

TRƯỜNG ĐH BÁCH KHOA TP. HCM
KHOA CÔNG NGHỆ THÔNG TIN



PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT
ALGORITHMS ANALYSIS AND DESIGN

<http://www.dit.hcmut.edu.vn/~nldkhoa/pttkgt/slides/>

TABLE OF CONTENTS

Chapter 1. FUNDAMENTALS.....	1
1.1. ABSTRACT DATA TYPE	1
1.2. RECURSION.....	2
1.2.1. Recurrence Relations.....	2
1.2.2. Divide and Conquer	3
1.2.3. Removing Recursion.....	4
1.2.4. Recursive Traversal.....	5
1.3. ANALYSIS OF ALGORITHMS	8
1.3.1. Framework.....	8
1.3.2. Classification of Algorithms.....	9
1.3.3. Computational Complexity.....	10
1.3.4. Average-Case-Analysis.....	10
1.3.5. Approximate and Asymptotic Results.	10
1.3.6. Basic Recurrences	11
Chapter 2. ALGORITHM CORRECTNESS	14
2.1. PROBLEMS AND SPECIFICATIONS	14
2.1.1. Problems.....	14
2.1.2. Specification of a Problem	14
2.2. PROVING RECURSIVE ALGORITHMS.....	15
2.3. PROVING ITERATIVE ALGORITHMS	16
Chapter 3. ANALYSIS OF SOME SORTING AND SEARCHING ALGORITHMS.....	20
3.1. ANALYSIS OF ELEMENTARY SORTING METHODS	20
3.1.1. Rules of the Game.....	20
3.1.2. Selection Sort.....	20
3.1.3. Insertion Sort.....	21
3.1.4. Bubble sort.....	22
3.2. QUICKSORT	23
3.2.1. The Basic Algorithm.....	23
3.2.2. Performance Characteristics of Quicksort.....	25
3.2.3. Removing Recursion.....	27
3.3. RADIX SORTING	27
3.3.1. Bits.....	27
3.3.2. Radix Exchange Sort	28
3.3.3. Performance Characteristics of Radix Sorts.....	29
3.4. MERGESORT	29
3.4.1. Merging	30
3.4.2. Mergesort.....	30
3.5. EXTERNAL SORTING.....	31
3.5.1. Block and Block Access	31
3.5.2. External Sort-merge	32
3.6. ANALYSIS OF ELEMENTARY SEARCH METHODS.....	34
3.6.1. Linear Search	34

3.6.2. Binary Search	35
Chapter 4. ANALYSIS OF SOME ALGORITHMS ON DATA STRUCTURES	36
4.1. SEQUENTIAL SEARCHING ON A LINKED LIST	36
4.2. BINARY SEARCH TREE	37
4.3. PRIORITIY QUEUES AND HEAPSORT	41
4.3.1. Heap Data Structure.....	42
4.3.2. Algorithms on Heaps	43
4.3.3. Heapsort	45
4.4. HASHING	48
4.4.1. Hash Functions.....	48
4.4.2. Separate Chaining.....	49
4.4.3. Linear Probing	50
4.5. STRING MATCHING AGORITHMS	52
4.5.1. The Naive String Matching Algorithm	52
4.5.2. The Rabin-Karp algorithm	53
Chapter 5. ANALYSIS OF GRAPH ALGORITHMS.....	56
5.1. ELEMENTARY GRAPH ALGORITHMS	56
5.1.1. Glossary.....	56
5.1.2. Representation.....	57
5.1.3. Depth-First Search	59
5.1.4. Breadth-first Search	64
5.2. WEIGHTED GRAPHS	65
5.2.1. Minimum Spanning Tree	65
5.2.2. Prim's Algorithm.....	67
5.3. DIRECTED GRAPHS.....	71
5.3.1. Transitive Closure	71
5.3.2. All Shortest Paths	73
5.3.3. Topological Sorting.....	74
Chapter 6. ALGORITHM DESIGN TECHNIQUES	78
6.1. DYNAMIC PROGRAMMING.....	78
6.1.1. Matrix-Chain Multiplication	78
6.1.2. Elements of Dynamic Programming	82
6.1.3. Longest Common Subsequence	83
6.1.4 The Knapsack Problem.....	86
6.1.4 The Knapsack Problem.....	87
6.2. GREEDY ALGORITHMS.....	88
6.2.1. An Activity-Selection Problem.....	89
6.2.2. Huffman Codes	93
6.3. BACKTRACKING ALGORITHMS	97
6.3.1. The Knight's Tour Problem.....	97
6.3.2. The Eight Queen's Problem	101
Chapter 7. NP-COMPLETE PROBLEMS	106
7.1. NP-COMPLETE PROBLEMS	106
7.2. NP-COMPLETENESS.....	108

7.3. COOK'S THEOREM.....	110
7.4. Some NP-Complete Problems.....	110
EXERCISES	112
REFERENCES.....	120

Chapter 1. FUNDAMENTALS

1.1. ABSTRACT DATA TYPE

It's convenient to describe a data structure in terms of the *operations* performed, rather than in terms of implementation details.

That means we should separate the concepts from particular implementations.

When a data structure is defined that way, it's called an *abstract data type* (ADT).

An *abstract data type* is a mathematical model, together with various operations defined on the model.

Some examples:

A *set* is a collection of zero or more entries. An entry may not appear more than once. A set of n entries may be denoted $\{a_1, a_2, \dots, a_n\}$, but the position of an entry has no significance.

A *multiset* is a set in which repeated elements are allowed. For example, $\{5, 7, 5, 2\}$ is a multiset.

initialize
insert,
is_empty,
delete
findmin

A *sequence* is an ordered collection of zero or more entries, denoted $\langle a_1, a_2, \dots, a_n \rangle$. The position of an entry in a sequence is significant.

initialize
length,
head,
tail,
concatenate,...

To see the importance of abstract data types, let consider the following problem.

Given an array of n numbers, $A[1..n]$, consider the problem of determining the k largest elements, where $k \leq n$. For example, if A contains $\{5, 3, 1, 9, 6\}$, and $k = 3$, then the result is $\{5, 9, 6\}$.

It's not easy to develop an algorithm to solve the above problem.

ADT: multiset

Operations:

Initialize,
Insert(x, M),
DeleteMin(M),
FindMin(M)

The Algorithm:

Initialize(M);
for $i := 1$ to k do

```

    Insert(A[i], M);
for i:= k + 1 to n do
    if A[i] > KeyOf(FindMin(M)) then
        begin
            DeleteMin(M); Insert(A[i],M)
        end;

```

In the above example, abstract data type simplifies the program by hiding details of their implementation.

ADT Implementation.

The process of using a concrete data structure to implement an ADT is called *ADT implementation*.

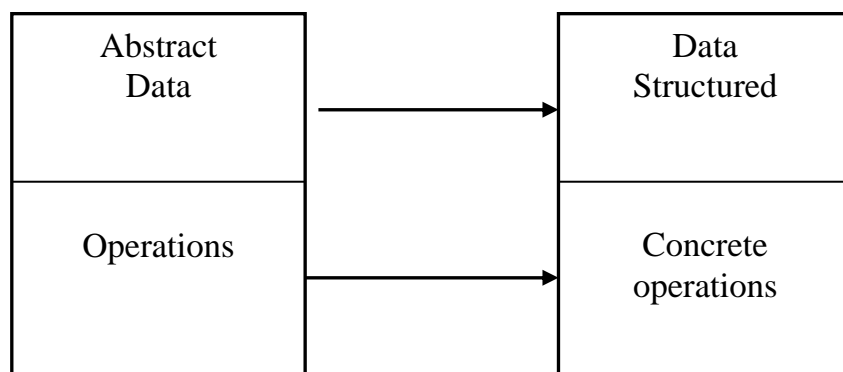


Figure 1.1: ADT Implementation

We can use arrays or linked list to implement sets.

We can use arrays or linked list to implement sequences.

As for the *multiset* ADT in the previous example, we can use *priority queue* data structure to implement it. And then we can use *heap* data structure to implement *priority queue*.

1.2. RECURSION

1.2.1. Recurrence Relations

Example 1: Factorial function

$$N! = N.(N-1)! \quad \text{for } N \geq 1$$

$$0! = 1$$

Recursive definition of function that involves integer arguments are called *recurrence relations*.

```

function factorial (N: integer): integer;
begin
    if N = 0
    then factorial: = 1
    else factorial: = N*factorial (N-1);
end;

```

Example 2: Fibonacci numbers

Recurrence relation:

$$F_N = F_{N-1} + F_{N-2} \quad \text{for } N \geq 2$$
$$F_0 = F_1 = 1$$

1, 1, 2, 3, 5, 8, 13, 21, ...

```
function fibonacci (N: integer): integer;  
begin  
  if N <= 1  
  then fibonacci: = 1  
  else fibonacci: = fibonacci(N-1) + fibonacci(N-2);  
end;
```

We can use an array to store previous results during computing fibonacci function.

```
procedure fibonacci;  
const max = 25  
var i: integer;  
F: array [0..max] of integer;  
begin  
  F[0]: = 1; F[1]: = 1;  
  for i: = 2 to max do  
    F[i]: = F[i-1] + F[i-2]  
end;
```

1.2.2. Divide and Conquer

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely-related subproblems.

These algorithms follow a *divide-and-conquer* approach: they *break* the problem into several subproblems, *solve* the subproblems and then *combine* these solutions to create a solution to the original problem.

This paradigm consists of 3 steps at each level of the recursion:

divide
conquer
combine

Example: Consider the task of drawing the markings for each inch in a ruler: there is a mark at the $\frac{1}{2}$ inch point, slightly shorter marks at $\frac{1}{4}$ inch intervals, still shorter marks at $\frac{1}{8}$ inch intervals etc.,...

Assume that we have a procedure *mark*(*x*, *h*) to make a mark *h* units at position *x*.

The “divide and conquer” recursive program is as follows:

```
procedure rule(l, r, h: integer);  
/* l: left position of the ruler; r: right position of the ruler */  
var m: integer;  
begin
```

```

if h > 0 then
  begin
    m: = (1 + r) div 2;
    mark(m, h);
    rule(l, m, h-1);
    rule(m, r, h-1)
  end;
end;

```

1.2.3. Removing Recursion

The question: how to translate a recursive program into non-recursive program.

The general method:

Give a recursive program P, each time there is a recursive call to P. The current values of parameters and local variables are *pushed* into the *stacks* for further processing.

Each time there is a recursive return to P, the values of parameters and local variables for the current execution of P are *restored* from the stacks.

The handling of the *return address* is done as follows:

Suppose the procedure P contains a recursive call in step K. The *return address* K+1 will be saved in a stack and will be used to return to the current level of execution of procedure P.

```

procedure Hanoi(n, beg, aux, end);
begin
  if n = 1 then
    writeln(beg, end)
  else
    begin
      hanoi(n-1, beg, end, aux) ;
      writeln(beg, end);
      hanoi(n-1, aux, beg, end);
    end;
end;

```

Non-recursive version:

```

procedure Hanoi(n, beg, aux, end: integer);
/* Stacks STN, STBEG, STAUX, STEND, and STADD correspond, respectively, to
variables N, BEG, AUX, END and ADD */
label 1, 3, 5;
var t: integer;
begin
  top: = 0;          /* preparation for stacks */
1: if n = 1 then
  begin writeln(beg, end); goto 5 end;

```



```

top: = top + 1; /* first recursive call to Hanoi */
STN[top]: = n; STBEG[top]: = beg;
STAUX [top]: = aux;
STEND [top]: = end;
STADD [top]: = 3; /* saving return address */
n: = n-1; t:= aux; aux: = end; end: = t;
goto 1;

3: writeln(beg, end);
top: = top + 1; /* second recursive call to hanoi */
STN[top]: = n; STBEG[top]: = beg;
STAUX[top]: = aux;
STEND[top]: = end;
STADD[top]: = 5; /* saving return address */
n: = n-1; t:= beg; beg: = aux; aux: = t;
goto 1;

5: /* translation of return point */
if top <> 0 then
  begin
    n: = STN[top]; beg: = STBEG [top];
    aux: = STAUX [top];
    end: = STEND [top]; add: = STADD [top];
    top: = top - 1;
    goto add
  end;
end;

```

1.2.4. Recursive Traversal

The simplest way to traverse the nodes of a tree is with recursive implementation.

Inorder traversal:

```

procedure traverse(t: link);
begin
  if t <> z then
    begin
      traverse(t↑.l);
      visit(t);
      traverse(t↑.r)
    end;
end;

```

Now, we study the question how to remove the recursion from the pre-order traversal program to get a non-recursive program.

```

procedure traverse (t: link)
begin

```

```

if t <> z then
  begin
    visit(t);
    traverse(t↑.l); traverse(t↑.r)
  end;
end;

```

First, the 2nd recursive call can be easily removed because there is no code following it.

The second recursive call can be transformed by a *goto* statement as follows:

```

procedure traverse (t: link);
label 0,1;
begin
0: if t = z then goto 1;
   visit(t);
   traverse(t↑. l);
   t := t↑.r; goto 0;
1: end;

```

This technique is called *tail-recursion* removal.

Removing the other recursive call requires more work.

Applying the general method, we can remove the second recursive call from our program:

```

procedure traverse(t: link);
label 0, 1, 2, 3;
begin
0: if t = z then goto 1;
   visit(t);
   push(t); t := t↑.l; goto 0;
3: t := t↑.r; goto 0;
1: if stack_empty then goto 2;
   t := pop; goto 3;
2: end;

```

Note: There is only one return address, 3, which is fixed, so we don't put it on the stack.

We can remove some *goto* statements by using a *while* loop.

```

procedure traverse(t: link);
label 0,2;
begin
0: while t <> z do
  begin
    visit(t);
    push(t↑.r); t := t↑.l;

```

```

    end;
    if stack_empty then goto 2;
    t: = pop; goto 0;
2: end;

```

Again, we can make the program gotoless by using a *repeat* loop.

```

procedure traverse(t: link);
begin
    push(t);
    repeat
        t: = pop;
        while t <> z do
            begin
                visit(t);
                push(t↑.r); t: = t↑.l;
            end;
        until stack_empty;
    end;

```

The *loop-within-a-loop* can be simplified as follows:

```

procedure traverse(t: link);
begin
    push(t);
    repeat
        t: = pop;
        if t <> z then
            begin
                visit(t);
                push(t↑.r); push(t↑.l);
            end;
        until stack_empty;
    end;

```

To avoid putting null subtrees on the stack, we can change the above program to:

```

procedure traverse(t: link);
begin
    push(t);
    repeat
        t: = pop;
        visit(t);
        if t↑.r <> z then push(t↑.r);
        if t↑.l <> z then push(t↑.l);
    until stack_empty;
end;

```

Exercise:

Translate the recursive procedure *Hanoi* to non-recursive version by using tail-recursion removal and then applying the general method of recursion removal.

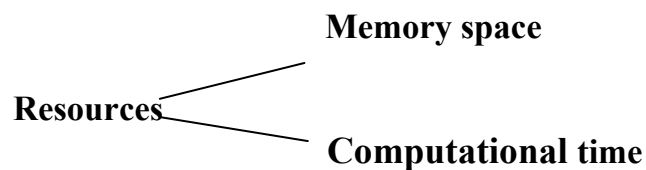
1.3. ANALYSIS OF ALGORITHMS

For most problems, there are many different algorithms available.

How to select the best algorithms?

How to compare algorithms?

Analyzing an algorithm: predicting the resources this algorithm requires.



Running time is the most important resource.

The running time of an algorithm \approx a function of the input size.

We are interested in

The **average case**, the amount of running time an algorithm would take with a “typical input data”.

The **worst case**, the amount of time an algorithm would take on the worst possible input configuration.

1.3.1. Framework

◆ The first step in the analysis of an algorithm is to characterize the input data and decide what type of analysis is appropriate.

Normally, we focus on:

Trying to prove that the running time is always less than some “**upper bound**”, or

Trying to derive the average running time for a “**random**” input.

◆ The second step in the analysis is to identify **abstract operations** on which the algorithm is based.

Example: **Comparisons** in sorting algorithm

The number of abstract operations depends on a few quantities.

◆ Third, we do the mathematical analysis to find **average** and **worst-case** values for each of the fundamental quantities.

It's not difficult to find an upper-bound on the running time of a program.
But the average-case analysis requires a sophisticated mathematical analysis.

In principle, the algorithm can be analyzed to a precise level of details. But in practice, we just do *estimating* in order to suppress details.

In short, we look for *rough estimates* for the running time of an algorithms (for purpose of classification).

1.3.2. Classification of Algorithms

Most algorithms have a primary parameter N , the number of data items to be processed. This parameter affects the running time most significantly.

Example:

The size of the array to be sorted/**searched**
The number of nodes in a graph.

The algorithms may have running time proportional to

1

If the operation is executed once or at most a few times.
 \Rightarrow The running time is constant.

$\lg N$ (logarithmic) $\log_2 N \equiv \lg N$
The algorithm grows slightly slower as N grows.

N (linear)

$N \lg N$

N^2 (quadratic) double – nested loop

N^3 (cubic) triple-nested loop

2^N few algorithms with exponential running time (combinatorics)

Some other algorithms may have running time
 $N^{3/2}$, \sqrt{N} , $\lg 2^N$

1.3.3. Computational Complexity

We focus on **worst-case** analysis. That is studying the worst-case performance, ignoring constant factors to determine the **functional dependence** of the running-time on the numbers of inputs.

Example: The running time of **mergesort** is proportional to $N \lg N$.

The concept of “**proportional to**”

The mathematical tool for making this notion precise is called the **O-notation**.

Notation: A function $g(N)$ is said to be $O(f(N))$ if there exists constant c_0 and N_0 such that $g(N)$ is less than $c_0 f(N)$ for all $N > N_0$.

The **O-notation** is a useful way to state upper-bounds on running time, which are independent of both inputs and implementation details.

We try to find both “**upper bound**” and “**lower bound**” on the worst-case running time.

But **lower-bound** is difficult to determine.

1.3.4. Average-Case-Analysis

We have to characterize the inputs of the algorithm
calculate the average number of times each instruction is executed.
calculate the average running time of the whole algorithm.

But it's difficult to
to determine the amount of time required by each instruction.
to characterize accurately the inputs encountered in practice.

1.3.5. Approximate and Asymptotic Results.

The results of a mathematical analysis are **approximate**: it might be an expression consisting of a sequence of decreasing terms.

We are most concerned with the **leading terms** of a mathematical expression.

Example: The average running time of a program (in μsec) is

$$a_0 N \lg N + a_1 N + a_2$$

It's also true that the running time is

$$a_0 N \lg N + O(N)$$

For large N , we don't need to find the values of a_1 or a_2 .

The O-notation gives us a way to get an **approximate** answer for large N .

So, normally we can ignore quantities represented by the O-notation when there is a well-specified leading term.

Example: If we know that a quantity is $N(N-1)/2$, we may refer to it as “about” $N^2/2$.

1.3.6. Basic Recurrences

There is a basic method for analyzing the algorithms which are based on recursively decomposing a large problem into smaller ones.

The nature of a recursive program \Rightarrow its running time for input of size N will depend on its running time for smaller inputs.

This can be described by a mathematical formula called **recurrence relation**.

To derive the running-time, we solve the **recurrence relation**.

Formula 1: A recursive program that loops through the input to eliminate one item.

$$C_N = C_{N-1} + N \quad N \geq 2 \text{ with } C_1 = 1$$

Iteration Method

$$\begin{aligned} C_N &= C_{N-1} + N \\ &= C_{N-2} + (N-1) + N \\ &= C_{N-3} + (N-2) + (N-1) + N \\ &\quad \cdot \\ &\quad \cdot \\ &\quad \cdot \\ &= C_1 + 2 + \dots + (N-2) + (N-1) + N \\ &= 1 + 2 + \dots + (N-1) + N \\ &= \frac{N(N+1)}{2} \\ &\approx \frac{N^2}{2} \end{aligned}$$

Formula 2: A recursive program that halves the input in one step:

$$C_N = C_{N/2} + 1 \quad N \geq 2 \text{ with } C_1 = 0$$

Assume $N = 2^n$

$$\begin{aligned} C(2^n) &= C(2^{n-1}) + 1 \\ &= C(2^{n-2}) + 1 + 1 \\ &= C(2^{n-3}) + 3 \\ &= C(2^0) + n \\ &= C_1 + n = n \end{aligned}$$

$$C_N = n = \lg N$$

$$C_N \approx \lg N$$

Formula 3. This recurrence arises for a recursive program that has to make a linear pass through the input, before, during, or after it is split into two halves:

$$C_N = 2C_{N/2} + N \quad \text{for } N \geq 2 \text{ with } C_1 = 0$$

Assume $N = 2^n$

$$\begin{aligned} C(2^n) &= 2C(2^{n-1}) + 2^n \\ C(2^n)/2^n &= C(2^{n-1})/2^{n-1} + 1 \\ &= C(2^{n-2})/2^{n-2} + 1 + 1 \\ &\cdot \\ &\cdot \\ &\cdot \\ &= n \end{aligned}$$

$$\begin{aligned} \Rightarrow C_2^n &= n \cdot 2^n \\ C_N &= N \lg N \\ C_N &\approx N \lg N \end{aligned}$$

Formula 4. This recurrence arises for a recursive program that splits the input into two halves with one step.

$$C(N) = 2C(N/2) + 1 \quad \text{for } N \geq 2 \text{ with } C(1) = 0$$

Assume $N = 2^n$.

$$\begin{aligned} C(2^n) &= 2C(2^{n-1}) + 1 \\ C(2^n)/2^n &= 2C(2^{n-1})/2^n + 1/2^n \\ &= C(2^{n-1})/2^{n-1} + 1/2^n \\ &= [C(2^{n-2})/2^{n-2} + 1/2^{n-1}] + 1/2^n \\ &\cdot \\ &\cdot \\ &\cdot \\ &= C(2^{n-i})/2^{n-i} + 1/2^{n-i+1} + \dots + 1/2^n \end{aligned}$$

Finally, when $i = n - 1$, we get

$$\begin{aligned} C(2^n)/2^n &= C(2)/2 + 1/4 + 1/8 + \dots + 1/2^n \\ &= 1/2 + 1/4 + \dots + 1/2^n \\ &\approx 1 \end{aligned}$$

$$C(N) \approx N$$

Minor variants of these formulas can be handled using the same solving techniques.

But some recurrence relations that seem similar may be actually difficult to solve.

Notes on Series

There are some types of series commonly used in complexity analysis of algorithms.

- Arithmetic Series

$$S = 1 + 2 + 3 + \dots + n = n(n+1)/2 \approx n^2/2$$

$$S = 1 + 2^2 + 3^2 + \dots + n^2 = n(n+1)(2n+1)/6 \approx n^3/3$$

- Geometric Series

$$S = 1 + a + a^2 + a^3 + \dots + a^n = (a^{n+1} - 1)/(a - 1)$$

If $0 < a < 1$ then $S \leq 1/(1-a)$

and as $n \rightarrow \infty$, the sum approaches $1/(1-a)$.

- Harmonic sum

$$H_n = 1 + 1/2 + 1/3 + 1/4 + \dots + 1/n = \log_e n + \gamma$$

$\gamma \approx 0.577215665$ is known as Euler's constant.

Another series is also useful, particularly in working with trees.

$$1 + 2 + 4 + \dots + 2^{m-1} = 2^m - 1$$

Chapter 2. ALGORITHM CORRECTNESS

There are several good reasons for studying the correctness of algorithms.

- When a program is finished, there is no formal way to demonstrate its correctness. *Testing* a program can not guarantee its correctness.
- So, writing a program and proving its correctness *should go hand in hand*. By that way, when you finish a program, you can ensure that it is correct.

Note: Every algorithm depends for its correctness on *some specific properties*. To prove an algorithm correct is to prove that the algorithm preserves that specific property.

The study of correctness is known as *axiomatic semantics*, from Floyd (1967) and Hoare (1969).

Using the methods of axiomatic semantics, we can prove that a program is correct as rigorously as one can prove a theorem in logic.

2.1. PROBLEMS AND SPECIFICATIONS

2.1.1. Problems

A *problem* is a general question to be answered, usually having several parameters.

Example: The minimum-finding problem is ‘S is a set of numbers. What is a minimum element of S?’

S is a *parameter*.

An *instance* of a problem is an assignment of values to the parameters.

An *algorithm* for a problem is a step-by-step procedure for taking any instance of the problem and producing a correct answer for that instance.

An algorithm is *correct* if it is guaranteed to produce a correct answer to every instance of the problem.

2.1.2. Specification of a Problem

A good way to state a *specification* precisely for a problem is to give two Boolean expressions:

- the *precondition* states what may be assumed to be true initially and
- the *post-condition*, states what is to be true about the result.

Example:

Pre: S is a finite, non-empty set of integers.

Post: m is a minimum element of S.

We could write:

Pre: $S \neq \emptyset$

Post: $\exists m \in S$ and for $\forall x \in S, m \leq x$.

2.2. PROVING RECURSIVE ALGORITHMS

We should use *induction* on the size of the instance to prove correctness.

Example:

{ *Pre:* n is an integer such that $n \geq 0$ }

x := Factorial(n);

{ *Post:* $x = n!$ }

integer **function** Factorial(n: integer);

begin

if n = 0 **then**

 Factorial := 1;

else

 Factorial := n*Factorial(n-1);

end;

Property 2.1 For all integer $n \geq 0$, Factorial(n) returns $n!$.

Proof: by induction on n.

Basic step: $n = 0$. Then the test $n=0$ succeeds, and the algorithm returns 1. This is correct, since $0!=1$.

Inductive step: The inductive hypothesis is that Factorial(j) return $j!$, for all $j : 0 \leq j \leq n - 1$.

It must be shown that Factorial(n) return $n!$.

Since $n > 0$, the test $n = 0$ fails and the algorithm return $n*$ Factorial(n-1). By the inductive hypothesis, Factorial(n-1) return $(n-1)!$, so Factorial(n) returns $n*(n-1)!$, which equals $n!$.

Example Binary Search

boolean **function** BinSearch(l, r: integer, x: KeyType);

/* A[l..r] is an array */

var mid: integer;

begin

if l > r **then**

 BinSearch := false;

else

```

begin
  mid := (l + r) div 2;
  if x = A[mid] then BinSearch := true;
  else if x < A[mid] then
    BinSearch := BinSearch(l, mid - 1, x)
  else
    BinSearch := BinSearch(mid + 1, r, x)
  end;
end;

```

Property 2.2. For all $n \geq 0$, where $n = r - l + 1$ equals to the number of elements in the array $A[l..r]$, $\text{BinSearch}(l, r, x)$ correctly returns the value of the condition $x \in A[l..r]$.

Proof: By induction on n .

Basic step: $n = 0$. The array is empty, so $l = r + 1$, the test $l > r$ succeeds, and the algorithm return false. This is correct, because x cannot be present in an empty array.

Inductive step: $n > 0$. The inductive hypothesis is that, for all j such that $0 \leq j \leq n - 1$, where $j = r' - l' + 1$, $\text{BinSearch}(l', r', x)$ correctly returns the value of the condition $x \in A[l', r']$.

From $\text{mid} := (r + 1) \text{ div } 2$, it derives that $l \leq \text{mid} \leq r$. If $x = A[\text{mid}]$, clearly $x \in A[l..r]$, and the algorithm correctly returns *true*.

If $x < A[\text{mid}]$, since A is sorted we can conclude that $x \in A[l..r]$ iff $x \in A[l..mid-1]$. By the inductive hypothesis, this second condition is returned by $\text{BinSearch}(l, \text{mid} - 1, x)$. The inductive hypothesis does apply, since $0 \leq (\text{mid} - 1) - l + 1 \leq n - 1$.

The case $x > A[\text{mid}]$ is similar, and so the algorithm works correctly on all instances of size n .

2.3. PROVING ITERATIVE ALGORITHMS

Example:

```

{ Pre: true }
i := 1; sum := 0;
while i ≤ 10 do
  begin
    sum := sum + i;
    i := i + 1;
  end;
  10
{ Post: sum =  $\sum_{i=1}^{10} i$  }

```

The key step in correctness proving: to invent a condition, called the *loop invariant*. It is supposed to be true at the beginning and end of each iteration of the *while* loop.

The *loop invariant* of the above algorithm is

$$\text{sum} = \sum_{j=1}^{i-1} j$$

which expresses the relationship between the variables *sum* and *i*.

Property 3.1 At the beginning of the *i*th iteration of the above algorithm, the condition

$$\text{sum} = \sum_{j=1}^{i-1} j$$

holds.

Proof:

By induction on *i*.

Basic step: *k* = 1. At the beginning of the first iteration, the initialization statements clearly ensure that *sum* = 0 and *i* = 1. Since

$$0 = \sum_{j=1}^{1-1} j$$

the condition holds.

Inductive step: The inductive hypothesis is

$$\text{sum} = \sum_{j=1}^{i-1} j$$

at the beginning of the *i*th iteration.

