110CSC403

Algorithms: Analysis and Applications

Ivor Spence

Course Administration

Lecturer

Dr. I. Spence, Room 2.19, Crossland Building.

Timetable

Three lectures per week

- Tue 14:00 Ashby 6.10
- Wed 12:00 Ashby 6.10
- Fri 10:00 Ashby 3.12

One tutorial per week, time to be arranged.

Book list

Books are available in the Science Library

- Computer Algorithms: Introduction to Design and Analysis (3rd ed.), Baase and Van Gelder, Adison Wesley
  ISBN 0201612445 (QA76.9.A43/BAAS) **
- Fundamentals of Algorithms, Brassard and Bratley, Prentice Hall
  ISBN 0133350681 *
- Algorithmics (2nd ed.), Harel, Addison Wesley
  ISBN 0201504014

Assessment

Assignments 20%

There will be eight assignments which will be weighted equally.

It is important that you do the assignments yourself.

Examination 80%

Aim

That students should know and understand some of the principal algorithms used in Computer Science and be able to design and analyse efficient algorithms to suit particular applications.

Lecture Notes

The details of a particular programming language are not important when analysing algorithms. A Java-like pseudocode will be used when necessary. The notes will be available in pdf (Acrobat) format at http://www.cs.qub.ac.uk/csc403.

Contents

Chapter 1

Motivation

Maximum Subsequence Sum

Given an array

```
int A[];
A = new int[N];
```

find

$$\max \left\{ \sum_{i \le k < j} A[k] \,\middle|\, 0 \le i < N \wedge i \le j \le N \right\}$$

For example,

$$\begin{aligned} N &= 9 \\ A &= [10, -2, -2, 4, 11, -27, 11, 2, -9] \end{aligned}$$

Method 1
Generate all subsequences

```
int max = 0;
int i,j;
for (i=0; i<N; i++)
    for (j=i; j<N; j++)
        Compute sum A[i]..A[j] and compare with max
```

Refinement of Method 1
Don't recalculate every sum from scratch, only start again from scratch when the starting element moves on by one.

```
int max = 0;
```

```
int i,j, sumSoFar;
for (i=0; i<N; i++)
{
    sumSoFar = 0;
    for (j=i; j<N; j++)
        sumSoFar = sumSoFar + A[j];
        // sumSoFar contains ∑_{k=i,j} A[k]
        Compare sumSoFar with max
}
```

Method 2 (linear)
If the total up to a particular point is negative, it will always be better to ignore what has gone before and start from the current point.

```
int i, max=0, sumSoFar=0;
for (i=0; i<N; i++)
{
    if (sumSoFar > 0)
        sumSoFar = sumSoFar + A[i];
    else
        sumSoFar = A[i];
    Compare sumSoFar with max
}
```

1.1 Efficiency

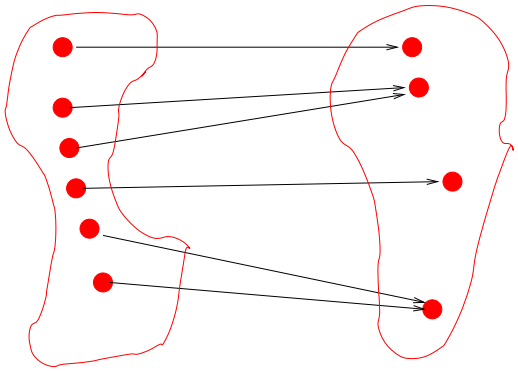
How efficient is an algorithm (e.g. Quicksort)? We can

- Time an implementation of the algorithm
- Analyse the algorithm mathematically

1.2 Comparing Algorithms

There may be many algorithms which solve a particular problem. How do we compare algorithms?

An algorithm is a representation of a function which maps (transforms) input values to output values.



INPUT OUTPUT

We consider an input to have certain size :-

INPUT	SIZE
a list, `l`	the length of `l`
an integer, `i`	the number of bits used to represent `i`
a tree, `t`	the number of nodes in `t`

For a given algorthm we define its SPACE COMPLEXITY and TIME COMPLEXITY.

Space Complexity The number (and size) of variables required by the algorithm *as a function of the input size*.

Time Complexity The number of elementary actions performed by the algorithm *as a function of input size*.

For example, an algorithm will probably require more time to process a long list than a short one.

Example

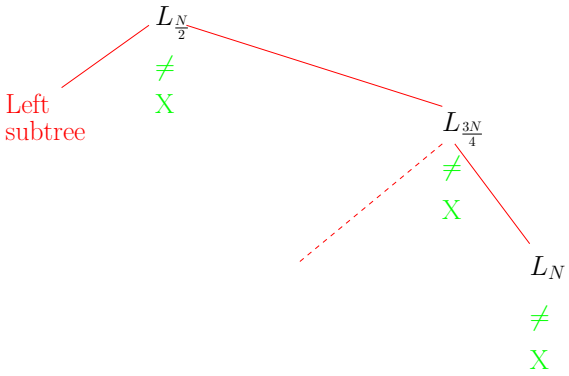
Suppose we search an *ordered* list L for an element X. If the list contains N elements, how many comparisons of X against an element of the list are needed in the worst possible case?

Linear Search

$L_1 \quad L_2 \quad \ldots \quad L_N$

N comparisons are required - that is, the number of comparisons is a function of (depends on) the input size N.

Binary Search



Approximately $\lg N$ (that is, log to the base 2 - see section 2.2.2) comparisons are needed for the worst case.

N	$\lg N$
10	3
100	6
1000	9
10^6	19

We can also carry out average case analysis.

1.3 Correctness

An algorithm to solve a problem should compute the correct output from a given input.

Example

Suppose that coins are available in the following denominations:
£1, 50p, 20p, 10p, 5p, 2p, 1p

Devise an alogorithm for paying out a given amount to a customer using the smallest number of coins.

Proposed Algorithm (Greedy)

At every stage, add to the coins already chosen a coin of the largest denomination available that does not exceed the amount left to paid.

For example, if the amount to be paid is £2.67 the algorithm produces the following:

Denomination	Quantity	Still to be paid
£1	2	£0.67
50p	1	£0.17
10p	1	£0.07
5p	1	£0.02
2p	1	£0.00

The total number of coins is 6.

Is this algorthms always correct?

Does it work for other coin sets?

Consider a country with the coins:

51ø, 50ø, 8ø, 1ø.

Suppose the amount 58ø is to be paid.

Greedy Algorithm Solution

51ø, 7 x 1ø.

Required solution

50ø, 8ø.

So, the greedy algorithm is not always correct.

> People who analyze algorithms have double happiness.
>
> First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures.
>
> Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.
>
> D.E. Knuth.

Chapter 2

Mathematical Background

2.1 Numbers

Natural numbers	$\mathbb{N}$	$= \{0, 1, 2, \ldots\}$
Positive naturals	$\mathbb{N}^+$	$= \{1, 2, \ldots\}$
Integers	$\mathbb{Z}$	$= \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
Real numbers	$\mathbb{R}$	
Positive reals	$\mathbb{R}^+$	
Non-negative reals	$\mathbb{R}^*$	$= \mathbb{R}^+ \cup \{0\}$

2.2 Functions

2.2.1 Floor and Ceiling

Floor $\quad \lfloor\ \rfloor : \mathbb{R} \to \mathbb{Z}$

Definition $\quad \lfloor x \rfloor = n \Leftrightarrow n \leq x < n + 1$
e.g. $\quad \lfloor 2.9 \rfloor = 2$
$\lfloor -2.9 \rfloor = -3$

Ceiling $\quad \lceil\ \rceil : \mathbb{R} \to \mathbb{Z}$

Definition $\quad \lceil x \rceil = n \Leftrightarrow n - 1 < x \leq n$
e.g. $\quad \lceil 2.9 \rceil = 3$
$\lceil -2.9 \rceil = -2$

2.2.2 Logarithms

$$\log : \underbrace{\mathbb{R}^+}_{\substack{base \\ >1}} \times \underbrace{\mathbb{R}^+}_{\substack{argument \\ >0}} \to \mathbb{R}$$

Definition:

$$\log_b x = y \Leftrightarrow b^y = x$$
$$(b > 1, x > 0)$$

e.g. $\log_2 4 = 2$
$\log_2 x = \lg x$
$\lg 8 = 3$
$\lg e = \lg 2.71829\ldots \approx 1.44$
$\log_e x = \ln x$
$\ln 2 \approx 0.7$

Properties

1. $x_1 > x_2 \Leftrightarrow \log_b x_1 > \log_b x_2$
   (log is a strict monotonically increasing function)
2. $\log_b x_1 = \log_b x_2 \Leftrightarrow x_1 = x_2$
3. $\log_b 1 = 0$
4. $\log_b(b^a) = a$
5. $\log_b(x_1 * x_2) = \log_b(x_1) + \log_b(x_2)$
6. $\log_b(x^a) = a \log_b x$
7. $x_1^{\log_b x_2} = x_2^{\log_b x_1}$
8. $\log_{b_1} x = \frac{\log_{b_2} x}{\log_{b_2} b_1}$

Example proof

Show that $\log_b(x^a) = a \log_b x$

Suppose that $\log_b(x^a) = y$

Then, by definition,

$$\begin{aligned} & b^y = x^a \\ \therefore \quad & b^{\frac{y}{a}} = x \\ \therefore \quad & \tfrac{y}{a} = \log_b x \\ \therefore \quad & y = a \log_b x \\ Q.E.D. & \end{aligned}$$

More properties

Let $n \in \mathbb{N}^+$

Case (i)	$n = 2^k$ for some $k \in \mathbb{Z}$
	$\log_2 n = \lg n = k$
Case (ii)	(n is not a power of 2)
	$2^k < n < 2^{k+1}$ for some $k \in \mathbb{Z}$
then	$k < \lg n < k + 1$
and	$\lfloor \lg n \rfloor = k$
	$\lceil \lg n \rceil = k + 1$
Verify	$n \leq 2^{\lceil \lg n \rceil} < 2n$
and	$\frac{n}{2} < 2^{\lfloor \lg n \rfloor} \leq n$

2.3 Probability

Suppose that in a given situation an event or experiment may have one of k outcomes, $S_1, S_2, \ldots, S_k$. With each S_i we associate a real number $p(S_i)$, the probability of S_i, such that

1. $0 \leq p(S_i) \leq 1, 1 \leq i \leq k$
2. $p(S_1) + p(S_2) + \cdots + p(S_k) = 1$

For example, with a fair 6-sided die

$$p(S_i) = 1/6, 1 \leq i \leq 6$$

If $\mathbf{S}$ is a subset of $\{S_1, S_2, \ldots, S_k\}$ then $\mathbf{S}$ represents any one of several outcomes,

or any outcome with a specified property, and

$$p(\mathbf{S}) = \sum_{S_i \in \mathbf{S}} p(S_i)$$

For example to find the probability that when a die is thrown the number appearing is divisible by 3,

$$\begin{aligned} \mathbf{S} &= \{S_3, S_6\} \\ p(\mathbf{S}) &= p(S_3) + p(S_6) = 1/6 + 1/6 = 1/3 \end{aligned}$$

2.3.1 Expected Value

Suppose that in a given situation an event or experiment may have one of k numeric outcomes, $x_1, x_2, \ldots, x_k$ where each x_i occurs with probabilty $p(i)$. The expected value ("average value") of the outcome is

$$\sum_{i=1}^{i=k} p(i) * x_i$$

For example, the expected value from rolling a fair die is

$\frac{1}{6} * 1 + \frac{1}{6} * 2 + \cdots + \frac{1}{6} * 6 = \frac{21}{6} = 3\frac{1}{2}$

If however the die is loaded so that the probability of rolling 6 is $\frac{1}{2}$ and the probability of rolling any other number is $\frac{1}{10}$ then the expected value is

$\frac{1}{10} * 1 + \frac{1}{10} * 2 + \cdots + \frac{1}{10} * 5 + \frac{1}{2} * 6 = \frac{15}{10} + 3 = 4\frac{1}{2}$

2.4 Summation

Example

Consider

```
for (i=1; i<=N; i++)
    for (j=1; j<=i; j++)
        s();
```

How many times is s() executed?

$$\boxed{1 + 2 + \cdots + N = \sum_{i=1..N} i = \sum_{i=1}^{N} i}$$

Summation formulae

1. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
2. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
3. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
4. $\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$
5. $\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$

2.5 Induction

Let $P(n)$ be a proposition (assertion) about n, for every $n \in \mathbb{N}$.

If $P(0)$ (Base case)

and $P(n) \Rightarrow P(n+1)$ (Inductive case)

then $P(0)$ is true, and $P(1)$ and $P(2)\ldots$, in fact

$$\forall k \in \mathbb{N}.P(k)$$

Alternatively, if the base case is $P(1)$ we get

$$\forall k \in \mathbb{N}^+.P(k)$$

Example

Prove that $\underbrace{\sum_{i=1}^{n} i = \frac{n(n+1)}{2}}_{P(n)}$ is true for every value of $n \in \mathbb{N}$.

We need to show the base case and the inductive case.

Base case

$$\begin{aligned} P(1) &= \left(\sum_{i=1}^{1} i = \frac{1(1+1)}{2}\right) \\ &= (1 = 1) \\ &= true \end{aligned}$$

Inductive case

We now assume that $P(n)$ is true and wish to prove $P(n+1)$ is true, where

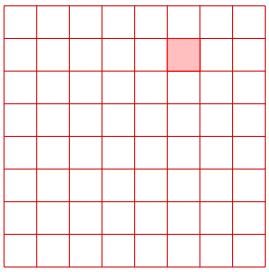
$$P(n+1) = \left(\sum_{i=1}^{n+1} i = \frac{(n+1)(n+1+1)}{2}\right)$$

$$\begin{aligned} \text{But } \sum_{i=1}^{n+1} i &= \sum_{i=1}^{n} i + (n+1) \\ &= \tfrac{n(n+1)}{2} + (n+1) \text{ (since we assume } P(n)) \\ &= \tfrac{n(n+1)+2(n+1)}{2} \\ &= \tfrac{(n+2)(n+1)}{2} \\ &= \tfrac{(n+1)(n+1+1)}{2} \end{aligned}$$

So, $P(n) \Rightarrow P(n+1)$.

Example

Consider a board divided into equal squares, where one arbitrary square is distinguished.



Consider a set of tiles, where each tile has shape

that is, a 2×2 square with one corner missing.

Can the board be covered with such tiles so that each square is covered exactly once, except for the distinguished square which remains uncovered?

Theorem

A board of size $2^k \times 2^k$ squares, where $k > 0$, can always be tiled.

Proof By Induction

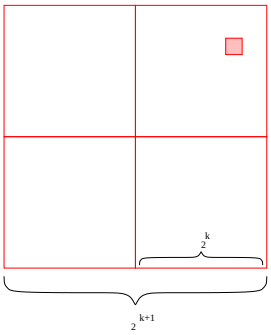
Base Case P(0)

Suppose $k = 0$, so since $2^0 = 1$ the board is of size 1×1. There is only one square, so it must be the distinguished square and the board is already tiled.

Inductive Case P(k+1)

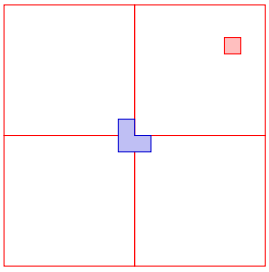
Assume that P(k) is true, i.e. that any board of size 2^k can be tiled.

Consider a board of size 2^{k+1}. Since $2^{k+1} = 2 * 2^k$ this board can be divided into four squares of size 2^k.



Place one tile so that it covers a square of each of the sub-boards not containing

the distinguished square.



Each sub-board now has one "distinguished square" and by induction each sub-board can be tiled, so the whole board can be tiled and P(k+1) holds.

Conclusions

We may reason about data structures such as

- Natural numbers
- Lists
- Arrays
- Trees
- etc.

using induction.

2.6 Permutations and Combinations

Permutation: How many different ways can a tuple of size j be selected from k elements?

Example: $\{1, 2, \ldots, k\}$ - a set of k elements.

How many 3-tuples?

$(1, 2, 3)$
$(1, 3, 2)$ - (order of elements is significant)
.
.

$(1, 2, 4)$
.
etc.

How many choices of first element are possible?

$(\_\_, \_\_, \ldots, \_\_)$

How many choices of the second element are possible?

(first, $\_\_, \ldots, \_\_$)

The total number of possible permutations of j elements from k:

$$\begin{aligned} &= k * (k-1) * \cdots * (k-(j-1)) \\ &= \frac{k * (k-1) * \cdots * (k-(j-1)) * (k-j) * \cdots * 2 * 1}{(k-j) * (k-j-1) * \cdots * 2 * 1} \\ &= \frac{k!}{(k-j)!} \end{aligned} \tag{2.1}$$

(Note that $0! = 1$)

Combination: How many different ways can a set of size j be selected from k elements, i.e. where the order of the elements chosen doesn't matter? That is,

$\{1, 2, 3\}$ and
$\{1, 3, 2\}$ are not counted as being different.

Consider again the result for permututations, equation 2.1. We can use this to calculate the number of combinations if we divide by the the number of ways in which the j elements of a set can be arranged into different tuples.

Permuations	Combinations
$(1, 2, 3)$	$\{1, 2, 3\}$
$(1, 3, 2)$	
$(2, 1, 3)$	
$(2, 3, 1)$	
$(3, 1, 2)$	
$(3, 2, 1)$	
$(1, 2, 4)$	$\{1, 2, 4\}$
$(1, 4, 2)$	
etc.	

To see how many permutations there are for each combination, note that the first position can be chosen from j elements, the second from $(j-1)$ elements etc., giving a total of $j!$ permutations from each combination.

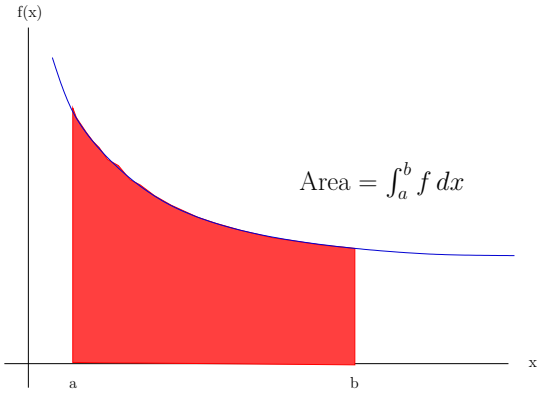
Thus there are

$$\frac{k!}{(k-j)!j!} = \left(\begin{array}{c} k \\ j \end{array} \right) \tag{2.2}$$

combinations of j elements from k.

2.7 Integration

The area between the x-axis and the graph of a function $y = f(x)$ can be calculated using the integral of f, written $\int f(x)\,dx$. The area under the graph as x increases from a to be is calculated by subtracting the value of the integral at a from that at b and is written $\int_a^b f(x)\,dx$



2.7.1 Rules for Integration

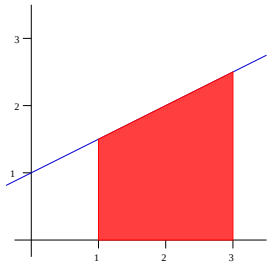
$f(x)$	$\int f(x)\,dx$
x^n	$\frac{x^{n+1}}{n+1}$ if n is not -1
x^{-1} (i.e. $\frac{1}{x}$)	$\ln(x)$
$c * f(x)$ where c is constant	$c * \int f(x)\,dx$
$f(x) + g(x)$	$\int f(x)\,dx + \int g(x)\,dx$

2.7.2 Example

Find the area under the graph of

$$y = 1 + \frac{x}{2}$$

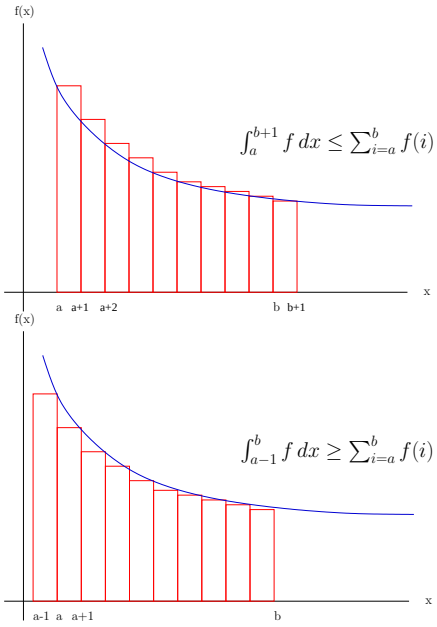
from $x = 1$ to $x = 3$



This area is

$$\begin{aligned}
& \int_1^3 (1 + \frac{x}{2})\, dx \\
= & \int_1^3 1\, dx + \int_1^3 \frac{x}{2}\, dx \\
= & \int_1^3 x^0\, dx + \frac{1}{2} \int_1^3 x^1\, dx \\
= & \left[x + \frac{x^2}{4} \right]_1^3 \\
= & [3 + 2\frac{1}{4}] - [1 + \frac{1}{4}] \\
= & \; 5\frac{1}{4} - 1\frac{1}{4} = 4
\end{aligned}$$

2.8 Summations using Integration



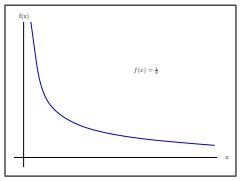
If $f(x)$ is continuous and non-increasing

$$\int_{a-1}^{b} f(x)\,dx \ge \sum_{i=a}^{b} f(i) \ge \int_{a}^{b+1} f(x)\,dx \tag{2.3}$$

If $f(x)$ is continuous and non-decreasing

$$\int_{a-1}^{b} f(x)\,dx \le \sum_{i=a}^{b} f(i) \le \int_{a}^{b+1} f(x)\,dx \tag{2.4}$$

Example - estimate $\sum_{i=2}^{n} \frac{1}{i}$



$f(x) = \frac{1}{x}$ is continuous and non-increasing in the range $2..n$ so

$$\sum_{i=2}^{n} \frac{1}{i} \le \int_{1}^{n} \frac{dx}{x} = [\ln x]_1^n = \ln n - \ln 1 = \ln n$$

$$\sum_{i=2}^{n} \frac{1}{i} \ge \int_{2}^{n+1} \frac{dx}{x} = [\ln x]_2^{n+1} = \ln n + 1 - \ln 2$$

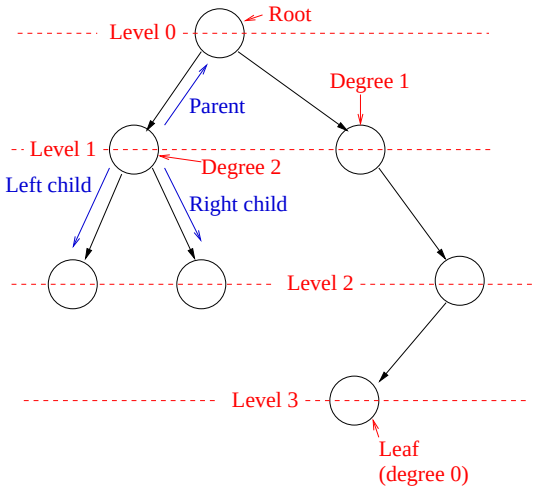
$$\sum_{i=2}^{n} \frac{1}{i} \approx \ln n$$

2.9 Data Structures

Familiarize yourself with

- Object References (Pointers)
- Linked Lists
- Stacks
- Queues

Binary Trees



The depth (= height) is the maximum of the levels of the tree's leaves.

