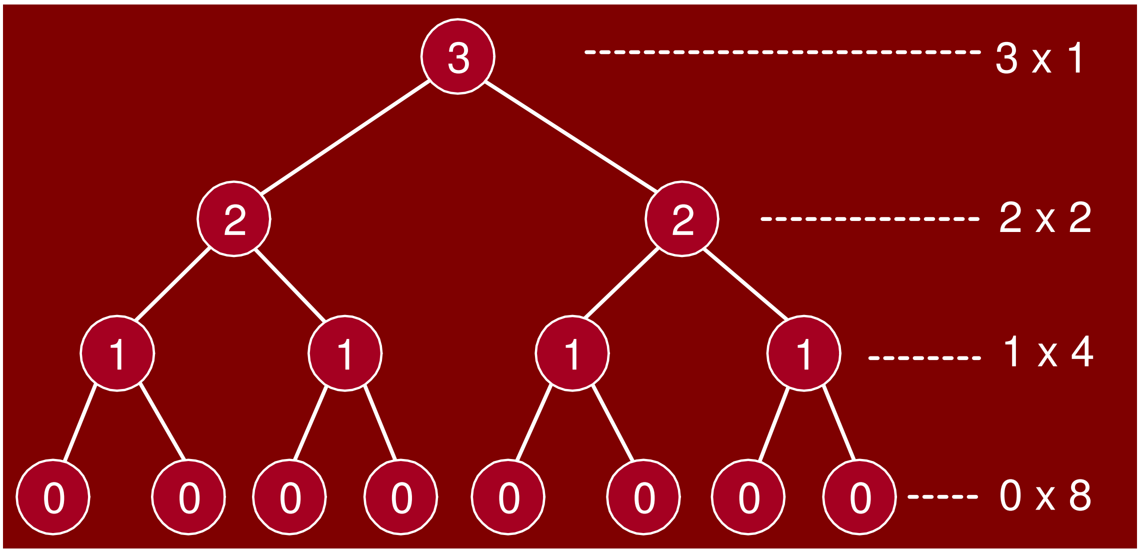


Figure 4.3: Total work performed in buildheap

At the bottom most level, there are 2^h nodes but we do not heapify these. At the next level up, there are 2^{h-1} nodes and each might shift down 1. In general, at level j, there are 2^{h-j} nodes and each may shift down j levels.

So, if count from bottom to top, level-by-level, the total time is

$$T(n) = \sum_{j=0}^{h} j 2^{h-j} = \sum_{j=0}^{h} j \frac{2^h}{2^j}$$

We can factor out the 2^h term:

$$T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

How do we solve this sum? Recall the geometric series, for any constant $x < 1$

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}$$

Take the derivative with respect to x and multiply by x

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2} \qquad \sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2}$$

We plug $x = 1/2$ and we have the desired formula:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1-(1/2))^2} = \frac{1/2}{1/4} = 2$$

In our case, we have a bounded sum, but since we the infinite series is bounded, we can use it as an easy approximation:

$$\begin{aligned} T(n) &= 2^h \sum_{j=0}^{h} \frac{j}{2^j} \\ &\leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \\ &\leq 2^h \cdot 2 = 2^{h+1} \end{aligned}$$

Recall that $n = 2^{h+1} - 1$. Therefore

$$T(n) \leq n + 1 \in O(n)$$

The algorithm takes at least $\Omega(n)$ time since it must access every element at once. So the total time for BuildHeap is $\Theta(n)$.

BuildHeap is a relatively complex algorithm. Yet, the analysis yield that it takes $\Theta(n)$ time. An intuitive way to describe why it is so is to observe an important fact about binary trees The fact is that the vast majority of the nodes are at the lowest level of the tree. For example, in a complete binary tree of height h, there is a total of $n \approx 2^{h+1}$ nodes.

The number of nodes at the bottom three levels alone is

$$2^h + 2^{h-1} + 2^{h-2} = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} = \frac{7n}{8} = 0.875n$$

Almost 90% of the nodes of a complete binary tree reside in the 3 lowest levels. Thus, algorithms that operate on trees should be efficient (as BuildHeap is) on the bottom-most levels since that is where most of the weight of the tree resides.

4.2.7 Analysis of Heapsort

Heapsort calls BuildHeap once. This takes $\Theta(n)$. Heapsort then extracts roughly n maximum elements from the heap. Each extract requires a constant amount of work (swap) and $O(\log n)$ heapify. Heapsort is thus $O(n \log n)$.

Is HeapSort $\Theta(n \log n)$? The answer is yes. In fact, later we will show that comparison based sorting algorithms can not run faster than $\Omega(n \log n)$. Heapsort is such an algorithm and so is Mergesort that we saw ealier.

4.3 Quicksort

Our next sorting algorithm is Quicksort. It is one of the fastest sorting algorithms known and is the method of choice in most sorting libraries. Quicksort is based on the divide and conquer strategy. Here is the algorithm:

```
QUICKSORT( array $A$, int $p$, int $r$)
if $(r > p)$
  then
       $i \leftarrow$ a random index from $[p..r]$
        swap $A[i]$ with $A[p]$
       $q \leftarrow$ PARTITION$(A, p, r)$
       QUICKSORT$(A, p, q - 1)$
       QUICKSORT$(A, q + 1, r)$
```

4.3.1 Partition Algorithm

Recall that the partition algorithm partitions the array $A[p..r]$ into three sub arrays about a pivot element x. $A[p..q-1]$ whose elements are less than or equal to x, $A[q] = x$, $A[q+1..r]$ whose elements are greater than x

We will choose the first element of the array as the pivot, i.e. $x = A[p]$. If a different rule is used for selecting the pivot, we can swap the chosen element with the first element. We will choose the pivot randomly.

The algorithm works by maintaining the following *invariant condition*. $A[p] = x$ is the pivot value. $A[p..q-1]$ contains elements that are less than x, $A[q+1..s-1]$ contains elements that are greater than or equal to x $A[s..r]$ contains elements whose values are currently unknown.

```
PARTITION( array A, int p, int r)
x ← A[p]
q ← p
for s ← p + 1 to r
do if (A[s] < x)
      then q ← q + 1
           swap A[q] with A[s]
 swap A[p] with A[q]
return q
```

Figure 4.4 shows the execution trace partition algorithm.

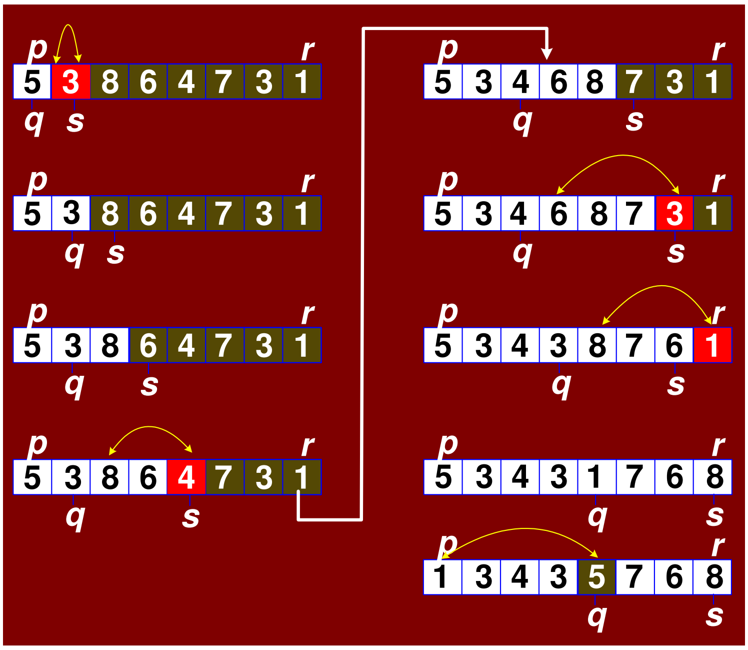
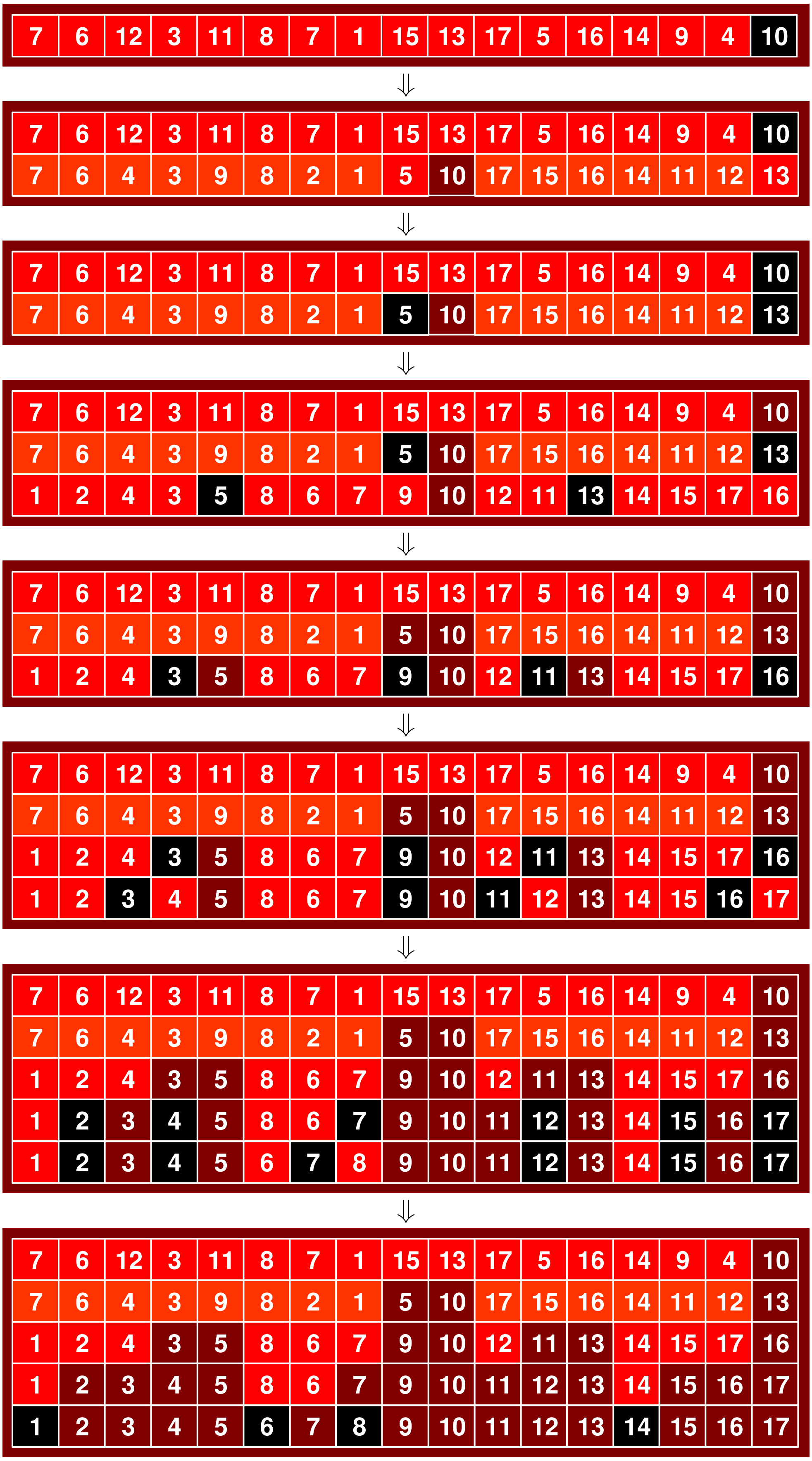


Figure 4.4: Trace of partitioning algorithm

4.3.2 Quick Sort Example

The Figure 4.5 trace out the quick sort algorithm. The first partition is done using the last element, 10, of the array. The left portion are then partitioned about 5 while the right portion is partitioned about 13. Notice that 10 is now at its final position in the eventual sorted order. The process repeats as the algorithm recursively partitions the array eventually sorting it.



7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10

⇓

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13

⇓

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13

⇓

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13
1	2	4	3	5	8	6	7	9	10	12	11	13	14	15	17	16

⇓

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13
1	2	4	3	5	8	6	7	9	10	12	11	13	14	15	17	16

⇓

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13
1	2	4	3	5	8	6	7	9	10	12	11	13	14	15	17	16
1	2	3	4	5	8	6	7	9	10	11	12	13	14	15	16	17

⇓

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13
1	2	4	3	5	8	6	7	9	10	12	11	13	14	15	17	16
1	2	3	4	5	8	6	7	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

⇓

7	6	12	3	11	8	7	1	15	13	17	5	16	14	9	4	10
7	6	4	3	9	8	2	1	5	10	17	15	16	14	11	12	13
1	2	4	3	5	8	6	7	9	10	12	11	13	14	15	17	16
1	2	3	4	5	8	6	7	9	10	11	12	13	14	15	16	17
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figure 4.5: Example of quick sort

It is interesting to note (but not surprising) that the pivots form a binary search tree. This is illustrated in Figure 4.6.

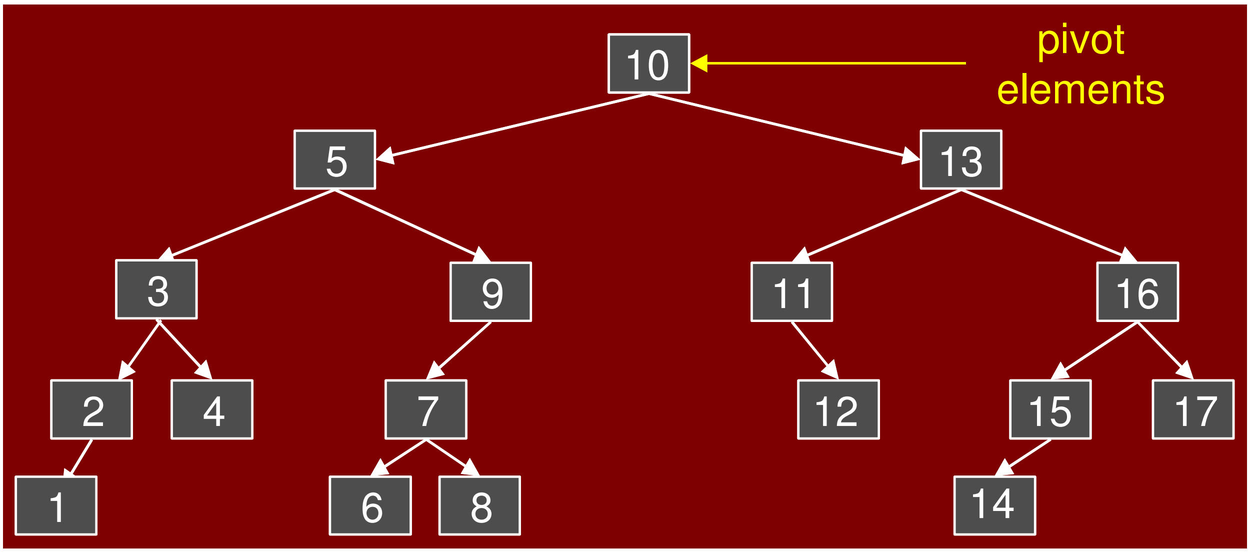


Figure 4.6: Quicksort BST

4.3.3 Analysis of Quicksort

The running time of quicksort depends heavily on the selection of the pivot. If the rank of the pivot is very large or very small then the partition (BST) will be unbalanced. Since the pivot is chosen randomly in our algorithm, the expected running time is $O(n \log n)$. The worst case time, however, is $O(n^2)$. Luckily, this happens rarely.

4.3.4 Worst Case Analysis of Quick Sort

Let's begin by considering the worst-case performance, because it is easier than the average case. Since this is a recursive program, it is natural to use a recurrence to describe its running time. But unlike MergeSort, where we had control over the sizes of the recursive calls, here we do not. It depends on how the pivot is chosen. Suppose that we are sorting an array of size n, $A[1 : n]$, and further suppose that the pivot that we select is of rank q, for some q in the range 1 to n. It takes $\Theta(n)$ time to do the partitioning and other overhead, and we make two recursive calls. The first is to the subarray $A[1 : q-1]$ which has $q-1$ elements, and the other is to the subarray $A[q+1 : n]$ which has $n-q$ elements. So if we ignore the $\Theta(n)$ (as usual) we get the recurrence:

$$T(n) = T(q-1) + T(n-q) + n$$

This depends on the value of q. To get the worst case, we maximize over all possible values of q. Putting is together, we get the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max_{1 \leq q \leq n} (T(q-1) + T(n-q) + n) & \text{otherwise} \end{cases}$$

Recurrences that have max's and min's embedded in them are very messy to solve. The key is determining which value of q gives the maximum. (A rule of thumb of algorithm analysis is that the worst cases tends to happen either at the extremes or in the middle. So I would plug in the value $q = 1$, $q = n$, and $q = n/2$ and work each out.) In this case, the worst case happens at either of the extremes. If we expand the recurrence for $q = 1$, we get:

$$\begin{aligned} T(n) &\leq T(0) + T(n-1) + n \\ &= 1 + T(n-1) + n \\ &= T(n-1) + (n+1) \\ &= T(n-2) + n + (n+1) \\ &= T(n-3) + (n-1) + n + (n+1) \\ &= T(n-4) + (n-2) + (n-1) + n + (n+1) \\ &= T(n-k) + \sum_{i=-1}^{k-2} (n-i) \end{aligned}$$

For the basis $T(1) = 1$ we set $k = n - 1$ and get

$$\begin{aligned} T(n) &\leq T(1) + \sum_{i=-1}^{n-3} (n-i) \\ &= 1 + (3 + 4 + 5 + \cdots + (n-1) + n + (n+1)) \\ &\leq \sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2} \in \Theta(n^2) \end{aligned}$$

4.3.5 Average-case Analysis of Quicksort

We will now show that in the average case, quicksort runs in $\Theta(n \log n)$ time. Recall that when we talked about average case at the beginning of the semester, we said that it depends on some assumption about the distribution of inputs. However, in the case of quicksort, the analysis does not depend on the distribution of input at all. It only depends upon the random choices of pivots that the algorithm makes. This is good, because it means that the analysis of the algorithm's performance is the same for all inputs. In this case the average is computed over all possible random choices that the algorithm might make for the choice of the pivot index in the second step of the QuickSort procedure above.

To analyze the average running time, we let $T(n)$ denote the average running time of QuickSort on a list of size n. It will simplify the analysis to assume that all of the elements are distinct. The algorithm has n random choices for the pivot element, and each choice has an equal probability of $1/n$ of occuring. So we can modify the above recurrence to compute an average rather than a max, giving:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \frac{1}{n}\sum_{q=1}^{n}(T(q-1) + T(n-q) + n) & \text{otherwise} \end{cases}$$

The time $T(n)$ is the weighted sum of the times taken for various choices of q. I.e.,

$$\begin{aligned} T(n) = \Big[&\frac{1}{n}(T(0) + T(n-1) + n) \\ &+ \frac{1}{n}(T(1) + T(n-2) + n) \\ &+ \frac{1}{n}(T(2) + T(n-3) + n) \\ &+ \cdots + \frac{1}{n}(T(n-1) + T(0) + n)\Big] \end{aligned}$$

We have not seen such a recurrence before. To solve it, expansion is possible but it is rather tricky. We will attempt a constructive induction to solve it. We know that we want a $\Theta(n \log n)$. Let us assume that $T(n) \leq cn \log n$ for $n \geq 2$ where c is a constant.

For the base case $n = 2$ we have

$$\begin{aligned} T(n) &= \frac{1}{2}\sum_{q=1}^{2}(T(q-1) + T(2-q) + 2) \\ &= \frac{1}{2}\Big[(T(0) + T(1) + 2) + (T(1) + T(0) + 2)\Big] \\ &= \frac{8}{2} = 4. \end{aligned}$$

We want this to be at most $c2\log 2$, i.e.,

$$T(2) \leq c2\log 2$$

or

$$4 \leq c2\log 2$$

therefore

$$c \geq 4/(2\log 2) \approx 2.88.$$

For the induction step, we assume that $n \geq 3$ and The induction hypothesis is that for any $n' < n$, we have $T(n') \leq cn' \log n'$. We want to prove that it is true for $T(n)$. By expanding $T(n)$ and moving the factor of n outside the sum we have

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q)+n) \\ &= \frac{1}{n}\sum_{q=1}^{n}(T(q-1)+T(n-q))+n \\ &= \frac{1}{n}\sum_{q=1}^{n}T(q-1)+\frac{1}{n}\sum_{q=1}^{n}T(n-q)+n \end{aligned}$$

$$T(n) = \frac{1}{n}\sum_{q=1}^{n}T(q-1)+\frac{1}{n}\sum_{q=1}^{n}T(n-q)+n$$

Observe that the two sums add up the same values $T(0)+T(1)+\cdots+T(n-1)$. One counts up and other counts down. Thus we can replace them with $2\sum_{q=0}^{n-1}T(q)$. We will extract $T(0)$ and $T(1)$ and treat them specially. These two do not follow the formula.

$$\begin{aligned} T(n) &= \frac{2}{n}\Big(\sum_{q=0}^{n-1}T(q)\Big)+n \\ &= \frac{2}{n}\Big(T(0)+T(1)+\sum_{q=2}^{n-1}T(q)\Big)+n \end{aligned}$$

We will apply the induction hypothesis for $q < n$ we have

$$\begin{aligned} T(n) &= \frac{2}{n}\Big(T(0)+T(1)+\sum_{q=2}^{n-1}T(q)\Big)+n \\ &\le \frac{2}{n}\Big(1+1+\sum_{q=2}^{n-1}(cq\log q)\Big)+n \\ &= \frac{2c}{n}\Big(\sum_{q=2}^{n-1}(cq\ln q)\Big)+n+\frac{4}{n} \end{aligned}$$

We have never seen this sum before:

$$S(n) = \sum_{q=2}^{n-1}(cq\ln q)$$

Recall from calculus that for any monotonically increasing function $f(x)$

$$\sum_{i=a}^{b-1}f(i) \le \int_a^b f(x)dx$$

The function $f(x) = x\ln x$ is monotonically increasing, and so

$$S(n) = \sum_{q=2}^{n-1} (cq\ln q) \leq \int_2^n x\ln x \, dx \tag{4.1}$$

We can integrate this by parts:

$$\int_2^n x\ln x \, dx = \frac{x^2}{2}\ln x - \frac{x^2}{4}\Big|_{x=2}^{n}$$

$$\begin{aligned}\int_2^n x\ln x \, dx &= \frac{x^2}{2}\ln x - \frac{x^2}{4}\Big|_{x=2}^{n} \\ &= \left(\frac{n^2}{2}\ln n - \frac{n^2}{4}\right) - (2\ln 2 - 1) \\ &\leq \frac{n^2}{2}\ln n - \frac{n^2}{4}\end{aligned}$$

We thus have

$$S(n) = \sum_{q=2}^{n-1} (cq\ln q) \leq \frac{n^2}{2}\ln n - \frac{n^2}{4}$$

Plug this back into the expression for $T(n)$ to get

$$T(n) = \frac{2c}{n}\left(\frac{n^2}{2}\ln n - \frac{n^2}{4}\right) + n + \frac{4}{n}$$

$$\begin{aligned}T(n) &= \frac{2c}{n}\left(\frac{n^2}{2}\ln n - \frac{n^2}{4}\right) + n + \frac{4}{n} \\ &= cn\ln n - \frac{cn}{2} + n + \frac{4}{n} \\ &= cn\ln n + n\left(1 - \frac{c}{2}\right) + \frac{4}{n}\end{aligned}$$

$$T(n) = cn\ln n + n\left(1 - \frac{c}{2}\right) + \frac{4}{n}$$

To finish the proof,we want all of this to be at most $cn\ln n$. For this to happen, we will need to select c such that

$$n\left(1 - \frac{c}{2}\right) + \frac{4}{n} \leq 0$$

If we select c $= 3$, and use the fact that $n \geq 3$ we get

$$\begin{aligned}n\left(1 - \frac{c}{2}\right) + \frac{4}{n} &= \frac{3}{n} - \frac{n}{2} \\ &\leq 1 - \frac{3}{2} = -\frac{1}{2} \leq 0.\end{aligned}$$

From the basis case we had c ≥ 2.88. Choosing c $= 3$ satisfies all the constraints. Thus $T(n) = 3n\ln n \in \Theta(n \log n)$.

4.4 In-place, Stable Sorting

An *in-place* sorting algorithm is one that uses no additional array for storage. A sorting algorithm is *stable* if duplicate elements remain in the same relative position after sorting.

9	3	$3'$	5	6	$5'$	2	1	$3''$	unsorted
1	2	3	$3'$	$3''$	5	$5'$	6	9	stable sort
1	2	$3'$	3	$3''$	$5'$	5	6	9	unstable

Bubble sort, insertion sort and selection sort are in-place sorting algorithms. Bubble sort and insertion sort can be implemented as stable algorithms but selection sort cannot (without significant modifications). Mergesort is a stable algorithm but not an in-place algorithm. It requires extra array storage. Quicksort is not stable but is an in-place algorithm. Heapsort is an in-place algorithm but is not stable.

4.5 Lower Bounds for Sorting

The best we have seen so far is $O(n \log n)$ algorithms for sorting. Is it possible to do better than $O(n \log n)$? If a sorting algorithm is solely based on comparison of keys in the array then it is *impossible* to sort more efficiently than $\Omega(n \log n)$ time. All algorithms we have seen so far are comparison-based sorting algorithms.

Consider sorting three numbers a_1, a_2, a_3. There are $3! = 6$ possible combinations:

$$\begin{array}{c}(a_1, a_2, a_3),\ (a_1, a_3, a_2)\ ,(a_3, a_2, a_1)\\(a_3, a_1, a_2),\ (a_2, a_1, a_3)\ ,(a_2, a_3, a_1)\end{array}$$

One of these permutations leads to the numbers in sorted order.

The comparison based algorithm defines a *decision tree*. Here is the tree for the three numbers.

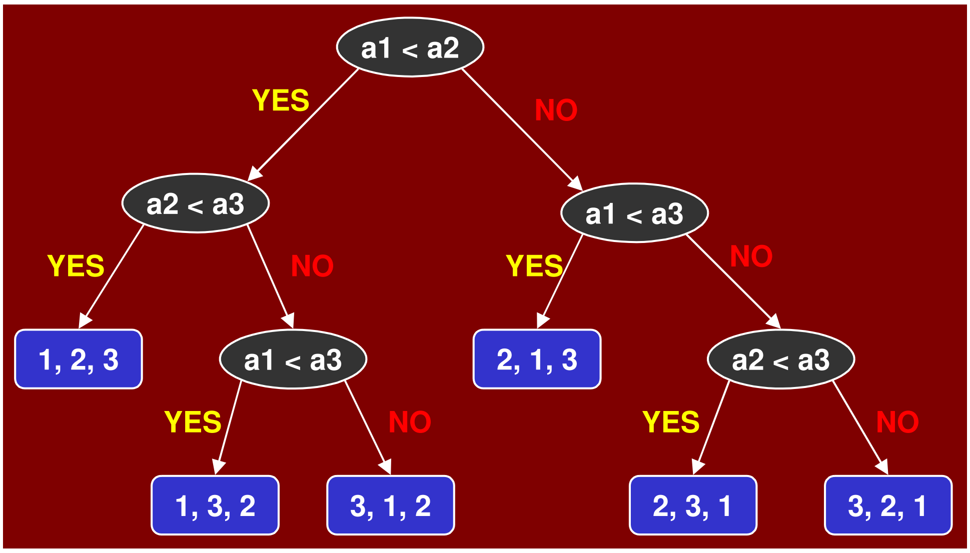


Figure 4.7: Decision Tree

For n elements, there will be $n!$ possible permutations. The height of the tree is exactly equal to $T(n)$, the running time of the algorithm. The height is $T(n)$ because any path from the root to a leaf corresponds to a sequence of comparisons made by the algorithm.

Any binary tree of height $T(n)$ has at most $2^{T(n)}$ leaves. Thus a comparison based sorting algorithm can distinguish between at most $2^{T(n)}$ different final outcomes. So we have

$$\begin{aligned} 2^{T(n)} &\geq n! \qquad \text{and therefore} \\ T(n) &\geq \log(n!) \end{aligned}$$

We can use *Stirling's approximation* for $n!$:

$$n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Thereofore

$$\begin{aligned} T(n) &\geq \log\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \\ &= \log(\sqrt{2\pi n}) + n \log n - n \log e \\ &\in \Omega(n \log n) \end{aligned}$$

We thus have the following theorem.

Theorem 1
Any comparison-based sorting algorithm has worst-case running time $\Omega(n \log n)$.

Chapter 5

Linear Time Sorting

The lower bound implies that if we hope to sort numbers faster than $O(n \log n)$, we cannot do it by making comparisons alone. Is it possible to sort without making comparisons? The answer is yes, but only under very restrictive circumstances. Many applications involve sorting small integers (e.g. sorting characters, exam scores, etc.). We present three algorithms based on the theme of speeding up sorting in special cases, by not making comparisons.

5.1 Counting Sort

We will consider three algorithms that are faster and work by not making comparisons. Counting sort assumes that the numbers to be sorted are in the range 1 to k where k is small. The basic idea is to determine the rank of each number in final sorted array.

Recall that the rank of an item is the number of elements that are less than or equal to it. Once we know the ranks, we simply copy numbers to their final position in an output array.

The question is how to find the rank of an element without comparing it to the other elements of the array?. The algorithm uses three arrays. As usual, $A[1..n]$ holds the initial input, $B[1..n]$ holds the sorted output and $C[1..k]$ is an array of integers. $C[x]$ is the rank of x in A, where $x \in [1..k]$. The algorithm is remarkably simple, but deceptively clever. The algorithm operates by first constructing C. This is done in two steps. First we set $C[x]$ to be the number of elements of $A[j]$ that are equal to x. We can do this initializing C to zero, and then for each j, from 1 to n, we increment $C[A[j]]$ by 1. Thus, if $A[j] = 5$, then the 5th element of C is incremented, indicating that we have seen one more 5. To determine the number of elements that are less than or equal to x, we replace $C[x]$ with the sum of elements in the sub array $R[1 : x]$. This is done by just keeping a running total of the elements of C.

$C[x]$ now contains the rank of x. This means that if $x = A[j]$ then the final position of $A[j]$ should be at position $C[x]$ in the final sorted array. Thus, we set $B[C[x]] = A[j]$. Notice We need to be careful if there are duplicates, since we do not want them to overwrite the same location of B. To do this, we decrement

$C[i]$ after copying.

```
COUNTING-SORT( array $A$, array $B$, int $k$)
for $i \leftarrow 1$ to $k$
    do $C[i] \leftarrow 0$              $\boxed{k \text{ times}}$
for $j \leftarrow 1$ to $length[A]$
    do $C[A[j]] \leftarrow C[A[j]] + 1$        $\boxed{n \text{ times}}$
// $C[i]$ now contains the number of elements $= i$
for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$         $\boxed{k \text{ times}}$
// $C[i]$ now contains the number of elements $\le i$
for $j \leftarrow length[A]$ downto $1$
    do $B[C[A[j]]] \leftarrow A[j]$
       $C[A[j]] \leftarrow C[A[j]] - 1$        $\boxed{n \text{ times}}$
```

There are four (unnested) loops, executed k times, n times, $k-1$ times, and n times, respectively, so the total running time is $\Theta(n+k)$ time. If $k = O(n)$, then the total running time is $\Theta(n)$.

Figure 5.1 through 5.19 shows an example of the algorithm. You should trace through the example to convince yourself how it works.

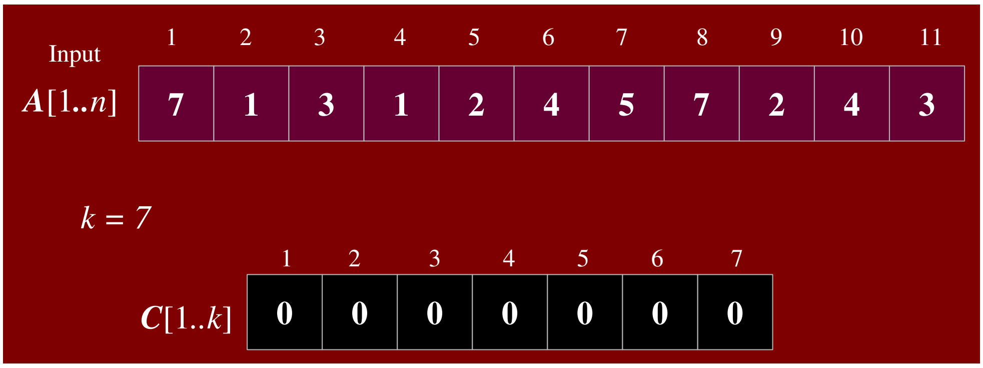


Figure 5.1: Initial A and C arrays.

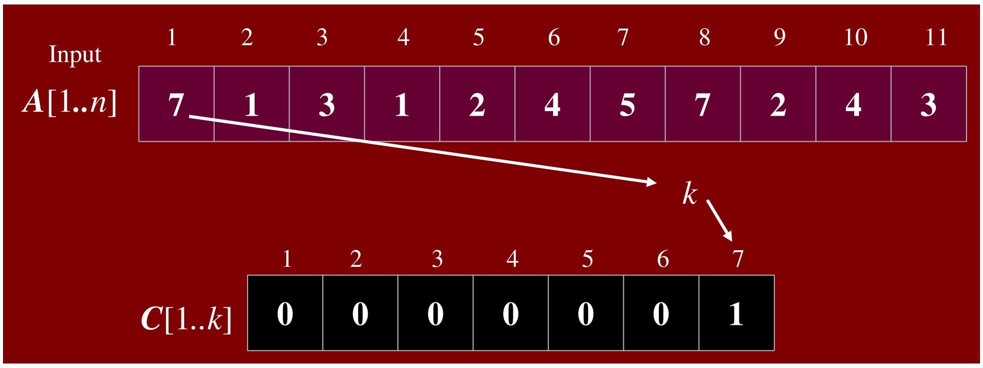


Figure 5.2: $A[1] = 7$ processed

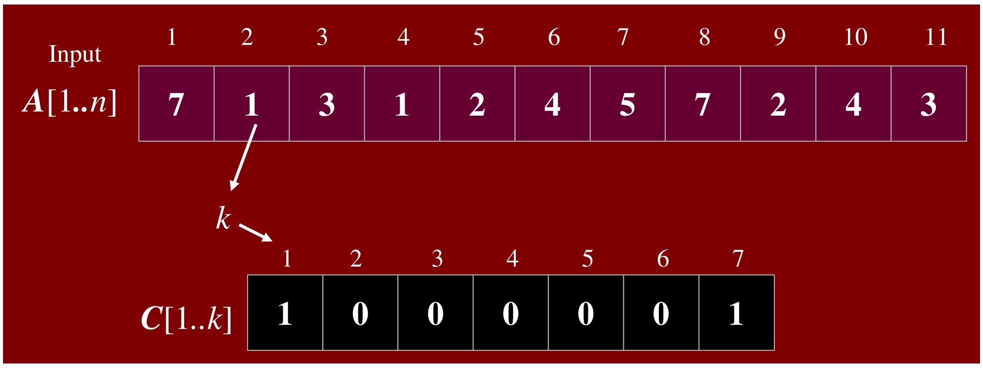


Figure 5.3: $A[2] = 1$ processed

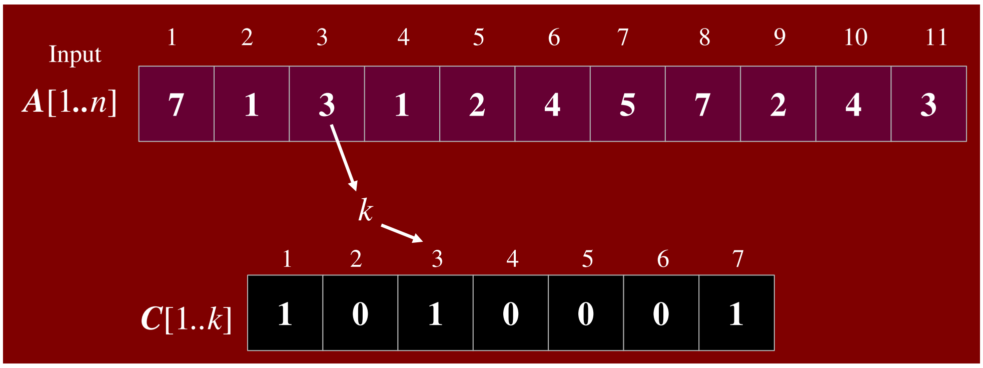


Figure 5.4: $A[3] = 3$ processed

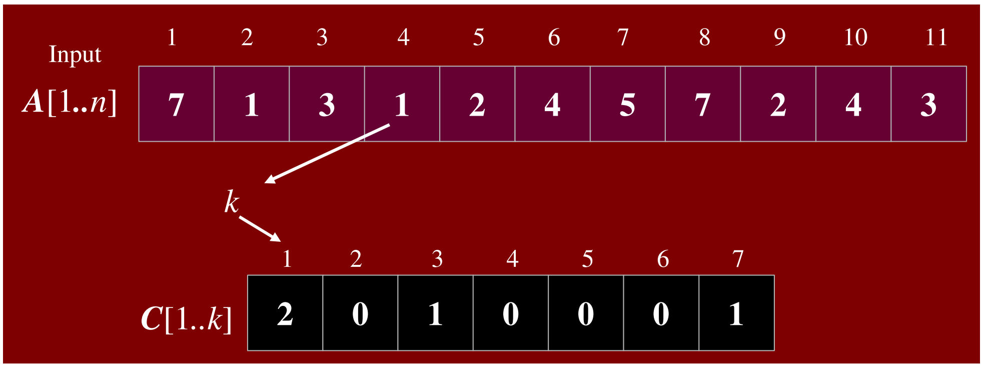


Figure 5.5: $A[4] = 1$ processed

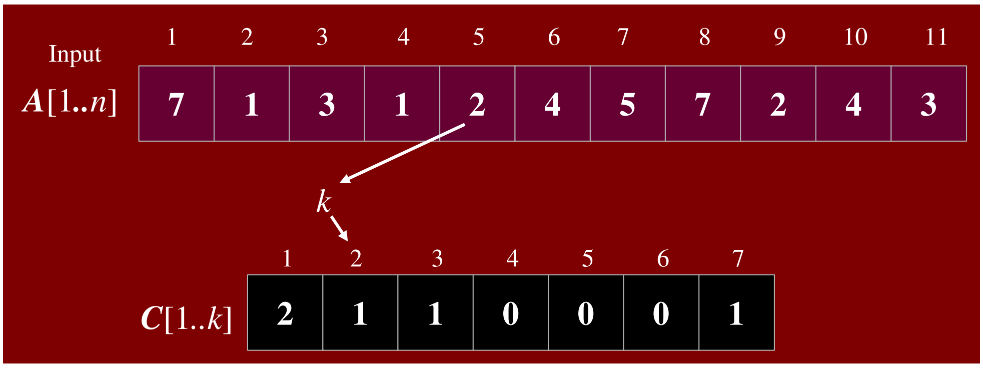


Figure 5.6: $A[5] = 2$ processed

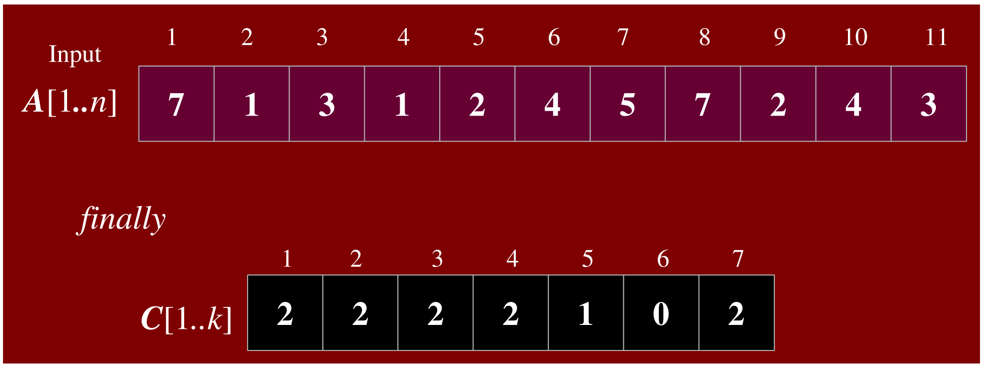
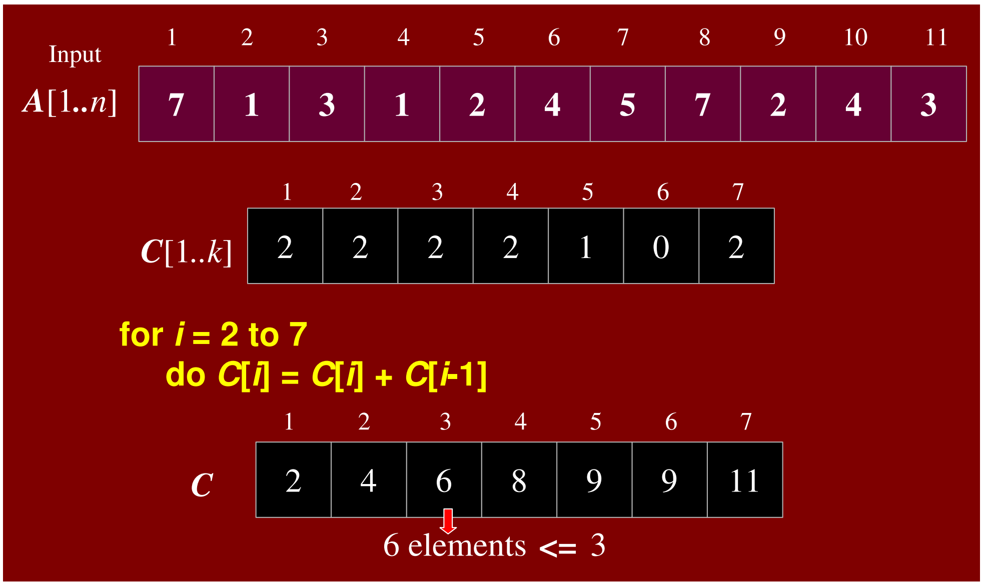


Figure 5.7: C now contains count of elements of A



Input

	1	2	3	4	5	6	7	8	9	10	11
$A[1..n]$	**7**	**1**	**3**	**1**	**2**	**4**	**5**	**7**	**2**	**4**	**3**

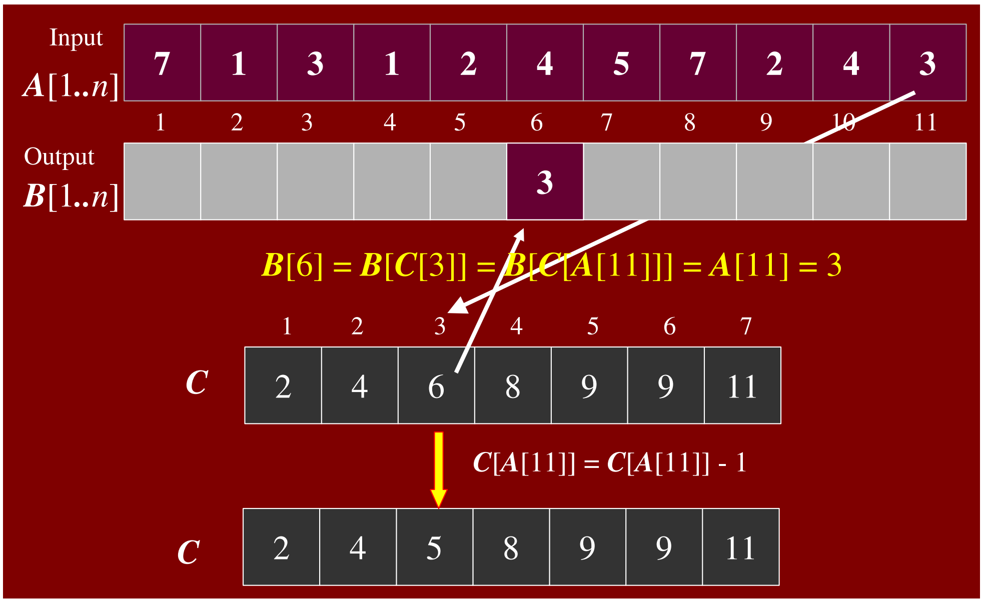
	1	2	3	4	5	6	7
$C[1..k]$	2	2	2	2	1	0	2

```
for i = 2 to 7
    do C[i] = C[i] + C[i-1]
```

	1	2	3	4	5	6	7
C	2	4	6	8	9	9	11

6 elements <= 3

Figure 5.8: C set to rank each number of A



Input

$A[1..n]$	**7**	**1**	**3**	**1**	**2**	**4**	**5**	**7**	**2**	**4**	**3**
	1	2	3	4	5	6	7	8	9	10	11

Output

$B[1..n]$						**3**					

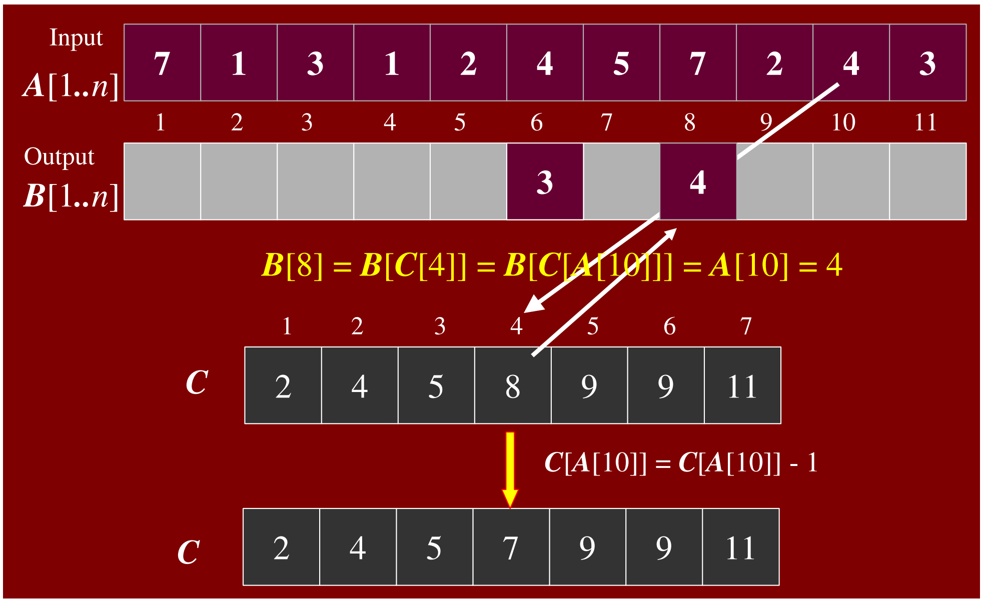
$\boldsymbol{B}[6] = \boldsymbol{B}[\boldsymbol{C}[3]] = \boldsymbol{B}[\boldsymbol{C}[\boldsymbol{A}[11]]] = \boldsymbol{A}[11] = 3$