Since it has to be proved that this condition holds after one more iteration, we assume that the loop is not about to terminate, that is *i* ≠ 10. Let *sum'* and *i'* be the value of *sum* and *i* at the beginning of (*i*+1)st iteration. We are required to show that

$$\text{sum}' = \sum_{j=1}^{i'-1} j$$

Since *sum'* = *sum* + *i* and *i'* = *i*+1, we have

$$\text{sum}' = \text{sum} + i$$

$$= \sum_{j=1}^{i-1} j + i$$

$$\begin{aligned}
 & i \\
 & = \sum_{j=1}^i j \\
 & i'-1 \\
 & = \sum_{j=1}^{i'-1} j
 \end{aligned}$$

So the condition holds at the beginning of the $(i+1)$ st iteration.

There is one more step to do:

- The postcondition must also hold at the end of the loop.

Consider the last iteration of the loop. At the end of it, the loop invariant holds. Then the test $i \leq 10$ fails and the execution passes to the statement after the loop. At that moment

$$\text{sum} = \sum_{j=1}^{i-1} j \wedge i = 11$$

holds.

From that we get,

$$\text{sum} = \sum_{j=1}^{10} j$$

which is the desired postcondition.

The loop invariant involves all the variables whose values change within the loop. But it expresses the *unchanging* relationship among these variables.

Guidance: The loop invariant may be obtained from the postcondition $Post$. Since the loop invariant must satisfy:

$$I \text{ and not } B \Rightarrow Post.$$

B and $Post$ are known. So, from B and $Post$, we can derive I .

Proving on Termination

The final step is to show that there is no risk of an infinite loop.

The method of proof is to identify some *integer quantity* that is strictly decreasing from one iteration to the next, and to show that when this become small enough the loop must terminate.

The integer quantity is called *bound function*.

In the body of the loop, the bound function must be positive (>0).

The suitable bound function for the summing algorithm is $11 - i$. This function is strictly decreasing and when it reaches 0, the loop must terminate.

The steps required to prove an iterative algorithm:

```
{ Pre }  
...  
while B do  
S  
{ Post }
```

is as follows:

1. Guess a condition I
2. Prove by induction that I is a loop invariant.
3. Prove that $I \text{ and not } B \Rightarrow \text{Post}$.
4. Prove that the loop is guaranteed to terminate.

Example

```
{ true }  
k := 1; r := 1;  
while  $k \leq 10$  do  
begin  
  r := r * 3;  
  k := k + 1;  
end;  
{ Post:  $r = 3^{10}$  }
```

The invariant: $r = 3^{k-1}$.

Bound function: $11 - k$

Chapter 3. ANALYSIS OF SOME SORTING AND SEARCHING ALGORITHMS

3.1. ANALYSIS OF ELEMENTARY SORTING METHODS

3.1.1. Rules of the Game

Let consider methods of sorting *file of records* containing *keys*. The key, which are parts of the records, are used to control the sort.

The objective: to rearrange the records so that their keys are ordered according to some ordering.

If the files to be sorted fits into memory (or if it fits into an array), then the sorting is called *internal*.

Sorting file from disk is called *external* sorting.

We'll be interested in the running time of sorting algorithms.

- The first four methods in this section require time proportional N^2 to sort N items.
- More advanced methods can sort N items in time proportional to $N \lg N$.

A characteristic of sorting is *stability* .A sorting methods is called *stable* if it preserves the relative order of equal keys in the file.

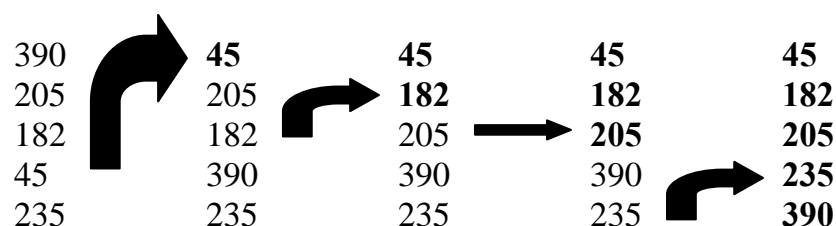
In order to focus on algorithmic issues, assume that our algorithms will sort *arrays of integers* into numerical order.

3.1.2. Selection Sort

The idea: “First find the *smallest* element in the array and exchange it with the element in the first position, then find the *second smallest* element and the exchange it with the element in the second position, and continue in this way until the entire array is ordered”.

This method is called *selection sort* because it repeatedly “selects” the smallest remaining element.

Figure 3.1.1. Selection sort.




```

procedure selection;
var i, j, min, t: integer;
begin
    for i :=1 to N-1 do
        begin
            min :=i;
            for j :=i+1 to N do
                if a[j]<a[min] then min := j;
            t :=a[min]; a[min] :=a[i];
            a[i] :=t;
        end;
    end;

```

The inner loop is executed the following number of times :

$$(N-1)+(N-2)+\dots+1 = N(N-1)/2 = O(N^2)$$

The outer loop is executed N-1 times.

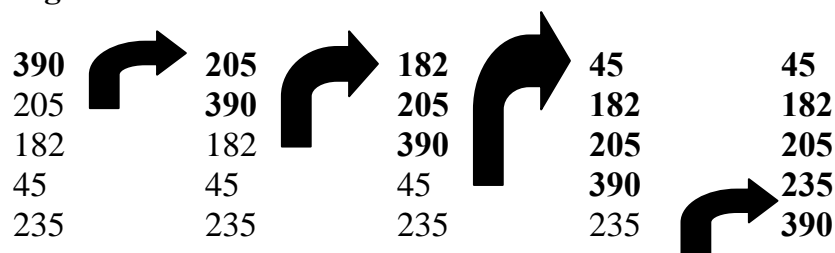
Property 3.1.1: Selection sort uses about N exchanges and $N^2/2$ comparisons.

Note: The running time of selection sort is quite insensitive to the input.

3.1.3. Insertion Sort

The idea: “The algorithm considers the elements one at a time, inserting each in its proper place among those already considered (keep them sorted)”.

Figure 3.1.2. Insertion sort.



```

procedure insertion;
var i, j, v: integer;
begin
    for i:=2 to N do
        begin
            v:=a[i]; j:= i;
            while a[j-1]> v do
                begin a[j] := a[j-1]; j:= j-1 end;
            a[j]:=v;
        end;
    end;

```

Note:

1. The procedure *insertion* doesn't work, because the *while* will run past the left end of the array when *v* is the smallest element in the array. To fix this, we put a “*sentinel*” key in *a[0]*, making it at least as small as the smallest element in the array.

2. The outer loop is executed *N*-1 times. The worst case occurs when the array is in reverse order, the inner loop is executed the following number of times:

$$(N-1) + (N-2) + \dots + 1 = N(N-1)/2 = O(N^2)$$

$$\text{Moves} = N(N-1)/2$$

$$\text{Comparisons} = N(N-1)/2$$

3. On the average, there is approximately $(i-1)/2$ comparisons in the inner loop. So, for the average case, the number of comparisons:

$$(N-1)/2 + (N-2)/2 + \dots + 1/2 = N(N-1)/4 = O(N^2)$$

Property 3.1.2: *Insertion sort uses about $N^2/2$ comparisons and $N^2/4$ exchanges in the worst case.*

Property 3.1.3: *Insertion sort uses about $N^2/4$ comparisons and $N^2/8$ exchanges in the average case.*

Property 3.1.4: *Insertion sort is linear for an “almost sorted” array.*

3.1.4. Bubble sort

It is also called *exchange sort*.

The idea: “keep passing through the array, exchanging *adjacent* elements, if necessary; when no exchanges are required on some pass, the array is sorted”.

Whenever the *largest* element is encountered during the first pass, it is exchanged with each of the elements to its right, until it gets into position at the right end of the array. Then on the second pass, the *second largest* element will be put into position, etc.

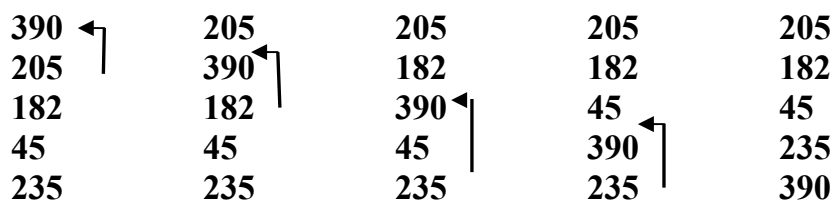


Figure 3.1.3. After the first pass of bubble sort

```
procedure bubble;  
var j,j, t: integer;  
begin
```

```

    for i := N downto 1 do
        for j := 2 to i do
            if a[j-1] > a[j] then swap(a[j],a[j-1]);
        end;
    end;

```

In the *average case* and in the *worst case*, the inner loop is executed the following of times:

$$(N-1) + (N-2) + \dots + 1 = N(N-1)/2 = O(N^2)$$

Property 3.1.5: Bubble sort uses about $N^2/2$ comparisons and $N^2/2$ exchanges in the average case and in the worst case.

Note: The running time of bubble sort depends on the input.

Bubble sort has two major drawbacks.

1. Its inner loop contains an exchange that requires three moves.
2. When an element is moved, it is always moved to an adjacent position.

Bubble sort is slowest sorting algorithm.

3.2. QUICKSORT

The basic algorithm of Quick sort was invented in 1960 by C. A. R. Hoare.

Quicksort is popular because it's not difficult to implement.

Quicksort requires only about $N \lg N$ operations on the average sort N items.

The drawbacks of Quicksort are that

- it is recursive, and
- it takes about N^2 operations in the worst case and
- it's fragile.

3.2.1. The Basic Algorithm

Quicksort is a “*divide and conquer*” method of sorting. It works by partitioning a file in two parts, then sorting the parts independently.

The algorithm has the following structure:

```

procedure quicksort1(left,right:integer);
var i: integer;
begin
    if right > left then
        begin
            i:= partition(left,right);
            quicksort(left,i-1);
        end
    end

```

```

        quicksort(i+1,right);
    end;
end;

```


The main point of the method is the *partition* procedure, which must rearrange the array to make the following three conditions hold:

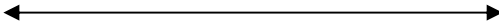
- (i) the element $a[i]$ is in its sorted place in the array for some i ,
- (ii) all the elements in $a[left]$, ..., $a[i-1]$ are less than or equal to $a[i]$
- (iii) all the elements in $a[i+1]$, ..., $a[right]$ are greater than or equal to $a[i]$

Example:

53	59	56	52	55	58	51	57	54
----	----	----	----	----	----	----	----	----

52	51	53	56	55	58	59	57	54
----	----	----	----	----	----	----	----	----





For the moment, we select the *first*, or *leftmost* element as the one to move to its sorted place (the *pivot*). Later, we will modify this selection process in order to obtain improved performance.

The refinement of the above algorithm is as follows:

```

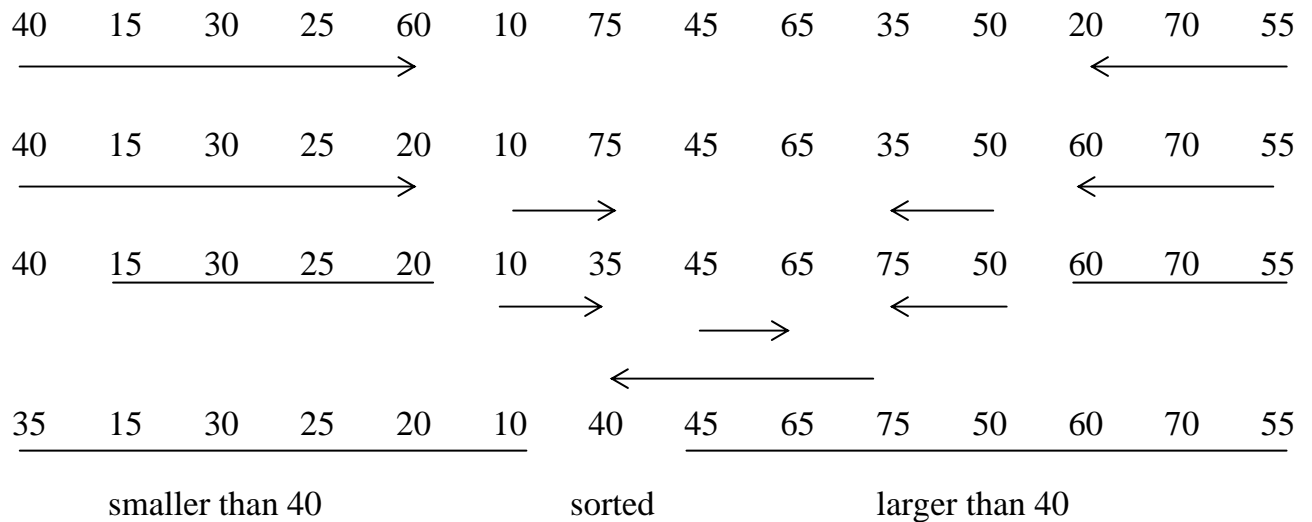
procedure quicksort2(left, right: integer);
var j, k: integer;
begin
    if right > left then
        begin
            j:=left; k:=right+1;
            repeat
                repeat j:=j+1 until a[j] >= a[left];
                repeat k:=k-1 until a[k] <= a[left];
                if j < k then swap(a[j],a[k])
            until j>k;
            swap(a[left],a[k]);

            quicksort2(left,k-1);
            quicksort2(k+1,right)
        end;
    end;

```

Note: A *sentinel* key is needed to stop the scan in the case that the partitioning element is the largest element in the file.

Example 1:



3.2.2. Performance Characteristics of Quicksort

•The Best Case

The best thing that could happen in Quicksort is that each partitioning divides the file exactly in half.

This would make the number of comparisons used by Quicksort satisfy the recurrence:

$$C_N = 2C_{N/2} + N.$$

The $2C_{N/2}$ covers the cost of sorting the two subfiles; the N is cost of examining each element.

From Chapter 1, the solution of this recurrence is

$$C_N \approx N \lg N.$$

The algorithm *quicksort2* simply selects the left element as the pivot.

•The Worst-Case

The worst case of Quicksort occurs when the list is already sorted.

Then the 1st element will require n comparisons to recognize that it remains in the 1st position. Furthermore, the first subfile will be empty, but the second subfile will have $n - 1$ elements. Accordingly, the 2nd element will require $n - 1$ comparisons to recognize that it remains in the 2nd position. And so on.

Consequently, there will be a total of

$$n + (n-1) + \dots + 2 + 1 = n(n+1)/2 = (n^2 + n)/2 = O(n^2).$$

comparisons.

So, the worst-case complexity of quicksort is $O(n^2)$.

•The Average-Case

The precise recurrence formula for the number of comparisons used by Quicksort for a random permutation of N elements is

$$C_N = (N+1) + \frac{1}{N} \sum_{k=1}^N (C_{k-1} + C_{N-k})$$

for $N \geq 2$ with $C_1 = C_0 = 0$

The $(N+1)$ term covers the cost of comparing the partitioning element with each of the others (two extra where the pointers cross). The rest comes from the fact that each element k is likely to be the partitioning element with probability $1/N$ after which we have random subsists of size $k-1$ and $N-k$.

First, $C_0 + C_1 + \dots + C_{N-1}$ is the same as
 $C_{N-1} + C_{N-2} + \dots + C_0$, so we have

$$C_N = (N+1) + \frac{2}{N} \sum_{k=1}^N C_{k-1}$$

We can eliminate the sum by multiplying both sides by N and subtracting the same formula by $N-1$:

$$NC_N - (N-1)C_{N-1} = N(N+1) - (N-1)N + 2C_{N-1}$$

This simplifies to

$$NC_N = (N+1)C_{N-1} + 2N$$

Dividing both sides by $N(N-1)$ gives a recurrence relation:

$$\begin{aligned} C_N/(N+1) &= C_{N-1}/N + 2/(N+1) \\ &= C_{N-2}/(N-1) + 2/N + 2/(N+1) \\ &\dots \\ &= C_2/3 + \sum_{k=3}^N 2/(k+1) \\ C_N/(N+1) &\approx 2 \sum_{k=1}^N 1/k \approx 2 \int_1^N 1/x \, dx = 2\ln N \\ C_N &\approx 2N\ln N \end{aligned}$$

Note that

$$\ln N = (\log_2 N) \cdot (\log_e 2) = 0.69 \lg N$$

$$2N\ln N \approx 1.38 N \lg N.$$

\Rightarrow The average number of comparisons is only about 38% higher than the best case.

Proposition. *Quicksort uses about $2N\ln N$ comparisons on the average.*

3.2.3. Removing Recursion

We can remove recursion in the basic algorithm of Quicksort by using a stack.

Any time we need a subfile to process, we *pop* the stack. When we partition, we create two subfiles to be processed which can be *pushed* on the stack.

```
procedure quicksort3;  
var t, i, left, right : integer;  
begin  
    left :=1; right:= N;  
    stackinit;  
    push(left); push(right);  
    repeat  
        if right > left then  
            begin  
                i:= partition(left,right);  
                if (i -left) > (right -i) then  
                    begin push(left); push(i-1);  
                        left := i+1 end  
                else  
                    begin push (i+1);push(right);  
                        right:=i-1 end;  
            end  
        else  
            begin right := pop; left := pop end;  
    until stackempty;  
end;
```

3.3. RADIX SORTING

For many applications, the *keys* can be numbers from some restricted range.

Sorting methods which take advantage of the digital properties of these numbers are called *radix sorts*.

These methods do not just compare keys: they process and compare *pieces of keys*.

Radix-sorting algorithms treats the keys as *numbers* represented in a *base-M number system* and work with individual digits of the numbers.

With most computers, it's convenient to work with $M=2$, rather than $M=10$.

3.3.1. Bits

Given a key represented as a *binary* number, the basic operation needed for radix sorts is *extracting* a contiguous set of bits from the number.

In machine language, bits are extracted from binary number by using bitwise "*and*" operation and *shifts*.

Example: The leading two bits of a ten-bit number are extracted by *shifting right* eight bit positions, then doing a bitwise “*and*” with the mask 0000000011.

In Pascal, these operation can be simulated by *div* and *mod*.

The leading two bits of a ten-bit number x are given by $(x \text{ div } 256) \text{ mod } 4$.

“shift x right k bit positions”	$x \text{ div } 2^k$
“zero all but the j right most bits of x ”	$(x \text{ div } 2^k) \text{ mod } 2^j$

In radix-sort algorithm, assume there exists a function $\text{bits}(x,k,j : \text{integer}) : \text{integer}$ which returns j bits which appear k bits from the right in x .

The basic method for radix sorting will examines the bits of the keys from left to right.

The idea: The outcome of comparisons between two keys depends only on the value of the bits in the first position at which they differ (reading from left to right).

3.3.2. Radix Exchange Sort

This is a recursive algorithm.

The rearrangement of the file is done very much like as in the partitioning in the Quicksort: scan from the left to find a key which starts with a 1 bits, scan from the right to find a key which starts with a 0 bit, exchange, and continue the process until the scanning pointers cross.

```

procedure radix_exchange(l, r, b : integer);
var t, i, j: integer;
begin
    if (r > l) and (b >= 0) then
        begin
            i := l; j := r;
            repeat
                while (bits(a[i], b, 1) = 0) and (i < j) do
                    i := i + 1;
                while (bits(a[j], b, 1) = 1) and (i < j) do
                    j := j - 1;
                swap(a[i], a[j]);
            until j = i;
            if bits(a[r], b, 1) = 0 then j := j + 1 ;
            radix_exchange(l, j - 1, b - 1);
            radix_exchange(j, r, b - 1);
        end
    end;

```

Assume that $a[1..N]$ contains positive integers less than 2^{32} (so that they can be represented as 31-bit binary numbers).

Then the *radix_exchange* (1,N,30) will sort the array.

The variable *b* keeps tracks of the bits being examined, ranging from 30 (leftmost) to 0 (rightmost).

The figure 3.3.1 shows the partition in terms of the binary representation of the keys. A simple five-bit code is used.

A 00001	A 00001	A 00001	A 00001	A 00001	A 00001
S 10011	E 00101	E 00101	A 00001	A 00001	A 00001
O 01111	O 01111	A 00001	E 00101	E 00101	E 00101
R 10010	L 01100	E 00101	E 00101	E 00101	E 00101
T 10100	M 01101	G 00111	G 00111	G 00111	G
I 01001	I 01001	I 01001	I 01001	I	I
N 01110	N 01110	N 01110	N 01110	L 01100	L 01100
G 00111	G 00111	M 01101	M 01101	M 01101	M 01101
E 00101	E 00101	L 01100	L 01100	N 01110	N 01110
X 11000	A 00001	O 01111	O 01111	O 01111	O 01111
A 00001	X 11000	S 10011	S 10011	P 10000	P
M 01101	T 10100	T 10100	R 10010	R 10010	R 10010
P 10000	P 10000	P 10000	P 10000	S 10011	S 10011
L 01100	R 10010	R 10010	T 10100	T	T
E 00101	S 10011	X 11000	X	X	X

Figure 3.3.1. Radix exchange sort (“left-to-right” radix sort)

3.3.3. Performance Characteristics of Radix Sorts

The running time of radix-exchange sort for sorting *N* records with *b*-bit keys are *Nb*.

On the other hand, one can think of this running time as the same as *NlogN*, since if the numbers are all different, *b* must be at least *logN*.

Property 3.3.1: *Radix-exchange sort uses on the average about $N \lg N$ bit comparisons.*

If the size is a power of two and the bits are random, then we expect half of the leading bits to be 0 and half to be 1, so the recurrence is $C_N = 2C_{N/2} + N$.

In radix-exchange sort, the partition is much more likely to be in the center than in Quicksort.

3.4. MERGESORT

First, we examine the process called *merging*, the operation of combining two sorted files to make a larger sorted file.

3.4.1. Merging

In many data processing environments, a large sorted data file is maintained.

New entries are regularly added to the large file.

A number of new entries are appended to the large file and the whole file is resorted.

This situation is suitable for *merging*.

Suppose that we have two sorted arrays $a[1..M]$ and $b[1..N]$ of integers. We want to merge into a third array $c[1..M+N]$.

```
i:= 1; j :=1;
for k:= 1 to M+N do
    if a[i] < b[j] then
        begin c [k] := a[i]; i:= i+1 end
    else
        begin c[k] := b[j]; j := j+1 end;
```

Note: The implementation uses $a[M+1]$ and $b[N+1]$ for *sentinel key* with values larger than all the other keys. When one of the two array is exhausted, the loop simply moves the rest of the remaining array into the c array.

3.4.2. Mergesort

Once we have a merging procedure, it's not difficult to use it as the basis for a recursive sorting procedure.

To sort a given file, divide it in half, sort the two halves (recursively) and then merge the two halves together.

The following algorithm sorts an array $a[1..r]$, using an auxiliary array $b[1..r]$,

```
procedure mergesort(1,r: integer);
var i, j, k, m : integer;
begin
    if r-1>0 then
        begin
            m:=(r+1)/2;
            mergesort(1,m); mergesort(m+1,r);
            for i := m downto 1 do b[i] := a[i];
            for j :=m+1 to r do
                b[r+m+1-j] := a[j];
            for k :=1 to r do
                if b[i] < b[j] then
                    begin a[k] := b[i] ; i := i+1 end
                else
                    begin a[k] := b[j]; j:= j-1 end;
            end;
        end;
end;
```

The algorithm manages the merging without sentinels by copying the second array into position back-to-back with the first, but in reverse order.

The file of sample keys is processed as in the Figure 3.4.1.

Performance Characteristics

Property 3.4.1: *Mergesort requires about $N \lg N$ comparisons to sort any file of N elements.*

For the recursive version, the number of comparisons is described by the recurrence $C_N = 2C_{N/2} + N$, with $C_1 = 0$.

$$C_N \approx N \lg N$$

```

A S O R T I N G E X A M P L E
A S
    O R
A O R S
    I T
        G N
        G I N T
A G I N O R S T

          E X
            A M
          A E M X
                L P
                E L P
          A E E L M P X
A A E E G I L M N O P R S T X

```

Figure 3.4.1. Recursive MergeSort

Property 3.4.2: *Merge sort uses extra space proportional to N .*

Note: Mergesort is stable while
Quicksort is not stable.

3.5. EXTERNAL SORTING

Sorting data organized as *files*, or sorting data stored in secondary memory is called *external sorting*.

External sorting is very important in database management systems (DBMSs).

3.5.1. Block and Block Access

The operating system divides secondary memory into equal-sized *block*. The size of blocks varies among operating systems, but around 512 to 4096 bytes.

The basic operation on files is

- to bring a single block to a buffer in main memory or
- to bring a block back to secondary storage.

When estimating the running time of algorithms that operate on data files, we have to consider *the number of times* we read a block into main memory or write a block onto secondary storage. Such an operation is called a *block access* or *disk access*.

3.5.2. External Sort-merge

The most commonly used technique for external sorting is the *external sort-merge* algorithm.

M: the number of *page frames* in the main memory-buffer (the number of disk *blocks* whose contents can be buffered in main memory).

1. In the first stage, a number of sorted *runs* are created.

```
i = 0;
repeat
  read M blocks of the file, or the rest of the
  file, whichever is smaller;
  sort the in-memory part of the file;
  write the sorted data to the run file  $R_i$ ;
  i = i+1;
until the end of the file.
```

2. In the second stage, the runs are *merged*.

• Special Case:

Suppose, the number of runs, N , $N < M$.

We can allocate *one* page frame to each run and have space left to hold *one* page of output.

The merge stage operates as follows:

read one block of each of the N files R_i into a buffer page in memory;

```
repeat
  choose the first record (in sort order)
  among all buffer pages;
  write the tuple to the output, and delete it
  from the buffer page;
  if the buffer page of any run  $R_i$  is empty
    and not end-of-file( $R_i$ ) then
      read the next block of  $R_i$  into the buffer
      page;
until all buffer pages are empty.
```

The output of the merge stage is the sorted file. The output file is buffered to reduce the number of disk write operation.

The merge operation is a generalization of the *two-way merge* used by the internal merge-sort algorithm. It merges N runs, so it is called an *n-way merge*.

- General Case:

In general, if the file is much larger than memory buffer, $N > M$.

It's not possible to allocate one page frame for each run during the merge stage. In this case, the merge is done in many *passes*.

Since there is enough memory for $M - 1$ input buffer pages, each merge can take $M - 1$ runs as input.

The initial *pass* works as follows:

The first $M - 1$ runs are merged to get a single run for the next pass. Then, the next $M - 1$ runs are similarly merged, and so on, until all the initial runs have been processed. At this point, the number of runs has been reduced by a *factor of $M - 1$* .

If this reduced number of runs is still $\geq M$, another pass is made with the runs created by the first pass as input.

Each pass *reduces* the number of runs by a factor of $M - 1$. These passes are repeated as many times as required, until the number of runs is less than M ; final pass then generates the sorted output.

Figure 3.5.1 illustrates the steps of the external sort-merge of an example file.

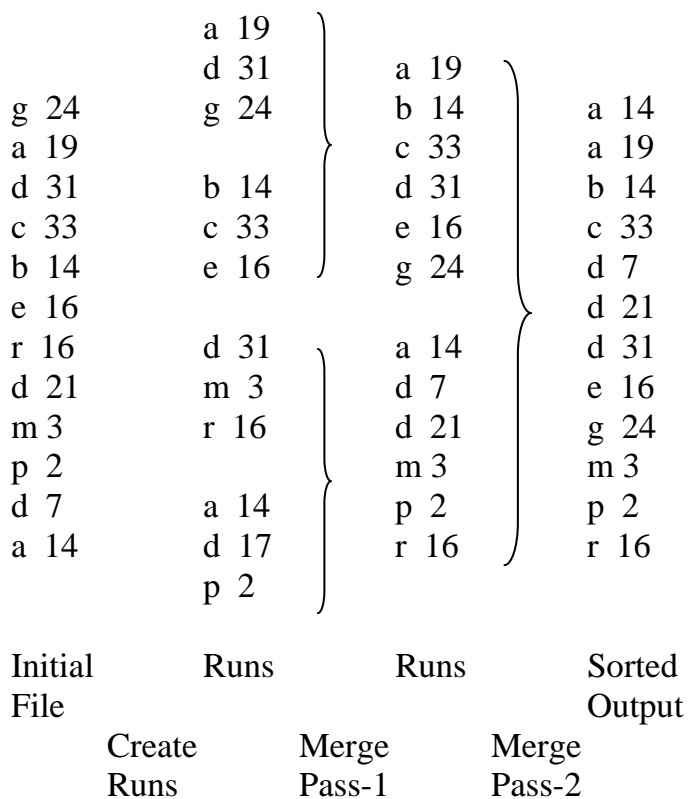


Figure 3.5.1 External sorting using sort-merge.

Assume that i) *one* record fits in a block and ii) memory buffer holds at most *three* page frames. During the merge stage, *two* page frames are used for input and *one* for output.

Complexity of External Sorting.

Let compute how many *block accesses* for the external sort-merge.

b_r : the number of blocks containing records of the file.

In the first stage, every block of the file is read and is written out again, giving a total of $2b_r$, disk accesses.

The initial number of runs is b_r/M .