1. There are at most 2^l nodes at level l of a binary tree.
2. A binary tree with depth d has at most $2^{d+1} - 1$ nodes.
3. A binary tree with n nodes has depth at least $\lfloor \lg n \rfloor$.

Chapter 3

Analyzing Algorithms and Problems

3.1 Amount of Work (Complexity Measure)

The measure we choose should help in comparing two algorithms for the same problem and determining which is the more efficient.

Possibilities

1. Execution time
   - Machine dependent
   - Need to write program
2. Count instructions executed
   - Language dependent
   - Dependent on programmer's skill
   - Need to write program

   We need a measure of work which is independent of:
   - Computer
   - Programming language
   - "Bookkeeping operations" e.g. incrementing loop indices, initialisation etc.
   - Programmer

3. Count loop iterations

Choose a basic operation, fundamental to the problem under study, and ignore bookkeeping etc.

Problem	Basic Operation
Find x in a list of names	Comparison of x with an entry in the list
Multiply two real matrices	Multiplication of two real numbers
Sort a list of numbers	Comparison of two list entries

Provided the basic operation is well chosen and the total number of operations is proportional to the number of basic operations we will have

- a good measure of work
- a good criterion for comparing several algorithms

We usually compare members of a class of algorithms to solve a particular problem - these should all use the same basic operation.

Complexity $\equiv$ amount of work $\equiv$ number of basic operations

3.2 Average and Worst Case Analysis

The results of the analysis are expressed in terms of input size and so we need a measure of this. The measure should be proportional to the amount of memory required to represent it.

Problem	Size of input
Find x in a list of names	The number of names in the list
Multiply two real matrices	The size of the matrices
Traverse a binary tree	The number of nodes in the tree
Is N a prime number	The number of bits in N

3.2.1 Worst Case $W(N)$

D_N – set of all inputs of size N $I \in D_N$

$t(I)$ – number of basic operations performed by algorithm given input I

$W(N) = \max \{ t(I) \mid I \in D_N \}$

3.2.2 Average Case $A(N)$

If $p(I)$ is the probability that input I occurs, the average ("expected") complexity (see section 2.3.1) $A(N)$ is

$$A(N) = \sum_{I \in D_N} p(I)t(I)$$

3.2.3 Example - Sequential Search

Problem

Let L be an array containing N entries. Given an entry X return an index at which X can be found in L if possible, return -1 if X is not in L.

Algorithm

Input values are `L,N,X` and output value is `index`.

```
int index = 0;
// ∀i : 0 ≤ i < index : L[i] ≠ X
while (index < N && L[index] != X)
    index++;
if (index == N)
    index = -1;
```

Basic Operation

Comparison of X with an entry in the list.

Worst Case Analysis

To discover that X is not in L it must be compared with each element of L, so $W(N) = N$.

Average Case Analysis

Assume that

- The probability that X is on the list is q.
- X is equally likely to occur in any position.
- The list entries are distinct.

There are (N+1) possible inputs

$$\begin{cases} I_i & 1 \le i \le N \quad \text{(X in position i-1)} \\ I_{N+1} & \quad\quad\quad\quad\quad\quad \text{(X not in the list)} \end{cases}$$

Then

$$\begin{aligned} p(I_i) &= \tfrac{q}{N} && 1 \le i \le N \\ p(I_{N+1}) &= 1 - q \\ t(I_i) &= i && 1 \le i \le N \\ t(I_{N+1}) &= N \end{aligned}$$

$$\begin{aligned} A(N) &= \sum_{i=1}^{N+1} p(I_i)t(I_i) \\ &= \sum_{i=1}^{N} \frac{q}{N} i + (1-q)N \\ &= q\tfrac{(N+1)}{2} + (1-q)N \end{aligned}$$

Note
If $I \in D_N$ I may be thought as a member of the set of all lists and values for X, where X may occur at some position in the list.

3.3 Complexity of problems

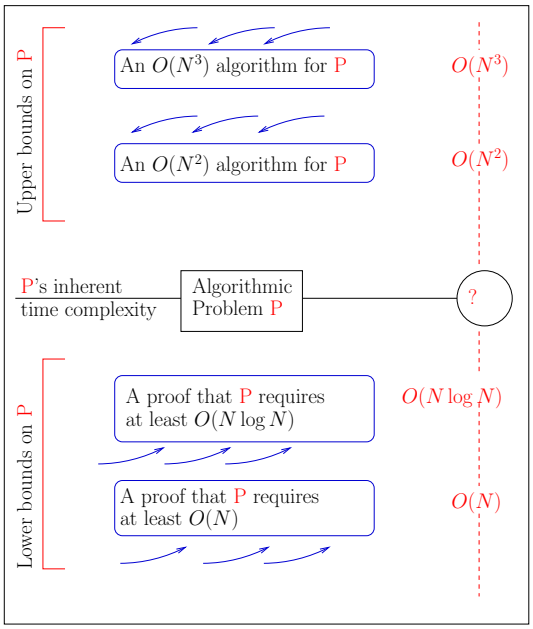
Given a problem, we choose a class of algorithms by specifying the type of operation allowed and a measure of complexity. Then

1. Devise a seemingly efficient algorithm A and work out its worst case complexity W(A).
2. For some function F (a lower bound) prove a theorem stating that for any algorithm in the class under consideration, there is some input of size N for which the algorithm must perform at least F(N) step.

If W $=$ F the algorithm is optimal.
If W $\neq$ F there may be a better algorithm or a better (bigger) lower bound.

3.4 Upper and Lower Bounds

Note that these are properties of a problem rather than of a particular algorithm.



In this case, P's complexity is no better than $O(N \log N)$ and no worse than $O(N^2)$.

> **Note**
> Calculation of a lower bound requires consideration of all algorithms, even those not yet discovered!.

3.4.1 Example - Largest List Entry

Problem

Find the largest entry in a list of numbers.

Class of algorithms

Algorithms that can compare and copy list entries, but no other list operations.

Basic operation

Comparison of two list entries.

Algorithm

Input - L (an array of numbers);
N ≥ 1, the number of entries;

Ouput - max, the largest entry in L;

```
max = L[0];
for (index=1; index<N; index++)
// max = max{L[0]...L[index − 1]}
    if (max < L[index])
        max = L[index];
```

Worst Case Analysis

Clearly $W(N) = N - 1$.

Lower Bound for Problem

We claim that $F(N) \geq N - 1$, i.e. that at least $N - 1$ comparisons would be needed by any algorithm in this class.

Proof by contradiction:

Suppose $F(N) \leq N - 2$, so that at most $N - 2$ comparisons have been made.

$\therefore$ there are at least two entries which have not been found to be less than any other entry.

Suppose that the algorithm selects one of these as max (or indeed any other value) - the correct answer could be the other one.

Contradiction

Using one $N - 2$ comparisons cannot provide enough information to make a correct choice of max.

Chapter 4

Classifying Functions by Growth Rate

4.1 Introduction

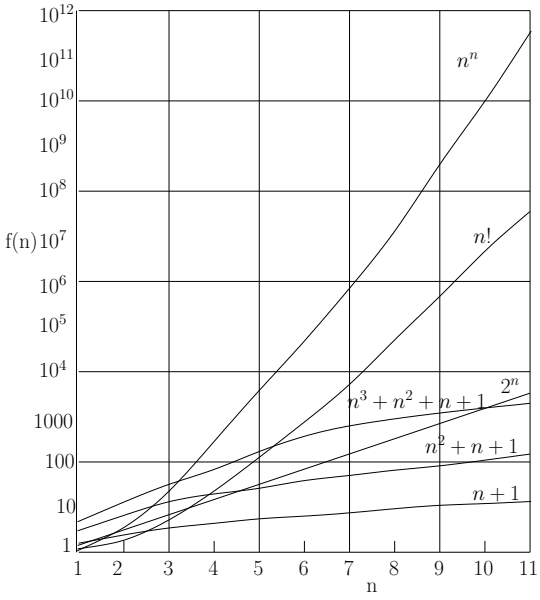
Consider the actual execution time of a number of algorithms, remembering that the absolute time taken to complete a basic operation may vary because of, for example, housekeeping operations.

Algorithm	No. of Basic Operations	Total No. of Ops
$A1$	$2N$	$2CN$
$A2$	$4.5N$	$4.5C'N$
$A3$	$\frac{N^3}{2}$	$\frac{CN^3}{2}$
$A4$	$5N^2$	$5C'N^2$

Is $A1$ better than $A2$? Is $A3$ better than $A4$?

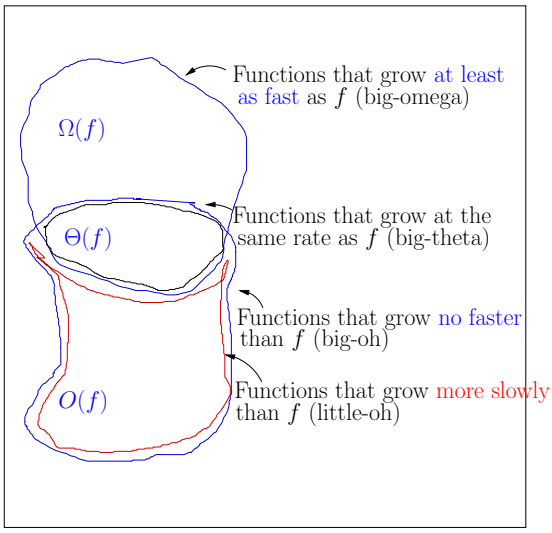
We want to classify functions ignoring constant factors and what happens with small values of N.

We will consider only their Asymptotic Growth Rate or Order.



4.2 Definitions of $O(f)$, $\Omega(f)$ and $\Theta(f)$

In the following definitions 4.2.1, 4.2.4, 4.2.6 and 4.2.8 let $f : N \to \mathbb{R}^*$.



Order f, written $O(f)$ consists of those functions which "grow no faster than f".

Definition 4.2.1

$$O(f) \triangleq \{g : \mathbb{N} \to \mathbb{R}^* \mid \exists c \in \mathbb{R}^+ . \exists N_0 \in \mathbb{N}. \forall n \geq N_0 . g(n) \leq c * f(n)\}$$

Note:

Even if $\forall n \in \mathbb{N}. g(n) > f(n)$, g may still be in $O(f)$.

E.g. $2 * f \in O(f)$.

If $g \in O(f)$ then f is an upper bound on the behaviour of g, i.e. g is no worse than f.

Definition 4.2.2 (Alternative to 4.2.1)

$$g \in O(f) \Leftrightarrow \lim_{N \to \infty} \frac{g(N)}{f(N)} = c$$

where $c \in \mathbb{R}^*$.

Note:

$0 \in \mathbb{R}^*$

Theorem 4.2.3 (L'Hôpital's Rule)

If

$$\lim_{N \to \infty} g(N) = \infty$$

and

$$\lim_{N \to \infty} f(N) = \infty$$

then

$$\lim_{N \to \infty} \frac{g(N)}{f(N)} = \lim_{N \to \infty} \frac{g'(N)}{f'(N)}$$

where $g'(N)$ *is the derivative of* g.

Differentiation

The gradient of the graph of a function $y = f(x)$ can be calculated using the derivative of f, written $\frac{df}{dx}$ or simply as f'. The process of finding the derivative is essentially the inverse of integration.

Rules for Differentiation	
$f(x)$	$\frac{df}{dx}$
x^n	$n * x^{n-1}$
e^{cx} where c is constant	$c * e^{cx}$
$c * f(x)$ where c is constant	$c * \frac{df}{dx}$
$f(x) + g(x)$	$\frac{df}{dx} + \frac{dg}{dx}$
$\ln x$	$\frac{1}{x}$

4.2.1 Example

Let $f(N) = \frac{N^3}{2}$ and $g(N) = 37N^2 + 120N + 17$. Show that $g \in O(f)$ and $f \notin O(g)$.

1. $g \in O(f)$

$$\begin{aligned} \lim_{N\to\infty} \frac{37N^2 + 120N + 17}{\frac{N^3}{2}} &= \lim_{N\to\infty} \frac{74}{N} + \frac{240}{N^2} + \frac{34}{N^3} \\ &= 0 (\in \mathbb{R}^*) \end{aligned}$$

   Alternative: Prove by induction that $\forall N \geq 78. g(N) < f(N)$.

2. $f \notin O(g)$

$$\begin{aligned} \lim_{N\to\infty} \frac{f}{g} &= \text{(L'Hôpital) } \lim_{N\to\infty} \frac{\frac{3}{2}N^2}{74N + 120} \\ &= \text{(L'Hôpital) } \lim_{N\to\infty} \frac{3N}{74} \\ &= \infty (\notin \mathbb{R}^*) \end{aligned}$$

Remembering that here $f : N \to \mathbb{R}^*$, $\Omega(f)$ consists of those functions which "grow at least as fast as f".

Definition 4.2.4

$$\Omega(f) \triangleq \{g : \mathbb{N} \to \mathbb{R}^* \mid \exists c \in \mathbb{R}^+. \exists N_0 \in \mathbb{N}. \forall n \geq N_0. g(n) \geq c * f(n)\}$$

Note:

If $g \in \Omega(f)$ then f acts as a lower bound for g, i.e. g is no better than f.

Definition 4.2.5 (Alternative to 4.2.4)

$g \in \Omega(f) \Leftrightarrow$

$$\lim_{N\to\infty} \frac{g(N)}{f(N)} = \infty \quad \textit{or}$$

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

where $c \in \mathbb{R}^+$.

Now we intersect $\Omega(f)$, the set of functions we grow "at least as quickly" as f, with $O(f)$, those which grow "no more quickly than f", to get $\Theta(f)$, i.e. those functions which grow "at the same rate" as f.

Definition 4.2.6

$$\Theta(f) = O(f) \cap \Omega(f)$$

Definition 4.2.7 (Alternative to 4.2.6)

$$g \in \Theta(f) \Leftrightarrow \lim_{N\to\infty} \frac{g(N)}{f(N} = c$$

where $c \in \mathbb{R}^+$.

Finally we consider $o(f)$, those function which definitely grow more slowly than f.

Definition 4.2.8

$$o(f) = O(f) - \Theta(f)$$

Definition 4.2.9 (Alternative to 4.2.8)

$$g \in o(f) \Leftrightarrow \lim_{N\to\infty} \frac{g(N)}{f(N)} = 0$$

4.3 Examples

4.3.1 Logs and powers

Show that

$$\lg n \in O(n^\alpha) \forall \alpha > 0$$

Consider

$$\lim_{n\to\infty} \frac{\lg n}{n^\alpha} = \lim_{n\to\infty} \frac{\frac{\lg e}{n}}{\alpha n^{\alpha-1}}$$

by L'Hôpital's Rule

$$= \lim_{n\to\infty} \frac{\lg e}{\alpha n^\alpha}$$

$= 0$ for any $\alpha > 0$.

4.3.2 Exponents

Show that

$$2^n \in o(3^n)$$

This follows because

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$$

Now show that

$$n^\alpha \in o(2^n), \forall \alpha \in \mathbb{N}^+$$

This follows because

$2^n = e^{n \ln 2}$ (check by taking $\ln$ of both sides)

so

$$\lim_{n\to\infty} \frac{n^\alpha}{2^n} = \lim_{n\to\infty} \frac{n^\alpha}{e^{n \ln 2}}$$

$$= \lim_{n\to\infty} \frac{\alpha n^{\alpha-1}}{\ln 2 * e^{n \ln 2}} \quad \text{(L'Hôpital)}$$

$$= \lim_{n\to\infty} \frac{\alpha!}{(\ln 2)^\alpha * e^{n \ln 2}} \quad \text{(L'Hôpital } \alpha \text{ times)}$$

$$= 0$$

4.4 Properties of O, Θ and Ω

Assume that $f, g, h : \mathbb{N} \to \mathbb{R}^*$.

1. Reflexivity

   O, Θ and Ω are reflexive, i.e. $f \in O(f)$, $f \in \Theta(f)$, $f \in \Omega(f)$.

2. Transitivity

   O, Θ and Ω are transitive, e.g. if $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.

   Proof:

   $f(n) \leq c_1 g(n) \quad \forall n \geq N_1$ (defn of $f \in O(g)$)

   $g(n) \leq c_2 h(n) \quad \forall n \geq N_2$ (defn of $g \in O(h)$)

   $\therefore$

   $f(n) \leq c_3 h(n) \quad \forall n \geq \max\{N_1, N_2\}$
   $c_3 = c_1 * c_2$

   This also holds for Θ, Ω and o.

3. $f \in O(g) \Leftrightarrow g \in \Omega(f)$

4. Symmetry

   Θ is symmetric, i.e. $f \in \Theta(g) \Rightarrow g \in \Theta(f)$

5. Equivalence

   Since Θ is reflexive, symmetric and transitive each set $\Theta(f)$ is an equivalence class (called a complexity class).

6. Rule of Sums

   $O(f + g) = O(\max\{f, g\})$.

   This also holds for Θ and Ω.

7. Scaling

   If $f \in O(g)$ then for any $k > 0$, $f \in O(k * g)$.

   Example

   Show that $\frac{n^3}{6} + n^2 + \lg n + 2 \in O(n^3)$

$$\begin{aligned}
\tfrac{n^3}{6} + n^2 + \lg n + 2 \quad & \in O(\tfrac{n^3}{6} + n^2 + \lg n + 2) \\
& \text{(O is reflexive)} \\
& = O(\max\{\tfrac{n^3}{6}, n^2, \lg n, 2\}) \\
& \text{(rule of sums)} \\
& = O(\tfrac{n^3}{6}) \\
& = O(n^3) \\
& \text{(rule of scaling)}
\end{aligned}$$

4.5 Importance of Order

Algorithm	1	2	3	4	
Time in μs	$33n$	$46n \ln n$	$13n^2$	$3.4n^3$	2^n
$n = 10$	.00033s	.0015s	.0013s	.34s	.001s
$= 1000$	.033s	.45s	13s	.94hrs	cent-uries
$= 100000$	3.3s	1.3min	1.5days	108years	!
Max input size 1 sec	30000	2000	280	67	20
1 min	1800000	82000	2200	260	26

4.5.1 Tractable and Intractable Problems

Suppose that $n = 50$ is the size of input and consider the following numbers of basic operations.

Tractable:

n	50
$5n$	250
$n \lg n$	282
n^2	2500
n^3	125000

Polynomial (bounded by n^k for fixed k)

Intractable:

2^n	16 digits
$n!$	65 digits
n^n	85 digits

Exponential (bounded by k^n, fixed $k > 1$)

Super-exponential

Number of microseconds since the "big bang" has 24 digits.

4.5.2 NP-Complete Problems

This is a class of problems for which the best known lower bound is $O(N)$ and the best known upper bound is $O(\text{exponential})$. We don't know whether these problem lie on the tractable or intractable side, but they all lie on the same side.

There are > 1000 diverse such problems from combinatorics, operations research, economics, graph theory, game theory, logic, ...

Chapter 5

Searching

We assume that data records contain a key field (having a key value).

General information fields
KEY
More general information fields

The data records may be ordered (sorted) using the ordering of the corresponding key values.

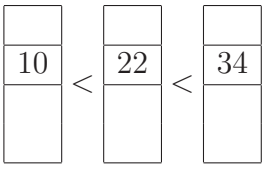
Searching an unordered collection data (of size N) takes time:

Worst Case (Linear Search) $= O(N)$
Average Case (Linear Search) $= O(N)$

To achieve fast searching methods it is necessary to sort the data. For the sake of simplicity, we will often

1. assume that the key of type natural number, e.g.

2. ignore all fields of a record except the key field



Records sorted on the key field

5.1 Binary Search

5.1.1 Algorithm

INPUT: L,N ≥ 0 and X where L is an ordered array with N entries

OUTPUT: index

L[index] == X if X is in L
index == -1 if X is not in L

PSEUDOCODE

```
first = 0; last = N-1; found = false;
while (first <= last && !found)
{
    index = ⌊(first+last)/2⌋;
    if (X == L[index])
        found = true;
    else if (X < L[index])
        last = index - 1;
    else
        first = index + 1;
}
if (!found)
    index = -1;
```

5.1.2 Worst Case Analysis

Basic operation - Comparison of X to a list entry.

We assume one comparison with a three-way branch for tests on X

$W(N) =$ number of comparisons.

	L_0	L_1	$\ldots$	$L_{\lfloor \frac{N-1}{2} \rfloor}$	$\ldots$	L_{N-1}
	X_0	X_1	$\ldots$	$X_{\lfloor \frac{N-1}{2} \rfloor}$	$\ldots$	X_{N-1}
N even	$\longleftarrow \frac{N}{2} - 1 \longrightarrow$				$\longleftarrow \frac{N}{2} \longrightarrow$	
N odd	$\longleftarrow \frac{N-1}{2} \longrightarrow$				$\longleftarrow \frac{N-1}{2} \longrightarrow$	

The list to be considered on the second pass has at most $\lfloor \frac{N}{2} \rfloor$ entries.

$$W(N) = 1 + W(\lfloor \tfrac{N}{2} \rfloor) \quad \text{Recurrence relation}$$

$$W(0) = 0 \quad \text{Boundary condition}$$

$$W(1) = 1$$

To find a general expression for $W(N)$ we expand as follows:

$$W(N) = 1 + W(\lfloor \tfrac{N}{2} \rfloor)$$

$$W(\lfloor \tfrac{N}{2} \rfloor) = 1 + W(\lfloor \tfrac{N}{2^2} \rfloor) \quad \text{so}$$

$$W(N) = 1 + 1 + W(\lfloor \tfrac{N}{2^2} \rfloor)$$

$$= 1 + 1 + 1 + W(\lfloor \tfrac{N}{2^3} \rfloor) \quad \text{and we guess that}$$

$$W(N) = \lfloor \lg N \rfloor + 1 \quad \text{as long as } N \geq 1$$

Proof by induction

Base Case $(N = 1)$
$W(1) = \lfloor \lg 1 \rfloor + 1 = 1$, but $W(1) = 1$ is already known, as required.

Inductive Case Assume that for $1 \leq K < N$,
$W(K) = \lfloor \lg K \rfloor + 1$.

$$W(N) = 1 + W(\lfloor \tfrac{N}{2} \rfloor) \quad \text{(by recurrence relation)}$$

$$= 1 + \lfloor \lg \lfloor \tfrac{N}{2} \rfloor \rfloor + 1 \quad \text{(by inductive hypothesis)}$$

$$= 2 + \lfloor \lg \lfloor \tfrac{N}{2} \rfloor \rfloor$$

Note that if N is even
$\lfloor \frac{N}{2} \rfloor = \frac{N}{2}$ and $\lg \frac{N}{2} = \lg N - 1$

whereas if N is odd
$\lfloor \frac{N}{2} \rfloor = \frac{N-1}{2}$ and $\lg \frac{N-1}{2} = \lg(N-1) - 1$.

$$W(N) = \begin{cases} 2 + \lfloor \lg N - 1 \rfloor & \text{if } N \text{ even;} \\ 2 + \lfloor \lg(N-1) - 1 \rfloor & \text{if } N \text{ odd;} \end{cases}$$

$$= \begin{cases} 1 + \lfloor \lg N \rfloor & \text{if } N \text{ even;} \\ 1 + \lfloor \lg(N-1) \rfloor & \text{if } N \text{ odd;} \end{cases}$$

If N is odd (in fact as long as N is not a power of 2), $\lfloor \lg N \rfloor = \lfloor \lg(N-1) \rfloor$.