	1	2	3	4	5	6	7
C	2	4	6	8	9	9	11

$\boldsymbol{C}[\boldsymbol{A}[11]] = \boldsymbol{C}[\boldsymbol{A}[11]] - 1$

C	2	4	5	8	9	9	11

Figure 5.9: A[11] placed in output array B



Input $A[1..n]$

1	2	3	4	5	6	7	8	9	10	11
7	1	3	1	2	4	5	7	2	4	3

Output $B[1..n]$

1	2	3	4	5	6	7	8	9	10	11
					3		4			

$B[8] = B[C[4]] = B[C[A[10]]] = A[10] = 4$

C

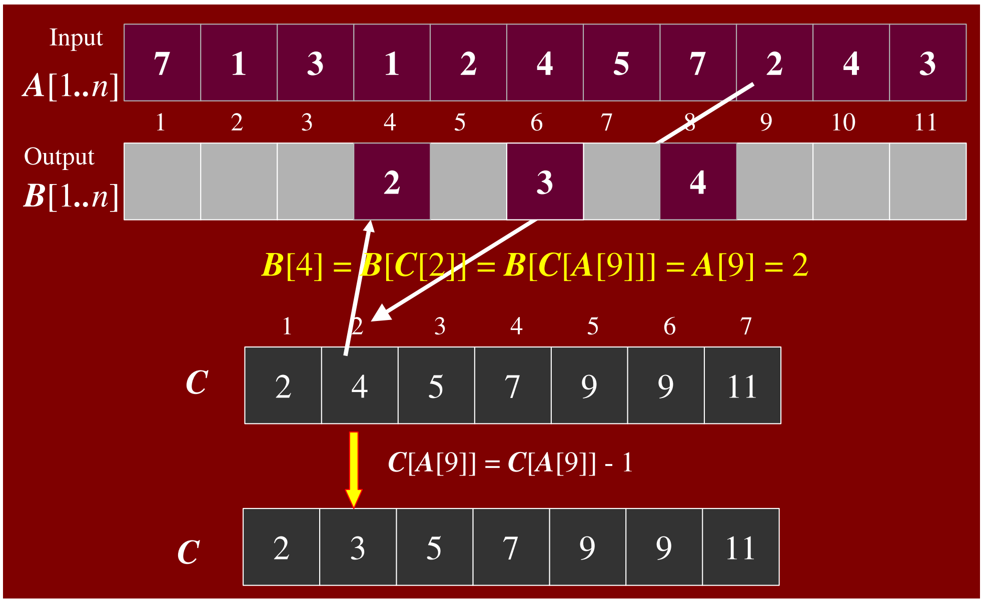
1	2	3	4	5	6	7
2	4	5	8	9	9	11

$C[A[10]] = C[A[10]] - 1$

C

2	4	5	7	9	9	11

Figure 5.10: $A[10]$ placed in output array B



Input $A[1..n]$

1	2	3	4	5	6	7	8	9	10	11
7	1	3	1	2	4	5	7	2	4	3

Output $B[1..n]$

1	2	3	4	5	6	7	8	9	10	11
			2		3		4			

$B[4] = B[C[2]] = B[C[A[9]]] = A[9] = 2$

C

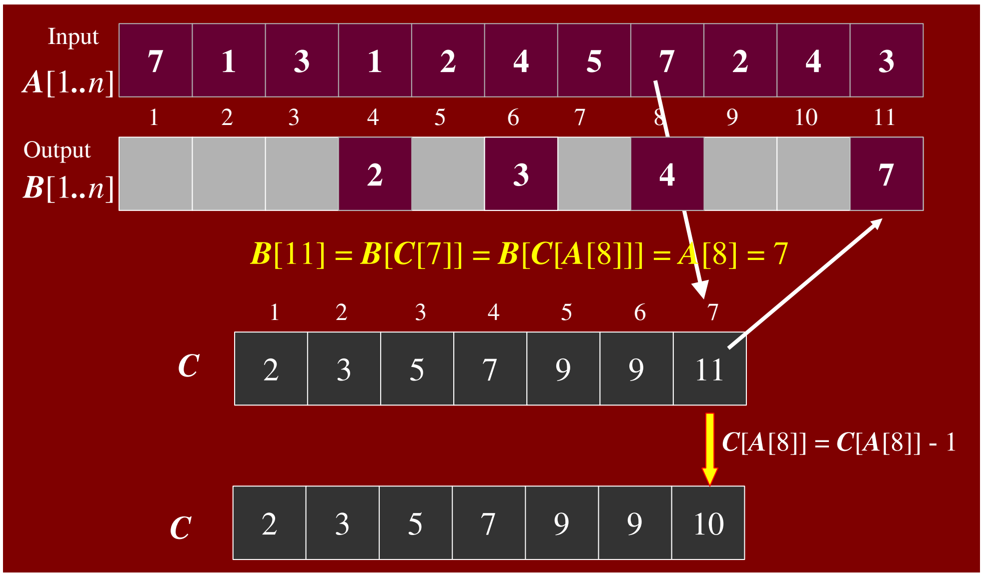
1	2	3	4	5	6	7
2	4	5	7	9	9	11

$C[A[9]] = C[A[9]] - 1$

C

2	3	5	7	9	9	11

Figure 5.11: $A[9]$ placed in output array B



	1	2	3	4	5	6	7	8	9	10	11
Input $A[1..n]$	**7**	**1**	**3**	**1**	**2**	**4**	**5**	**7**	**2**	**4**	**3**
Output $B[1..n]$				**2**		**3**		**4**			**7**

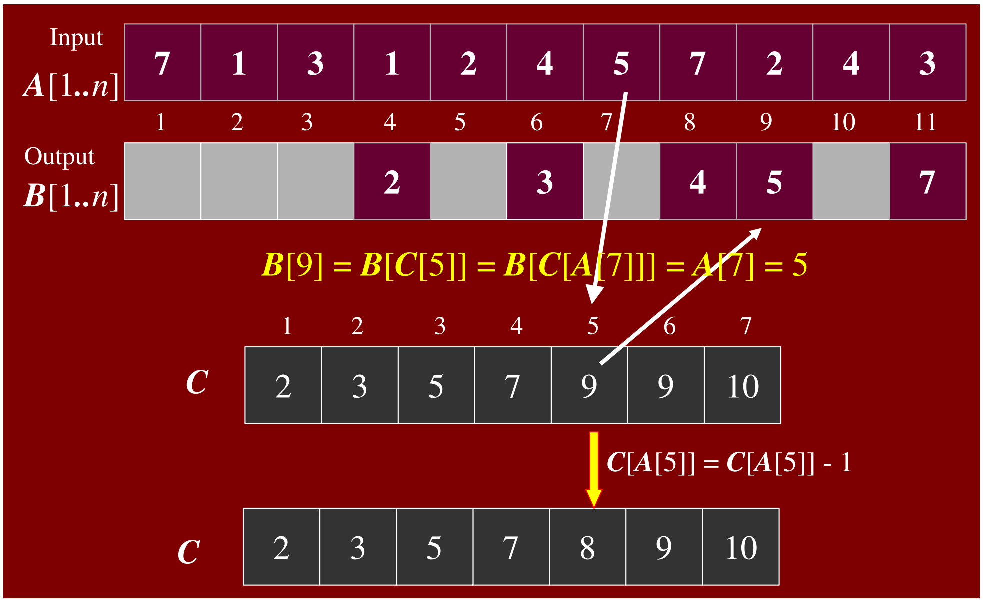
$B[11] = B[C[7]] = B[C[A[8]]] = A[8] = 7$

	1	2	3	4	5	6	7
C	2	3	5	7	9	9	11

$C[A[8]] = C[A[8]] - 1$

C	2	3	5	7	9	9	10

Figure 5.12: A[8] placed in output array B



	1	2	3	4	5	6	7	8	9	10	11
Input $A[1..n]$	**7**	**1**	**3**	**1**	**2**	**4**	**5**	**7**	**2**	**4**	**3**
Output $B[1..n]$				**2**		**3**		**4**	**5**		**7**

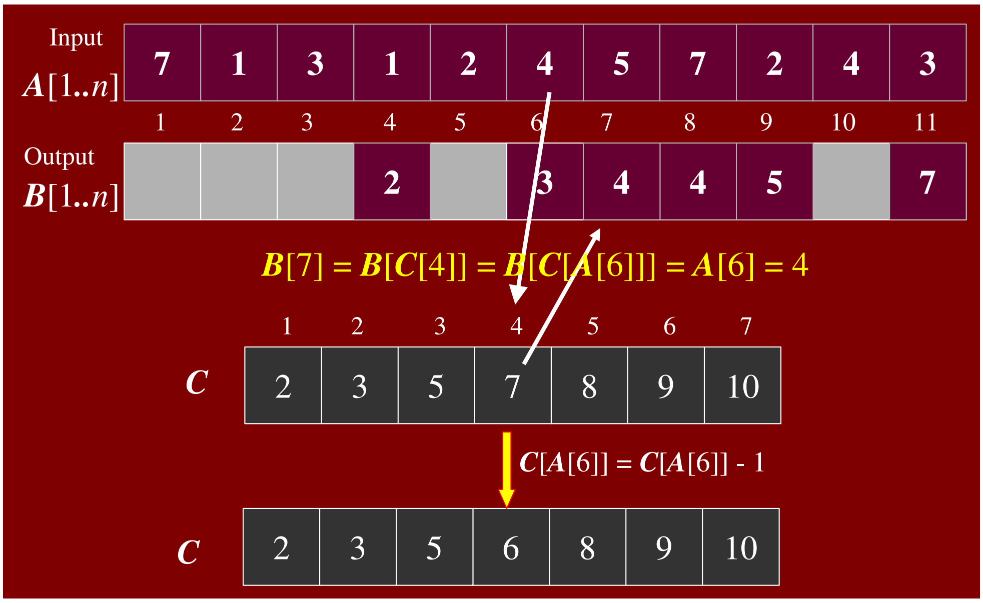
$B[9] = B[C[5]] = B[C[A[7]]] = A[7] = 5$

	1	2	3	4	5	6	7
C	2	3	5	7	9	9	10

$C[A[5]] = C[A[5]] - 1$

C	2	3	5	7	8	9	10

Figure 5.13: A[7] placed in output array B



	1	2	3	4	5	6	7	8	9	10	11
Input $A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
Output $B[1..n]$				2		3	4	4	5		7

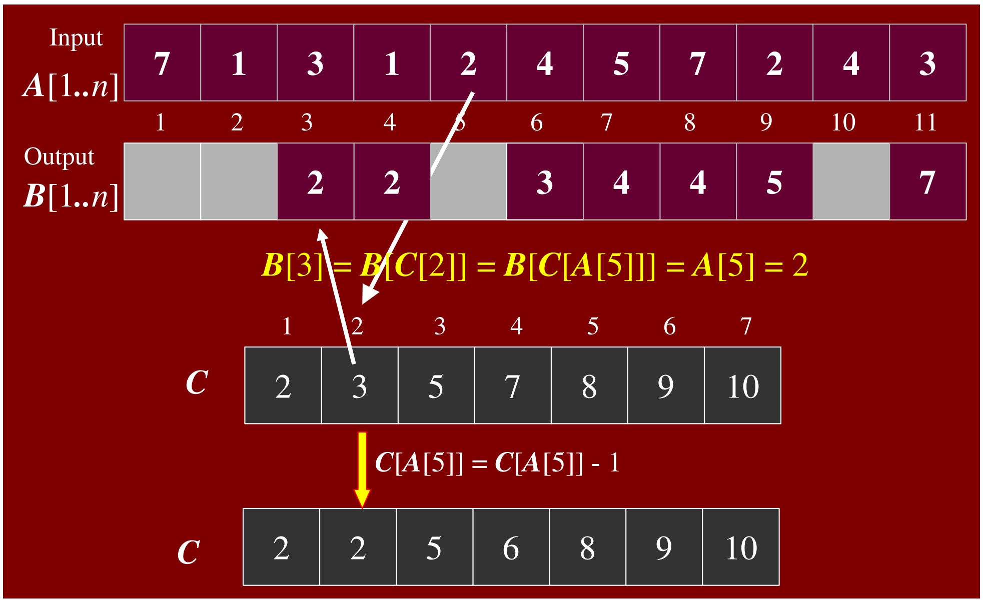
$B[7] = B[C[4]] = B[C[A[6]]] = A[6] = 4$

	1	2	3	4	5	6	7
C	2	3	5	7	8	9	10

$C[A[6]] = C[A[6]] - 1$

C	2	3	5	6	8	9	10

Figure 5.14: A[6] placed in output array B



	1	2	3	4	5	6	7	8	9	10	11
Input $A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
Output $B[1..n]$			2	2		3	4	4	5		7

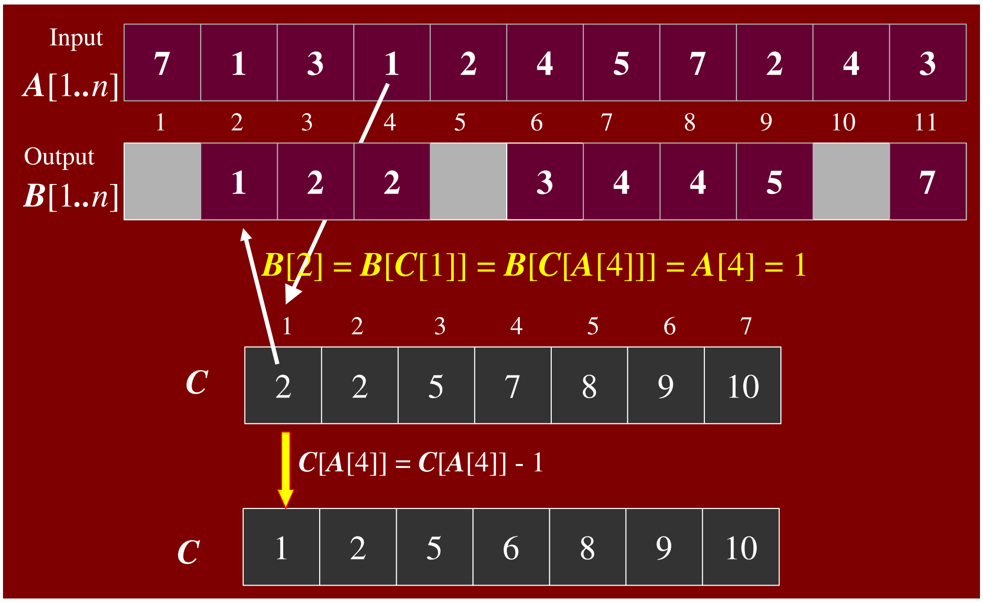
$B[3] = B[C[2]] = B[C[A[5]]] = A[5] = 2$

	1	2	3	4	5	6	7
C	2	3	5	7	8	9	10

$C[A[5]] = C[A[5]] - 1$

C	2	2	5	6	8	9	10

Figure 5.15: A[5] placed in output array B



Input $A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11
Output $B[1..n]$		1	2	2		3	4	4	5		7

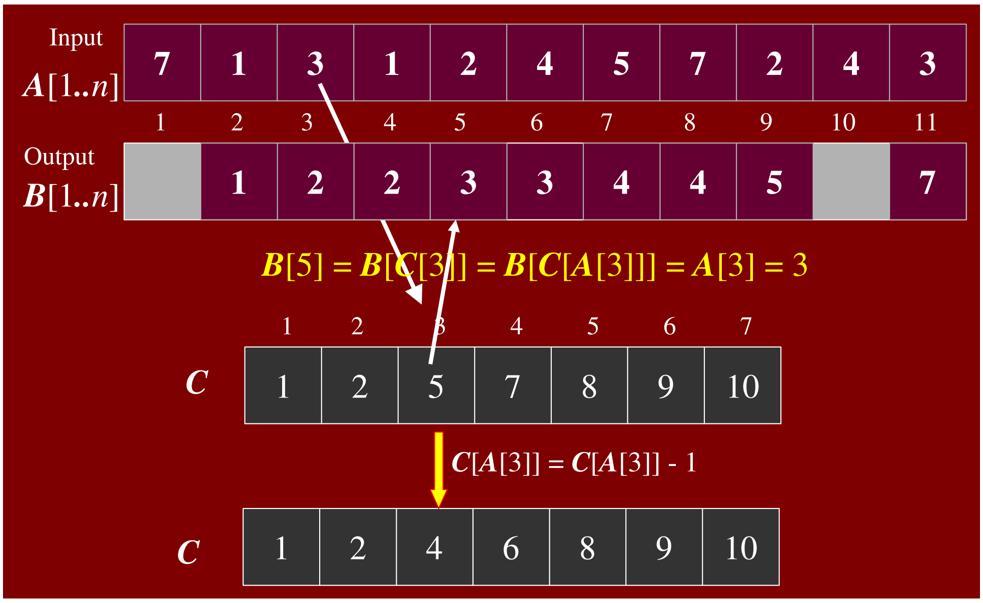
$B[2] = B[C[1]] = B[C[A[4]]] = A[4] = 1$

	1	2	3	4	5	6	7
C	2	2	5	7	8	9	10

$C[A[4]] = C[A[4]] - 1$

C	1	2	5	6	8	9	10

Figure 5.16: A[4] placed in output array B



Input $A[1..n]$	7	1	3	1	2	4	5	7	2	4	3
	1	2	3	4	5	6	7	8	9	10	11
Output $B[1..n]$		1	2	2	3	3	4	4	5		7

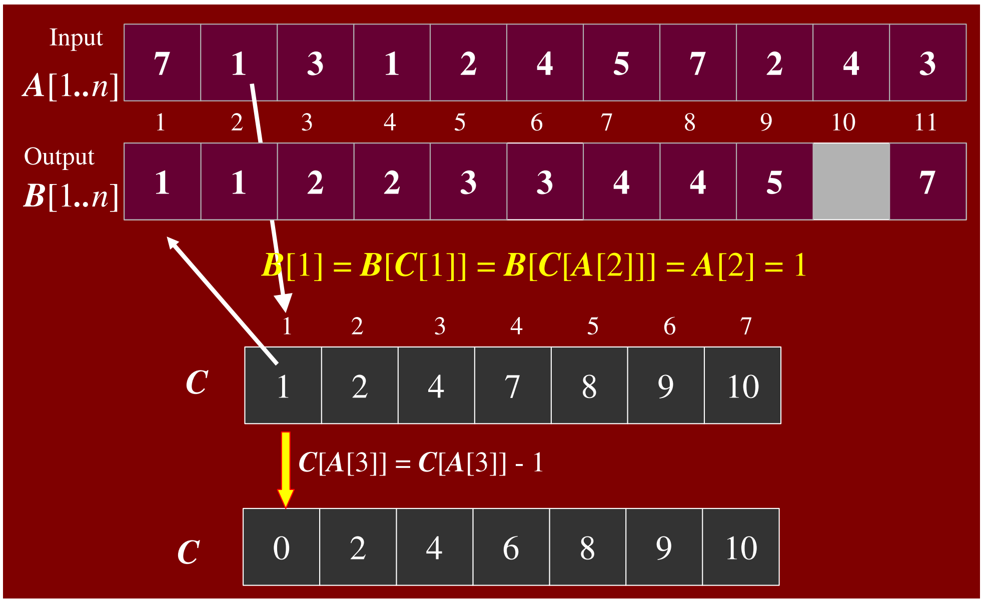
$B[5] = B[C[3]] = B[C[A[3]]] = A[3] = 3$

	1	2	3	4	5	6	7
C	1	2	5	7	8	9	10

$C[A[3]] = C[A[3]] - 1$

C	1	2	4	6	8	9	10

Figure 5.17: A[3] placed in output array B



Input

A[1..*n*]

7 1 3 1 2 4 5 7 2 4 3

1 2 3 4 5 6 7 8 9 10 11

Output

B[1..*n*]

1 1 2 2 3 3 4 4 5 7

B[1] = *B*[*C*[1]] = *B*[*C*[*A*[2]]] = *A*[2] = 1

1 2 3 4 5 6 7

C

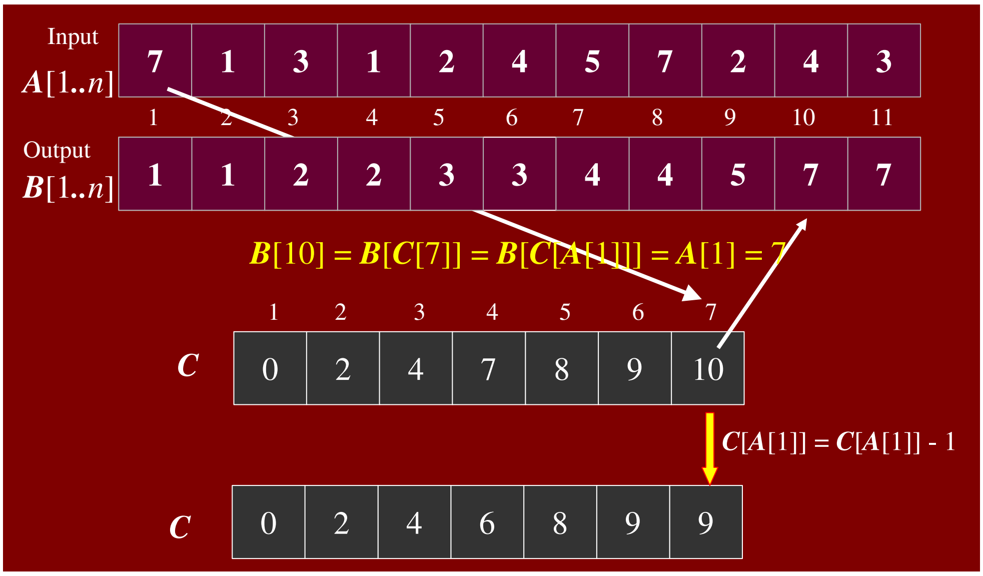
1 2 4 7 8 9 10

C[*A*[3]] = *C*[*A*[3]] - 1

C

0 2 4 6 8 9 10

Figure 5.18: $A[2]$ placed in output array B



Input

A[1..*n*]

7 1 3 1 2 4 5 7 2 4 3

1 2 3 4 5 6 7 8 9 10 11

Output

B[1..*n*]

1 1 2 2 3 3 4 4 5 7 7

B[10] = *B*[*C*[7]] = *B*[*C*[*A*[1]]] = *A*[1] = 7

1 2 3 4 5 6 7

C

0 2 4 7 8 9 10

C[*A*[1]] = *C*[*A*[1]] - 1

C

0 2 4 6 8 9 9

Figure 5.19: B now contains the final sorted data.

Counting sort is not an in-place sorting algorithm but it is stable. Stability is important because data are often carried with the keys being sorted. radix sort (which uses counting sort as a subroutine) relies on it to work correctly. Stability achieved by running the loop down from n to 1 and not the other way around:

```
COUNTING-SORT( array $A$, array $B$, int $k$)
⋮
for $j \leftarrow length[A]$ downto $1$
do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$
```

Figure 5.20 illustrates the stability. The numbers 1, 2, 3, 4, and 7, each appear twice. The two 4’s have been given the superscript “*****”. Numbers are placed in the output B array starting from the right. The two 4’s maintain their relative position in the B array. If the sorting algorithm had caused 4^{**} to end up on the left of 4^{*}, the algorithm would be termed unstable.

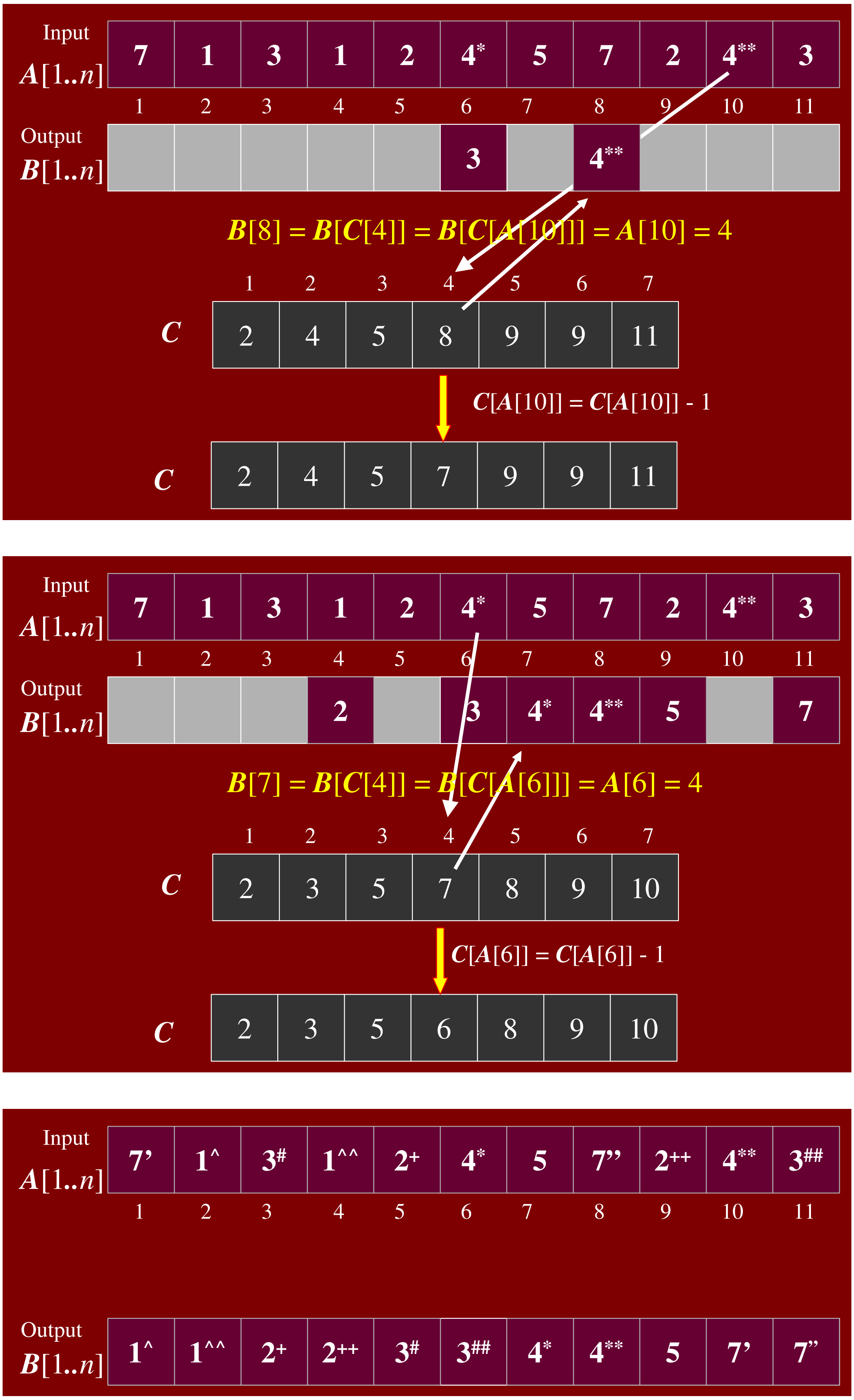


Figure 5.20: Stability of counting sort

5.2 Bucket or Bin Sort

Assume that the keys of the items that we wish to sort lie in a small fixed range and that there is only one item with each value of the key. Then we can sort with the following procedure:

1. Set up an array of “bins” - one for each value of the key - in order,

2. Examine each item and use the value of the key to place it in the appropriate bin.

Now our collection is sorted and it only took n operations, so this is an $O(n)$ operation. However, note that it will only work under very restricted conditions. To understand these restrictions, let’s be a little more precise about the specification of the problem and assume that there are m values of the key. To recover our sorted collection, we need to examine each bin. This adds a third step to the algorithm above,

3. Examine each bin to see whether there’s an item in it.

which requires m operations. So the algorithm’s time becomes:

$$T(n) = c_1 n + c_2 m$$

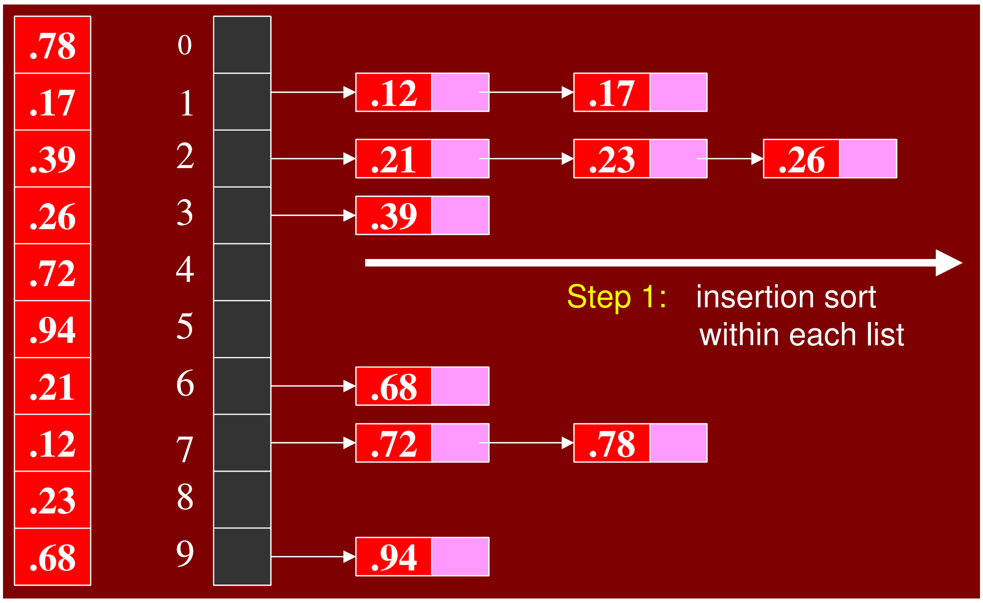
and it is strictly $O(n+m)$. If $m \leq n$, this is clearly $O(n)$. However if $m >> n$, then it is $O(m)$. An implementation of bin sort might look like:

```
BUCEKTSORT( array $A$, int $n$, int $M$)
// Pre-condition: for $1 \leq i \leq n$, $0 \leq a[i] < M$
// Mark all the bins empty
for $i \leftarrow 1$ to $M$
do $bin[i] \leftarrow Empty$
for $i \leftarrow 1$ to $n$
do $bin[A[i]] \leftarrow A[i]$
```

If there are *duplicates*, then each bin can be replaced by a *linked list*. The third step then becomes:

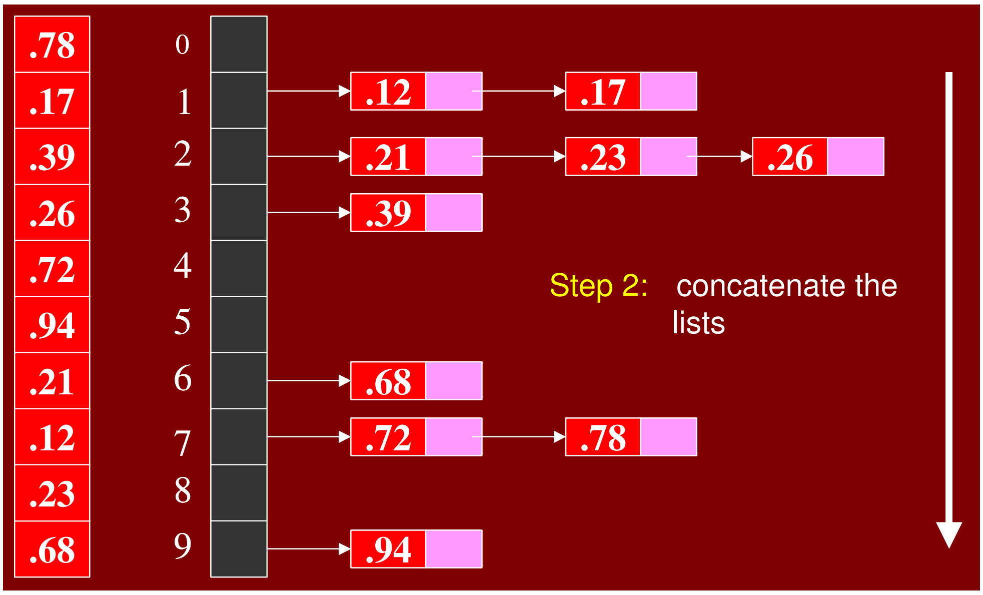
3. Link all the lists into one list.

We can add an item to a linked list in $O(1)$ time. There are n items requiring $O(n)$ time. Linking a list to another list simply involves making the tail of one list point to the other, so it is $O(1)$. Linking m such lists obviously takes $O(m)$ time, so the algorithm is still $O(n+m)$. Figures 5.21 through 5.23 show the algorithm in action using linked lists.



Step 1: insertion sort within each list

Figure 5.21: Bucket sort: step 1, placing keys in bins in sorted order



Step 2: concatenate the lists

Figure 5.22: Bucket sort: step 2, concatenate the lists

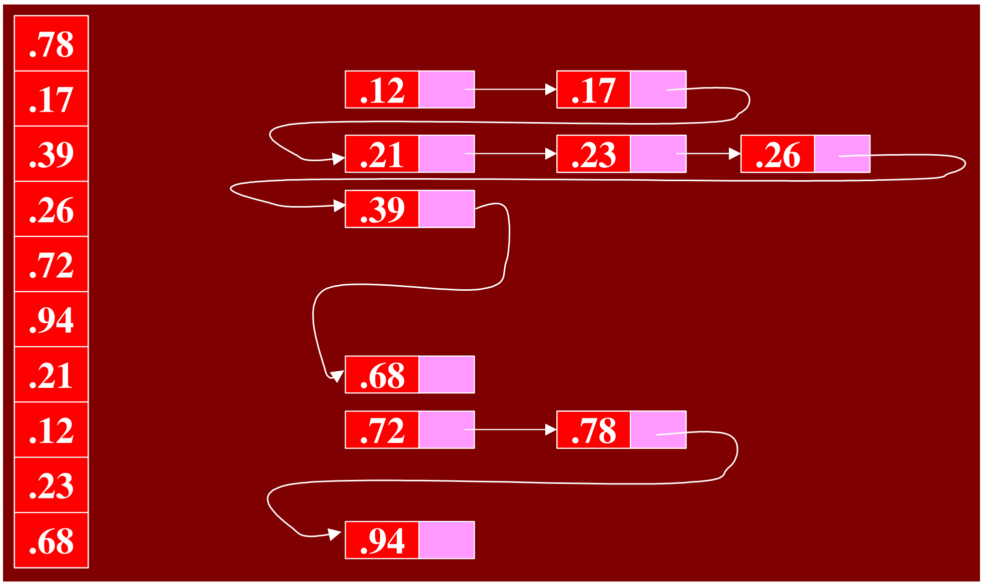


Figure 5.23: Bucket sort: the final sorted sequence

5.3 Radix Sort

The main shortcoming of counting sort is that it is useful for small integers, i.e., $1..k$ where k is small. If k were a million or more, the size of the rank array would also be a million. Radix sort provides a nice work around this limitation by sorting numbers one digit at a time.

576		49[4]		9[5]4		[1]76		176
494		19[4]		5[7]6		[1]94		194
194		95[4]		1[7]6		[2]78		278
296	⇒	57[6]	⇒	2[7]8	⇒	[2]96	⇒	296
278		29[6]		4[9]4		[4]94		494
176		17[6]		1[9]4		[5]76		576
954		27[8]		2[9]6		[9]54		954

Here is the algorithm that sorts $A[1..n]$ where each number is d digits long.

```
RADIX-SORT( array A, int n, int d)
for i ← 1 to d
    do stably sort A w.r.t i^th lowest order digit
```

Chapter 6

Dynamic Programming

6.1 Fibonacci Sequence

Suppose we put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive? Resulting sequence is $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ where each number is the sum of the two preceding numbers.

This problem was posed by Leonardo Pisano, better known by his nickname Fibonacci (son of Bonacci, born 1170, died 1250). This problem and many others were in posed in his book *Liber abaci*, published in 1202. The book was based on the arithmetic and algebra that Fibonacci had accumulated during his travels. The book, which went on to be widely copied and imitated, introduced the Hindu-Arabic place-valued decimal system and the use of Arabic numerals into Europe. The rabbits problem in the third section of Liber abaci led to the introduction of the Fibonacci numbers and the Fibonacci sequence for which Fibonacci is best remembered today.

This sequence, in which each number is the sum of the two preceding numbers, has proved extremely fruitful and appears in many different areas of mathematics and science. The *Fibonacci Quarterly* is a modern journal devoted to studying mathematics related to this sequence. The Fibonacci numbers F_n are defined as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

The recursive definition of Fibonacci numbers gives us a recursive algorithm for computing them:

```
FIB($n$)
if ($n < 2$)
    then return $n$
    else  return FIB($n - 1$) + FIB($n - 2$)
```

Figure **??** shows four levels of recursion for the call $fib(8)$:

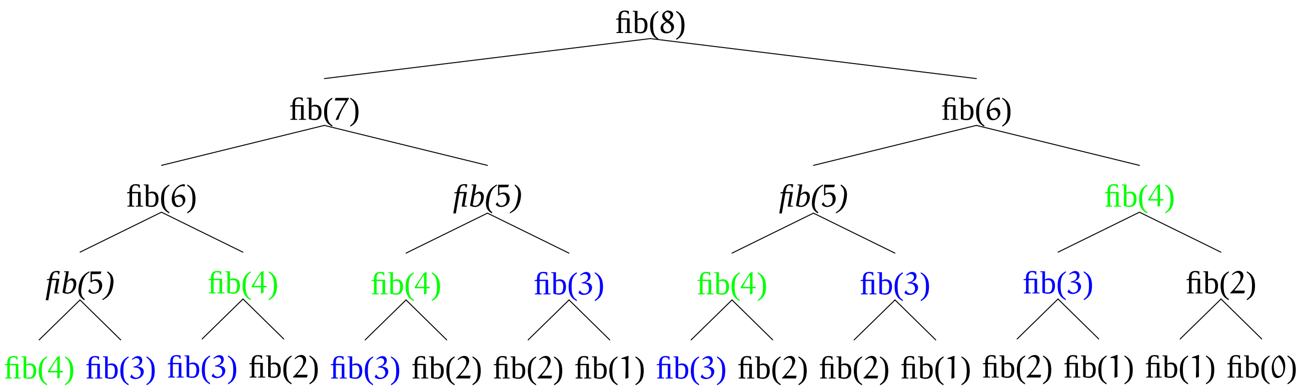


Figure 6.1: Recursive calls during computation of Fibonacci number

A single recursive call to fib(n) results in one recursive call to fib$(n - 1)$, two recursive calls to fib$(n - 2)$, three recursive calls to fib$(n - 3)$, five recursive calls to fib$(n - 4)$ and, in general, F_{k-1} recursive calls to fib$(n - k)$ For each call, we're recomputing the same fibonacci number from scratch.

We can avoid this unnecessary repetitions by writing down the results of recursive calls and looking them up again if we need them later. This process is called *memoization*. Here is the algorithm with memoization.

```
MEMOFIB($n$)
if ($n < 2$)
    then return $n$
if ($F[n]$ is undefined)
    then $F[n] \leftarrow$ MEMOFIB($n - 1$) + MEMOFIB($n - 2$)
return $F[n]$
```

If we trace through the recursive calls to MEMOFIB, we find that array $F[]$ gets filled from bottom up. I.e., first $F[2]$, then $F[3]$, and so on, up to $F[n]$. We can replace recursion with a simple for-loop that just fills up the array $F[]$ in that order.

This gives us our first explicit *dynamic programming* algorithm.

```
ITERFIB($n$)
F[0] $\leftarrow$ 0
F[1] $\leftarrow$ 1
for $i \leftarrow 2$ to $n$
do
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$
```

This algorithm clearly takes only $O(n)$ time to compute F_n. By contrast, the original recursive algorithm takes $\Theta(\phi^n)$, $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$. ITERFIB achieves an exponential speedup over the original recursive algorithm.

6.2 Dynamic Programming

Dynamic programming is essentially recursion without repetition. Developing a dynamic programming algorithm generally involves two separate steps:

- **Formulate problem recursively**. Write down a formula for the whole problem as a simple combination of answers to smaller subproblems.
- **Build solution to recurrence from bottom up**. Write an algorithm that starts with base cases and works its way up to the final solution.

Dynamic programming algorithms need to store the results of intermediate subproblems. This is often *but not always* done with some kind of table. We will now cover a number of examples of problems in which the solution is based on dynamic programming strategy.

6.3 Edit Distance

The words “computer” and “commuter” are very similar, and a change of just one letter, p-¿m, will change the first word into the second. The word “sport” can be changed into “sort” by the deletion of the ‘p’, or equivalently, ‘sort’ can be changed into ‘sport’ by the insertion of ‘p’. The edit distance of two strings, s1 and s2, is defined as the minimum number of point mutations required to change s1 into s2, where a point mutation is one of:

- change a letter,
- insert a letter or

- delete a letter

For example, the edit distance between *FOOD* and *MONEY* is at most four:

$$\begin{aligned} \mathsf{\underline{F}OOD} \longrightarrow \mathsf{MO\underline{O}D} \longrightarrow \mathsf{MON}_{\curlywedge}\mathsf{D} \\ \longrightarrow \mathsf{MONE\underline{D}} \longrightarrow \mathsf{MONEY} \end{aligned}$$

6.3.1 Edit Distance: Applications

There are numerous applications of the Edit Distance algorithm. Here are some examples:

Spelling Correction

If a text contains a word that is not in the dictionary, a ‘close’ word, i.e. one with a small edit distance, may be suggested as a correction. Most word processing applications, such as Microsoft Word, have spelling checking and correction facility. When Word, for example, finds an incorrectly spelled word, it makes suggestions of possible replacements.

Plagiarism Detection

If someone copies, say, a C program and makes a few changes here and there, for example, change variable names, add a comment of two, the edit distance between the source and copy may be small. The edit distance provides an indication of similarity that might be too close in some situations.

Computational Molecular Biology DNA is a polymer. The monomer units of DNA are nucleotides, and the polymer is known as a “polynucleotide.” Each nucleotide consists of a 5-carbon sugar (deoxyribose), a nitrogen containing base attached to the sugar, and a phosphate group. There are four different types of nucleotides found in DNA, differing only in the nitrogenous base. The four nucleotides are given one letter abbreviations as shorthand for the four bases.

- A-adenine
- G-guanine
- C-cytosine
- T-thymine

Double-helix of DNA molecule with nucleotides Figure of Double-helix of DNA molecule with nucleotides goes here

The edit distance like algorithms are used to compute a distance between DNA sequences (strings over A,C,G,T, or protein sequences (over an alphabet of 20 amino acids), for various purposes, e.g.:

- to find genes or proteins that may have shared functions or properties
- to infer family relationships and evolutionary trees over different organisms.

Speech Recognition

Algorithms similar to those for the edit-distance problem are used in some speech recognition systems. Find a close match between a new utterance and one in a library of classified utterances.

6.3.2 Edit Distance Algorithm

A better way to display this editing process is to place the words above the other:

$$\begin{array}{cccccc} S & D & I & M & D & M \\ \hline \mathrm{M} & \mathrm{A} & \_ & \mathrm{T} & \mathrm{H} & \mathrm{S} \\ \mathrm{A} & \_ & \mathrm{R} & \mathrm{T} & \_ & \mathrm{S} \end{array}$$

The first word has a gap for every insertion (I) and the second word has a gap for every deletion (D). Columns with two different characters correspond to substitutions (S). Matches (M) do not count. The *Edit transcript* is defined as a string over the alphabet M, S, I, D that describes a transformation of one string into another. For example

$$\begin{array}{ccccccc} S & D & I & M & D & M & \\ 1+ & 1+ & 1+ & 0+ & 1+ & 0+ & = 4 \end{array}$$

In general, it is not easy to determine the optimal edit distance. For example, the distance between *ALGORITHM* and *ALTRUISTIC* is at most 6.

$$\begin{array}{ccccccccccc} \mathrm{A} & \mathrm{L} & \mathrm{G} & \mathrm{O} & \mathrm{R} & \_ & \mathrm{I} & \_ & \mathrm{T} & \mathrm{H} & \mathrm{M} \\ \mathrm{A} & \mathrm{L} & \_ & \mathrm{T} & \mathrm{R} & \mathrm{U} & \mathrm{I} & \mathrm{S} & \mathrm{T} & \mathrm{I} & \mathrm{C} \end{array}$$

Is this optimal?