The number of merge passes:

$$\lceil \log_{M-1}(b_r/M) \rceil$$

Each of these passes reads every block of the file once and writes it out once.

The total number of disk accesses for external sorting of the file is:

$$2b_r + 2b_r \lceil \log_{M-1}(b_r/M) \rceil = 2b_r(\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

3.6. ANALYSIS OF ELEMENTARY SEARCH METHODS

3.6.1. Linear Search

type node = **record** key, info: integer **end**;

var a: **array** [0..maxN] **of** node;

N: integer;

procedure initialize;

begin N: = 0 **end**;

function seq_search (v: integer): integer;

var j: integer;

begin

 a[N+1].key: = v; /*sentinel */

 j := 0;

repeat j: = j + 1 **until** v = a[j].key;

 seq_search: = j

end;

function seq_insert (v: integer): integer;

begin

 N: = N+1; a[N].key : = v;

 seq_insert: = N

end;

If the search is unsuccessful, the function *sep_search* returns the value $N+1$.

Property 3.6.1: *Sequential search uses $N+1$ comparisons for an unsuccessful search and about $N/2$ comparisons for a successful (on the average).*

Proof:

Worst case

Clearly, the worst case occurs when v is the last element in the array or is not there at all.

In either situation, we have $C(N) = N$ or $N+1$.

Accordingly, $C(n) = n$ is the worst-case complexity of the linear search algorithm.

Average Case

Here we assume that v does appear in the array, and that is equally likely to occur at any position in the array. Accordingly, the number of comparisons can be any of the numbers 1, 2, 3, ..., N , and each numbers occurs with probability $p = 1/N$. Then

$$\begin{aligned} C(N) &= 1.(1/N) + 2.(1/N) + \dots + N.(1/N) \\ &= (1 + 2 + \dots + N).(1/N) \\ &= (1+2+\dots+N)/N \\ &= N(N+1)/2.(1/N) \\ &= (N+1)/2. \end{aligned}$$

3.6.2. Binary Search

This method is applied when the array is sorted.

```
function binarysearch(v:integer): integer;  
var x, l, r: integer;  
begin  
  l := 1; r := N;  
  repeat  
    x := (l+r) div 2  
    if v < a[x].key then r := x - 1 else l := x+1  
  until (v = a[x].key) or (l>r);  
  if v = a[x].key then  
    binarysearch := x  
  else  
    binarysearch := N+1  
end;
```

The recurrence relation in this algorithm is $C_N \equiv C_{N/2} + 1$

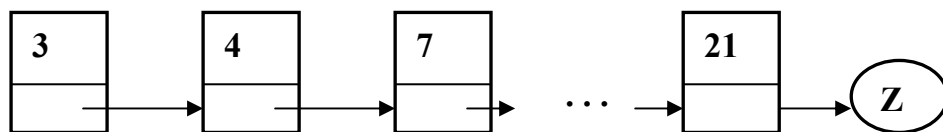
Property: *Binary search never uses more than $\lg N + 1$ comparisons for either successful or unsuccessful search.*

Chapter 4. ANALYSIS OF SOME ALGORITHMS ON DATA STRUCTURES

4.1. SEQUENTIAL SEARCHING ON A LINKED LIST

Sequential searching can be achieved using a *linked-list representation* for the records.

One advantage: it's easy to keep the list sorted, which leads to a quicker search:



Convention: Z is a dummy node. The last node of the linked list will point to z and z will point to itself.

```

type link = ↑ node
      node = record key, info: integer;
              next: link
            end;

var head, t, z: link;
i: integer;
procedure initialize;
begin
  new(z); z↑.next := z;
  new(head); head↑.next := z
end;

function listsearch(v: integer; t: link): link;
begin
  z↑.key := v;
  repeat t := t↑.next until v ≤ t↑.key;
  if v = t↑.key then listsearch := t
  else listsearch := z
end;

function listinsert (v: integer; t: link): link;
begin
  z↑.key := v;
  while t↑.next↑.key < v do t := t↑.next;
  new(x); x↑.next := t↑.next; t↑.next := x;
  x↑.key := v;
  listinsert := x;
end;
  
```


With a sorted list, a search terminates unsuccessfully when a record with a key larger than the search key is found.

Property 4.1.1: *Sequential search (sorted list implementation) uses about $N/2$ comparisons for both successful and unsuccessful search (on the average).*

Proof:

For successful search, if we assume that each record is equally likely to be sought, the average number of comparisons is:

$$(1 + 2 + \dots + N)/N = N(N+1)/(2N) = (N+1)/2.$$

For unsuccessful search, if we assume that the search is equally likely to be terminated by the tail node z or by each of the elements in the list, then average number of comparisons is:

$$(1 + 2 + \dots + (N+1))/(N+1) = (N+2)(N+1)/(2(N+1)) = (N+2)/2.$$

4.2. BINARY SEARCH TREE

In a binary search tree,

- all records with smaller keys are in the left subtree and
- all records in the right subtree have the larger (or equal) key values.

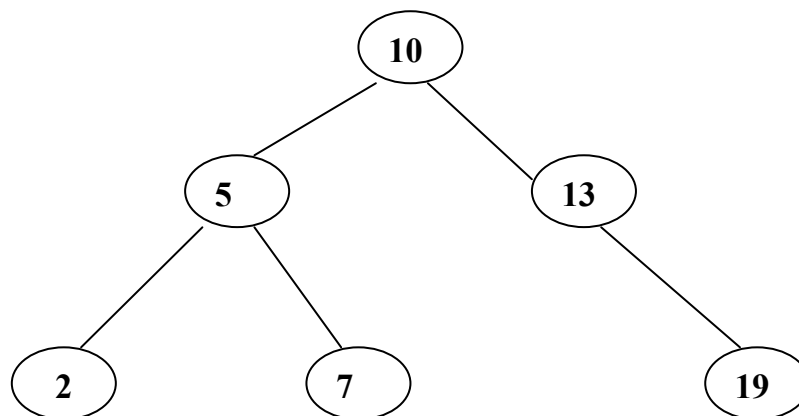


Figure 4.2.1 A Binary Search Tree

Search Operation

type link = \uparrow node;

node = **record** key, info: integer;

l, r: link **end**;

var t, head, z: link;

function treesearch(v: integer, x: link): link;

/* search the node with the key v in the binary search tree x */

begin

while v <> x \uparrow .key **and** x <> z **do**

if v < x \uparrow .key **then** x := x \uparrow .l

else x := x \uparrow .r

```

treesearch: = x
end;

```

Note When the search is unsuccessful, the function *treesearch* returns the dummy link *z*.

Tree Initialization

The empty tree is represented by having the right link of *head* point to *z*.

```

procedure tree_init;
begin
  new(z); z↑.l: = z; z↑.r: = z;
  new(head); head↑.key: = 0; head↑.r: = z;
end;

```

Insert Operation

To insert a node into the tree, we do an unsuccessful search for it, then attach it in place of *z* at the point at which the search terminated.

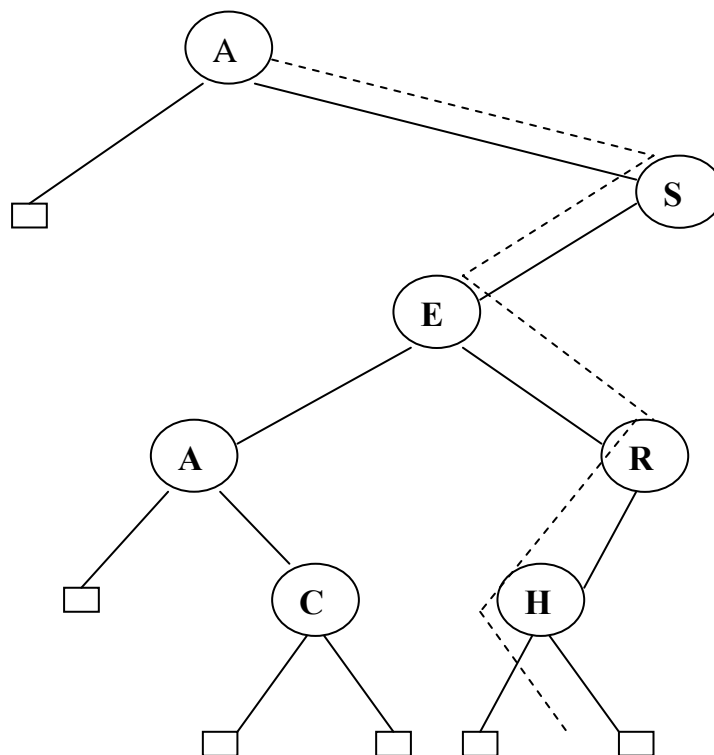


Figure 4.2.2 Searching (for I) in a binary search tree

```

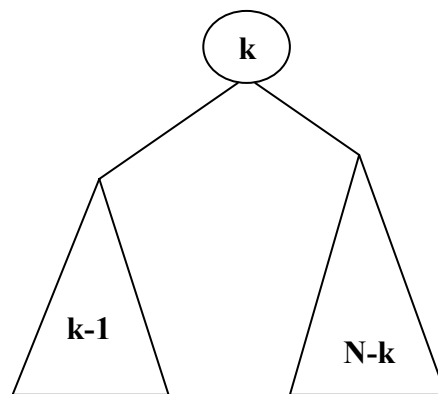
procedure tree_insert (v: integer; x: link): link;
var p: link;
begin
  repeat
    p: = x;
    if v < x↑.key then x: = x↑.l else x: = x↑.r
  until x = z;

```


$$C_N = N-1 + (1/N) \sum_1^N (C_{k-1} + C_{N-k}) \quad \text{with } C_1 = 1.$$

The $N-1$ comes from the fact that the root contributes 1 to the path length of each of the other $N-1$ nodes.

The second term comes from observing that the key at the root (the first inserted) is equal likely to be the k -th largest, leaving two random subtrees of size $k-1$ and $N-k$.



The recurrence is very much the same we solve for Quicksort, and it is easily be solved in the same way to derived the stated results.

So the average path length of the tree with N nodes is

$$C_N \approx 2N \ln N.$$

Accordingly, the average path length of a node is $2 \ln N$.

In other words, a search or insertion requires about $2 \ln N$ comparisons, on the average, in a tree built from N random keys.

Property *In the worst case, a search in a binary search tree with N keys can require N comparisons.*

Deletion Operation

To delete a node is easy if the node has no children or if it has just one child.

To delete a node with two children is quite complex: we have to replace it with the *next highest* key (the *leftmost* node in its *right subtree*).

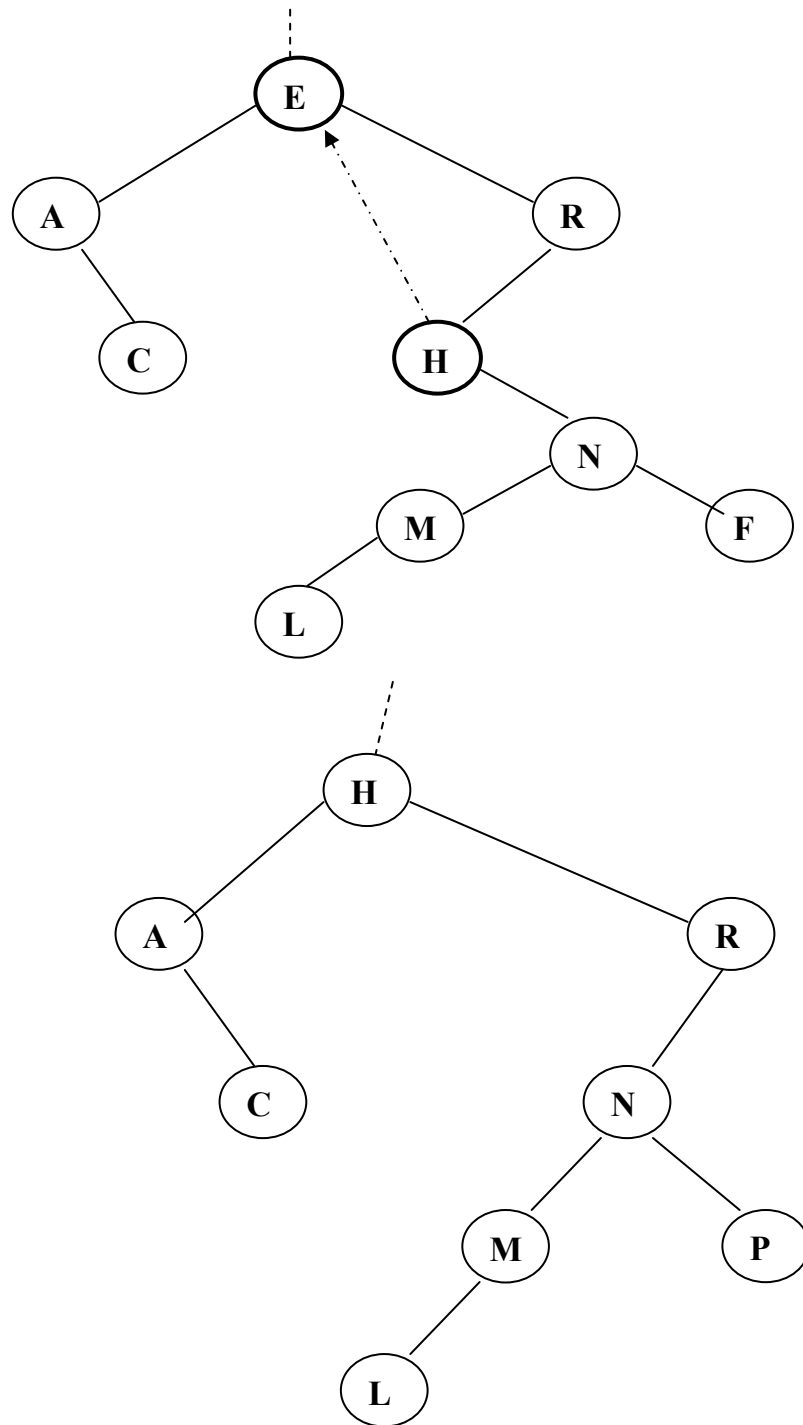


Figure 4.2.4 Deletion (of E) from a binary search tree

4.3. PRIORITY QUEUES AND HEAPSORT

A data structure which supports the operations of inserting a new element and deleting the largest element is called a *priority-queue*.

We want to build and maintain a data structure containing records with numerical keys (*priorities*) and supporting some of the following operations:

- Construct a priority queue from N given items.

- *Insert* a new item.
- *Remove* the largest item.
- Replace the largest item with a new item
- Change the priority of an item.
- Delete an arbitrary specified item.
- Join two priority queues into one larger one.

4.3.1. Heap Data Structure

The data structure that can support the *priority queue operations* stores the records in an array so that:

each key must be larger than the keys at two other specific positions. In turn, each of those keys must be larger than two more keys, and so on.

This ordering is very easy to see if we draw the array in a *tree structure* with lines down from each key to the two keys known to be smaller.

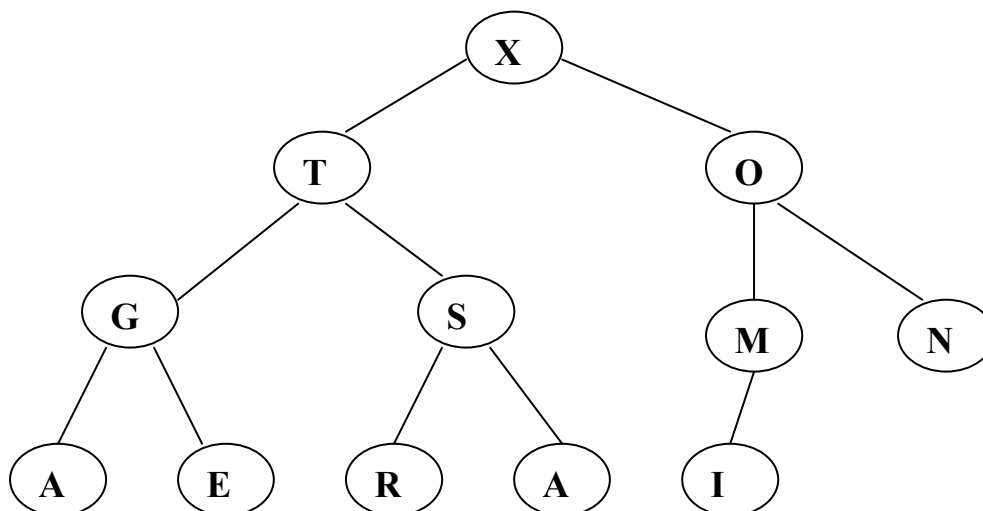


Figure 4.3.1. Complete tree representation of a heap.

We want the keys in the tree to satisfy the *heap condition*:

The key in each node should be larger than (or equal to) the keys in its children (if it has any). This implies the largest key is in the root.

We can represent the tree with an array by putting the root at position 1, its children at positions 2 and 3, the nodes at the next level in positions 4, 5, 6 and 7, etc.,

k	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	X	T	O	G	S	M	N	A	E	R	A	I

It's easy to get from a node to its parent and children.

- The parent of the node in position j is in position $j \div 2$.
- The two children of the node in position j are in position $2j$ and $2j+1$.

A *heap* is a binary tree, represented as an array, in which every node satisfies the heap condition. In particular, the *largest key* is always in the 1st position in the array.

All the algorithms operate along some *path* from the root to the bottom of heap. \Rightarrow In a heap of N nodes, all paths have about $\lg N$ nodes on them.

4.3.2. Algorithms on Heaps

There are 2 important operation on heap : *insert* and *remove the largest element*.

- The *insert* operation.

This operation will increase the size of the heap by one, N must be incremented.

Then the record to be inserted is put into $a[N]$, but this may violate the heap property.

If the heap property is violated, the violation can be fixed by exchanging the new node with its parent. This may, in turn, cause a violation, and thus can be fixed in the same way.

```
procedure upheap(k:integer)
var v: integer;
begin
    v := a[k]; a[0] := maxint;
    while a[k div 2] <= v do
        begin a[k] := a[k div 2]; k := k div 2 end;
    a[k] := v
end;
```

```
procedure insert(v:integer);
begin
    N := N+1; a[N] := v; upheap(N)
end;
```

The procedure *upheap(k)* can fix the heap condition violation at k .

A sentinel key must be put in $a[0]$ to stop the loop for the case v is greater than all the keys in the heap .

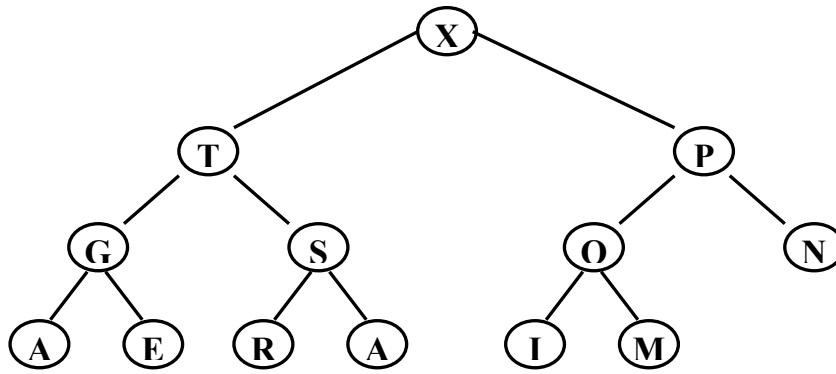


Figure 4.3.2 Inserting a new element (P) into a heap.

- The “*remove the largest*” operation.

The operation will decrease the size of the heap by one. It decrements N .

But the largest element (i.e. $a[1]$) is to be removed and replaced with the element that was in $a[N]$. If the key at the root is too small, it must be *moved down* the heap in order to satisfy the heap property.

The *downheap* procedure moves down the heap, exchanging the node at k with the larger of its two children, if necessary and stopping when the node at k is larger than children or the bottom is reached.

```

procedure downheap(k: integer);
label 0 ;
var j, v : integer;
begin
    v := a[k];
    while k <= N div 2 do
        begin
            j := 2*k;
            if j < N then if a[j] < a[j+1] then
                j := j+1;
            if v >= a[j] then goto 0;
            a[k] := a[j]; k := j;
        end;
    0: a[k] := v;
end;

function remove: integer;
begin
    remove := a[1];
    a[1] := a[N]; N := N-1;
    downheap(1);
end;
  
```

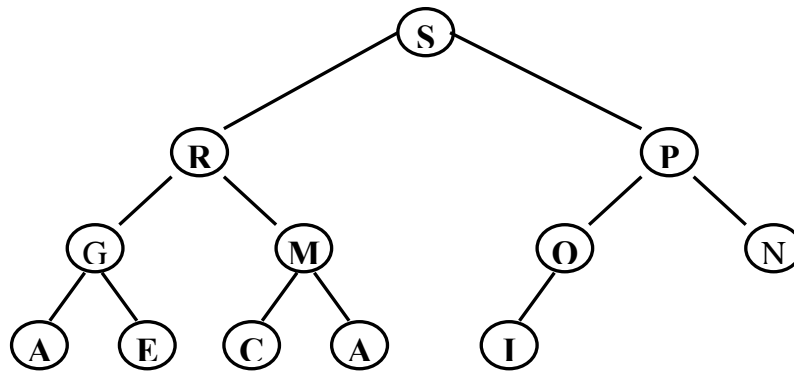



Figure 4.3.3 Removing a largest element in a heap.

Property 4.3.1: *All the operations insert, remove, downheap, upheap require less than $2\lg N$ comparisons when performed on a heap of N elements.*

All these operations involves moving along a *path* between the root and the bottom of the heap, which includes not more than $\lg N$ elements for a heap of size N .

The factor of 2 comes from *downheap*, which makes *two* comparisons in its inner loop; the other operations requires only $\lg N$ comparisons.

4.3.3. Heapsort

The idea is simply (1) to build a heap containing the elements to be sorted and then (2) to remove them all in the order.

M :the size of the heap and N : the number of elements to be sorted.

$N:=0$;

for $k:= 1$ **to** M **do**

insert($a[k]$); /* construct the heap */

for $k:= M$ **downto** 1 **do**

$a[k]:=$ remove; /*putting the element removed into the array a */

Property 4.3.3: *Heapsort uses fewer than $3M\lg M$ comparisons to sort M elements.*

This upper bound immediately comes from the property 4.3.1

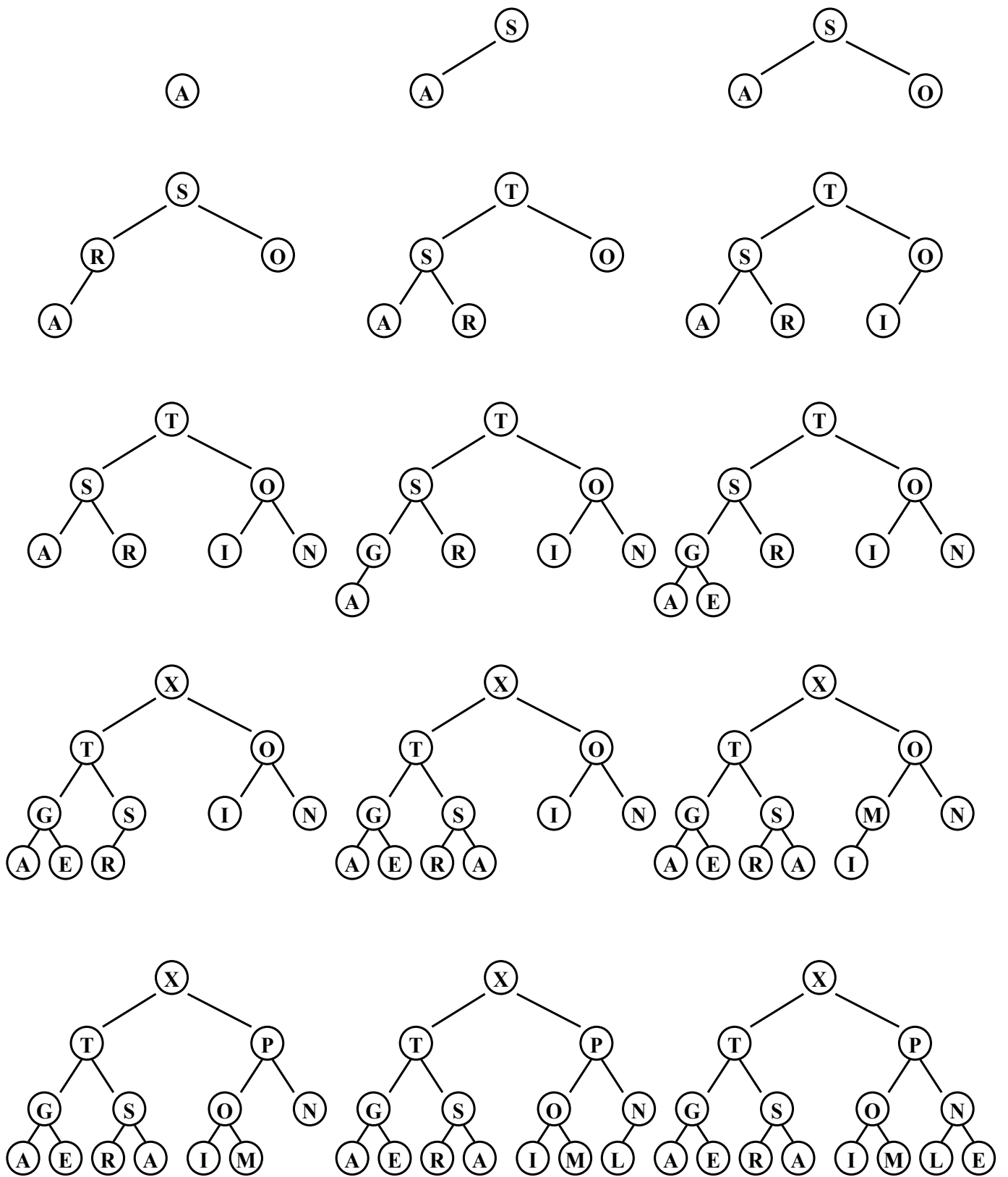


Figure 4.3.4 Top-down heap construction.



4.4. HASHING

Hashing is a method for directly referencing records in a table by doing arithmetic transformations on keys into table addresses.

If the keys are distinct integers from 1 to N, then we can store the record with the key i in table position i , ready for immediate access with the key value.

The first step in a search using hashing is to compute a *hash function* which transforms the search key into a table address.

Ideally, different keys should map to different addresses, but no hash function is perfect and two or more different keys hash to the same table address.

The second step of a hashing search is a *collision-resolution* process which deals with the case that two or more different keys hash to the same table address.

4.4.1. Hash Functions

Hash function is a function which transforms keys into integers in the range $[0 .. M-1]$, where M is number of records that can fit into the amount of memory available.

An ideal hash function is one which is

- easy to compute and
- approximates a “random” function.

Suppose that we have a large number which corresponds to our key.

The most commonly used method for hashing is to choose M to be *prime* and for any key k , compute.

$$h(k) = k \bmod M$$

This is a straightforward method which is easy to compute and spreads the key values out well.

Example: Table size = 101. Each of key consists of 4 characters. If the key (“AKEY”) is encoded to the simple 5-bit code, we may view it as the binary number.

00001 01011 00101 11001

$$1 \times 32^3 + 11 \times 32^2 + 5 \times 32^1 + 25 \times 32^0$$

which is equivalent to 44217 in decimal.

Now, $44217 \equiv 80 \bmod 101$, so our key hashes to position 80 in the table.

Why does the hash table size M have to be prime? The reason is that *we want to take all the characters of a key into account*.

In the above example, if $M = 32$, the hash function of any key is simply the value of its last character!

If the key is a long alphanumeric string, we can still compute a hash function by transform the key piece by piece.

```
h: = key[1]
for j:= 2 to key_size do
begin
  h:= ((h*32) + key[j]) mod M; /*25 is 32, used for 5-bit code */
end;
```

4.4.2. Separate Chaining

In hashing method, we have to decide how to handle the case when two keys hash to the same address.

The most simple method: to build for each table address a *linked list* of the records whose keys hash to that address.