So, in all cases, $W(N) = 1 + \lfloor \lg N \rfloor$, completing the inductive case.

In the worst case, Binary Search is $\Theta(\lg N)$

5.1.3 Average Case Analysis

There are $2N + 1$ distinctly different input possibilities:

I_i	where $1 \le i \le N$	if	`X = L[i-1]`
I_{N+1}			`X < L[0]`
I_{N+i}	$2 \le i \le N$		`L[i-2] < X < L[i-1]`
I_{2N+1}			`L[N-1] < X`

Assume that

1. $p(I_i) = \frac{1}{2N+1}$ $\quad 1 \le i \le 2N+1$
2. $N = 2^k - 1$, where $k \in \mathbb{N}^+$

Then

$$A(N) = \sum_{i=1}^{2N+1} p(I_i) t(I_i)$$

but what is

$$t(I_i) \text{ where } 1 \le i \le 2N+1?$$

For example, consider the number of comparisons needed to find a value where $N = 15$.

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}
4	3	4	2	4	3	4	1	4	3	4	2	4	3	4

If t is the number of comparisons, then $1 \leq t \leq K$, where $K = \lfloor \lg N \rfloor + 1 = W(N)$.

Let S_t be the number of inputs for which the algorithm performs t comparisons.

$$\begin{array}{lll} S_1 & = 1 & \\ S_2 & = 2 & \\ S_3 & = 4 & \\ S_t & = 2^{t-1} & 1 \leq t < K \\ S_K & = 2^{K-1} + (N+1) & \text{(includes the } N+1 \text{ gaps)} \end{array}$$

Thus, because the inputs are assumed to be equally likely, we can group the inputs which lead to the same number of comparisons, giving

$$\begin{aligned} A(N) &= \frac{1}{2N+1} \sum_{t=1}^{K} t S_t \\ &= \frac{1}{2N+1} [\underbrace{\sum_{t=1}^{K} t 2^{t-1}}_{\frac{1}{2}\sum_{t=1}^{K} t2^t} + K(N+1)] \\ &= \tfrac{1}{2N+1}[\tfrac{1}{2}((K-1)2^{K+1}+2) + K(N+1)] \\ &\quad \text{(by Summation Formula 5)} \\ &= \tfrac{1}{2N+1}[(K-1)2^K + 1 + K(N+1)] \end{aligned}$$

but $N = 2^K - 1$ and $K = \lfloor \lg N \rfloor + 1 = W(N)$, so

$$\begin{aligned} A(N) &= \frac{(2K-1)2^K + 1}{2^{K+1} - 1} \\ &\approx \tfrac{2K-1}{2} = (K-1) + \tfrac{1}{2} = \lfloor \lg N \rfloor + \tfrac{1}{2} \end{aligned}$$

For lists with N entries, Binary Search does approximately $\lfloor \lg N \rfloor + \frac{1}{2}$ comparisons on average.

5.1.4 Optimality

To discuss whether Binary Search is optimal we need to specify the class of algorithms under consideration. We assume that any algorithms here

- performs comparisons against list elements
- performs no other list operations

We will exam decision trees for search algorithms in this class.

Definition 5.1.1

A decision tree for an algorithm `A` of input size `N` is a binary tree with nodes labelled $i, 0 \leq i < N$.

1. *The root is labelled with the index of the first entry to which `X` is compared.*
2. *If a node is labelled i then*

   Label on left child *index of next entry to which `X` will be compared if `X < L[i]`*

   Label on right child *index of next entry to which `X` will be compared if `X > L[i]`*

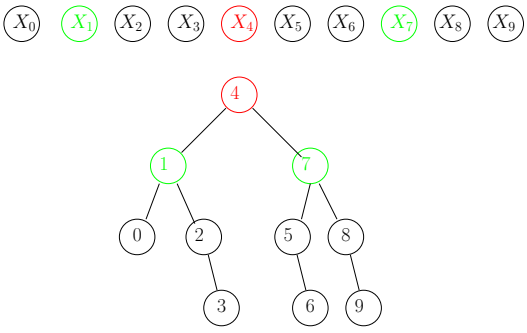


Figure 5.1: Decision Tree for Binary Search

Note that, in general, the number of nodes in a decision tree does not have to equal the size of the input, see for example the decision tree for linear search (Fig. 5.2. Let

d be the depth of the decision tree

M be the number of nodes in the decision tree

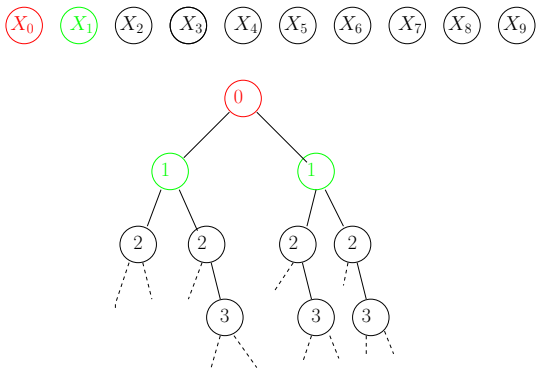


Figure 5.2: Decision Tree for Linear Search

N be the input size

Then $d \geq \lfloor \lg M \rfloor$ (property of binary trees in general) and we claim that $M \geq N$.

Proof: Suppose we have a correct searching algorithm with $M < N$, so some label i is not used $(0 \leq i \leq N-1)$. Consider two inputs `L1` and `L2` which are constructed as follows:

```
L1[j] = L2[j]   != X          0 ≤ j ≤ N − 1, i ≠ j
L1[i] = X       L2[i] != X
```

that is, `L1` and `L2` are only different in position i, and `X` only occurs in `L1[i]`. Since no node in the decision tree is labelled i, the algorithm does not compare `X` against `L[i]`, and so the algorithm is incorrect.

The decision tree must have at least N nodes.

So we now have $d \geq \lfloor lgM \rfloor \geq \lfloor \lg N \rfloor$, and since the maxiumum number of comparisons is $d+1$,

Any algorithm to find `X` in a list of `N` entries (by comparing `X` to list entries) must do at least $\lfloor \lg N \rfloor + 1$ comparisons for some input.

5.2 Dynamic Data Structures

5.2.1 Introduction

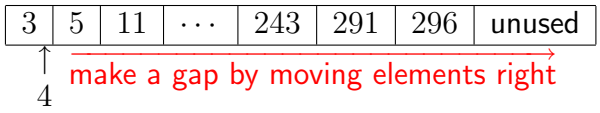
For large databases, it is desirable to minimize the the execution time of the following operations:

- search
- insert
- delete

Arrays are inappropriate as a means of realising large databases because:

1. size is fixed
2. insertion/deletion may be inefficient in a dense (i.e. no gaps) ordered table.

Insertion of `X` into an ordered array `L`.



Worst case: `X < L[0]`:
Insertion is $\Theta(N)$ shifts. This is expensive for what may well be a common operation, so we shall consider other data structures to try to improve on this.

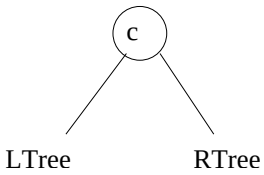
5.2.2 Sorted Binary Tree

Definition 5.2.1

A sorted binary tree is either

- *(empty)*

or of the form



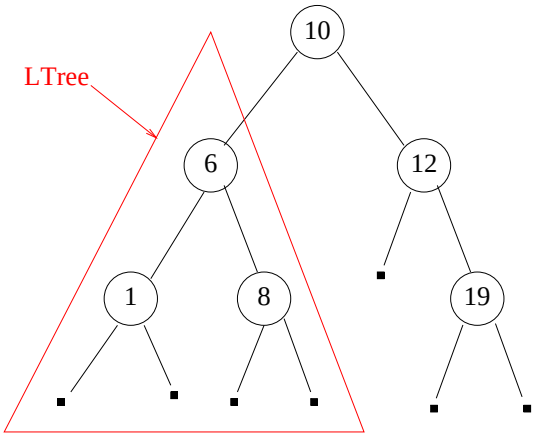
where LTree and RTree are sorted binary trees, and 

is a node containing the value c such that

$$\forall v \in LTree.v \leq c$$

$$\forall v \in RTree.v \geq c$$

Example



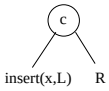
Note that the values in the nodes of LTree are all smaller than the value in the root node. Searching such a tree is similar to binary search.

5.2.3 Binary Tree Insertion

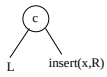
```
insert (x,▪) =
```



```
insert (x,
```

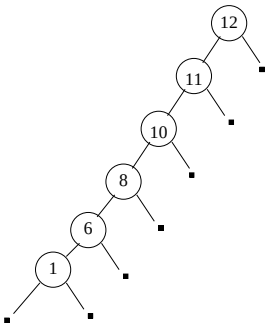

```
) =
    if (x <= c)
```


```
    else
```



Problem

There may be a poor data distribution. Suppose the data arrives in descending order, e.g. 12,11,10,8,6, 1. The tree will be unbalanced.



In the worst case, N elements are stored in the same path within the tree - a search will require $\Theta(N)$ comparisons!

On average, when elements are insterted in random order, such degenerate trees are very rare. The average number of comparisons for successful search is $\approx 1.4 \lg N$.

Balanced Trees

Goal: Devise a "binary tree" structure which balances the heights (or possibly the number of nodes) of the left and right sub-trees.

Various data structures approximate to this goal:

- B-trees
- $2-3$ trees
- $2-3-4$ trees
- AVL trees
- red-black trees

- B^*-trees
- and there are algorithms for balancing unbalanced trees

5.2.4 B-trees

A B-tree of order m is a tree which satisifies:

1. every node has $\leq m$ children;
2. every node, except for the root and leaves, has $\geq \lceil m/2 \rceil$ children;
3. the root has at least 2 children (unless it is a leaf);
4. all leaves are at the same level (and carry no information);
5. a non-leaf node with $j+1$ children contains j data elements (often just called keys);

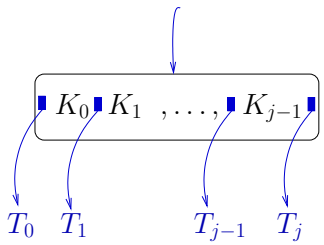
Each node contains a list of key values. If the length of this list is j, so that the node contains j keys

$$\{K_0, K_1, \ldots, K_{j-1}\}$$

then the node has links to $j+1$ subtrees

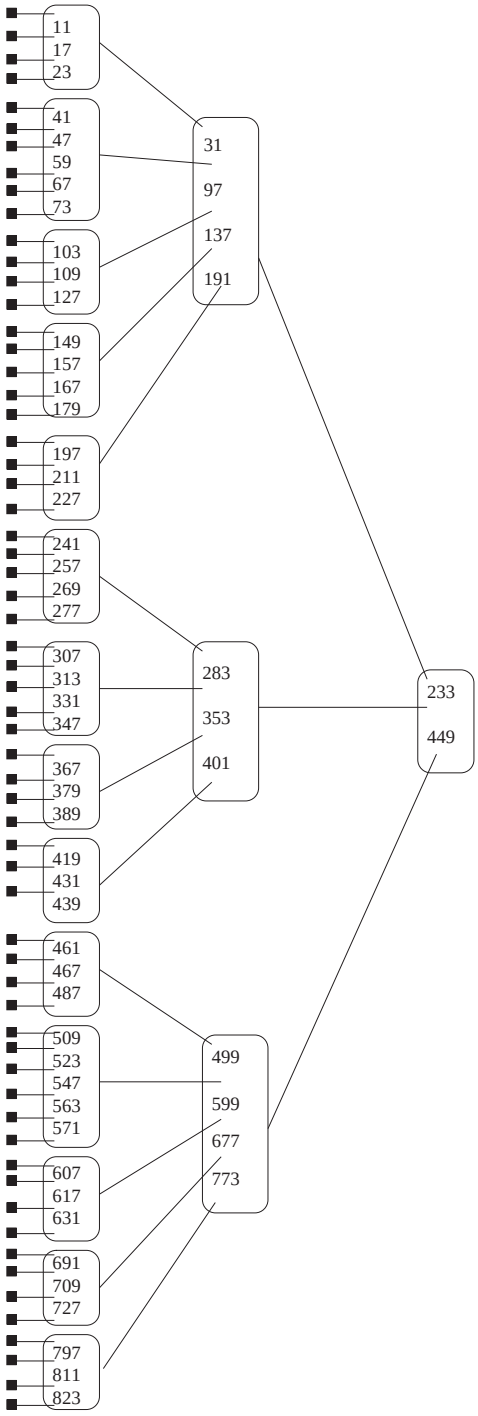
$$\{T_0, T_1, \ldots, T_j\}$$

.



Then

1. $K_0 < K_1 < \ldots < K_{j-1}$
2. every key value within T_0 is less than K_0
3. every key value within T_j is greater than K_{j-1}
4. every key value within T_i where $0 < i < j$ lies between K_{j-1} and K_j.



order 5

every node ≤ 5 children

every internal node ≥ 4 children

all leaves at the same level

a node with K sons has

$K - 1$ keys

Figure 5.3: B-tree of order 5

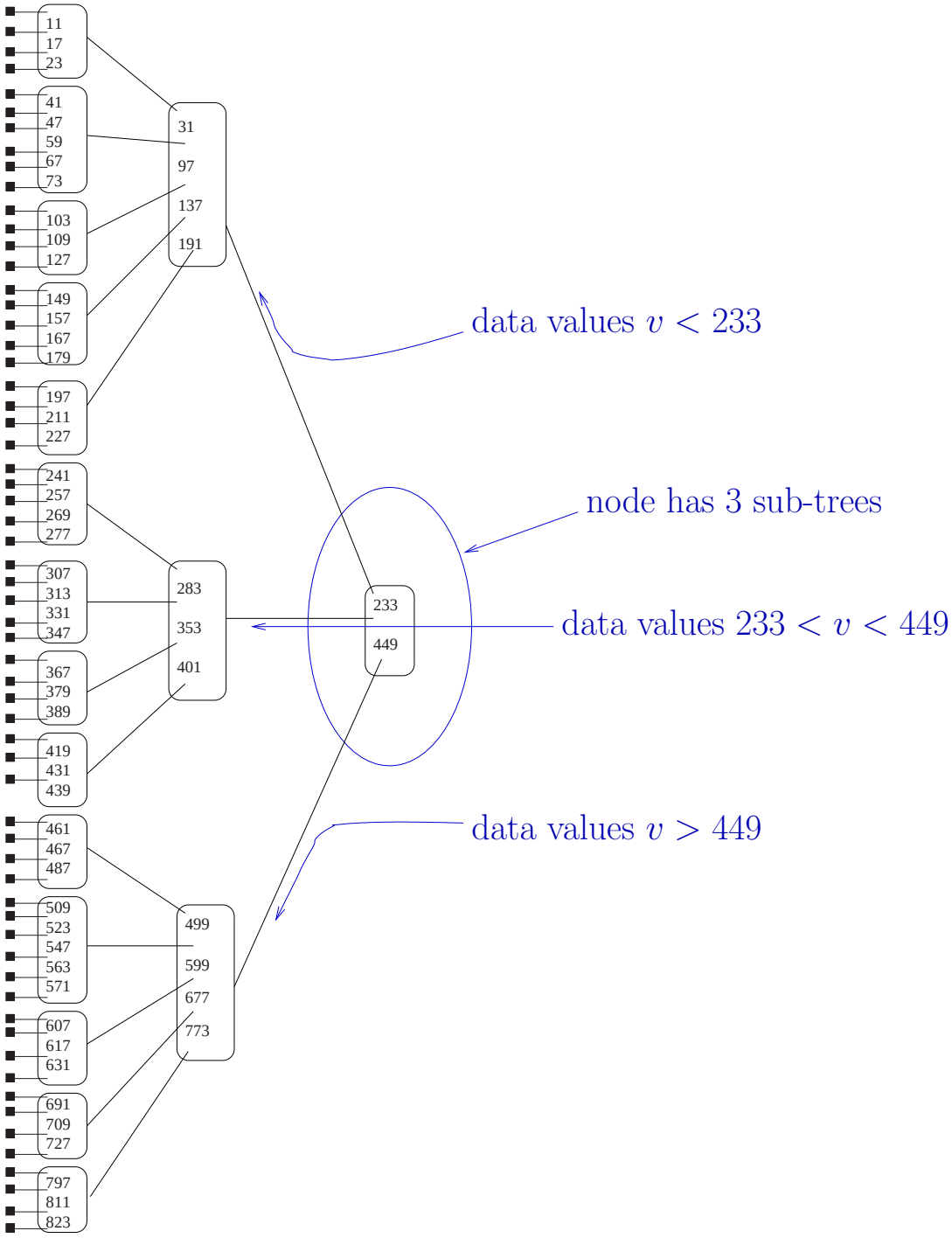
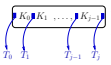


Figure 5.4: Routing of values

5.2.5 B-tree Search

We define an operation to search a B-tree for a given value. Note that the operation returns a `boolean` value to indicate whether the value occurs, it does not indicate where it occcurs:

`B-search (x,▪) = false`

`B-search (x,`  `) =`

`if` $\exists i : 0 \leq i < j.x = K_i$

`true`

`else if` $x < K_0$

`B-search (x,` T_0`)`

`else if` $\exists i : 0 \leq i < j - 1.K_i < x < K_{i+1}$

`B-search (x,` T_i`)`

`else`

`B-search (x,` T_j`)`

5.2.6 Worst Case Analysis (B-tree search)

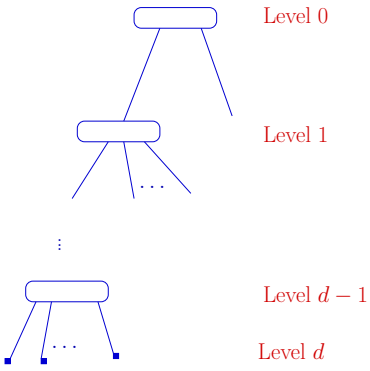
Suppose we are searching within a B-tree of order m containing N elements.

Basic operation - 3-way comparison

Suppose that the depth of the tree is d and that the number of operations taken to search through the $m-1$ keys of a node is $W_{\text{node}}(m-1)$. Then

$$W(N,m) \leq d * W_{\text{node}}(m-1)$$

For a given number of nodes, the depth of a B-tree is maximised by minimizing the number of keys in a node. (Why?)



Level	Min. No. of nodes at this level
0	1
1	2
2	$2\lceil \frac{m}{2} \rceil$
3	$2(\lceil \frac{m}{2} \rceil)^2$
$\vdots$	$\vdots$
l	$2(\lceil \frac{m}{2} \rceil)^{l-1}$

The depth is the maximum level, and equals the number of node searches carried out.

Proposition 5.2.2 *A B-tree with N data (key) values has $N+1$ leaves.*

Exercise: Prove this (by induction).

Thus, since each leaf is a node at level d,

$N+1$	$\geq$	$2(\lceil \frac{m}{2} \rceil)^{d-1}$
no. of leaves		min no. of nodes at level d

$$\therefore \lg(N+1) - 1 \geq \lg(\left\lceil \frac{m}{2} \right\rceil) * (l-1)$$

$$\text{or } d \lesssim \left(\frac{\lg(N+1) - 1}{\lg(m) - 1} \right) + 1$$

Now, noting each node in a B-tree of order m has at most $m-1$ key values, we assume that $W_{\text{node}}(m-1) \leq m$. Recalling that $W(N,m) \leq d * W_{\text{node}}(m-1)$,

$$W(N,m) \lesssim \left(\frac{\lg(N+1) - 1}{\lg(m) - 1} + 1 \right) * m$$

But, for a given tree, the order is fixed. Thus m is a constant and we write

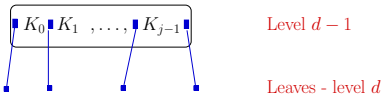
$$W(N) \leq C * \lg(N+1)$$

where C is a constant.

5.2.7 B-tree Insertion

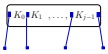
Suppose that a B-tree T is of order m and has depth d. Assuming that the value x is not in T, we insert x into T as follows:

1. Use the same algorithm as `B-search(`x`,`T`)` to find a path through the tree to level $d-1$ (the last level of nodes before the leaves). This leads to a node of the form



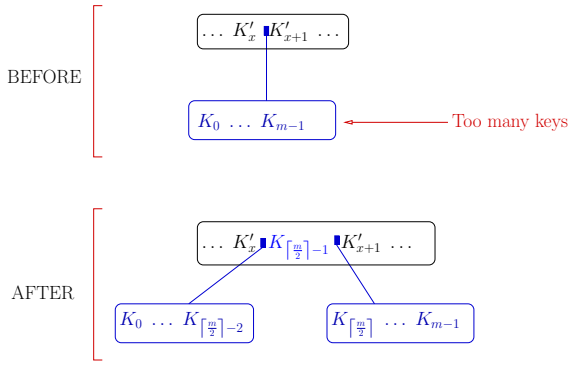
2. Use `keyinsert` to insert x into a node at level $d-1$, restructuring the tree if necessary;

Definition 5.2.3

`keyinsert(`x`,`  *`)` inserts x into the node,*

1. *preserving the ascending order property of the keys;*
2. *preserving the constraints on the number of children that a node may have;*

If for this node $j+1 = m$ (node overflow) then the node has to be split and the links from its parent have to be updated.



As a result, the parent node may now contain m keys and it too will need to be split. In the worst case splitting occurs all the way back to the root and a new root node may need to be created. This is how all the leaves are maintained at the same level - new levels are created at the top of the tree.

5.2.8 Worst Case Analysis (B-tree Insertion)

We are now modifying data structures, so the definition of basic operation is extended to include any operation on the list of keys in a node, either access or modification.

In the worst case, remembering that the length of the list of keys at a node is bounded by $m - 1$, each node in a path through the tree is

1. searched: $O(m)$;
2. split - 3 new lists may be created (parent and 2 children) - $O(m)$.

As before the worst case depth is $\approx \dfrac{\lg N}{\lg m}$

So, the worst insertion requires $\lg N * \dfrac{Km}{\lg m}$

where K is some constant.

But, for any given tree, m is a constant, so we get

$$\lg N * \frac{Km}{\lg m} = \boxed{O(\lg N)}.$$

Deletion

Deletion of a key may cause "underflow", i.e. a node with $< m/2$ children. Deletion may thus require node merging.