6.3.3 Edit Distance: Dynamic Programming Algorithm

Suppose we have an m-character string A and an n-character string B. Define $E(i, j)$ to be the edit distance between the first i characters of A and the first j characters of B. For example,

$$\begin{array}{l} \overbrace{A\quad L\quad G\quad O\quad R}^{i}\quad I\quad T\quad H\quad M \\ \underbrace{A\quad L\quad \_\quad T\quad R}_{j}\quad U\quad I\quad S\quad T\quad I\quad C \end{array}$$

The edit distance between entire strings A and B is $E(m, n)$. The gap representation for the edit sequences has a crucial "*optimal substructure*" property. If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining substrings. The edit distance is 6 for the following two words.

$$\begin{array}{ccccccccccc} A & L & G & O & R & \_ & I & \_ & T & H & M \\ A & L & \_ & T & R & U & I & S & T & I & C \end{array}$$

If we remove the last column, the edit distance reduces to 5.

$$\begin{array}{cccccccccc} A & L & G & O & R & \_ & I & \_ & T & H \\ A & L & \_ & T & R & U & I & S & T & I \end{array}$$

We can use the optimal substructure property to devise a recursive formulation of the edit distance problem. There are a couple of obvious base cases:

- The only way to convert an empty string into a string of j characters is by doing j insertions. Thus

$$E(0, j) = j$$

- The only way to convert a string of i characters into the empty string is with i deletions:

$$E(i, 0) = i$$

There are four possibilities for the last column in the shortest possible edit sequence:

Deletion: Last entry in bottom row is empty.

$$\begin{array}{l} \overbrace{A\ \ L\ \ G}^{i=3}\ \ O\ \ R\ \ I\ \ T\ \ H\ \ M \\ \underbrace{A\ \ L\ \ \_}_{j=2}\ \ T\ \ R\ \ U\ \ I\ \ S\ \ T\ \ I\ \ C \end{array}$$

In this case

$$E(i, j) = E(i - 1, j) + 1$$

Insertion: The last entry in the top row is empty.

$$\begin{array}{l} \overbrace{A\ \ L\ \ G\ \ O\ \ R\ \ \_}^{i=5}\ \ I\ \ T\ \ H\ \ M \\ \underbrace{A\ \ L\ \ \_\ \ T\ \ R\ \ U}_{j=5}\ \ I\ \ S\ \ T\ \ I\ \ C \end{array}$$

In this case

$$E(i, j) = E(i, j - 1) + 1$$

Substitution: Both rows have characters in the last column.

$$\begin{array}{l} \overbrace{A\ \ L\ \ G\ \ O}^{i=4}\ \ R\ \ I\ \ T\ \ H\ \ M \\ \underbrace{A\ \ L\ \ \_\ \ T}_{j=3}\ \ R\ \ U\ \ I\ \ S\ \ T\ \ I\ \ C \end{array}$$

If the characters are different, then

$$E(i,j) = E(i-1,j-1)+1$$

$$\begin{array}{l} \overbrace{A\ \ L\ \ G\ \ O\ \ R}^{i=5}\ \ I\ \ T\ \ H\ \ M \\ \underbrace{A\ \ L\ \ \_\ \ T\ \ R}_{j=4}\ \ U\ \ I\ \ S\ \ T\ \ I\ \ C \end{array}$$

If characters are same, no substitution is needed:

$$E(i,j) = E(i-1,j-1)$$

Thus the edit distance $E(i,j)$ is the smallest of the four possibilities:

$$E(i,j) = \min \left(\begin{array}{ll} E(i-1,j)+1 & \\ E(i,j-1)+1 & \\ E(i-1,j-1)+1 & \text{if } A[i] \neq B[j] \\ E(i-1,j-1) & \text{if } A[i] = B[j] \end{array} \right)$$

Consider the example of edit between the words “ARTS” and “MATHS”:

A	R	T	S	
M	A	T	H	S

The edit distance would be in $E(4,5)$. If we recursion to compute, we will have

$$E(4,5) = \min \left(\begin{array}{ll} E(3,5)+1 & \\ E(4,4)+1 & \\ E(3,4)+1 & \text{if } A[4] \neq B[5] \\ E(3,4) & \text{if } A[4] = B[5] \end{array} \right)$$

Recursion clearly leads to the same repetitive call pattern that we saw in Fibonnaci sequence. To avoid this, we will use the DP approach. We will build the solution bottom-up. We will use the base case $E(0,j)$ to fill first row and the base case $E(i,0)$ to fill first column. We will fill the remaining E matrix row by row.

		A	R	T	S
	0	→1	→2	→3	→4
M					
A					
T					
H					
S					

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1				
A	↓ 2				
T	↓ 3				
H	↓ 4				
S	↓ 5				

Table 6.1: First row and first column entries using the base cases

We can now fill the second row. The table not only shows the values of the cells $E[i, j]$ but also arrows that indicate how it was computed using values in $E[i-1, j]$, $E[i, j-1]$ and $E[i-1, j-1]$. Thus, if a cell $E[i, j]$ has a down arrow from $E[i-1, j]$ then the minimum was found using $E[i-1, j]$. For a right arrow, the minimum was found using $E[i, j-1]$. For a diagonal down right arrow, the minimum was found using $E[i-1, j-1]$. There are certain cells that have two arrows pointed to it. In such a case, the minimum could be obtained from the diagonal $E[i-1, j-1]$ and either of $E[i-1, j]$ and $E[i, j-1]$. We will use these arrows later to determine the edit script.

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1			
A	↓ 2				
T	↓ 3				
H	↓ 4				
S	↓ 5				

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2		
A	↓ 2				
T	↓ 3				
H	↓ 4				
S	↓ 5				

Table 6.2: Computing $E[1, 1]$ and $E[1, 2]$

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	
A	↓ 2				
T	↓ 3				
H	↓ 4				
S	↓ 5				

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2				
T	↓ 3				
H	↓ 4				
S	↓ 5				

Table 6.3: Computing $E[1, 3]$ and $E[1, 4]$

An edit script can be extracted by following a unique path from $E[0, 0]$ to $E[4, 5]$. There are three possible paths in the current example. Let us follow these paths and compute the edit script. In an actual implementation of the dynamic programming version of the edit distance algorithm, the arrows would be recorded using an appropriate data structure. For example, each cell in the matrix could be a record with fields for the value (numeric) and flags for the three incoming arrows.

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘↓ 3	↘↓ 3	↘ 3
S	↓ 5	↓ 4	↘↓ 4	↘↓ 4	↘ 3

Table 6.4: The final table with all $E[i, j]$ entries computed

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘↓ 3	↘↓ 3	↘ 3
S	↓ 5	↓ 4	↘↓ 4	↘↓ 4	↘ 3

Solution path 1:

1+	0+	1+	1+	0	= 3
D	**M**	**S**	**S**	**M**	
M	A	T	H	S	
_	A	R	T	S	

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘↓ 3	↘↓ 3	↘ 3
S	↓ 5	↓ 4	↘↓ 4	↘↓ 4	↘ 3

Table 6.5: Possible edit scripts. The red arrows from E[0, 0] to E[4, 5] show the paths that can be followed to extract edit scripts.

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘↓ 3	↘↓ 3	↘ 3
S	↓ 5	↓ 4	↘↓ 4	↘↓ 4	↘ 3

Solution path 2:

1+	1+	0+	1+	0	= 3
S	**S**	**M**	**D**	**M**	
M	A	T	H	S	
A	R	T	_	S	

		A	R	T	S
	0	→1	→2	→3	→4
M	↓ 1	↘ 1	↘ →2	↘ →3	↘ →4
A	↓ 2	↘ 1	↘ →2	↘ →3	↘ →4
T	↓ 3	↓ 2	↘ 2	↘ 2	→3
H	↓ 4	↓ 3	↘↓ 3	↘↓ 3	↘ 3
S	↓ 5	↓ 4	↘↓ 4	↘↓ 4	↘ 3

Solution path 3:

1+	0+	1+	0+	1+	0	= 3
D	**M**	**I**	**M**	**D**	**M**	
M	A	_	T	H	S	
_	A	R	T	_	S	

6.3.4 Analysis of DP Edit Distance

There are $\Theta(n^2)$ entries in the matrix. Each entry $E(i,j)$ takes $\Theta(1)$ time to compute. The total running time is $\Theta(n^2)$.

6.4 Chain Matrix Multiply

Suppose we wish to multiply a series of matrices:

$$A_1 A_2 \ldots A_n$$

In what order should the multiplication be done? A $p \times q$ matrix A can be multiplied with a $q \times r$ matrix B. The result will be a $p \times r$ matrix C. In particular, for $1 \leq i \leq p$ and $1 \leq j \leq r$,

$$C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j]$$

There are $(p \cdot r)$ total entries in C and each takes $O(q)$ to compute.

$$C[i,j] = \sum_{k=1}^{q} A[i,k]B[k,j]$$

Thus the total number of multiplications is $p \cdot q \cdot r$. Consider the case of 3 matrices: A_1 is 5×4, A_2 is 4×6 and A_3 is 6×2 The multiplication can be carried out either as $((A_1A_2)A_3)$ or $(A_1(A_2A_3))$. The cost of the two is

$$\begin{aligned} ((A_1A_2)A_3) &= (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180 \\ (A_1(A_2A_3)) &= (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) \ = 88 \end{aligned}$$

There is considerable savings achieved even for this simple example. In general, however, in what order should we multiply a series of matrices $A_1A_2\ldots A_n$. Matrix multiplication is an associative but not commutative operation. We are free to add parenthesis the above multiplication but the order of matrices can not be changed. The *Chain Matrix Multiplication Problem* is stated as follows:

> Given a sequence $A_1, A_2, \ldots, A_n$ and dimensions $p_0, p_1, \ldots, p_n$ where A_i is of dimension $p_{i-1} \times p_i$, determine the order of multiplication that minimizes the number of operations.

We could write a procedure that tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is very large. If there are n items, there are $n-1$ ways in which outer most pair of parentheses can placed.

$$\begin{array}{rl} & (A_1)(A_2A_3A_4\ldots A_n) \\ \text{or} & (A_1A_2)(A_3A_4\ldots A_n) \\ \text{or} & (A_1A_2A_3)(A_4\ldots A_n) \\ \ldots & \ldots \\ \text{or} & (A_1A_2A_3A_4\ldots A_{n-1})(A_n) \end{array}$$

Once we split just after the k^{th} matrix, we create two sublists to be parethesized, one with k and other with $n-k$ matrices.

$$(A_1A_2\ldots A_k)\ (A_{k+1}\ldots A_n)$$

We could consider all the ways of parenthesizing these two. Since these are independent choices, if there are L ways of parenthesizing the left sublist and R ways to parenthesize the right sublist, then the total is $L \cdot R$. This suggests the following recurrence for $P(n)$, the number of different ways of parenthesizing n items:

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

This is related to a famous function in combinatorics called the *Catalan numbers*. Catalan numbers are related the number of different binary trees on n nodes. Catalan number is given by the formula:

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

In particular, $P(n) = C(n-1)\ C(n) \in \Omega(4^n/n^{3/2})$ The dominating term is the exponential 4^n thus $P(n)$ will grow large very quickly. So this approach is not practical.

6.4.1 Chain Matrix Multiplication-Dynamic Programming Formulation

The dynamic programming solution involves breaking up the problem into subproblems whose solutions can be combined to solve the global problem. Let $A_{i..j}$ be the result of multiplying matrices i through j. It is easy to see that $A_{i..j}$ is a $p_{i-1} \times p_j$ matrix.

$$\underset{4\times 5}{A_3}\quad \underset{5\times 2}{A_4}\quad \underset{2\times 8}{A_5}\quad \underset{8\times 7}{A_6} = \underset{4\times 7}{A_{3..6}}$$

At the highest level of parenthesization, we multiply two matrices

$$A_{1..n} = A_{1..k} \cdot A_{k+1..n} \quad 1 \leq k \leq n-1.$$

The question now is: what is the optimum value of k for the split and how do we parenthesis the sub-chains $A_{1..k}$ and $A_{k+1..n}$. We can not use divide and conquer because we do not know what is the optimum k. We will have to consider all possible values of k and take the best of them. We will apply this strategy to solve the subproblems optimally.

We will store the solutions to the subproblem in a table and build the table bottom-up (why)? For $1 \leq i \leq j \leq n$, let $m[i,j]$ denote the minimum number of multiplications needed to compute $A_{i..j}$. The optimum can be described by the following recursive formulation.

- If $i = j$, there is only one matrix and thus $m[i,i] = 0$ (the diagonal entries).
- If $i < j$, the we are asking for the product $A_{i..j}$.
- This can be split by considering each k, $i \leq k < j$, as $A_{i..k}$ times $A_{k+1..j}$.

The optimum time to compute $A_{i..k}$ is $m[i,k]$ and optimum time for $A_{k+1..j}$ is in $m[k+1,j]$. Since $A_{i..k}$ is a $p_{i-1} \times p_k$ matrix and $A_{k+1..j}$ is $p_k \times p_j$ matrix, the time to multiply them is $p_{i-1} \times p_k \times p_j$. This suggests the following recursive rule:

$$\begin{aligned} m[i,i] &= 0 \\ m[i,j] &= \min_{i \leq k < j} (m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) \end{aligned}$$

We do not want to calculate m entries recursively. So how should we proceed? We will fill m along the diagonals. Here is how. Set all $m[i,i] = 0$ using the base condition. Compute cost for multiplication of a sequence of 2 matrices. These are $m[1,2], m[2,3], m[3,4], \ldots, m[n-1,n]$. $m[1,2]$, for example is

$$m[1,2] = m[1,1] + m[2,2] + p_0 \cdot p_1 \cdot p_2$$

For example, for m for product of 5 matrices at this stage would be:

$m[1,1]$	$\leftarrow m[1,2]$ $\downarrow$			
	$m[2,2]$	$\leftarrow m[2,3]$ $\downarrow$		
		$m[3,3]$	$\leftarrow m[3,4]$ $\downarrow$	
			$m[4,4]$	$\leftarrow m[4,5]$ $\downarrow$
				$m[5,5]$

Next, we compute cost of multiplication for sequences of three matrices. These are $m[1,3], m[2,4], m[3,5], \ldots, m[n-2,n]$. $m[1,3]$, for example is

$$m[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 \\ m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 \end{cases}$$

We repeat the process for sequences of four, five and higher number of matrices. The final result will end up in $m[1,n]$.

Example: Let us go through an example. We want to find the optimal multiplication order for

$$\underset{(5\times4)}{A_1} \cdot \underset{(4\times6)}{A_2} \cdot \underset{(6\times2)}{A_3} \cdot \underset{(2\times7)}{A_4} \cdot \underset{(7\times3)}{A_5}$$

We will compute the entries of the m matrix starting with the base condition. We first fill that main diagonal:

0				
	0			
		0		
			0	
				0

Next, we compute the entries in the first super diagonal, i.e., the diagonal above the main diagonal:

$$\begin{aligned} m[1,2] &= m[1,1] + m[2,2] + p_0 \cdot p_1 \cdot p_2 = 0 + 0 + 5 \cdot 4 \cdot 6 = 120 \\ m[2,3] &= m[2,2] + m[3,3] + p_1 \cdot p_2 \cdot p_3 = 0 + 0 + 4 \cdot 6 \cdot 2 = 48 \\ m[3,4] &= m[3,3] + m[4,4] + p_2 \cdot p_3 \cdot p_4 = 0 + 0 + 6 \cdot 2 \cdot 7 = 84 \\ m[4,5] &= m[4,4] + m[5,5] + p_3 \cdot p_4 \cdot p_5 = 0 + 0 + 2 \cdot 7 \cdot 3 = 42 \end{aligned}$$

The matrix m now looks as follows:

0	120			
	0	48		
		0	84	
			0	42
				0

We now proceed to the second super diagonal. This time, however, we will need to try two possible values for k. For example, there are two possible splits for computing $m[1,3]$; we will choose the split that yields the minimum:

$$\begin{aligned} m[1,3] &= m[1,1] + m[2,3] + p_0 \cdot p_1 \cdot p_3 == 0 + 48 + 5 \cdot 4 \cdot 2 = 88 \\ m[1,3] &= m[1,2] + m[3,3] + p_0 \cdot p_2 \cdot p_3 = 120 + 0 + 5 \cdot 6 \cdot 2 = 180 \\ &\text{the minimum } m[1,3] = 88 \text{ occurs with } k = 1 \end{aligned}$$

Similarly, for $m[2,4]$ and $m[3,5]$:

$$m[2,4] = m[2,2] + m[3,4] + p_1 \cdot p_2 \cdot p_4 = 0 + 84 + 4 \cdot 6 \cdot 7 = 252$$
$$m[2,4] = m[2,3] + m[4,4] + p_1 \cdot p_3 \cdot p_4 = 48 + 0 + 4 \cdot 2 \cdot 7 = 104$$
$$\text{minimum } m[2,4] = 104 \text{ at } k = 3$$

$$m[3,5] = m[3,3] + m[4,5] + p_2 \cdot p_3 \cdot p_5 = 0 + 42 + 6 \cdot 2 \cdot 3 = 78$$
$$m[3,5] = m[3,4] + m[5,5] + p_2 \cdot p_4 \cdot p_5 = 84 + 0 + 6 \cdot 7 \cdot 3 = 210$$
$$\text{minimum } m[3,5] = 78 \text{ at } k = 3$$

With the second super diagonal computed, the m matrix looks as follow:

0	120	88		
	0	48	104	
		0	84	78
			0	42
				0

We repeat the process for the remaining diagonals. However, the number of possible splits (values of k) increases:

$$m[1,4] = m[1,1] + m[2,4] + p_0 \cdot p_1 \cdot p_4 = 0 + 104 + 5 \cdot 4 \cdot 7 = 244$$
$$m[1,4] = m[1,2] + m[3,4] + p_0 \cdot p_2 \cdot p_4 = 120 + 84 + 5 \cdot 6 \cdot 7 = 414$$
$$m[1,4] = m[1,3] + m[4,4] + p_0 \cdot p_3 \cdot p_4 = 88 + 0 + 5 \cdot 2 \cdot 7 = 158$$
$$\text{minimum } m[1,4] = 158 \text{ at } k = 3$$

$$m[2,5] = m[2,2] + m[3,5] + p_1 \cdot p_2 \cdot p_5 = 0 + 78 + 4 \cdot 6 \cdot 3 = 150$$
$$m[2,5] = m[2,3] + m[4,5] + p_1 \cdot p_3 \cdot p_5 = 48 + 42 + 4 \cdot 2 \cdot 3 = 114$$
$$m[2,5] = m[2,4] + m[5,5] + p_1 \cdot p_4 \cdot p_5 = 104 + 0 + 4 \cdot 7 \cdot 3 = 188$$
$$\text{minimum } m[2,5] = 114 \text{ at } k = 3$$

The matrix m at this stage is:

0	120	88	158	
	0	48	104	114
		0	84	78
			0	42
				0

That leaves the $m[1,5]$ which can now be computed:

$$\begin{aligned}
m[1,5] &= m[1,1] + m[2,5] + p_0 \cdot p_1 \cdot p_5 = 0 + 114 + 5 \cdot 4 \cdot 3 = 174 \\
m[1,5] &= m[1,2] + m[3,5] + p_0 \cdot p_2 \cdot p_5 = 120 + 78 + 5 \cdot 6 \cdot 3 = 288 \\
m[1,5] &= m[1,3] + m[4,5] + p_0 \cdot p_3 \cdot p_5 = 88 + 42 + 5 \cdot 2 \cdot 3 = 160 \\
m[1,5] &= m[1,4] + m[5,5] + p_0 \cdot p_4 \cdot p_5 = 158 + 0 + 5 \cdot 7 \cdot 3 = 263 \\
&\text{minimum } m[1,5] = 160 \text{ at } k = 3
\end{aligned}$$

We thus have the final cost matrix.

0	120	88	158	***160***
0	0	48	104	114
0	0	0	84	78
0	0	0	0	42
0	0	0	0	0

Here is the order in which m entries are calculated

0	1	5	8	10
0	0	2	6	9
0	0	0	3	7
0	0	0	0	4
0	0	0	0	0

and the split k values that led to a minimum $m[i,j]$ value

0	1	1	3	3
	0	2	3	3
		0	3	3
			0	4
				0

Based on the computation, the minimum cost for multiplying the five matrices is 160 and the optimal order for multiplication is

$$((A_1(A_2A_3))(A_4A_5))$$

This can be represented as a binary tree

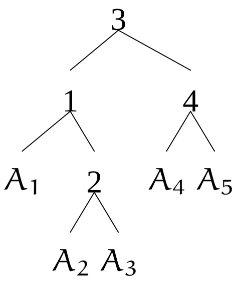


Figure 6.2: Optimum matrix multiplication order for the five matrices example

Here is the dynamic programming based algorithm for computing the minimum cost of chain matrix multiplication.

```
MATRIX-CHAIN$(p, N)$
for $i = 1, N$
do $m[i, i] \leftarrow 0$
for $L = 2, N$
do
   for $i = 1, n - L + 1$
   do $j \leftarrow i + L - 1$
      $m[i, j] \leftarrow \infty$
      for $k = 1, j - 1$
      do $t \leftarrow m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j$
         if $(t < m[i, j])$
            then $m[i, j] \leftarrow t$;  $s[i, j] \leftarrow k$
```

Analysis: There are three nested loops. Each loop executes a maximum n times. Total time is thus $\Theta(n^3)$.

The s matrix stores the values k. The s matrix can be used to extracting the order in which matrices are to be multiplied. Here is the algorithm that caries out the matrix multiplication to compute $A_{i..j}$:

```
MULTIPLY$(i, j)$
if $(i = j)$
  then return $A[i]$
  else $k \leftarrow s[i, j]$
       $X \leftarrow$ MULTIPLY$(i, k)$
       $Y \leftarrow$ MULTIPLY$(k + 1, j)$
       return $X \cdot Y$
```

6.5 0/1 Knapsack Problem

A thief goes into a jewelry store to steal jewelry items. He has a knapsack (a bag) that he would like to fill up. The bag has a limit on the total weight of the objects placed in it. If the total weight exceeds the limit, the bag would tear open. The value of of the jewelry items varies for cheap to expensive. The thief's goal is to put items in the bag such that the value of the items is maximized and the weight of the items does not exceed the weight limit of the bag. Another limitation is that an item can either be put in the bag or not - fractional items are not allowed. The problem is: what jewelry should the thief choose that satisfy the constraints?

Formally, the problem can be stated as follows: Given a knapsack with maximum capacity W, and a set S consisting of n items Each item i has some weight w_i and value value v_i (all w_i , v_i and W are integer values) How to pack the knapsack to achieve maximum total value of packed items? For example, consider the following scenario:

Item i	Weight w_i	Value v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Figure 6.3: Knapsack can hold $W = 20$

The knapsack problem belongs to the domain of optimization problems. Mathematically, the problem is

$$\text{maximize} \sum_{i \in T} v_i$$

$$\text{subject to} \sum_{i \in T} w_i \leq W$$

The problem is called a *“0-1”* problem, because each item must be entirely accepted or rejected. How do we solve the problem. We could try the brute-force solution:

- Since there are n items, there are 2^n possible combinations of the items (an item either chosen or not).
- We go through all combinations and find the one with the most total value and with total weight less or equal to W

Clearly, the running time of such a brute-force algorithm will be $O(2^n)$. Can we do better? The answer is “yes”, with an algorithm based on dynamic programming Let us recap the steps in the dynamic programming strategy:

1. **Simple Subproblems**: We should be able to break the original problem to smaller subproblems that have the same structure
2. **Principle of Optimality**: Recursively define the value of an optimal solution. Express the solution of the original problem in terms of optimal solutions for smaller problems.
3. **Bottom-up computation**: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.
4. **Construction of optimal solution**: Construct an optimal solution from computed information.

Let us try this: If items are labelled $1, 2, \ldots, n$, then a subproblem would be to find an optimal solution for

$$S_k = \text{items labelled } 1, 2, \ldots, k$$

This is a valid subproblem definition. The question is: can we describe the final solution S_n in terms of subproblems S_k? Unfortunately, we cannot do that. Here is why. Consider the optimal solution if we can choose items 1 through 4 only.

Solution S_4

- Items chosen are $1, 2, 3, 4$
- Total weight: $2 + 3 + 4 + 5 = 14$
- Total value: $3 + 4 + 5 + 8 = 20$

Item	w_i	v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Now consider the optimal solution when items 1 through 5 are available.

Solution S_5

- Items chosen are $1, 3, 4, 5$
- Total weight: $2 + 4 + 5 + 9 = 20$
- Total value: $3 + 5 + 8 + 10 = 26$

  S_4 is not part of S_5!!

Item	w_i	v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

The solution for S_4 ***is not part of the solution for*** S_5. So our definition of a subproblem is flawed and we need another one.

6.5.1 0/1 Knapsack Problem: Dynamic Programming Approach

For each $i \leq n$ and each $w \leq W$, solve the knapsack problem for the first i objects when the capacity is w. Why will this work? Because solutions to larger subproblems can be built up easily from solutions to smaller ones. We construct a matrix $V[0 \ldots n, 0 \ldots W]$. For $1 \leq i \leq n$, and $0 \leq j \leq W$, $V[i, j]$ will store the maximum value of any set of objects $\{1, 2, \ldots, i\}$ that can fit into a knapsack of weight j. $V[n, W]$ will contain the maximum value of all n objects that can fit into the entire knapsack of weight W.

To compute entries of V we will imply an inductive approach. As a basis, $V[0, j] = 0$ for $0 \leq j \leq W$ since if we have no items then we have no value. We consider two cases:

Leave object i**:** If we choose to not take object i, then the optimal value will come about by considering how to fill a knapsack of size j with the remaining objects $\{1, 2, \ldots, i-1\}$. This is just $V[i-1, j]$.

Take object i**:** If we take object i, then we gain a value of v_i. But we use up w_i of our capacity. With the remaining $j - w_i$ capacity in the knapsack, we can fill it in the best possible way with objects $\{1, 2, \ldots, i-1\}$. This is $v_i + V[i-1, j-w_i]$. This is only possible if $w_i \leq j$.

This leads to the following recursive formulation:

$$V[i, j] = -\infty \qquad \text{if } j < 0$$

$$V[0, j] = 0 \qquad \text{if } j \geq 0$$

$$V[i, j] = \begin{cases} V[i-1, j] & \text{if } w_i > j \\ \max\{V[i-1, j],\ \ v_i + V[i-1, j-w_i]\} & \text{if } w_i \leq j \end{cases}$$

A naive evaluation of this recursive definition is exponential. So, as usual, we avoid re-computation by making a table.

Example: The maximum weight the knapsack can hold is W is 11. There are five items to choose from. Their weights and values are presented in the following table:

Weight limit (j):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$												
$w_2 = 2$ $v_2 = 6$												
$w_3 = 5$ $v_3 = 18$												
$w_4 = 6$ $v_4 = 22$												
$w_5 = 7$ $v_5 = 28$												

The $[i, j]$ entry here will be $V[i, j]$, the best value obtainable using the first i rows of items if the maximum capacity were j. We begin by initializating and first row.

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0											
$w_3 = 5$ $v_3 = 18$	0											
$w_4 = 6$ $v_4 = 22$	0											
$w_5 = 7$ $v_5 = 28$	0											

Recall that we take $V[i, j]$ to be 0 if either i or j is ≤ 0. We then proceed to fill in top-down, left-to-right always using

$$V[i, j] = \max\big\{V[i-1, j],\ \ v_i + V[i-1, j-w_i]\big\}$$

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0											
$w_4 = 6$ $v_4 = 22$	0											
$w_5 = 7$ $v_5 = 28$	0											

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7	18	19	*24*	25	25	25	25
$w_4 = 6$ $v_4 = 22$	0											
$w_5 = 7$ $v_5 = 28$	0											

As an illustration, the value of $V[3, 7]$ was computed as follows:

$$\begin{aligned} V[3,7] =& \max\big\{V[3-1,7],\ \ v_3 + V[3-1, 7-w_3]\big\} \\ =& \max\big\{V[2,7],\ \ 18 + V[2, 7-5]\big\} \\ =& \max\big\{7,\ \ 18+6\big\} \\ =& 24 \end{aligned}$$

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6$ $v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7$ $v_5 = 28$	0											

Finally, we have

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5$ $v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6$ $v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7$ $v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	*40*

The maximum value of items in the knapsack is 40, the bottom-right entry). The dynamic programming approach can now be coded as the following algorithm:

```
KNAPSACK($n, W$)
for $w = 0, W$
do $V[0, w] \leftarrow 0$
for $i = 0, n$
do $V[i, 0] \leftarrow 0$
    for $w = 0, W$
    do if $(w_i \leq w$ & $v_i + V[i-1, w-w_i] > V[i-1, w])$
          then $V[i, w] \leftarrow v_i + V[i-1, w-w_i]$
          else  $V[i, w] \leftarrow V[i-1, w]$
```

The time complexity is clearly $O(n \cdot W)$. It must be cautioned that as n and W get large, both time and space complexity become significant.

Constructing the Optimal Solution

The algorithm for computing $V[i, j]$ does not keep record of which subset of items gives the optimal solution. To compute the actual subset, we can add an auxiliary boolean array $keep[i, j]$ which is 1 if we decide to take the i^{th} item and 0 otherwise. We will use all the values $keep[i, j]$ to determine the optimal subset T of items to put in the knapsack as follows:

- If $keep[n, W]$ is 1, then $n \in T$. We can now repeat this argument for $keep[n-1, W-w_n]$.

- If $kee[n, W]$ is 0, the $n \notin T$ and we repeat the argument for $keep[n-1, W]$.

We will add this to the knapsack algorithm:

```
KNAPSACK$(n, W)$
**for** $w = 0, W$
**do** $V[0, w] \leftarrow 0$
**for** $i = 0, n$
**do** $V[i, 0] \leftarrow 0$
    **for** $w = 0, W$
    **do if** $(w_i \leq w \;\&\; v_i + V[i-1, w-w_i] > V[i-1, w])$
          **then** $V[i, w] \leftarrow v_i + V[i-1, w-w_i];\ \mathrm{keep}[i, w] \leftarrow 1$
          **else** $V[i, w] \leftarrow V[i-1, w];\ \mathrm{keep}[i, w] \leftarrow 0$
// output the selected items
$k \leftarrow W$
**for** $i = n$ downto $1$
**do if** $(\mathrm{keep}[i, k] = 1)$
      **then** output $i$
            $k \leftarrow k - w_i$
```

Here is the keep matrix for the example problem.

Weight limit:	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1$ $v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2$ $v_2 = 6$	0	0	1	1	1	1	1	1	1	1	1	1
$w_3 = 5$ $v_3 = 18$	0	0	0	0	0	$\boxed{1}$	1	1	1	1	1	1
$w_4 = 6$ $v_4 = 22$	0	0	0	0	0	0	1	0	1	1	1	$\boxed{1}$
$w_5 = 7$ $v_5 = 28$	0	0	0	0	0	0	0	1	1	1	1	0

When the item selection algorithm is applied, the selected items are 4 and 3. This is indicated by the boxed entries in the table above.

Chapter 7

Greedy Algorithms

An *optimization problem* is one in which you want to find, not just a solution, but the best solution. Search techniques look at many possible solutions. E.g. dynamic programming or backtrack search. A “ greedy algorithm” sometimes works well for optimization problems

A greedy algorithm works in phases. At each phase:

- You take the best you can get right now, without regard for future consequences.
- You hope that by choosing a local optimum at each step, you will end up at a global optimum.

For some problems, greedy approach always gets optimum. For others, greedy finds good, but not always best. If so, it is called a greedy heuristic, or approximation. For still others, greedy approach can do very poorly.

7.1 Example: Counting Money

Suppose you want to count out a certain amount of money, using the fewest possible bills (notes) and coins. A greedy algorithm to do this would be: at each step, take the largest possible note or coin that does not overshoot.

```
while $(N > 0)\{$
    give largest denomination coin $\le N$
    reduce $N$ by value of that coin
$\}$
```

Consider the currency in U.S.A. There are paper notes for one dollar, five dollars, ten dollars, twenty dollars, fifty dollars and hundred dollars. The notes are also called “bills”. The coins are one cent, five cents (called a “nickle”), ten cents (called a “dime”) and twenty five cents (a “quarter”). In Pakistan, the currency notes are five rupees, ten rupees, fifty rupees, hundred rupees, five hundred rupees and thousand rupees. The coins are one rupee and two rupees. Suppose you are asked to give change of \$6.39 (six dollars and thirty nine cents), you can choose:

- a \$5 note
- a \$1 note to make \$6
- a 25 cents coin (quarter), to make \$6.25
- a 10 cents coin (dime), to make \$6.35
- four 1 cents coins, to make \$6.39

Notice how we started with the highest note, \$5, before moving to the next lower denomination.

Formally, the Coin Change problem is: Given k denominations $d_1, d_2, \ldots, d_k$ and given N, find a way of writing

$$N = i_1 d_1 + i_2 d_2 + \cdots + i_k d_k$$

such that

$$i_1 + i_2 + \cdots + i_k \text{ is minimized.}$$

The “size” of problem is k.

The greedy strategy works for the coin change problem but not always. Here is an example where it fails. Suppose, in some (fictional) monetary system, “ krons” come in 1 kron, 7 kron, and 10 kron coins Using a greedy algorithm to count out 15 krons, you would get A 10 kron piece Five 1 kron pieces, for a total of 15 krons This requires six coins. A better solution, however, would be to use two 7 kron pieces and one 1 kron piece This only requires three coins The greedy algorithm results in a solution, but not in an optimal solution

The greedy approach gives us an optimal solution when the coins are all powers of a fixed denomination.

$$N = i_0 D^0 + i_1 D^1 + i_2 D^2 + \cdots + i_k D^k$$

Note that this is N represented in based D. U.S.A coins are multiples of 5: 5 cents, 10 cents and 25 cents.

7.1.1 Making Change: Dynamic Programming Solution

The general coin change problem can be solved using Dynamic Programming. Set up a Table, $C[1..k, 0..N]$ in which the rows denote available denominations, d_i; $(1 \leq i \leq k)$ and columns denote the amount from $0 \ldots N$ units, $(0 \leq j \leq N)$. $C[i, j]$ denotes the minimum number of coins, required to pay an amount j using only coins of denominations 1 to i. C[k, N] is the solution required.

To pay an amount j units, using coins of denominations 1 to i, we have two choices:

1. either chose NOT to use any coins of denomination i,
2. or chose at least one coin of denomination i, and also pay the amount $(j - d_i)$.

To pay $(j - d_i)$ units it takes $C[i, j - d_i]$ coins. Thus,

$$C[i, j] = 1 + C[i, j - d_i]$$

Since we want to minimize the number of coins used,

$$C[i, j] = \min(C[i - 1, j], 1 + C[i, j - d_i])$$

Here is the dynamic programming based algorithm for the coin change problem.

```
COINS($N$)
$d[1..n] = \{1, 4, 6\}$   // (coinage, for example)
for $i = 1$ to $k$
do $c[i, 0] \leftarrow 0$
for $i = 1$ to $k$
do for $j = 1$ to $N$
    do if $(i = 1 \,\&\, j < d[i])$
          then $c[i, j] \leftarrow \infty$
          else if $(i = 1)$
                 then $c[i, j] \leftarrow 1 + c[1, j - d[1]]$
                 else if $(j < d[i])$
                        then $c[i, j] \leftarrow c[i - 1, j]$
                        else $c[i, j] \leftarrow \min(c[i - 1, j], 1 + c[i, j - d[i]])$
return $c[k, N]$
```

7.1.2 Complexity of Coin Change Algorithm

Greedy algorithm (non-optimal) takes $O(k)$ time. Dynamic Programming takes $O(kN)$ time. Note that N can be as large as 2^k so the dynamic programming algorithm is really exponential in k.

7.2 Greedy Algorithm: Huffman Encoding

The Huffman codes provide a method of encoding data efficiently. Normally, when characters are coded using standard codes like ASCII. Each character is represented by a fixed-length codeword of bits, e.g., 8 bits per character. Fixed-length codes are popular because it is very easy to break up a string into its individual characters, and to access individual characters and substrings by direct indexing. However, fixed-length codes may not be he most efficient from the perspective of minimizing the total quantity of data.

Consider the string " abacdaacac". if the string is coded with ASCII codes, the message length would be $10 \times 8 = 80$ bits. We will see shortly that the same string encoded with a variable length Huffman encoding scheme will produce a shorter message.

7.2.1 Huffman Encoding Algorithm

Here is how the Huffman encoding algorithm works. Given a message string, determine the frequency of occurrence (relative probability) of each character in the message. This can be done by parsing the message and counting how many time each character (or symbol) appears. The probability is the number of occurrence of a character divided by the total characters in the message. The frequencies and probabilities for the example string “ abacdaacac” are

character	a	b	c	d
frequency	5	1	3	1
probability	0.5	0.1	0.3	0.1

Next, create binary tree (leaf) node for each symbol (character) that occurs with nonzero frequency Set node weight equal to the frequency of the symbol. Now comes the greedy part: Find two nodes with smallest frequency. Create a new node with these two nodes as children, and with weight equal to the sum of the weights of the two children. Continue until we have a single tree.

Finding two nodes with the smallest frequency can be done efficiently by placing the nodes in a heap-based priority queue. The min-heap is maintained using the frequencies. When a new node is created by combining two nodes, the new node is placed in the priority queue. Here is the Huffman tree building algorithm.

```
HUFFMAN($N, symbol[1..N], freq[1..N]$)
for $i = 1$ to $N$
do $t \leftarrow TreeNode(symbol[i], freq[i])$
    $pq.insert(t, freq[i])$ // priority queue
for $i = 1$ to $N - 1$
do $x = pq.remove(); \; y = pq.remove()$
    $z \leftarrow new$ TreeNode
    $z.left \leftarrow x; \;\; z.right \leftarrow y$
    $z.freq \leftarrow x.freq + y.freq$
    $pq.insert(z, z.freq);$
return $pq.remove();$ // root
```

Figure 7.1 shows the tree built for the example message “abacdaacac”

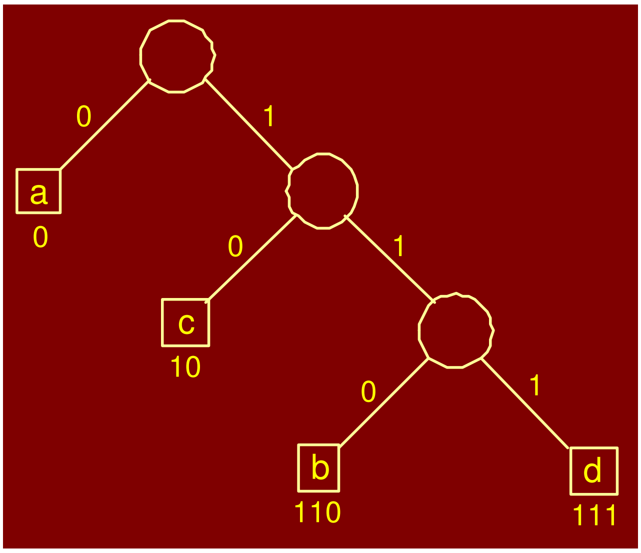


Figure 7.1: Huffman binary tree for the string “abacdaacac”

Prefix Property:

The codewords assigned to characters by the Huffman algorithm have the property that no codeword is a prefix of any other:

character	a	b	c	d
frequency	5	1	3	1
probability	0.5	0.1	0.3	0.1
codeword	0	110	10	111

The prefix property is evident by the fact that codewords are leaves of the binary tree. Decoding a prefix code is simple. We traverse the root to the leaf letting the input 0 or 1 tell us which branch to take.

Expected encoding length:

If a string of n characters over the alphabet $C = \{a, b, c, d\}$ is encoded using 8-bit ASCII, the length of encoded string is $8n$. For example, the string “abacdaacac” will require $8 \times 10 = 80$ bits. The same string encoded with Huffman codes will yield

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

This is just 17 bits, a significant saving!. For a string of n characters over this alphabet, the expected encoded string length is

$$n(0.5 \cdot 1 + 0.1 \cdot 3 + 0.3 \cdot 2 + 0.1 \cdot 3) = 1.7n$$

In general, let $p(x)$ be the probability of occurrence of a character, and let $d_T(x)$ denote the length of the codeword relative to some prefix tree T. The expected number of bits needed to encode a text with n characters is given by

$$B(T) = n \sum_{x \in C} p(x) d_T(x)$$

7.2.2 Huffman Encoding: Correctness

Huffman algorithm uses a greedy approach to generate a prefix code T that minimizes the expected length $B(T)$ of the encoded string. In other words, Huffman algorithm generates an optimum prefix code. The question that remains is that *why is the algorithm correct?*

Recall that the cost of any encoding tree T is

$$B(T) = n \sum_{x \in C} p(x) d_T(x)$$

Our approach to prove the correctness of Huffman Encoding will be to show that any tree that differs from the one constructed by Huffman algorithm can be converted into one that is equal to Huffman's tree without increasing its costs. Note that the binary tree constructed by Huffman algorithm is a full binary tree.

Claim:

Consider two characters x and y with the smallest probabilities. Then there is optimal code tree in which these two characters are siblings at the maximum depth in the tree.

Proof:

Let T be any optimal prefix code tree with two siblings b and c at the maximum depth of the tree. Such a tree is shown in Figure 7.2Assume without loss of generality that

$$p(b) \leq p(c) \qquad \text{and} \qquad p(x) \leq p(y)$$

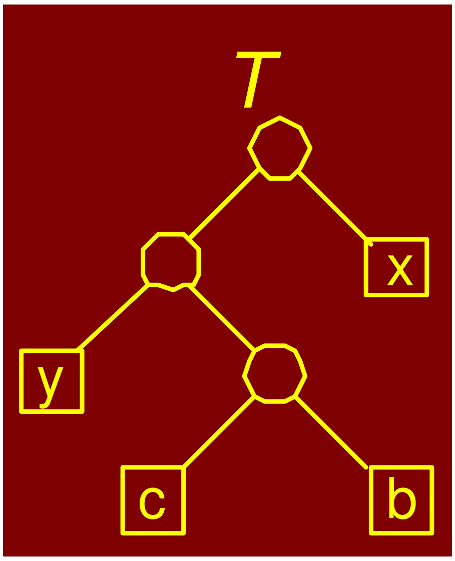


Figure 7.2: Optimal prefix code tree T

Since x and y have the two smallest probabilities (we claimed this), it follows that

$$p(x) \leq p(b) \qquad \text{and} \qquad p(y) \leq p(c)$$

Since b and c are at the deepest level of the tree, we know that

$$d(b) \geq d(x) \qquad \text{and} \qquad d(c) \geq d(y) \qquad \text{(d is the depth)}$$

Thus we have

$$p(b) - p(x) \geq 0$$

and

$$d(b) - d(x) \geq 0$$

Hence their product is non-negative. That is,

$$(p(b) - p(x)) \cdot (d(b) - d(x)) \geq 0$$

Now swap the positions of x and b in the tree

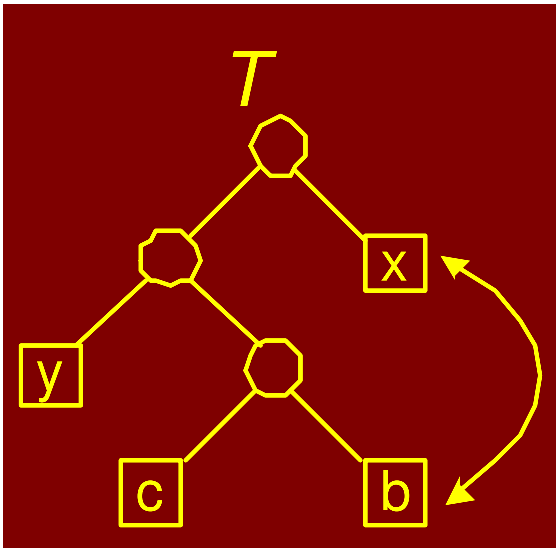


Figure 7.3: Swap x and b in tree prefix tree T

This results in a new tree T'

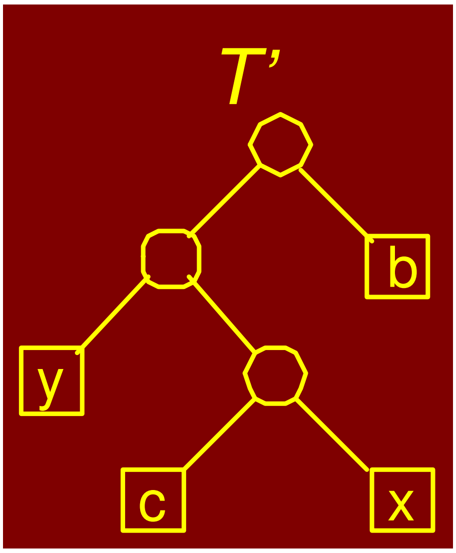


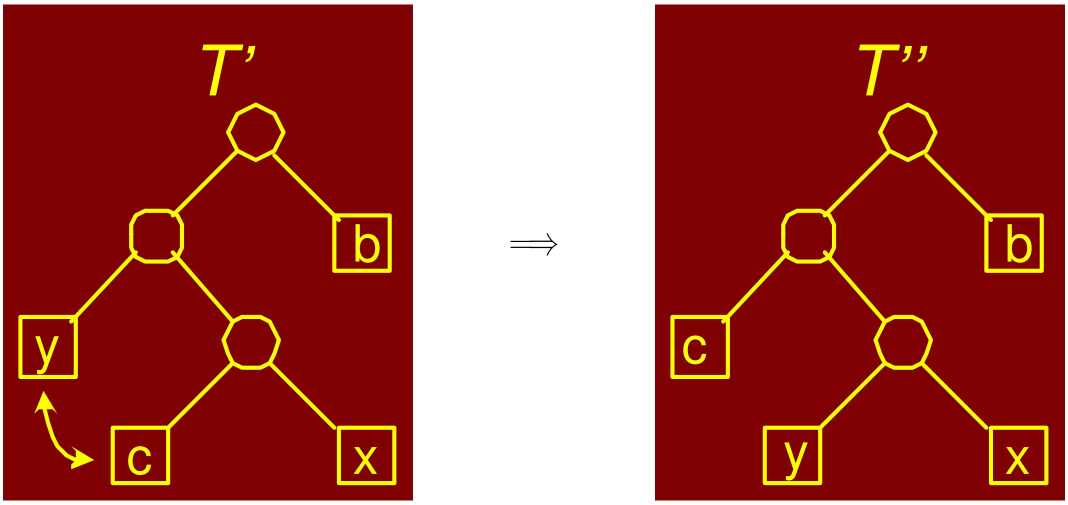
Figure 7.4: Prefix tree T' after x and b are swapped

Let's see how the cost changes. The cost of T' is

$$\begin{aligned} B(T') &= B(T) - p(x)d(x) + p(x)d(b) - p(b)d(b) + p(b)d(x) \\ &= B(T) + p(x)[d(b) - d(x)] - p(b)[d(b) - d(x)] \\ &= B(T) - (p(b) - p(x))(d(b) - d(x)) \\ &\leq B(T) \quad \text{because} \;\; (p(b) - p(x))(d(b) - d(x)) \geq 0 \end{aligned}$$

Thus the cost does not increase, implying that T' is an optimal tree.