Since the keys which hash to the same table position are kept in a linked list, they should be kept in order.

```
type link = ↑ node;
      node = record key, info: integer;
              next: link end;
var heads: array [0..M] of link; t, x: link;

procedure initialize;
var i: integer;
begin
  new (z); z↑.next: = z;
  for i: = 0 to M-1 do
    begin new (heads [i]); heads [i]↑.next: = z
    end;
end;
```

Property. *Separate chaining reduces the number of comparisons for sequential search by a factor of M (on average), using extra space for M links.*

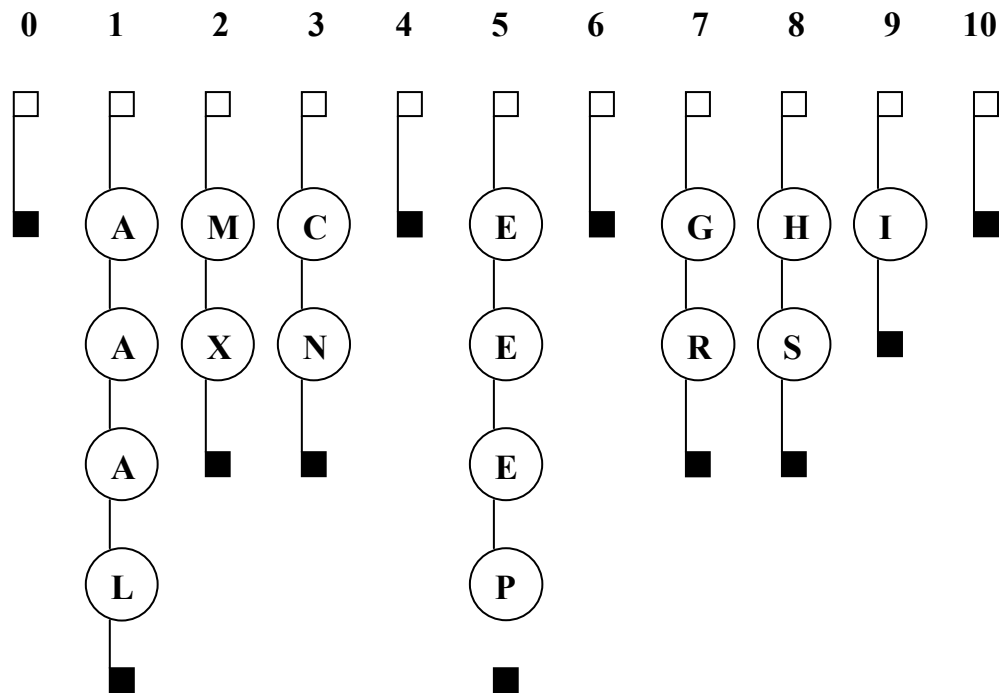
Proof:

If N , the number of keys in the table, is much larger than M , then the average length of the lists is approximately N/M . So the number of comparisons for linear search on the linked list in average case is $(N/M)/2$.

Therefore, the number of comparisons for the search is reduced by the factor of M .

$M = 11, N = 17$

Key:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
Value:	1	19	5	1	18	3	8	9	14	7	5	24	1	13	16	12	5
Hash:	1	8	5	1	7	3	8	9	3	7	5	2	1	2	5	1	5



4.4.3. Linear Probing

There are several methods that stores N records in a table of size $M > N$, relying on empty places in the table to deal with collision. Such methods are called *open addressing* hashing methods.

The simplest open-addressing method is called *linear probing*: when there is a collision, then just probe the *next* position in the table, i.e., compare the key in the record against the search key.

There are three outcomes of the probe:

- If the keys match, the search terminates successfully.
- If there is no record there, the search terminates unsuccessfully.
- Otherwise, probe the next position, continuing until either the search key or an empty position is found.

procedure hash_initialize;

var i: integer;

begin

for i: = 0 **to** M **do** a[i].key:= maxint; /* the special value maxint means an empty position */

end;

```

function hash_insert (v: integer): integer;
var x: interger
begin
    x:= h(v);
    while a[x].key <> maxint do /* collision */
        x: = (x+1) mod M;
    a[x].key: = v;
    hash-insert: = x;
end;

```

```

function hash_search(v: integer): integer;
var x: integer;
begin
    x: = h(v);
    while a[x].key <> maxit and a[x].key <> v do
        x: = (x+1) mod M;
    if a[x].key: = v then hash_search: = x
    else hash_search: = M;
end;

```

The table size for linear probing is greater that for separate chaining, since we must have $M > N$, but the total amount of memory space is less, since no links are use.

M = 19, N = 17

Key: A S E A R C H I N G E X A M P L E
Hash: 1 0 5 1 18 3 8 9 14 7 5 5 1 13 16 12 5

S
A
A
C
A
E
E
G
H
I
X
E
L
M
N
P
R

Property. *Linear probing uses less than five probes, on the average, for a hash table which is less than two-thirds full.*

The exact formula for the average number of probes required for an unsuccessful search:
 $\frac{1}{2} + \frac{1}{2(1 - \alpha)^2}$ with $\alpha = N/M$
 if $\alpha = 2/3$, we get 5 probes.

4.5. STRING MATCHING ALGORITHMS

String Matching: finding all occurrences of a pattern in a text.

The *text* is an array $T[1..n]$ of length n and the *pattern* is an array $P[1..m]$ of length m .

The elements of P and T are characters taken from a finite *alphabet* Σ .

Pattern P occurs with shift s in the text T (or, equivalently, P occurs beginning at position $s+1$ in text T) if $0 \leq s \leq n - m$ and $T[s+1 \dots s+m] = P[1..m]$.

If P occurs with shift s in T , then we call s a *valid shift*; otherwise, we call s an *invalid shift*.

The string-matching problem is the problem of finding *all valid shifts* with which a given pattern P occurs in a given text T .

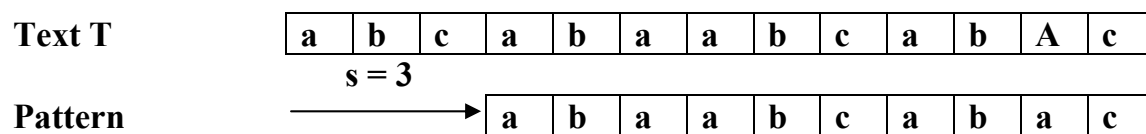


Figure 4.5.1

Notation and Terminology

Σ^* : the set of all finite-length strings formed using characters from Σ . Empty string is denoted by ϵ .

$|x|$ means the length of string x .

xy denotes the *concatenation* of two strings x and y .

String w is a *prefix* of a string x , denote $w \subset x$, if $x = wy$ for some string $y \in \Sigma^*$.

String w is a *suffix* of a string x , denote $w \supset x$, if $x = yw$ for some string $y \in \Sigma^*$.

4.5.1. The Naive String Matching Algorithm

The naive algorithm finds all valid shifts using a *loop* that checks the conditions $P[1..m] = T[s+1..s+m]$ for each of the $n - m + 1$ possible values of s .


```

procedure NATIVE-STRING-MATCHING(T,P);
begin
  n: = |T|;  m: = |P|;
  for s:= 0 to n – m do
    if P[1..m] = T[s+1,..,s+m] then
      print “Pattern occurs with shift” s;
end

```

$s=0$ | a | c | a | a | b | c |
 | ↗
 | a | a | b |

$s=1$ | a | c | a | a | b | c |
 ↗
 → | a | a | b |

$s=2$ | a | c | a | a | b | c |
 | | |
 → | a | a | b |

$s=3$ | a | c | a | a | b | c |
 | ↗
 → | a | a | b |

Figure 4.5.2

The NAIVE STRING MATCHER takes time $O((n - m + 1)m)$ in the worst case.

It is not an optimal procedure for this problem.

It's inefficient because information gained about the text for one value of s is ignored in considering other values of s .

4.5.2. The Rabin-Karp algorithm

The algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers *modulo* a third number.

Assume that $\Sigma = \{0, 1, 2, \dots, 9\}$, so that each character is a decimal digit. (In general case, each character is a digit in *radix-d* notation, where $d = |\Sigma|$.)

We can view a string of k consecutive characters as representing a *length-k* decimal number. The character string 31415 corresponds to the decimal number 31,415.

Give a pattern $P[1..m]$, let p be its corresponding decimal value.

Give a text $T[1..n]$, let t_s be the decimal value of the *length-m* substring $T[s+1...s+m]$, for $s = 0, 1, \dots, n-m$.

Certainly, $t_s = p$ iff $T[s+1 \dots s+m] = P[1..m]$ and s is a valid shift iff $t_s = p$

We can compute p in time $O(m)$ using Horner's rule:

$$p = P[m] + 10*(P[m-1] + 10*(P[m-2] + \dots + 10*(P[2] + 10*P[1])\dots))$$

The value t_0 can be similarly computed from $T[1..m]$ in time $O(m)$.

Notice that t_{s+1} can be computed from t_s :

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \quad (5.1)$$

Example: If $m = 5$ and $t_s = 31415$, then we want to remove the high-order digit $T[s+1] = 3$ and bring in the new low-order digit 2 to obtain.

$$t_{s+1} = 10(31415 - 10000.3) + 2 = 14152$$

Each execution of equation (6.1) takes a constant number of arithmetic operations.

Computing t_1, t_2, \dots, t_{n-m} costs $O(n-m)$ times.

Thus, p and t_0, t_1, \dots, t_{n-m} can all be computed in time $O(m) + O(m) + O(n-m) \approx O(n + m)$.

But p and t_s may be too large to work. To cure this problem, compute p and the t_s 's modulo a suitable modulus q .

The q is typically chosen as a *prime* such that $10q$ just fits one computer word.

In general, with a d -ary alphabet $\{0, 1, \dots, d-1\}$, we choose q so that dq fits within a computer word.

And the recurrence equation (5.1) becomes:

$$t_{s+1} = d(t_s - hT[s+1]) + T[s+m+1] \bmod q \quad (5.2)$$

where $h = d^{m-1} \pmod{q}$

However, $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$.

On the other hand, if $t_s \not\equiv p \pmod{q}$ then we definitely have $t_s \neq p$, so that shift s is invalid. We can use the test $t_s \equiv p \pmod{q}$ to rule out invalid shifts s .

Any shift s that satisfies $t_s \equiv p \pmod{q}$ must be tested further to see if s is really valid or we just have a *spurious hit*.

$$\begin{array}{cccccccccccccccc} |2| & |3| & |5| & |9| & |0| & |2| & |3| & |1| & |4| & |1| & |5| & |2| & |6| & |7| & |3| & |9| & |9| & |2| & |1| \\ & & & & & & \underbrace{\hspace{1.5cm}} & & & & & & & & & & & & \\ & & & & & & \downarrow & & & & & & & & & & & & \\ & & & & & & |7| & & & & & & & & & & & & \end{array}$$

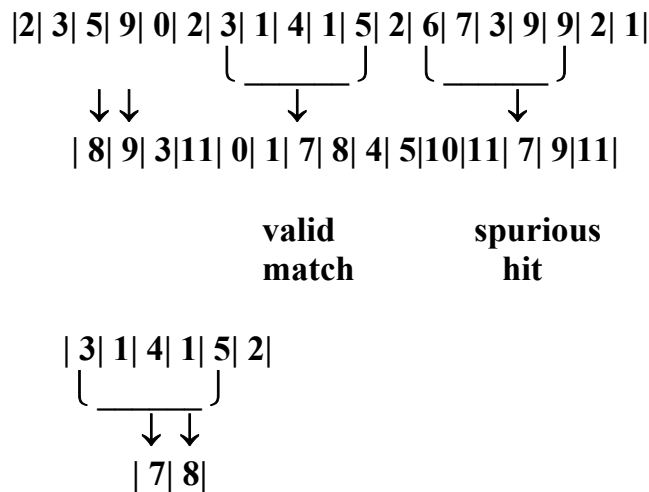


Figure 4.5.3 The Rabin-Karp algorithm

$$14152 = (31415 - 3 \times 10000) \times 10 + 2 \pmod{13} = 8 \pmod{13}$$

procedure RABIN-KARP-MATCHER(T, P, d, q);

/* T is the text, P is the pattern, d is the radix and q is the prime */

begin

n: = |T|; m: = |P|;

h: = $d^{m-1} \pmod{q}$;

p: = 0; t₀: = 0;

for i: = 1 **to** m **do**

begin

p: = (d*p + P[i]) mod q;

t₀: = (d*t₀ + T[i]) mod q

end

for s: = 0 **to** n - m **do**

begin

if p = t_s **then** /* there may be a hit */

if P[1..m] = T[s+1..s+m] **then**

Print "Pattern occurs with shift "s;

if s < n - m **then**

t_{s+1}: = (d(t_s - T[s + 1]h) + T[s+m+1]) mod q

end

end

The running time of RABIN-KARP-MATCHER is $O((n - m + 1)m)$ in the worst-case since the algorithm explicitly verifies each valid shift.

In many applications, we expect few valid shifts and so the expected running time is $O(n+m)$ plus the time required to process spurious hits.

Chapter 5. ANALYSIS OF GRAPH ALGORITHMS

5.1. ELEMENTARY GRAPH ALGORITHMS

Many problems are naturally formulated in terms of objects and connections between them. A *graph* is a mathematical object that accurately models such situations.

Transportation
Telecommunication
Electricity

Computer Network
Databases
Compiler
Operating systems
Graph theory

Graphs in algorithmic point of view

5.1.1. Glossary

A *graph* is a collection of *vertices* and *edges*. Vertices are simple objects that can have names and other properties; and edge is a connection between two vertices.

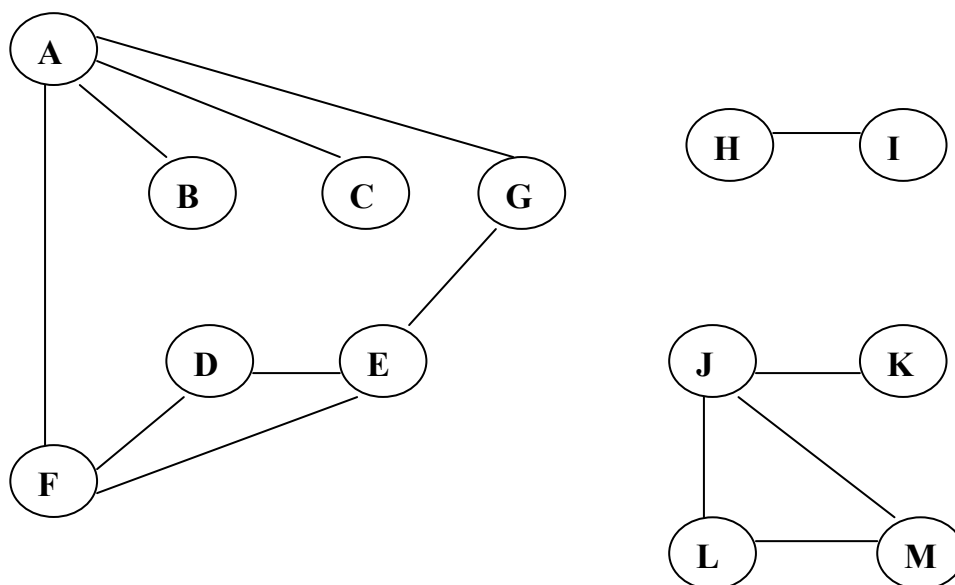


Figure 5.1.1.

A *path* from vertex x to y in a graph is a list of vertices in which successive vertices are connected by edges in the graph.

A graph is *connected* if there is a path from every node to every other node in the graph.

A graph which is not connected is made up of *connected components*.

A *simple path* is a path in which no vertex is repeated.

A *cycle* is a path that is simple except that the first and the last vertex are the same (a path from a point back to itself).

A graph with no cycle is called a *tree*. A group of disconnected trees is called a *forest*.

A *spanning tree* of a graph is a subgraph that contains all the vertices but only enough of the edges to form a tree.

Let denote the number of vertices in a graph by V , the number of edges by E . Note that E can range from 0 to $V(V-1)/2$. (Proof by induction).

Graphs with all edges present are called *complete graphs*.

Graphs with relatively few edges are called *sparse*; graphs with relatively few edges missing are called *dense*.

Graphs as described to this point are called *undirected graphs*. In *weighted graphs*, numbers (weights) are assigned to each edge to represent, e.g., distance or costs.

In *directed graphs*, edges are “one way”: an edge may go from x to y but not from y to x . Directed weighted graphs are sometimes called *networks*.

5.1.2. Representation

We have to map the vertex names to integers between 1 and V .

Assume that

- the function *index*: to convert from vertex names to integers and
- the function *name*: to convert from integers to vertex names.

Adjacency matrix representation

A V -by- V array of Boolean values is maintained, with $a[x, y]$ set to true if there is an edge from vertex x to vertex y and *false* otherwise.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	0	0	1	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	1	0	0	0	0	0	0
F	1	0	0	1	1	1	0	0	0	0	0	0	0
G	1	0	0	0	1	0	1	0	0	0	0	0	0
H	0	0	0	0	0	0	0	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	1	1	0	0
L	0	0	0	0	0	0	0	0	0	1	0	1	1
M	0	0	0	0	0	0	0	0	0	1	0	1	1

Note: Each edge is represented by 2 bits: an edge connecting x and y is represented by *true* values in both $a[x, y]$ and $a[y, x]$.

Also, it is convenient to assume that there's an edge from each vertex to itself.

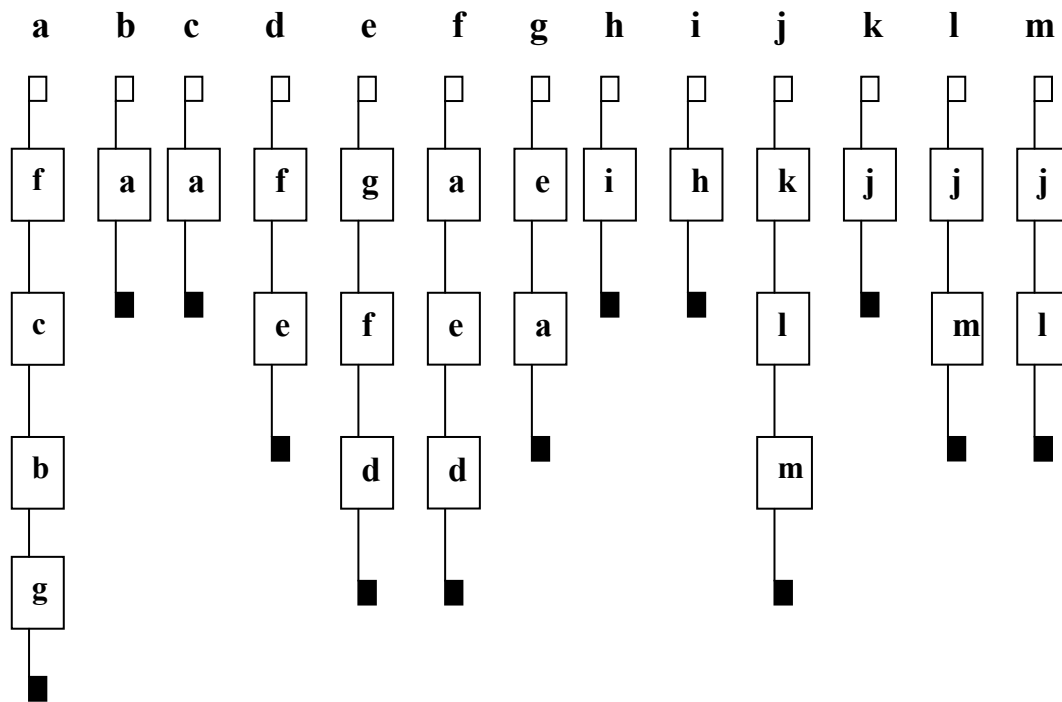
```
program adjmatrix (input, output);
const maxV = 50;
var j, x, y, V, E: integer;
    a: array[1..maxV, 1..maxV] of boolean;
begin
    readln (V, E);
    for x: = 1 to V do /*initialize the matrix */
        for y: = 1 to V do a[x, y]: = false;
    for x: = 1 to V do a[x, x]: = true;
    for j: = 1 to E do
        begin
            readln (v1, v2);
            x := index(v1); y := index(v2);
            a[x, y] := true; a[y, x] := true
        end;
    end.
```

This representation is satisfactory only if the graphs to be processed are dense.

Adjacency list representation.

In this representation, all the vertices connected to each vertex are listed on an *adjacency-list* from that vertex.

```
program adjlist (input, output);
const maxV = 100;
type link = ↑node
    node = record v: integer; next: link end;
var j, x, y, V, E: integer;
    t, x: link;
    adj: array[1..maxV] of link;
begin
    readln(V, E);
    new(z); z↑.next: = z;
    for j: = 1 to V do adj[j]: = z;
    for j:= 1 to E do
        begin
            readln(v1, v2);
            x = index(v1); y = index(v2);
            new(t); t↑.v: = x; t↑.next: = adj[y];
            adj[y]: = t; /* insert x to the first element of y's adjacency list */
            new(t); t↑.v = y; t↑.next: = adj[x];
            adj[x]: = t; /* insert y to the first element of x's adjacency list */
        end;
    end.
```



Note: The order in which edge appear on the adjacency list affects the order in which the edges are processed by algorithms.

5.1.3. Depth-First Search

Search or traverse a graph: to visit every edge in the graph systematically.

There are two main ways to traverse the graph: *depth-first-search* and *breadth-first-search*.

```

procedure dfs;
procedure visit(n:vertex);
begin
  add  $n$  to the ready stack;
  while the ready stack is not empty do
    begin
      get a vertex from the stack, process it,
      and add any neighbor vertex that has not
      been processed to the stack.
      if a vertex has already appeared in the stack,
      there is no need to push it to the stack, but it
      is moved to the top of the stack.
    end;
  end;
begin
  Initialize status;
  for each vertex, say  $n$ , in the graph do
    if the status of  $n$  is “not yet visited” then visit( $n$ )
end;

```

A vertex may be in one of the three status: “not yet visisted”, “in the stack” (ready), and *visited*.

```

procedure list-dfs;
var id, k: integer;
    val: array[1..maxV] of integer;
procedure visit (k: integer);
var t: link;
begin
    id := id + 1; val[k] := id; /* change the status of k to “visited” */
    t := adj[k];           /* find the neighbors of the vertex k */
    while t <> z do
        begin
            if val[t ↑.v] = 0 then visit(t ↑.v);
            t := t ↑.next
        end
    end;
begin
    id := 0;
    for k := 1 to V do val[k] := 0; /* initialize the status of all vetices */
    for k := 1 to V do
        if val[k] = 0 then visit(k)
    end;

```

Note: The array *val*[1..V] keeps the status of the vertices.

$\text{val}[k] = 0$ if the vertex k is “not yet visited”,

$\text{val}[k] < 0$ if the vertex k is visited.

$\text{val}[k] = j$ means the j th vertex to be visited in the traversal is the vertex k .

Figure 5.1.2 traces through the operation of depth-first-search on the example graph.

Property 5.1.1 *Depth-first-search of a graph represented with adjacency lists requires time proportional to $V + E$.*

Proof: We set each of the val value (hence $O(V)$), and we examine each edge twice (hence $O(E)$).

The same method can be applied to graphs represented with adjacency matrices by using the following *visit* procedure:

```

procedure visit(k: integer);
var t: integer;
begin
    id := id + 1; val[k] := id;
    for t := 1 to V do
        if a[k, t] then
            if val[t] = 0 then visit(t)
    end;

```

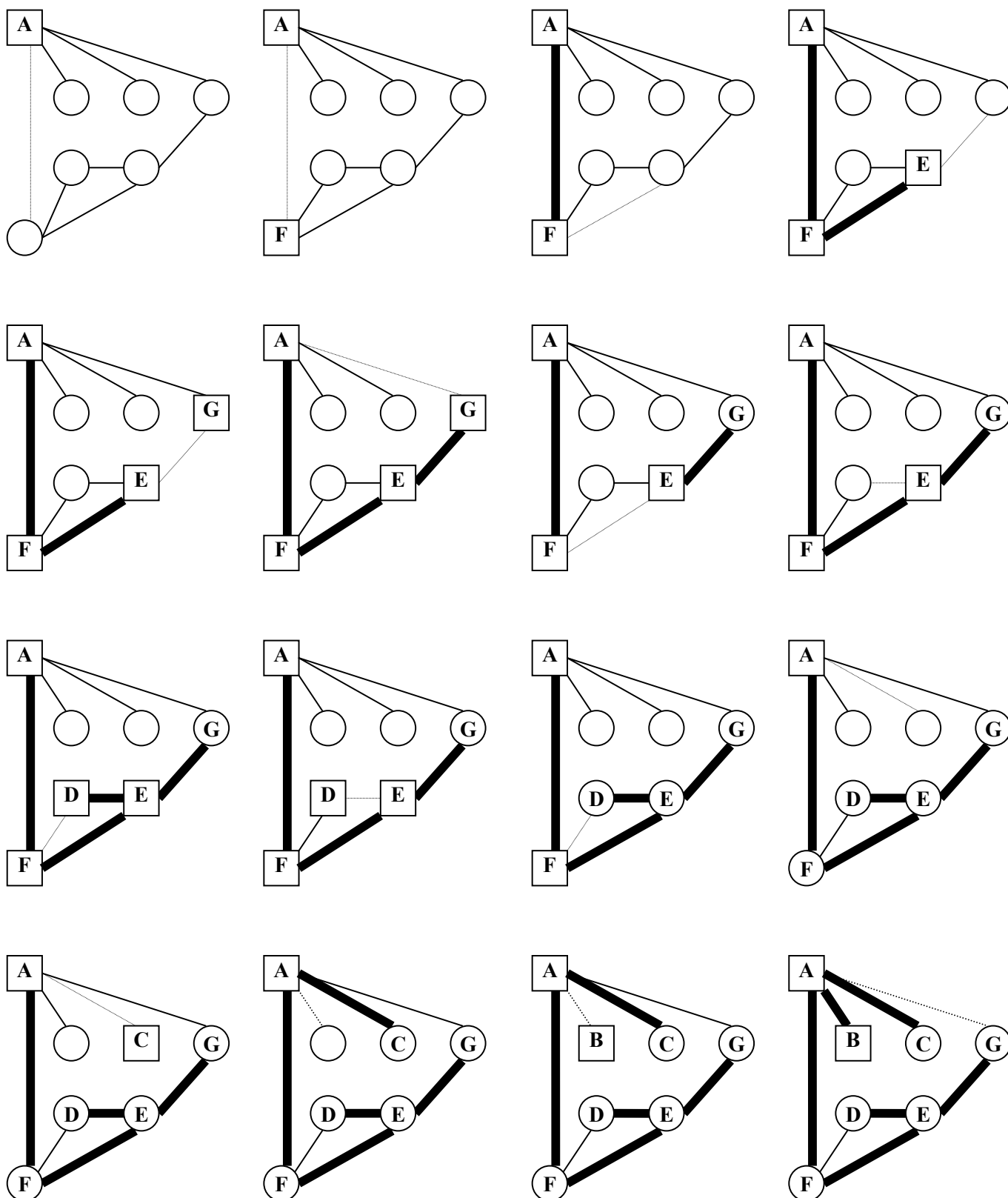



Figure 5.1.2 Depth-first search of large graph component (recursive).

Property 5.1.2 *Depth-first-search of a graph represented with adjacency matrix requires time proportional to V^2 .*

Proof: Each bit in the adjacency matrix is checked.

Non-recursive Depth-First-Search

The recursion in depth-first search can be removed by using a stack.

```
procedure list-dfs;
var id, k: integer;
val: array[1..max V] of integer;
  procedure visit(k: integer);
  var t: link;
  begin
    push(k);
    repeat
      k := pop;
      id := id + 1; val[k] := id; /* change the status of k to “visited” */
      t := adj[k]; /* find the neighbors of the vertex k */
      while t <> z do
        begin
          if val[t↑.v] = 0 then
            begin
              push(t↑.v);
              val[t↑.v] := -1 /* change the status of t↑.v to “ready” */
            end
          else if val[t↑.v] = -1 then
            shift t↑.v to the top of the stack ;
            t := t↑.next ;
          end;
        until stackempty
      end;
    begin
      id := 0; stackinit;
      for k := 1 to V do val[k] := 0; /* initialize the
                                status of all vertices */
      for k := 1 to V do
        if val[k] = 0 then visit(k)
    end;
```

Note: val[k] = 0 if the vertex k is “not yet visited”,
val[k] = -1 if the vertex k is in the ready stack, and
val[k] is positive if the vertex k is visited.

