Techniques for the Design of Algorithms

The first part of this course is concerned with:

Algorithmic Paradigms

That is:

General approaches to the construction of *efficient* solutions to problems.

Such methods are of interest because:

1. They provide templates suited to solving a broad range of diverse problems.
2. They can be translated into common control and data structures provided by most high-level languages.
3. The temporal and spatial requirements of the algorithms which result can be precisely analysed.

Over the next few lectures we shall examine the following paradigms:

1. Divide and Conquer.
2. Dynamic Programming
3. Greedy Method
4. Backtracking.

Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.

The choice of design paradigm is an important aspect of algorithm synthesis.

Basic Algorithm Analysis

Questions

1. How does one calculate the running time of an algorithm?
2. How can we compare two different algorithms?
3. How do we know if an algorithm is ‘optimal’?

1. *Count* the number of
basic operations

performed by the algorithm on the
worst-case input

A *basic operation* could be:

- An assignment
- A comparison between two variables
- An arithmetic operation between two variables.

The **worst-case** input is that input assignment for which the *most* basic operations are performed.

Simple Example:

n := 5;
loop
  get(m);
  n := n -1;
until (m=0 **or** n=0)

Worst-case: 5 iterations

Usually we are **not** concerned with the number of steps for a

single fixed case

but wish to estimate the running time in terms of the 'input size'.

```
get(n);
loop
   get(m);
   n := n -1;
until (m=0 or n=0)
```

Worst-case: n iterations

Examples of 'input size':

Sorting:

$n \equiv$ The number of items to be sorted;
Basic operation: Comparison.

Multiplication (of x and y):

$n \equiv$ The number of *digits* in x plus the number of digits in y.
Basic operations: single digit arithmetic.

Graph 'searching':

$n \equiv$ the number of nodes in the graph or the number of edges in the graph.

Counting the Number of Basic Operations

Sequence: P and Q are two algorithm sections:

$$Time(P; Q) = Time(P) + Time(Q)$$

Iteration:

```
while <condition> loop
    P;
end loop;
```

or

```
for i in 1..n loop
   P;
end loop
```

$$Time = Time(P) * (Worst\text{-}case\ number\ of\ iterations)$$

Conditional

```
if <condition> then
   P;
else
   Q;
end if;
```

$$Time = \begin{cases} Time(P) & \text{if } <condition>=\textbf{true} \\ Time(Q) & \text{if } <condition>=\textbf{false} \end{cases}$$

We shall consider recursive procedures later in the course.

Example:

```
for i in 1..n loop
   for j in 1..n loop
      if i < j then
         swop (a(i,j), a(j,i)); -- Basic operation
      end if;
   end loop;
end loop;
```

$$Time < n*n*1$$
$$= n^2$$

Asymptotic Performance

We are usually concerned with the growth in running time as the input size increases, e.g.
Suppose P, Q and R are 3 algorithms with the following worst-case run times:

$P : 2^n$

$Q : 5n^2$

$R : 100n$

n	P	Q	R
1	1	5	100
10	1024	500	1000
100	2^{100}	50,000	10,000
1000	2^{1000}	5 million	100,000

If each is run on a machine that executes one million (10^6) operations per second

n	P	Q	R
1	$1\mu s$	$5\mu s$	$100\mu s$
5	1 millisec	0.5 millisec	1 millisec
100	2^{70} years	0.05 secs	0.01 secs
1000	2^{970} years	5 secs	0.1 secs

Thus,

The growth of run-time in terms of n (2^n; n^2; n) is more significant than the exact constant factors (1; 5; 100)

'O'-notation

Let

$f : \mathbf{N} \rightarrow \mathbf{R}^+$

$g : \mathbf{N} \rightarrow \mathbf{R}^+$

Then

$$f(n) = O(g(n))$$

means that

There are values, n_0 and c, such that

$$f(n) \leq c * g(n)$$

whenever $n \geq n_0$.

Thus we can say that an algorithm has, for example,

$$\text{Run-time } O(n^2)$$

Examples

1. There is an algorithm (*mergesort*) to sort n items which has run-time $O(n \log_2 n)$.
2. There is an algorithm to multiply 2 n-digit numbers which has run-time $O(n^2)$.
3. There is an algorithm to compute the n'th Fibonacci number which has run-time $O(\log_2 n)$.

Ω-notation

To express the concept of an algorithm taking *at least* some number of steps

Ω-*notation*

can be used.

Again let,

$f : \mathbf{N} \rightarrow \mathbf{R}^+$

$g : \mathbf{N} \rightarrow \mathbf{R}^+$

$$f(n) = \Omega(g(n))$$

means that

There are values, n_0 and c, such that

$$f(n) \geq c * g(n)$$

whenever $n \geq n_0$.