5.3 Uses of B-trees

5.3.1 Internal memory searching

A B-tree of order 3 is a 2-3 tree (each node has either 2 or 3 children) and is used for searching in main memory, where all data can be accessed quickly.

5.3.2 External memory

We assume of external (disc-based) memory that

1. the memory is divided into fixed-size pages;
2. the I/O time taken to read or write a page completely dominates the page processing time;

The time needed to search an external database is dominated by the number of disc accesses. If each node is stored on one disc page then the number of disc accesses is (at worst) the depth of the tree.

Example

Suppose

$N = 1,999,999$ - number of data values
$m = 199$ - order of tree, node on one page

At most 3 disc accesses are needed to search such a tree if the root node is stored in main memory.

Worst Case Analysis
Maximum number of disc accesses $=$

maximum depth of a B-tree of order m, with N elements

$$\leq \frac{\lg\left(\frac{N+1}{2}\right)}{\lg\left(\left\lceil\frac{m}{2}\right\rceil\right)} + 1 =$$

For $N = 1,999,999$ and $m = 199$

$$\text{Maximum depth of B-tree} \leq 4$$

So, remembering that the root node is in main memory, at most 3 other nodes are searched and so at most 3 page accesses are required.

To maximise the depth of a B-tree we assume that each node contains the minimum amount of data. In this case, the minimum amount of branching at each node other than the root is $\lceil m/2 \rceil = 100$.

Level	No. of nodes at level	Subtrees per node	Data items at level
0	1 (root)	2	1
1	2	100	2*99
2	2*100	100	2*100*99
3	2*100*100	100	2*100*100*99
4	2*100*100*100	0 (leaves)	0

Total number of items:

Main memory:	1 (the single key in the root)
Secondary memory	2*99 + 2*100*99 + 2*100*100*99

giving a total of 1,999,999.

Constraints on m (order of B-tree)

- m should be as large as possible to minimize the depth of the tree (Why?);
- m is constrained from above by the page size;

5.3.3 Hash Tables

As an alternative to tree structures we can create tables based on hash functions. We assume that

1. the key is a natural number;
2. we have a hash function h: $\mathbb{N} \rightarrow \{0, \ldots, m-1\}$ where $m > 1$;
3. h can be evaluated using arithmetic operations, and $h(n)$ can be computed in constant time (i.e. $\Theta(1)$) for all n;
4. `T` is table (array) with indices $0..m-1$;

A set of items $\mathbf{S}$, where $|\mathbf{S}|$ (`size of` $\mathbf{S}$) $\leq m$ can be represented (in certain circumstances) as:

$$\texttt{T}[h(r)] = r, \forall r \in \mathbf{S}$$

All other enries in T are considered void.

5.3.4 Properties of Hash Functions

It is often the case that

$$r_1, r_2 \in \mathbf{S} \wedge r_1 \neq r_2 \text{ yet}$$

$$h(r_1) = h(r_2)$$

Such a situation is called a collision. Are collisions common?

Example

Find the smallest number of people that need to be present for the probability that two of them share a birthday to be $> 1/2$, where same birthday means same month and day but not necessarily same year.

Let r be the number of people present. Then the probability P of all different birthdays is

$$P = \underbrace{\frac{365}{365}}_{\text{first person}} * \underbrace{\frac{364}{365}}_{\text{2nd person}} * \ldots * \underbrace{\frac{365-(r-1)}{365}}_{\text{rth person}}$$

$$= \frac{365!}{(365-r)!365^r}$$

If $r \geq 23$ then $(1-P) > 1/2$.

So, even though there are 365 possible output values, with just 23 input values there is $> 1/2$ chance of a collision.

In general, how many hash functions of type

$$\mathbb{N} \to \{0, \ldots, m-1\}$$

map a given n distinct input values to distinct output values?

n distinct values

- first function $\longrightarrow$ m possible targets
- second function $\longrightarrow$ $m-1$ possible targets
- $\vdots$
- n^{th} function $\longrightarrow$ $m-(n-1)$ possible targets

i.e. there are $m * (m-1) * \ldots * (m-n+1)$ perfect hash functions from n to m.

$$= \frac{m * (m-1) * \ldots * (m-n+1) * (m-n) * \ldots * 1}{(m-n) * \ldots * 1}$$

$$= \frac{m!}{(m-n)!}$$

However, altogether there are

n distinct values $\left\{ \begin{array}{ll} \bullet \xrightarrow{\text{first function}} & m \text{ possible targets} \\ \bullet \xrightarrow{\text{second function}} & m \text{ possible targets} \\ \vdots & \\ \bullet \xrightarrow{n^{\text{th}} \text{ function}} & m \text{ possible targets} \end{array} \right.$

m^n possible functions from n values to m values.

Examples (approximate figures)

n	m	$\frac{m!}{(m-n)!}$	m^n	% of function space that is PERFECT
7	13	$8 * 10^6$	$63 * 10^6$	12
10	13	10^9	$140 * 10^9$	< 1

When we do not know in advance the key values to be stored, it is likely that collisions will occur.

5.3.5 Some Possible Hash Functions

1. $h(k) \triangleq k \mod m$ (where $\mod \equiv$ % in Java)

   (a) If m is even then

   $$\text{even}(k) \Rightarrow \text{even}(h(k))$$
   $$\text{odd}(k) \Rightarrow \text{odd}(h(k))$$

   Thus if there happens to be a severe bias in the data towards even numbers (as may well happen), approximately only $m/2$ hash values are used - i.e. there is an increased chance of collisions.

(b) A key might be an alphanumeric character string (of length l), e.g. JEAN DO FLORETTE.

Typically, each letter is represented using a fixed number of bits w (wordlength) giving a possible $b = 2^w$ characters. For example,

w	b	
5	32	upper-case letters
6	64	upper- and lower-case letters
7	128	7-bit ASCII
8	256	8-bit ASCII
16	65536	Unicode (Java type `char`)

A string of length l can then be represented "to base b" by

$$\sum_{i=0}^{l-1} b^i * \text{representation of } i^{\text{th}} \text{ letter}$$

For example using 8-bit ASCII,

...	E	T	T	E
...	$+256^3 * 69$	$+256^2 * 80$	$+256 * 80$	$+69$

The hash function can then be applied to this numeric representation. If $m = b$ then

$$k \mod m$$

reduces to the representation of the last (least significant) character. Again there may be a data bias, for example many words end in "e", few end in "x", leading to unnecessary collision problems.

It is also desirable that m is not a multiple of b. In order to avoid such problems, m is often required to be PRIME.

Problem

Consider a symbol table in a compiler. Such a table has to store information about all user-defined identifiers such as veriable names and method names. Some identifier names are more likely than others - e.g. `i` and `j`. If a hash function h is used to store the identifiers and h is fixed there may be many collisions every time a particular program is compiled.

Solution

Consider a set of hash functions

$$H = \{g : \mathbb{N} \rightarrow 0..m-1 | \ldots\}$$

where

$$\forall h \in H. \forall x, y \in \mathbb{N}$$

$$x \neq y \Rightarrow \text{probability}(h(x) = h(y)) \leq \frac{1}{m}$$

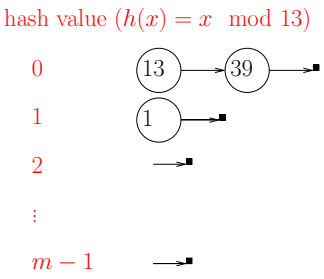
Every time that a fresh compilation takes place hash function is chosen at random from H.

2. $h_{ij}(x) \triangleq ((i * x) + j) \mod p) \mod m$ where p is a large prime number. This parameterised hash function can be used to define a class

$$H = \{h_{ij} | 0 < i < p \wedge 0 \leq j < p\}$$

5.3.6 Collision Resolution- Chaining

Searching (and inertion/deletion) algorithms must include mechanisms for resolving collisions. Chaining involves associating a linked list with each hash value.



Situation after inserting 13, 1 and 39

Chaining Algorithms

INPUT:

```
Node []table;
int x; // x >= 0
```

OUTPUT:

```
boolean present; // output from Search
```

PSEUDOCODE

Assume the following declarations (for simplicity fields are made public).

```
class Node
{
    public int key;
    public Node next;
}

int h(int x)
{     // implements the hash function h(x)
    return ...;
}
```

Search then becomes:

```
Node node = table[h(x)];
while (node != null && node.key != x)
    node = node.next;
present = node != null;
```

The basic operation (probe) combines `node != null` and `node.key != x`.

Insertion is

```
Node newNode = new Node();
newNode.key = x;
newNode.next = table[h(x)];
table[h(x)] = newNode;
```

Exercise

Write a deletion algorithm.

5.3.7 Worst Case - Hashing with Chains

In the worst case, all n data elements are mapped by h to a single hash value. Then the time for a search is

$$\underbrace{C}_{\substack{\text{hash function}\\ \text{evaluation}}} + W\begin{pmatrix}\text{search a list}\\ \text{of length } n\end{pmatrix}$$

$= C + n$ comparisons

$= \Theta(N)$

If the worst case behaviour is so poor, why continue with hash tables? Because the average (expected) behaviour is very good.

5.3.8 Average Case - Hashing with Chains

We assume that

1. The set of available key values $\mathbb{N}' \subset \mathbb{N}$ is finite and, for convenience, a multiple of m. The number of values used at a given point in time is n.
2. The hash function $h : \mathbb{N}' \to 0..m-1$ distributes the domain (key values) uniformly over the range, so that the same number of key values are mapped to each hash table index.

$$\forall i, j \in \{0..m-1\}.|h^{-1}(i)| = |h^{-1}(j)|$$

3. All key values are equally likely to be chosen as an argument to h.

We can thus infer that

$$x \in \mathbb{N}', i \in 0..m-1 \Rightarrow \text{probability}(h(x) = i) = \frac{1}{m}$$

Let

$$\text{prob}(\overbrace{\#}^{\text{size}}\ \underbrace{l}_{\text{chain}}(\overbrace{i}^{\text{index}}) = j)$$

denote the probability that the i^{th} list has length j.

The chances of collision clearly increase as the table becomes fuller, so we define

Definition 5.3.1

$$\lambda = \frac{n}{m}$$

is the LOAD FACTOR of the hash table.

The average time for an unsuccessful search of a table with load factor λ is

$$A_u(\lambda) = \sum_{j=0}^{n} \text{prob}(\#l(i) = j) * \underbrace{(1+j)}_{\substack{\text{search list of} \\ \text{size } j \text{ values} \\ \text{plus } \texttt{null}}}$$

That is, the situations are grouped by the length of the chain (just as the average case analysis for binary search grouped situations by the number of comparisons).

Now

$$\text{prob}(\#l(i) = j) =$$

$$\underbrace{\binom{n}{j}}_{\substack{\text{combinations} \\ \text{of } j \text{ items} \\ \text{from } n}} * \underbrace{\left(\frac{1}{m}\right)^j}_{\substack{\text{probability of } j \\ \text{items, } x_k, \text{ be-} \\ \text{ing such that} \\ h(x_k) = i}} * \underbrace{\left(\frac{m-1}{m}\right)^{(n-j)}}_{\substack{\text{probability} \\ \text{of remaining} \\ \text{items } x_k \text{ be-} \\ \text{ing such that} \\ h(x_k) \neq i}}$$

So $A(\lambda) =$

$$\sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{m}\right)^j \left(\frac{m-1}{m}\right)^{(n-j)} (1+j)$$

$$= 1 + \sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{m}\right)^j \left(\frac{m-1}{m}\right)^{(n-j)} * j$$

as $\sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{m}\right)^j \left(\frac{m-1}{m}\right)^{(n-j)} =$

$$\left(\frac{1}{m} + \frac{m-1}{m}\right)^n = 1 \qquad (*)$$

$$= 1 + \sum_{j=1}^{n} \binom{n}{j} \left(\frac{1}{m}\right)^{j} \left(\frac{m-1}{m}\right)^{(n-j)} * j$$

$$= 1 + \lambda \sum_{j=1}^{n} \binom{n-1}{j-1} \left(\frac{1}{m}\right)^{j-1} \left(\frac{m-1}{m}\right)^{(n-j)}$$

as $\left(\frac{j}{m}\right)\binom{n}{j} = \left(\frac{j}{m}\right)\left(\frac{n}{j}\right)\binom{n-1}{j-1} = \lambda\binom{n-1}{j-1}$

now put $k = j - 1$

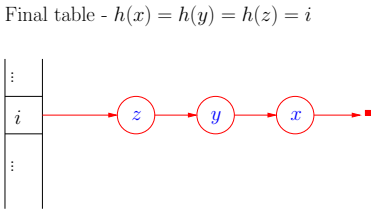
$$= 1 + \lambda \sum_{k=0}^{n-1} \binom{n-1}{k} \left(\frac{1}{m}\right)^{k} \left(\frac{m-1}{m}\right)^{(n-1-k)}$$

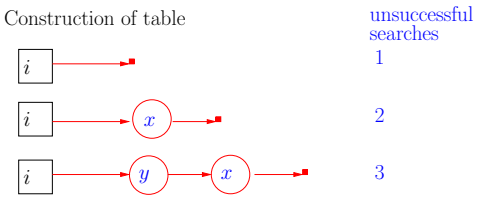
$= 1 + \lambda$ see $(*)$ again

Example

Find the average time for a successful search of such a table.

The average cost of a successful search of a table with load factor λ, $A_s(\lambda)$, is approximated by considering the costs of unsuccessful searching during the construction of the table:





$A_s(\lambda) \approx$ average number of unsuccessful searches at each point during the construction of the table

But the average value of a function $f(x)$ over a range $a..b$ is

$$\frac{1}{b-a}\int_a^b f(x)\,dx$$

$$\therefore A_s(\lambda) \approx \frac{1}{\lambda}\int_0^\lambda 1+x\,dx = \left[x+\frac{x^2}{2}\right]_0^\lambda$$

$$= \frac{1}{\lambda}\left(\lambda+\frac{\lambda^2}{2}\right) = \boxed{1+\frac{\lambda}{2}}$$

Example

For successful search

Average no. of comparisons	Load factor .5	.6	.7	.8	.9
experiment	1.19	1.25	1.28	1.34	1.38
theory	1.25	1.3	1.35	1.4	1.45

5.3.9 Other Methods of Collision Resolution

Here eEach table entry contains a data item rather than a reference to a linked list.

1. Probing (Open Addressing)

   To search for x, first evaluate $h(x)$. If this leads to a table entry which is non-empty but not x, examine adjacent entries in the cyclic order $h(x)-1, h(x)-2, \ldots, 0, m-1, m-2, \ldots, h(x)+1$.

   If an entry containing x is found: success
   If the complete table has been searched or a "hole" is found:unsuccessful search

   Insertion uses the same probe sequence until a "gap" is found.

2. Double Hashing

   This is similar to the (linear) probing above, except that the table is searched in the cyclic order

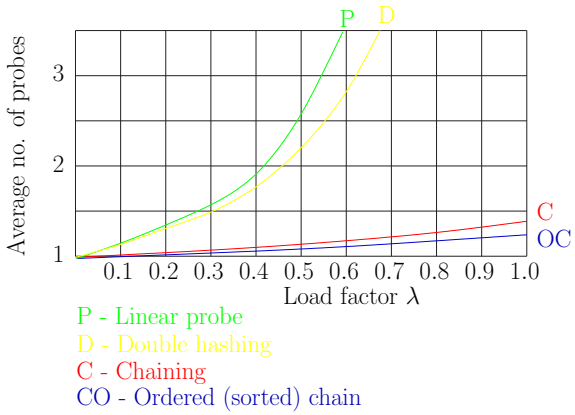


Figure 5.5: Unsuccessful search (limit as $m \to \infty$)

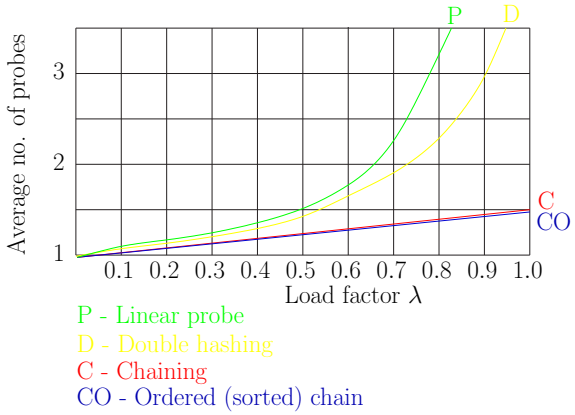


Figure 5.6: Successful search (limit as $m \to \infty$)

$(h(x) - h'(x)) \mod m$,
$(h(x) - 2h'(x)) \mod m$,
... etc.
where $h'(x)$ is a second hash function.

5.3.10 External Memory

Hashing with separate chains can easily be modified to an algorithm for searching external memory. The hash table itself is stored on internal memory but each entry points to the location of a chain in external memory, where each chain is held in the same block.

If a page becomes full (overloaded table) then either

1. a chain of pages can be searched; or

2. an overflow region can be accessed

5.3.11 Advantages and Disadvantages of Hashing

Advantages

- excellent average behaviour;
- relatively simple

Disadvantages

- after an unsuccessful search we only know that a key is not present - we cannot easily determine the next largest (or smallest) key;
- terrible worst case behaviour;
- need faith in probability theory for average case analysis

Chapter 6

Sorting

Sorting - rearranging the elements of a list into order by comparing fields containing key values. We will assume that lists are represented as arrays and that the required order is nondecreasing.

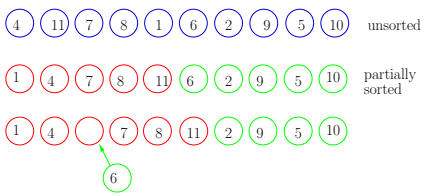
Key - element of some linearly ordered set.

Internal Sort - data is assumed to be in main memory.

In Place - if the amount of extra space used is constant with respect to input size the algorithm is said to work in place.

6.1 Insertion Sort

6.1.1 The Strategy



Algorithm 6.1.1 (Insertion Sort)

```
void insertionSort(Key l[])
{
    int n = l.length;
```

```
    Key x;
    int xIndex, j;

    for (xIndex = 1; xIndex < n; xIndex++)
    {
        x = l[xIndex];
        j = xIndex-1;
        while (j >= 0 && l[j] > x)
        {
            l[j+1] = l[j];
            j = j - 1;
        }
        l[j+1] = x;
    }
}
```

6.1.2 Worst Case Analysis of Insertion Sort

The basic operation is key comparison. Let $i \equiv$ `xIndex`. For each value of i, the maximum number of key comparisons is i. So

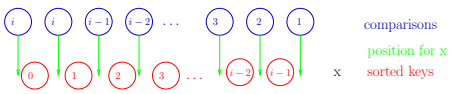
$$W(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

So $W(n) \in \Theta(n^2)$ and this worst case occurs when the keys are initially in decreasing order.

6.1.3 Average Case Analysis of Insertion Sort

We assume that the keys are distinct and that all initial permutations are equally likely.

For each i, how many iterations of the `while` loop are performed on average? It depends upon which position `x` belongs in.



The probability that `x` belongs in any particular positions is $\frac{1}{i+1}$, so the average number of comparisons to insert `x`

$$= \sum_{j=1}^{i} \frac{1}{i+1} * j + \frac{1}{i+1} * i = \frac{i}{2} + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}$$

$$A(n) = \sum_{i=1}^{n-1} \left(\frac{i}{2} + 1 - \frac{1}{i+1} \right)$$

$$= \frac{n(n-1)}{4} + (n-1) - \sum_{i=1}^{n-1} \frac{1}{i+1}$$

$$= \frac{n^2}{4} + \frac{3n}{4} - 1 - \sum_{i=1}^{n-1} \frac{1}{i+1}$$

But $\sum_{i=1}^{n-1} \frac{1}{i+1} \approx \ln n$

$$\therefore A(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

6.1.4 Space

Insertion Sort is In-Place.

6.1.5 Lower Bounds (by inversions)

Note that indices $1..N$ rather than $0..N-1$ are conventionally used in discussion of permutations.

Suppose that a final sorted list L is

$$K_1, K_2, \ldots, K_N$$

and that a $1-1$ permutation function

$$\Pi : 1..N \to 1..N = (\Pi(1), \Pi(2), \ldots, \Pi(N))$$

maps K_i in the list L to position $\Pi(i)$ in the list $\Pi(L)$.

For example, with $N = 5$,

$L = (\boxed{K_1}, K_2, K_3, K_4, K_5)$ value in position 1 is $\boxed{K_1}$

$\Pi = (\boxed{2}, 4, 1, 5, 3)$ 1 is mapped to $\boxed{2}$

$\Pi(L) = (K_3, \boxed{K_1}, K_5, K_2, K_4)$ now $\boxed{K_1}$ is in position 2

Exercise: How many permutation functions are there for a list of size N?

We use L to represent the sorted list (i.e. the output of the algorithm) and $\Pi(L)$ to represent the input list which has to be sorted.

Then $i < j \wedge \Pi(i) > \Pi(j)$ indicates that K_i and K_j are out of order in the original list $\Pi(L)$.

i.e. $i < j$ $\Rightarrow K_i$ (position i) comes before K_j (position j) in final sorted list so $K_i < K_j$

but $\Pi(i) > \Pi(j)$ $\Rightarrow K_i$ (position $\Pi(i)$) comes after K_j (position $\Pi(j)$) in initial unsorted list

Definition 6.1.1

An inversion of a permutation Π is a pair $(\Pi(i), \Pi(j))$ where $i < j \wedge \Pi(i) > \Pi(j)$

That is, an inversion is a pair of target positions which are the wrong way round. For example, the permutation $(2, 4, 1, 5, 3)$ has the following inversions:

$$(2, 1), (4, 3), (5, 3), (4, 1)$$

If a sorting algorithm removes at most one inversion after each key comparison (e.g. by swapping adjacent values) then the number of comparisons performed on the input $(\Pi(1)..\Pi(N))$ is at least the number of inversions of Π.

If $\Pi = (N, N-1, \ldots, 1)$ there are $\frac{N(N-1)}{2}$ inversions, so the worst-case behaviour of any sorting algorithm that removes at most one inversion per key comparison must be $\in \Omega(N^2)$.

Lower Bound

Definition 6.1.2

The transpose of a permutation Π is the permutation $(\Pi(N), \Pi(N-1), \ldots, \Pi(1)$.

For example, the transpose of $(2, 4, 1, 5, 3)$ is $(3, 5, 1, 4, 2)$. Each permutation has a unique transpose. Consider the inversions of a permutation and its transpose:

(2,4,1,5,3)	(3,5,1,4,2)
(2,1),	(3,1),(3,2),
(4,1),(4,3),	(5,1),(5,4),(5,2),
(5,3)	(4,2)
(1,2,3,4,5)	(5,4,3,2,1)
	(5,4),(5,3),(5,2),(5,1),
	(4,3),(4,2),(4,1),
	(3,2),(3,1),
	(2,1)

In the lists of inversions for a permutation and its transpose, for each $i, j, i > j$ the inversion (i, j) appears exactly once. There are $\frac{N(N-1)}{2}$ inversions to be distributed across each pair of permutations and so the average number of inversions per permutations is $\frac{N(N-1)}{4}$.