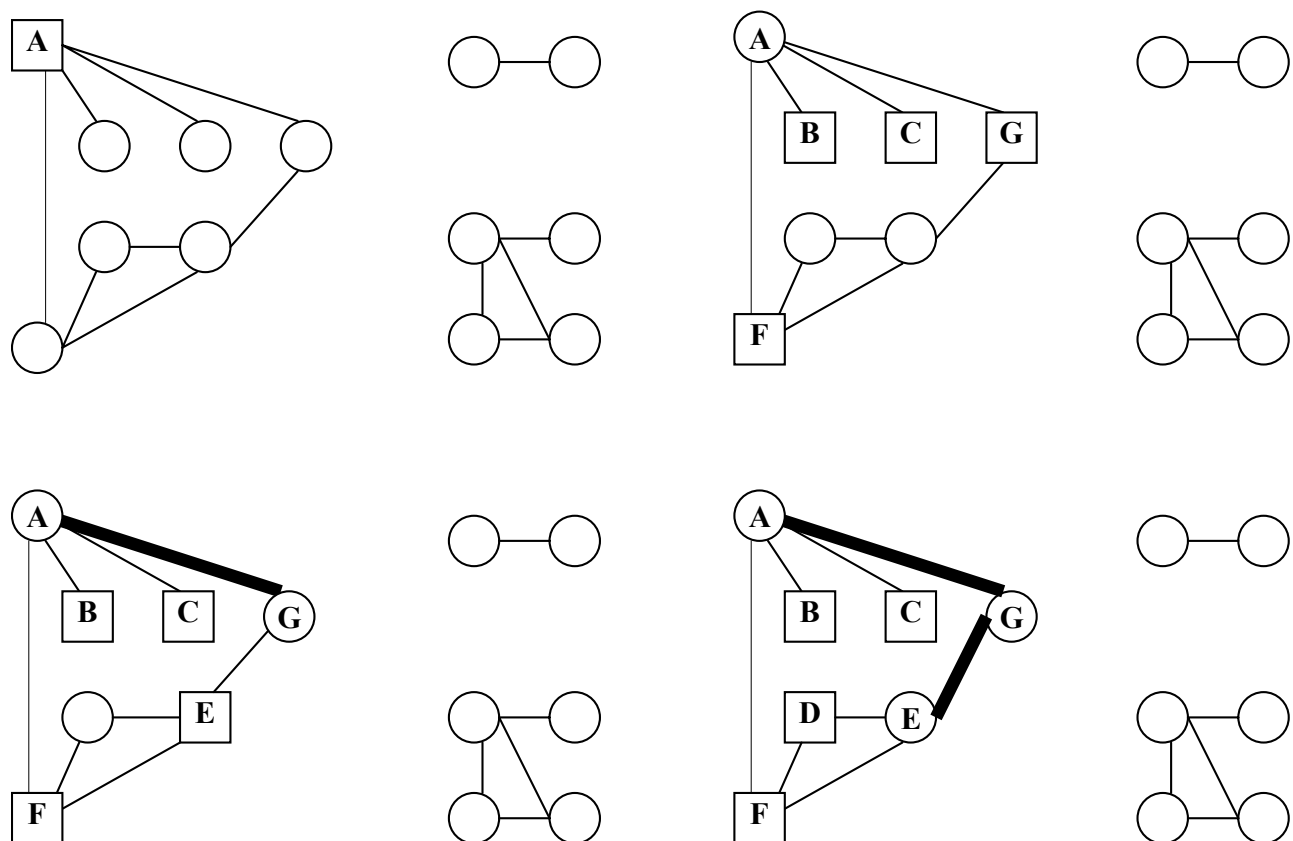


Figure 5.1.3a Start of stack-based search.

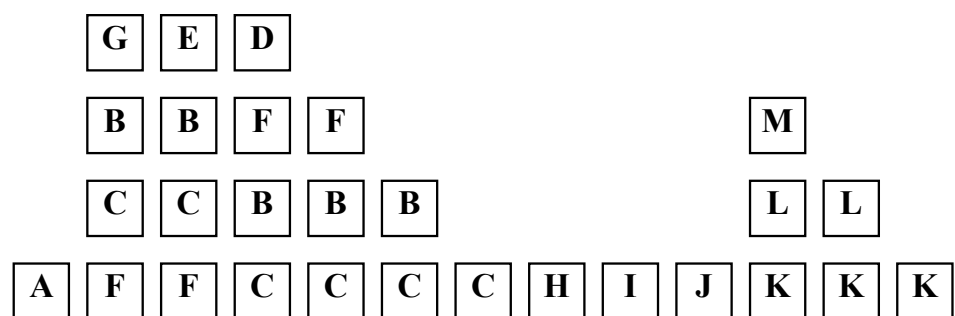


Figure 5.1.3b Contents of stack during stack-based search.

5.1.4. Breadth-first Search

In graph traversal, we can use a queue rather than a stack as the data structure to hold vertices. This leads to *breadth-first-search*.

```
procedure bfs;
  procedure visit(n: vertex);
  begin
    add  $n$  to the ready queue;
    while the ready queue is not empty do
      get a vertex from the queue, process it, and
      add any neighbor vertex that has not
      been processed to the queue and change their
      status to ready.
  end;
  begin
    Initialize status;
    for each vertex, say  $n$ , in the graph
      if the status of  $n$  is “not yet visited”
      then visit( $n$ )
  end;

procedure list-bfs;
var id, k: integer;  val: array[1..max V] of integer;
  procedure visit(k: integer);
  var t: link;
  begin
    put(k); /* put a vertex to the queue */
    repeat
      k := get; /* get a vertex from the queue */
      id := id + 1; val[k] := id; /* change the status of k to “visited” */
      t := adj[k]; /* find the neighbors of the vertex k */
      while t  $\neq$  z do
        begin
          if val[t  $\uparrow$ .v] = 0 then
            begin
              put(t  $\uparrow$ .v);
              val [t  $\uparrow$ .v] := -1 /* change the status of the vertex t  $\uparrow$ .v to “ready” */
            end;
            t := t  $\uparrow$ .next
          end
        until queueempty
      end;
  begin
    id := 0; queue-initialze;
    for k := 1 to V do val[k] := 0; /* initialize the status of all vertices */
    for k := 1 to V do
      if val[k] = 0 then visit(k)
  end;
```

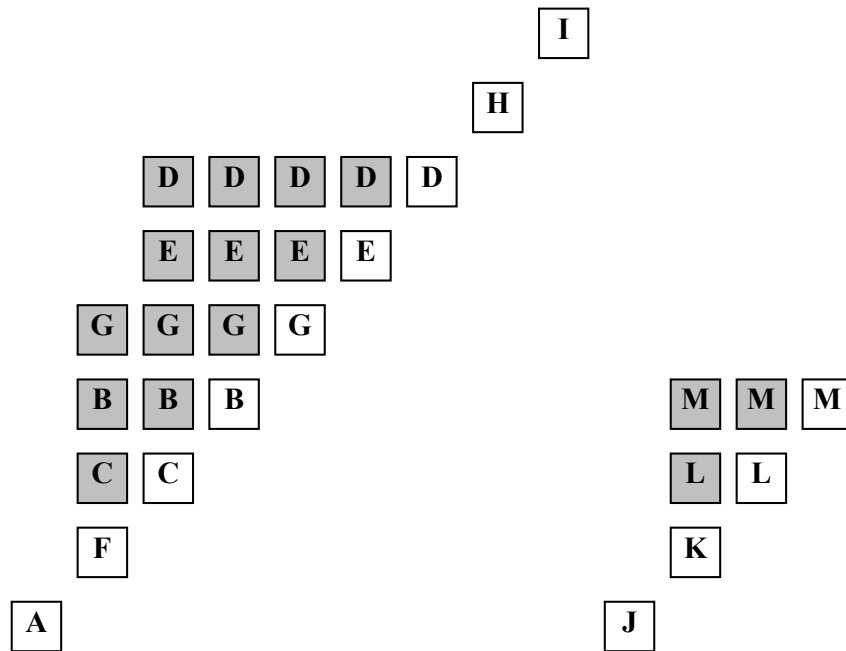


Figure 5.1.4 Contents of queue during breadth-first search.

The time complexity of DFS and BFS are the same.

5.2. WEIGHTED GRAPHS

We often want to model practical problems using graphs in which weights or costs are associated with each edge.

We'll examine algorithms for two problems:

- find the lowest-cost way to connect all of the points.
- find the lowest-cost path between two given points.

The first is called the *minimum spanning tree* problem; the second is called the *shortest-path* problem.

5.2.1. Minimum Spanning Tree

A *minimum spanning tree* of a weighted graph is a collection of edges connecting all the vertices such that the sum of the weights of the edges is minimized.

The minimum spanning tree need not be unique.

Figure 5.2.1 shows an example of a connected graph and minimum spanning tree.

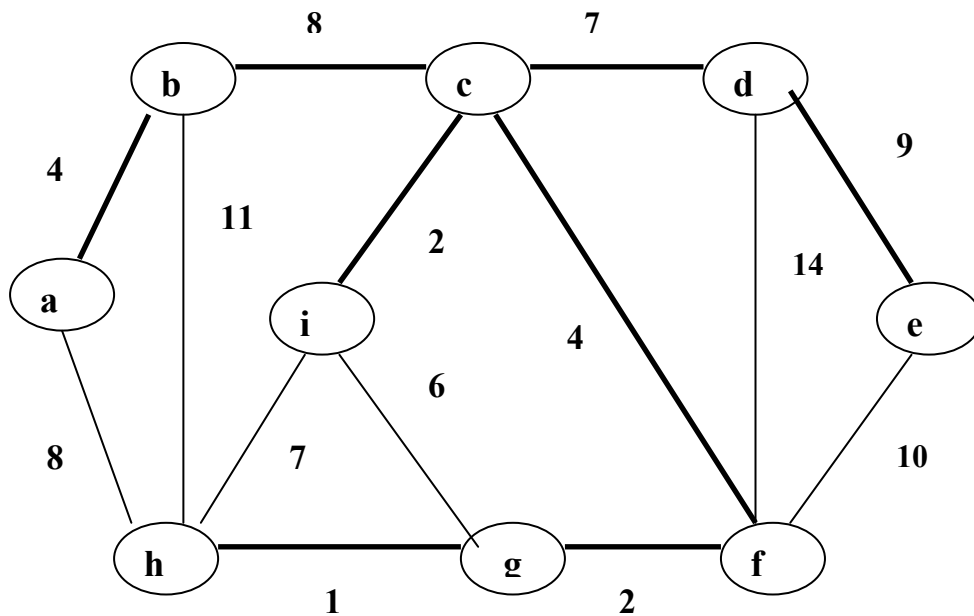


Figure 5.2.1 A minimum spanning tree

We shall examine one algorithm for solving the minimum spanning tree: Prim's algorithm.

The algorithm also illustrates a heuristic for optimization called "*greedy*" strategy:

At each step of an algorithm, one of several possible choices must be made. The greedy strategy advocates making the choice that is the best at the moment.

Such a strategy is not guaranteed to find globally optimal solutions to problems.

For the minimum spanning tree, however, we can prove that certain greedy strategies do yield a spanning tree with minimum weight.

Growing a Minimum Spanning Tree

Assume that we have a connected, undirected graph $G = (V, E)$ with a weight function $w: E \rightarrow \mathbb{R}$ and wish to find a minimum spanning tree for G .

There is a "generic" algorithm which grows the minimum spanning tree one edge at a time.

The algorithm manages a set A that is always a *subset* of some minimum spanning tree.

At each step, an edge (u, v) is determined that can be added to A without violating the invariant that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree.

We call such an edge a *safe edge* for the set A , since it can be safely added to A without destroying the invariant.

```

procedure GENERIC_MST(G, w);
/* G is a weighted graph with the weight function w */
begin
  A :=  $\emptyset$ ;
  while A does not form a spanning tree do
    begin
      find an edge (u,v) that is safe for A;
      add (u,v) to A.
    end
  /* the set A at this point is the result */
end;

```

5.2.2. Prim's Algorithm

In Prim's algorithm, the set A from a single tree. The *safe edge* added to A is always a least-weight-edge connecting the tree to a vertex not in the tree.

The spanning tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V . At each step, a *light edge* connecting a vertex in A to a vertex in $V-A$ is added to the tree.

During execution of the algorithm, all vertices that are not in the tree reside in a priority queue Q based on a key field.

For each vertex v , $\text{key}[v]$ is the minimum weight of any edge connecting v to a vertex in the tree.

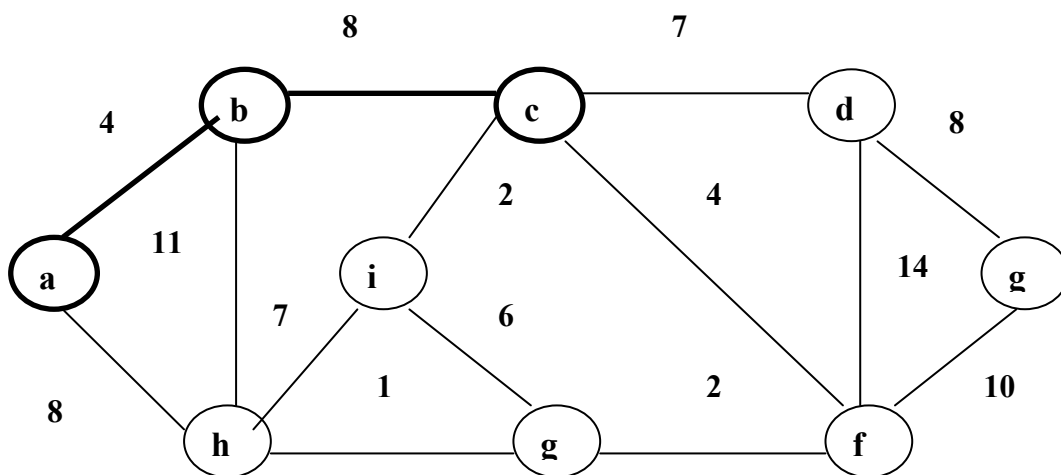
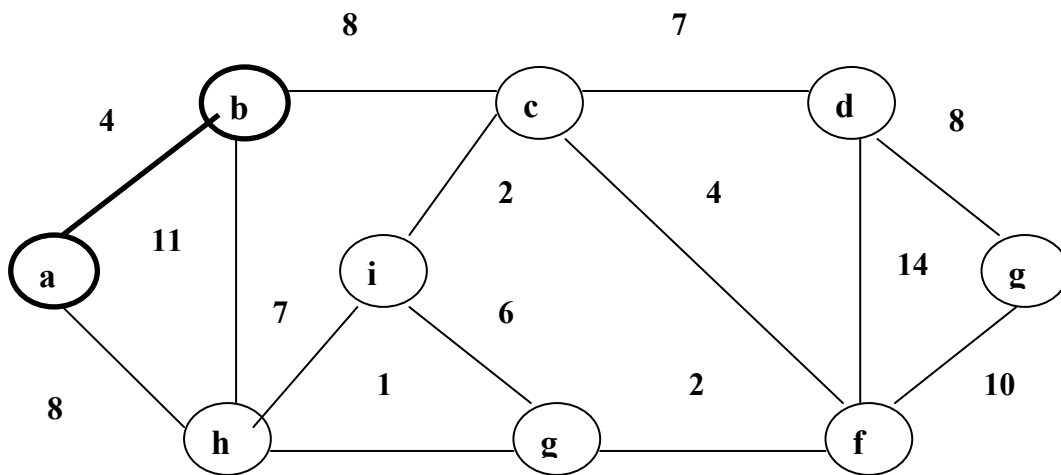
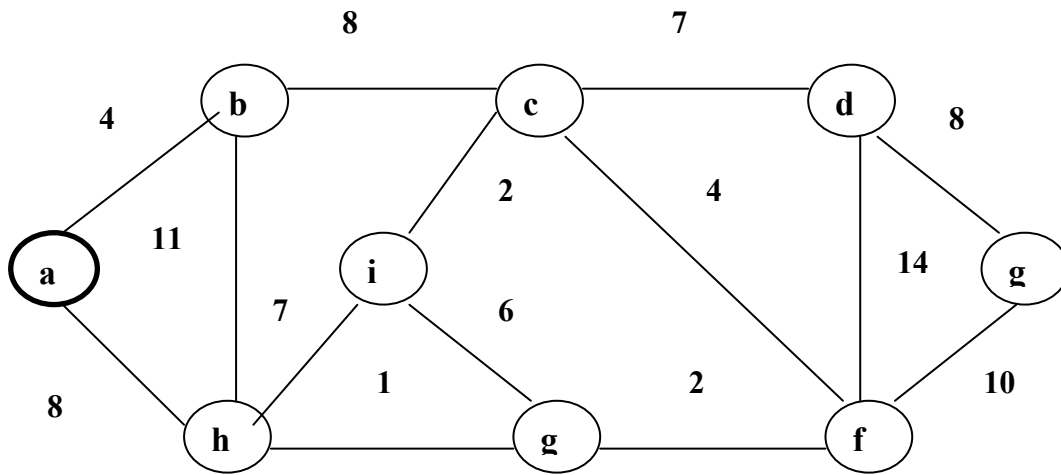
By convention, $\text{key}[v] = \infty$ if there is no such edge.

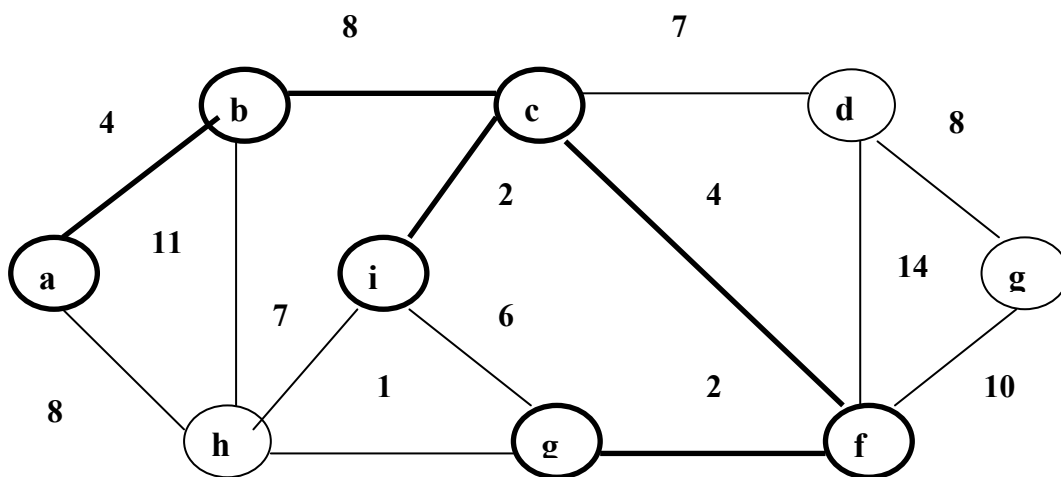
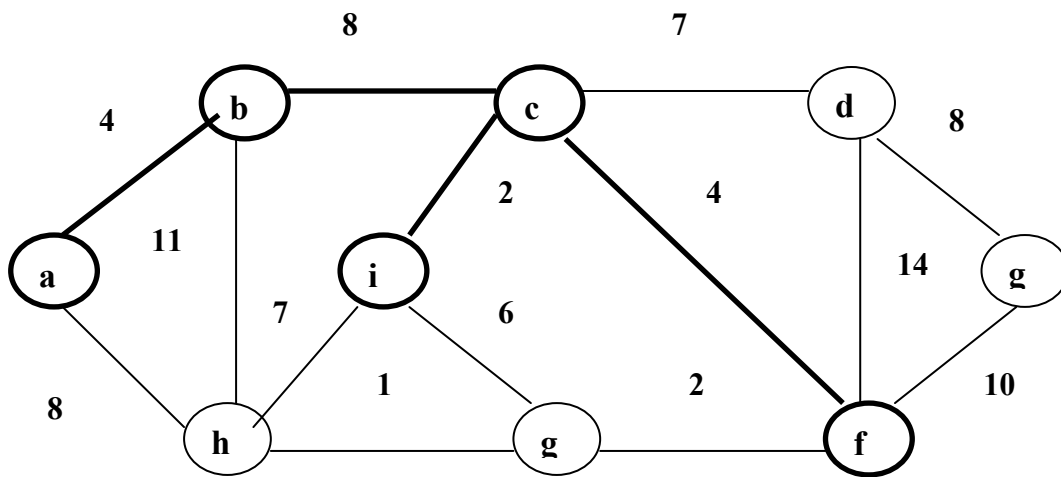
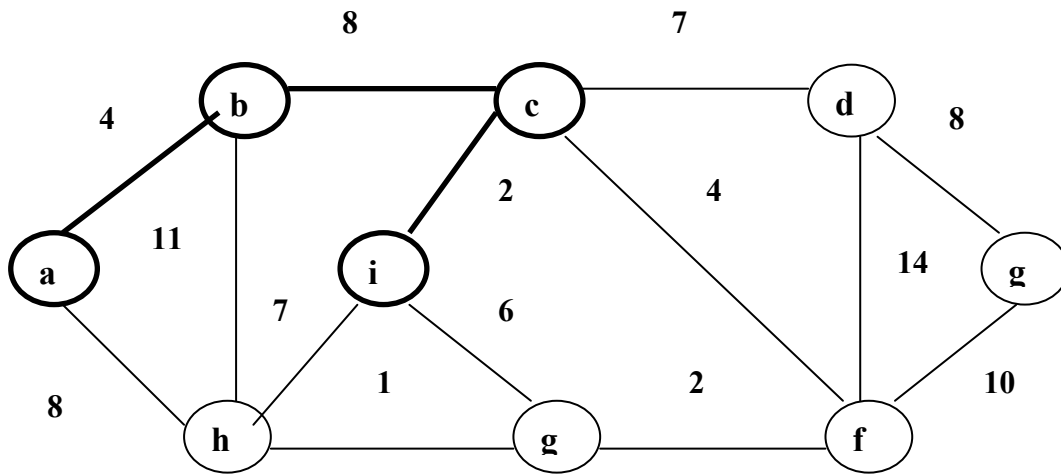
The field $p[v]$ names the "parent" of v in the tree.

```

procedure MST-PRIM (G, w, r);
/* G is weighted graph with the weight function w, and r is an arbitrary root vertex */
begin
  Q := V[G];
  for each  $u \in Q$  do  $\text{key}[u] := \infty$ ;
   $\text{key}[r] := 0$ 
   $p[r] := \text{NIL}$ 
  while Q is not empty do
    begin
       $u := \text{EXTRACT-MIN}(Q)$ ;
      for each  $v \in Q$  and  $w(u, v) < \text{key}[v]$  then
        /* update the key field of vertice v */
        begin
           $p[v] := u$ ;  $\text{key}[v] := w(u, v)$ 
        end
      end
    end
  end;

```





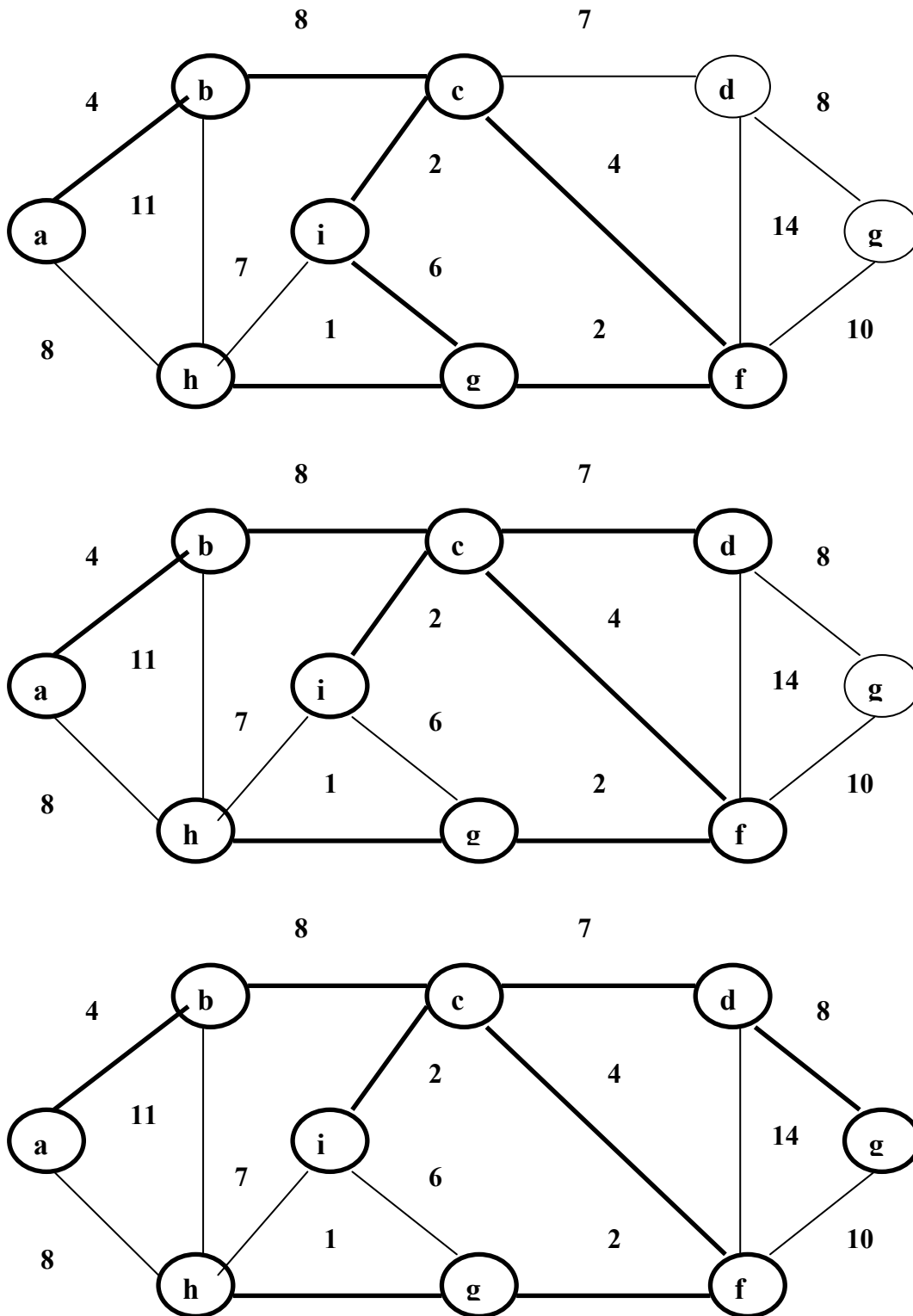


Figure 5.2.2. The execution of Prim's Algorithm

The performance of Prim's algorithm depends on how we implement the priority queue.

Property: If Q is implemented as a binary heap, the running time for Prim's algorithm is $O(E \lg V)$.

If Q is implemented as a binary heap, we can build a heap to perform the initialization in $O(V)$ time.

The while loop is executed V times, and since each EXTRACT-MIN operation take $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$.

The for loop inside the while loop is executed $O(E)$ time altogether, since the sum of the length of all adjacency lists is $2E$. Updating the key of vertex v in the heap requires $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + 2E \lg V) = O(E \lg V)$.

5.3. DIRECTED GRAPHS

Directed graphs are graphs in which the edges connecting the nodes have direction.

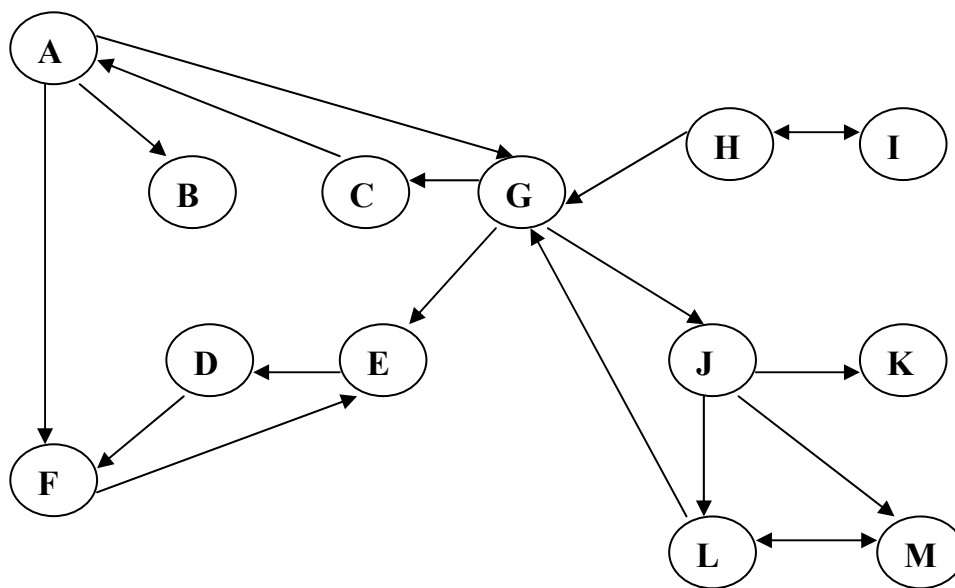


Figure 5.3.1. A directed graph.

Often the edge direction reflects some type of precedence relationship in the application to be modeled.

For example, a directed graph might be used to model a manufacturing line.

In this section, we'll look at some algorithms:

- computing transitive closure
- topological sorting

5.3.1. Transitive Closure

In directed graph, we're interested in the set of vertices that can be reached from a given vertex by traversing edges in the graph in the indicated direction.

One operation we might want to perform is "to add an edge directly from x to y if there is some way to get from x to y ".

The graph that results from adding all edges of this nature to a directed graph is called the *transitive closure* of the graph.

Since the transitive closure graph is likely to be dense, so we use an adjacency-matrix representation.

There is a very simple algorithm for computing the transitive closure of a graph represented with an adjacency matrix:

```

for y : = 1 to V do
  for x : = 1 to V do
    if a[x, y] then
      for j: = 1 to V do
        if a[y, j] then a[x, j]: = true;

```

S. Warshall invented this method in 1962, using the simple observation: “If there is a way to get from node x to node y and a way to get from node y to node j, then there’s a way to get from node x to node j.”