Θ-notation

Again let,

$f : \mathbf{N} \rightarrow \mathbf{R}^+$

$g : \mathbf{N} \rightarrow \mathbf{R}^+$

Then if,

$$f(n) = O(g(n)) \textit{ and } f(n) = \Omega(g(n))$$

Then we write,

$$f(n) = \Theta(g(n))$$

In this case, $f(n)$ is said to be *asymptotically equal* to $g(n)$.

If $f(n) = \Theta(g(n))$ then algorithms with running times $f(n)$ and $g(n)$ appear to perform similarly as n gets larger.

Manipulating O- and Ω-notation

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

If $f(n) = O(g(n))$ then

$f(n) + g(n) = O(g(n))$

$f(n) + g(n) = \Omega(g(n))$

$f(n) * g(n) = O(g(n)^2)$

$f(n) * g(n) = \Omega(f(n)^2)$

Examples

Suppose $f(n) = 10n$ and $g(n) = 2n^2$

A. $f(n) = O(g(n))$

B. $10n + 2n^2 = O(n^2)$

C. $10n + 2n^2 = \Omega(n^2)$

D. $(10n) * (2n^2) = O(n^4)$

E. $(10n) * (2n^2) = \Omega(n^2)$

Divide-and-Conquer

This is a method of designing algorithms that (informally) proceeds as follows:

Given an instance of the problem to be solved, split this into several, smaller, sub-instances (*of the same problem*) independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance.

This description raises the question:

By what methods are the *sub-instances* to be *independently solved*?

The answer to this question is central to the concept of *Divide-&-Conquer algorithm* and is a key factor in gauging their efficiency.

Consider the following:

We have an algorithm, α say, which is known to solve all problem instances of size n in at most $c n^2$ steps (where c is some constant).

We then discover an algorithm, β say, which solves the same problem by:

1. Dividing an instance into 3 sub-instances of size $n/2$.
2. Solves these 3 sub-instances.
3. Combines the three sub-solutions taking $d\,n$ steps to do this.

Suppose our original algorithm, α, is used to carry out the 'solves these sub-instances' step 2.

Let

$T_\alpha(n) =_{def}$ *Running time of* α

$T_\beta(n) =_{def}$ *Running time of* β

Then,

$T_\alpha(n) = c\,n^2$ *(by definition of* α*)*

But

$$T_\beta(n) = 3\,T_\alpha(n/2) + d\,n$$
$$= (3/4)(c\,n^2) + dn$$

So if $dn < cn^2/4$ (i.e. $d < cn/4$) then β is *faster than* α

In particular for all large enough n, ($n > 4d/c = Constant$), β is faster than α.

This realisation of β improves upon α by just a constant factor.

But if the problem size, n, is large enough then

$$n > 4d/c$$
$$n/2 > 4d/c$$
$$\ldots$$
$$n/2^i > 4d/c$$

which suggests that using β instead of α for the '*solves these*' stage *repeatedly until the sub-sub-sub..sub-instances are of size* $n_0 \le (4d/c)$ will yield a still faster algorithm.

So consider the following new algorithm for instances of size n

```
procedure γ(n : problem size ) is
  begin
    if n ≤ n0 then
      Solve problem using Algorithm α;
    else
      Split into 3 sub-instances of size n/2;
      Use γ to solve each sub-instance;
      Combine the 3 sub-solutions;
    end if;
  end γ;
```

Let $T_\gamma(n)$ denote the running time of this algorithm.

$$T_\gamma(n) = \begin{cases} c\,n^2 & \text{if } n \le n_0 \\ 3\,T_\gamma(n/2) + dn & \textit{otherwise} \end{cases}$$

We shall show how relations of this form can be estimated later in the course. With these methods it can be shown that

$$T_\gamma(n) = O(n^{\log_2 3})\ \ P = O(n^{1.59..})$$

This is an asymptotic improvement upon algorithms α and β.

The improvement that results from applying algorithm γ is due to the fact that it maximises the savings achieved β.

The (relatively) inefficient method α is applied only to *"small"* problem sizes.

The precise form of a divide-and-conquer algorithm is characterised by:

I. The *threshold* input size, n_0, below which the problem size is not sub-divided.

II. The *size* of sub-instances into which an instance is split.

III. The *number* of such sub-instances.

IV. The algorithm used to combine sub-solutions.

In (II) it is more usual to consider the *ratio* of initial problem size to sub-instance size. In our example this was 2. The *threshold* in (I) is sometimes called the *(recursive) base value*.

In summary, the generic form of a divide-and-conquer algorithm is:

```
procedure D-and-C (n : input size) is
  begin
    if n ≤ n0 then
      Solve problem without further
      sub-division;
    else
      Split into r sub-instances
      each of size n/k;
      for each of the r sub-instances do
        D-and-C (n/k);
      Combine the r resulting
      sub-solutions to produce
      the solution to the original problem;
    end if;
  end D-and-C;
```

Such algorithms are naturally and easily realised as:

Recursive Procedures

in (suitable) high-level programming languages.