> Any algorithm that sorts by comparison of keys and removes at most one inversion after each comparison must perform at least $\frac{N(N-1)}{2}$ comparisons in the worst case and at least $\frac{N(N-1)}{4}$ comparisons on average.

So if we want to improve on insertion sort, we must remove more than one inversion per comparison...

6.2 Quicksort

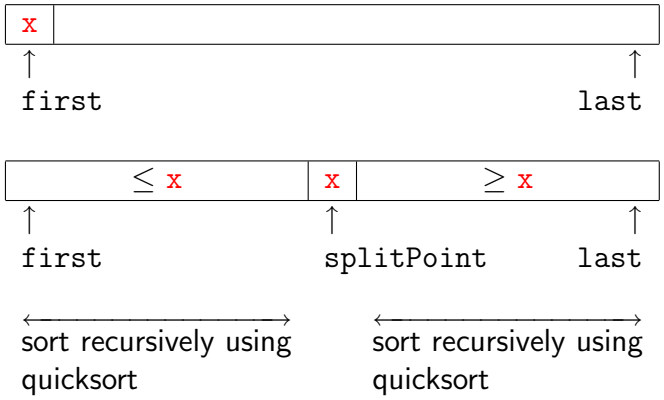
(due to Tony Hoare, once Professor of Computer Science at Q.U.B.)

6.2.1 The Strategy

`L` is an array of keys
`first` = index of first entry in `L`
`last` = index of last entry in `L`

Choose a key x then arrange the values in L so that for some index splitPoint

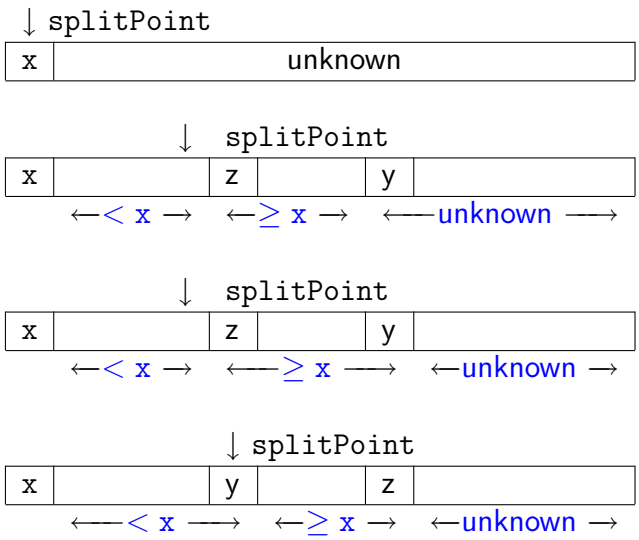
$$
\begin{array}{lll}
\texttt{L}[i] & \leq \texttt{x} & \text{if } \texttt{first} \leq i < \texttt{splitPoint} \\
\texttt{L}[i] & \geq \texttt{x} & \text{if } \texttt{splitPoint} < i \leq \texttt{last} \\
\texttt{L}[i] & = \texttt{x} & \text{if } i = \texttt{splitPoint}
\end{array}
$$



Algorithm 6.2.1 (Quicksort)

```
void quickSort(int first, int last)
// Assumes a global array
{
    int splitPoint;
    if (first < last)
    {
        splitPoint = split (first, last);
        quickSort (first, splitPoint-1);
        quickSort (splitPoint+1, last);
    }
}
```

Strategy for Split



Algorithm 6.2.2 (Split)

```
int split (int first, int last)
{
  int splitPoint, unknown;
  Key x;
  x = l[first];
  splitPoint = first;
  for (unknown=first+1;unknown<=last;unknown++)
    if (l[unknown] < x)
    {
      splitPoint = splitPoint + 1;
      interchange (splitPoint, unknown);
    }
  interchange (first, splitPoint);
  return splitPoint;
```

}

6.2.2 Worst Case Analysis of Quicksort

If `split` is working on a section of the array with K keys it performs $K-1$ comparisons.

If `l[first]` is the smallest key we are left with two lists; one empty (keys smaller than `l[first]`) and one with $K-1$ keys.

That is, if each time `split` is called `l[first]` is the smallest key in the given range,

$$W(n) = \sum_{K=2}^{n}(K-1) = \frac{n(n-1)}{2}$$

When does this occur?

6.2.3 Average Case Analysis of Quicksort

Assume that all keys are distinct and that all permutations are equally likely. The probabilty of using any given `splitPoint` (i) is $\frac{1}{n}$ and the cost of the recursive calls is $A(i) + A(n-1-i)$ so

$$A(n) = \underbrace{n-1}_{\text{split cost}} + \sum_{i=0}^{n-1}\frac{1}{n}\underbrace{\{A(i)+A(n-1-i)\}}_{\text{typical recursive cost}}, n \geq 2$$

$A(1) = A(0) = 0$

So the recurrence relation is

$$\begin{aligned} A(n) &= n-1+\frac{2}{n}\sum_{i=2}^{n-1}A(i), n \geq 2 \qquad (6.1)\\ A(1) &= 0 \end{aligned}$$

Now

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

scale to match summation terms

$$\begin{aligned} nA(n) - (n-1)A(n-1) &= n(n-1) + 2\sum_{i=2}^{n-1} A(i) - \\ &\quad (n-1)(n-2) - 2\sum_{i=2}^{n-2} A(i) \\ &= 2A(n-1) + 2n - 2 \end{aligned}$$

$$\boxed{nA(n) = (n+1)A(n-1) + 2n - 2}$$

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2n-2}{n(n+1)}$$

Let $B(n) = \dfrac{A(n)}{n+1}$ so that $B(1) = \dfrac{A(1)}{2} = 0$.

$$B(n) = B(n-1) + \frac{2n-2}{n(n+1)}$$

So [1] $B(n) = \displaystyle\sum_{i=2}^{n} \frac{2i-2}{i(i+1)} \approx 2\ln n$

$$\begin{aligned} A(n) &\approx 2(n+1)\ln n \\ &\approx 1.4(n+1)\lg n \end{aligned}$$

$$A(n) \in \Theta(n \lg n)$$

Algorithm 6.2.3 (Quicksort with Explicit Stacking)

```
void quickSortWithStack (Key l[])
{
  int first, last, splitPoint;
  Stack stack = new Stack();
  stack.push (<0,l.length-1>); // not java
  while (!stack.empty())
```

[1] Given the recurrence relation $Y(1) = 0$ and $Y(n) = Y(n-1) + X(n)$

$$\begin{aligned} Y(2) &= Y(1) + X(2) &= X(2) \\ Y(3) &= Y(2) + X(3) &= X(2) + X(3) \\ Y(n) &= \sum_{i=2}^{n} X(i) \end{aligned}$$

```
  {
    <first,last> = stack.pop();
    while (first < last)
    {
      splitPoint = split (first, last);
      stack.push (<splitPoint+1,last>);
      // Note not <first,splitPoint-1>
      last = splitPoint - 1;
    }
  }
}
```

Each time a section of the array is split, only the limits for one part are placed on the stack, so stack usage is more efficient than recursive Quicksort.

6.2.4 Space Usage of Quicksort (Explicit Stack)

Worst Case

Given a list of size i suppose `split` produces 2 sub-lists of sizes 0 and $(i-1)$. Where is the missing element? Suppose that the end-points of the smaller sub-list are put on the stack:

New elements added to stack	Size of list still to be sorted
	n
2	n-1
2	n-2
⋮	⋮
2	1 (sorted)

Thus we may have

$$\sum_{i=1}^{n-1} 2 = 2(n-1)$$

elements on the stack. Thus, we need an additional amount of storage that depends on n (in the worst case).

Quicksort is not an in-place sort.

6.2.5 Other Improvements

1. Quicksort works well if the key `x` used by `split` to partition a list is "close" the median - so rather than choosing `x = l[first]` we can use `x = median (l[first], l[(first+last)/2], l[last])`.

2. Alternative coding of `split` - see assignments.

3. The amount of space needed can be reduced by stacking the bounds of the larger sublist.

6.3 Mergesort

The problem with Quicksort is that the two sublists may not be of equal size - the advantage is that combining them is trivial

With Mergesort, we force the two sublists to be equal in size (± 1) by simply cutting the list in half, but then find that it takes more effort to recombine them because keys in the left half may belong in the right half.

6.3.1 Merging Sorted Lists

Problem: Given two lists `A` and `B` sorted in nondecreasing order, merge them to create a sorted list `C`.

Complexity Measure: Number of comparisons of keys.

Strategy: `A` is a list of n items, `B` is a list of m items.

Compare the first remaining keys in `A` and `B`, and move the smaller one to the next vacant position in `C`. When `A` or `B` is empty, move the items in the remaining list to `C`.

A$(n=4)$	B$(m=3)$	C
$2, 6, 14, 15$	$4, 5, 9$	2
$\not{2}, 6, 14, 15$	$4, 5, 9$	$2, 4$
$\not{2}, 6, 14, 15$	$\not{4}, 5, 9$	$2, 4, 5$
$\not{2}, 6, 14, 15$	$\not{4}, \not{5}, 9$	$2, 4, 5, 6$
$\not{2}, \not{6}, 14, 15$	$\not{4}, \not{5}, 9$	$2, 4, 5, 6, 9$
$\not{2}, \not{6}, 14, 15$	$\not{4}, \not{5}, \not{9}$	$2, 4, 5, 6, 9, 14, 15$

Algorithm 6.3.1 (Merge)

```
indexA=0; indexB=0; indexC=0;
while (indexA<n && indexB<m)
  if (A[indexA] < B[indexB])
    C[indexC++] = A[indexA++];
  else
    C[indexC++] = B[indexB++];
if (indexA == n)
  C[indexC..(n+m-1)] = B[indexB..m-1];
else
  C[indexC..(n+m-1)] = A[indexA..n-1];
```

6.3.2 Worst Case Analysis of Merge

Measure the input size by $n' = n + m$. Then

$W(n') = n + m - 1 = n' - 1$ and this occurs when `A[n-1]` and `B[m-1]` belong in the last two positions in `C`.

When $n = m$, $W(n') = 2n - 1$ and in this case the algorithm is optimal.

6.3.3 Optimality when $n = m$

Theorem 6.3.1 *Any algorithm to merge two sorted lists, each containing n keys, which works by comparison of keys, does at least $2n - 1$ comparisons in the worst case.*

Proof:

Consider the input lists `A` and `B` such that
$b_0 < a_0 < b_1 < a_1 < \ldots < b_{n-1} < a_{n-1}$

i.e. $\begin{cases} a_i < b_j & i < j \\ a_i > b_j & i \geq j \end{cases}$

By comparing $\begin{cases} a_i \text{ with } b_i & 0 \leq i < n \text{ and} \\ a_i \text{ with } b_{i+1} & 0 \leq i < n - 1 \end{cases}$

we obtain the correct ordering ($2n - 1$ comparisons).

Let us now try fewer comparisons so that e.g. the chosen algorithm does not

compare a_j with b_j. It is true that

$$a_{j-1} < \left\{ \begin{array}{c} a_j \\ b_j \end{array} \right\} < b_{j+1}$$

but unless a_j is compared with b_j the algorithm cannot tell which order to put them in.

Suppose on the other hand that the algorithm does not compare a_j with b_{j+1}. It is true that

$$b_j < \left\{ \begin{array}{c} a_j \\ b_{j+1} \end{array} \right\} < a_{j+1}$$

but unless a_j is compared with b_{j+1} the algorithm cannot tell which order to put them in.

Therefore, at least $2n - 1$ comparisons are needed in the worst case.

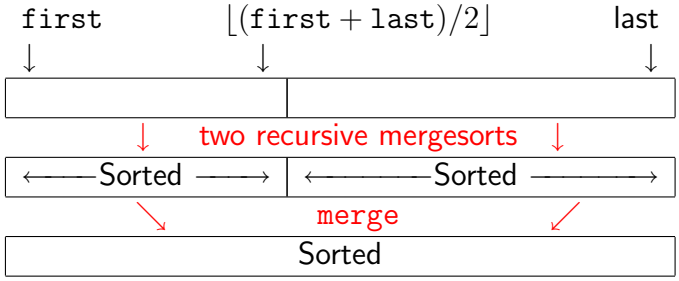
6.3.4 Space Usage

At first sight it appears that merging two lists with a total of n entries requires memory locations for $2n$ entries (n for `A` and `B` together, n for `C`). By reusing space occupied by the larger initial list as the location for `C` it is possible to reduce the amount of extra space needed, but it is still $\Theta(n)$.

By modifying the `merge` algorithm so that it merges adjacent sections of an array and leaves the result in the same locations, we obtain the following:

Algorithm 6.3.2 (Mergesort)

```
void mergeSort(int first, int last)
// Assumes a global array
{
    if (first < last)
    {
        mergeSort (first, ⌊(first + last)/2⌋);
        mergeSort (⌊(first + last)/2⌋+1,last);
        merge (first,⌊(first + last)/2⌋,
            ⌊(first + last)/2⌋+1,last);
    }
}
```



Worst Case Analysis

$$\begin{aligned} W(n) &= W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1 \\ W(1) &= 0 \end{aligned}$$

$$\begin{aligned} W(n) &\approx n \lg n - n \\ W(n) &\in \Theta(n \lg n) \end{aligned}$$

Proof: left as an exercise.

6.4 Lower Bounds for Sorting by Comparison of Keys

6.4.1 Decision Trees for Sorting Algorithms

Given n and keys $K_0, K_1, \ldots, K_n$ each sorting algorithm is associated with a binary decision tree, e.g. Fig. 6.1.

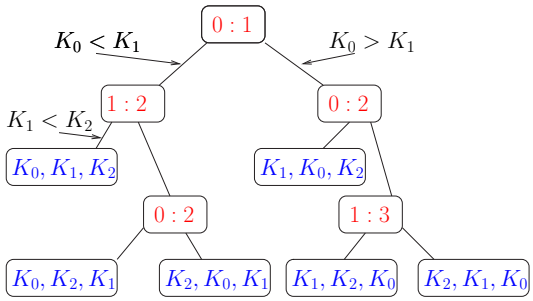


Figure 6.1: Insertion Sort Decision Tree (n=3)

Such a tree has at least $n!$ leaves.

Note

1. Each node corresponds to the comparison of two keys.

2. Any paths that are never followed are removed.

3. Comparison nodes with only one child are removed and replaced by the child.

The "pruned" tree has internal nodes of degree 2; its algorithm is at least as efficient as the original. We will consider trees with $n!$ leaves where all internal nodes are of degree 2; the lower bounds will therefore be valid for all algorithms that sort by comparison of keys.

6.4.2 Lower Bound for Worst Case

Lemma 6.4.1 *Let L be the number of leaves in a binary tree and let d be the depth of the tree. Then $L \leq 2^d$.*

Proof: see section 2.9.

Lemma 6.4.2 *Let L and d be as in lemma 6.4.1. Then $d \geq \lceil \lg L \rceil$.*

Proof:
$$\begin{array}{lcll} 2^d & \geq & L & \\ d & \geq & \lg L & \text{and since } d \in \mathbb{N} \\ d & \geq & \lceil \lg L \rceil & \end{array}$$

Lemma 6.4.3 *For a given n, the decision tree for any algorithm that sorts by comparison of keys has depth at least $\lceil \lg n! \rceil$.*

Proof: Let $L = n!$ in Lemma 6.4.2

Theorem 6.4.4 *Any algorithm to sort n keys by comparison of keys must do at least $\lceil \lg n! \rceil$, or[2] approximately $\lceil n \lg n - 1.5n \rceil$ key comparisons in the worst case.*

Proof: The number of comparisons done in the worst case is the number of internal nodes on the the longest path, i.e. the depth of the tree, $\lceil \lg n! \rceil$. Also,

[2] Stirling's formula $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$

$$
\begin{aligned}
\lg n! &= \sum_{j=2}^{n} \lg j \\
\sum_{j=2}^{n} \lg j &\geq \int_{1}^{n} \lg x \, dx \qquad \text{(Section 2.8, Equation 2.4)} \\
&= \int_{1}^{n} \lg e \ln x \, dx \\
&= [\lg e(x \ln x - x)]_{1}^{n} \\
&= \lg e(n \ln n - n + 1) \\
&= n \lg n - n \lg e + \lg e \\
&\geq n \lg n - n \lg e \\
&\geq n \lg n - 1.5n
\end{aligned}
$$

So $\lg n! \geq n \lg n - 1.5n$ and

d is at least $\lceil n \lg n - 1.5n \rceil$.

Mergesort is very close to optimal. For example, if $n = 5$,

Insertion Sort	10 comparisons in worst case
Mergesort	8 comparisons in worst case
Lower bound	7

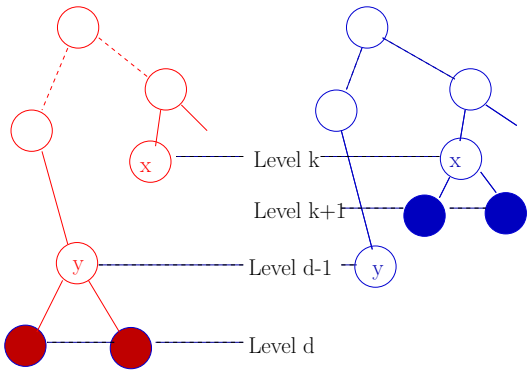
Is the lower bound not good enough, or can we do better than Mergesort?

6.4.3 Lower Bound for Average Behaviour

We look at the External Path Length (EPL) - the sum of the lengths of all paths from the root to a leaf. The trees considered are 2-trees, that is binary trees where each node is either a leaf or is of degree 2.

Lemma 6.4.5 *Among 2-trees with L leaves, the EPL is minimized if all the leaves are on at most two adjacent levels.*

Proof: Suppose we have a 2-tree with depth d that has a leaf x at level k, where $k \leq d - 2$. We will show how to build a 2-tree with the same number of leaves and lower EPL.



Choose a node y at level $d-1$ that is not a leaf, remove its children (which are leaves) and re-attach them as children of x. The total number of leaves is unchanged. The net change in EPL is

$-$	$(2d+k)$	(loss of y's children, loss of x)
$+$	$2(k+1)+d-1$	(gain of x's children, gain of y)

net $2(k+1)+(d-1)-(2d+k)$
$= k+1-d<0$ since $k \leq d-2$

Lemma 6.4.6 *The minimum EPL for 2-trees with L leaves is $L\lfloor \lg L \rfloor + 2(L - 2^{\lfloor \lg L \rfloor})$.*

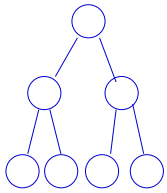
Proof: If $L = 2^k$, all the leaves are at level $\lg L$ (consequence of Lemma 6.4.5). If $L \neq 2^k$, $d = \lceil \lg L \rceil$ and all the leaves are at levels d and $d-1$.
The sum of all the path lengths (for all leaves) down to level $d-1$ is $L(d-1)$. Let L_d be the number of leaves at level d. Then as each pair of leaves at level d is parented by one node at level $d-1$, the number of nodes at level $d-1$ is $2^{d-1} = L_{d-1} + L_d/2$. But
$L = L_d + L_{d-1} \Rightarrow L_{d-1} = L - L_d$,

$$\begin{array}{lrcl} \therefore & 2^{d-1} & = & L - L_d + L_d/2 \\ \Rightarrow & L_d/2 & = & L - 2^{d-1} \Rightarrow L_d = 2(L - 2^{d-1}) \end{array}$$

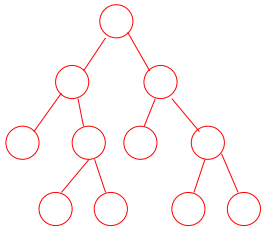
The sum of the lengths of the paths to the leaves is therefore

$$L(d-1) + 2(L - 2^{d-1})$$

$$= L\lfloor \lg L \rfloor + 2\left(L - 2^{\lfloor \lg L \rfloor}\right)$$



$L = 4$
$EPL = 4 * 2 = 8$



$L = 6 \quad d = 3 \quad L(d-1) = 12$
$2(L - 2^{d-1}) = 2(6-4) = 4$
$\text{EPL} = 12 + 4 = 16$

Lemma 6.4.7 *The average external path length in a 2-tree with L leaves is at least $\lfloor \lg L \rfloor$.*

Proof: The minimum average external path length is

$$\frac{L \lfloor \lg L \rfloor + 2\left(L - 2^{\lfloor \lg L \rfloor}\right)}{L}$$

$$= \lfloor \lg L \rfloor + \epsilon$$

where $0 \leq \epsilon < 1$ since $L - 2^{\lfloor \lg L \rfloor} < L/2$ (why?).

Theorem 6.4.8 *The average number of comparisons done by an algorithm to sort n items by comparison of keys is at least $\lfloor \lg n! \rfloor \approx \lfloor n \lg n - 1.n \rfloor$.*

6.4.4 Summary

Algorithm	Worst	Average	Space Usage
Insertion Sort	$\frac{n^2}{2}$	$\frac{n^2}{4}$	in place
Quicksort	$\frac{n^2}{2}$	$\Theta(n \lg n)$	extra space proportional to $\lg n$
Mergesort	$n \lg n$	$\Theta(n \lg n)$	extra space proportional to n

Chapter 7

String Matching

7.1 The Problem and a Straightforward Solution

Problem - Detect the occurrences of a particular substring called a pattern in another string called the text.

`P` represents the pattern

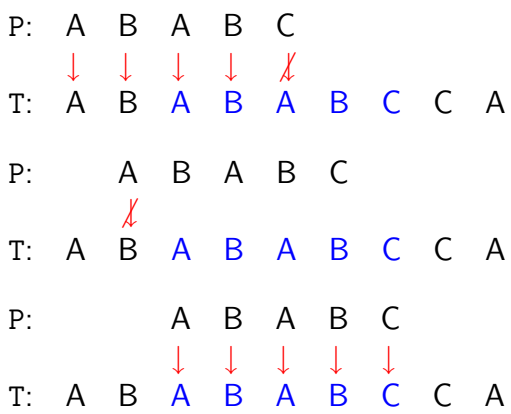
`P.length` (or m) represents the length of (number of characters in) `P`

`T` represents the text

`T.length` (or n) represents the length of `T`

`T.length` $\gg$ `P.length`

Example



Algorithm 7.1.1 (Straightforward String Matching)

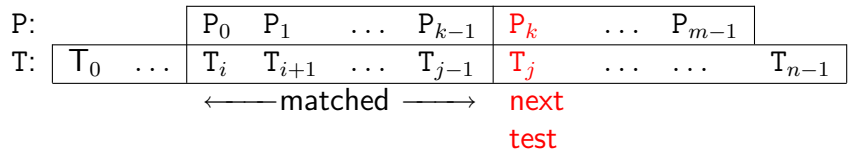
Input : `P,T`. If `P.length` or in particular `T.length` is not known in advance, their explicit use can be replaced by end-of-text tests. Note that in Java the first character of an array is in position 0, denoted `T[0]`, but by convention we use here a T_0 to indicate the first character. Note also the Java type `String` provides more (and different) operations then `char[]`.