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	0	0	0	1	1	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	0	1	0	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0	0	0	0
F	0	0	0	0	1	1	0	0	0	0	0	0	0
G	0	0	1	0	1	0	1	0	0	1	0	0	0
H	0	0	0	0	0	0	1	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	0	0	0	0	0	0	1	0	0	0	0	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1

(a) Initial stage of Warshall’s algorithm

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	1	1	1	1	0	0	1	1	1	1
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	1	1	1	1	0	0	1	1	1	1
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	0	0	0	0	0	0	0
F	0	0	0	1	1	1	0	0	0	0	0	0	0
G	1	1	1	1	1	1	1	0	0	1	1	1	1
H	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	1	1	1	1	1	1	1	1	1
J	1	1	1	1	1	1	1	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	1	1	1	1	1	1	1	0	0	1	1	1	1
M	1	1	1	1	1	1	1	0	0	1	1	1	1

(b) Last stage of Warshall’s algorithm.

Property 5.3.1 Warshall’s algorithm finds the transitive closure in $O(V^3)$ steps.

5.3.2. All Shortest Paths

For weighted graph (directed or not) one might want to build a matrix allowing one to find the shortest path from x to y for all pairs of vertices. This is the all-pairs shortest path problem.

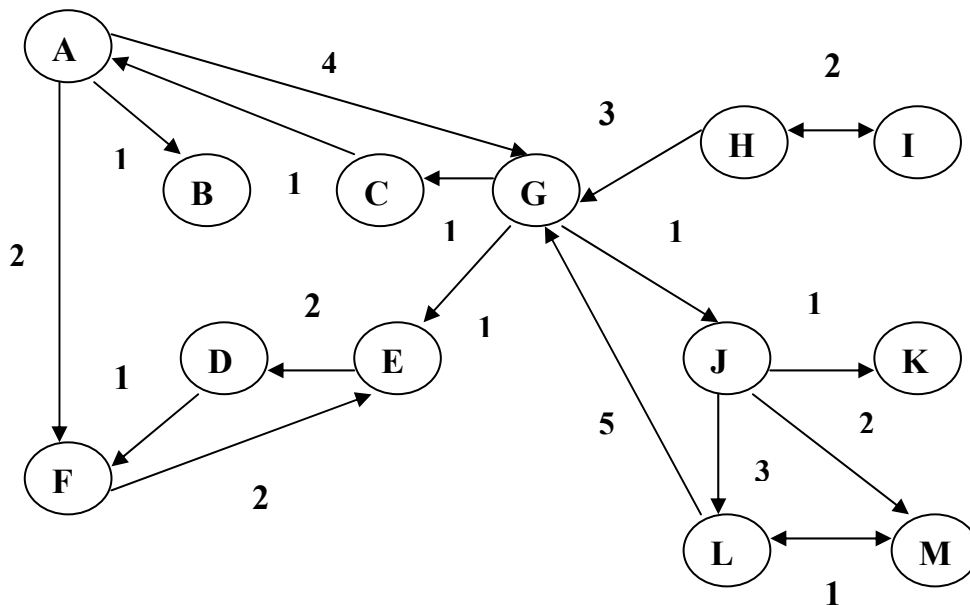


Figure 5.3.2 A weighted directed graph.

It is possible to use a method just like Warshall's method, which is invented by R. W. Floyd:

```

for  $y := 1$  to  $V$  do
  for  $x := 1$  to  $V$  do
    if  $a[x, y] > 0$  then
      for  $j := 1$  to  $V$  do
        if  $a[y, j] > 0$  then
          if  $(a[x, j] = 0)$  or  $(a[x, y] + a[y, j] < a[x, j])$ 
            then
               $a[x, j] = a[x, y] + a[y, j];$ 

```

The structure of the algorithm is the same as in Warshall's method. Instead of using "or" to keep track of the paths, we do a little computation for each edge.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	0	0	0	2	4	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	2	0	0	0	0	0	0	0	0	0
F	0	0	0	0	2	0	0	0	0	0	0	0	0
G	0	0	0	0	2	0	0	0	0	1	0	0	0
H	0	0	0	0	0	0	3	0	1	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	3	2
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	5	5	0	0	0	0	0	1
M	0	0	0	0	0	0	0	0	0	0	0	1	0

(a) Initial stage of of Floyd's algorithm

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	6	1	5	6	4	2	4	0	0	5	6	8	7
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	6	7	5	3	5	0	0	6	7	9	8
D	0	0	0	5	3	1	0	0	0	0	0	0	0
E	0	0	0	2	5	3	0	0	0	0	0	0	0
F	0	0	0	4	2	5	0	0	0	0	0	0	0
G	2	3	1	3	1	4	6	0	0	1	2	4	3
H	5	6	4	6	4	7	3	2	1	4	5	7	6
I	6	7	5	7	5	8	4	1	2	5	6	8	7
J	10	11	9	11	9	12	8	0	0	9	1	3	2
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	7	8	6	8	6	9	5	0	0	6	7	2	1
M	8	9	7	9	7	10	6	0	0	7	8	1	2

(b) Last stage of Eloyd's algorithm

Property 5.3.2 *Floyd's algorithm solves the all-pairs shortest path problem in $O(V^3)$ steps.*

5.3.3. Topological Sorting

Directed Acyclic Graphs

Directed graphs with no directed cycles are called *directed acyclic graphs* or *dags*.

Partially Ordered Set and Topological Sorting

Let G be a directed acyclic graph. Consider the relation $<$ defined as follows:

$u < v$ if there is a path from u to v in G .

The relation has the three properties:

- (1) For each vertex in $V[G]$, not $(u < u)$. (*Irreflexivity*)
- (2) If $u < v$, then not $(v < u)$. (*Asymmetry*)
- (3) If $u < v$ and $v < w$, then $u < w$. (*Transitivity*)

Such a relation $<$ on a set S is called a *partial ordering* of S , and a set with such a partial ordering is called a *partially order set* or *poset*.

Thus, any directed acyclic graph may be called as a partially ordered set.

On the other hand, suppose S is a *poset* with the partial ordering denoted by $<$. Then S may be viewed as a *directed graph* whose vertices are the elements of S and whose edges are defined as follows:

(u, v) is an edge if $u < v$.

Furthermore, one can show that a poset S , regarded as a directed graph, has no cycles.

Topological Sorting

Let G be a directed acyclic graph. A *topological sort* T of G is a linear ordering of the vertices of G which preserves the original partial ordering of $V[G]$.

That is: if $u < v$ in V (i.e. if there is a path from u to v in G), then u comes before v in the linear ordering T .

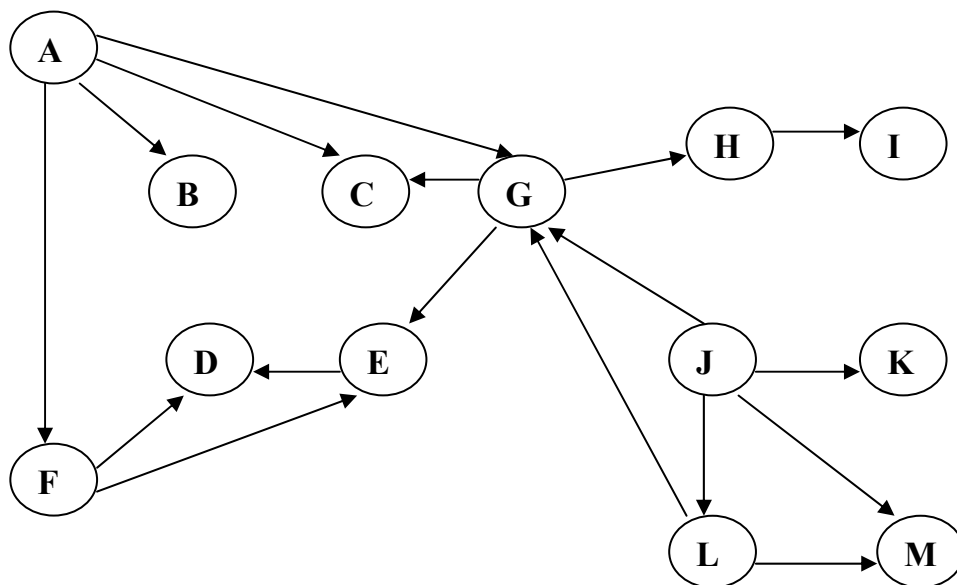


Figure 5.3.3 A directed acyclic graph

For example, the nodes in the graph in Figure 5.3.3 could be topologically sorted in the following order:

J K L M A G H I F E D B C

There are some approaches to topological sorting.

Method 1

The basic idea is to find a node with *no successor*, remove it from the graph, and add it to the list.

Repeating this process until the graph is empty will produce a list that is the reverse of some topological sorting.

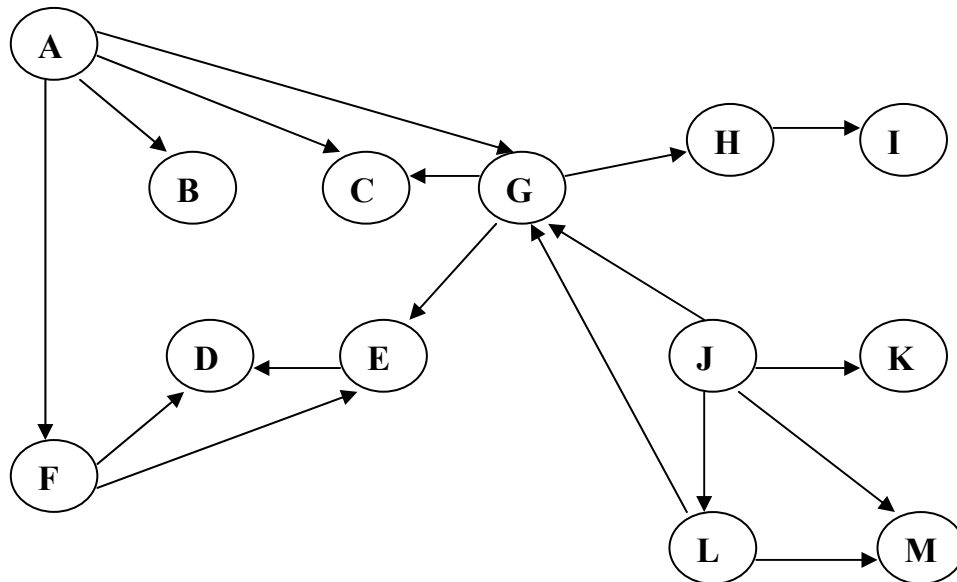


Figure 5.3.3

C	B	D	E	F	I	H	G	A	M	L	K	J
J	K	L	M	A	G	H	I	F	E	D	B	C

Algorithm:

while the graph has a node with no successors **do**
 remove one such node from the graph and
 add it to the end of a list.

if the loop terminates with the graph empty
then the list shows the reverse of a topological order.
else the graph contains a cycle.

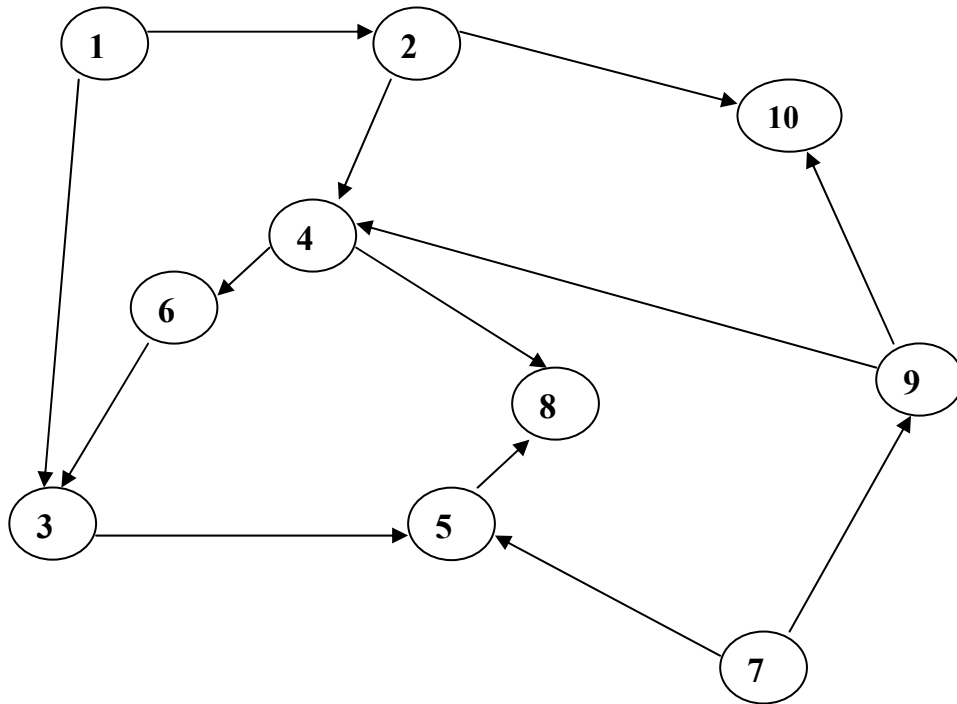
Method 2

The second approach is to perform a *depth-first search* and to add a node to the list each time it is necessary to take a node off the stack in order to continue. The stack is popped when a node has no successors.

Algorithm:

Start with nodes with no predecessor, put them in the stack.

while the stack is not empty **do**
 if the node at top of the stack has some successors **then**
 push all its successors onto the stack
 else pop it from the stack and add it to the list.



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

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

Figure 5.3.4

Chapter 6. ALGORITHM DESIGN TECHNIQUES

6.1. DYNAMIC PROGRAMMING

Dynamic programming solves problems by combining the solutions to sub-problems. It is applicable when the sub-problems are *not* independent, i.e., when sub-problems *share* subsubproblems.

Dynamic programming algorithm solves every subsubproblem just once and save its answer in a *table*, thereby avoiding recomputing the answer every time the subsubproblem is encountered.

Dynamic programming is applied to *optimization problems*. (In such problems there can be many possible solutions. We want to find a solution with the *optimal solution*.)

The development of a dynamic-programming algorithm can be divided into a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a *bottom-up* fashion.
4. Construct an optimal solution from computed information.

6.1.1. Matrix-Chain Multiplication

This is the 1st example of dynamic-programming algorithm.

Given a sequence $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product.

$$A_1 A_2 \dots A_n \quad (6.1)$$

A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

For example, the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in five ways:

$(A_1(A_2(A_3A_4)))$
 $(A_1((A_2A_3)A_4))$
 $((A_1A_2)(A_3A_4))$
 $(A_1(A_2A_3))A_4$
 $((A_1A_2)A_3)A_4$

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

Example:

A_1	10×100
A_2	100×5
A_3	5×50

$((A_1A_2)A_3)$ performs $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$ scalar multiplications

$(A_1(A_2A_3))$ performs $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$

The *matrix-chain multiplication problem* can be stated as follows:

"Given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices, where for $i = 1, 2, \dots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product in a way that *minimizes* the number of scalar multiplications".

The structure of an Optimal Parenthesization.

The first step is to characterize the structure of an optimal solution.

Let use the notation $A_{i..j}$ for the matrix that results from evaluating the product $A_i A_{i+1} \dots A_j$.

An *optimal* parenthesization of the product

$A_1.A_2 \dots A_n$ splits the product between A_k and A_{k+1} for some integer k , $1 \leq k < n$. That is, first we compute the matrices $A_{1..k}$ and $A_{k+1..n}$ and then multiply them together to produce the final product $A_{1..n}$.

The cost of this optimal parenthesization = the cost of computing the matrix $A_{1..k}$ + the cost of computing $A_{k+1..n}$, + the cost of multiplying them together.

The key observation is that

"the parenthesization of the subchain $A_1A_2 \dots A_k$ within the optimal parenthesization of $A_1A_2 \dots A_n$ must be also an optimal parenthesization".

Thus, an optimal solution to the matrix-chain multiplication problem contains within optimal solutions to subproblems.

A Recursive Solution

The 2nd step of the dynamic programming paradigm is *to define the value of an optimal solution recursively* in terms of *the optimal solutions to subproblems*.

Here, we select as our subproblems the problems of determining the minimum cost of a parenthesization of $A_i.A_{i+1} \dots A_j$ for $1 \leq i \leq j \leq n$.

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; the cost of the cheapest way to compute $A_{1..n}$ would be $m[1, n]$.

Assume that the optimal parenthesization *splits* the product $A_i A_{i+1} \dots A_j$ between A_k and A_{k+1} , where $i \leq k < j$. Then $m[i, j]$ is equal to the *minimum cost* for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together.

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j.$$

So, recursive definition for the minimum cost of parenthesization of $A_i A_{i+1} \dots A_j$ is as follows:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min \{m[i, k] + m[k+1, j] + p_{i-1}p_kp_j\} & \text{if } i < j. \end{cases} \quad (6.2)$$

To help us keep track of how to construct an optimal solution, let define

$s[i, j]$: the value of k at which we split the product $A_i A_{i+1} \dots A_j$ to get an optimal parenthesization.

Computing the Optimal Costs

Instead of computing the solution to recurrence equation (6.2) by recursive algorithm, we achieve the 3rd step of the dynamic programming: to compute the optimal cost by using a *bottom-up* approach.

The following assumes that the matrix A_i has the dimension $p_{i-1} \times p_i$ for $i = 1, 2, \dots, n$.

The input is a sequence $\langle p_0, p_1, \dots, p_m \rangle$.

The procedure uses

- an auxiliary table $m[1..n, 1..n]$ for storing the $m[i, j]$ costs and
- an auxiliary table $s[1..n, 1..n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$.

The procedure returns two arrays m and s .

procedure MATRIX-CHAIN-ORDER(p, m, s);

begin

$n := \text{length}[p] - 1$;

for $i := 1$ **to** n **do**

$m[i, i] := 0$;

for $l := 2$ **to** n **do** /* l : length of the chain */

for $i := 1$ **to** $n - l + 1$ **do**

begin

$j := i + l - 1$;

$m[i, j] := \infty$; /* initialization */

for $k := i$ **to** $j-1$ **do**

begin

$q := m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$;

```

    if  $q < m[i, j]$  then
    begin
         $m[i, j] := q$ ;
         $s[i, j] := k$ 
    end
end
end
end
end

```

Since we define $m[i, j]$ only for $i < j$, only the portion of the table m strictly above the main diagonal is used.

Figure 6.1.1 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

Matrix	Dimention
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Array m

		i					
		1	2	3	4	5	6
	6	15125	10500	51375	3500	5000	0
	5	11875	7125	2500	1000	0	
j	4	9357	4375	750	0		
	3	7875	2625	0			
	2	15750	0				
	1	0					

Array s

		i				
		1	2	3	4	5
	6	3	3	3	5	5
	5	3	3	3	4	
j	4	3	3	3		
	3	1	2			
	2	1				

Figure 6.1.1

$$\begin{aligned}
 m[2,5] = \min \quad & \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35.15.20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35.5.20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35.10.20 = 11375 \end{cases} \\
 & = 7125
 \end{aligned}$$

so $\Rightarrow k = 3$ for $A_{2...5}$

Construction an Optimal Solution

The step 4 of the dynamic programming paradigm *is to construct an optimal solution from computed information.*

We use the table $s[1..n, 1..n]$ to determine the best way to multiply the matrix chain. Each entry $s[i, j]$ records the value of k such that the optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1} .

The following procedure computes the matrix-chain product $A_{i..j}$, given the matrices $A = \langle A_1, A_2, \dots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER and the indices i and j . It returns the result as the parameter AIJ.

The initial call is MATRIX-CHAIN-MULTIPLY($A, s, 1, n, A1N$).

```

procedure MATRIX-CHAIN-MULTIPLY( $A, s, i, j, AIJ$ );
begin
  if  $j > i$  then
    begin
      MATRIX-CHAIN-MULTIPLY( $A, s, i, s[i, j], X$ );
      MATRIX-CHAIN-MULTIPLY( $A, s, s[i, j]+1, j, Y$ );
      MATRIX-MULTIPLY( $X, Y, AIJ$ );
    end
  else
    assign  $A_i$  to  $AIJ$ ;
  end;

```

6.1.2. Elements of Dynamic Programming

Now, we examine the two key elements that an optimization problem must have for dynamic programming to be applicable: (1) optimal substructure and (2) *overlapping subproblems*.

Optimal Substructure

We say that a problem exhibits *optimal sub-structure* if an optimal solution to the problem contains within it optimal solutions to sub-problems.

Whenever a problem exhibits optimal sub-structure, dynamic programming *might apply*.

Overlapping Subproblems

The space of subproblems must be "small" in the sense that *a recursive algorithm for the problem solves the same subproblems over and over*. When a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has *overlapping subproblems*.

Dynamic programming algorithms take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a *table* where it can be looked up when needed.

Compare this top-down, recursive algorithm with the bottom-up, dynamic programming algorithm. The latter is more efficient because it takes advantage of the overlapping-subproblems property.

Whenever a *recursion-tree* for the recursive solution to a problem contains the same subproblem repeatedly, and the total number of different subproblems is small, dynamic programming can be applied.

Example: The iterative algorithm to compute Fibonacci function.

6.1.3. Longest Common Subsequence

The next problem is the longest common subsequence problem.

A *subsequence* of a given sequence is just the given sequence with some elements (possibly none) left out.

For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Given two sequences X and Y , we say that a sequence Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y .

In the *longest-common subsequence* problem, we are given two sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ and wish to find a maximum-length common subsequence of X and Y .

Example: $X = \langle A, B, C, B, D, A, B \rangle$ and
 $Y = \langle B, D, C, A, B, A \rangle$

The sequence $\langle B, D, A, B \rangle$ is an LCS of X and Y .

The LCS problem has an optimal-substructure property as the following theorem shows.

Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, we define the *i th prefix* of X , for $i = 0, 1, \dots, m$, as

$X_i = \langle x_1, x_2, \dots, x_i \rangle$.

$X_0 =$ empty sequence

Theorem 6.1 (Optimal-substructure of an LCS)

Let $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ be sequences, and let

$Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$, then $x_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1} .

Theorem 6.1 shows that LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal substructure property.

A Recursive Solution to Subproblems

We can see the overlapping subproblems property in the LCS problem.

To find an LCS of X and Y , we may need to find the LCS's of X and Y_{n-1} and of X_{m-1} and Y . But each of these subproblems has the sub-subproblem of finding the LCS of X_{m-1} and Y_{n-1} . Many other sub-problems share sub-subproblems.

The recursive solution to the LCS problem involves establishing a recurrence for the cost of an optimal solution.

Let define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j . If either $i = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula.

$$c[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1, j-1]+1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \quad (6.3)$$

Computing the Length of an LCS

Based on equation (6.3), we can write a exponent-time recursive algorithm to compute the length of an LCS of two sequences. However, we can use dynamic programming to compute the solution *bottom-up*.

The procedure LCS-LENGTH takes two sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ as inputs. It stores the $c[i, j]$ values in a table $c[0..m, 0..n]$. It also maintains the table $b[1..m, 1..n]$ to simplify construction of an optimal solution.

procedure LCS-LENGTH(X, Y)

begin

$m := \text{length}[X];$

$n := \text{length}[Y];$

for $i := 1$ **to** m **do** $c[i, 0] := 0;$

for $j := 1$ **to** n **do** $c[0, j] := 0;$

for $i := 1$ **to** m **do**

for $j := 1$ **to** n **do**

if $x_i = y_j$ **then**

begin

$c[i, j] := c[i-1, j-1] + 1; b[i, j] := \text{"↖"}$


```

    end
    else if  $c[i - 1, j] \geq c[i, j-1]$  then
    begin
         $c[i, j] := c[i - 1, j]$ ;  $b[i, j] := "\uparrow"$ 
    end
    else
    begin
         $c[i, j] := c[i, j-1]$ ;  $b[i, j] := "\leftarrow"$ 
    end
    /*At this point, two tables c and b are established */
end;
```

The running time of the procedure is $O(nm)$.

Constructing an LCS

The b table computed by LCS-LENGTH can be used to quickly construct an LCS of $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$

The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial invocation is PRINT-LCS($b, X, \text{length}[X], \text{length}[Y]$).

```

procedure PRINT-LCS( $b, X, i, j$ )
begin
    if  $i \leq 0$  and  $j \leq 0$  then
        if  $b[i, j] = "\nwarrow"$  then
            begin
                PRINT-LCS( $b, X, i - 1, j - 1$ ); print  $x_i$ 
            end
        else if  $b[i, j] = "\uparrow"$  then
            PRINT-LCS( $b, X, i-1, j$ )
        else PRINT-LCS( $b, X, i, j-1$ )
end;
```

The running-time of the procedure PRINT-LCS is $O(m+n)$

		3	0	1	2	3	4	5	6
		i							
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	←-1	↖ 1
2	B		0	↖ 1	←-1	←-1	↑ 1	↖ 2	←-2
3	C		0	↑ 1	↑ 1	↖ 2	←-2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	←-3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 6.1.2 The c and b table computed by LCS-LENGTH on the sequences $X = (A, B, C, B, D, A, B)$ and $Y = (B, D, C, A, B, A)$. The square in row i and column j contains the value of $c[i,j]$ and the appropriate arrow for the value of $b[i,j]$.

6.1.4 The Knapsack Problem

"A thief robbing a store finds it contains N types of items of varying size and value, but has only a small knapsack of capacity M to use to carry the goods. The *knapsack problem* is to find the combination of items which the thief should choose for his knapsack in order to maximize the total value of all the items he takes."

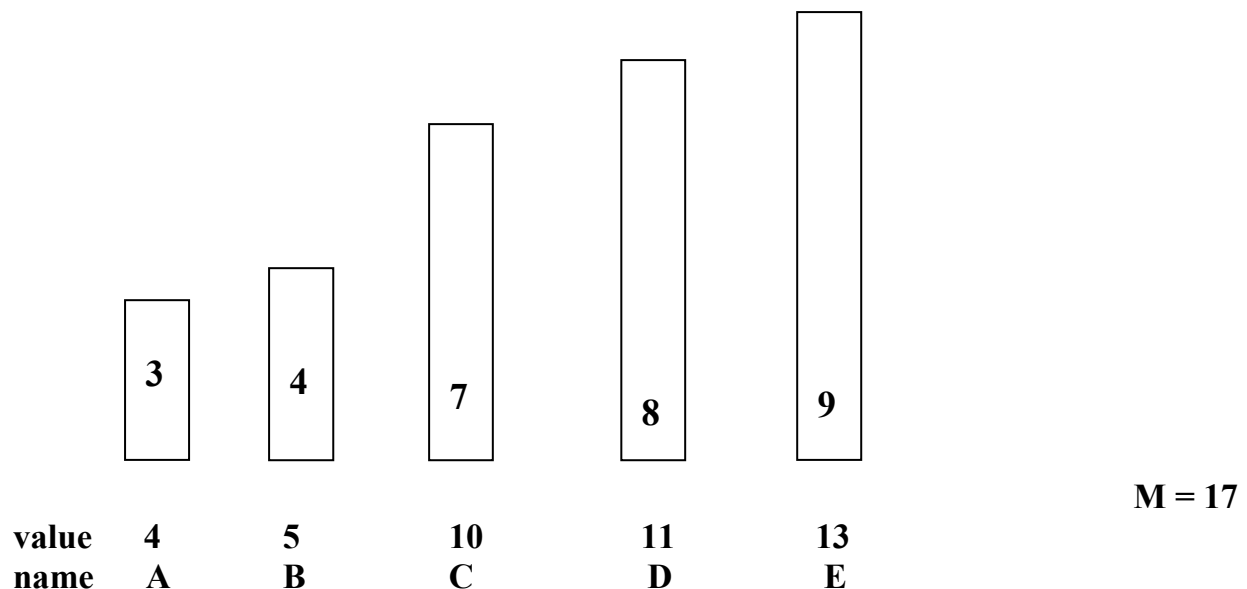


Figure 6.1.4

```

for i:= 0 to M do cost[i] := 0;
for j:= 1 to N do /* each of item type */
begin
  for i:= 1 to M do /* i means capacity */
    if i - size[j] >= 0 then
      if cost[i] < (cost[i - size[j]] + val[j]) then
        begin
          cost[i] := cost[i - size[j]] + val[j];
          best[i] := j
        end;
    end;
end;

```

$cost[i]$ is the highest value that can be achieved with a knapsack of capacity i

$$cost[i] = cost[i - size[j]] + val[j]$$

Suppose an item j is chosen for the knapsack: then the best value that could be achieved for the total would be

$$val[j] + cost[i - size[j]].$$

If this value is greater than the best value that can be achieved without an item j , then we update $cost[i]$; otherwise we leave them alone.

$best[i]$: the last item that was added to achieve that maximum.

The actual contents of the optimal knapsack can be computed with the aid of the *best* array.

By definition, $best[M]$ is included in the optimal knapsack and the remaining contents are the same as for the optimal knapsack of size $M - size[best[M]]$. Therefore, $best[m - size[best[M]]]$ is included, and so on.

K	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1																	
cost[k]	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
best[k]			A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2																	
cost[k]	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
best[k]			A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3																	
cost[k]	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
best[k]			A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4																	
cost[k]	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
best[k]			A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5																	
cost[k]	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
best[k]			A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Figure 6.1.5

Note: The knapsack problem is easily solved if M is not large, but the running time can become unacceptable for large capacities.

The method does not work if M and the sizes or values are real numbers instead of integers.

Property 6.1.1 *The dynamic programming solution to the knapsack problem takes time proportional to NM .*

6.2. GREEDY ALGORITHMS

Algorithms for optimization problems go through a sequence of steps, with a set of choices at each step. A *greedy algorithm* always makes the choice that looks best at the moment.