By switching y with c we get the tree T''. Using a similar argument, we can show that T'' is also optimal.



The final tree T'' satisfies the claim we made earlier, i.e., consider two characters x and y with the smallest probabilities. Then there is optimal code tree in which these two characters are siblings at the maximum depth in the tree.

The claim we just proved asserts that the first step of Huffman algorithm is the proper one to perform (the greedy step). The complete proof of correctness for Huffman algorithm follows by induction on n.

Claim: Huffman algorithm produces the optimal prefix code tree.

Proof: The proof is by induction on n, the number of characters. For the basis case, $n = 1$, the tree consists of a single leaf node, which is obviously optimal. We want to show it is true with exactly n characters.

Suppose we have exactly n characters. The previous claim states that two characters x and y with the lowest probability will be siblings at the lowest level of the tree. Remove x and y and replace them with a new character z whose probability is $p(z) = p(x) + p(y)$. Thus $n - 1$ characters remain.

Consider any prefix code tree T made with this new set of $n - 1$ characters. We can convert T into prefix code tree T' for the original set of n characters by replacing z with nodes x and y. This is essentially undoing the operation where x and y were removed an replaced by z. The cost of the new tree T' is

$$\begin{aligned}
B(T') &= B(T) - p(z)d(z) + p(x)[d(z) + 1] + p(y)[d(z) + 1] \\
&= B(T) - (p(x) + p(y))d(z) + (p(x) + p(y))[d(z) + 1] \\
&= B(T) + (p(x) + p(y))[d(z) + 1 - d(z)] \\
&= B(T) + p(x) + p(y)
\end{aligned}$$

The cost changes but the change depends in no way on the structure of the tree T (T is for $n - 1$ characters). Therefore, to minimize the cost of the final tree T', we need to build the tree T on $n - 1$ characters optimally. By induction, this is exactly what Huffman algorithm does. Thus the final tree is optimal.

7.3 Activity Selection

The activity scheduling is a simple scheduling problem for which the greedy algorithm approach provides an optimal solution. We are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities that are to be scheduled to use some resource. Each activity a_i must be started at a given start time s_i and ends at a given finish time f_i.

An example is that a number of lectures are to be given in a single lecture hall. The start and end times have be set up in advance. The lectures are to be scheduled. There is only one resource (e.g., lecture hall) Some start and finish times may overlap. Therefore, not all requests can be honored. We say that two activities a_i and a_j are non-interfering if their start-finish intervals do not overlap. I.e, $(s_i, f_i) \cap (s_j, f_j) = \emptyset$. The activity selection problem is to select a maximum-size set of mutually non-interfering activities for use of the resource.

So how do we schedule the largest number of activities on the resource? Intuitively, we do not like long activities Because they occupy the resource and keep us from honoring other requests. This suggests the greedy strategy: Repeatedly select the activity with the smallest duration $(f_i - s_i)$ and schedule it, provided that it does not interfere with any previously scheduled activities. Unfortunately, this turns out to be non-optimal

Here is a simple greedy algorithm that works: Sort the activities by their finish times. Select the activity that finishes first and schedule it. Then, among all activities that do not interfere with this first job, schedule the one that finishes first, and so on.

```
SCHEDULE($a[1..N]$)
    sort $a[1..N]$ by finish times
    $A \leftarrow \{a[1]\}$;   // schedule activity 1 first
    $prev \leftarrow 1$;   // most recently scheduled
    for $i = 2$ to $N$
    do if $(a[i].start \geq a[prev].finish)$
          then $A \leftarrow A \cup a[i]$;  $prev \leftarrow i$
```

Figure 7.5 shows an example of the activity scheduling algorithm. There are eight activities to be scheduled. Each is represented by a rectangle. The width of a rectangle indicates the duration of an activity. The eight activities are sorted by their finish times. The eight rectangles are arranged to show the sorted order. Activity a_1 is scheduled first. Activities a_2 and a_3 interfere with a_1 so they ar not selected. The next to be selected is a_4. Activities a_5 and a_6 interfere with a_4 so are not chosen. The last one to be chosen is a_7. Eventually, only three out of the eight are scheduled.

Timing analysis: Time is dominated by sorting of the activities by finish times. Thus the complexity is $O(N \log N)$.

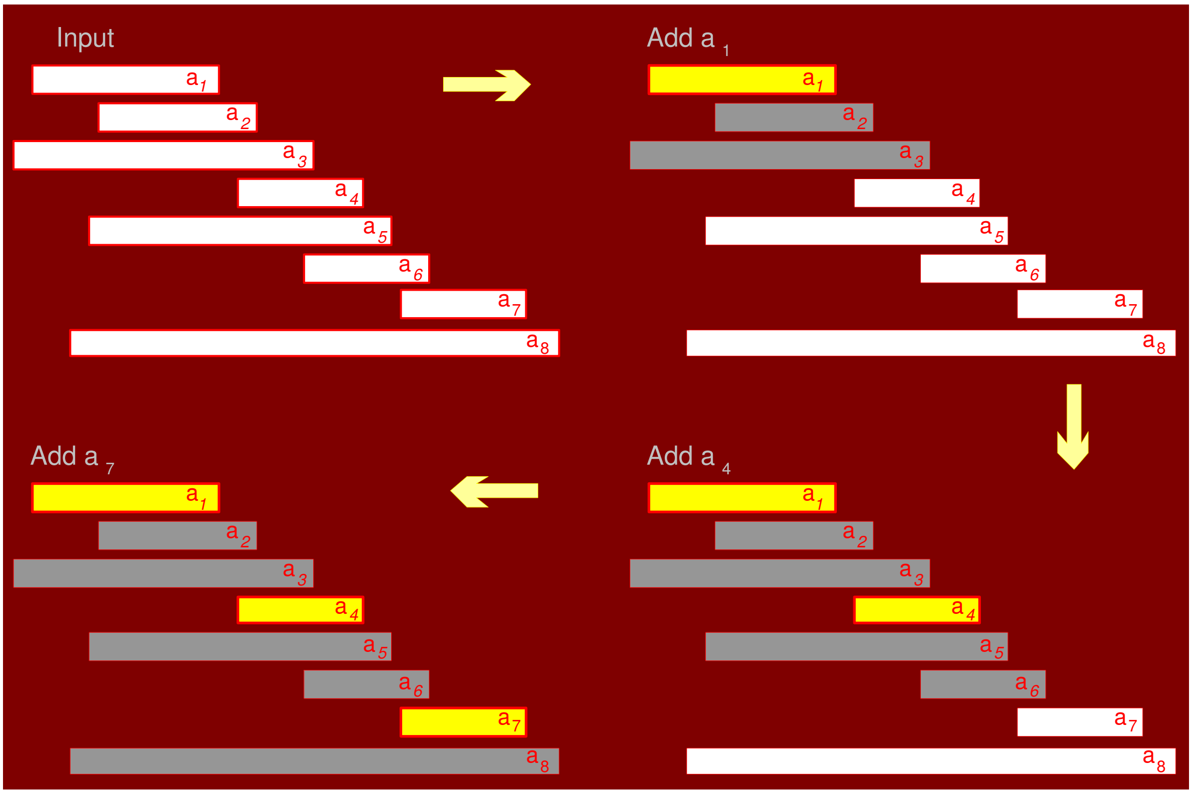


Figure 7.5: Example of greedy activity scheduling algorithm

7.3.1 Correctness of Greedy Activity Selection

Our proof of correctness is based on showing that the first choice made by the algorithm is the best possible. And then using induction to show that the algorithm is globally optimal. The proof structure is noteworthy because many greedy correctness proofs are based on the same idea: Show that any other solution can be converted into the greedy solution without increasing the cost.

Claim:

Let $S = \{a_1, a_2, \ldots, a_n\}$ of n activities, sorted by increasing finish times, that are to be scheduled to use some resource. Then there is an optimal schedule in which activity a_1 is scheduled first.

Proof:

Let A be an optimal schedule. Let x be the activity in A with the smallest finish time. If $x = a_1$ then we are done. Otherwise, we form a new schedule A' by replacing x with activity a_1.

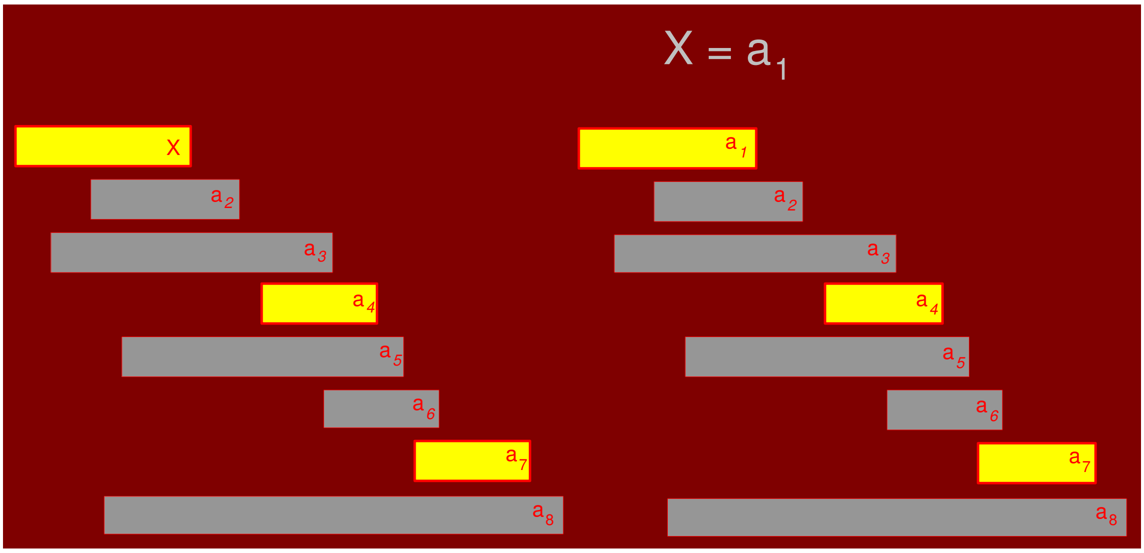


Figure 7.6: Activity $X = a_1$

We claim that $A' = A - \{x\} \cup \{a_1\}$ is a feasible schedule, i.e., it has no interfering activities. This because $A - \{x\}$ cannot have any other activities that start before x finishes, since otherwise, these activities will interfere with x.

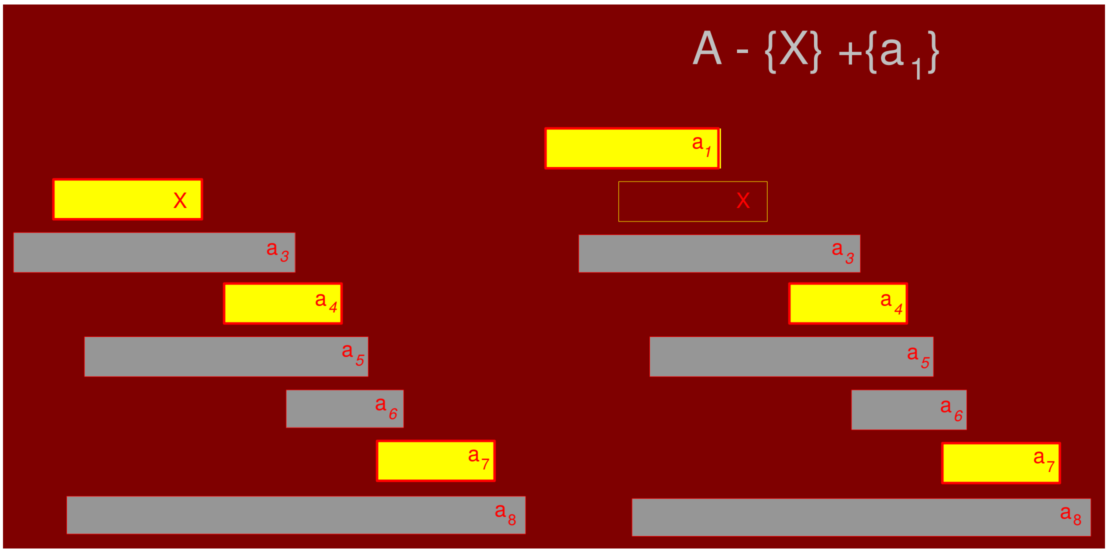


Figure 7.7: New schedule A' by replacing x with ctivity a_1.

Since a_1 is by definition the first activity to finish, it has an earlier finish time than x. Thus a_1 cannot interfere with any of the activities in $A - \{x\}$. Thus, A' is a feasible schedule. Clearly A and A' contain the same number of activities implying that A' is also optimal.

Claim:

The greedy algorithm gives an optimal solution to the activity scheduling problem.

Proof:

The proof is by induction on the number of activities. For the basis case, if there are no activities, then the greedy algorithm is trivially optimal. For the induction step, let us assume that the greedy algorithm is optimal on any set of activities of size strictly smaller than $|S|$ and we prove the result for S. Let S' be the set of activities that do not interfere with activity a_1, That is

$$S' = \{a_i \in S | s_i \geq f_1\}$$

Any solution for S' can be made into a solution for S by simply adding activity a_1, and vice versa. Activity a_1 is in the optimal schedule (by the above previous claim). It follows that to produce an optimal schedule for the overall problem, we should first schedule a_1 and then append the optimal schedule for S'. But by induction (since $|S'| < |S|$), this exactly what the greedy algorithm does.

7.4 Fractional Knapsack Problem

Earlier we saw the 0-1 knapsack problem. A knapsack can only carry W total weight. There are n items; the i^{th} item is worth v_i and weighs w_i. Items can either be put in the knapsack or not. The goal was to maximize the value of items without exceeding the total weight limit of W. In contrast, in the fractional knapsack problem, the setup is exactly the same. But, one is allowed to take *fraction* of an item for a fraction of the weight and fraction of value. The 0-1 knapsack problem is hard to solve. However, there is a simple and efficient greedy algorithm for the fractional knapsack problem.

Let $\rho_i = v_i/w_i$ denote the *value per unit weight* ratio for item i. Sort the items in decreasing order of ρ_i. Add items in decreasing order of ρ_i. If the item fits, we take it all. At some point there is an item that does not fit in the remaining space. We take as much of this item as possible thus filling the knapsack completely.

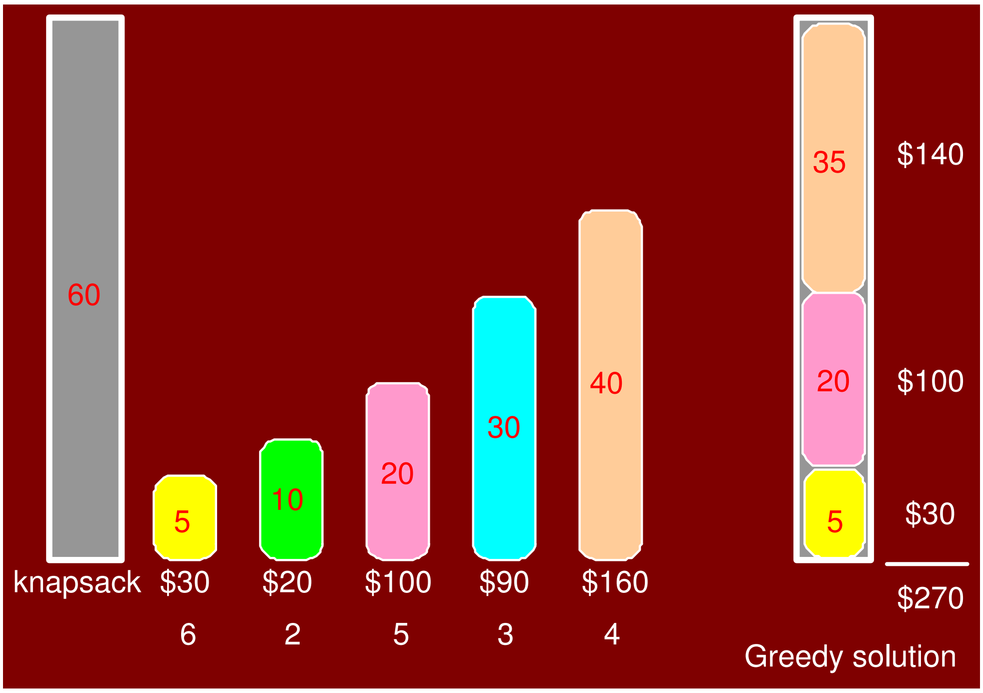


Figure 7.8: Greedy solution to the fractional knapsack problem

It is easy to see that the greedy algorithm is optimal for the fractional knapsack problem. Given a room with sacks of gold, silver and bronze, one (thief?) would probably take as much gold as possible. Then take as much silver as possible and finally as much bronze as possible. It would never benefit to take a little less gold so that one could replace it with an equal weight of bronze.

We can also observe that the greedy algorithm is not optimal for the 0-1 knapsack problem. Consider the example shown in the Figure 7.9. If you were to sort the items by ρ_i, then you would first take the items of weight 5, then 20, and then (since the item of weight 40 does not fit) you would settle for the item of weight 30, for a total value of $30 + $100 + $90 = $220. On the other hand, if you had been less greedy, and ignored the item of weight 5, then you could take the items of weights 20 and 40 for a total value of $100+$160 = $260. This is shown in Figure 7.10.

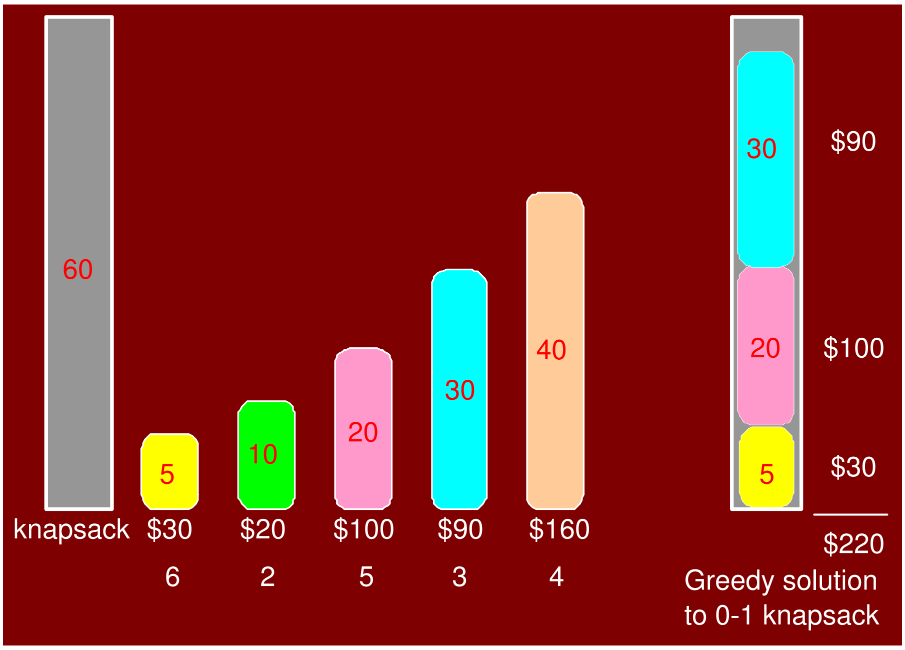


Figure 7.9: Greedy solution for the 0-1 knapsack problem (non-optimal)

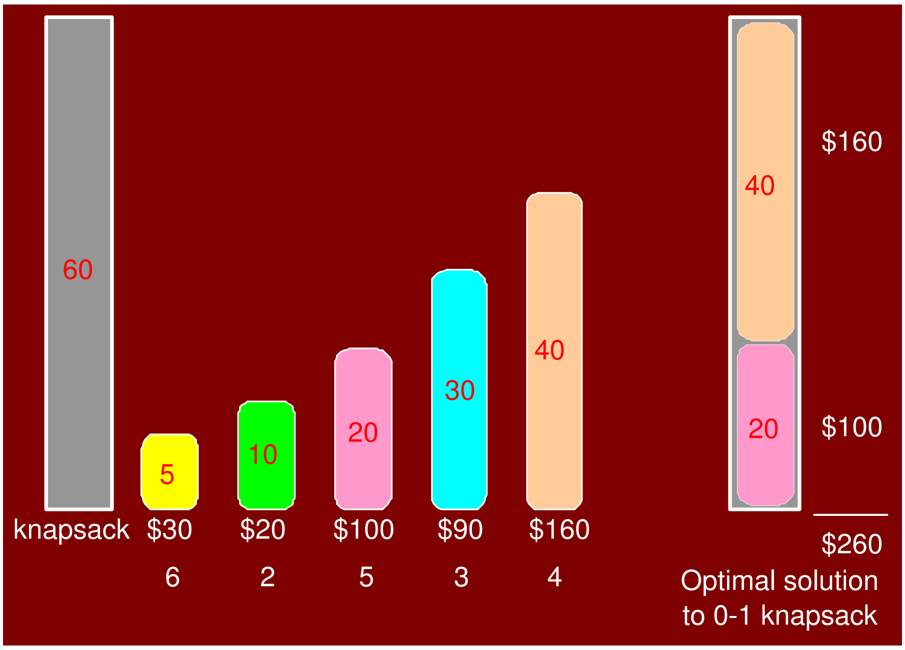


Figure 7.10: Optimal solution for the 0-1 knapsack problem

Chapter 8

Graphs

We begin a major new topic: Graphs. Graphs are important discrete structures because they are a flexible mathematical model for many application problems. Any time there is a set of objects and there is some sort of "connection" or "relationship" or "interaction" between pairs of objects, a graph is a good way to model this. Examples of this can be found in computer and communication networks transportation networks, e.g., roads VLSI, logic circuits surface meshes for shape description in computer-aided design and GIS precedence constraints in scheduling systems.

A *graph* $G = (V, E)$ consists of a finite set of *vertices* V (or nodes) and E, a binary relation on V called *edges*. E is a set of pairs from V. If a pair is *ordered*, we have a *directed* graph. For *unordered* pair, we have an *undirected* graph.

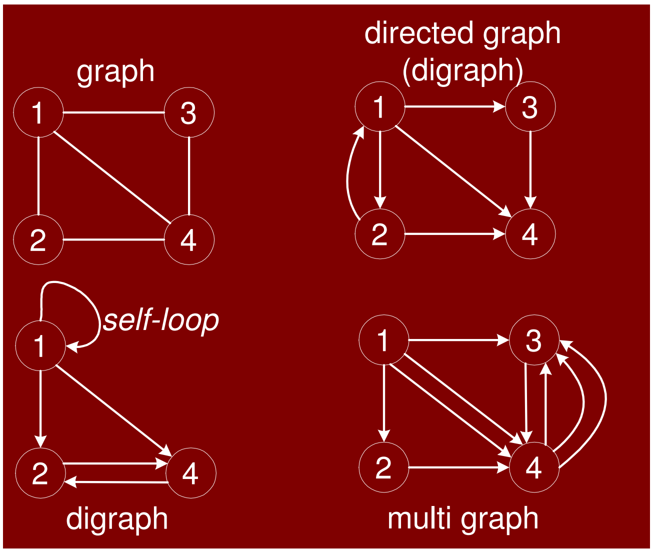
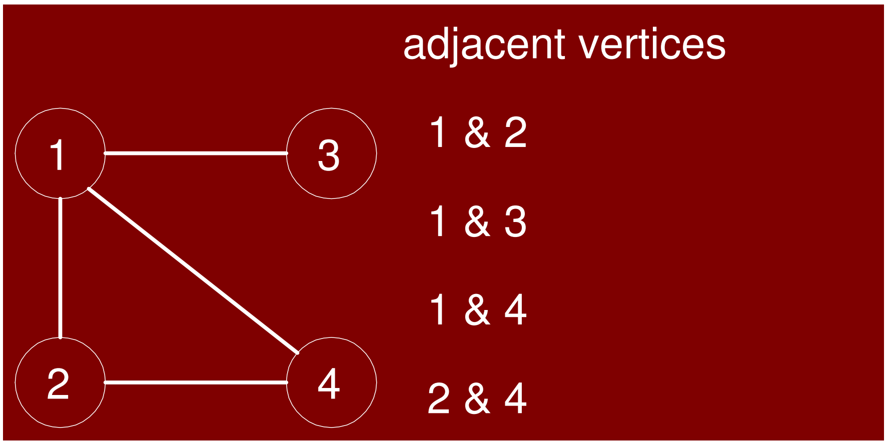


Figure 8.1: Types of graphs

A vertex w is *adjacent* to vertex v if there is an edge from v to w.



adjacent vertices

1 & 2

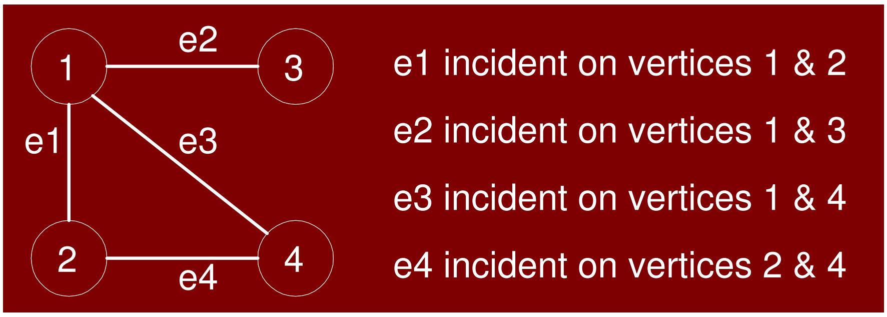
1 & 3

1 & 4

2 & 4

Figure 8.2: Adjacent vertices

In an undirected graph, we say that an edge is *incident* on a vertex if the vertex is an endpoint of the edge. of the edge



e1 incident on vertices 1 & 2

e2 incident on vertices 1 & 3

e3 incident on vertices 1 & 4

e4 incident on vertices 2 & 4

Figure 8.3: Incidence of edges on vertices

In a digraph, the number of edges coming out of a vertex is called the *out-degree* of that vertex. Number of edges coming in is the *in-degree*. In an undirected graph, we just talk of degree of a vertex. It is the number of edges incident on the vertex.

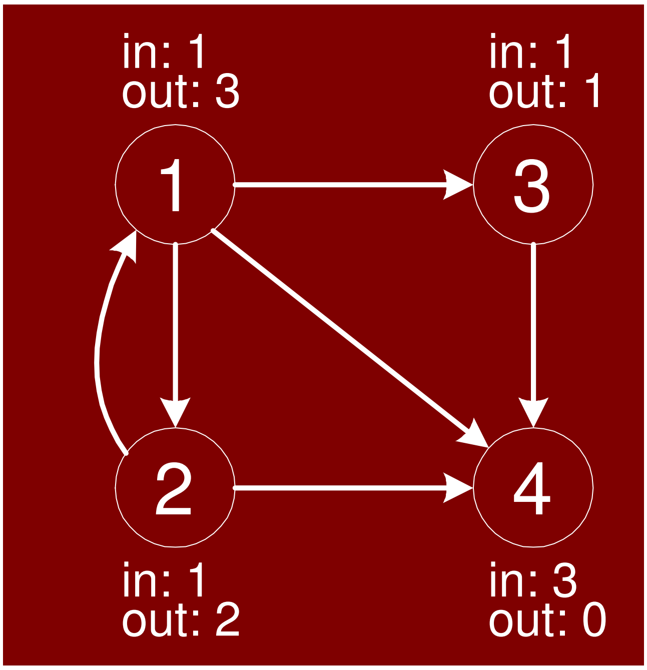


Figure 8.4: In and out degrees of vertices of a graph

For a digraph $G = (V, E)$,

$$\sum_{v \in V} \mathtt{in\text{-}degree}(v) = \sum_{v \in V} \mathtt{out\text{-}degree}(v) = |E|$$

where $|E|$ means the cardinality of the set E, i.e., the number of edges.

For an undirected graph $G = (V, E)$,

$$\sum_{v \in V} \mathtt{degree}(v) = 2|E|$$

where $|E|$ means the cardinality of the set E, i.e., the number of edges.

A *path* in a directed graphs is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$ such that (v_{i-1}, v_i) is an edge for $i = 1, 2, \ldots, k$. The *length* of the paths is the number of edges, k. A vertex w is *reachable* from vertex u is there is a path from u to w. A path is simple if all vertices (except possibly the fist and last) are distinct.

A *cycle* in a digraph is a path containing at least one edge and for which $v_0 = v_k$. A *Hamiltonian* cycle is a cycle that visits every vertex in a graph exactly once. A *Eulerian* cycle is a cycle that visits every edge of the graph exactly once. There are also “path” versions in which you do not need return to the starting vertex.

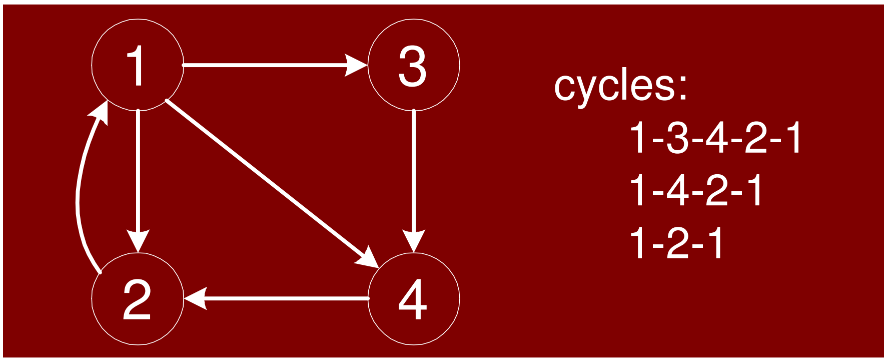


Figure 8.5: Cycles in a directed graph

A graph is said to be *acyclic* if it contains no cycles. A graph is *connected* if every vertex can reach every other vertex. A directed graph that is acyclic is called a *directed acyclic graph (DAG).*

There are two ways of representing graphs: using an adjacency matrix and using an adjacency list. Let $G = (V, E)$ be a digraph with $n = |V|$ and let $e = |E|$. We will assume that the vertices of G are indexed $\{1, 2, \ldots, n\}$.

An *adjacency matrix* is a $n \times n$ matrix defined for $1 \leq v, w \leq n$.

$$A[v, w] = \begin{cases} 1 & \text{if } (v, w) \in E \\ 0 & \text{otherwise} \end{cases}$$

An *adjacency list* is an array $Adj[1..n]$ of pointers where for $1 \leq v \leq n$, $Adj[v]$ points to a linked list containing the vertices which are adjacent to v

Adjacency matrix requires $\Theta(n^2)$ storage and adjacency list requires $\Theta(n + e)$ storage.

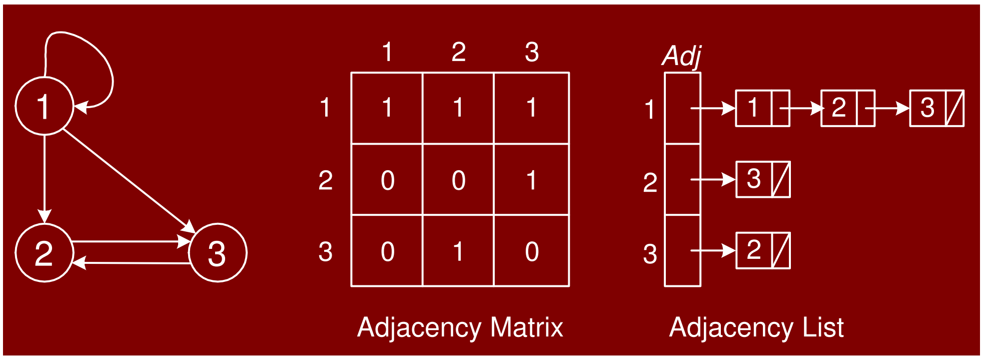


Figure 8.6: Graph Representations

8.1 Graph Traversal

To motivate our first algorithm on graphs, consider the following problem. We are given an undirected graph $G = (V, E)$ and a *source vertex* $s \in V$. The *length* of a path in a graph is the number of edges on the path. We would like to find the shortest path from s to each other vertex in the graph. The final result will be represented in the following way. For each vertex $v \in V$, we will store $d[v]$ which is the *distance* (length of the shortest path) from s to v. Note that $d[s] = 0$. We will also store a predecessor (or parent) pointer $\pi[v]$ which is the first vertex along the shortest path if we walk from v backwards to s. We will set $\pi[s] = \text{Nil}$.

There is a simple brute-force strategy for computing shortest paths. We could simply start enumerating all simple paths starting at s, and keep track of the shortest path arriving at each vertex. However, there can be as many as $n!$ simple paths in a graph. To see this, consider a fully connected graph shown in Figure 8.7

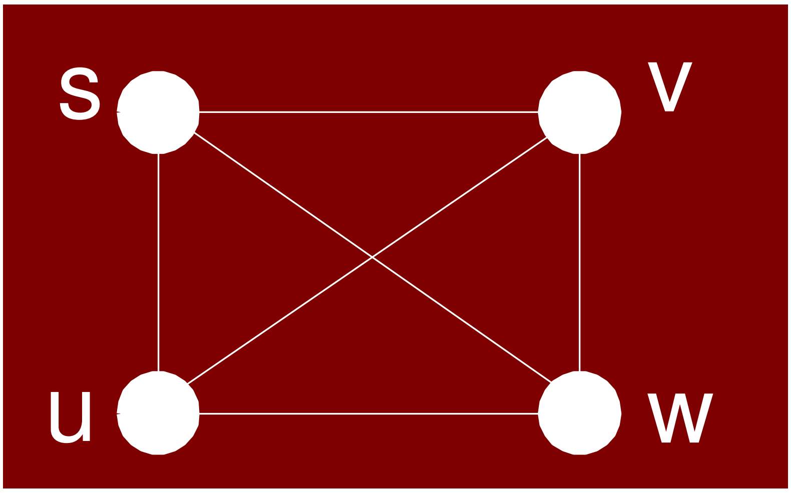


Figure 8.7: Fully connected graph

There n choices for source node s, $(n-1)$ choices for destination node, $(n-2)$ for first hop (edge) in the path, $(n-3)$ for second, $(n-4)$ for third down to $(n-(n-1))$ for last leg. This leads to $n!$ simple paths. Clearly this is not feasible.

8.1.1 Breadth-first Search

Here is a more efficient algorithm called the *breadth-first search* (BFS) Start with s and visit its adjacent nodes. Label them with distance 1. Now consider the neighbors of neighbors of s. These would be at distance 2. Now consider the neighbors of neighbors of neighbors of s. These would be at distance 3. Repeat this until no more unvisited neighbors left to visit. The algorithm can be visualized as a *wave front* propagating outwards from s visiting the vertices in bands at ever increasing distances from s.

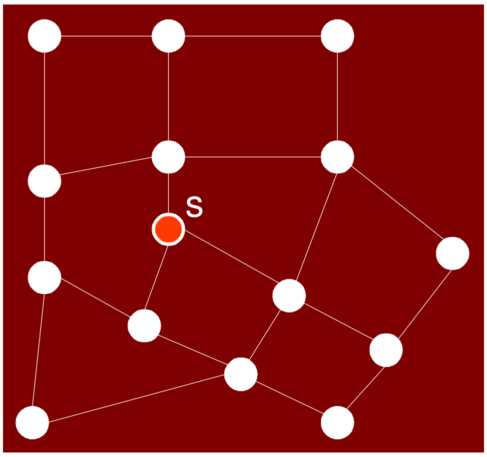


Figure 8.8: Source vertex for breadth-first-search (BFS)

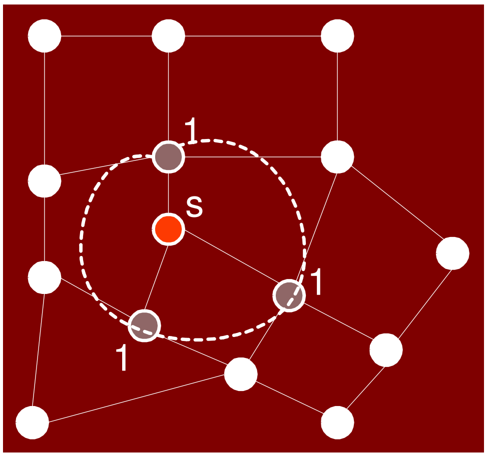


Figure 8.9: Wave reaching distance 1 vertices during BFS

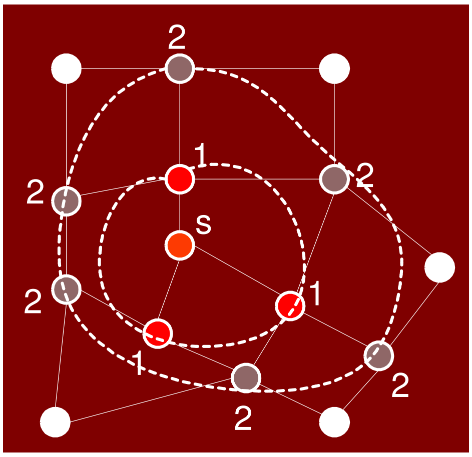


Figure 8.10: Wave reaching distance 2 vertices during BFS

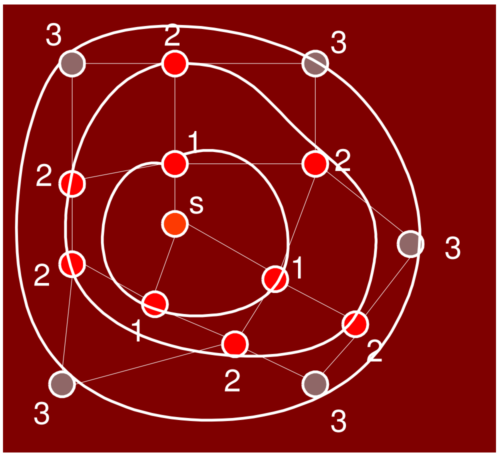


Figure 8.11: Wave reaching distance 3 vertices during BFS

8.1.2 Depth-first Search

Breadth-first search is one instance of a general family of *graph traversal algorithms*. Traversing a graph means visiting every node in the graph. Another traversal strategy is *depth-first search* (DFS). DFS procedure can be written recursively or non-recursively. Both versions are passed s initially.

```
RECURSIVEDFS($v$)
if ($v$ is unmarked )
   then mark $v$
        for each edge ($v, w$)
        do RECURSIVEDFS($w$)
```

```
ITERATIVEDFS($s$)
PUSH($s$)
while stack not empty
do $v \leftarrow$ POP()
   if $v$ is unmarked
      then mark $v$
           for each edge ($v, w$)
           do PUSH($w$)
```

8.1.3 Generic Graph Traversal Algorithm

The *generic graph traversal* algorithm stores a set of candidate edges in some data structures we'll call a "*bag*". The only important properties of the "bag" are that we can put stuff into it and then later take stuff back out. Here is the generic traversal algorithm.

```
TRAVERSE($s$)
1  put $(\varnothing, s)$ in bag
2  while bag not empty
3  do take $(p, v)$ from bag
4     if ($v$ is unmarked )
5       then mark $v$
6            parent $(v) \leftarrow p$
7            for each edge $(v, w)$
8            do put $(v, w)$ in bag
```

Notice that we are keeping edges in the bag instead of vertices. This is because we want to remember, whenever we visit v for the first time, which previously-visited vertex p put v into the bag. The vertex p is call the *parent of* v.

The running time of the traversal algorithm depends on how the graph is represented and what data structure is used for the bag. But we can make a few general observations.

- Since each vertex is visited at most once, the for loop in line 7 is executed at most V times.
- Each edge is put into the bag exactly twice; once as (u, v) and once as (v, u), so line 8 is executed at most $2E$ times.
- Finally, since we can't take out more things out of the bag than we put in, line 3 is executed at most $2E + 1$ times.
- Assume that the graph is represented by an adjacency list so the overhead of the for loop in line 7 is constant per edge.

If we implement the bag by using a *stack*, we have *depth-first* search (DFS) or traversal.

```
TRAVERSE($s$)
1  push($\varnothing, s$)
2  while stack not empty
3  do pop($p, v$)
4     if ($v$ is unmarked )
5       then mark $v$
6            parent $(v) \leftarrow p$
7            for each edge $(v, w)$
8            do push($v, w$)
```

Figures 8.12 to 8.20 show a trace of the DFS algorithm applied to a graph. The figures show the content of the stack during the execution of the algorithm.