Output : The index in `T` where a copy of `P` begins. The index will be `T.length` if no match is found.

Pseudocode

```
int match (char[] P, char[] T)
{
    int i=0,j=0,k=0, result;
    while (j<T.length && k < P.length)
    {
        if (T_j == P_k)
        { j = j+1; k = k+1; }
        else
        { i = i+1; j = i; k = 0; }
    }
    if (k >= P.length)
        result = i; // match found
    else
        result = j; // no match found
    return result;
}
```

At any stage $j = i + k$ and the current state can be represented by:



7.1.1 Analysis

The complexity measure is the number of character comparisons made.
If `P` occurs at the beginning of the text this will be m.
If P_0 is not in the text this will be n.

Worst Case Analysis of Straightforward String Matching

What is the worst case? This will happend if for every start position of `P`, all but the last character of `P` matches. In this worst case the algorithm will do no more than $n * m$ comparisons, so is $O(nm)$.

To show that the algorithm is $\Theta(nm)$ we need to show that an input exists requiring order nm comparisons. Consider

`P` = A A ... A B ($m-1$ A's followed by B)
`T` = A A ... A (n A's)

This input require $m(n+1-m)+1 \approx nm$ comparisons.

In practice this algorithm works well for natural language text. In some empirical studies the algorithm performed ≈ 1.1 comparisons for each character in `T`.

Problem - we may need to "back up" in the text by setting `j = i;` In addition to an increased number of comparisons this may introduce a need for buffering if the text string is being read from an input stream. The K.M.P. algorithm (Section 7.2) was designed to avoid this problem.

Average Case Analysis of Straightforward String Matching

Suppose that the characters are drawn from an alphabet of c symbols where all the symbols are equally likely to occur.

For each "start" position (value of `i` in the algorithm) the expected of comparisons can be deduced as follows:

Character in P	Probability of comparing	Probability of continuing
P_0	1	$\frac{1}{c}$
P_1	$\frac{1}{c}$	$\frac{1}{c^2}$
P_2	$\frac{1}{c^2}$	$\frac{1}{c^3}$
$\vdots$		
P_{m-1}	$\frac{1}{c^{m-1}}$	0

So the expected number of comparisons is

$$1 + \frac{1}{c} + \frac{1}{c^2} \ldots + \frac{1}{c^{m-1}} = \sum_{l=0}^{m-1} \frac{1}{c^l}$$

$$\text{but } \sum_{l=0}^{m-1} \frac{1}{c^l} = \frac{\left(\frac{1}{c}\right)^m - 1}{\frac{1}{c} - 1} \text{ (see Section 2.4)}$$

$$= \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right)$$

If the pattern is not found, there will have been $n - m + 1$ different positions in which it has been tried, so the expected number of symbol comparisons made by the naïve algorithm in an unsuccessful search is

$$\frac{c}{c-1}\left(1 - \frac{1}{c^m}\right)(n - m + 1)$$

For large m this is approximately $\frac{c}{c-1}(n - m + 1)$ and when $n \gg m$

$$\approx \frac{c}{c-1} * n$$

Note that this average case analysis assumes that each letter (symbol) is equally likely to occur. For Natural Language texts this is not the case, e.g. $P(\text{e}) > P(\text{z})$ etc.

7.2 K.M.P. Algorithm

(Knuth, Morris and Pratt)

Suppose that there is a mismatch at the $k+1^{\text{st}}$ character of P:

$$\underbrace{P_0 \ldots P_{k-1}}_{\text{matches text}} \underbrace{P_k}_{\text{mismatch}}$$

We know that $P_0 \ldots P_{k-1} =$ the preceding TEXT string.

Example

			P_0				P_k
Pattern			A	B	A	B	C
Text	D	D	A	B	A	B	A
			MATCHED				$\neq$

Next step

Keep the text position fixed; match a prefix of P (i.e. $P_0..P_{r-1}, r < k$) with the string immediately preceding the mismatch (i.e. $P_{k-r} \ldots P_{k-1}$)

Example

×		A	B	A	$P_0 \dots P_2$	try $r = 3$, no
	A	B	A	B	$P_{k-3} \dots P_{k-1}$	
✓			A	B	$P_0 \dots P_1$	try $r = 2$, yes
	A	B	A	B	$P_{k-2} \dots P_{k-1}$	

In general, align the pattern as far left as possible (i.e. use as big a value of r as possible) subject to:

$$[P_0 \dots P_{r-1}] = [P_{k-r} \dots P_{k-1}] = \text{underlying text}$$

with $P_r \neq P_k$. The length of this prefix is denoted by *next*(k).

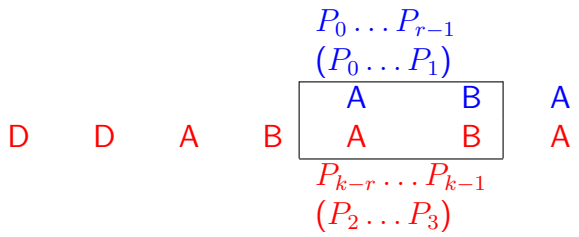
Definition 7.2.1 (next)

$$\textit{next}(k) \triangleq \max\{r| \quad -1 \leq r < k \wedge \\ P_0 \dots P_{r-1} = P_{k-r} \dots P_{k-1} \wedge \\ P_r \neq P_k\}$$

Note

- The function *next* depends only the the pattern, not the text, so it can be a pre-computed table;
- If there are no matching prefixes of P_k then the question is whether the search should continue with the first pattern character against the current text character (*next*$(k) = 0$) or whether there is no point in any further comparisons against the current text character (*next*$(k) = -1$). So in this case *next*$(k) = -1$ if $P_0 = P_k$ and $= 0$ if $P_0 \neq P_k$.

Example



i.e. $r = 2, k = 4$. Note that $P_r \neq P_k$ because P_k caused a mismatch. If $P_r = P_k$ then P_r would be bound to cause a mismatch as well.

Note also that r is made as large as possible so that the pattern does not move too far to the right, thus we make sure that no possible matches are missed.

Algorithm 7.2.1 (KMPMatch)

```
int k=0,j=0;
while (j < T.length && k < P.length)
{
    // if mismatch then slide pattern
    while (k >= 0 && T_j != P_k)
        k = next(k);
    k = k + 1; j = j + 1;
}
```

7.2.1 Construction of the Table Next

Given $P_0 \dots P_{k-1}$ we want to find the next possible pattern position that matches what we know about the text. That is, we want the largest $r < k$ with

$$P_0 \dots P_{r-1} = P_{k-r} \dots P_{k-1} \wedge P_r \neq P_k$$

We could use a brute force algorithm which tries $r = k - 1$ (the largest possibility) and checks whether

$$P_0 \dots P_{k-2} = P_1 \dots P_{k-1}$$

and if that fails try successive smaller values of r, i.e. next

$$P_0 \dots P_{k-3} = P_2 \dots P_{k-1}$$

but instead...

A More Efficient Calculation of Next

We consider the construction of a table `fail`, similar to `next` except that it would laid to a slightly less efficient algorithm because it ignores whether or not $P_r = P_k$. Thus its use in the `scan` algorithm might lead to additional comparisons against the current text character which have no hope of success.

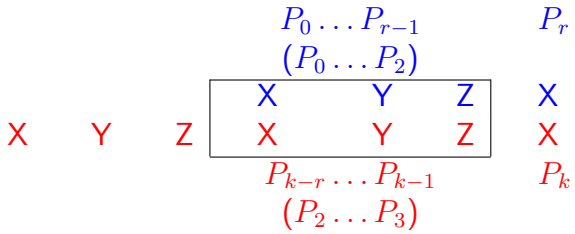
Definition 7.2.2 (fail)

$$\mathit{fail}(k) \triangleq \max\{r \mid \begin{array}{l} -1 \leq r < k \wedge \\ P_0 \dots P_{r-1} = P_{k-r} \dots P_{k-1}\} \end{array}$$

Example

$P =$ "	X	Y	Z	X	Y	Z	X	Z	X	Y	"
`fail` = [	-1	0	0	0	1	2	3	4	0	1	]
`next` = [	-1	0	0	-1	0	0	-1	4	-1	0	]

Thus, for $k = 6$,



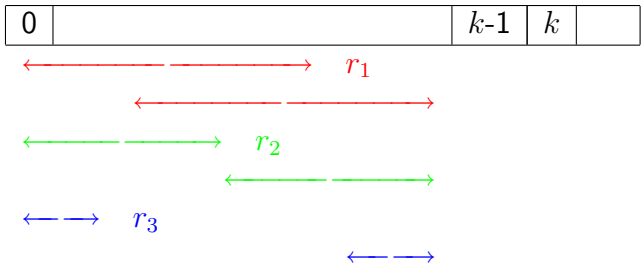
This match means that `fail`$[6] = 3$ but `next`$[6] = -1$ because $P_3 \neq P_6$ and there are no other matching prefixes. The reasons for introducing `fail` are:

- There is an efficient algorithm for its construction; and
- It can be used to calculate `next` (see assignments)

7.2.2 Efficient Calculation of Fail

1. `fail`$(0) = -1$
2. Clearly if $k > 0$ then `fail`$(k) \leq$ `fail`$(k-1) + 1$ as removing the last element from any non-empty matching prefix ending at P_k gives a matching prefix ending at P_{k-1}. But can we extend the matching prefix from just before P_{k-1} to just before P_k?

   Consider the (possibly empty) set of values whose maximum (r_1 in this diagram) gives us `fail`$(k-1)$.



   As we move to look at `fail`(k) it may be possible to extend these matching prefixes, depending on the values of P_{r_1}, P_{r_2} etc.

3. What if $P_{\texttt{fail}(k-1)} = P_{k-1}$?

   $r_1 = \texttt{fail}(k-1)$ gives the length of the longest prefix ending at position $k-2$.

   $P_0 \ldots P_{\texttt{fail}(k-1)-1}$ $P_{\texttt{fail}(k-1)}$

   $=$

   $P_{(k-1)-\texttt{fail}(k-1)} \ldots P_{k-2}$ P_{k-1}

   (see the definition of $\texttt{fail}(k-1)$)
   Suppose the next elements match as well, i.e. that

   $$P_{\texttt{fail}(k-1)} = P_{k-1}$$

   This means that there is also a matching prefix ending at position P_{k-1}, so

   $$\texttt{fail}(k) = \texttt{fail}(k-1) + 1$$

4. Otherwise, $P_{\texttt{fail}(k-1)} \neq P_{k-1}$?

   Could $\texttt{fail}(k)$ still be $\texttt{fail}(k-1)+1$?
   Let $r = \texttt{fail}(k-1)+1$ and see if it work as one of the r's in the definition of `fail`.
   But $P_{r-1} = P_{\texttt{fail}} \neq P_{k-1}$ and

   $$P_0 \ldots P_{r-1} \text{ cannot match } P_{k-r} \ldots P_{k-1}$$

   So if we can't have a prefix of length r, we must look for a smaller one, i.e. of length at most $r-1$, which must also be a matching prefix of $P_0..P_{r-1}$. But the longest matching of $P_0..P_{r-1}$ is $P_0..P_{\texttt{fail}(r)-1}$, i.e. $P_0..P_{r_2}$. So we check whether $P_{\texttt{fail}(r)} = P_{k-1}$.

   This process continues through $r_1, r_2, \ldots$ until a matching (possibly empty) prefix is found.

Algorithm 7.2.2 (Construction of `fail`)
(Construction of KMP flowchart)

```
void KMPSetup (char P[], int fail[])
{
    int k,r;
    fail[0] = -1;
    for (k=1; k<P.length; k++)
    {
        r = fail[k-1];
```

```
        while (r >= 0 && P[r] != P[k-1])
            r = fail[r];
        fail[k] = r+1;
    }
}
```

Example

Caculate $\texttt{fail}(9)$ for the pattern

X	Y	X	X	Y	X	X	Y	A	X

Suppose $\texttt{fail}(0) \ldots \texttt{fail}(9)$ have already been computed:

	X	Y	X	X	Y	X	X	Y	A	X
fail	-1	0	0	1	1	2	3	4	5	?

Either (brute force method)

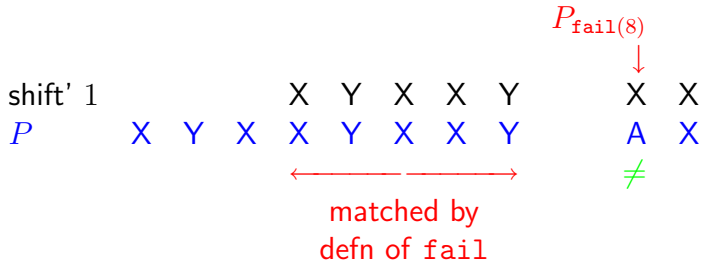
shift 1		X	Y	X	X	Y	X	X	Y	A
P	X	Y	X	X	Y	X	X	Y	A	X

shift 2			X	Y	X	X	Y	X	X	Y
P	X	Y	X	X	Y	X	X	Y	A	X

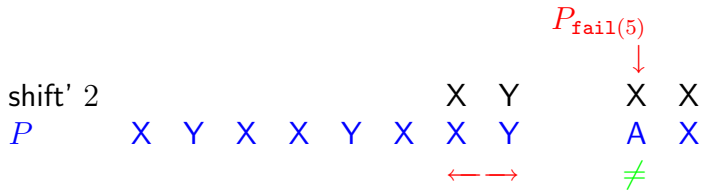
⋮ etc. until

shift 9										X
P	X	Y	X	X	Y	X	X	Y	A	X

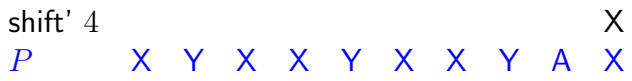
or (efficient method)
use the fact that $\texttt{fail}(8) = 5$



then try using $\texttt{fail}(5) = 2$



then try $\texttt{fail}(2) = 0$ and finally $\texttt{fail}(0) = -1$



So we get $\texttt{fail}(9) = 0$. Note that this does not guarantee that $P_0 = P_9$ as in this case X=X, it just guarantees that there is no match earlier.

7.2.3 Analyis of KMP Flowchart Construction

Let $m = P.length$. It would seem that algorithm 7.2.2 is in $O(m^2)$ because:

the `for` loop iterates $m - 1$ times;
the `while` loop iterates at most m times.

We will now be more exact and count character comparisons.

Successful comparison -	$P_r = P_{k-1}$
Unsuccessful comparison -	$P_r \neq P_{k-1}$

A successful comparison terminates the `while` loop, so there are at most $m - 1$ successful comparisons.

After very unsuccessful comparison r is decreased (since $\texttt{fail}(r) < r$), so we can bound the number of unsuccessful comparisons by determining how many times r can decrease. Note that:

(a) r is initially set to $\texttt{fail}(0) = -1$;

(b) r is increased by exactly 1 each subsequent time that the first statement in the body of the `for` loop is executed (for $k \geq 2$ the statement `r = fail[k-1];` effectively set r to $r + 1$;

(c) So r is incremented $m - 2$ times;

(d) At all times, $r \geq -1$.

Since r starts at -1 and is incremented by 1 a total of $m-2$ times, r cannot be decreased more than $m-2$ times. As r is decreased on each unsuccessful comparison, the number of unsusccessful comparisons is at most $m-2$.

The total number of comparisons is therefore at most $2m-3$.

KMP Match performs at most $2n$ comparisons, where $n = T.length$. (Excercise - show this). Thus the KMP pattern-matching algorithm, which comprises algorithms 7.2.1 and 7.2.2 performs $\Theta(n+m)$ comparisons in the worst case. This is much better than the $\Theta(m*n)$ worst case complexity of the straightforward solution (algorithm 7.1.1).

Some emprical studies have shown that both algorithms do roughly the same number of comparisons on average - for natural language text.

7.3 The Boyer-Moore Algorithm

Let P be a pattern of length m and T be a text of length n.

In both the previous algorithms, if P is found beginning at T_i, then each of the characters $T_0 \ldots T_i - 1$ has been examined.

The key insight of the Boyer-Moore algorithm is that some characters can be skipped over entirely.

7.3.1 Heuristic one - charJump

Rather than starting with the first character of the pattern we begin by comparing the last pattern character with the corresponding text character. If there is a match we will have to scan left through the pattern, but if there is not a match we may be able to slide the pattern much further foward.

Note that because (in turn) 'Y', 'W', '▽' are not anywhere in the pattern, we slide the pattern through its entire length after each mismatch.

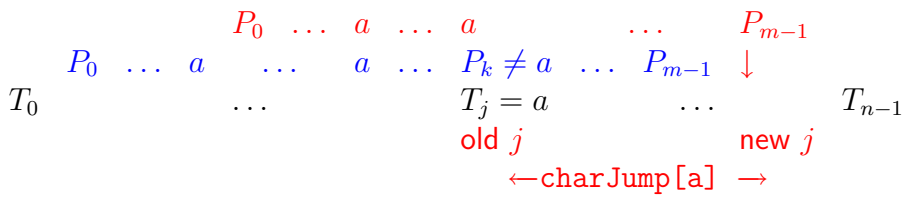
Suppose as usual that $P = P_0 \dots P_{m-1}$ is the pattern and $T = T_0 \dots T_{n-1}$ is the text.

If when comparing P_{m-1} with some T_j we find that T_j is known not to occur in the pattern, we can jump forward m places so that the pattern is just past T_j.

If at any stage $T_j = P_k$ and $k > 0$ we move to the left in both pattern and text. If we get to P_0 still matching the text we have located the pattern.

Any other mismatch $T_j \neq P_k$ where T_j is not in the pattern will result in the pattern sliding $k + 1$ places so that again the pattern is just past T_j.

If at some stage $T_j \neq P_k$ but T_j is in the pattern to the left of position k (possibly more than once) we slide the pattern the smallest distance to a matching position with T_j. For example,



Note that the jump to the next comparison position (new value of j) is to the end of P, and this may be longer than the distance through which P slides.

Let Σ denote the alphabet of P and T, i.e. the set of symbols which occur in P and T.

Definition 7.3.1 *charJump*

$$\forall \texttt{ch} \in \Sigma. \quad \texttt{charJump[ch]} = \min\{(m-1) - k | k = -1 \vee P_k = \texttt{ch}\}$$

So if ch is not in P, charJump[ch] $= m$.

Algorithm 7.3.1 (computeCharJump for Boyer-Moore)

Note that allowing $\Sigma =$ char, i.e. the full Unicode character set,will lead to large tables. If at all possible restrict Σ to e.g. ASCII.

```
void computeCharJumps (char P[], int charJump[])
{
   char ch;
   int k;
```

```
   for each ch in Σ
      charJump[(int)ch] = P.length;

   for (k=0; k<P.length; k++)
      charJump[(int) P[k]] = P.length-1 - k;
}
```

The complexity of this algorithm is $\Theta(|\Sigma| + m)$.

7.3.2 Heuristic two - matchJump

This heuristic is similar to the use of the table `fail` (or the `fail arrows`) in KMP except that now we scan the pattern from right to left so they are sometimes known as backward failure arrows.

```
P:  B A T S A N D C A T S
T:        ...       D A T S
                    ↑
                    T_j
```

Using `charJump` would result in

```
P:    B A T S A N D C A T S
T:        ...       D A T S
```

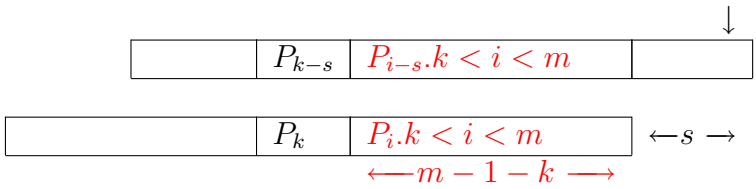
which is not much help.

If we know that P does not have another instance of the suffix just scanned ("A T S") we can slide P all the way past "A T S" in T. If P does have an earlier instance of this suffix we can line that up with the copy in T:

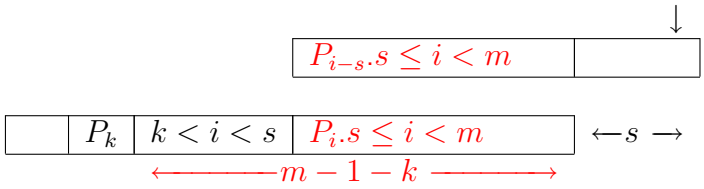
```
   P:              B A T S A N D C A T S
   T:      ...     D A T S
```

We shift the pattern the smallest amount which is consistent with what we know about the text and then restart at the last pattern character. This may involve the whole pattern moving beyond the current text position.

So, either the shift (s) is small enough to leave some pattern still opposite the current text position ($T_k \neq P_k$), i.e. $s \leq k$,



or else the pattern is is shifted right beyond the current text position ($s > k$)



Definition 7.3.2 (matchJump)

$$\forall k \in 0..m-1.\quad \texttt{matchJump}[k] = \min\{s+(m-1)-k \mid s \geq 1 \wedge (s > k \vee P_{k-j} \neq P_k) \wedge (\forall i.k < i < m \cdot i < s \vee P_{i-s} = P_i)\}$$

Constructing matchJump

Example $P =$ "WOWWOW"

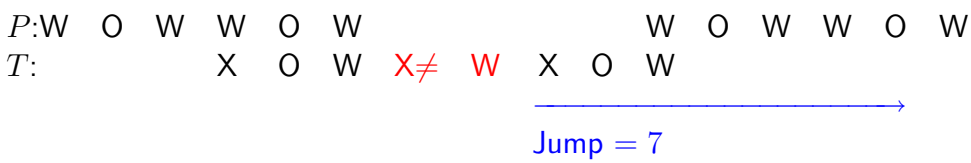
To find `matchJump[5]`

P:		W	O	W	W	O	W				W	O	W	W	O	W
T:							X	$\neq$	W						X	
															$\rightarrow$	
															Jump $= 1$	

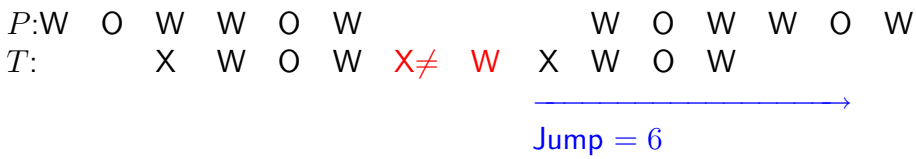
To find `matchJump[4]`

P:		W	O	W	W	O	W				W	O	W	W	O	W
T:						X	W	X$\neq$	O				X	W		
													$\longrightarrow$			
													Jump $= 3$			

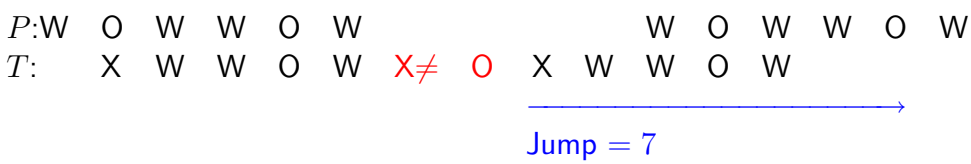
To find matchJump[3]



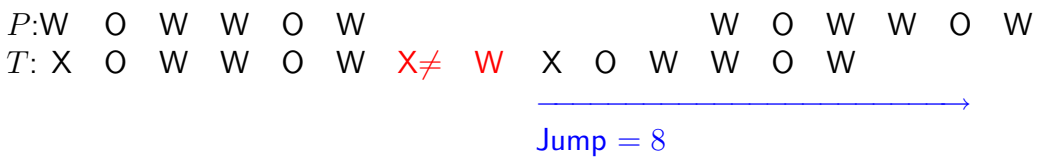
To find matchJump[2]



To find matchJump[1]



To find matchJump[0]



So matchJump = [8,7,6,7,3,1]

Algorithm 7.3.2 (Boyer-Moore scan Algorithm)

```
int BMMatch (char P[],char T[],
                int charJump[],int matchJump[])
{
    int j,k;
    j=P.length-1; k=P.length-1;
    while (j<T.length && k >= 0)
        if (T[j] == P[k])
        { j--; k--; }
        else
        {
            j = j + max(charJump[(int)T[j]],
```

```
                        matchJump[k]);
                k = P.length-1;
            }
        if (k == -1)
            return j+1; // Match found
        else
            return T.length; // No match
}
```

The description of the efficient calculation of `matchJump` is omitted and not examinable.