That is, the algorithm makes *locally optimal* choice in the hope this choice will lead to a *globally optimal* solution.

Some examples of greedy algorithms:

- Prim's algorithm for computing minimum spanning tree and
- Dijkstra's algorithm for single-source shortest paths problem.

6.2.1. An Activity-Selection Problem

Suppose we have a set $S = \{1, 2, \dots, n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time.

Each activity i has a *start time* s_i and a *finish time* f_i , where $s_i \leq f_i$. If selected, activity i takes place during the interval $[s_i, f_i)$. Activities i and j are *compatible* if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., i and j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$).

The *activity-selection problem* is to select a maximum-size set of mutually compatible activities.

In the following procedure which applies greedy algorithm for the *activity-selection problem*, we assume that the input activities are in order by increasing finishing time:

$$f_1 \leq f_2 \leq \dots \leq f_n.$$

procedure GREEDY-ACTIVITY-SELECTOR(S, f) ;

```
/* s is the array keeping the set of activities and f is the array keeping the finishing times */
```

begin

```
n := length[s];
```

$$A := \{1\};$$
$$j := 1;$$
for i: = 2 to n do

if $s_i \geq f_j$ **then** /* i is compatible with all activities in A */

begin

$$A := A \cup \{i\}; j := i$$

end

```
/* at this point, the set A is the result */
```

end

The activity selected by GREEDY-ACTIVITY-SELECTER is always the one with the *earliest finish time* that can be legally scheduled. The activity picked is thus a “greedy” choice in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled.

Greedy algorithms do not always produce optimal solutions. However, GREEDY-ACTIVITY-SELECTOR always finds an optimal solution to an instance of the activity-selection problem.

There are elements that are exhibited by most problems that lend themselves to a greedy strategy: (1) the *greedy-choice property* and (2) *optimal substructure*.

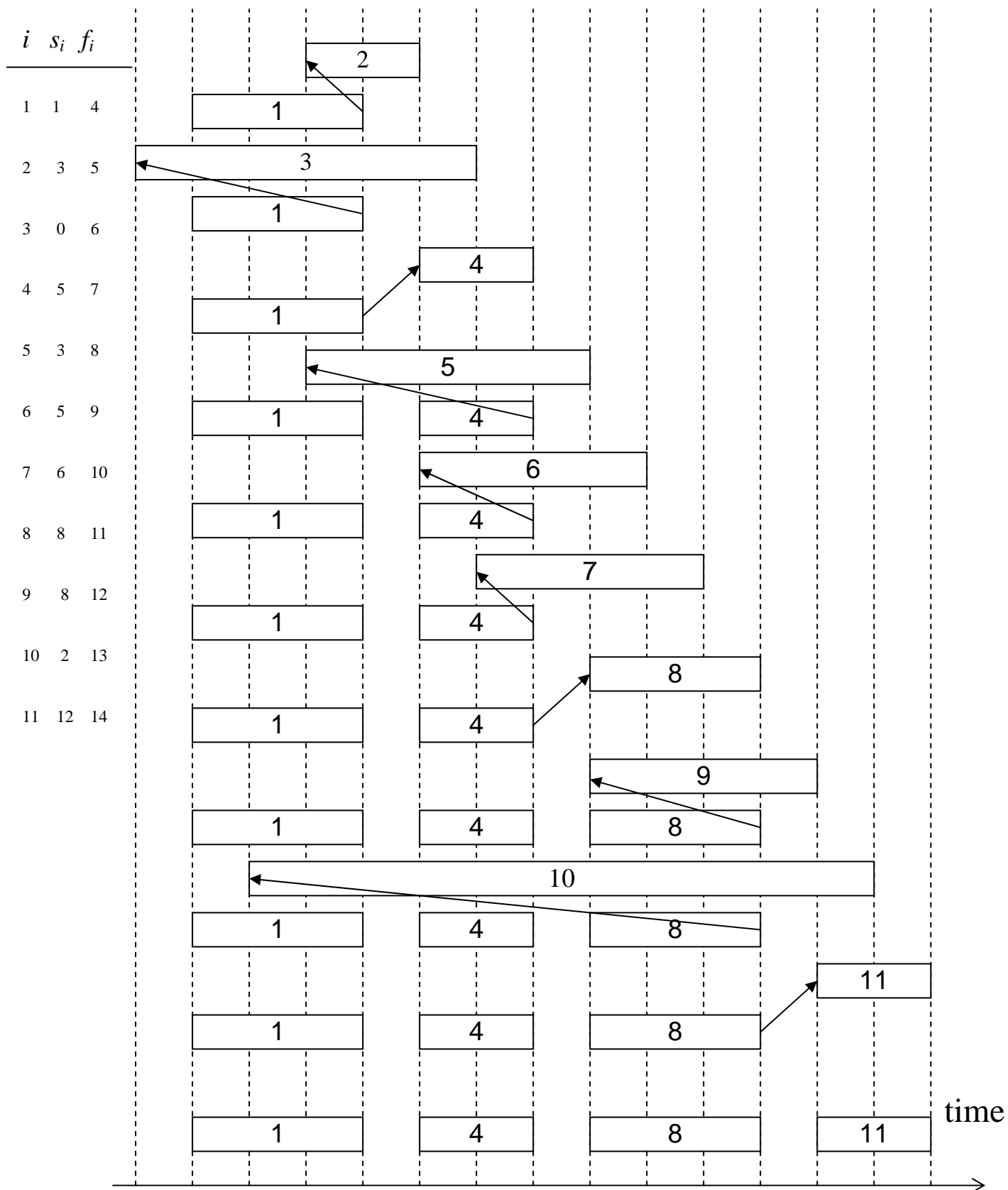


Figure 6.2.1 The operation of Greedy-Activity-Selector on 11 activities given at the left. Each row of the figure corresponds to an iteration of the for loop in lines 4-7. The activities that have been selected to be in set A are shaded, and activity i , shown in white, is being considered. If the starting time s_i of activity i occurs before the finishing time f_j of the most recently selected activity j (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is accepted and put into set A .

The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, a greedy algorithm progresses in a *top-down* fashion, making one greedy choice after another.

Optimal Substructure

A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.

This property is a key element of assessing the applicability of dynamic programming as well as greedy algorithms.

Greedy versus Dynamic Programming

The difference between greedy strategy and dynamic programming is subtle.

The 0-1 *knapsack problem* is defined as follows: “A thief robbing a store finds it contains n types of items of varying size and value (the i th item is worth v_i dollars and weights w_i), but has only a small knapsack of capacity M to use to carry the goods. The knapsack problem is to find the combination of items which the thief should choose for his knapsack in order to maximize the total value of all the items he takes”.

This is called the *0-1 knapsack problem* because each item must either be taken or left behind; the thief cannot take a fraction amount of an item.

In the *fractional knapsack problem*, the set up is the same, but the thief can take *fractions* of items, rather than having to make a binary (0-1) choice for each item.

Both problems have the optimal-substructure property.

For the 0-1 problem, consider the most valuable load that weights at most M pounds. If we remove item j from this load, the remaining load must be most valuable load at most $M - w_j$ that the thief can take from the $n-1$ original items excluding j .

For the fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighting at most $M - w$ that the thief can take from the $n-1$ original items plus $w_j - w$ pounds of item j .

The fractional knapsack problem is solvable by a *greedy strategy* whereas the 1-0 problem is solvable by *dynamic programming*.

To solve the fractional-problem, we first compute the *value per pound* v_i/w_i for each item. The thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the *next* greatest value per pound, and so on until he can't carry any more.

Example: Consider the problem in Figure 6.2.2. There are 3 items, and the knapsack can hold 50 pounds. Item 1 weighs 10 pounds and is worth \$60. Item 2 weighs 20 pounds and is worth \$100. Item 3 weighs 30 pounds and is worth \$120.

```

procedure GREEDY_KNAPSACK(V, W, M, X, n);
/* V, W are the arrays contain the values and weights respectively of the n objects ordered so
that  $V_i/W_i \geq V_{i+1}/W_{i+1}$ . M is the knapsack capacity and X is the solution vector */
var rc: real;
    i: integer;
begin
  for i:= 1 to n do X[i]:= 0;
  rc := M ; // rc = remaining knapsack capacity //
  for i := 1 to n do
    begin
      if W[i] > rc then exit;
      X[i] := 1; rc := rc - W[i]
    end;
    if i ≤ n then X[i] := rc/W[i]
  end

```

Disregarding the time to initially sort the objects, the above algorithm requires only $O(n)$.

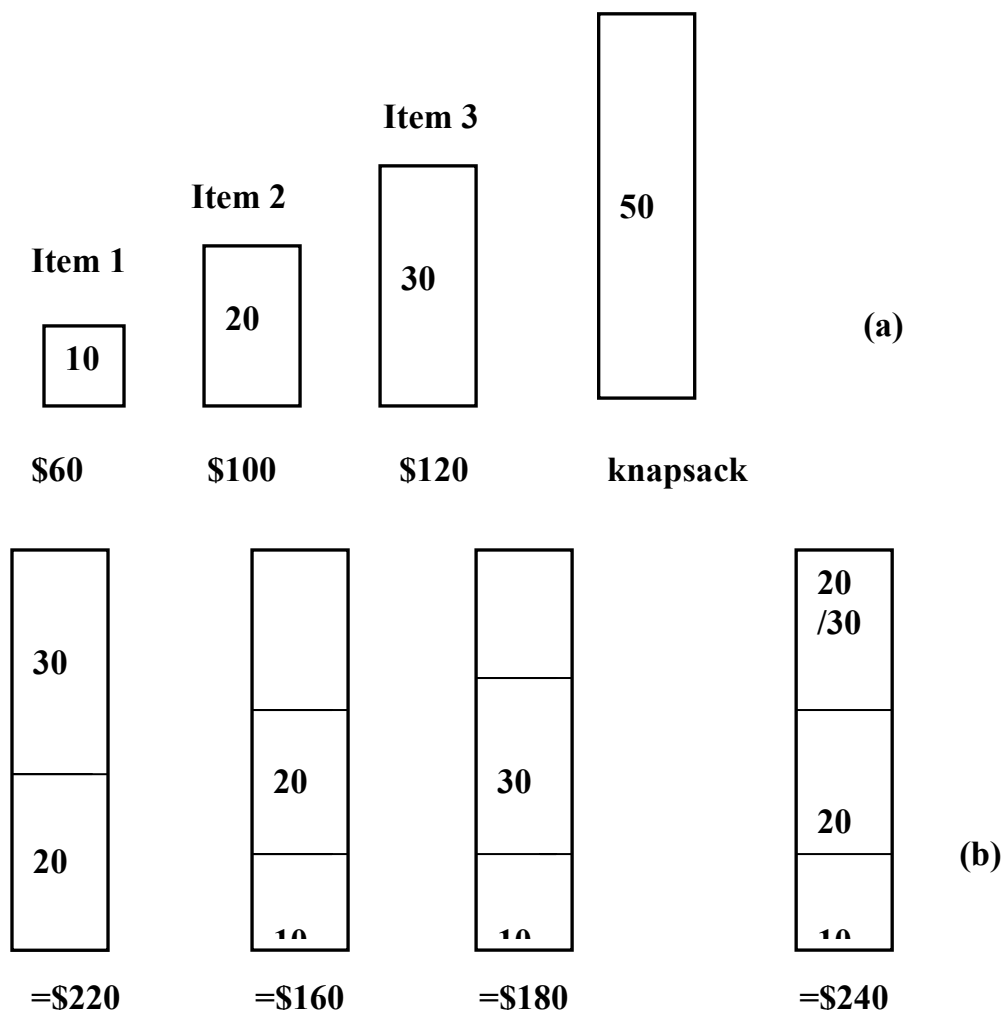


Figure 6.2.2

6.2.2. Huffman Codes

This topic is related to *file compression*. Huffman codes are a widely used and very effective technique for compression data; savings of 20% to 90% are typical.

The first step in building the Huffman code is to count the *frequencies* of each character in the file to be encoded.

Assume that we have a 100000 character data file that we want to store in compressed form. The characters in the file occur with the frequencies given by Table 6.2.1

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Table 6.2.1

We consider the problem of designing a *binary character code* wherein each character is represented by a unique binary string.

If we use a fixed length code, 3 bits to represent six characters :

a = 000, b = 001, . . . , f = 101.

then this method requires 300000 bits to code the entire file.

A *variable-length code* can do better than a fixed-length code, by giving frequent characters short codeword and infrequent characters long codewords.

a = 0, b = 101, . . . f = 1100

This code requires

$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224000$ bits

to represent the file, savings of $\approx 25\%$.

In fact, this is an optimal character code for this file.

Prefix Codes

Here we consider only codes in which no codeword is also a prefix of some other codeword. Such codes are called *prefix-free-codes* or *prefix-codes*.

It is possible to show that the optimal data compression achievable by a character code and always be achieved with a prefix code.

Prefix codes are desirable because they simplify encoding and decoding.

Encoding is simple; we just concatenate the codewords representing each character of the file.

The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off.

A binary tree whose leaves are the given characters provides one such representation.

We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child"

An optimal code for a file is always represented by a *full binary tree*. A full binary tree is a binary tree in which every nonleaf node has two children.

The fixed length code in our example is not optimal since its tree is not a full binary tree.

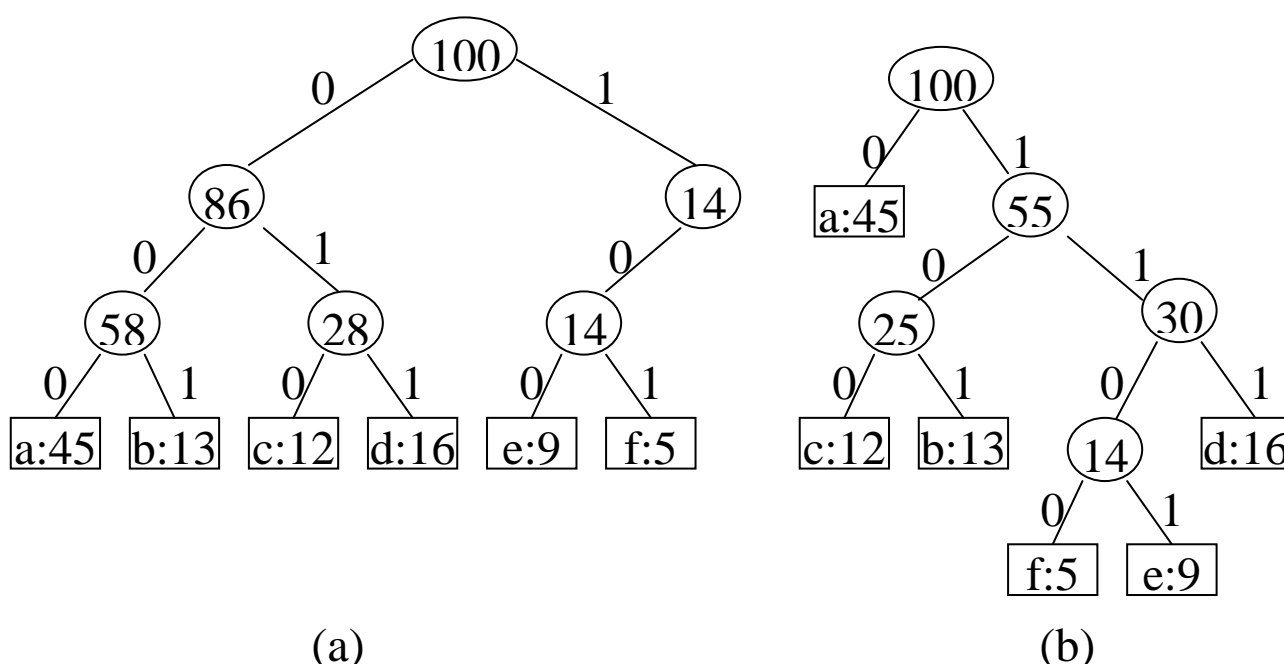
If C is the alphabet from which the characters are taken, then the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C|-1$ internal nodes.

Given a tree T corresponding to a prefix code, we can compute the number of bits required to encode a file.

For each character c in the alphabet C , let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ is also the length of the codeword for character c . The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

which we define as the cost of the tree T .



Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix-code called a *Huffman code*.

The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. Its begin with a set of $|C|$ leaves and perform a sequence of $|C|$ "merging" operations to create the final tree.

A priority queue Q , keyed on f , is used to identify the two *least-frequent* objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
procedure HUFFMAN( $C$ ) ;  
begin  
   $n := |C|$  ;  
   $Q := C$  ;  
  for  $i := 1$  to  $n - 1$  do  
    begin  
       $z := \text{ALLOCATE-NODE}()$  ;  
       $\text{left}[z] := \text{EXTRACT-MIN}(Q)$  ;  
       $\text{right}[z] := \text{EXTRACT-MIN}(Q)$  ;  
       $f[z] := f[\text{left}[z]] + f[\text{right}[z]]$  ;  
       $\text{INSERT}(Q, z)$  ;  
    end  
  end
```

The Huffman 's algorithm proceeds as in Figure 6.2.4

Assume that Q is implemented as a binary heap. For a set C of n characters, the initialization of Q can be performed in $O(n)$ time. The *for* loop is executed exactly $|n|-1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters in $O(n \lg n)$.

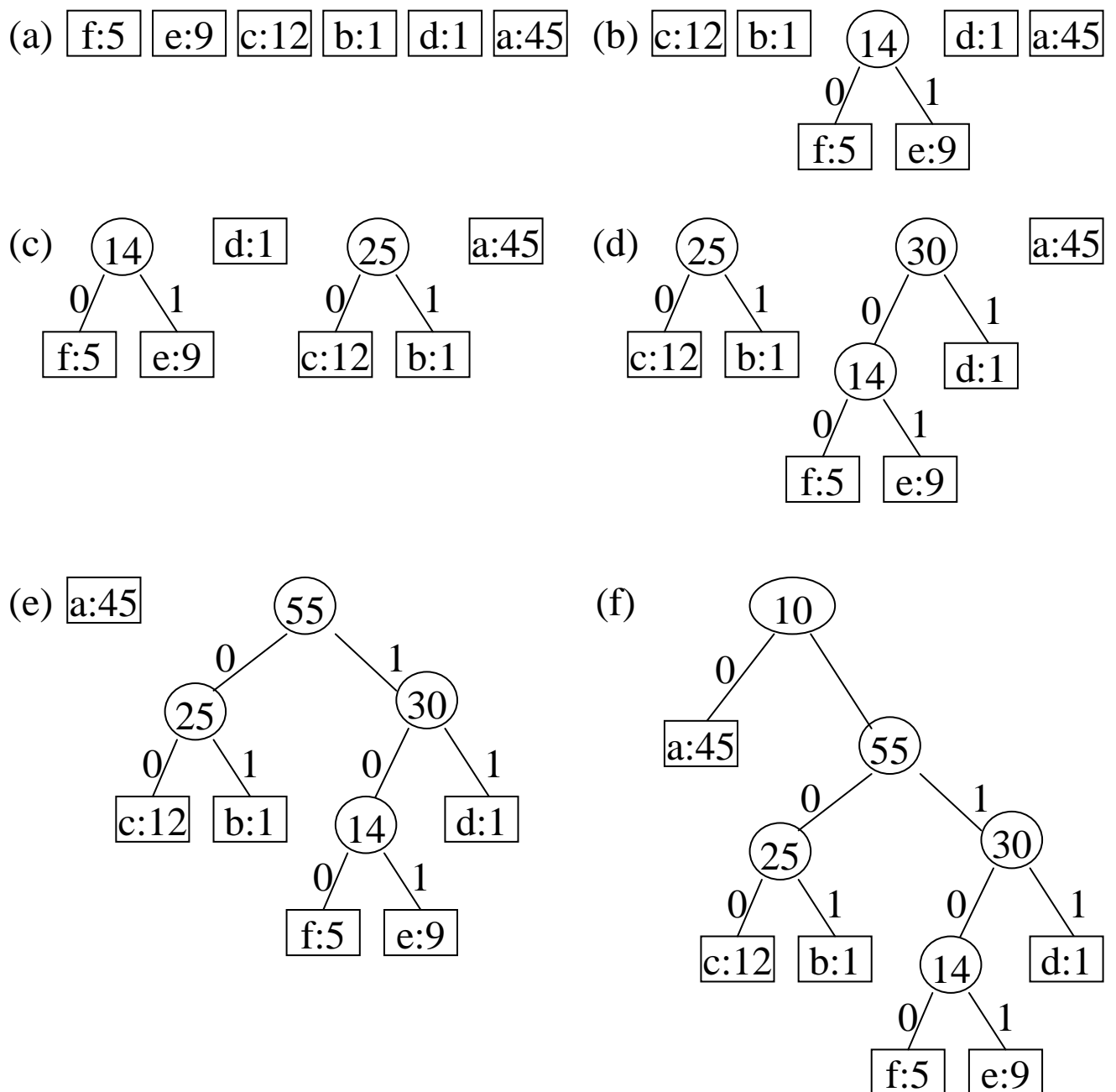


Figure 6.2.4. The steps of Huffman's algorithm for the frequency given in Figure 17.3. Each part shows the contents of the queue sorted into increasing order by frequency.

6.3. BACKTRACKING ALGORITHMS

A general method of problem solving: to design algorithms for finding solutions to specific problems not by following a fixed rule of computation, but by *trial and error*.

The common pattern is to decompose the trial- and-error process into partial tasks. Often these tasks are naturally expressed in recursive terms and consist of *the exploration of a finite number of subtasks*.

We can view the entire process as a search process that gradually constructs and scans a tree of subtasks.

In many problems this *search tree* grows very rapidly, usually exponentially, depending on a given parameter. The search effort increases accordingly.

Frequently, the search tree can be *pruned* by the use of *heuristic* only, therefore reducing computations to tolerable bounds.

6.3.1. The Knight's Tour Problem

Given is a $n \times n$ board with n^2 fields. A knight – being allowed to move according to the rules of chess – is placed on the field with initial coordinates x_0, y_0 .

The problem is to find *a covering of the entire board*, if there exists one, i.e., to compute a tour of $n^2 - 1$ moves such that every field of the board is visited exactly once.

The obvious way to reduce the problem of covering n^2 fields is to consider the problem of either performing a next move or finding out that none is possible.

Let define an algorithm trying to perform a next move.

```
procedure try next move;  
begin  
  initialize selection of moves;  
  repeat  
    select next candidate from list of next moves;  
    if acceptable then  
      begin  
        record move;  
        if board not full then  
          begin  
            try next move; (6.3.1)  
            if not successful then erase previous recording  
          end  
        end  
      until (move was successful)  $\vee$  (no more candidates)  
    end  
end
```

Data Representation

We represent the board by a matrix h . Let us also introduce a type to denote index values:

type index = 1..n ;

var h: **array**[index, index] **of** integer;

$h[x, y] = 0$: field $\langle x, y \rangle$ has not been visited

$h[x, y] = i$: field $\langle x, y \rangle$ has been visited in the i th move ($1 \leq i \leq n^2$)

Condition “board not full” can be expressed as “ $i < n^2$ ”.

u, v : the coordinated of possible move destination.

Condition “acceptable” can be expressed as $(1 \leq u \leq n) \wedge (1 \leq v \leq n) \wedge (h[u, v] = 0)$

Now, we come to a more refined program:

procedure try(i : integer; x, y : index; **var** q : boolean);

var u, v : integer; $q1$: boolean;

begin

 initialize selection for moves;

repeat

 let u, v be the coordinates of the next move

 defined by the rules of chess;

if $(1 \leq u \leq n) \wedge (1 \leq v \leq n) \wedge (h[u, v] = 0)$ **then**

begin

$h[u, v] := i$;

if $i < \text{sqr}(n)$ **then** (6.3.2)

begin

 try($i + 1, u, v, q1$);

if $\neg q1$ **then** $h[u, v] := 0$

end

else $q1 := \text{true}$

end

until $q1 \vee$ (no more candidates);

$q := q1$

end

Given a starting coordinate pair $\langle x, y \rangle$, there are eight potential candidates for coordinates $\langle u, v \rangle$ of the destination. They are numbered 1 to 8 in Figure 6.3.1

	3		2	
4				1
		\oplus		
5				8
	6		7	

Figure 6.3.1 The eight possible moves of a knight

A simple method of obtaining u , v from x , y is by addition of the coordinate differences stored in two arrays of single differences. Let these arrays be a and b , appropriately initialized.

And k is used to number the “next candiate.”

The details are shown in following procedure:

```

program knightstour (output);
const n = 5; nsq = 25;
type index = 1..n
var   i,j: index;
       q: boolean;
       s: set of index;
       a,b: array [1..8] of integer;
       h: array [index, index] of integer;

procedure try (i: integer; x, y: index; var q: boolean);
var k,u,v : integer; q1: boolean;
begin
    k:=0;
    repeat
        k:=k+1; q1:=false;
        u:=x+a[k]; v:=y+b[k];
        if (u in s)  $\wedge$  (v in s) then
            if h[u,v]=0 then
                begin h[u,v]:=i;
                    if i < nsq then
                        begin
                            try(i+1, u,v,q1);
                            if  $\neg$  q1 then h[u,v]:=0
                        end
                    else q1:=true
                end
            until q1  $\vee$  (k =8);
        q:=q1
    end {try};
begin
    s:=[1,2,3,4,5];
    a[1]:= 2;  b[1]:= 1;
    a[2]:= 1;  b[2]:= 2;
    a[3]:= -1; b[3]:= 2;
    a[4]:= -2; b[4]:= 1;
    a[5]:= -2; b[5]:= -1;
    a[6]:= -1; b[6]:= -2;

```

```

a[7]:= 1; b[7]:= -2;
a[8]:= 2; b[8]:= -1;
for i:=1 to n do
    for j:=1 to n do h[i,j]:=0;
h[1,1]:=1; try (2,1,1,q);
if q then
    for i:=1 to n do
        begin
            for j:=1 to n do write(h[i,j]:5);
            writeln
        end
    else writeln ('NO SOLUTION')
end.

```

The recursive procedure is initiated by a call with the coordinate x_0, y_0 as parameters, from which the tour is to start. This field must be given the value 1; all others are to be marked free.

$H[x_0, y_0] := 1; \text{try}(2, x_0, y_0, q)$

Figure 6.3.1 indicates solution obtained with initial position $\langle 1, 1 \rangle$ for $n = 5$.

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

Table 6.3.1 A Knight's Tour

From the above example, we get a new kind of “problem solving”:

The characteristic feature is that steps toward the total solution are attempted and recorded that may be later be taken back and erased in the recordings when it is discovered that the step does not lead to the total solution, i.e. the step leads into a “*dead-end*.” This action is called *backtracking*.

The general pattern of a backtracking algorithm:

```

procedure try;
begin initialize selection of candidates;
    repeat
        select next;
    if acceptable then
        begin
            record it;
            if solution incomplete then
                begin

```



```

        try next step;                                (6.3.3)
        if not successful then cancel recording
    end
end
until successful  $\vee$  no more candidates
end

```

Moreover, if at each step the number of candidates to be investigated is fixed, then the above pattern can be modified as follows:

```

procedure try (i: integer);
var k : integer;
begin k:=0;
    repeat
        k:=k+1; select k-th candidate;
        if acceptable then
            begin
                record it;
                if i<n then                                (6.3.4)
                    begin
                        try (i+1);
                        if not successful then cancel recording
                    end
                end
            until successful  $\vee$  (k=m)
    end

```

The procedure is to be invoked by the statement *try(1)*.

6.3.2. The Eight Queen's Problem

It was investigated by C.F. Gauss in 1850, but he did not completely solve it.

Eight queens are to be places on a chess board in such a way that no queen checks against any other queen.

Using the general pattern in section 6.3.1, we obtain the following version of the procedure:

```

procedure try (i: integer);
begin
    initialize selection of positions for i-th queen;
    repeat
        make next selection;
        if safe then
            begin
                setqueen;
            end
        end
    until successful

```

```

    if i < 8 then
    begin
        try (i + 1);
        if not successful then remove queen
    end
end
until successful ∨ no more positions
end

```

The rules of chess: A queen can attack all other queens lying in either the same column, row, or diagonals on the board.

Data Representation

How to represent the eight queens on the board?