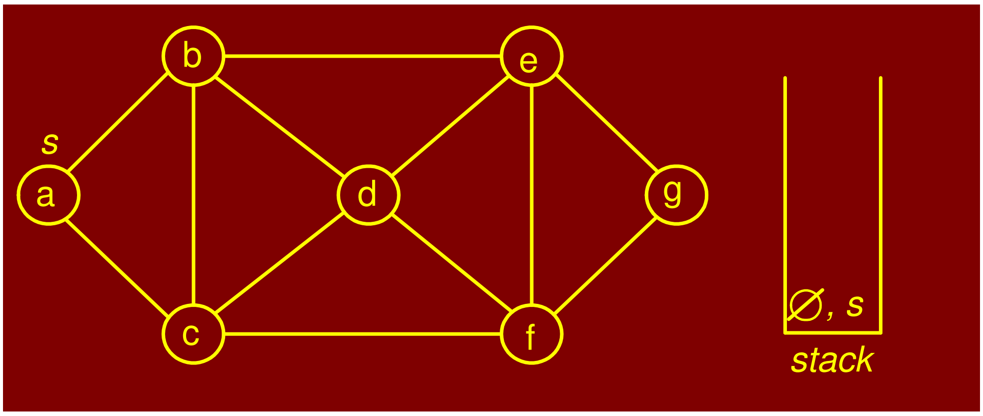


Figure 8.12: Trace of Depth-first-search algorithm: source vertex ‘s’

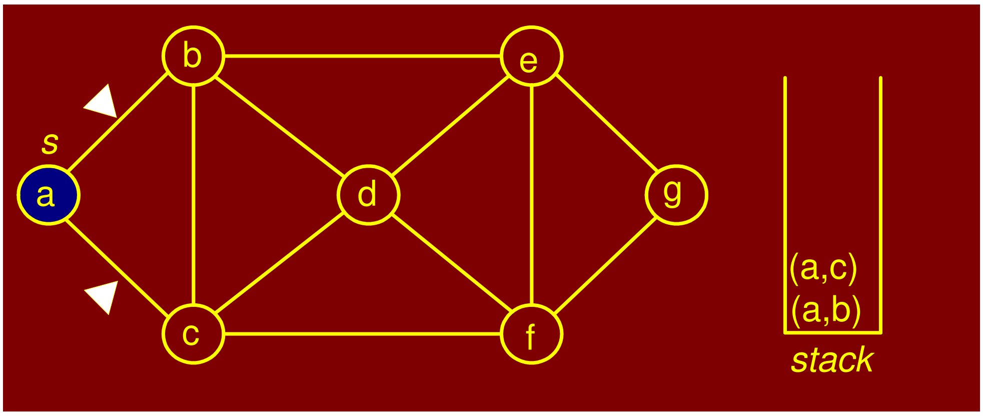


Figure 8.13: Trace of DFS algorithm: vertex ‘a’ popped

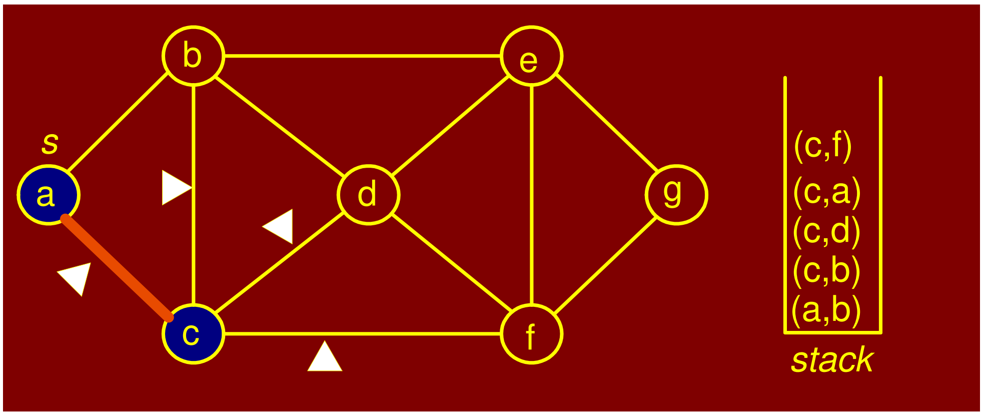


Figure 8.14: Trace of DFS algorithm: vertex ‘c’ popped

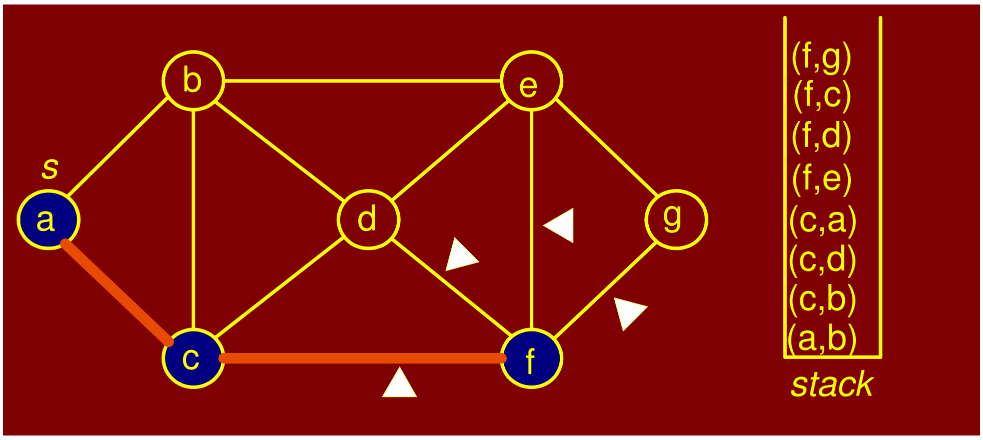


Figure 8.15: Trace of DFS algorithm: vertex ‘f’ popped

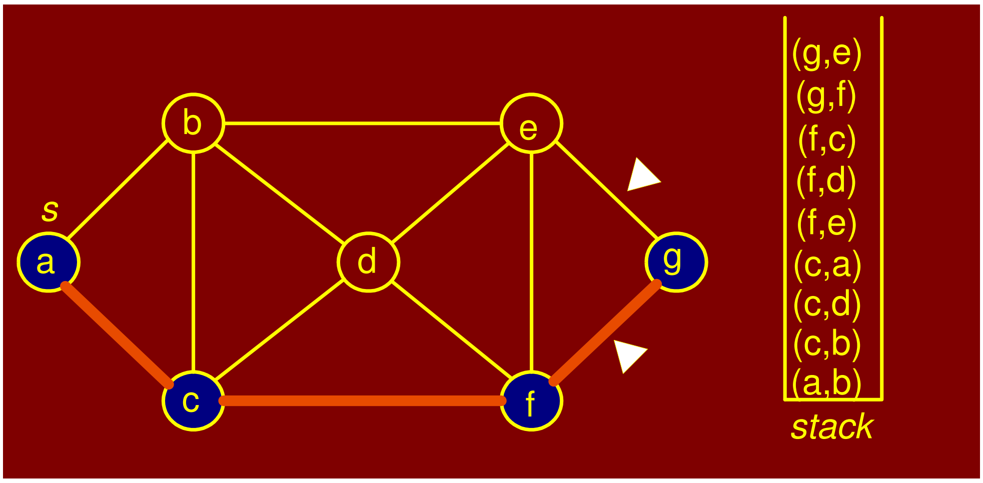


Figure 8.16: Trace of DFS algorithm: vertex ‘g’ popped

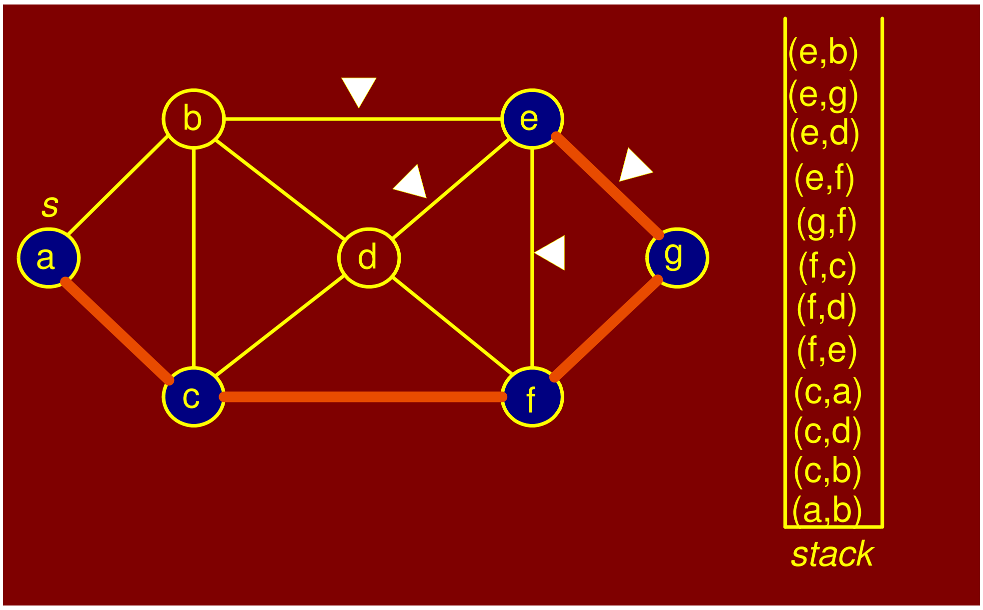


Figure 8.17: Trace of DFS algorithm: vertex ‘e’ popped

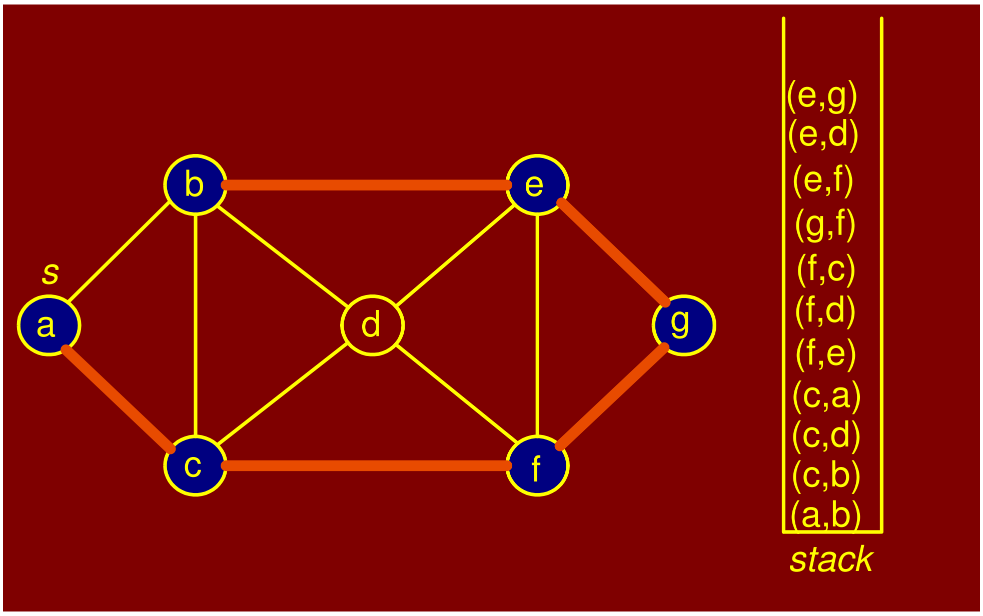


Figure 8.18: Trace of DFS algorithm: vertex ‘b’ popped

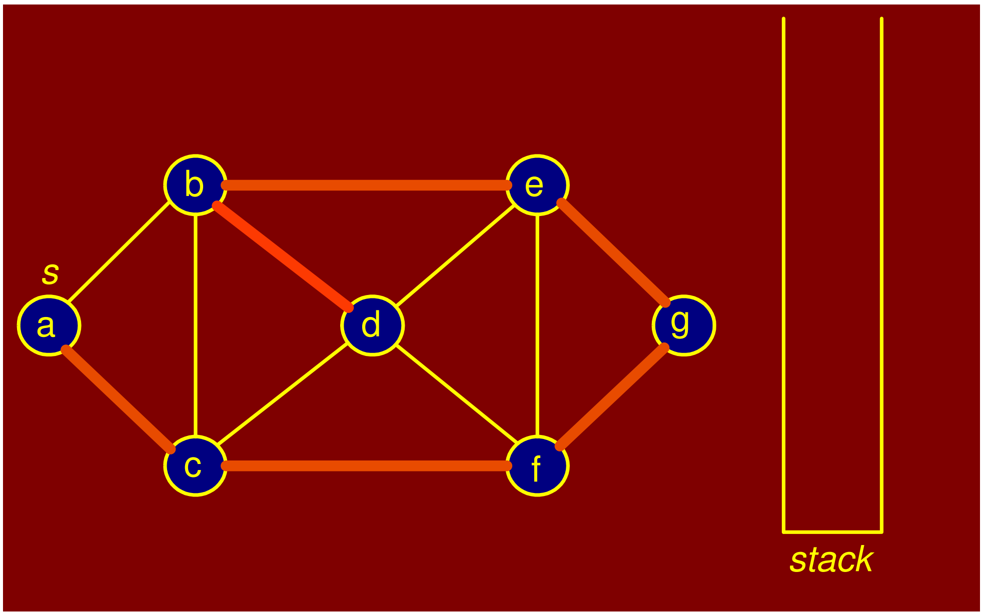


Figure 8.19: Trace of DFS algorithm: vertex ‘d’ popped

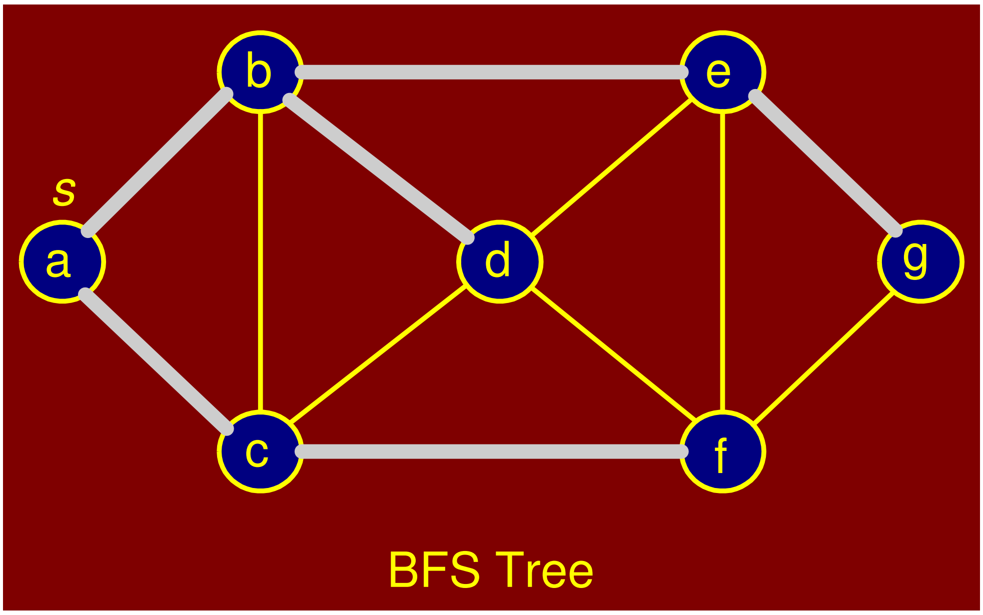


Figure 8.20: Trace of DFS algorithm: the final DFS tree

Each execution of line 3 or line 8 in the TRAVERSE-DFS algorithm takes constant time. So the overall running time is $O(V + E)$. Since the graph is connected, $V \leq E + 1$, this is $O(E)$.

If we implement the bag by using a *queue*, we have *breadth-first* search (BFS). Each execution of line 3 or line 8 still takes constant time. So overall running time is still $O(E)$.

```
TRAVERSE(s)
1  enqueue(∅, s)
2  while queue not empty
3  do dequeue(p, v)
4     if (v is unmarked )
5       then mark v
6            parent (v) ← p
7            for each edge (v, w)
8            do enqueue(v, w)
```

If the graph is represented using an *adjacency matrix*, the finding of all the neighbors of vertex in line 7 takes $O(V)$ time. Thus depth-first and breadth-first take $O(V^2)$ time overall.

Either DFS or BFS yields a spanning tree of the graph. The tree visits every vertex in the graph. This fact is established by the following lemma:

Lemma:

The generic TRAVERSE(S) marks every vertex in any connected graph exactly once and the set of edges $(v, \mathit{parent}(v))$ with $\mathit{parent}(v) \neq \emptyset$ form a spanning tree of the graph.

Proof:

First, it should be obvious that no vertex is marked more than once. Clearly, the algorithm marks s. Let $v \neq s$ be a vertex and let $s \rightarrow \cdots \rightarrow u \rightarrow v$ be a path from s to v with the minimum number of edges.

Since the graph is connected, such a path always exists. If the algorithm marks u, then it must put (u, v) into the bag, so it must take (u, v) out of the bag at which point v must be marked. Thus, by induction on the shortest-path distance from s, the algorithm marks every vertex in the graph.

Call an edge $(v, \mathit{parent}(v))$ with $\mathit{parent}(v) \neq \emptyset$, a *parent edge*. For any node v, the path of parent edges $v \rightarrow \mathit{parent}(v) \rightarrow \mathit{parent}(\mathit{parent}(v)) \rightarrow \ldots$ eventually leads back to s. So the set of parent edges form a connected graph.

Clearly, both end points of every parent edge are marked, and the number of edges is exactly one less than the number of vertices. Thus, the parent edges form a *spanning tree*.

8.1.4 DFS - Timestamp Structure

As we traverse the graph in DFS order, we will associate two numbers with each vertex. When we first discover a vertex u, store a counter in $d[u]$. When we are finished processing a vertex, we store a counter in $f[u]$. These two numbers are *time stamps*.

Consider the *recursive* version of depth-first traversal

```
DFS(G)
for (each u ∈ V)
do color[u] ← white
   pred[u] ← nil
time ← 0
for each u ∈ V
do if (color[u] = white)
      then DFSVISIT(u)
```

The DFSVISIT routine is as follows:

```
DFSVISIT(u)
color[u] ← gray;  // mark u visited
d[u] ← ++ time
for (each v ∈ Adj[u])
do if (color[v] = white)
      then pred[v] ← u
           DFSVISIT(v)
color[u] ← black;  // we are done with u
f[u] ← ++ time;
```

Figures 8.21 through 8.25 present a trace of the execution of the time stamping algorithm. Terms like “2/5” indicate the value of the counter (time). The number before the “/” is the time when a vertex was discovered (colored gray) and the number after the “/” is the time when the processing of the vertex finished (colored black).

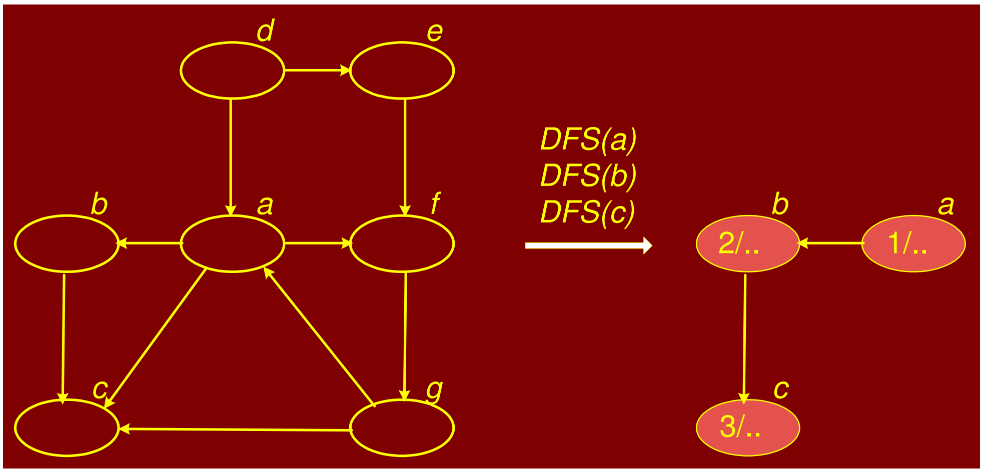


Figure 8.21: DFS with time stamps: recursive calls initiated at vertex ‘a’

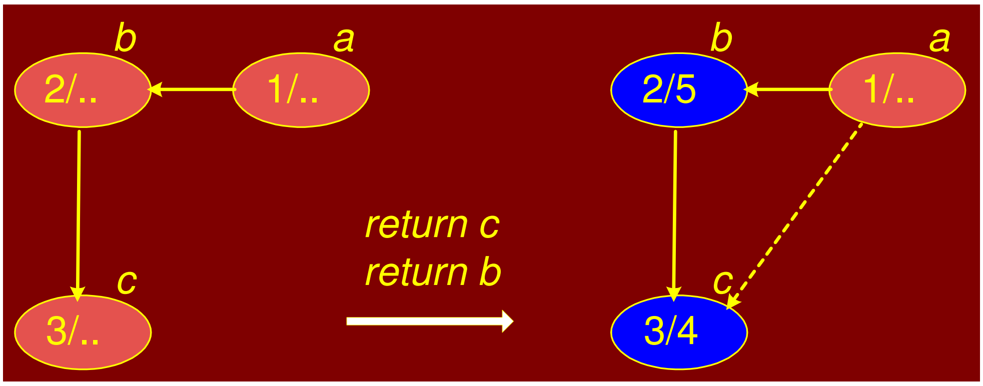


Figure 8.22: DFS with time stamps: processing of ‘b’ and ‘c’ completed

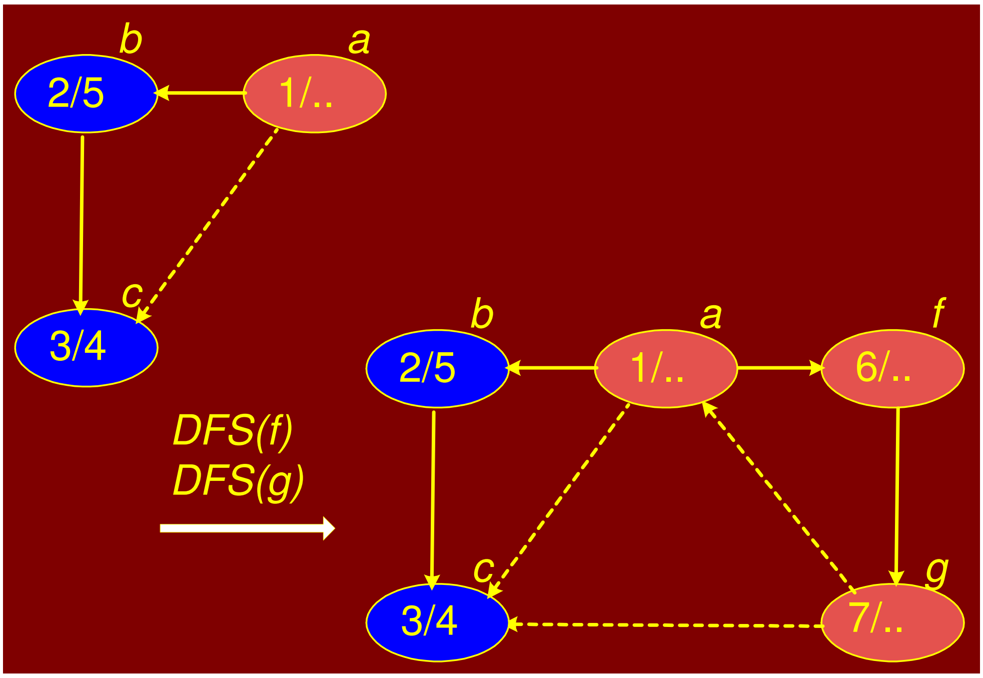
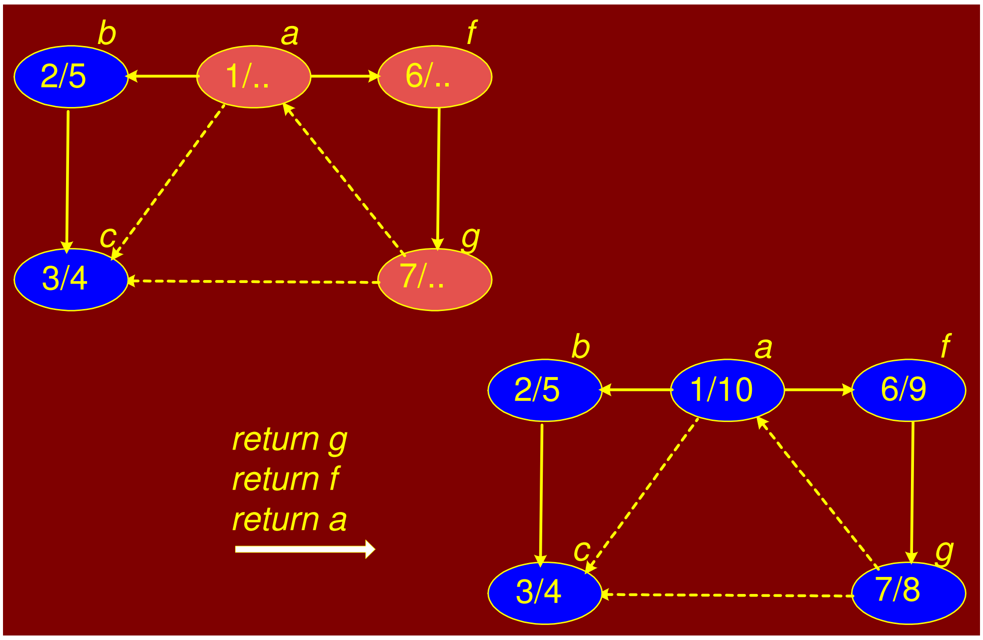
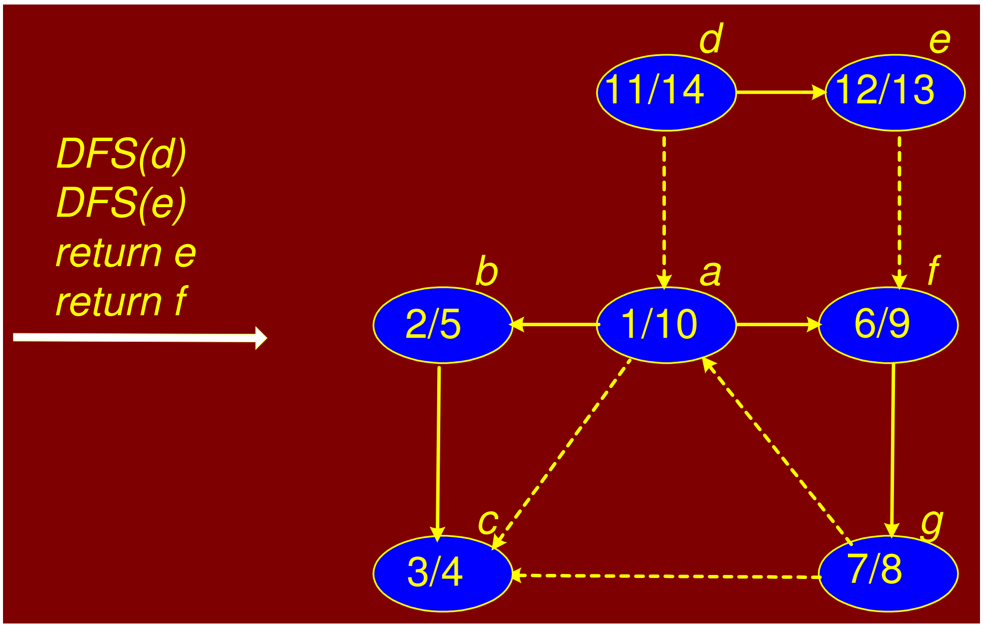


Figure 8.23: DFS with time stamps: recursive processing of ‘f’ and ‘g’



return g
return f
return a

Figure 8.24: DFS with time stamps: processing of ‘f’ and ‘g’ completed



DFS(d)
DFS(e)
return e
return f

Figure 8.25: DFS with time stamps: processing of ‘d’ and ‘e’

Notice that the DFS tree structure (actually a collection of trees, or a forest) on the structure of the graph is just the recursion tree, where the edge (u, v) arises when processing vertex u we call DFSVISIT(V) for some neighbor v. For *directed graphs* the edges that are not part of the tree (indicated as dashed edges in Figures 8.21 through 8.25) edges of the graph can be classified as follows:

Back edge: (u, v) where v is an ancestor of u in the tree.

Forward edge: (u, v) where v is a proper descendent of u in the tree.

Cross edge: (u, v) where u and v are not ancestor or descendent of one another. In fact, the edge may go between different trees of the forest.

The ancestor and descendent relation can be nicely inferred by the *parenthesis* lemma. u is a descendent of v if and only if $[d[u], f[u]] \subseteq [d[v], f[v]]$. u is a ancestor of v if and only if $[d[u], f[u]] \supseteq [d[v], f[v]]$. u is unrelated to v if and only if $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint. The is shown in Figure 8.26. The width of the rectangle associated with a vertex is equal to the time the vertex was discovered till the time the vertex was completely processed (colored black). Imagine an opening parenthesis '(' at the start of the rectangle and and closing parenthesis ')' at the end of the rectangle. The rectangle (parentheses) for vertex 'b' is completely enclosed by the rectangle for 'a'. Rectangle for 'c' is completely enclosed by vertex 'b' rectangle.

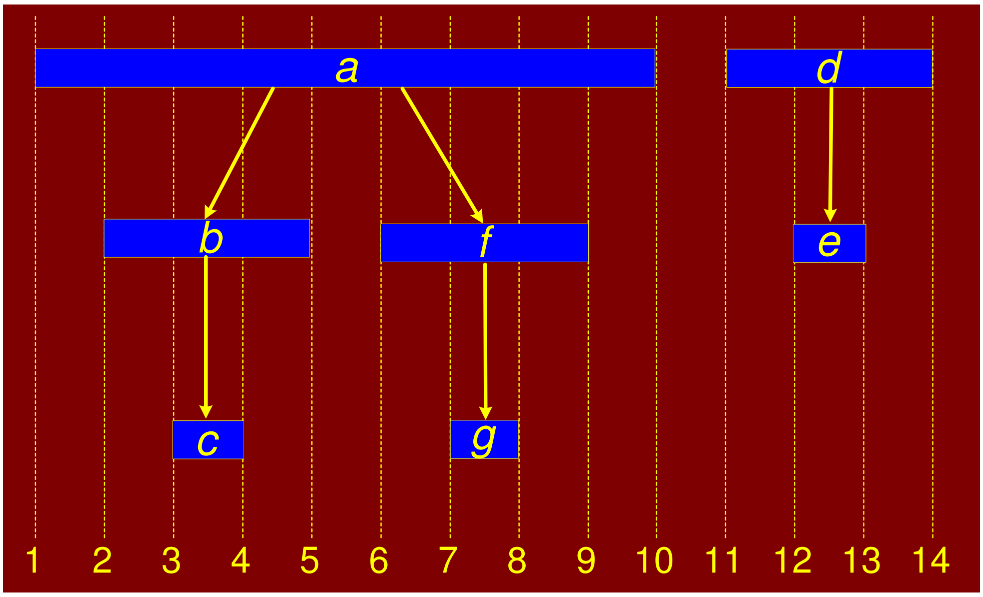


Figure 8.26: Parenthesis lemma

Figure 8.27 shows the classification of the non-tree edges based on the parenthesis lemma. Edges are labelled 'F', 'B' and 'C' for forward, back and cross edge respectively.

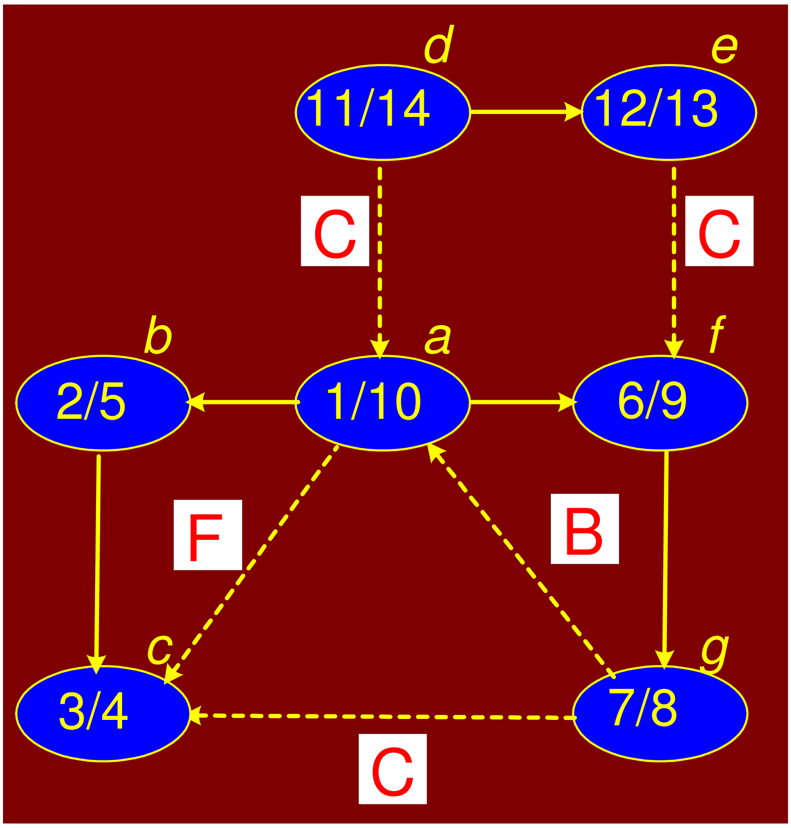


Figure 8.27: Classfication of non-tree edges in the DFS tree for a graph

For *undirected* graphs, there is no distinction between forward and back edges. By convention they are all called back edges. Furthermore, there are no cross edges (can you see why not?)

8.1.5 DFS - Cycles

The time stamps given by DFS allow us to determine a number of things about a graph or digraph. For example, we can determine whether the graph contains any *cycles*. We do this with the help of the following two lemmas.

Lemma: Given a digraph $G = (V, E)$, consider any DFS forest of G and consider any edge $(u, v) \in E$. If this edge is a tree, forward or cross edge, then $f[u] > f[v]$. If this edge is a back edge, then $f[u] \leq f[v]$.

Proof: For the non-tree forward and back edges the proof follows directly from the parenthesis lemma. For example, for a forward edge (u, v), v is a descendent of u and so v's start-finish interval is contained within u's implying that v has an earlier finish time. For a cross edge (u, v) we know that the two time intervals are disjoint. When we were processing u, v was not white (otherwise (u, v) would be a tree edge), implying that v was started before u. Because the intervals are disjoint, v must have also finished before u.

Lemma: Consider a digraph $G = (V, E)$ and any DFS forest for G. G has a cycle if and only if the DFS forest has a *back* edge.

Proof: If there is a back edge (u, v) then v is an ancestor of u and by following tree edge from v to u, we get a cycle.

We show the contrapositive: suppose there are no back edges. By the lemma above, each of the remaining types of edges, tree, forward, and cross all have the property that they go from vertices with higher finishing time to vertices with lower finishing time. Thus along any path, finish times decrease monotonically, implying there can be no cycle.

The DFS forest in Figure 8.27 has a back edge from vertex 'g' to vertex 'a'. The cycle is 'a-g-f'.

Beware: No back edges means no cycles. But you should not infer that there is some simple relationship between the number of back edges and the number of cycles. For example, a DFS tree may only have a single back edge, and there may anywhere from one up to an exponential number of simple cycles in the graph.

A similar theorem applies to undirected graphs, and is not hard to prove.

8.2 Precedence Constraint Graph

A *directed acyclic graph* (DAG) arise in many applications where there are precedence or ordering constraints. There are a series of tasks to be performed and certain tasks must precede other tasks. For example, in construction, you have to build the first floor before the second floor but you can do electrical work while doors and windows are being installed. In general, a *precedence constraint graph* is a DAG in which vertices are tasks and the edge (u, v) means that task u must be completed before task v begins.

For example, consider the sequence followed when one wants to dress up in a suit. One possible order and its DAG are shown in Figure 8.28. Figure 8.29 shows the DFS with time stamps of the DAG.

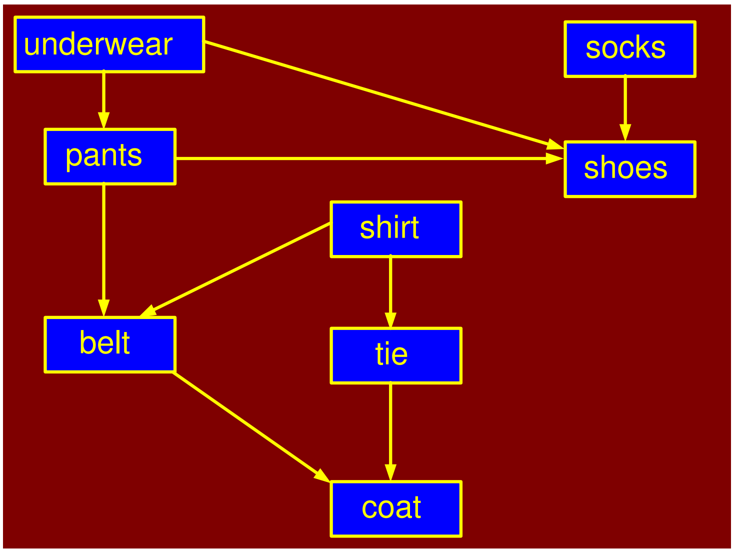


Figure 8.28: Order of dressing up in a suit

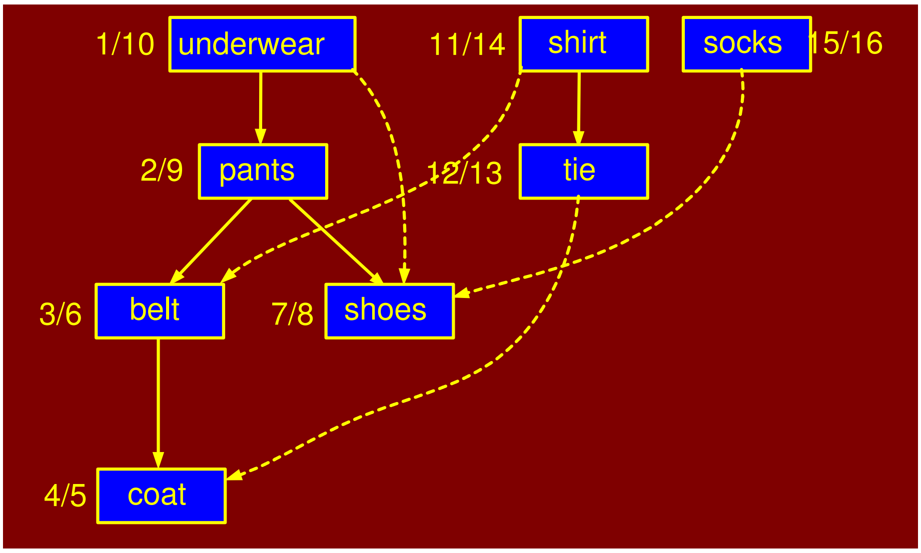


Figure 8.29: DFS of dressing up DAG with time stamps

Another example of precedence constraint graph is the sets of prerequisites for CS courses in a typical undergraduate program.

C1	Introduction to Computers	
C2	Introduction to Computer Programming	
C3	Discrete Mathematics	
C4	Data Structures	C2
C5	Digital Logic Design	C2
C6	Automata Theory	C3
C7	Analysis of Algorithms	C3, C4
C8	Computer Organization and Assembly	C2
C9	Data Base Systems	C4, C7
C10	Computer Architecture	C4, C5,C8
C11	Computer Graphics	C4,C7
C12	Software Engineering	C7,C11
C13	Operating System	C4,C7,C11
C14	Compiler Construction	C4,C6,C8
C15	Computer Networks	C4,C7,C10

Table 8.1: Prerequisites for CS courses

The prerequisites can be represented with a precedence constraint graph which is shown in Figure 8.30

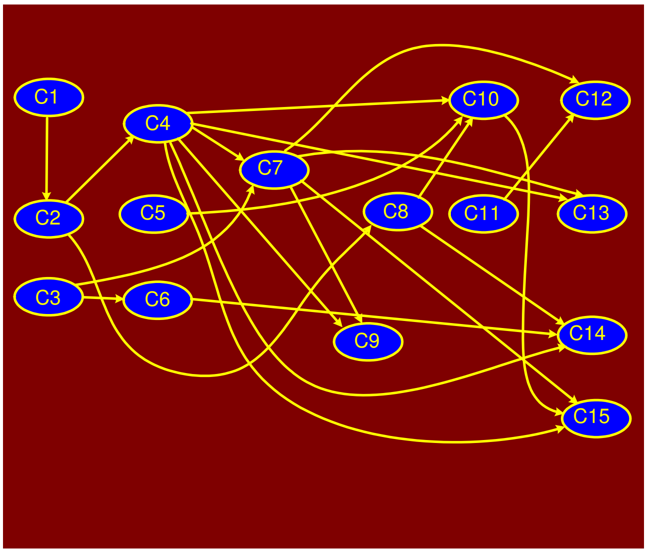


Figure 8.30: Precedence constraint graph for CS courses

8.3 Topological Sort

A topological sort of a DAG is a linear ordering of the vertices of the DAG such that for each edge (u, v), u appears before v in the ordering.

Computing a topological ordering is actually quite easy, given a DFS of the DAG. For every edge (u, v) in a DAG, the finish time of u is greater than the finish time of v (by the lemma). Thus, it suffices to output the vertices in the reverse order of finish times.

We run DFS on the DAG and when each vertex is finished, we add it to the front of a linked. Note that in general, there may be many legal topological orders for a given DAG.

```
TopologicalSort(G)
for (each u ∈ V)
    do color[u] ← white
L ← new LinkedList()
for each u ∈ V
    do if (color[u] = white)
        then TopVisit(u)
return L
```

```
TopVisit(u)
color[u] ← gray;  // mark u visited
for (each v ∈ Adj[u])
    do if (color[v] = white)
        then TopVisit(v)
Append u to the front of L
```

Figure 8.31 shows the linear order obtained by the topological sort of the sequence of putting on a suit. The DAG is still the same; it is only that the order in which the vertices of the graph have been laid out is special. As a result, all directed edges go from left to right.

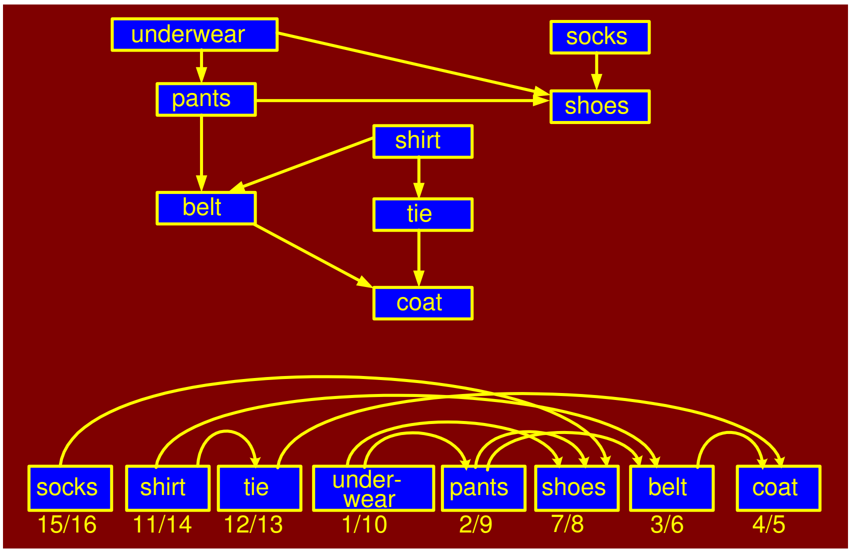


Figure 8.31: Topological sort of the dressing up sequence

This is a typical example of how DFS used in applications. The running time is $\Theta(V + E)$.

8.4 Strong Components

We consider an important connectivity problem with digraphs. When diagraphs are used in communication and transportation networks, people want to know that their networks are *complete*. Complete in the sense that that it is possible from any location in the network to reach any other location in the digraph.

A digraph is *strongly connected* if for every pair of vertices $u, v \in V$, u can reach v and vice versa. We would like to write an algorithm that determines whether a digraph is strongly connected. In fact, we will solve a generalization of this problem, of computing the *strongly connected components* of a digraph.

We partition the vertices of the digraph into subsets such that the induced subgraph of each subset is strongly connected. We say that two vertices u and v are *mutually reachable* if u can reach v and vice versa. Consider the directed graph in Figure 8.32. The strong components are illustrated in Figure 8.33.

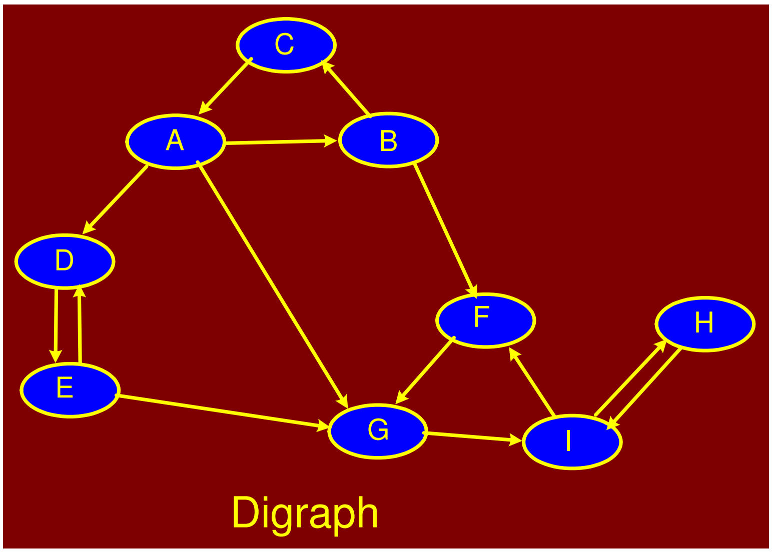


Figure 8.32: A directed graph

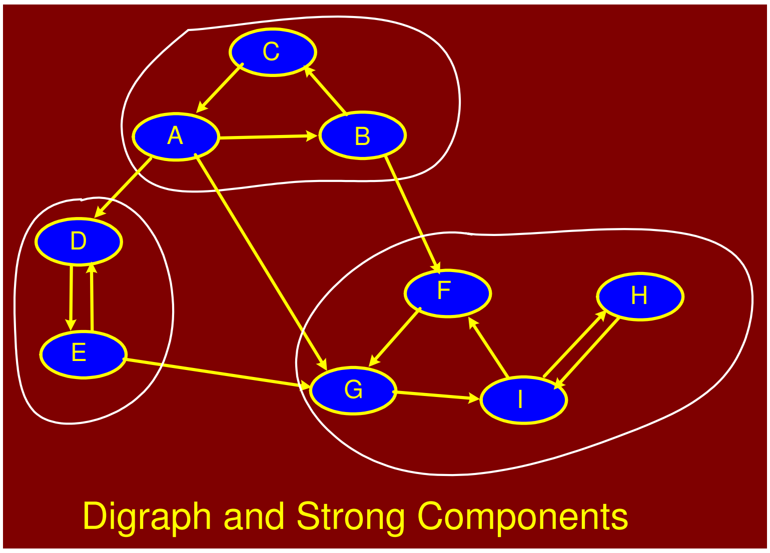


Figure 8.33: Digraph with strong components

It is easy to see that mutual reachability is an *equivalence relation*. This equivalence relation partitions the vertices into equivalence classes of mutually reachable vertices and these are the strong components.

If we merge the vertices in each strong component into a single *super vertex*, and join two super vertices (A, B) if and only if there are vertices $u \in A$ and $v \in B$ such that $(u, v) \in E$, then the resulting digraph is called the *component digraph*. The component digraph is necessarily acyclic. The is illustrated in Figure 8.34.

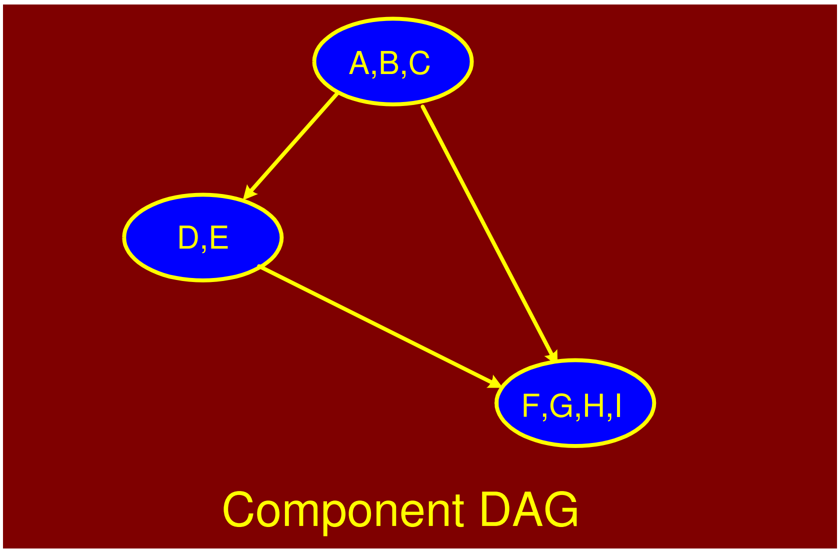


Figure 8.34: Component DAG of super vertices

8.4.1 Strong Components and DFS

Consider DFS of a digraph given in Figure **??**. Once you enter a strong component, every vertex in the component is reachable. So the DFS does not terminate until all the vertices in the component have been visited. Thus all vertices in a strong component must appear in the same tree of the DFS forest.

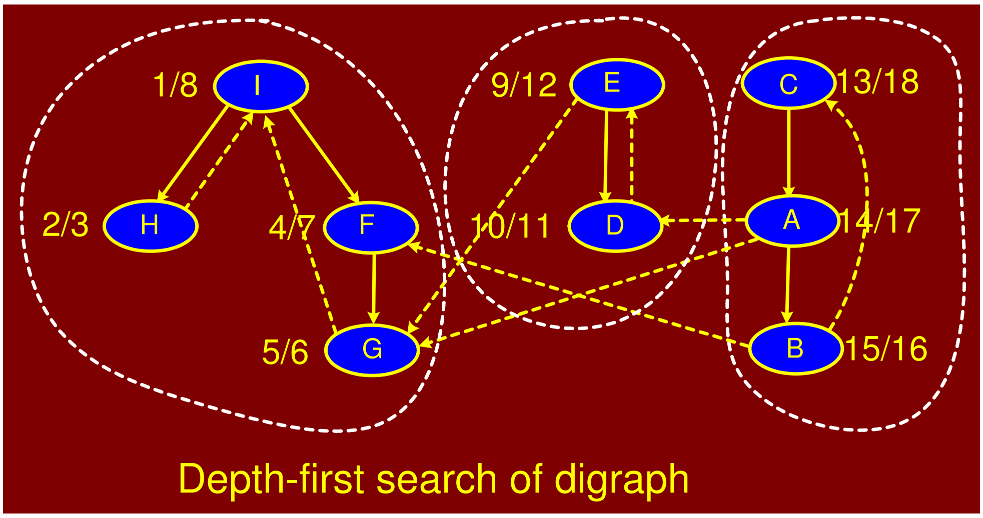


Figure 8.35: DFS of a digraph

fig:dfsofdigraph

Observe that each strong component is a subtree in the DFS forest. Is it always true for any DFS? The answer is "no". In general, many strong components may appear in the same DFS tree as illustrated in Figure 8.36

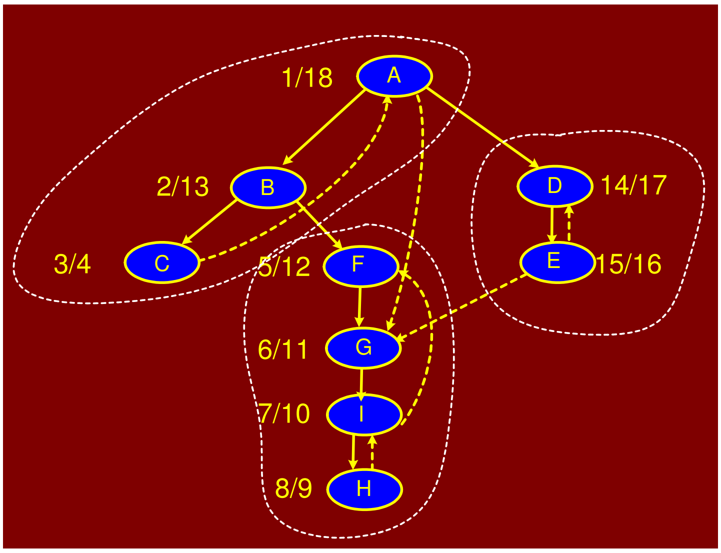


Figure 8.36: Another DFS tree of the digraph

Is there a way to order the DFS such that it true? Fortunately, the answer is “yes”. Suppose that you knew the component DAG in advance. (This is ridiculous, because you would need to know the strong components and this is the problem we are trying to solve.) Further, suppose that you computed a *reversed topological order* on the component DAG. That is, for edge (u, v) in the component DAG, then v comes before u. This is presented in Figure 8.37. Recall that the component DAG consists of super vertices.