For binary strings BM (Boyer-Moore) performs ≈ 0.7 comparisons per character. Figure 7.1[1] summarizes the results of an empirical comparison of the three studied algorithms. For example when $m \geq 5$, BM examined only $1/4$ to $1/3$ of the text characters.

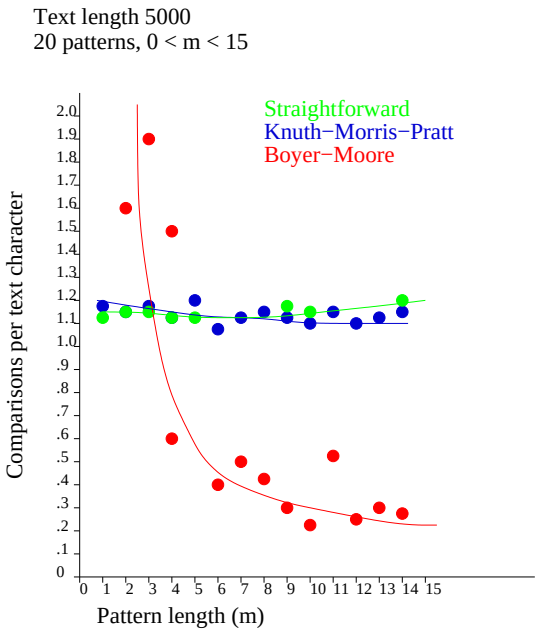


Figure 7.1: Comparison of algorithms

[1]From G. de V. Smit, "A Comparison of Three String Matching Algorithms", *Software - Practice and Experience*, Vol.12 (1982)

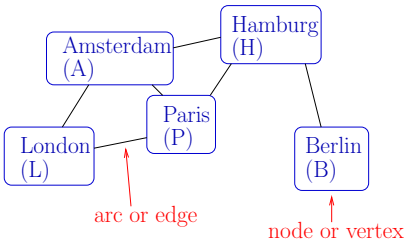
In all cases, BM is expected to be sublinear, i.e. bounded by cn for some $c < 1$.

If $m \leq 3$ BM does more copmarisons than KMP and the straightforward algorithm.

Chapter 8

Graph Algorithms

Example of a graph



8.1 Definitions

Definition 8.1.1 (Graph)

A graph is a pair (V, E) *where*

- V *is a finite non-empty set whose elements are called vertices (nodes);*
- E *is a set of subsets of* V*, each of order* 2*, called edges (arcs)*

For example

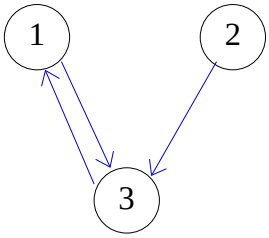
$$\left(\underbrace{\{A, B, H, L, P\}}_{V}, \underbrace{\left\{ \begin{array}{c} \{A, H\}, \{P, A\}, \{L, A\}, \\ \{L, P\}, \{P, H\}, \{H, B\} \end{array} \right\}}_{E} \right)$$

Definition 8.1.2 (Directed Graph)

A directed graph (or digraph) is a graph where each edge has a direction.

That is, each edge becomes a pair (u, v) rather than a set $\{u, v\}$.

Example



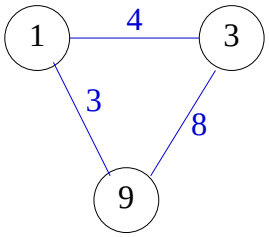
$(\{1,2,3\},\{(1,3),(3,1),(2,3)\})$

Definition 8.1.3 (Weighted Graph)

A weight graph (or, similarly a weighted digraph) is a graph where each edge is associated with a numeric value (weight).

Example

A weighted graph (or digraph) is a triple (V,E,W) where (V,E) is a graph (or digraph) and W is a function from E to $\mathbb{N}^+$. E.g.



$$\left(\underbrace{\{1,3,9\}}_{V},\underbrace{\left\{\begin{array}{l}\{1,3\},\\ \{1,9\},\\ \{3,9\}\end{array}\right\}}_{E},\underbrace{\left\{\begin{array}{l}\{1,3\}\mapsto 4,\\ \{1,9\}\mapsto 3,\\ \{3,9\}\mapsto 8\end{array}\right\}}_{W}\right)$$

Uses of Graphs

- Representing distances (or costs) between places;
- Networks (traffic, pipes, circuits) - and, in particular, computer networks.

Graph Problems

- What is the cheapest way to fly to ...?
- Which route involves the least travelling time/fewest stopovers etc?
- If a computer fails can messages still be sent between any of the remaining machines in the network?
- etc.

8.1.1 Definitions and Terminology

Definition 8.1.4 (Subgraph)

A subgraph of $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

Definition 8.1.5 (Complete)

A complete graph is a graph with an edge between each pair of vertices. uv is used to denote the edge $\{u, v\}$ (or $\{v, u\}$ or (u, v)).

Definition 8.1.6 (Path)

A path from u to w in G is a sequence of edges:

$$u_0u_1, u_1u_2, \ldots, u_{k-1}u_k$$

such that $u = u_0$, $u_k = w$ and $u_0, u_1, \ldots, u_k$ are all distinct (unless $u_0 = u_k$). The length of a path is the number of its consituent edges.

Definition 8.1.7 (Connected)

A graph is connected if, for every pair of vertices u and w, there is a path from u to w.

Definition 8.1.8 (Cycle)

A cycle in a graph is a path (expect that the first and last nodes are equal) $u_0u_1, u_1u_2, \ldots, u_{k-1}u_k$ with $k \geq 2$ and $v_k = v_0$.

Definition 8.1.9 (Acyclic)

A graph is acyclic if it has no cycles.

Definition 8.1.10 (Tree)

A tree may be represented as a connected acyclic graph.

Definition 8.1.11 (Adjacent)

Let $v, w \in V$. v is adjacent to w if and only if vw is in E.

In a tree T (i.e. a connected acyclic graph):

- There is at least one path between every pair of vertices. (Why?)
- There is at most one path between any pair of vertices. (Why?)

That is, there is exactly one path between every pair of vertices in T.

8.1.2 Algorithm: Minimum Spanning Tree

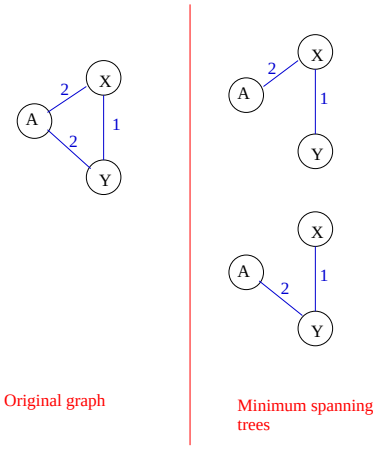
- A spanning tree of a (weighted) graph $G = (V, E, W)$ is a subgraph of G which
  - is a tree;
  - contains all of the vertices of G.
- The weight of a subgraph (tree) is the sum of the edges of the subgraph.
- A minimum spanning tree of G is a spanning tree of minimum weight.

Uses of the algorithm

Find the cheapest way to connect a set of terminals (cities, computers) by roads, wires, telephone lines etc.

Example

Note the the answer need not be unique. As a tree (which is connected) is being constructed as a subgraph, the inital graph G must be connected.



Strategy

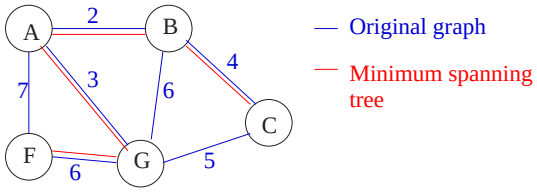
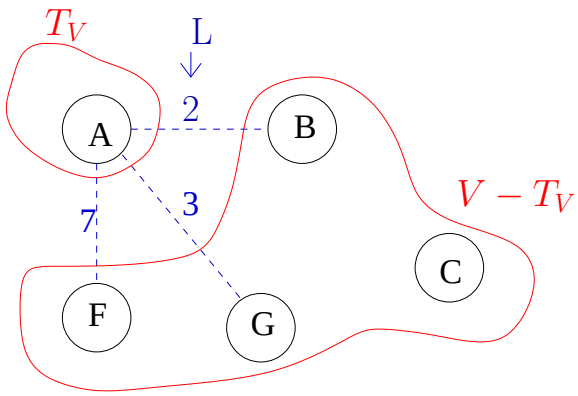


Figure 8.1: Example of Minimum Spanning Trees

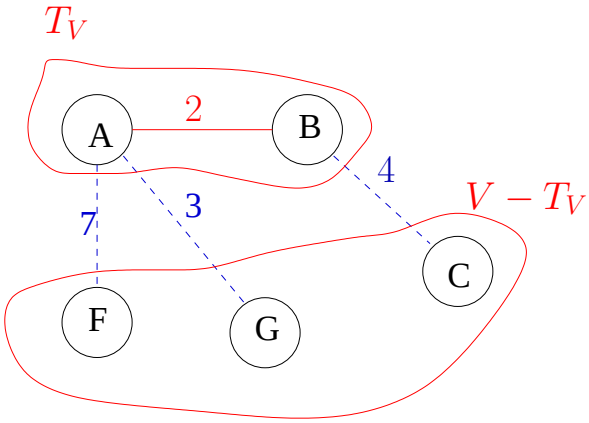
(See Figure 8.1) Build up the minimum spanning tree one vertex at a time.



Let

T_V	$=$	tree vertices (so far);
T_E	$=$	tree edges (so far);
$V - T_V$	$=$	vertices that can still be added;
$L(v)$	$=$	for each vertex $v \in T - T_V$, the edge of least weight from v to any w in T_V (if there is such an edge).

Next step



$L(F) = AF$
$L(G) = AG$
$L(C) = BC$

Algorithm 8.1.1 (Prim's Min'm Spanning Tree Alg'm)

INPUT: $G = (V, E, W)$
OUTPUT: T_E (the edges of a minimum spanning tree of G).

T_E = {};
T_V = *any vertex in* V, *say* u;
for (all $v \in (V - T_V)$)
    if ($uv \in E$)
        $L(v) = W(uv)$;
    else
        $L(v) = \infty$;
while ($T_V \neq V$)
{
    find a $w : L(w) = \min\{L(v) | v \in (V - T_V)\}$
    and denote the associated edge from T_V *by* e
    $T_E = T_E \cup \{e\}$;

```
    T_V = T_V ∪ {w};
    for (all v ∈ (V − T_V) )
        if (W(vw) < L(v))
            L(v) = W(vw);
}
```

This is a greedy algorithm - an algorithm for an optimization problem (to minimize or maximize some quantity) that makes locally optimal choices in each step. For some problems this strategy works, i.e. produces a globally optimal solution, but for many others it does not!.

8.1.3 Correctness of Min. Span. Tree Algorithm

Theorem 8.1.12

Let $T = (V, E_T)$ be a minimum spanning tree of $G = (V, E, W)$.

Let $T' = (T_V, T_E)$ be a graph such that

$$T_E \subseteq E_T \quad \text{and}$$
$$T_V = \{v | \exists uv \in T_E\}$$

Note that E_T is the set of edges in the final minimum spanning tree, whereas T_E is the current set of edges as the tree is constructed. We don't need a separate V_T as the final set of vertices is V.

IF

$$xy \in E \wedge x \in T_V \wedge y \in (V - T_V) \wedge$$
$$W(xy) \leq W(wz) \forall w \in T_V, z \in (V - T_V)$$
(i.e. x is in the tree so far, y is not, and no edge cheaper then xy links the tree so far with the remainder)

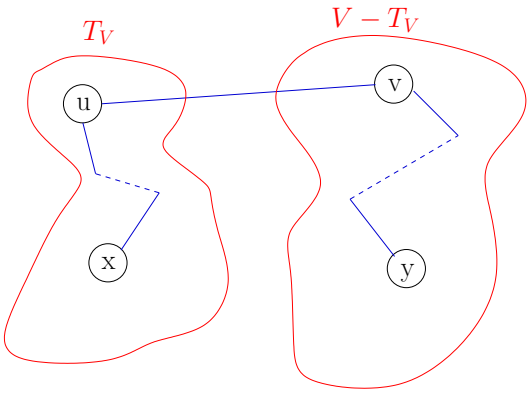
THEN

$$(T_V \cup \{y\}, T_E \cup \{xy\})$$

is a subgraph of a minimum spanning tree of G.

Proof

Suppose $xy \notin E_T$. There a path from x to y in T (since T is connected). Let uv be the first edge in that path with exactly one vertex in T_V:



Consider the tree T' with vertices V and edges

$$E_{T'} = E_T - \{uv\} + \{xy\}$$

T' is a spanning tree (why?)

Also, $W(E_{T'}) \leq W(E_T)$ as $W(xy) \leq W(uv)$.

Hence, $T' = (V, E_{T'})$ is a minimum spanning tree and the result is proved since $T_E \cup \{xy\} \subseteq E_{T'}$.

The correctness of the algorithm now follows by induction, since trivially in the initial condition $T_E = \{\} \subseteq E_T$ for any minimum spanning tree and the algorithm terminates with $V = T_V$.

8.1.4 Complexity of Min. Span. Tree Algorithm

Suppose that a graph has n nodes and m edges.

Let the basic operation be comparison of weights.

- The `while` loop body is executes $n - 1$ times (why?)
- The complexity of "searching L" depends on the underlying representation of nodes, edges and L.

Suppose that L is an unordered list. On the i^{th} iteration of the `while` loop:

$$|T_V| = i$$
$$|V - T_V| = n - i$$

$\therefore$ the minimum value in L (which is of size $n - i$) can be found in $n - i - 1$ comparisons.

The update of L in the i^{th} iteration required $n - i$ comparisons.

So the total number of comparisons

$$\approx 2\sum_{i=1}^{n-1}(n-i)$$

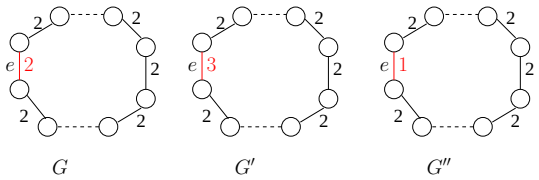
$$= 2\sum_{i=1}^{n-1} i$$

$$= (n-1)n \in \Theta(n^2)$$

Lower Bound

Suppose that there is a purported minimum spanning tree algorithm which does not make a comparison involving the weight of every edge.

Consider the "necklace" graph G consisting of a linked list whose first and last elements are the same and where each edge has a weight of 2. A minimum spanning tree is obtained by deleting any edge. Suppose that in calculating the minimum spanning tree the algorithm ignores the weight of edge e.



1. If e is in the proposed minimum spanning tree, apply the algorithm to G', in which the weight of e is 3. As the algorithm ignores the weight of e it must gave the same spanning tree, i.e. including e, which is the one edge to delete.

2. If e is not in the proposed minimum spanning tree, apply the algorithm to

G'', in which the weight of e is 1. The returned tree must exclude e which this time is the one edge which has to be in the tree.

Therefore, the algorithm is incorrect, and any correct algorithm must compare the weight of every edge in G, and so a lower bound for the minimum spanning tree problem is $m/2$ operations (each comparison involves two edges).

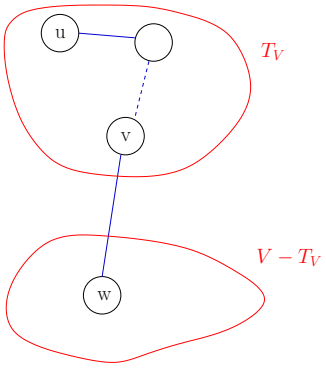
Note that if G is fully connected, then it has $\frac{n(n-1)}{2}$ edges.

8.2 Shortest Paths

What are the shortest (i.e. lowest sum of weights) paths from a node u to every node connected to u in the undirected graph $G = (V, E, W)$?

Note: The shortest path from u to v does not necessarily correspond to a path from u to v on a minimum spanning tree of G (for example, see the path AF in Figure 8.1).

A greedy algorithm (similar to Prim's algorithm) can be devised with two complementary sets of node T_V and $V - T_V$ but where L is used to record the minimum path distance from u (in T_V) to each w in $V - T_V$ where the path is restricted to have at most one node in $V - T_V$.



8.2.1 Dijkstra's Shortest Path Algorithm

INPUT: $G = (V, E, W)$ and a node $u \in V$.
OUTPUT: L, where $L(x)$ is the shortest distance along a path from u to x.

```
for (all w ∈ V)
    if (w == u)
        L(w) = 0;
```

```
    else if (uw ∈ E)
        L(w) = W(uw);
    else
        L(w) = ∞;
T_V = {};
while (T_V ≠ V)
{
    find y ∈ V − T_V such that
        L(y) = min{L(x)|x ∈ (V − T_V)}
    T_V = T_V ∪ {y};
    for (all x ∈ (V − T_V))
        if (yx ∈ E && L(x) > L(y) + W(yx))
            L(x) = L(y) + W(yx);
}
```

8.2.2 Correctness of Shortest Path Algorithm

Suppose that at the start of the `while` statement body $L(v) =$

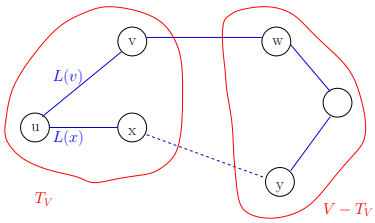
$$\begin{cases} \text{shortest distance from } u \text{ to } v & v \in T_V \\ \min\{L(x) + W(xv) | x \in T_V\} & v \notin T_V \wedge \exists x \in T_V \cdot xv \in E \\ \infty & v \notin T_V \wedge \nexists x \in T_V \cdot xv \in E \end{cases}$$

Thus, in particular, L contains the shortest distances from u to all nodes in T_V.

We claim that after the while statement body this is still true (i.e. it is an invariant of the loop).

Let y be the node added to T_V by the algorithm, so that

- $y \in V - T_V$
- $L(y) = \min\{L(z) | z \in (V - T_V)\}$
- For some $x \in T_V, L(y) = L(x) + W(xy)$ (why?)



Then a shortest path from u to y goes through x.

Proof
Suppose that there is a shorter path p whose first element in $V - T_V$ is w.

$$\begin{array}{rcll} \text{But } L(y) & = & L(x) + W(xy) & \\ & \leq & L(w) & \text{By choice of } y \\ & \leq & L(v) + W(vw) & \text{By inductive hypothesis} \\ & & & \\ & \leq & \text{cost of } p & \end{array}$$

Contradiction

$\therefore L(y)$ is the shortest distance from u to y.
This proves that the first component of the invariant holds.

Exercise
Prove the remainder of the invariant and complete the inductive proof.

Complexity of Dijsktra's Shortest Path Algorithm

Like the minimum spanning tree algorithm, the complexity of the shortest path algorithm is $\Theta(n^2)$.

8.3 Representation of Graphs

Label the set V by $\{0, 1, \ldots, |V| - 1\}$. The adjacency matrix of a graph $G = (V, E)$ is an $n \times n$ matrix A where $n = |V|$ and

$$a_{ij} = \begin{cases} 1 & ij \in E, 0 \leq i, j < n \\ 0 & \text{otherwise} \end{cases}$$

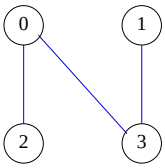
Similarly, the adjacency matrix of a weighted graph $G = (V, E, W)$, where $|V| = n$, is

$$a_{ij} = \begin{cases} W(ij) & ij \in E \\ c & \text{otherwise} \end{cases}$$

where c is a problem-dependent constant, usually viewed as representing ∞, to show that the nodes i and j are not directly connected.

Example

Let $G =$

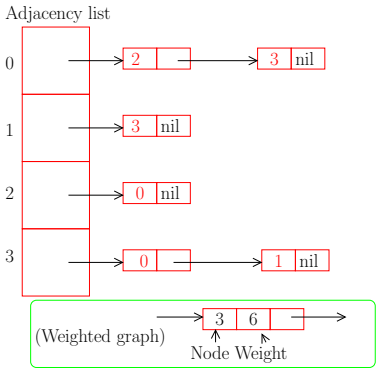


The adjacency matrix A for $G =$

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

The adjacency matrix for an undirected graph is symmetric and so only the upper diagonal needs to be stored.

An adjacency matrix will, in general, be sparse. An alternative representation of $G = (V, E)$ is a list (or set, or array) of linked lists (sometimes called an ADJACENCY LIST structure). Each node on a linked list represents a linked vertex of G. For weighted graphs the weight of each edge is also stored.



This can be represented in Java by:

```
class EdgeNode
```

```
{
    int vertex,weight;
    EdgeNode link;

    EdgeNode (int vertex, int weight,
                    EdgeNode link)
    {
        this.vertex = vertex;
        this.weight = weight;
        this.link = link;
    }

class ALS
{
    EdgeNode edgeNodes[];
    int maxVertices;
    ALS (int maxVertices)
    {
        this.maxVertices = maxVertices;
        edgeNodes = new EdgeNode[maxVertices];
        for (int v=0; v<maxVertices; v++)
            edgeNodes[v] = null;
    }
}
```

8.4 Traversing Graphs

Many graph algorithms require a systematic method of visiting each of the vertices of the graph. Two useful methods are:

Depth First Search Follow a path as far as possible. At a dead end, backtrack to the first point with a different path to follow (if there is one);

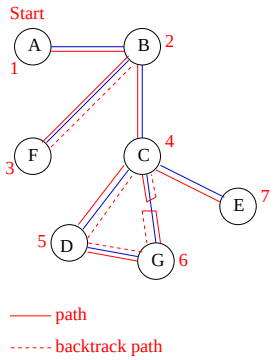
Breadth First Search Vertices are visited in order of increasing distance from the start vertex.

Notation

Let $A(v)$ denote the set of vertices that are adjacent to v in G. $A(v)$ can easily be determined from the adjacency list representation of G.