```

var   x: array[1..8] of integer;
      a: array[1..8] of Boolean;
      b: array[b1..b2] of Boolean;
      c: array[c1..c2] of Boolean;

```

where $x[i]$ denotes the position of the queen in the i th column;
 $a[j]$ denotes no queen lies in the j th row;
 $b[k]$ means no queen occupies the k th \swarrow diagonal
 $c[k]$ means no queen sits on the k th \searrow diagonal.

The choice for index bounds $b1$, $b2$, $c1$, $c2$ is dictated by the way that indices of b and c are computed. We note that in the \swarrow diagonal all fields have the same sums of their coordinates i and j , and in a \searrow diagonal, the coordinate differences $i - j$ are constant.

Given these data, the statement *setqueen* is refined to:

$x[i] := j$; $a[j] := \text{false}$; $b[i+j] := \text{false}$; $c[i-j] := \text{false}$;

The statement *removequeen* is refined to:

$a[j] = \text{true}$; $b[i+j] = \text{true}$; $c[i-j] := \text{true}$

The condition *safe* is expressed as:

$a[j] \wedge b[i+j] \wedge c[i-j]$

Figure 6.3.2. A solution to the eight queens problem.

H							
						H	
				H			
							H
	H						
			H				
					H		
		H					
1	2	3	4	5	6	7	8

The full program is shown as follows:

```
program eightqueen1(output);
{ find one solution to eight queens problem}
var   i : integer; q: boolean;
      a : array [1..8] of boolean;
      b : array [2..16] of boolean;
      c : array [-7..7] of boolean;
      x : array [1..8] of integer;
procedure try(i: integer; var q: boolean);
var j: integer;
begin
  j:=0;
  repeat
    j:=j+1; q:=false;
    if a[j]  $\wedge$  b[i+j]  $\wedge$  c[i-j] then
      begin
        x[i]:=j;
        a[j]:=false; b[i+j]:=false; c[i-j]:=false;
        if i<8 then
          begin
            try (i+1, q);
            if  $\neg$  q then
              begin
                a[j]:=true; b[i+j]:=true; c[i-j]:=true
              end
            end
            else q:=true
          end
        until q  $\vee$  (j=8)
      end {try};
  begin
    for i:= 1 to 8 do a[i]:=true;
    for i:= 2 to 16 do b[i]:=true;
    for i:= -7 to 7 do c[i]:=true;
    try (1,q);
    if q then
      for i:=1 to 8 do write (x[i]:4);
    writeln
  end
```

Extension: Finding all Solutions

The extension is to find not only one, but all solutions to a posed problems.

Method: Once a solution is found and recorded, we proceed to the next candidate delivered by the systematic candidate selection process.

The general pattern is derived from (6.3.4) and shown as follows:

```
procedure try(i: integer);  
var k: integer;  
begin  
  for k:=1 to m do  
    begin  
      select k-th candidate;  
      if acceptable then  
        begin  
          record it;  
          if i<n then try (i+1) else print solution;  
          cancel recording  
        end  
      end  
    end  
end
```

In the extended algorithm, to simplify the termination condition of the selection process, the *repeat* statement is replaced by a *for* statement.

```
program eightqueens(output);  
var i: integer;  
    a: array [1.. 8] of boolean;  
    b: array [2.. 16] of boolean;  
    c: array [-7.. 7] of boolean;  
    x: array [1.. 8] of integer;  
procedure print;  
var k : integer;  
begin  
  for k:=1 to 8 do write(x[k]:4);  
  writeln  
end {print};  
procedure try (i:integer);  
  var j: integer;  
begin  
  for j:=1 to 8 do  
    if a[j]  $\wedge$  b[i+j]  $\wedge$  c[i-j] then  
      begin  
        x[i]:=j;  
        a[j]:=false; b[i+j]:= false; c[i-j]:=false;  
        if i < 8 then try(i+1) else print;  
        a[j]:=true; b[i+j]:= true; c[i-j]:= true;  
      end  
    end  
end {try};  
begin
```

```

for i:= 1 to 8 do a[i]:=true;
for i:= 2 to 16 do b[i]:=true;
for i:= -7 to 7 do c[i]:=true;
try(1);
end.

```

The extended algorithm can produce all 92 solutions of the eight queens problem. Actually, there are only 12 significantly different solutions. The 12 solutions are listed as in the following table.

x₁	x₂	x₃	x₄	x₅	x₆	x₇	x₈	N
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	72
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

Table 6.3.2 Twelve Solutions to the 8 queens problems

The numbers N indicates the number of tests for safe fields. Its average over all 92 solutions is 161.

It is true that the running time of backtracking algorithms remains *exponential*.

Roughly, if each node in the search tree has α children, on the average, and the length of the solution path is N, then the number of nodes in the tree to be proportional to α^N .

Chapter 7. NP-COMPLETE PROBLEMS

7.1. NP-COMPLETE PROBLEMS

For many problems we have several efficient algorithms to solve.

Unfortunately, so many other problems in practice do not have efficient solving algorithms.

And for a large class of such problem we can't even tell *whether or not an efficient algorithm might exist*.

A lot of research has been done and has led to the development of *mechanisms* by which new problems can be classified as being “as difficult as” some old problems.

Sometimes the line between "easy" and "hard" problems is a fine one.

Example:

Easy: Is there a path from x to y with $\text{weight} \leq M$?

Hard: Is there a path from x to y with $\text{weight} \geq M$?

Breadth-first-search produces a solution for the first problem in linear time, but all known algorithms for the second problem could take exponential time.

Deterministic and Nondeterministic Polynomial Time Algorithms

P: the set of all problems that can be solved by deterministic algorithm in polynomial time.

“*Deterministic*” means that whatever the algorithm is doing, there is only one thing it could do next.

Example: Sorting belong to P because insertion sort runs in time proportional to N^2 .

One way to extend the power of a computer is to give it with the power of nondeterminism.

Nondeterministic means when an algorithm is faced with a choice of several options, it has the power to "guess" the right one.

Nondeterministic Algorithms

Example: Let A is an unsorted array of positive integers. The *nondeterministic* algorithm NSORT(A, n) sorts the numbers into ascending order and then outputs them in this order. An auxiliary array B is used as temporary array.

```

procedure NSORT(A,n)
// sort n positive integers //
begin
  for i:= 1 to n do B[i]:= 0;
  for i:= 1 to n do
    begin
      j := choice(1:n);
      if B[j] <> 0 then failure
      else B[j]:= A[i]
    end
  for i:= 1 to n-1 do
    if B[i] > B[i-1] then failure;
  print(B);
  success
end;

```

The order of the numbers in B is some permutation of the initial order in A.

A deterministic interpretation of a non-deterministic algorithm can be made by allowing *unbounded parallelism* in computation.

Each time a choice is to be made, the algorithm makes several *copies of itself*. One copy is made for each of the possible choices. Thus, many copies are executing at the same time.

- The first copy to reach a successful completion terminates all other computations.
- If a copy reaches a failure completion then only that copy of the algorithm terminates.

In fact, a *nondeterministic machine* does not make any copies of an algorithm every time a choice is to be made. Instead, it has the ability to select an “correct” element from the set of allowable choices every time a choice is to be made.

A “correct” element is defined relative to the shortest sequence of choices that leads to a successful termination.

In case there is *no* sequence of choices leading to a successful termination, we’ll assume that the algorithm terminates in one unit of time with output “unsuccessful computation.”

Note:

1. The *success* and *failure* signals are equivalent to *stop* statement in deterministic algorithm.
2. The complexity of NSORT is $O(n)$.

NP: the set of all problems that can be solved by nondeterministic algorithms in polynomial time.

Example : The "yes-no" version of the longest path problem is in NP.

The *circuit satisfiability* problem (CSP). Given a logical formula of the form

$$(x_1 + x_3 + x_5) * (x_1 + \sim x_2 + x_4) * (\sim x_3 + x_4 + x_5) * (x_2 + \sim x_3 + x_5)$$

where the x_i 's represent Boolean variables (*true* or *false*), "+" represent OR, "*" represents AND, and \sim represent NOT.

The CSP is to determine whether or not there exists an assignment of truth values to the variables that makes the formula *true*.

CSP is also a NP problem.

Note: The P class is a subclass of NP.

7.2. NP-COMPLETENESS

There are a list of problems that are known to belong to NP but might or might not belong to P. (That is, they are easy be solved on a non-deterministic machine but, no one has been able to find an efficient algorithm on a conventional machine for any of them).

These problems have an additional property: "If *any* of these problems can be solved in polynomial time on a deterministic machine, then *all* problems in NP can be also solved in polynomial time on a deterministic machine."

Such problems are said to be *NP-complete*.

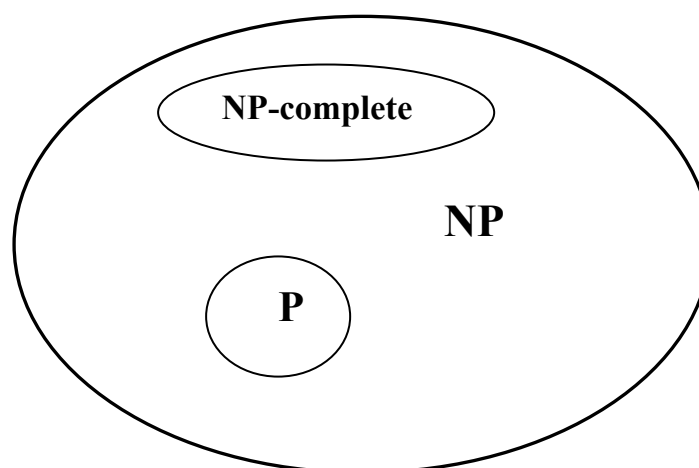


Figure 7.1

The subset *NP-complete* problems are the hardest problems within NP class.

The primary tool used to prove that problems are NP-complete employs the idea of *polynomial reducibility*.

Any algorithm to solve a new problem in NP can be used to solve some known NP-complete problem by the following process:

transform any instance of the known NP-complete problem to an instance of the new problem, solve the problem using the given algorithm, then transform the solution back to a solution of the NP-complete problem.

To prove a problem in NP is NP-complete, we need only show that some known NP-complete problem is polynomially reducible to it: that is, that *a polynomial-time algorithm for the new problem can be used to solve the known NP-complete problem, and then can, in turn, be used to solve all problems in NP.*

Definition: (*Reduces to*) We say the problem L_1 *reduces to* L_2 , written $L_1 \alpha L_2$ if any algorithm for L_2 can be used for L_1 .

To prove that the new problem L is NP-complete, we should prove:

- (1) The problem L belongs to NP
- (2) A known NP-complete problem reduces to L .

Example:

TRAVELING SALESMAN: Give a set of cities and distances between all pairs, find a tour of all the cities of distance less than M .

HAMILTON CYCLE: Given a graph, find a simple cycle that includes all the vertices.

Suppose we know HCP to be *NP-complete* and wish to determine whether or not TSP is also *NP-complete*. Any algorithm for solving the TSP can be used to solve the HCP, through the following reduction:

Given an instance of the HCP (a graph), construct an instance of TSP (a set of cities, with distances between pairs) as follows:

- for *cities* for the TSP use the set of *vertices* in the graph;
- for *distances* between each pair of cities use 1 if there is an edge between the corresponding vertices in the graph, 2 if there is no edge.

Then use the algorithm for the TSP to find a tour of distance $\leq N$ (N is the number of vertices in the graph). An efficient algorithm for the TSP would also be an efficient algorithm for the HCP.

That is the HCP *reduces to* the TSP, so the NP-completeness of HCP implies the NP-completeness of the TSP.

The reduction of the HCP to the TSP is relatively simple because the problems are similar. Actually, *polynomial time reductions* can be complicated when we connect problems which seem to be quite dissimilar.

Example: It's possible to reduce the circuit satisfiability problem to the HCP.

7.3. COOK'S THEOREM

Reduction uses the NP-completeness of one problem to imply the NP-completeness of another. But: how was the first problem to be NP-complete?

S.A. Cook (1971) gave the direct proof that the circuit satisfiability problem is NP-complete.

“If there is a polynomial time algorithm for the circuit-satisfiability-problem, then all problems in NP can be solved in polynomial time.”

The Cook's proof is extremely complex, but it is mainly based on the general-purpose computer known as a *Turing machine*.

7.4. Some NP-Complete Problems

Thousands of diverse problems are known to be NP-complete. The list begins with circuit-satisfiability, traveling-salesman and Hamilton cycle.

Some additional problems are as follows:

PARTITION: Given a set of integers, can they be divided into two sets whose sum is equal?

INTEGER LINEAR PROGRAMMING: Given a linear program, is there a solution in integers?

MULTIPROCESSOR SCHEDULING: Given a deadline and a set of tasks of varying length to be performed on two identical processors, can the tasks be arranged so that the deadline is met.

VERTEX COVER: Give a graph and an integer N , is there a set of fewer than N vertices which touches all the edges?

These and many related problems have important practical applications.

The fact that no good algorithms has been found for any of these problems is strong evidence that $P \neq NP$.

Whether or not $P = NP$, the practical fact is that we have at present no algorithms guaranteed to solve any of the NP-complete problems efficiently.

Several techniques have been developed to cope with NP-complete problems.

1. One approach is to change the problem and find an “approximation algorithm” that finds not the best solution but a near-optimal solution.

2. Another approach is to rely on “average time” performance and develop an algorithm that finds the solution in some cases, but doesn’t necessarily work in all cases.
3. A third approach is to work with “efficient” exponential algorithms, using backtracking techniques.
4. **Heuristic** approach

There are NP-complete problems in

- numerical analysis,
- sorting and searching,
- string processing,
- geometry modeling and
- graph processing.

The most important contribution of the theory of NP-completeness is that it provides a mechanism to discover whether a new problem from any of these areas is “easy” or “hard”.

We classify problems into four classes according to their degrees of difficulty.

1. *Undecidable problems* (unsolvable problems) : These are the problems for which no algorithm can be written.

Example: The problem of deciding whether a program will halt on a Turing machine.

2. *Intractable problems* (provably difficult problems): These are the problems which no polynomial algorithm can be developed to solve them. Only exponential algorithms can solve them.
3. NP problems
The class of NP-complete problems is a subclass of NP.
4. P-problems.

EXERCISES

Chapter 1

1. Translate the recursive procedure *Hanoi* to non-recursive version by first using tail-recursion removal and then applying the general method of recursion removal.

2. Given the following procedure **hanoi**:

```
procedure hanoi(n, beg, aux, end);  
begin  
  if n = 1 then  
    writeln(beg, end)  
  else  
    begin  
      hanoi(n-1, beg, end, aux) ;  
      writeln(beg, end);  
      hanoi(n-1, aux, beg, end);  
    end  
end;
```

Let $C(n)$ be the number of disk moves from a peg to another peg. Find the recurrence relation for the above program. And prove that $C(n) = 2^n - 1$.

3. The Ackermann Function A is defined for all non-negative integer arguments m and n as follows:

$$\begin{aligned} A(0, n) &= n+1 \\ A(m, 0) &= A(m-1, 1) && (m > 0) \\ A(m, n) &= A(m-1, A(m, n-1)) && (m, n > 0) \end{aligned}$$

- a) Write a recursive function that computes $A(m, n)$.
- b) Applying the general method, convert the recursive function in a) to a non-recursive.

4. Applying the general method, convert the following recursive procedure to a non-recursive.

```
integer function DANDC(p,q)  
/* n and array A(1:n) are global variables */  
begin  
  integer m, p, q; /* 1 ≤ p ≤ q */  
  if p < q then DANDC:= G(p,q);  
  else  
    begin  
      m := DIVIDE(p,q); /* p ≤ m < q */  
      DANDC:= COMBINE(DANDC(p,m), DANDC(m+1, q))  
    end  
end;
```

5. Write a recursive procedure for *postorder* tree traversal algorithm and then convert it to non-recursive procedure.

6. Given the following procedure that finds the maximum and minimum elements in an array.

procedure MAXMIN(A, n, max, min)

/* Set max to the maximum and min to the minimum of A(1:n) */

begin

integer i, n;

max := 1; min:= 1;

for i:= 2 **to** n **do**

if A[i] > max **then** max := A[i]

else if A[i] < min **then** min := A[i];

end

Let C(n) be the complexity function of the above algorithm, which measures the number of element comparisons.

- (a) Describe and find C(n) for the worst-case.
- (b) Describe and find C(n) for the best-case.
- (c) Describe and find C(n) for the average-case.

7. Suppose Module A requires M units of time to be executed, where M is a constant. Find the complexity C(n) of the given algorithm, where n is the size of the input data and b is a positive integer greater than 1.

j:= 1;

while j <= n **do**

begin

call A;

j := j*b

end

8. Consider the merging algorithm that merges two sorted arrays into one sorted array, as follows:

/* Two sorted arrays a[1..M] and b[1..N] of integers which we want to merge into a third array c[1..M+N] */

i:= 1; j :=1;

for k:= 1 **to** M+N **do**

if a[i] < b[j] **then**

begin c [k] := a[i]; i:= i+1 **end**

else

begin c[k] := b[j]; j := j+1 **end**;

What is the time complexity of this algorithm?

9. Given a recursive program with the following recurrence relation:

$$C_N = 4C_{N/2} + N, \quad \text{for } N \geq 2 \text{ with } C_1 = 1$$

when N is a power of two.

Solve the recurrence.

10. Given a recursive program with the following recurrence relation:

$$C(n) = c + C(n-1) \quad \text{for } n > 1 \text{ with } C(1) = d$$

where c, d are two constants.

Solve the recurrence

11. Given a recursive program with the following recurrence relation:

$$C(n) = 2C(n/2) + 2 \quad \text{for } n > 1 \text{ with } C(2) = 1$$

Solve the recurrence

12. Given a recursive program with the following recurrence relation:

$$C(n) = 2C(n/2) + 3 \quad \text{for } n > 1 \text{ with } C(2) = 1$$

Solve the recurrence

13. Given a recursive program with the following recurrence relation:

$$C_N = 4C_{N/2} + N^2, \quad \text{for } N \geq 2 \text{ with } C_1 = 1$$

when N is a power of two.

Solve the recurrence.

Chapter 3

1. Analyze the computational complexity of *insertion sort* in both worst-case (when the list of numbers are in reverse order) and average-case.

2. Write an algorithm that finds the k -th smallest element in an array of size N by modifying *selection-sort* algorithm.

3. Write the Quicksort algorithm that uses the rightmost element as the pivot (by modifying the *quicksort2* procedure).

4. Given the following list of integers 66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30. Trace by hand the Quicksort algorithm that uses the leftmost element as the pivot to sort these integers.

5. If the array is already in ascending order, estimate the total number of comparisons when we apply *Quicksort* on that array. Derive the worst-case complexity of the Quicksort.

6. Show the merges done when the recursive Mergesort is used to sort the keys E A S Y Q U E S T I O N.

State the time complexity of heap-sort.

7. Trace by hand the action of radix exchange sort on the following list: 001, 011, 101, 110, 000, 001, 010, 111, 110, 010.

State the time complexity of radix exchange-sort.

8. Given the data file of 23 records with the following keys: 28, 3, 93, 10, 54, 65, 30, 90, 10, 69, 8, 22, 31, 5, 96, 40, 85, 9, 39, 13, 8, 77, 10.

Assume that one record fits in a block and memory buffer holds at most three page frames. During the merge stage, two page frames are used for input and one for output. Trace by hand the external sorting (external sort-mege) for the above data file.

9. Give the recursive implementation of binary search. Analyze the time complexity of binary search.

Chapter 4

1. Prove the following property:

Sequential search (sorted linked list implementation) uses about $N/2$ comparisons for both successful and unsuccessful search (on the average).

2. Draw the binary search tree that results from inserting into an initially empty tree records with the keys E A S Y Q U E S T I O N, and then delete Q.

In the average-case, how many comparisons can a search in a binary search tree with N keys require?

3. Draw the binary search tree that results from inserting into an initially empty tree records with the keys 5, 10, 30, 22, 15, 20, 31. And then delete 10 from the tree

In the worst case, how many comparisons can a search in a binary search tree with N keys require? Explain your answer.

4. Write a recursive program to compute the *height* of a binary tree: the longest distance from the root to an external node.

5. Write a nonrecursive program to print out the keys from a binary search tree in order.

6. Give the heap constructed by successive application of insert on the keys E A S Y Q U E S T I O N.

7. Given the heap-sort algorithm:

$N := 0;$

for $k := 1$ **to** M **do** insert($a[k]$);

for $k := M$ **downto** 1 **do** $a[k] := \text{remove};$

By hand, trace the action of heap-sort on the following list of keys 44, 30, 50, 22, 60, 55, 77, 55. State the time complexity of heap-sort.

8. Give the contents of the hash table that results when the keys E A S Y Q U E S T I O N are inserted in that order into an initially empty table of size 13 using linear probing. (Use $h(k) = k \bmod 13$ for the hash function for the k -th letter of the alphabet and assume that the decimal value of 'A' is 0, of 'B' is 2, etc..).

9. Give the contents of the hash table that results when the keys E A S Y Q U E S T I O N are inserted in that order into an initially empty table of size 13 using separate chaining. (Use

$h(k) = k \bmod 13$ for the hash function for the k -th letter of the alphabet assume that the decimal value of 'A' is 0, of 'B' is 2, etc..).

10. Given a hash table using separate chaining for collision handling. Write two functions *hash_search()* and *hash_insert()* for searching a search key v in the hash table and inserting a search key v into the hash table, respectively.

11. How long could it take in the worst case to insert N keys into an initially empty table, using separate chaining with unordered lists? Answer the same question for sorted lists.

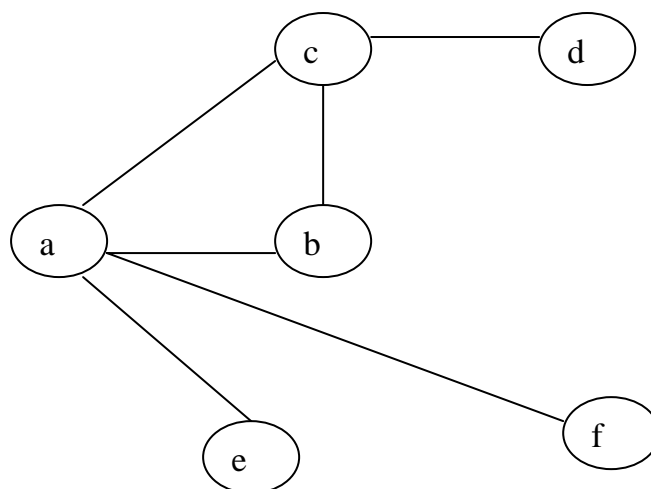
12. Implement a naïve string matching algorithm that scans the pattern from right to left.

13. Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

Chapter 5

1. Describe an algorithm to insert and delete edges in the adjacency list representation for an undirected graph. Remember that an edge (i, j) appears on the adjacency lists for both vertex i and vertex j .

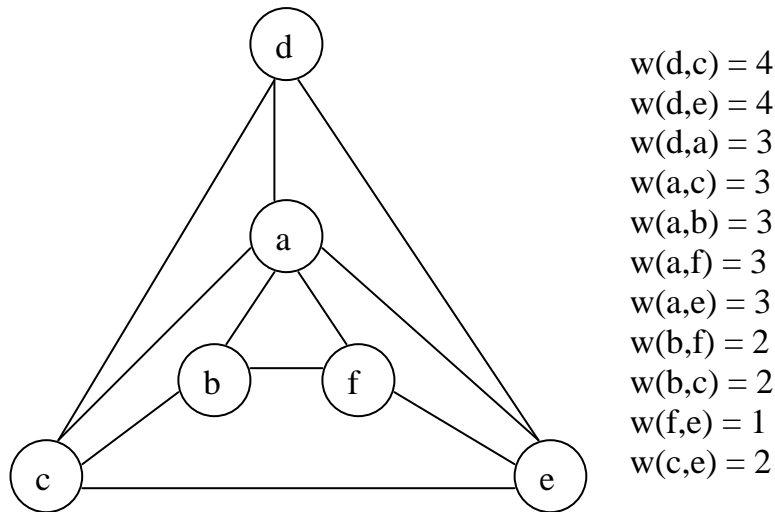
2. Given an undirected graph as follows:



- Construct the adjacency list representation of the above graph.
- Construct the adjacency matrix that represents the graph.
- By hand, trace step by step the status of the stack when you use it in a *depth-first-search* on the above graph (starting from vertex a). Then show the corresponding order in which the vertices might be processed during the *depth-first-search*.
- State the time complexity of *depth-first-search*.
- By hand, trace step by step the status of the queue when you use it in a *breadth-first-search* on the above graph (starting from vertex a). Then show the corresponding order in which the vertices might be processed during the *breadth-first-search*.

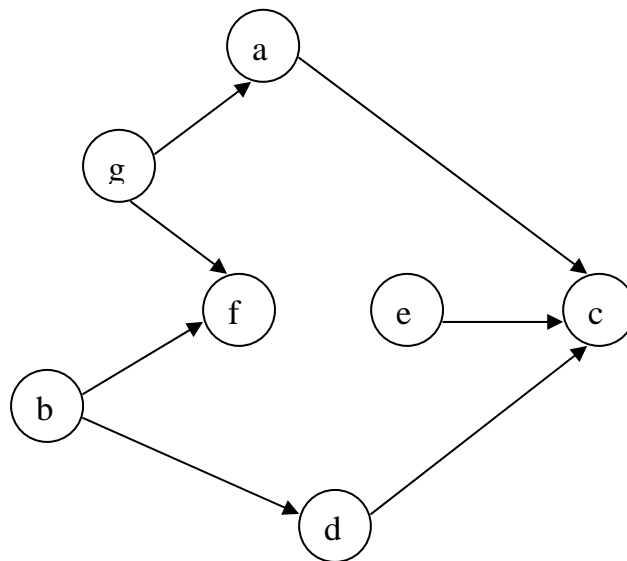
3. Modify the depth-first-search algorithm in order that it can be used to check whether a graph G has a cycle.

4. Given the following weighted graph



Trace the actions of finding a minimum spanning tree, using Prim's algorithm.

5. Given the directed graph



- Construct an adjacency list representation for the above directed graph.
- Using method 1, find two different topological sorts for the above directed graph.
- Using method 2, find two different topological sorts.

6. Given a directed graph whose adjacency-matrix is as follows:

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- a. Show its adjacency-list representation.
- b. Apply Warshall algorithm to find the transitive closure of the above directed graph (You have to show the matrices of 4 stages: $y = 1, y=2, y = 3, y = 4$).

7. Given a weighted, directed graph whose adjacency-matrix is as follows:

$$A = \begin{bmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{bmatrix}$$

Apply Floyd's algorithm to solve the all-pairs shortest path problem of the above directed graph (You have to show the matrices of 4 stages: $y = 1, y=2, y = 3, y = 4$).

8. True/false: Topological sorting can be used to check if there is a cycle in a directed graph. Explain your answer.

9. If array is used to implement the priority queue in the Prim's algorithm, analyse the worst-case complexity of the algorithm (assume that *adjacency list* representation is used for the undirected graph).

10.

a. Modify the Floyd algorithm in order that you can recover the shortest path from one vertice to another.

Hint: Use another matrix P , where $P[x,j]$ holds the vertex y that led Floyd algorithm to find the smallest value of $a[x,j]$. If $P[x,j] = 0$ then the shortest path from x and j is direct, following the edge from x to j .

b. Develop the procedure *path* that prints out the shortest path from one vertex to another.

Chapter 6

1. Consider the problem of finding the n th Fibonacci number, as defined by the recurrence equation

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2)$$

Develop a dynamic programming algorithm for finding the n th Fibonacci number.

2. Modify the dynamic programming algorithm for 0-1 knapsack problem to take into account another constraint defined by an array $num[1..N]$ which contains the number of available items of each type.

3. We can recursively define the number of combinations of m things out of n , denoted $C(m,n)$, for $n \geq 1$ and $0 \leq m \leq n$, by

$$C(m, n) = 1 \text{ if } m = 0 \text{ or } m = n$$

$$C(m, n) = C(m, n-1) + C(m-1, n-1) \text{ if } 0 < m < n$$

- a) Give a recursive function to compute $C(m, n)$.
- b) Give a dynamic programming algorithm to compute $C(m, n)$. *Hint:* The algorithm constructs a table generally known as Pascal's triangle.

4. Modify the greedy algorithm for fractional knapsack problem to take into account another constraint defined by an array $num[1..N]$ which contains the number of available items of each type.

5. Given the following characters and their occurrence frequencies in the text file:

Character	Frequency
A	12
B	40
C	15
D	8
E	25

Find the Huffman codes for these above characters. What is the average code length?

6. Develop a backtracking algorithm that can generate all permutations of N distinct items a_1, \dots, a_n .

Hint: Consider the task of generating all permutations of the elements a_1, \dots, a_m as consisting of the m subtasks of generating all permutations of a_1, \dots, a_{m-1} followed by a_m , where in the i th subtask the two elements a_i and a_m had initially been interchanged.

7. A *coloring* of a graph is an assignment of a color to each vertex of the graph so that no two vertices connected by an edge have the same color. Develop a recursive backtracking algorithm for graph coloring problem. Assume that the graph is represented by adjacency-matrix.

REFERENCES

- [1] Sedgewick, R., *Algorithms*, Addison – Wesley, 1990.
- [2] Cormen, T. H., Leiserson, C. E., and Rivest, R.L., *Introduction to Algorithms*, The MIT Press, 1997.
- [3] Kingston, J. H., *Algorithms and Data Structures – Design, Correctness, Analysis*, Addison – Wesley, 1990.
- [4] Kruse, R. L. and Ryba, A. J., *Data Structures and Program Design in C++*, Prentice Hall, 1999.
- [5] Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *Data Structures and Algorithms*, Addison – Wesley, 1987.