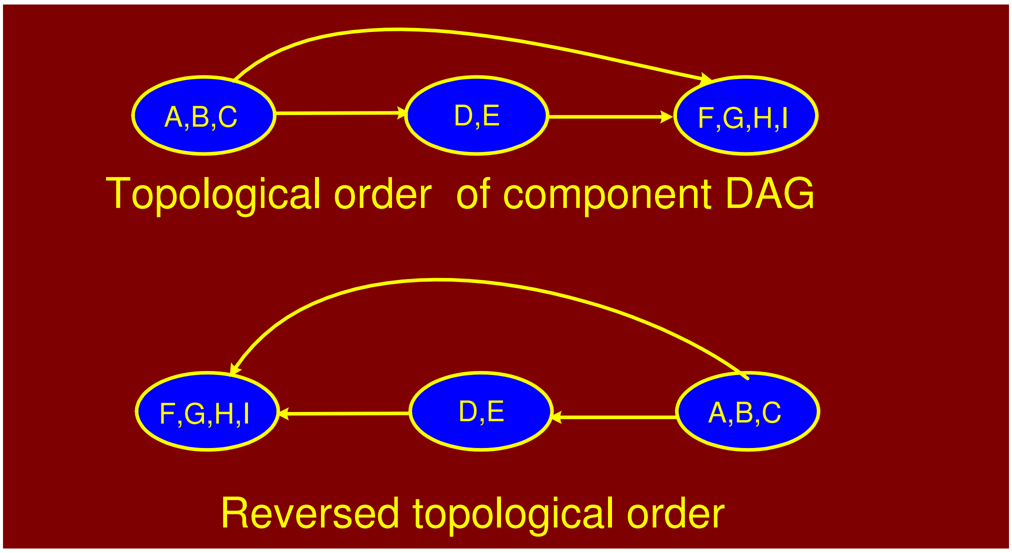

A,B,C
D,E
F,G,H,I
Topological order of component DAG
F,G,H,I
D,E
A,B,C
Reversed topological order

Figure 8.37: Reversed topological sort of component DAG

Now, run DFS, but every time you need a new vertex to start the search from, select the next available vertex according to this reverse topological order of the component digraph. Here is an informal justification. Clearly once the DFS starts within a given strong component, it must visit every vertex within the component (and possibly some others) before finishing. If we do not start in reverse topological, then the search may “leak out” into other strong components, and put them in the same DFS tree. For example, in the Figure 8.36, when the search is started at vertex ‘a’, not only does it visit its component with ‘b’ and ‘c’, but it also visits the other components as well. However, by visiting components in reverse topological order of the component tree, each search cannot “leak out” into other components, because other components would have already have been visited earlier in the search.

This leaves us with the intuition that if we could somehow order the DFS, so that it hits the strong components according to a reverse topological order, then we would have an easy algorithm for computing strong components. However, we do not know what the component DAG looks like. (After all, we are trying to solve the strong component problem in the first place). The trick behind the strong component algorithm is that we can find an ordering of the vertices that has essentially the necessary property, without actually computing the component DAG.

We will discuss the algorithm without proof. Define G^T to be the digraph with the same vertex set at G but in which all edges have been reversed in direction. This is shown in Figure 8.38. Given an adjacency list for G, it is possible to compute G^T in $\Theta(V + E)$ time.

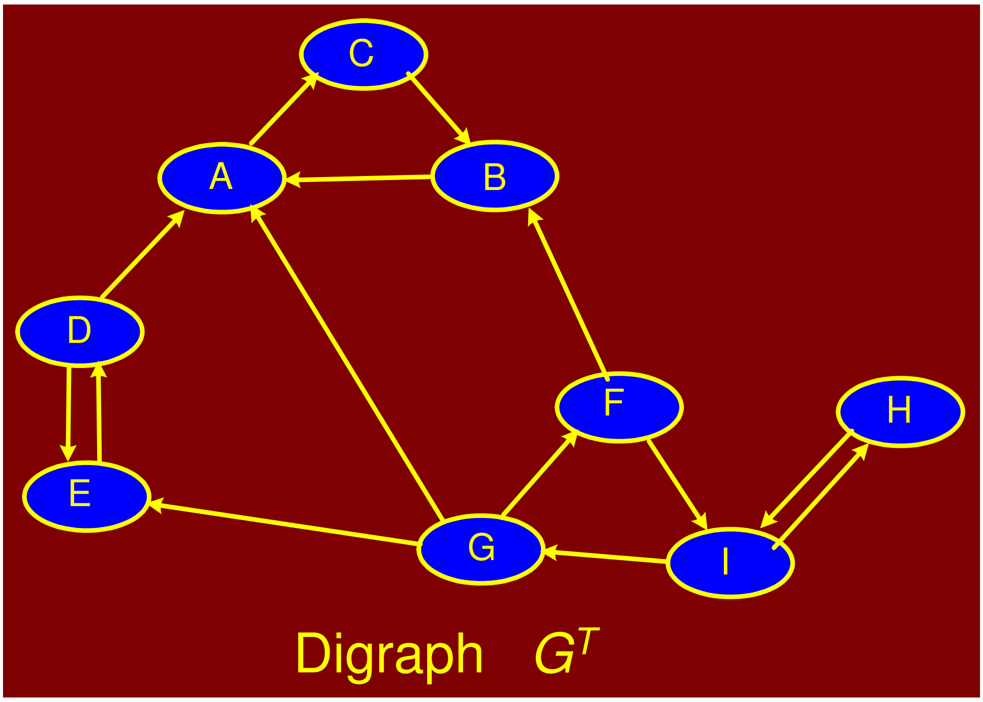
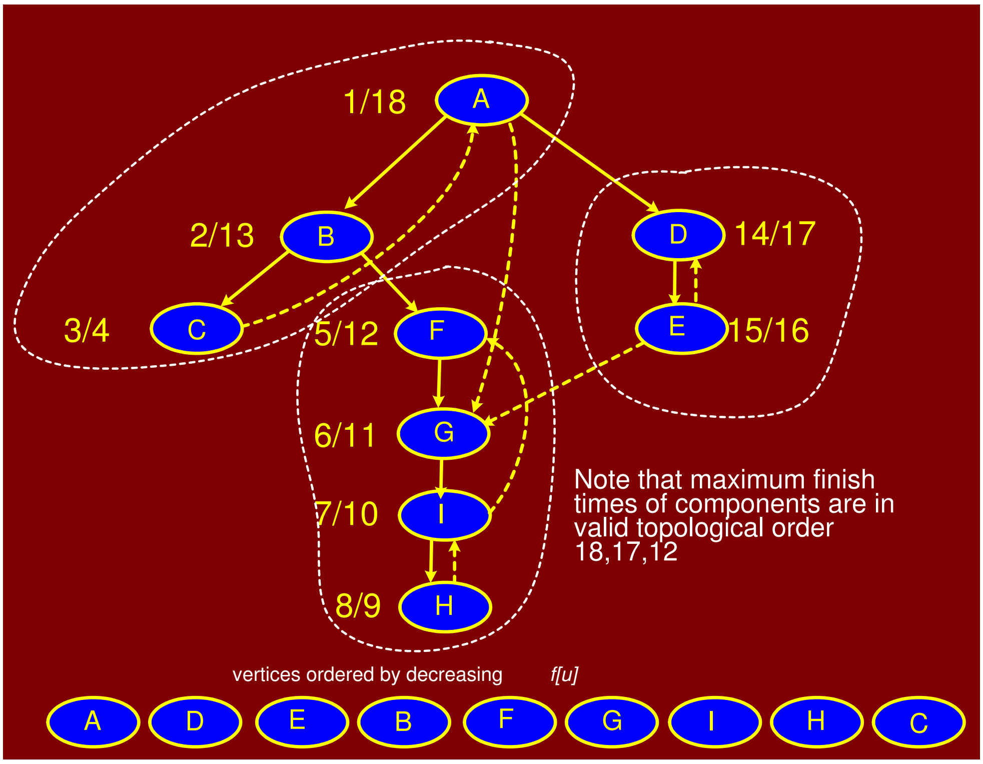


Figure 8.38: The digraph G^T

Observe that the strongly connected components are not affected by reversal of all edges. If u and v are mutually reachable in G, then this is certainly true in G^T. All that changes is that the component DAG is completely reversed. The ordering trick is to order the vertices of G according to their finish times in a DFS. Then visit the nodes of G^T in decreasing order of finish times. All the steps of the algorithm are quite easy to implement, and all operate in $\Theta(V + E)$ time. Here is the algorithm:

```
STRONGCOMPONENTS(G)
1   Run DFS(G) computing finish times $f[u]$
2   Compute $G^T$
3   Sort vertices of $G^T$ in decreasing $f[u]$
4   Run DFS($G^T$) using this order
5   Each DFS tree is a strong component
```

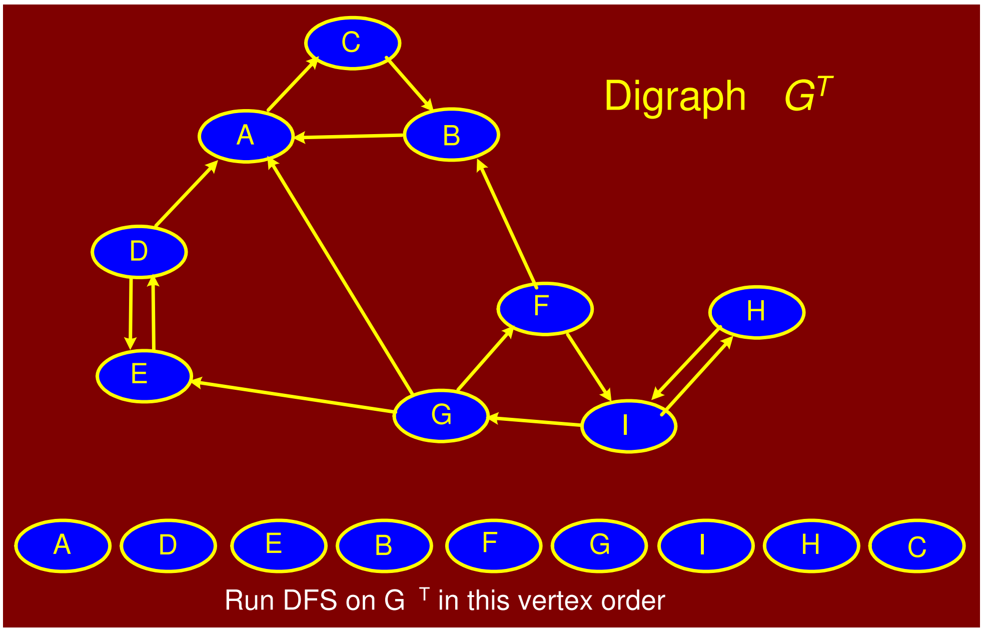
The execution of the algorithm is illustrated in Figures 8.39, 8.40 and 8.41.



Note that maximum finish times of components are in valid topological order 18,17,12

vertices ordered by decreasing *f[u]*

Figure 8.39: DFS of digraph with vertices in descending order by finish times



Digraph G^T

Run DFS on G^T in this vertex order

Figure 8.40: Digraph G^T and the vertex order for DFS

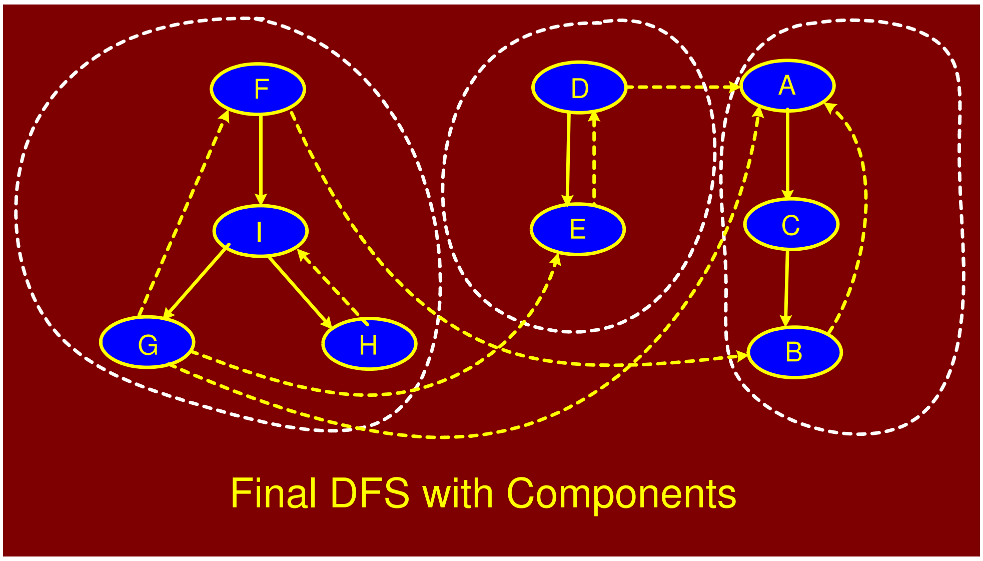


Figure 8.41: Final DFS with strong components of G^T

The complete proof for why this algorithm works is in CLR. We will discuss the intuition behind why the algorithm visits vertices in decreasing order of finish times and why the graph is revered. Recall that the main intent is to visit the strong components in a reverse topological order. The problem is how to order the vertices so that this is true. Recall from the topological sorting algorithm, that in a DAG, finish times occur in reverse topological order (i.e., the first vertex in the topological order is the one with the highest finish time). So, if we wanted to visit the components in reverse topological order, this suggests that we should visit the vertices in increasing order of finish time, starting with the lowest finishing time.

This is a good starting idea, but it turns out that it doesn't work. The reason is that there are many vertices in each strong component, and they all have different finish times. For example, in Figure 8.36, observe that in the first DFS, the lowest finish time (of 4) is achieved by vertex 'c', and its strong component is first, not last, in topological order.

However, there is something to notice about the finish times. If we consider the *maximum finish time* in each component, then these are related to the topological order of the component graph. In fact it is possible to prove the following (but we won't).

Lemma: Consider a digraph G on which DFS has been run. Label each component with the maximum finish time of all the vertices in the component, and sort these in decreasing order. Then this order is a topological order for the component digraph.

For example, in Figure 8.36, the maximum finish times for each component are 18 (for $\{a, b, c\}$), 17 (for $\{d,e\}$), and 12 (for $\{f,g,h,i\}$). The order (18, 17, 12) is a valid topological order for the component digraph. The problem is that this is not what we wanted. We wanted a reverse topological order for the component digraph. So, the final trick is to reverse the digraph. This does not change the component graph, but it reverses the topological order, as desired.

8.5 Minimum Spanning Trees

A common problem is communications networks and circuit design is that of connecting together a set of nodes by a network of total minimum length. The length is the sum of lengths of connecting wires. Consider, for example, laying cable in a city for cable t.v.

The computational problem is called the *minimum spanning tree* (MST) problem. Formally, we are given a connected, undirected graph $G = (V, E)$ Each edge (u, v) has numeric weight of cost. We define the cost of a spanning tree T to be the sum of the costs of edges in the spanning tree

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

A minimum spanning tree is a tree of minimum weight.

Figures **??**, **??** and **??** show three spanning trees for the same graph. The first is a spanning tree but is not a MST; the other two are.

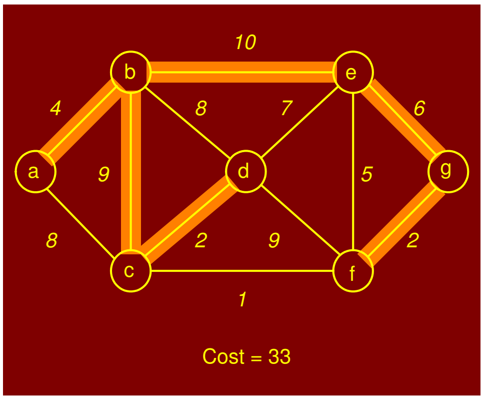


Figure 8.42: A spanning tree that is *not* MST

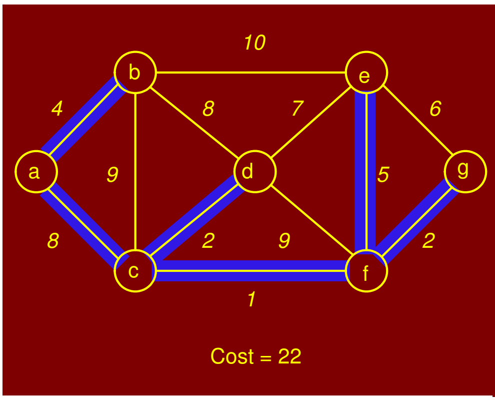


Figure 8.43: A minimum spanning tree

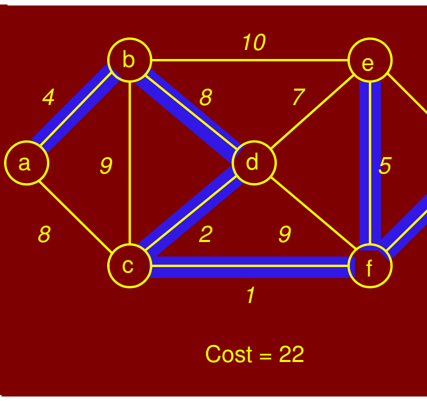


Figure 8.44: Another minimum spanning tree

We will present two *greedy* algorithms (Kruskal's and Prim's) for computing MST. Recall that a greedy algorithm is one that builds a solution by repeatedly selecting the cheapest among all options at each stage. Once the choice is made, it is never undone.

Before presenting the two algorithms, let us review facts about *free trees*. A free tree is a tree with no vertex designated as the root vertex. A free tree with n vertices has exactly $n - 1$ edges. There exists a unique path between any two vertices of a free tree. Adding any edge to a free tree creates a unique cycle. Breaking any edge on this cycle restores the free tree. This is illustrated in Figure 8.45. When the edges (b, e) or (b, d) are added to the free tree, the result is a cycle.

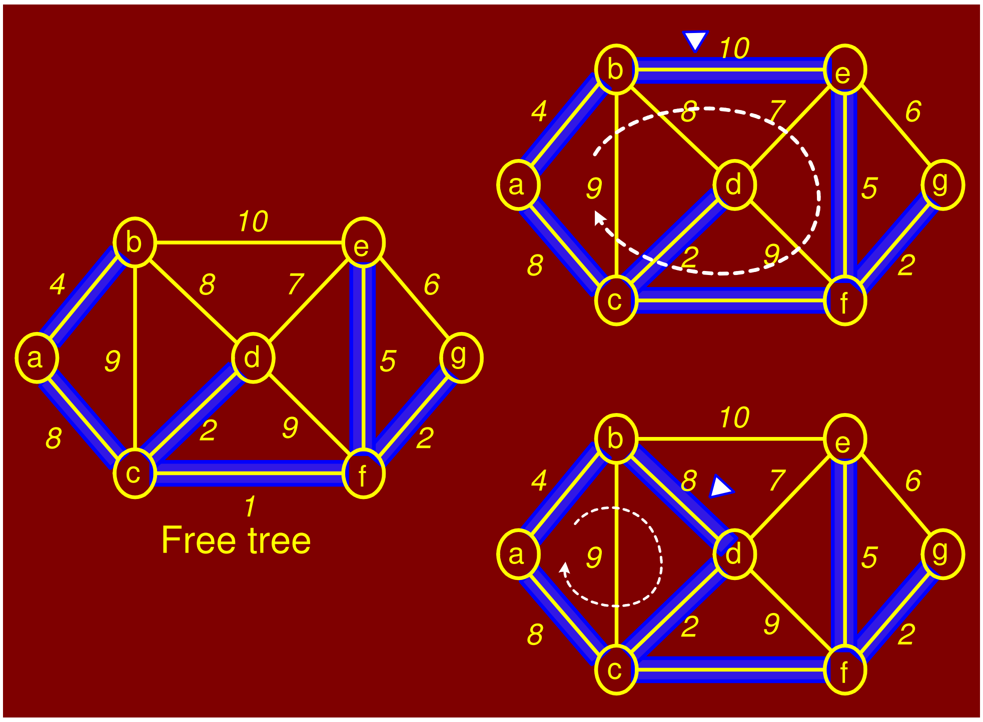


Figure 8.45: Free tree facts

8.5.1 Computing MST: Generic Approach

Let $G = (V, E)$ be an undirected, connected graph whose edges have numeric weights. The intuition behind greedy MST algorithm is simple: we maintain a subset of edges E of the graph . Call this subset A. Initially, A is empty. We will add edges one at a time until A equals the MST.

A subset $A \subseteq E$ is *viable* if A is a subset of edges of *some* MST. An edge $(u, v) \in E - A$ is *safe* if $A \cup \{(u, v)\}$ is viable. In other words, the choice (u, v) is a safe choice to add so that A can still be extended to form a MST.

Note that if A is viable, it cannot contain a cycle. A generic greedy algorithm operates by repeatedly adding any *safe* edge to the current spanning tree.

When is an edge safe? Consider the theoretical issues behind determining whether an edge is safe or not. Let S be a subset of vertices $S \subseteq V$. A *cut* $(S, V - S)$ is just a partition of vertices into two disjoint subsets. An edge (u, v) *crosses* the cut if one endpoint is in S and the other is in $V - S$.

Given a subset of edges A, a cut *respects* A if no edge in A crosses the cut. It is not hard to see why respecting cuts are important to this problem. If we have computed a partial MST and we wish to know which edges can be added that *do not* induce a cycle in the current MST, any edge that crosses a respecting cut is a possible candidate.

8.5.2 Greedy MST

An edge of E is a *light edge* crossing a cut if among all edges crossing the cut, it has the minimum weight. Intuition says that since all the edges that cross a respecting cut do not induce a cycle, then the lightest edge crossing a cut is a natural choice. The main theorem which drives both algorithms is the following:

MST Lemma: Let $G = (V, E)$ be a connected, undirected graph with real-valued weights on the edges. Let A be a viable subset of E (i.e., a subset of some MST). Let $(S, V - S)$ be any cut that respects A and let (u, v) be a light edge crossing the cut. Then the edge (u, v) is *safe* for A. This is illustrated in Figure 8.46.

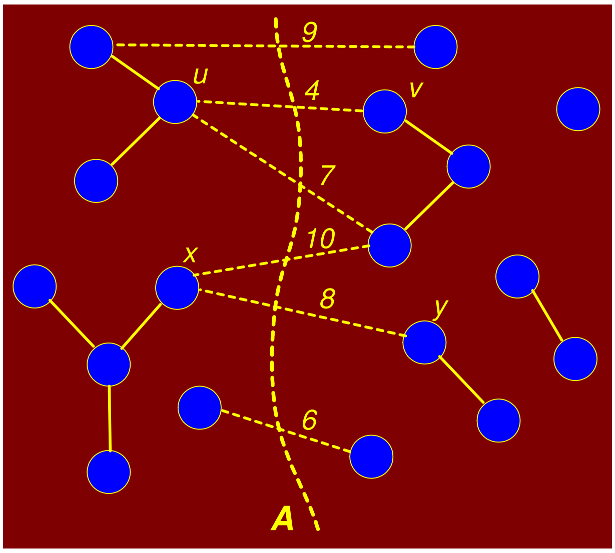


Figure 8.46: Subset A with a cut (wavy line) that respects A

MST Proof: It would simplify the proof if we assume that all edge weights are distinct. Let T be any MST for G. If T contains (u, v) then we are done. This is shown in Figure 8.47 where the lightest edge (u, v) with cost 4 has been chosen.

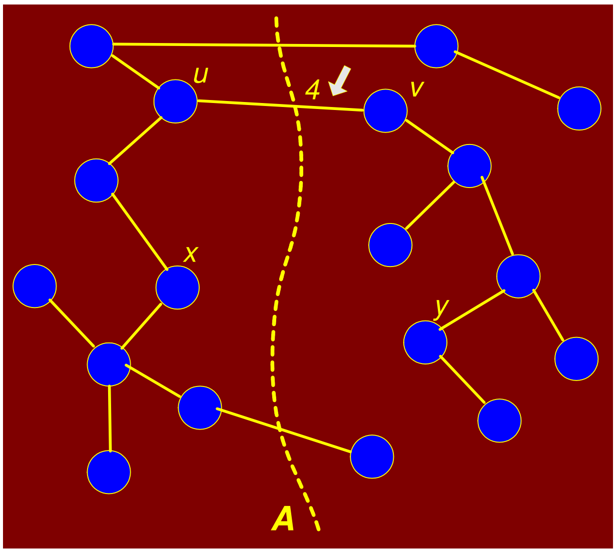


Figure 8.47: MST T which contains light edge (u, v)

Suppose no MST contains (u, v). Such a tree is shown in Figure 8.48. We will derive a contradiction.

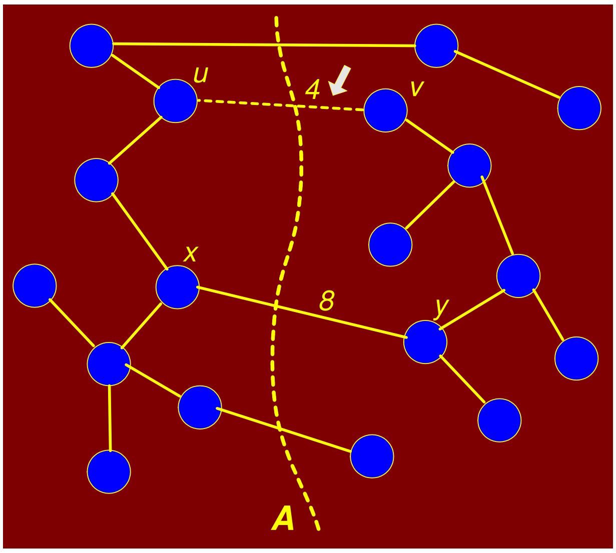


Figure 8.48: MST T which *does not* contains light edge (u, v)

Add (u, v) to T thus creating a cycle as illustrated in Figure 8.49.

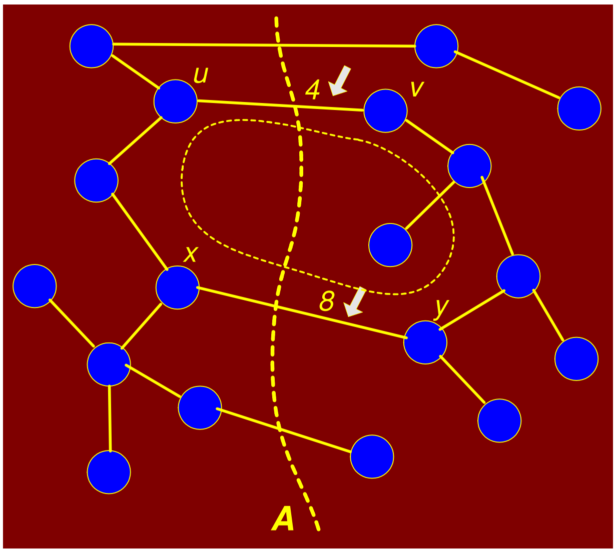


Figure 8.49: Cycle created due to $T + (u, v)$

Since u and v are on opposite sides of the cut, and any cycle must cross the cut an even number of times, there must be at least one other edge (x, y) in T that crosses the cut. The edge (x, y) is not in A because the cut respects A. By removing (x, y) we restore a spanning tree, call it T'. This is shown in Figure 8.50

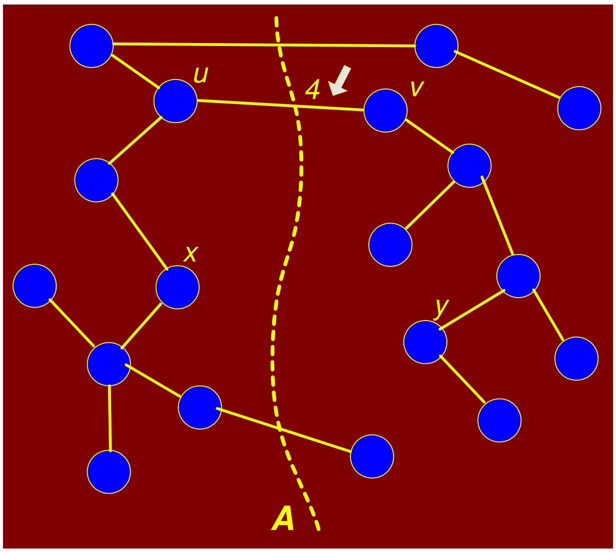


Figure 8.50: Tree $T' = T - (x, y) + (u, v)$

We have $w(T') = w(T) - w(x, y) + w(u, v)$. Since (u, v) is the lightest edge crossing the cut we have $w(u, v) < w(x, y)$. Thus $w(T') < w(T)$ which contradicts the assumption that T was an MST.

8.5.3 Kruskal’s Algorithm

Kruskal’s algorithm works by adding edges in increasing order of weight (lightest edge first). If the next edge does not induce a cycle among the current set of edges, then it is added to A. If it does, we skip it and consider the next in order. As the algorithm runs, the edges in A induce a *forest* on the vertices. The trees of this forest are eventually merged until a single tree forms containing all vertices.

The tricky part of the algorithm is how to detect whether the addition of an edge will create a cycle in A. Suppose the edge being considered has vertices (u, v). We want a fast test that tells us whether u and v are in the same tree of A. This can be done using the *Union-Find* data structure which supports the following $O(\log n)$ operations:

Create-set(u): Create a set containing a single item u.

Find-set(u): Find the set that contains u

Union(u,v): merge the set containing u and set containing v into a common set.

In Kruskal’s algorithm, the vertices will be stored in sets. The vertices in each tree of A will be a set. The edges in A can be stored as a simple list. Here is the algorithm: Figures 8.51 through **??** demonstrate the algorithm applied to a graph.

```
KRUSKAL$(G = (V, E))$
$A \leftarrow \{\}$
for ( each $u \in V$)
do $create\_set(u)$
sort $E$ in increasing order by weight $w$
for ( each $(u, v)$ in sorted edge list)
do if $(find(u) \neq find(v))$
      then add $(u, v)$ to $A$
           $union(u, v)$
return $A$
```

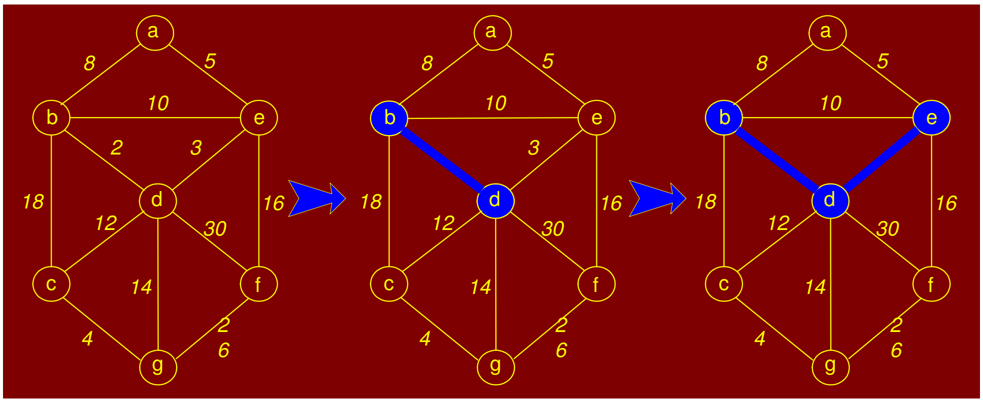


Figure 8.51: Kruskal algorithm: (b, d) and (d, e) added

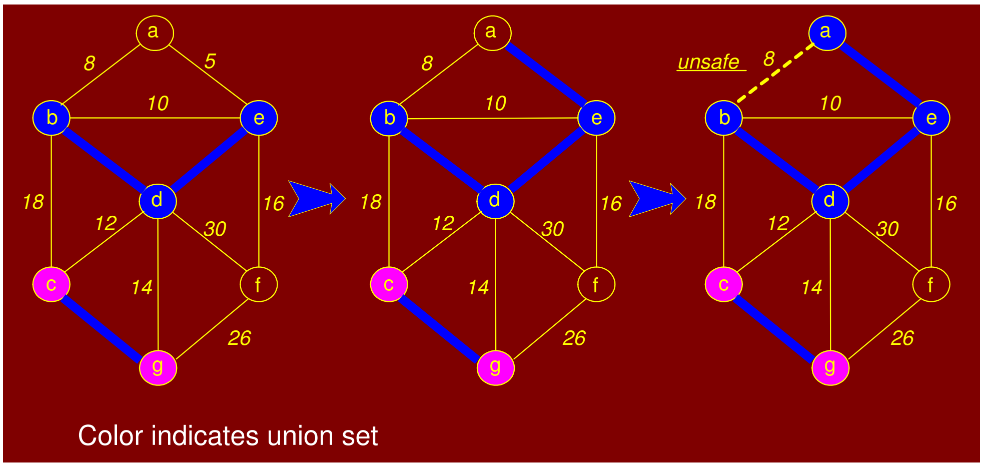


Figure 8.52: Kruskal algorithm: (c, g) and (a, e) added

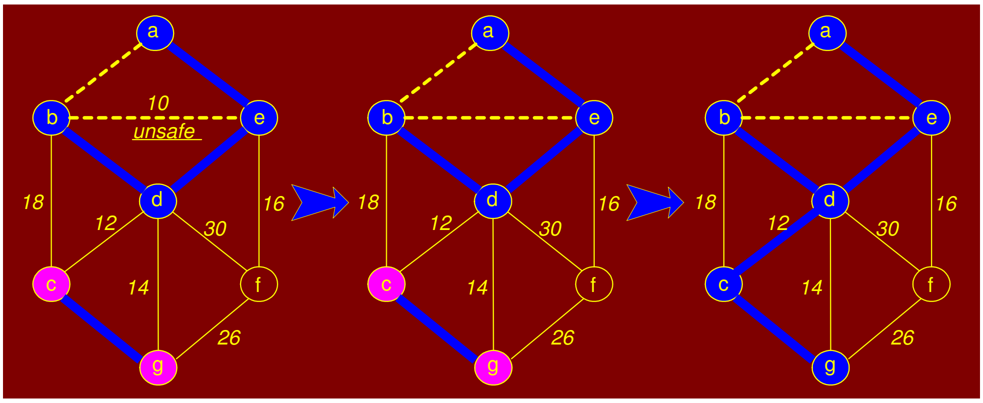


Figure 8.53: Kruskal algorithm: unsafe edges

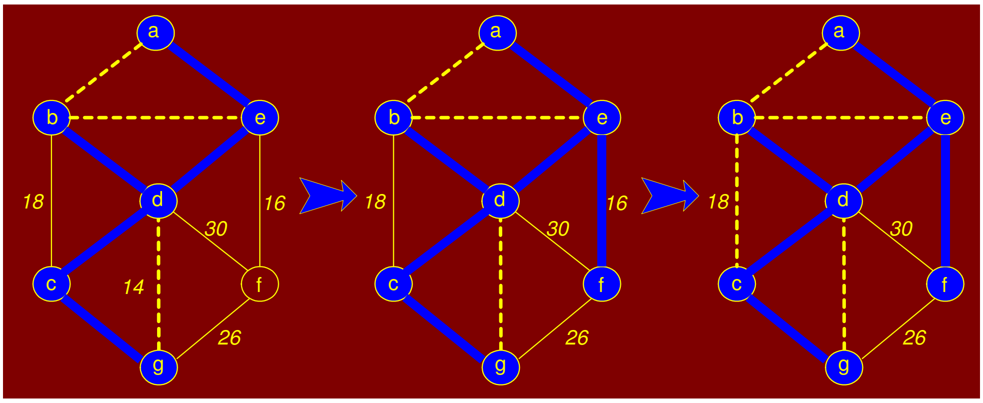


Figure 8.54: Kruskal algorithm: (e, f) added

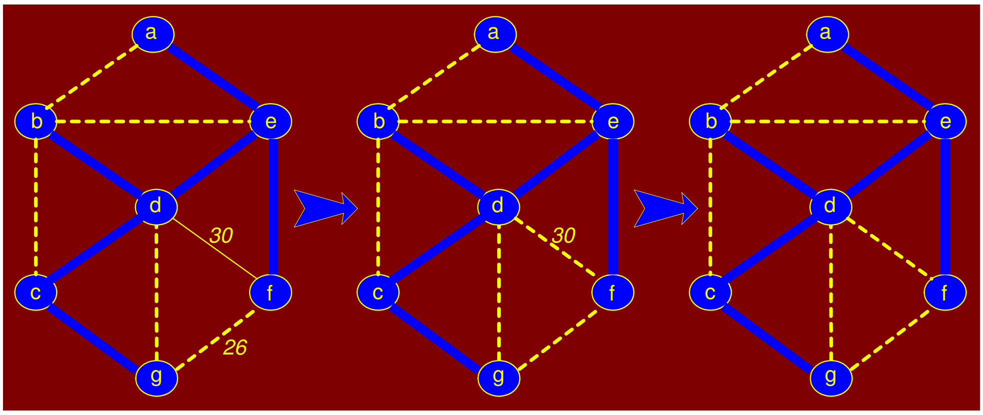


Figure 8.55: Kruskal algorithm: more unsafe edges and final MST

Analysis:

Since the graph is connected, we may assume that $E \geq V - 1$. Sorting edges (*line 4*) takes $\Theta(E \log E)$. The for loop (*line 5*) performs $O(E)$ $\mathtt{find}$ and $O(V)$ $\mathtt{union}$ operations. Total time for $\mathtt{union} - \mathtt{find}$ is $O(E\alpha(V))$ where $\alpha(V)$ is the inverse Ackerman function. $\alpha(V) < 4$ for V less the number of atoms in the entire universe. Thus the time is dominated by sorting. Overall time for Kruskal is $\Theta(E \log E) = \Theta(E \log V)$ if the graph is sparse.

8.5.4 Prim's Algorithm

Kruskal's algorithm worked by ordering the edges, and inserting them one by one into the spanning tree, taking care never to introduce a cycle. Intuitively Kruskal's works by merging or splicing two trees together, until all the vertices are in the same tree.

In contrast, Prim's algorithm builds the MST by adding leaves one at a time to the current tree. We start with a root vertex r; it can be any vertex. At any time, the subset of edges A forms a single tree (in Kruskal's, it formed a forest). We look to add a single vertex as a leaf to the tree.

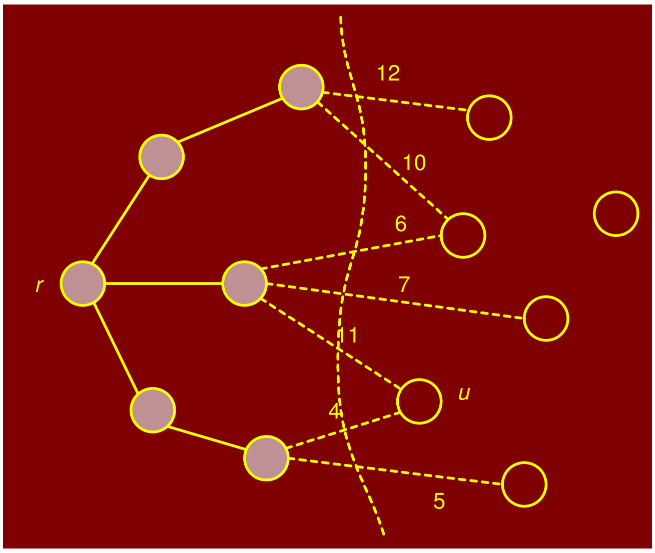


Figure 8.56: Prim's algorithm: a *cut* of the graph

Consider the set of vertices S currently part of the tree and its complement $(V - S)$ as shown in Figure 8.56. We have *cut* of the graph. Which edge should be added next? The greedy strategy would be to add the lightest edge which in the figure is edge to 'u'. Once u is added, Some edges that crossed the cut are no longer crossing it and others that were not crossing the cut are as shown in Figure 8.57

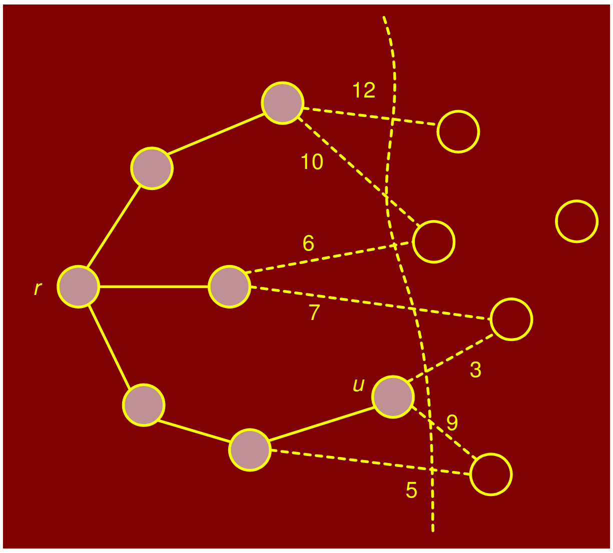


Figure 8.57: Prim's algorithm: u selected

We need an efficient way to update the cut and determine the light edge quickly. To do this, we will make use of a *priority queue*. The question is what do we store in the priority queue? It may seem logical that edges that cross the cut should be stored since we choose light edges from these. Although possible, there is more elegant solution which leads to a simpler algorithm.

For each vertex $u \in (V - S)$ (not part of the current spanning tree), we associate a key $\mathtt{key}[u]$. The $\mathtt{key}[u]$ is the weight of the lightest edge going from u to any vertex in S. If there is no edge from u to a vertex in S, we set the key value to ∞. We also store in $\mathtt{pred}[u]$ the end vertex of this edge in S. We will also need to know which vertices are in S and which are not. To do this, we will assign a color to each vertex. If the color of a vertex is black then it is in S otherwise not. Here is the algorithm:

```
PRIM($(G, w, r)$)
  for ( each  $u \in V$)
  do  key $[u] \leftarrow \infty$;   pq.insert $(u, \mathtt{key}[u])$
        color $[u] \leftarrow$ white
   key $[r] \leftarrow 0$;   pred $[r] \leftarrow \mathtt{nil}$;  pq.decrease_key $(r,$ key $[r])$;
  while ( pq.not_empty ())
  do $u \leftarrow$ pq.extract_min ()
      for ( each  $u \in$ adj $[u]$)
      do if ( color $[v] ==$ white $)\mathtt{and}($ w $(u, v) <$ key $[v])$
              then  key $[v] =$ w $(u, v)$
                    pq.decrease_key $(v,$ key $[v])$
                    pred $[v] = u$
        color $[u] =$ black
```

Figures 8.58 through 8.60 illustrate the algorithm applied to a graph. The contents of the priority queue are shown as the algorithm progresses. The arrows indicate the predecessor pointers and the numeric labels in each vertex is its key value.