8.4.1 Depth First Search

Example



Algorithm 8.4.1 (Depth First Search)

INPUT:

- A graph $G = (V, E)$, where $V = \{0, 1, \ldots, n-1\}$, represented by an adjacency list;
- A start node $u \in V$.

Note that the graph will will only be traversed if it is connected.

OUTPUT: A list `index`: $V \to \{1 \ldots n\}$ which indicates the order in which the vertices have been visited.

```
int visitCount;
int index[];

void DFS (int v)
{
    index[v] = visitCount;
    // v is the visitCount^th node visited

    // Now process nodes connected to v
    visitCount++;
    for (all w ∈ A(v))
        if (index[w] == 0)
            // w has not been visited
            DFS (w);
}
```

```
int [] depthFirstSearch (int u)
{
    visitCount = 0;
    index = new int[maxVertices];
    for (all w ∈ V)
        index[w] = 0;
    DFS (u);
    return index;
}
```

As with QuickSort, the recursive structure of this algorithm can be represented using an explicit stack.

8.4.2 Complexity of Depth First Search

Basic Operation - comparing and updating elements of `index`;

Initialisation

n steps.

Procedure

For each node in V `DFS` is called at most once (why?)

The amount of work done for a node w is one more than the number of edges, i.e. $1 + d(w)$ where the degree (number of edges) of w is $d(w)$.

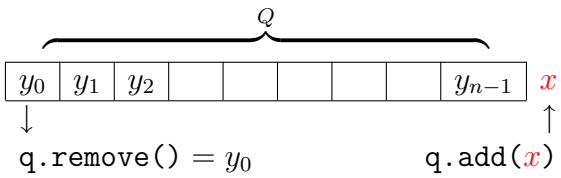
So the total amount of work done is

$$\sum_{i=0}^{n-1}(1 + d(i)) + n \leq 2m + 2n$$ why?

which is $\Theta(\max(n, m))$.

8.4.3 Breadth First Search

Recall the operations available for a queue:



The most recent element added is the last element extracted.

Algorithm 8.4.2 (Breadth First Search)

INPUT:

- A graph $G = (V, E)$, where $V = \{0, 1, \ldots, n-1\}$, represented by an adjacency list;
- A start node $u \in V$.

As above, the graph will will only be traversed if it is connected.

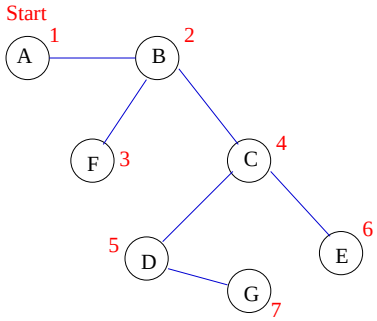
OUTPUT: A list index: $V \to \{1 \ldots n\}$ which indicates the order in which the vertices have been visited.

```
int index[];

void BFS (int u)
{
    int visitCount = 0;
    Queue q = new Queue();// empty queue
    int x;
    index[u] = visitCount++;
    q.add (u);
    while (q.length() > 0)
    {
        x = q.remove();
        for (all w ∈ A(x))
            if (index[w] == 0)
                {
                    index[w] = visitCount++;
                    q.add (w);
                }
    }
}
```

```
int [] breadthFirstSearch (int u)
{
    index = new int[maxVertices];
    for (all w ∈ V)
        index[w] = 0;
    BFS (u);
    return index;
}
```

Example



Queue

$[] \rightarrow [A] \rightarrow [B] \rightarrow [F, C] \rightarrow [C] \rightarrow [D, E] \rightarrow [E, G] \rightarrow [G] \rightarrow []$

Chapter 9

NP-Complete Problems

9.1 Sample Problems

9.1.1 Graph Colouring

A colouring of a graph $G = (V, E)$ is a mapping $C : V \to S$ where S is a finite set (of "colours") such that if $uv \in E$ then $C(u) \neq C(v)$, i.e. adjacent vertices do not have the same colour.

The chromatic number of G, $\chi(G)$, is the smallest number of colours needed to colour G, the smallest k such that there is a colouring of G with $|C(V)| = k$.

Decision Problem - Given G and a positive integer k, is there a colouring of G using at most kK colours?

Optimization Problem - Given G, determine $\chi(G)$ and produce an optimal colouring.

Graph colouring is an abstraction of certain types of scheduling problem. For example:

- Exams are to be scheduled during one week;
- Three exam time slots each day giving a total of 15 slots;
- Exams for some pairs of modules must be at different times, e.g. *Algorithms*, *Innovation*.

Let V be the set of modules, and let E be the set of pairs of modules whose exams must not be at the same time. The exams can be scheduled in the 15 time slots if and only if the graph $G = (V, E)$ can be coloured with 15 colours.

9.1.2 Job Scheduling With Penalties

Suppose that n jobs $J_1, J_2, \ldots, J_n$ are to be executed one at a time.

$t_1, t_2, \ldots, t_n$	execution times
$d_1, d_2, \ldots, d_n$	deadlines (measured from start of first job)
$p_1, p_2, \ldots, p_n$	penalties for missing deadlines
t_i, d_i, p_i	are all $\in \mathbb{N}^+$

A schedule for the jobs is a permutation Π of $\{1, 2, \ldots, n\}$ where $J_{\Pi(1)}$ is the job done first, $J_{\Pi(2)}$ is the job done second, etc. The total penalty for a particular schedule Π is:

$$P_\Pi = \sum_{j=1}^{n} \left\{ (t_{\Pi(1)} + \cdots + t_{\Pi(j)}) > d_{\Pi(j)} ? P_{\Pi(j)} : 0 \right\}$$

Decision Problem - Given, in addition to the inputs described, an nonnegative integer k, is there a schedule with $P_\Pi \leq k$?

9.1.3 Subset Problem

Given $C \in \mathbb{N}^+$ and n objects whose sizes are positive integers $b_1, b_2, \ldots, b_n$, is there a subset of objects whose sizes add up to exactly C?

9.1.4 CNF-Satisfiability

A logical expression in CNF (Conjunctive Normal Form) is a sequence of clauses separated by the boolean operator $\wedge$, e.g.

$$\underline{(p \vee q \vee s)} \wedge \underline{(\overline{q} \vee r)} \wedge \underline{(\overline{p} \vee r)} \wedge \underline{(\overline{r} \vee s)}$$

Four clauses (underlined).

Decision Problem - Is there a truth assignment for the variables in the expression so that the expression has the value true?

9.1.5 Other Problems

1. Bin Packing
2. Knapsack Problem
3. Hamiltonian Paths
4. Travelling salesperson problem

See [1]

[1] Garey, Johnson (1979), "Computers and Intractability, A Guide to the Theory of NP-Completeness", W.H. Freeman, San Francisco

9.2 The Class P

None of the algorithms known for the problems in §9.1 is guaranteed to run in a "reasonable" amount of time.

An algorithm is said to be polynomial-bounded if its worst-case complexity is bounded by a polynomial function p of the input size n. For each input of size n the algorithm terminates after at most $p(n)$ steps, e.g. $n^{27} + 41n^3 + 63$.

A problem is said to be polynomial-bounded if there is a polynomial bounded algorithm for it.

Definition 9.2.1 (P)
P is the class of decision problems (yes/no problems) that are polynomial-bounded.

Note that not every problem in P has an "acceptably efficient" algorithm. However, if a problem is not in P then it is intractable.

It is widely believed but not proved that the the problems described in §9.1 are not in P.

9.3 Overview of the Class NP

Consider three problems:

1. Shortest Path;
2. Travelling salesman (A salesman starts his own city, visits another $n-1$ cities, and returns home, taking the shortest overall route);
3. Towers of Hanoi.

All three may be viewed as optimum path finding problems.

Shortest Path - we can can avoid searching all paths by using a greedy algorithm;

Travelling Salesman - open problem - it is not known whether there is an efficient way of solving it - can we avoid searching "many" paths? A problem of size n has n edges.

Towers of Hanoi - CANNOT be solved efficiently. Even if we correctly "guess" the correct way to move the discs, a problem of size n requires $O(2^n)$ moves. Even to check the solution requires exponential time.

NP characterises Shortest Path and Travelling Salesman, but not Towers of Hanoi.

9.4 The Class NP

NP is the class of Decision Problems for which a given proposed solution for a particular input can be checked "quickly" (i.e. in polynomial time) to see if it is a solution.

The formal definition of NP uses the notion of nondeterministic algorithms, which are defined as follows:

Definition 9.4.1 *A nondeterministic algorithm has two phases:*

1. Nondeterministic Phase - some arbitrary string of characters, S, is written at some designated place in memory. Each time the algorithm is run the string may be different. (This string may be thought of as a "guess" of the solution).

2. Deterministic Phase - a deterministic (repeatable) algorithm begins execution. In addition to the decision problem's original input, this algorithm may read S, or it may ignore S. Eventually it halts with an output of yes or no, or it may go into an infinite loop and never halt. (This is the checking phase - checking S to see if it is indeed a solution to the decision problem).

The number of steps carried out during one execution of a nondeterministic algorithm is the sum of the number of steps of the two phases.

Nondeterministic algorithms do not necessarily produce the same output each time they are run with the same input.

> A nondeterministic algorithm's answer for an input x is yes if and only if there is some execution of the algorithm that gives a yes output.

9.4.1 Example - Graph Colouring

Problem

Determine whether a given graph G is k-colourable.

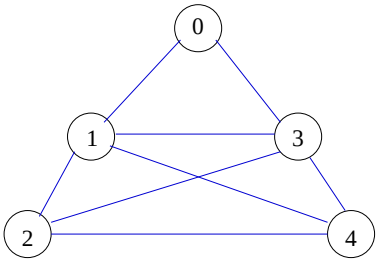
Nondeterministic Phase

Write some string S which is interpreted as a proposed colouring, e.g. $C_0C_1C_2\ldots C_{n-1}$ representing a list of colours to be assigned to vertices $v_0, v_1, v_2, \ldots, v_{n-1}$ respectively.

Deterministic Phase

- Check that there are n coloured vertices;
- scan the list of edges to that the two vertices incident with each edge have different colours
- count the distinct colours used and check that there are at most k.

Example



INPUT:

k	4,
No. of vertices	5,
Edges of G	{0,1}, {0,3}, {1,3}, {1,2}, {2,4}, {1,4}, {2,3}, {3,4}

Denote colours by letters:
B(blue),R(red),G(green),O(orange), Y(yellow).

S	Output	Reason
RGRBG	No	v_1,v_4 connected but both green
RGRB	No	Not all vertices (i.e. not v_4) coloured
RBYGO	No	More than $k = 4$ colours
RGRBY	Yes	
R%*,G	No	Bad syntax
RRGBY	No	v_0,v_1 both red

Definition 9.4.2 (NP)
NP is the class of decision problems for which there is a polynomial-bounded nondeterministic algorithm.

Theorem 9.4.3
All the decision problems described in §9.1 are in NP.

For example, given an assignment of truth value to variables, a CNF expression (string of size n) can be evaluated in $O(n)$.

Theorem 9.4.4
$P \subseteq NP$

Proof:

A deterministic algorithm for a decision problem is a special case of a nondeterministic algorithm. If A is a deterministic algorithm create a nondeterministic algorithm be letting A be its second phase. A ignores what was written by the first phase and proceeds with its computation. A nondeterministic algorithm can do zero steps in its first phase (writing a null string), so if A runs in polynomial time then the nondeterministic algorithm with A as its second phase can also run in polynomial time and give the same answer. Q.E.D.

Does $P = NP$ or is P a proper subset of P?

9.5 A Deterministic Interpretation of an NP Algorithm

One way of executing an NP algorithm A on a conventional (deterministic) machine is:

1. Generate all possible strings S_i where $i \in I$;
2. For each string, carry out the checking phase of A.

Example - Graph Colouring

Suppose that only strings of length n (number of nodes) made up of arrangements of k colours are generated.

For each string we must carry out the edge test:

$$uv \in E \Rightarrow \texttt{colour}(u) \neq \texttt{colour}(v)$$

If $|E| = m$ then the total number of edge checks is

$$\underbrace{k^n}_{\text{No. of strings}} * \underbrace{m}_{\text{edge tests per string}}$$

This algorithm $\notin P$!

Can we solve this problem without generating all possible solutions (place it in P) or demonstrate that this can't be done (place it in $NP - P$ and therefore $P \neq NP$)?

9.6 Polynomial Reductions

Definition 9.6.1 (NP-complete (informal))
(NPC) - Problems that are the hardest in NP. If there were a polynomial-bounded algorithm for any NP-complete problem, there would be a polynomial-bounded algorithm for every NP problem.

The problems described in §9.1 are all NP-complete. The formal definition of NP-complete uses reductions (transformations) of one problem to another.

Suppose we want to solve a problem Π_1 (where $\Pi_1 : I_1 \rightarrow \{\text{Yes,No}\}$) and we have an algorithm for a problem Π_2 (where $\Pi_2 : I_2 \rightarrow \{\text{Yes,No}\}$). Suppose we have a function $T : I_1 \rightarrow I_2$ that takes an input x for Π_1 and produces and input $T(x)$ for Π_2 such that the correct answer for Π_1 on x is "Yes" if and only if the correct answer for Π_2 on $T(x)$ is "Yes", i.e.

$$\forall x \in I_1 \cdot \Pi_1(x) = \Pi_2(T(x))$$

By composing T and the algorithm for Π_2 we have an algorithm for Π_1. T has reduced Π_1 to Π_2 (see Fig. 9.6).

Example

Π_1: Given n boolean variables, does at least one of them have the value `true` ($\equiv$ boolean `or`)

Π_2: Given n integers, is their maximum positive (i.e. > 0)?

Let $T(x_0, x_1, \ldots, x_{n-1}) = y_0, y_1, \ldots, y_{n-1}$

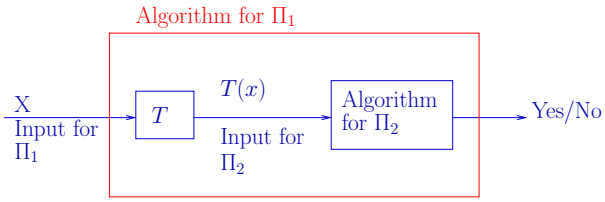


Figure 9.1: Reduction of Π_1 to Π_2

where

$$y_i = \begin{cases} 1 & x_i = true \\ 0 & x_i = false \end{cases}$$

Definition 9.6.2 (Polynomial Reduction)

Let T be a function from the domain of a decision problem Π_1 to the domain of a decision problem Π_2.

T is a polynomial reduction (transformation) from Π_1 to Π_2 if

1. T can be computed in polynomial time;
2. $\forall$ input x for Π_1, the correct answer for Π_2 on $T(x)$ is the same as the correct answer for Π_1 on x.

Definition 9.6.3 (Polynomially Reducible)
Π_1 is polynomially reducible to Π_2 iff $\exists$ a polynomial reduction from Π_1 to Π_2. We write $\Pi_1 \propto \Pi_2$ (Π_1 is reducible to Π_2).

Theorem 9.6.4
If $\Pi_1 \propto \Pi_2$ and Π_2 is in P then Π_1 is in P.

See Baase

Definition 9.6.5 (NP-complete)
A problem Π is NP-complete if

- *it is in NP;*
- *for every other problem Π' in NP, $\Pi' \propto \Pi$. (It is in this sense that NP-complete problems are the hardest in NP)*

Theorem 9.6.6
If any NP-complete problem is in P, then $P = NP$.

Proof

Let $\Pi \in NPC$ and $\Pi \in P$

Then (by Definition 9.6.5 of NPC)

$$\forall \Pi' \in NP \cdot \Pi' \propto \Pi \wedge \Pi \in P$$

Then from Theorem 9.6.4

$$\forall \Pi' \in NP \cdot \Pi' \in P$$

i.e. $P = NP$.

Theorem 9.6.7 (Cook's Theorem)
The CNF-satisfiability problem is NP-complete.

Not proved here

To show that a problem is NP-complete choose some problem Π' which is known to be NP-complete and reduce Π' to Π.

Since Π' is NP-complete, all problems in $NP \propto \Pi'$. Thus is we show that $\Pi' \propto \Pi$ then all problems in $NP \propto \Pi$ and so Π is NP-complete.

Theorem 9.6.8
All the problems in §9.1 are NP-complete.

Not proved here

9.7 Examples

9.7.1 SAT (CNF) $\propto$ 3SAT

SAT is the CNF satisfiability problem and 3SAT is CNF satisfiability where each clause has exactly 3 literals e.g. in

$$(A \vee \overline{B} \vee C) \wedge (D \vee E \vee \overline{A}) \ldots$$

$(A \vee \overline{B} \vee C)$ is a clause, and A and $\overline{B}$ are literals.

Not proved here

9.7.2 3SAT $\propto$ Vertex Cover

Definition 9.7.1 (Vertex Cover)
A vertex cover (of size k) of a graph $G = (V, E)$ is a subset $V' \subseteq V$ such that $|V'| = k$ and for every edge $uv \in E$ either $u \in V'$ or $v \in V'$.

Vertex Cover Problem: Given a graph $G = (V, E)$ and a positive integer $k \leq |V|$ is there a vertex cover for G of size k or less?

Consider any instance of 3SAT (SC). Let
$U = \{u_0, u_1, \ldots, u_{n-1}\}$ denote the variables of SC and

$C = \{c_0, c_1, \ldots c_{m-1}\}$ denote the clauses of SC.

Example

If SC $= (P \vee \overline{R} \vee \overline{S}) \wedge (\overline{P} \vee Q \vee \overline{S})$ then

$U = \{P, Q, R, S\}, C = \{\{P, \overline{R}, \overline{S}\}, \{\overline{P}, Q, \overline{S}\}\}$

Problem

To reduce 3SAT to vertex cover we need to construct a graph $G = (V, E)$ and find a k such that SC is satisfiable IFF ($\Leftrightarrow$) G has a vertex cover of size k or less.

Solution

Construct G as follows:

1. Truth Setting Vertices $(T_i, 0 \leq i < n)$
   For each variable u_i include u_i and $\overline{u_i}$ as vertices and $(u_i, \overline{u_i})$ as an edge.
2. Clause Setting Vertices $(S_j, 0 \leq j < m)$
   For each clause c_j include vertices $a1[j], a2[j]$ and $a3[j]$ and edges $(a1[j], a2[j])$, $(a2[j], a3[j])$, $(a3[j], a1[j])$
3. Problem Specific Edges $(P_i, 0 \leq i < m)$
   Let the variables of the clause c_j be x_j, y_j, z_j. Include the edges $(a1[j], x_j)$, $(a2[j], y_j)$, $(a3[j], z_j)$.

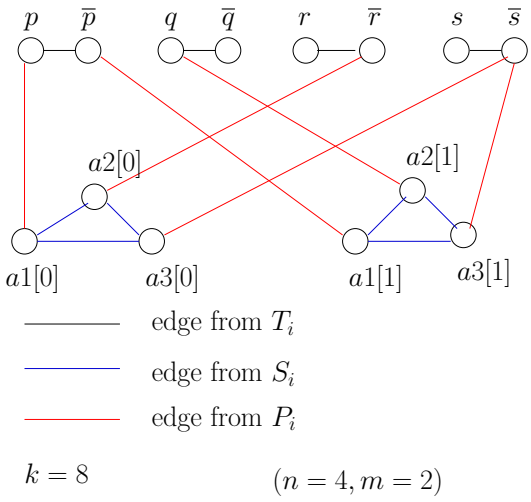
4. k
   Let $k = n + 2m$.

Example

For the previous example

$$(P \vee \overline{R} \vee \overline{S}) \wedge (\overline{P} \vee Q \vee \overline{S})$$

the following graph is contructed:



$k = 8$ $(n = 4, m = 2)$

Why can the construction be generated in polynomial time?

Proof of Correctness

3SAT ⇒ Vertex Cover

Suppose that the proposition is satisfiable. Let $t : U \to \{T, F\}$ be an assignment that solves SC. V', the required vertex cover of G is constructed as follows:

- T_i component
  For $0 \leq i < n$:
  `if` $(t(u_i) = T)$
  `then` $u_i \in V'$
  `else` $\overline{u_i} \in V'$
  Thus, each edge introduced through T_i is covered, and n vertices are added to V'.

- P_j component
  For each P_j, consider the three edges. The assignment t satisfies clause

c_j, so at least one of these three edges, going to $a1[j], a2[j]$ or $a3[j]$, has already been covered. Include the other two members of $S_j = \{a1[j], a2[j], a3[j]\}$ in V'. This will also cover the edges linking the members of S_j and adds a further $2m$ vertices to V'

The total number vertices in V' is $n + 2m = k$, as required.

Vertex Cover $\Rightarrow$ 3SAT

If V' is a vertex cover of G then it must contain at least one vertex from each T_i in order to cover the edges of T_i (n vertices) and at least two vertices from each S_j to cover the edges of S_j ($2m$ vertices). This gives a cover of size $k = n+2m$, so exactly one from each T_i must be included.

For each $u \in U$, set

$$t(u) = \begin{cases} \texttt{true} & u \text{ is in the cover} \\ \texttt{false} & \overline{u} \text{ is in the cover} \end{cases}$$

(Note that we have just seen that there are too few vertices available for both u and $\overline{u}$ to be in the cover).

Now, consider a clause c_j which corresponds to P_j whose edges must be covered. Two are covered by the vertices selected to cover S_j, $\therefore$ one must be covered by a vertex in u in some T_i.

The assignment t must satisfy clause c_j because $t(u)$ matches the corresponding literal in c_j.

As this holds for all clauses c_j, the original expression must be satisfiable.

Note:
A "quick" solution for any NP-complete problem would imply a "quick" solution for every NP problem.

If for any NP problem it could be proved that there is no quick solution, then there could be no quick solution for any NP-complete problem.

9.8 Size of Input

Consider the following problem:

Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? I.e. is n non-prime?

```
found = false;
j = 2;
while (!found && j < n)
    if (n % j == 0)
        found = true;
    else
        j++;
```

Is this problem in P?

The loop body is executed fewer than n times, worst-case is $\Theta(n)$. However, the problem of finding whether an integer is prime is not known to be in P - an apparent paradox?

The problem arises from our use of the variable name n in the (correct) assertion that the algorithm is $\Theta(n)$. When we say the testing whether an integer is prime is not known to be in P we measure the complexity against the size of the input. For many situations, e.g. the length of an input list or the number of edges in a graph, the size is an obvious measure but when the input is a single integer n what is the size of n?

For example, if $n = 90_{10} = 1011010_2$ the size of n is 7 bits. The size of n is propotional to $\lg n$.

We need to consider the size of the input rather than its value.

The input size s is $\lg n$ and the running time is $\Theta(n) = \Theta(2^s)$ i.e. the time is an exponential function of input size.

Worst-case complexity can be expressed in terms of a convenient variable, but to determine membership of P, NP it needs to be expressed in terms of input size.

For example, the complexity of Towers of Hanoi was given earlier as 2^n where n is the number of disks. To determine complexity class if the input to the algorithm is n the size is $s = \lg n$ and the complexity is 2^{2^s} - "even more" exponential.