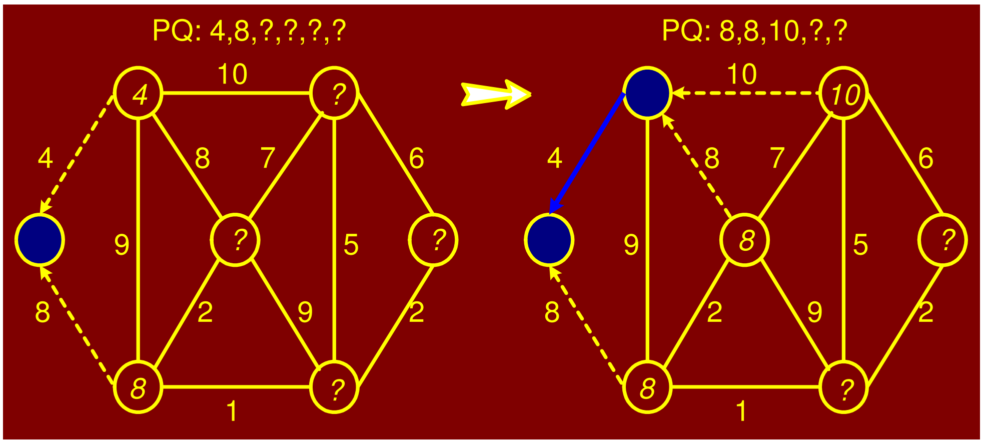


Figure 8.58: Prim's algorithm: edge with weight 4 selected

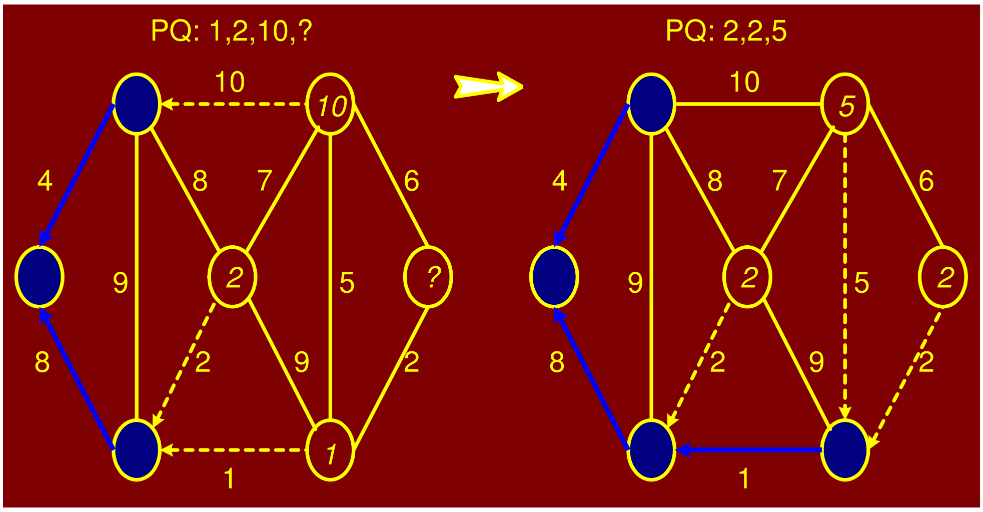


Figure 8.59: Prim's algorithm: edges with weights 8 and 1 selected

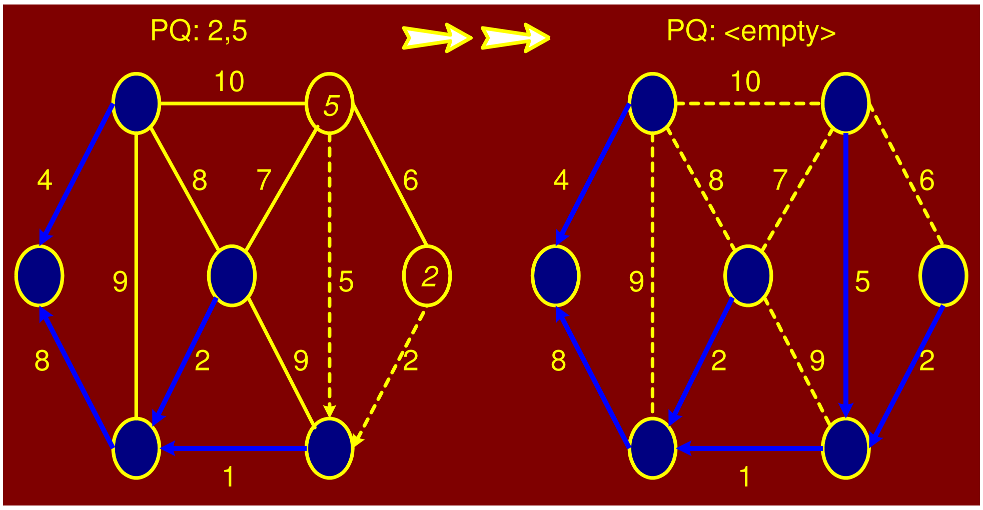


Figure 8.60: Prim's algorithm: the final MST

Analysis:

It takes $O(\log V)$ to extract a vertex from the priority queue. For each incident edge, we spend potentially $O(\log V)$ time decreasing the key of the neighboring vertex. Thus the total time is $O(\log V + \deg(u) \log V)$. The other steps of update are constant time.

So the overall running time is

$$\begin{aligned} T(V, E) &= \sum_{u \in V} (\log V + \deg(u) \log V) \\ &= \log V \sum_{u \in V} (1 + \deg(u)) \\ &= (\log V)(V + 2E) = \Theta((V + E) \log V) \end{aligned}$$

Since G is connected, V is asymptotically no greater than E so this is $\Theta(E \log V)$, same as Kruskal's algorithm.

8.6 Shortest Paths

A motorist wishes to find the shortest possible route between Peshawar and Karachi. Given a road map of Pakistan on which the distance between each pair of adjacent cities is marked Can the motorist determine the shortest route?

In the *shortest-paths problem* We are given a weighted, directed graph $G = (V, E)$ The weight of path $p =< v_0, v_1, \ldots, v_k >$ is the sum of the constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the *shortest-path weight* from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Problems such as shortest route between cities can be solved efficiently by modelling the road map as a graph. The nodes or vertices represent cities and edges represent roads. Edge weights can be interpreted as distances. Other metrics can also be used, e.g., time, cost, penalties and loss.

Similar scenarios occur in computer networks like the Internet where data packets have to be routed. The vertices are *routers*. Edges are communication links which may be be wire or wireless. Edge weights can be distance, link speed, link capacity link delays, and link utilization.

The breadth-first-search algorithm we discussed earlier is a shortest-path algorithm that works on un-weighted graphs. An un-weighted graph can be considered as a graph in which every edge has weight one unit.

There are a few variants of the shortest path problem. We will cover their definitions and then discuss algorithms for some.

Single-source shortest-path problem: Find shortest paths from a given (single) *source* vertex $s \in V$ to every other vertex $v \in V$ in the graph G.

Single-destination shortest-paths problem: Find a shortest path to a given destination vertex t from each vertex v. We can reduce the this problem to a single-source problem by reversing the direction of each edge in the graph.

Single-pair shortest-path problem: Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also. No algorithms for this problem are known to run asymptotically faster than the best single-source algorithms in the worst case.

All-pairs shortest-paths problem: Find a shortest path from u to v for *every pair* of vertices u and v. Although this problem can be solved by running a single-source algorithm once from each vertex, it can usually be solved faster.

8.6.1 Dijkstra's Algorithm

Dijkstra's algorithm is a simple *greedy* algorithm for computing the **single-source shortest-paths** to all other vertices. Dijkstra's algorithm works on a weighted directed graph $G = (V, E)$ in which all edge weights are non-negative, i.e., $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Negative edges weights maybe counter to intuition but this can occur in real life problems. However, we will *not allow negative cycles* because then there is no shortest path. If there is a negative cycle between, say, s and t, then we can always find a shorter path by going around the cycle one more time.

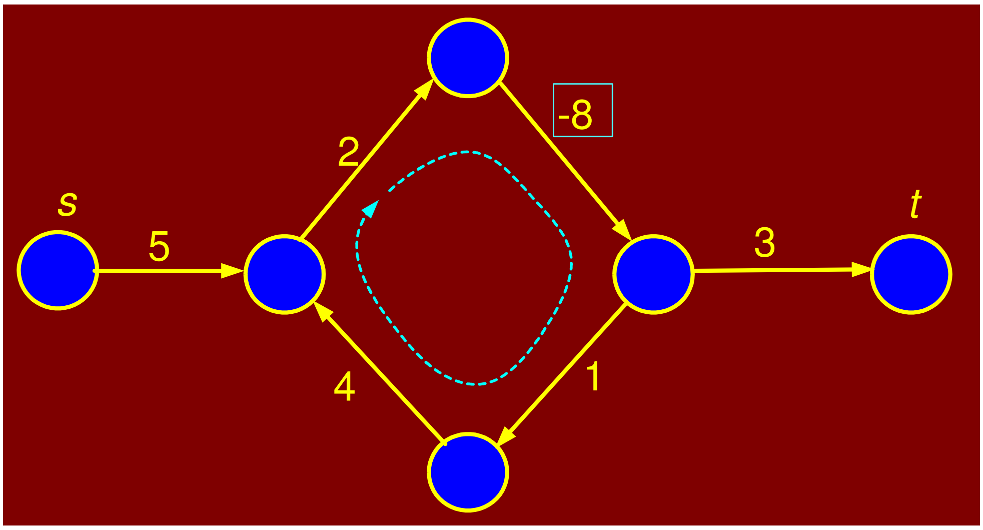


Figure 8.61: Negative weight cycle

The basic structure of Dijkstra's algorithm is to maintain an *estimate* of the shortest path from the source vertex to each vertex in the graph. Call this estimate $d[v]$. Intuitively, $d[v]$ will the length of the shortest path *that the algorithm knows of* from s to v. This value will always be greater than or equal to the true shortest path distance from s to v. I.e., $d[v] \geq \delta(u, v)$. Initially, we know of no paths, so $d[v] = \infty$. Moreover, $d[s] = 0$ for the source vertex.

As the algorithm goes on and sees more and more vertices, it attempts to update $d[v]$ for each vertex in the graph. The process of updating estimates is called *relaxation*. Here is how relaxation works.

Intuitively, if you can see that your solution is not yet reached an optimum value, then push it a little closer to the optimum. In particular, if you discover a path from s to v shorter than $d[v]$, then you need to update $d[v]$. This notion is common to many optimization algorithms.

Consider an edge from a vertex u to v whose weight is $w(u, v)$. Suppose that we have already computed current estimates on $d[u]$ and $d[v]$. We know that there is a path from s to u of weight $d[u]$. By taking this path and following it with the edge (u, v) we get a path to v of length $d[u] + w(u, v)$. If this path is better than the existing path of length $d[v]$ to v, we should take it. The relaxation process is illustrated in the following figure. We should also remember that the shortest way back to the source is through u by updating the predecessor pointer.

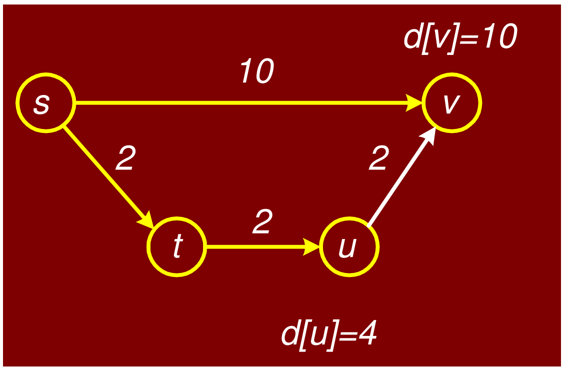


Figure 8.62: Vertex u relaxed

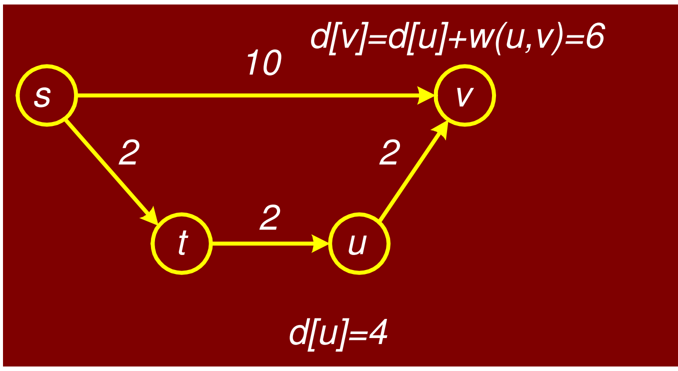


Figure 8.63: Vertex v relaxed

```
RELAX($(u, v)$)
if ($d[u] + w(u, v) < d[v]$)
   then $d[v] \leftarrow d[u] + w(u, v)$
        $\mathtt{pred}[v] = u$
```

Observe that whenever we set $d[v]$ to a finite value, there is always evidence of a path of that length. Therefore $d[v] \geq \delta(s, v)$. If $d[v] = \delta(s, v)$, then further relaxations cannot change its value.

It is not hard to see that if we perform RELAX(U,V) repeatedly over all edges of the graph, the $d[v]$ values will eventually converge to the final true distance value from s. The cleverness of any shortest path algorithm is to perform the updates in a judicious manner, so the convergence is as fast as possible.

Dijkstra's algorithm is based on the notion of performing repeated relaxations. The algorithm operates by maintaining a subset of vertices, $S \subseteq V$, for which we claim we *know* the true distance, $d[v] = \delta(s, v)$.

Initially $S = \emptyset$, the empty set. We set $d[u] = 0$ and all others to ∞. One by one we select vertices from $V - S$ to add to S.

How do we select which vertex among the vertices of $V - S$ to add next to S? Here is *greediness* comes in. For each vertex $u \in (V - S)$, we have computed a distance estimate $d[u]$.

The greedy thing to do is to take the vertex for which $d[u]$ is minimum, i.e., take the unprocessed vertex that is closest by our estimate to s. Later, we justify why this is the proper choice. In order to perform this selection efficiently, we store the vertices of $V - S$ in a *priority queue*.

```
DIJKSTRA$((G, w, s))$
for ( each $u \in V$)
do $d[u] \leftarrow \infty$
    pq.insert $(u, d[u])$
$d[s] \leftarrow 0$;  pred $[s] \leftarrow nil$;  pq.decrease_key $(s, d[s])$;
while ( pq.not_empty ())
do $u \leftarrow$ pq.extract_min ()
    for ( each $v \in adj[u]$)
    do if $(d[u] + w(u, v) < d[v])$
          then $d[v] = d[u] + w(u, v)$
               pq.decrease_key $(v, d[v])$
               pred $[v] = u$
```

Note the similarity with Prim's algorithm, although a different key is used here. Therefore the running time is the same, i.e., $\Theta(E \log V)$.

Figures 8.64 through **??** demonstrate the algorithm applied to a directed graph with no negative weight edges.

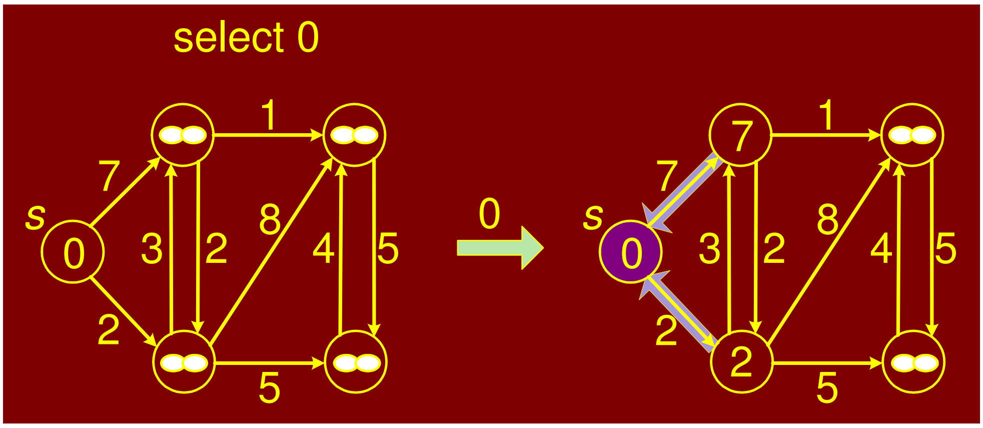


Figure 8.64: Dijkstra's algorithm: select 0

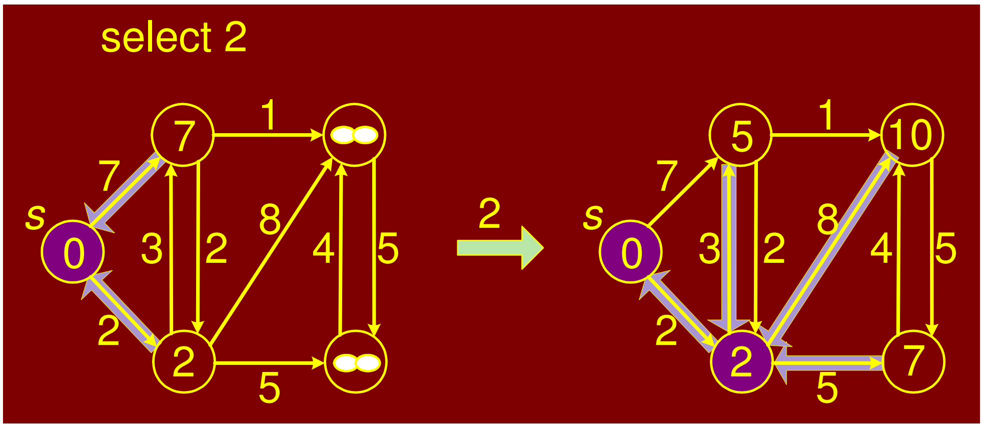


Figure 8.65: Dijkstra's algorithm: select 2

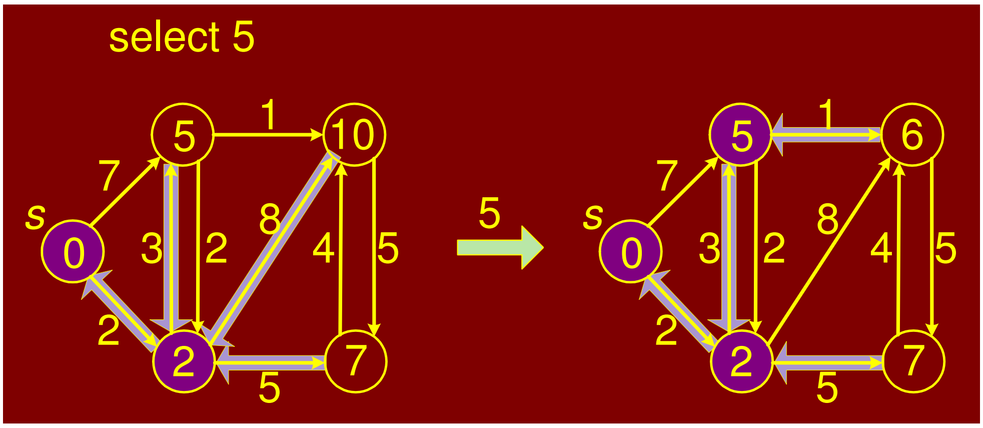


Figure 8.66: Dijkstra's algorithm: select 5

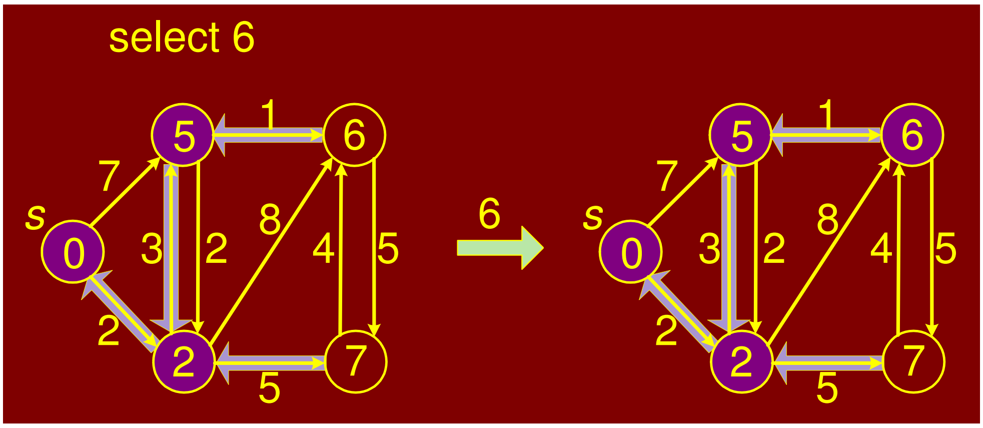


Figure 8.67: Dijkstra's algorithm: select 6

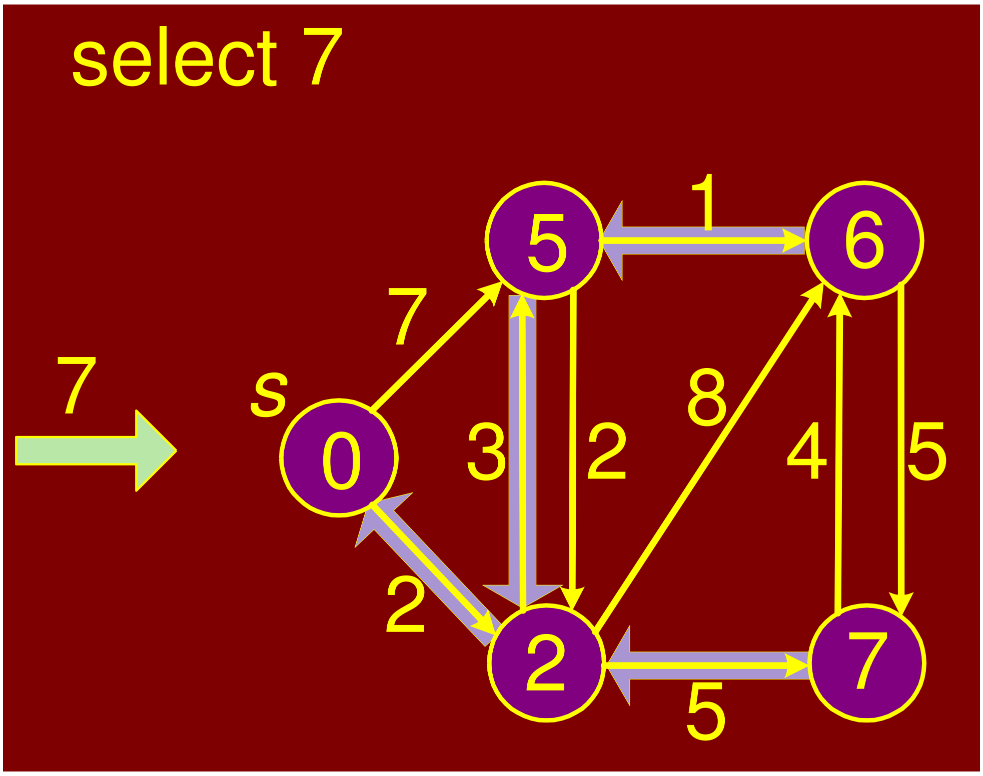


Figure 8.68: Dijkstra's algorithm: select 7

fig:dijlast

8.6.2 Correctness of Dijkstra's Algorithm

We will prove the correctness of Dijkstr's algorithm by Induction. We will use the definition that $\delta(s, v)$ denotes the minimal distance from s to v.

For the base case

1. $S = \{s\}$
2. $d(s) = 0$, which is $\delta(s, s)$

Assume that $d(v) = \delta(s, v)$ for every $v \in S$ and all neighbors of v have been relaxed. If $d(u) \leq d(u')$ for every $u' \in V$ then $d(u) = \delta(s, u)$, and we can transfer u from V to S, after which $d(v) = \delta(s, v)$ for every $v \in S$.

We do this as a proof by contradiction. Suppose that $d(u) > \delta(s, u)$. The shortest path from s to u, $p(s, u)$, must pass through one or more vertices exterior to S. Let x be the last vertex inside S and y be the first vertex outside S on this path to u. Then $p(s, u) = p(s, x) \cup \{(x, y)\} \cup p(y, u)$.

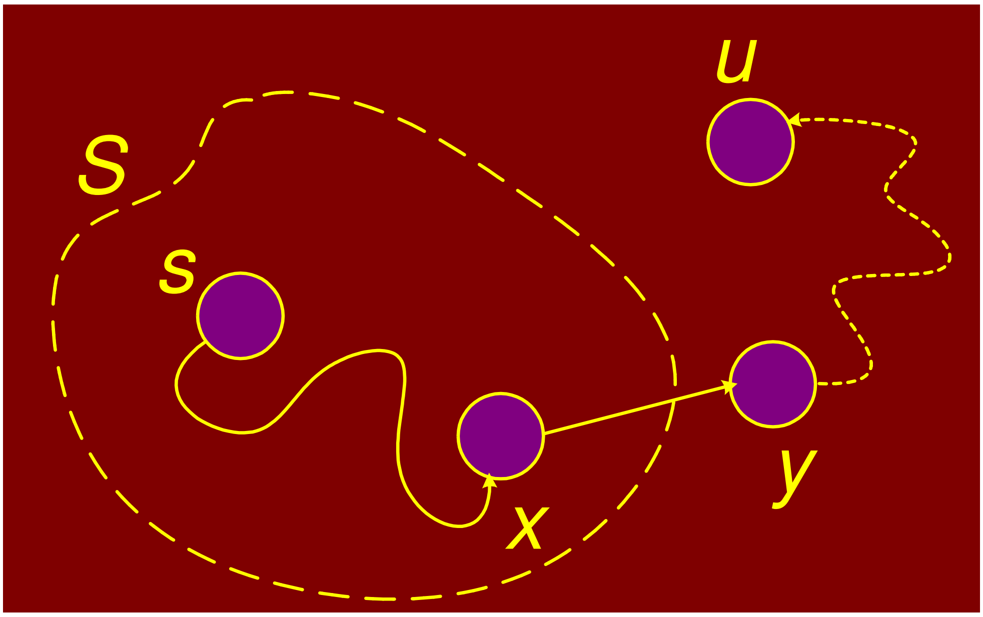


Figure 8.69: Correctness of Dijkstra's algorithm

The length of $p(s, u)$ is $\delta(s, u) = \delta(s, y) + \delta(y, u)$. Because y was relaxed when x was put into S, $d(y) = \delta(s, y)$ by the convergence property. Thus $d(y) \leq \delta(s, u) \leq d(u)$. But, because $d(u)$ is the smallest among vertices not in S, $d(u) \leq d(y)$. The only possibility is $d(u) = d(y)$, which requires $d(u) = \delta(s, u)$ contradicting the assumption.

By the upper bound property, $d(u) \geq \delta(s, u)$. Since $d(u) > \delta(s, u)$ is false, $d(u) = \delta(s, u)$, which is what we wanted to prove. Thus, if we follow the algorithm's procedure, transferring from V to S, the vertex in V with the smallest value of $d(u)$ then all vertices in S have $d(v) = \delta(s, v)$

8.6.3 Bellman-Ford Algorithm

Dijkstra's single-source shortest path algorithm works if all edges weights are non-negative and there are no negative cost cycles. Bellman-Ford *allows* negative weights edges and no negative cost cycles. The algorithm is slower than Dijkstra's, running in $\Theta(VE)$ time.

Like Dijkstra's algorithm, Bellman-Ford is based on performing repeated relaxations. Bellman-Ford applies relaxation to *every edge* of the graph and repeats this $V - 1$ times. Here is the algorithm; its is illustrated in Figure 8.70.

```
BELLMAN-FORD$(G, w, s)$
for ( each $u \in V$)
do $d[u] \leftarrow \infty$
    $pred[u] = nil$

$d[s] \leftarrow 0$;
for $i = 1$ to $V - 1$
do for ( each $(u, v)$ in $E$ )
    do RELAX$(u, v)$
```

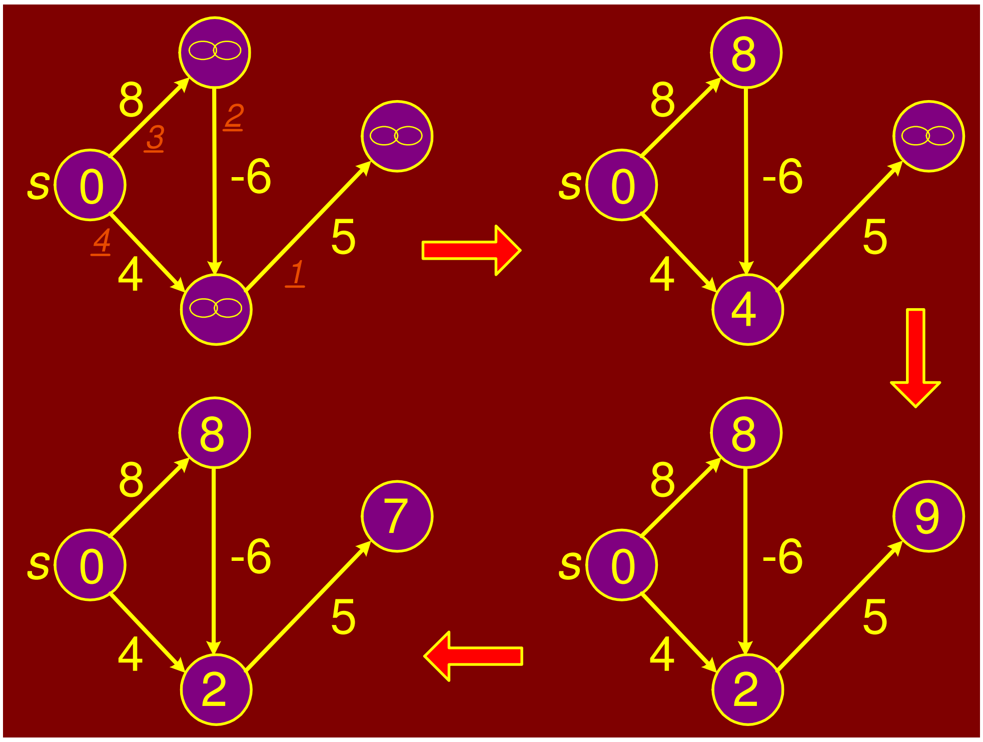


Figure 8.70: The Bellman-Ford algorithm

8.6.4 Correctness of Bellman-Ford

Think of Bellman-Ford as a sort of bubble-sort analog for shortest path. The shortest path information is propagated sequentially along each shortest path in the graph. Consider any shortest path from s to some other vertex u: $\langle v_0, v_1, \ldots, v_k \rangle$ where $v_0 = s$ and $v_k = u$.

Since a shortest path will never visit the same vertex twice, we know that $k \le V - 1$. Hence the path consists of at most $V - 1$ edges. Since this a shortest path, it is $\delta(s, v_i)$, the true shortest path cost from s

to v_i that satisfies the equation:

$$\delta(s, v_i) = \delta(s, v_{i-1}) + w(v_{i-1}, v_i)$$

Claim: We assert that after the i^{th} pass of the "`for-i`" loop, $d[v_i] = \delta(s, v_i)$.

Proof: The proof is by induction on i. Observe that after the initialization (pass 0), $d[v_1] = d[s] = 0$.

In general, prior to the i^{th} pass through the loop, the induction hypothesis tells us that $d[v_{i-1}] = \delta(s, v_{i-1})$. After the i^{th} pass, we have done relaxation on the edge (v_{i-1}, v_i) (since we do relaxation along all edges). Thus after the i^{th} pass we have

$$\begin{aligned} d[v_i] &\le d[v_{i-1}] + w(v_{i-1}, v_i) \\ &= \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \\ &= \delta(s, v_i) \end{aligned}$$

Recall from Dijkstra's algorithm that $d[v_i]$ is never less than $\delta(s, v_i)$. Thus, $d[v_i]$ is in fact equal to $\delta(s, v_i)$. This completes the induction proof.

In summary, after i passes through the `for` loop, all vertices that are i edges away along the shortest path tree from the source have the correct values stored in $d[u]$. Thus, after the $(V-1)^{\text{st}}$ iteration of the for loop, all vertices u have correct distance values stored in $d[u]$.

8.6.5 Floyd-Warshall Algorithm

We consider the generalization of the shortest path problem: to compute the shortest paths between all pairs of vertices. This is called the all-pairs shortest paths problem.

Let $G = (V, E)$ be a directed graph with edge weights. If $(u, v) \in E$ is an edge then $w(u, v)$ denotes its weight. $\delta(u, v)$ is the distance of the minimum cost path between u and v. We will allow G to have negative edges weights but will not allow G to have negative cost cycles. We will present an $\Theta(n^3)$ algorithm for the all pairs shortest path. The algorithm is called the *Floyd-Warshall algorithm* and is based on *dynamic programming*.

We will use an adjacency matrix to represent the digraph. Because the algorithm is matrix based, we will employ the common matrix notation, using i, j and k to denote vertices rather than u, v and w.

The input is an $n \times n$ matrix of edge weights:

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w(i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

The output will be an $n \times n$ distance matrix $D = d_{ij}$, where $d_{ij} = \delta(i, j)$, the shortest path cost from vertex i to j.

The algorithm dates back to the early 60's. As with other dynamic programming algorithms, the genius of the algorithm is in the clever recursive formulation of the shortest path problem. For a path $p = \langle v_1, v_2, \ldots, v_l$, we say that the vertices $v_2, v_3, \ldots, v_{l-1}$ are the *intermediate vertices* of this path.

Formulation: Define $d_{ij}^{(k)}$ to be the shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, \ldots, k\}$. The path is free to visit any subset of these vertices and in any order. How do we compute $d_{ij}^{(k)}$ assuming we already have the previous matrix $d^{(k-1)}$? There are two basic cases:

1. Don't go through vertex k at all.

2. Do go through vertex k.

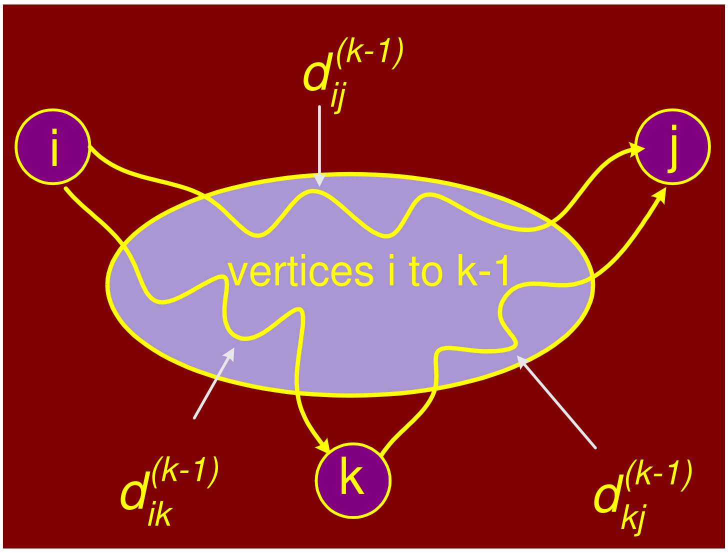


Figure 8.71: Two cases for all-pairs shortest path

Don't go through k at all

Then the shortest path from i to j uses only intermediate vertices $\{1, 2, \ldots, k-1\}$. Hence the length of the shortest is $d_{ij}^{(k-1)}$

Do go through k

First observe that a shortest path does not go through the same vertex twice, so we can assume that we pass through k exactly once. That is, we go from i to k and then from k to j. In order for the overall path to be as short as possible, we should take the shortest path from i to k and the shortest path from k to j. Since each of these paths uses intermediate vertices $\{1, 2, \ldots, k-1\}$, the length of the path is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

The following illustrate the process in which the value of $d_{3,2}^{k}$ is updated as k goes from 0 to 4.

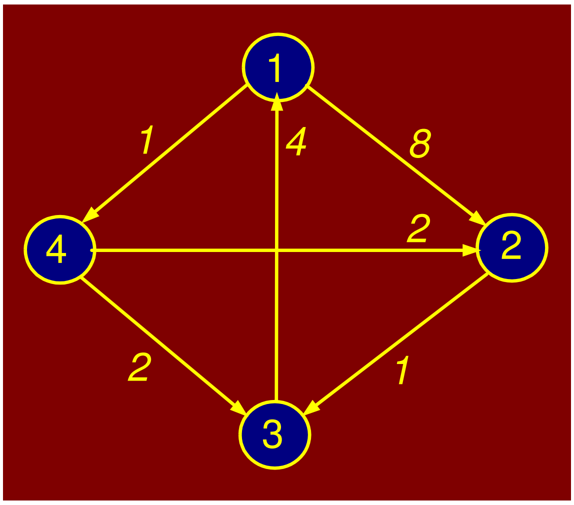


Figure 8.72: $k = 0,\ d^{(0)}_{3,2} = \infty$ (no path)

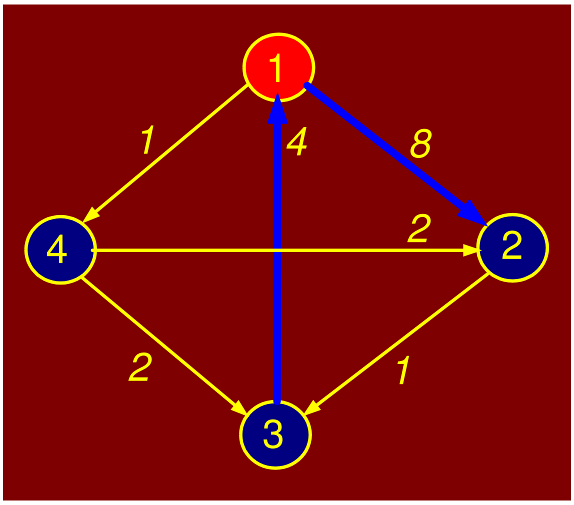


Figure 8.73: $k = 1,\ d^{(1)}_{3,2} = 12\ \ (3 \to 1 \to 2)$

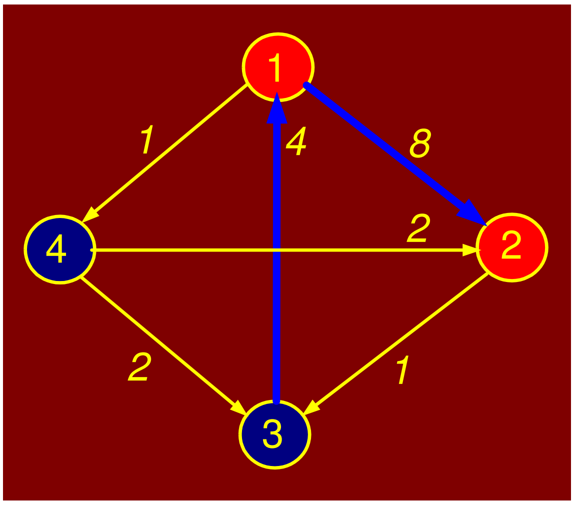


Figure 8.74: $k = 2,\ d^{(2)}_{3,2} = 12\ \ (3 \to 1 \to 2)$

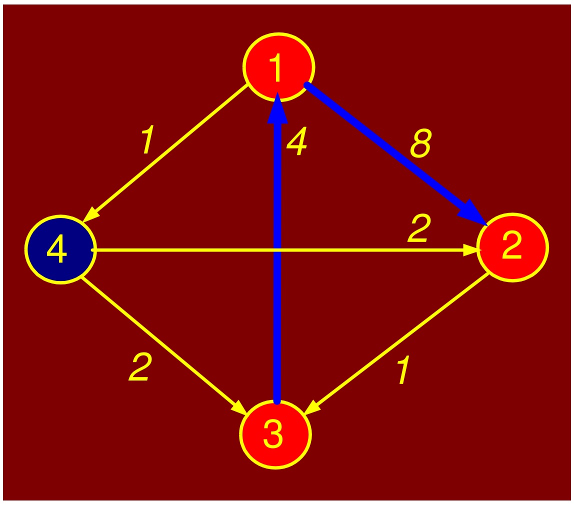


Figure 8.75: $k = 3,\ d^{(3)}_{3,2} = 12\ \ (3 \to 1 \to 2)$

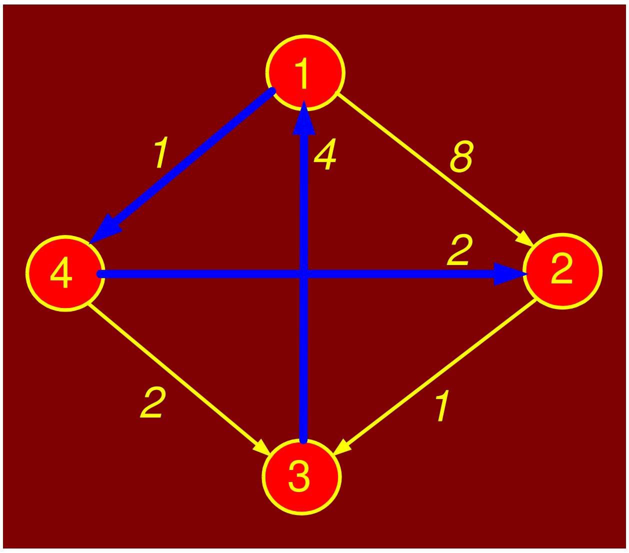


Figure 8.76: $k = 4,\ d^{(4)}_{3,2} = 7\ \ (3 \to 1 \to 4 \to 2)$

This suggests the following recursive (DP) formulation:

$$d^{(0)}_{ij} = w_{ij}$$
$$d^{(k)}_{ij} = \min\left(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}\right)$$

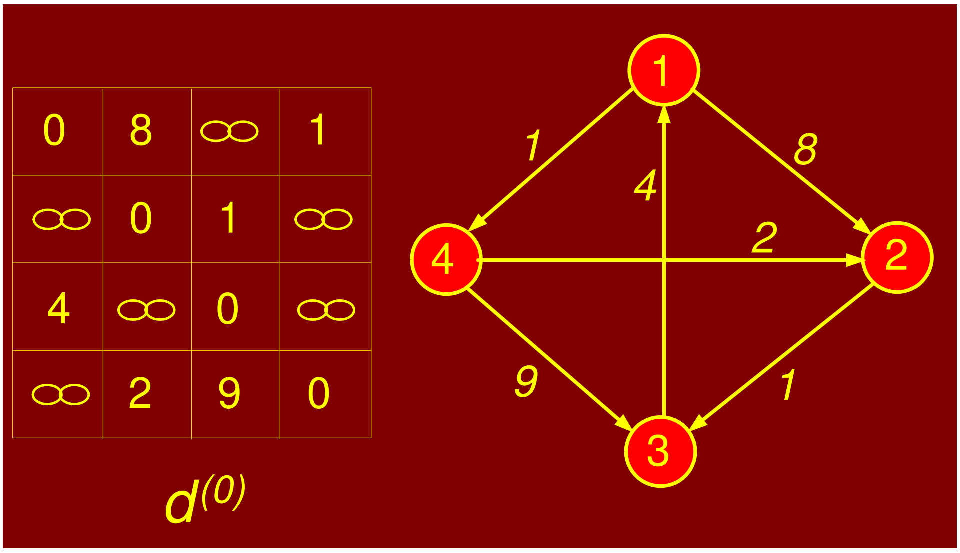
The final answer is $d^{(n)}_{ij}$ because this allows all possible vertices as intermediate vertices.

As is the case with DP algorithms, we will avoid recursive evaluation by generating a table for $d^{(k)}_{ij}$. The algorithm also includes mid-vertex pointers stored in $mid[i, j]$ for extracting the final path.

```
FLOYD-WARSHALL($n, w[1..n, 1..n]$)
for $(i = 1, n)$
do for $(j = 1, n)$
    do $d[i, j] \leftarrow w[i, j];\ mid[i, j] \leftarrow null$
for $(k = 1, n)$
do for $(i = 1, n)$
    do for $(j = 1, n)$
        do if $(d[i, k] + d[k, j] < d[i, j])$
            then $d[i, j] = d[i, k] + d[k, j]$
                $mid[i, j] = k$
```

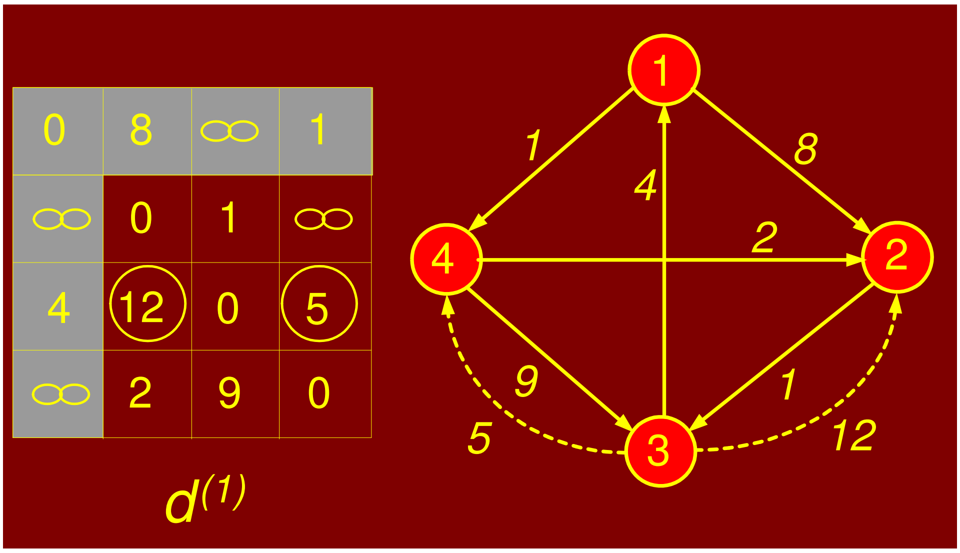
Clearly, the running time is $\Theta(n^3)$. The space used by the algorithm is $\Theta(n^2)$.

Figure 8.77 through 8.81 demonstrate the algorithm when applied to a graph. The matrix to left of the graph contains the matrix d entries. A circle around an entry $d_{i,k}$ indicates that it was updated in the current k iteration.



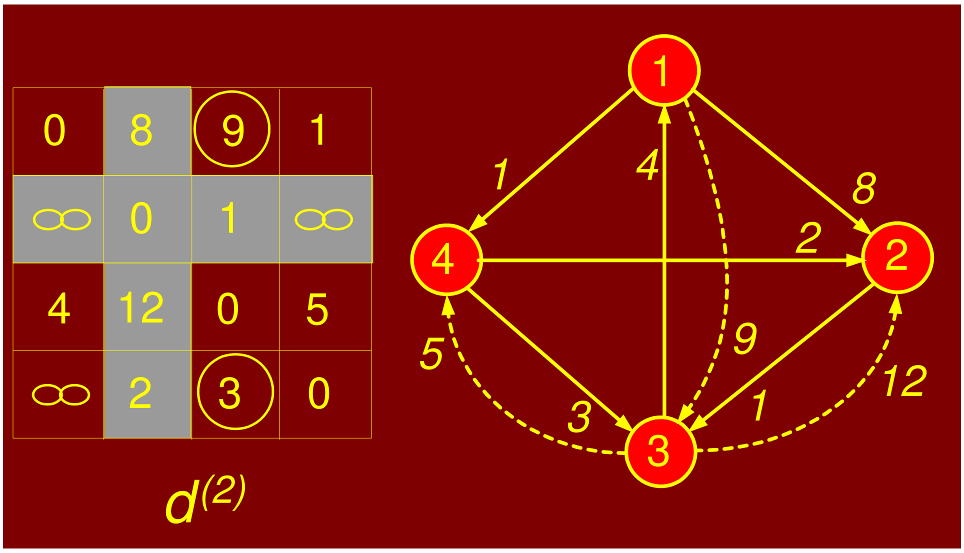
0	8	∞	1
∞	0	1	∞
4	∞	0	∞
∞	2	9	0

Figure 8.77: Floyd-Warshall Algorithm example: d$^($0)



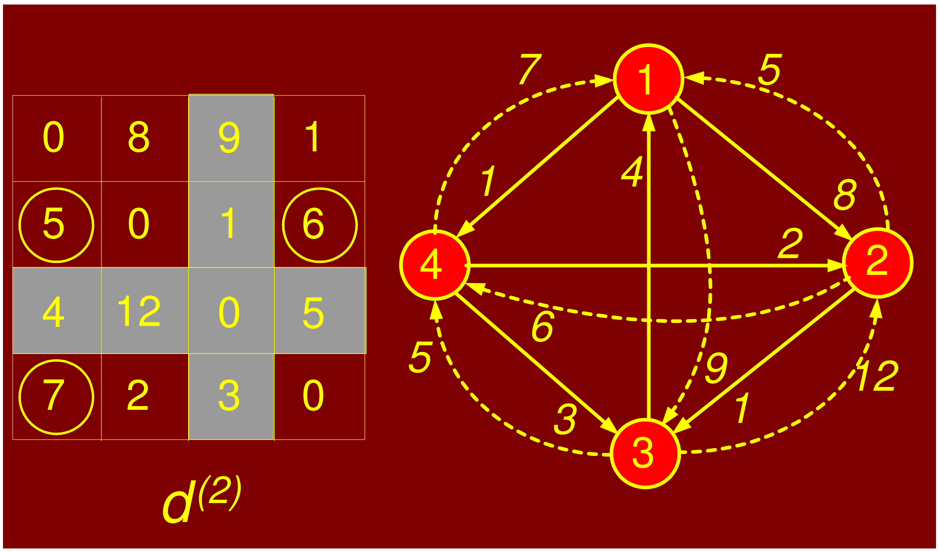
0	8	∞	1
∞	0	1	∞
4	12	0	5
∞	2	9	0

Figure 8.78: Floyd-Warshall Algorithm example: d$^($1)



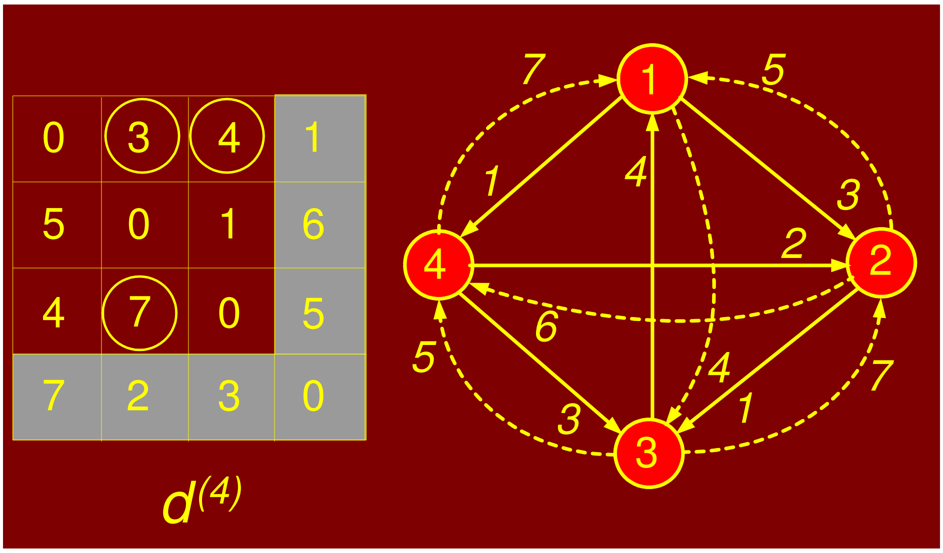
0	8	9	1
∞	0	1	∞
4	12	0	5
∞	2	3	0

Figure 8.79: Floyd-Warshall Algorithm example: d$^($2)



0	8	9	1
5	0	1	6
4	12	0	5
7	2	3	0

Figure 8.80: Floyd-Warshall Algorithm example: d$^($3)



0	3	4	1
5	0	1	6
4	7	0	5
7	2	3	0

Figure 8.81: Floyd-Warshall Algorithm example: $d^{(}4)$

Extracting Shortest Path:

The matrix d holds the final shortest distance between pairs of vertices. In order to compute the shortest path, the mid-vertex pointers $\mathrm{mid}[i,j]$ can be used to extract the final path. Whenever we discovered that the shortest path from i to j passed through vertex k, we set $\mathrm{mid}[i,j] = k$. If the shortest path did not pass through k then $\mathrm{mid}[i,j] = \mathrm{null}$.

To find the shortest path from i to j, we consult $\mathrm{mid}[i,j]$. If it is null, then the shortest path is just the edge (i,j). Otherwise we recursively compute the shortest path from i to $\mathrm{mid}[i,j]$ and the shortest path from $\mathrm{mid}[i,j]$ to j.

```
PATH(i, j)
if (mid[i, j] == null)
  then output(i, j)
  else PATH(i, mid[i, j])
       PATH(mid[i, j], j)
```

Chapter 9

Complexity Theory

So far in the course, we have been building up a “bag of tricks” for solving algorithmic problems. Hopefully you have a better idea of how to go about solving such problems. What sort of design paradigm should be used: divide-and-conquer, greedy, dynamic programming.

What sort of data structures might be relevant: trees, heaps, graphs. What is the running time of the algorithm. All of this is fine if it helps you discover an acceptably efficient algorithm to solve your problem.

The question that often arises in practice is that you have tried every trick in the book and nothing seems to work. Although your algorithm can solve small problems reasonably efficiently (e.g., $n \leq 20$), for the really large problems you want to solve, your algorithm never terminates. When you analyze its running time, you realize that it is running in exponential time, perhaps $n^{\sqrt{n}}$, or 2^n, or 2^{2^n}, or $n!$ or worse!.

By the end of 60’s, there was great success in finding efficient solutions to many combinatorial problems. But there was also a growing list of problems for which there seemed to be no known efficient algorithmic solutions.

People began to wonder whether there was some unknown paradigm that would lead to a solution to there problems. Or perhaps some proof that these problems are inherently hard to solve and no algorithmic solutions exist that run under exponential time.

Near the end of the 1960’s, a remarkable discovery was made. Many of these hard problems were interrelated in the sense that if you could solve any one of them in polynomial time, then you could solve all of them in polynomial time. this discovery gave rise to the notion of NP-completeness.

This area is a radical departure from what we have been doing because the emphasis will change. The goal is no longer to prove that a problem *can* be solved efficiently by presenting an algorithm for it. Instead we will be trying to show that a problem *cannot* be solved efficiently.

Up until now all algorithms we have seen had the property that their worst-case running time are bounded above by some *polynomial* in n. A *polynomial time algorithm* is any algorithm that runs in $O(n^k)$ time. A problem is solvable in polynomial time if there is a polynomial time algorithm for it.

Some functions that do not look like polynomials (such as $O(n \log n)$ are bounded above by polynomials (such as $O(n^2)$). Some functions that do look like polynomials are not. For example, suppose you have an algorithm that takes as input a graph of size n and an integer k and run in $O(n^k)$ time.

Is this a polynomial time algorithm? No, because k is an input to the problem so the user is allowed to choose $k = n$, implying that the running time would be $O(n^n)$. $O(n^n)$ is surely not a polynomial in n. The important aspect is that the exponent must be a constant independent of n.

9.1 Decision Problems

Most of the problems we have discussed involve optimization of one form of another. Find the shortest path, find the minimum cost spanning tree, maximize the knapsack value. For rather technical reasons, the NP-complete problems we will discuss will be phrased as *decision problems*.

A problem is called a *decision problem* if its output is a simple "yes" or "no" (or you may this of this as true/false, 0/1, accept/reject.) We will phrase may optimization problems as decision problems. For example, the MST decision problem would be: Given a weighted graph G and an integer k, does G have a spanning tree whose weight is at most k?

This may seem like a less interesting formulation of the problem. It does not ask for the weight of the minimum spanning tree, and it does not even ask for the edges of the spanning tree that achieves this weight. However, our job will be to show that certain problems cannot be solved efficiently. If we show that the simple decision problem cannot be solved e¡ciently, then the more general optimization problem certainly cannot be solved efficiently either.

9.2 Complexity Classes

Before giving all the technical definitions, let us say a bit about what the general classes look like at an intuitive level.

Class P: This is the set of all decision problems that can be *solved* in polynomial time. We will generally refer to these problems as being "easy" or "efficiently solvable".

Class NP: This is the set of all decision problems that can be *verified* in polynomial time. This class contains P as a subset. It also contains a number of problems that are believed to be very " hard" to solve.

Class NP: The term "NP" does not mean "not polynomial". Originally, the term meant " non-deterministic polynomial" but it is a bit more intuitive to explain the concept from the perspective of verification.

Class NP-hard: In spite of its name, to say that a problem is NP-hard does not mean that it is hard to solve. Rather, it means that if we could solve this problem in polynomial time, then we could solve *all* NP problems in polynomial time. Note that for a problem to NP-hard, it does not have to be in the class NP.

Class NP-complete: A problem is NP-complete if (1) it is in NP and (2) it is NP-hard.

The Figure 9.1 illustrates one way that the sets P, NP, NP-hard, and NP-complete (NPC) might look. We say might because we do not know whether all of these complexity classes are distinct or whether they are all solvable in polynomial time. The Graph Isomorphism, which asks whether two graphs are identical up to a renaming of their vertices. It is known that this problem is in NP, but it is not known to be in P. The other is QBF, which stands for Quantified Boolean Formulas. In this problem you are given a boolean formula and you want to know whether the formula is true or false.

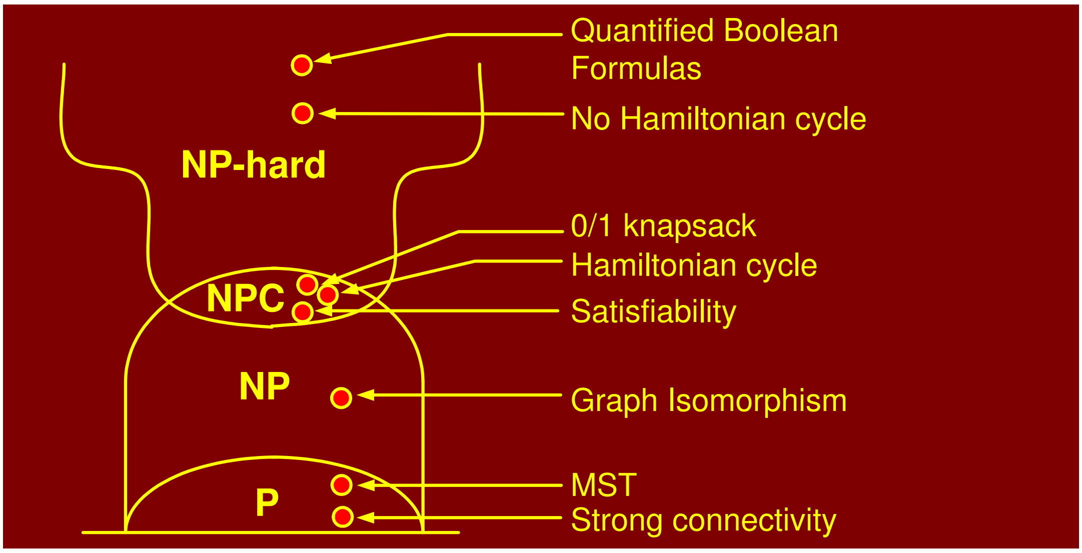


Figure 9.1: Complexity Classes

9.3 Polynomial Time Verification

Before talking about the class of NP-complete problems, it is important to introduce the notion of a *verification algorithm*. Many problems are hard to solve but they have the property that it easy to verify the solution if one is provided. Consider the Hamiltonian cycle problem.

Given an undirected graph G, does G have a cycle that visits every vertex exactly once? There is no known polynomial time algorithm for this problem.

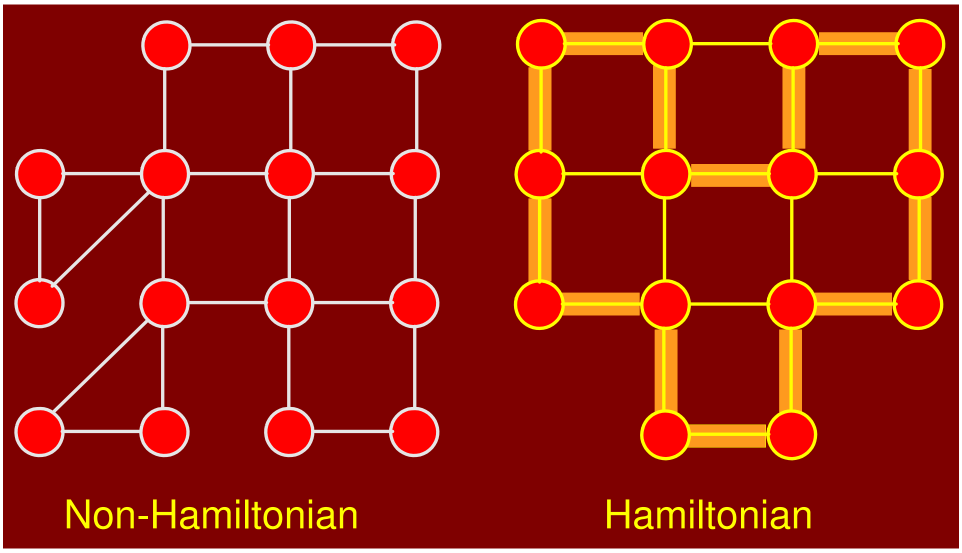


Figure 9.2: Hamiltonian Cycle

However, suppose that a graph did have a Hamiltonian cycle. It would be easy for someone to convince of this. They would simply say: "the cycle is $\langle v_3, v_7, v_1, \ldots, v_13\rangle$ We could then inspect the graph and check that this is indeed a legal cycle and that it visits all of the vertices of the graph exactly once. Thus, even though we know of no efficient way to *solve* the Hamiltonian cycle problem, there is a very efficient way to *verify* that a a given cycle is indeed a Hamiltonian cycle.

The piece of information that allows verification is called a *certificate*. Note that not all problems have the property that they are easy to verify. For example, consider the following two:

1. $UHC = \{(G)|G$ has a unique Hamiltonian cycle$\}$
2. $\overline{HC} = \{(G)|G$ has no Hamiltonian cycle$\}$

Suppose that a graph G is in UHC. What information would someone give us that would allow us to verify this? They could give us an example of the unique Hamiltonian cycle and we could verify that it is a Hamiltonian cycle. But what sort of certificate could they give us to convince us that this is the *only* one?

They could give another cycle that is *not* Hamiltonian. But this does not mean that there is not another cycle somewhere that is Hamiltonian. They could try to list every other cycle of length n, but this is not efficient at all since there are $n!$ possible cycles in general. Thus it is hard to imagine that someone could give us some information that would allow us to efficiently verify that the graph is in UHC.

9.4 The Class NP

The class NP is a set of all problems that can be verified by a polynomial time algorithm. Why is the set called "NP" and not "VP"? The original term NP stood for *non-deterministic polynomial time*. This referred to a program running on a non-deterministic computer that can make guesses. Such a computer could non-deterministically guess the value of the certificate. and then verify it in polynomial time. We have avoided introducing non-determinism here; it is covered in other courses such as automata or complexity theory.

Observe that $P \subseteq NP$. In other words, if we can solve a problem in polynomial time, we can certainly verify the solution in polynomial time. More formally, we do not need to see a certificate to solve the problem; we can solve it in polynomial time anyway.

However, it is not known whether $P = NP$. It seems unreasonable to think that this should be so. Being able to verify that you have a correct solution does not help you in finding the actual solution. The belief is that $P \neq NP$ but no one has a proof for this.

9.5 Reductions

The class NP-complete (NPC) problems consists of a set of decision problems (a subset of class NP) that no one knows how to solve efficiently. But if there were a polynomial solution for even a single NP-complete problem, then ever problem in NPC will be solvable in polynomial time. For this, we need the concept of *reductions*.

Consider the question: Suppose there are two problems, A and B. You know (or you strongly believe at least) that it is impossible to solve problem A in polynomial time. You want to prove that B cannot be solved in polynomial time. We want to show that

$$(A \notin P) \Rightarrow (B \notin P)$$

How would you do this? Consider an example to illustrate reduction: The following problem is well-known to be NPC:

3-color: Given a graph G, can each of its vertices be labelled with one of 3 different colors such that two adjacent vertices have the same label (color).

Coloring arises in various partitioning problems where there is a constraint that two objects cannot be assigned to the same set of partitions. The term “coloring” comes from the original application which was in map drawing. Two countries that share a common border should be colored with different colors.

It is well known that planar graphs can be colored (maps) with *four colors*. There exists a polynomial time algorithm for this. But determining whether this can be done with 3 colors is hard and there is no polynomial time algorithm for it. In Figure 9.3, the graph on the left can be colored with 3 colors while the graph on the right cannot be colored.

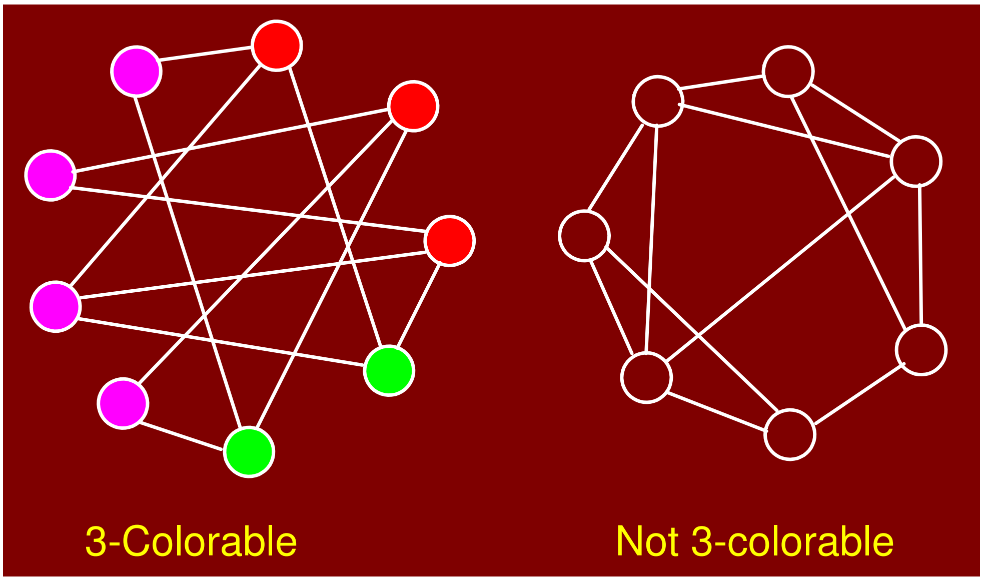


Figure 9.3: Examples of 3-colorable and non-3-colorable graphs

Example1: Fish tank problem

Consider the following problem than can be solved with the graph coloring approach. A tropical fish hobbyist has six different types of fish designated by A, B, C, D, E, and F, respectively. Because of predator-prey relationships, water conditions and size, some fish can be kept in the same tank. The following table shows which fish cannot be together:

Type	Cannot be with
A	B, C
B	A ,C, E
C	A, B, D, E
D	C, F
E	B, C, F
F	D, E

These constraints can be displayed as a graph where an edge between two vertices exists if the two species cannot be together. This is shown in Figure 9.4. For example, A cannot be with B and C; there is an edge between A and B and between A and C.

Given these constraints, What is the *smallest* number of tanks needed to keep all the fish? The answer can be found by coloring the vertices in the graph such that no two adjacent vertices have the same color. This particular graph is 3-colorable and therefore, 3 fish tanks are enough. This is depicted in Figure 9.5. The 3 fish tanks will hold fish as follows:

Tank 1	Tank 2	Tank 3
A, D	F, C	B, E

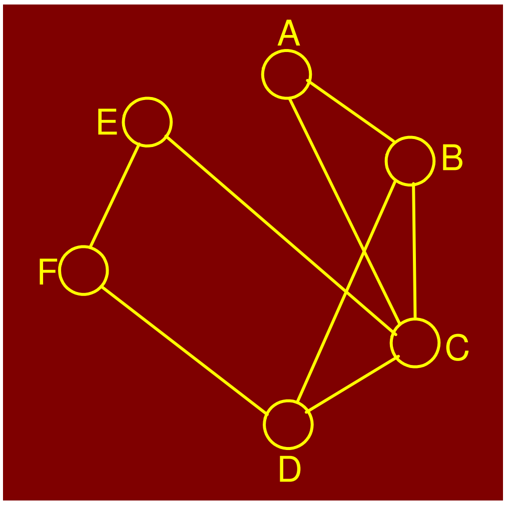


Figure 9.4: Graph representing constraints between fish species

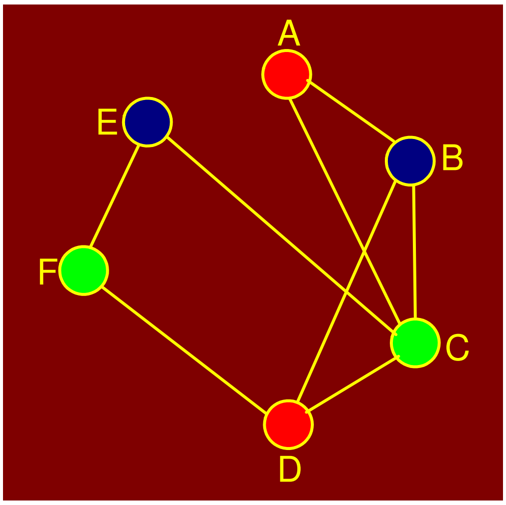


Figure 9.5: Fish tank graph colored with 3 colors

The 3-color (3Col) problem will the play the role of A, which we strongly suspect to not be solvable in polynomial time. For our problem B, consider the following problem: Given a graph $G = (V, E)$, we say that a subset of vertices $V' \subseteq V$ forms a *clique* if for every pair of vertices $u, v \in V'$, the edge $(u, v) \in V'$ That is, the subgraph induced by V' is a complete graph.

Clique Cover: Given a graph G and an integer k, can we find k subsets of vertices $V_1, V_2, \ldots, V_k$, such that $\bigcup_i V_i = V$, and that each V_i is a clique of G.

The following figure shows a graph that has a clique cover of size 3. There are three subgraphs that are complete.

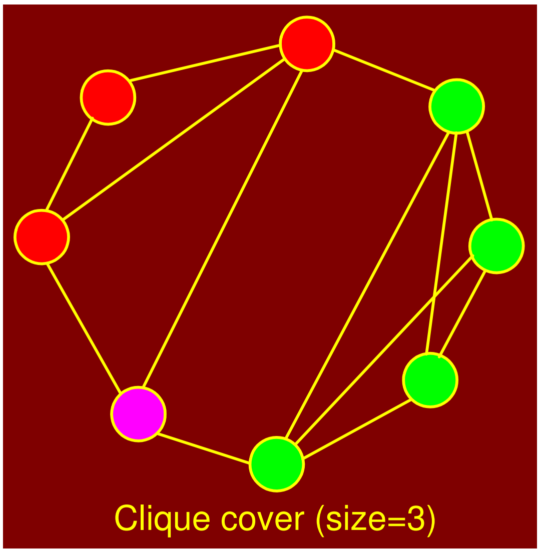


Figure 9.6: Graph with clique cover of size 3

The clique cover problem arises in applications of clustering. We put an edge between two nodes if they are similar enough to be clustered in the same group. We want to know whether it is possible to cluster all the vertices into k groups.

Suppose that you want to solve the CCov problem. But after a while of fruitless effort, you still cannot find a polynomial time algorithm for the CCov problem. How can you prove that CCov is likely to not have a polynomial time solution?

You know that 3Col is NP-complete and hence, experts believe that $3\text{Col} \notin P$. You feel that there is some connection between the CCov problem and the 3Col problem. Thus, you want to show that

$$(3\text{Col} \notin P) \Rightarrow (\text{CCov} \notin P)$$

Both problems involve partitioning the vertices into groups. In the clique cover problem, for two vertices to be in the same group, they must be adjacent to each other. In the 3-coloring problem, for two vertices to be in the same color group, they must not be adjacent. In some sense, the problems are almost the same but the adjacency requirements are exactly reversed.

We claim that we can reduce the 3-coloring problem into the clique cover problem as follows: Given a graph G for which we want to determine its 3-colorability, output the pair $(\overline{G}, 3)$ where $\overline{G}$ denotes the complement of G. Feed the pair $(\overline{G}, 3)$ into a routine for clique cover.

For example, the graph G in Figure 9.7 is 3-colorable and its complement $(\overline{G}, 3)$ is coverable by 3 cliques. The graph G in Figure 9.8 is not 3-colorable; it is also not coverable by cliques.

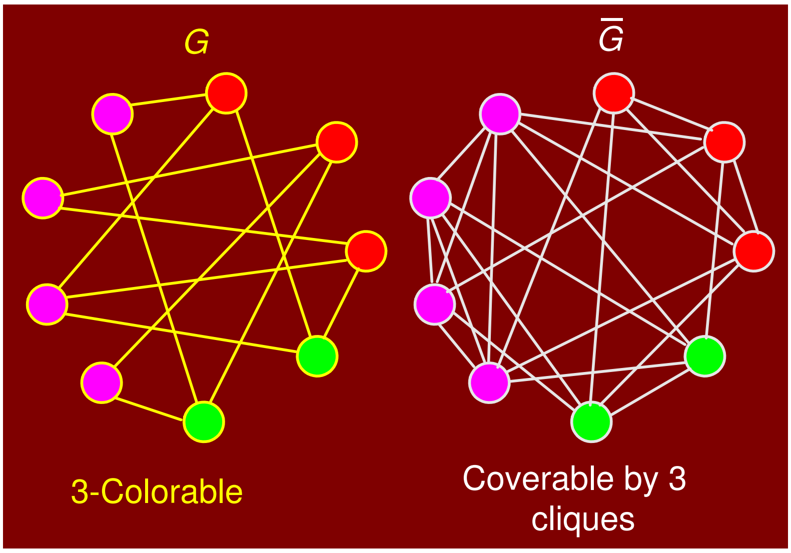


Figure 9.7: 3-colorable G and clique coverable $(\overline{G}, 3)$

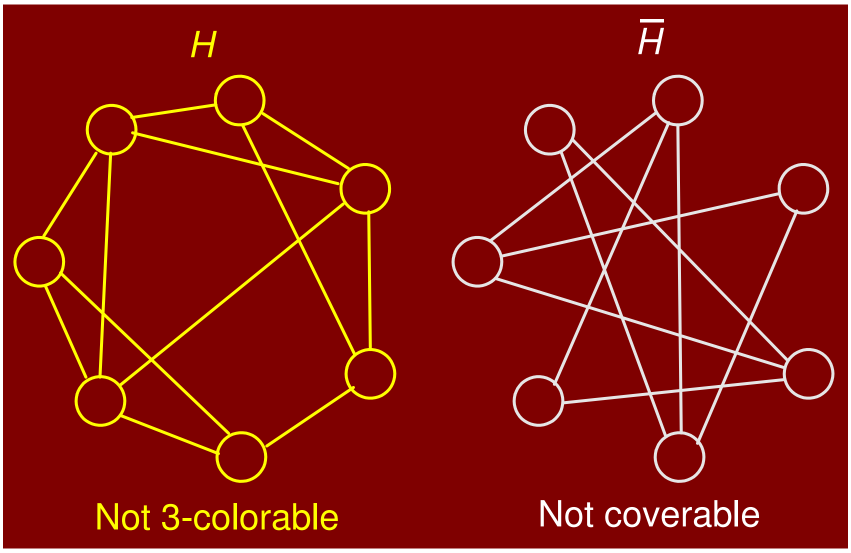


Figure 9.8: G is not 3-colorable and $(\overline{G}, 3)$ is not clique coverable

9.6 Polynomial Time Reduction

Definition: We say that a decision problem L_1 is polynomial-time reducible to decision problem L_2 (written $L_1 \leq_p L_2$) if there is polynomial time computable function f such that for all x, $x \in L_1$ if and only if $f(x) \in L_2$.

In the previous example we showed that

$$3\text{Col} \leq_P \text{CCov}$$

In particular, we have $f(G) = (\overline{G}, 3)$. It is easy to complement a graph in $O(n^2)$ (i.e., polynomial time). For example, flip the 0's and 1's in the adjacency matrix.

The way this is used in NP-completeness is that we have strong evidence that L_1 is not solvable in polynomial time. Hence, the reduction is effectively equivalent to saying that "since L_1 is not likely to be solvable in polynomial time, then L_2 is also not likely to be solvable in polynomial time.

9.7 NP-Completeness

The set of NP-complete problems is all problems in the complexity class NP for which it is known that if any one is solvable in polynomial time, then they all are. Conversely, if any one is not solvable in polynomial time, then none are.

Definition: A decision problem L is NP-Hard if

$$L' \leq_P L \quad \text{for all } L' \in \text{NP}.$$

Definition: L is NP-complete if

1. $L \in$ NP and
2. $L' \leq_P L$ for some known NP-complete problem L'.

Given this formal definition, the complexity classes are:

P: is the set of decision problems that are solvable in polynomial time.

NP: is the set of decision problems that can be verified in polynomial time.

NP-Hard: L is NP-hard if for all $L' \in$ NP, $L' \leq_P L$. Thus if we could solve L in polynomial time, we could solve all NP problems in polynomial time.

NP-Complete L is NP-complete if

1. $L \in$ NP and
2. L is NP-hard.

The importance of NP-complete problems should now be clear. If any NP-complete problem is solvable in polynomial time, then every NP-complete problem is also solvable in polynomial time. Conversely, if we can prove that any NP-complete problem cannot be solved in polynomial time, the every NP-complete problem cannot be solvable in polynomial time.

9.8 Boolean Satisfiability Problem: Cook's Theorem

We need to have at least one NP-complete problem to start the ball rolling. Stephen Cook showed that such a problem existed. He proved that the *boolean satisfiability problem* is NP-complete. A boolean formula is a logical formulation which consists of variables x_i. These variables appear in a logical expression using logical operations

1. negation of x: $\overline{x}$
2. boolean or: $(x \vee y)$
3. boolean and: $(x \wedge y)$

For a problem to be in NP, it must have an efficient verification procedure. Thus virtually all NP problems can be stated in the form, "does there exists X such that $P(X)$", where X is some structure (e.g. a set, a path, a partition, an assignment, etc.) and $P(X)$ is some property that X must satisfy (e.g. the set of objects must fill the knapsack, or the path must visit every vertex, or you may use at most k colors and no two adjacent vertices can have the same color). In showing that such a problem is in NP, the certificate consists of giving X, and the verification involves testing that $P(X)$ holds.

In general, any set X can be described by choosing a set of objects, which in turn can be described as choosing the values of some boolean variables. Similarly, the property $P(X)$ that you need to satisfy, can be described as a boolean formula. Stephen Cook was looking for the most general possible property he could, since this should represent the hardest problem in NP to solve. He reasoned that computers (which represent the most general type of computational devices known) could be described entirely in terms of boolean circuits, and hence in terms of boolean formulas. If any problem were hard to solve, it would be one in which X is an assignment of boolean values (true/false, 0/1) and $P(X)$ could be any boolean formula. This suggests the following problem, called the *boolean satisfiability problem*.

SAT: Given a boolean formula, is there some way to assign truth values (0/1, true/false) to the variables of the formula, so that the formula evaluates to true?

A boolean formula is a logical formula which consists of variables x_i , and the logical operations $\overline{x}$ meaning the *negation* of x, *boolean-or* $(x \vee y)$ and *boolean-and* $(x \wedge y)$. Given a boolean formula, we say that it is satisfiable if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1. (As opposed to the case where no matter how you assign truth values the result is always 0.) For example

$$(x_1 \wedge (x_2 \vee \overline{x_3})) \wedge ((\overline{x_2} \wedge \overline{x_3}) \vee \overline{x_1})$$

is satisfiable, by the assignment $x_1 = 1$, $x_2 = 0$ and $x_3 = 0$. On the other hand,

$$(\overline{x_1} \vee (x_2 \wedge x_3)) \wedge (x_1 \vee (\overline{x_2} \wedge \overline{x_3})) \wedge (x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$$

is not satisfiable. Such a boolean formula can be represented by a logical circuit made up of OR, AND and NOT gates. For example, Figure 9.9 shows the circuit for the boolean formula

$$((x_1 \wedge x_4) \vee x_2) \wedge ((x_3 \wedge \overline{x_4}) \vee \overline{x_2}) \wedge \overline{x_5}$$

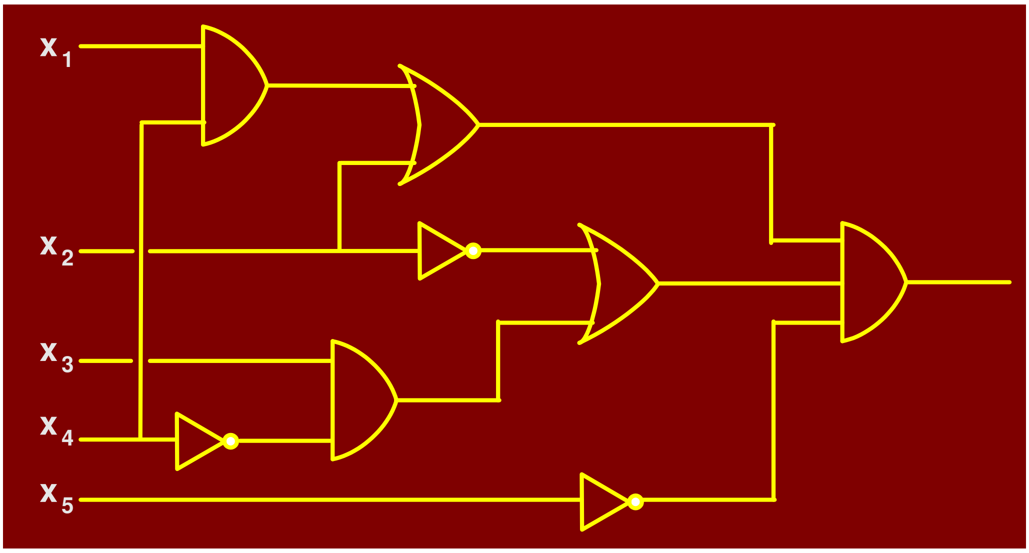


Figure 9.9: Logical circuit for a boolean formula

Cook's Theorem: SAT is NP-complete

We will not prove the theorem; it is quite complicated. In fact, it turns out that a even more restricted version of the satisfiability problem is NP-complete.

A *literal* is a variable x or its negation $\overline{x}$. A boolean formula is in *3-Conjunctive Normal Form* (3-CNF) if it is the boolean-and of clauses where each clause is the boolean-or of exactly three literals. For example,

$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_3 \vee x_4) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_4})$$

is in 3-CNF form. 3SAT is the problem of determining whether a formula is 3-CNF is satisfiable. 3SAT is NP-complete. We can use this fact to prove that other problems are NP-complete. We will do this with the *independent set problem*.

Independent Set Problem: Given an undirected graph $G = (V, E)$ and an integer k, does G contain a subset V' of k vertices such that no two vertices in V' are adjacent to each other.

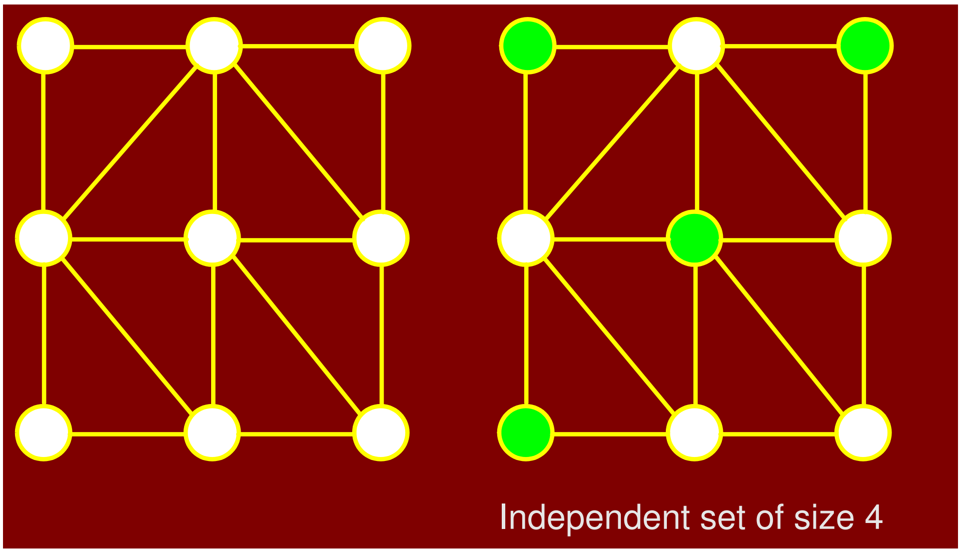


Figure 9.10:

The independent set problem arises when there is some sort of selection problem where there are mutual restrictions pairs that cannot both be selected. For example, a company dinner where an employee and his or her immediate supervisor cannot both be invited.

Claim: IS is NP-complete

The proof involves two parts. First, we need to show that IS $\in$ NP. The certificate consists of k vertices of V'. We simply verify that for each pair of vertices $u, v \in V'$, there is no edge between them. Clearly, this can be done in polynomial time, by an inspection of the adjacency matrix.

Second, we need to establish that IS is NP-hard This can be done by showing that some known NP-compete (3SAT) is polynomial-time reducible to IS. That is, 3SAT $\leq_P$ IS.

An important aspect to reductions is that we do not attempt to solve the satisfiability problem. Remember: It is NP-complete, and there is not likely to be any polynomial time solution. The idea is to translate the similar elements of the satisfiable problem to corresponding elements of the independent set problem.

What is to be selected?

3SAT: Which variables are to be assigned the value true, or equivalently, which literals will be true and which will be false.

IS: Which vertices will be placed in V'.

Requirements:

3SAT: Each clause must contain at least one true valued literal.

IS: V' must contain at least k vertices.

Restrictions:

3SAT: If x_i is assigned true, then $\overline{x_i}$ must be false and vice versa.

IS: If u is selected to be in V' and v is a neighbor of u then v cannot be in V'.

We want a function which given any 3-CNF boolean formula F , converts it into a pair (G, k) such that the above elements are translated properly. Our strategy will be to turn each literal into a vertex. The vertices will be in clause clusters of three, one for each clause. Selecting a true literal from some clause will correspond to selecting a vertex to add to V'. We will set k equal to the number of clauses, to force the independent set subroutine to select one true literal from each clause. To keep the IS subroutine from selecting two literals from one clause and none from some other, we will connect all the vertices in each clause cluster with edges. To keep the IS subroutine from selecting a literal and its complement to be true, we will put an edge between each literal and its complement.

A formal description of the reduction is given below. The input is a boolean formula F in 3-CNF, and the output is a graph G and integer k.

```
3SAT-TO-IS($F$)
$k \leftarrow$ number of clauses in $F$
for ( each clause $C$ in $F$ )
do create a clause cluster of
      3 vertices from literals of $C$
for ( each clause cluster $(x_1, x_2, x_3)$ )
do create an edge $(x_i, x_j)$ between
      all pairs of vertices in the cluster
for ( each vertex $x_i$ )
do create an edge between $x_i$ and
      all its complement vertices $\overline{x_i}$
```

```
return $(G, k)$   // output is graph $G$ and integer $k$
```

If F has k clauses, then G has exactly $3k$ vertices. Given any reasonable encoding of F , it is an easy programming exercise to create G (say as an adjacency matrix) in polynomial time. We claim that F is satisfiable if and only if G has an independent set of size k.

Example: Suppose that we are given the 3-CNF formula:

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3)$$

The following series of figures show the reduction which produces the graph and sets k = 4. First, each of the four literals is converted into a three-vertices graph. This is shown in Figure 9.11

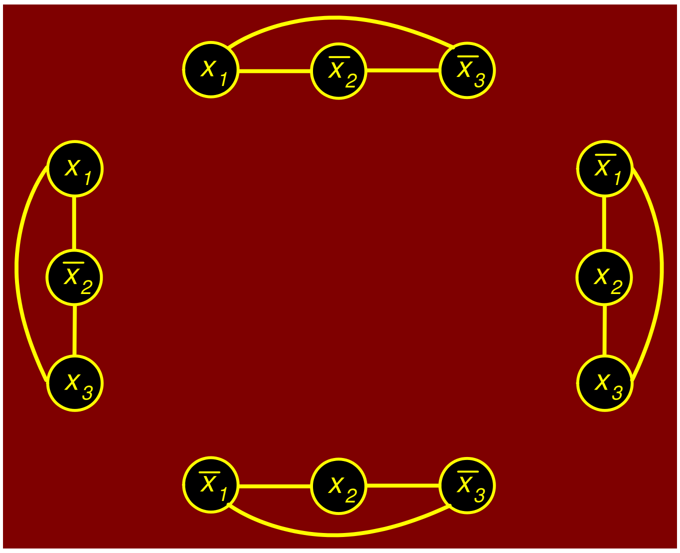


Figure 9.11: Four graphs, one for each of the 3-terms literal

Next, each term is connected to its complement. This is shown in Figure 9.12.

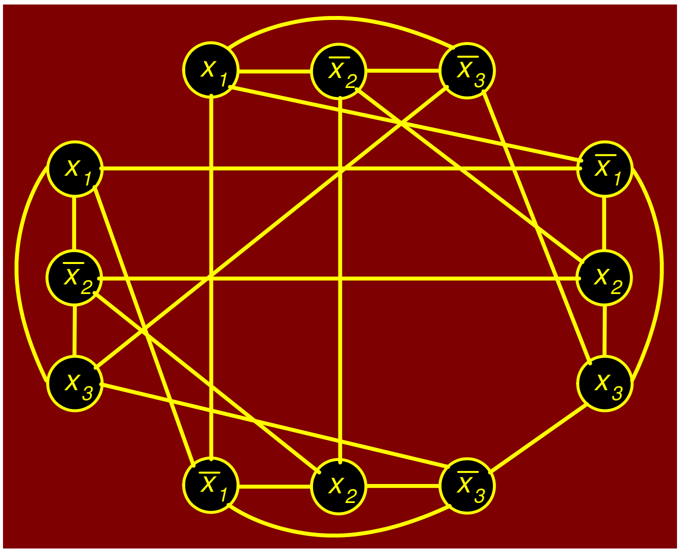


Figure 9.12: Each term is connected to its complement

The boolean formula is satisfied by the assignment $x_1 = 1,\ x_2 = 1\ x_3 = 0$. This implies that the first literal of the first and last clauses are 1, the second literal of the second clause is 1, and the third literal of the third clause is 1. By selecting vertices corresponding to true literals in each clause, we get an independent set of size $k = 4$. This is shown in Figure 9.13.

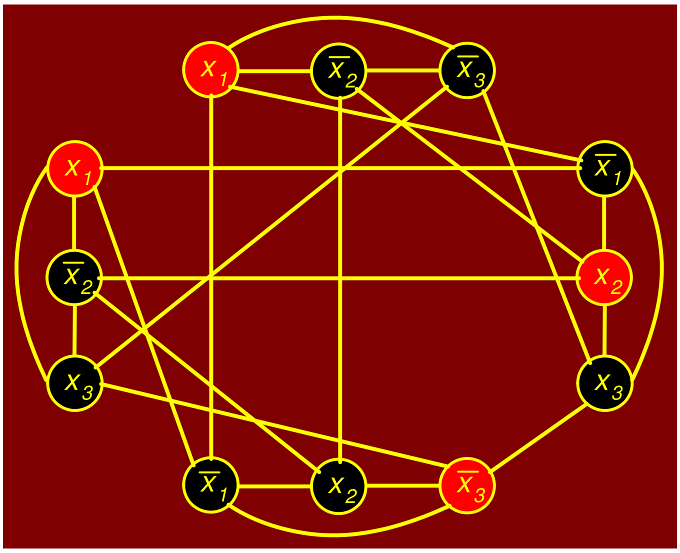


Figure 9.13: Independent set corresponding to $x_1 = 1,\ x_2 = 1\ x_3 = 0$

Correctness Proof:

We claim that F is satisfiable if and only if G has an independent set of size k. If F is satisfiable, then each of the k clauses of F must have at least one true literal. Let V' denote the corresponding vertices from each of the clause clusters (one from each cluster). Because we take vertices from each cluster, there are no inter-cluster edges between them, and because we cannot set a variable and its complement to both be true, there can be no edge of the form $(x_i, \overline{x_i})$ between the vertices of V' . Thus, V' is an independent set of size k.

Conversely, G has an independent set V' of size k. First observe that we must select a vertex from each clause cluster, because there are k clusters, and we cannot take two vertices from the same cluster (because they are all interconnected). Consider the assignment in which we set all of these literals to 1. This assignment is logically consistent, because we cannot have two vertices labelled x_i and $\overline{x_i}$ in the same cluster.

Finally the transformation clearly runs in polynomial time. This completes the NP-completeness proof.

Also observe that the the reduction had no knowledge of the solution to either problem. Computing the solution to either will require exponential time. Instead, the reduction simple translated the input from one problem into an equivalent input to the other problem.

9.9 Coping with NP-Completeness

With NP-completeness we have seen that there are many important optimization problems that are likely to be quite hard to solve exactly. Since these are important problems, we cannot simply give up at this point, since people do need solutions to these problems. Here are some strategies that are used to cope with NP-completeness:

Use brute-force search: Even on the fastest parallel computers this approach is viable only for the smallest instance of these problems.

Heuristics: A heuristic is a strategy for producing a valid solution but there are no guarantees how close it to optimal. This is worthwhile if all else fails.

General search methods: Powerful techniques for solving general combinatorial optimization problems. Branch-and-bound, A*-search, simulated annealing, and genetic algorithms

Approximation algorithm: This is an algorithm that runs in polynomial time (ideally) and produces a solution that is within a guaranteed factor of the optimal solution.