Example 1:

Binary Search

Consider the following problem: one has a directory containing a set of *names* and a telephone *number* associated with each name.

The directory is sorted by alphabetical order of names. It contains n entries which are stored in 2 arrays:

names (1..n)
numbers (1..n)

Given a *name* and the value n the problem is to find the *number* associated with the name.

We assume that any given input name actually *does occur* in the directory, in order to make the exposition easier.

The Divide-&-Conquer algorithm to solve this problem is the simplest example of the paradigm.

It is based on the following observation

Given a name, X say,

X occurs in the *middle* place of the *names* array

Or

X occurs in the *first* half of the *names* array. (U)

Or

X occurs in the *second* half of the *names* array. (L)

U (respectively L) are true *only if* X comes *before* (respectively *after*) that name stored in the *middle* place.

This observation leads to the following algorithm:

```
function search (X : name;
                 start, finish : integer)
                 return integer is
  middle : integer;
  begin
    middle := (start+finish)/2;
    if names(middle)=x then
      return numbers(middle);
    elsif X<names(middle) then
      return search(X,start,middle-1);
    else -- X>names(middle)
      return search(X,middle+1,finish);
    end if;
  end search;
```


Exercise: How should this algorithm be modified to cater for the possibility that a given name does not occur in the directory?

In terms of the generic form of divide-&-conquer algorithms, at which stage is this modification made?

We defer analysis of the algorithm's performance until later.

Example 2

Closest Pair

Input:

$$P = \{p_1, p_2, \ldots, p_n\}$$

where $p_i = (x_i, y_i)$.

A set of n points in the plane.

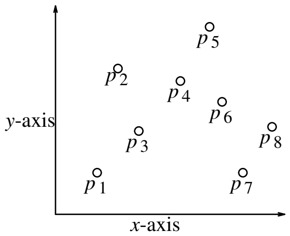
Output

The *distance* between the two points that are closest.

Note: The distance $\Delta(i, j)$ between p_i and p_j is defined by the expression:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

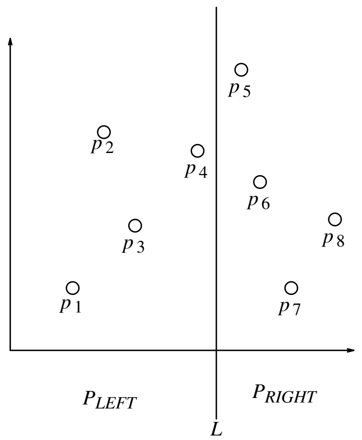
We describe a divide-and-conquer algorithm for this problem.



We assume that:

1. n is an exact power of 2, $n = 2^k$.
2. For each i, $x_i \le x_{i+1}$, i.e. the points are ordered by increasing x from left to right.

Consider drawing a vertical line (L) through the set of points P so that half of the points in P lie to the *left* of L and half lie to the *right* of L.



There are three possibilities:

I. The closest pair lie in P_{LEFT}.

II. The closest pair lie in P_{RIGHT}.

III. The closest pair contains:

One Point from P_{LEFT}

and

One Point from P_{RIGHT}

So we have a (rough) Divide-and-Conquer Method as follows:

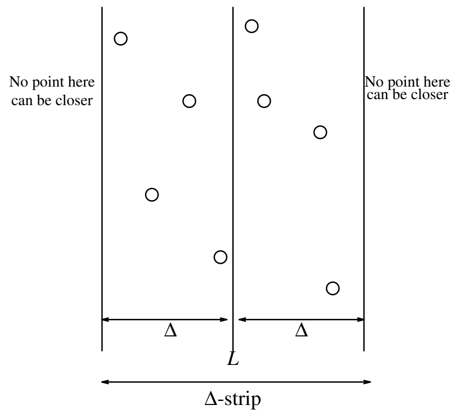

```
function closest_pair (P: point set; n: integer )
                    return float is
  Δ_LEFT, Δ_RIGHT : float;
  Δ : float;
  begin
    if n = 2 then
      return distance from p_1 to p_2;
    else
      P_LEFT := ( p_1, p_2, . . . , p_n/2 );
      P_RIGHT := ( p_n/2+1, p_n/2+2, . . . , p_n );
      Δ_LEFT := closest_pair ( P_LEFT, n/2 );
      Δ_RIGHT := closest_pair ( P_RIGHT, n/2 );
      Δ := minimum ( Δ_LEFT, Δ_RIGHT );
      --*****************************************
      Determine whether there are points p_l in
      P_LEFT and p_r in P_RIGHT with
      distance ( p_l, p_r ) < Δ. If there
      are such points, set Δ to be the smallest
      distance.
      --*****************************************
      return Δ;
    end if;
end closest_pair;
```


The section between the two comment lines is the 'combine' stage of the Divide-and-Conquer algorithm.

If there are points p_l and p_r whose distance apart is less than Δ then it must be the case that

I. The x-coordinates of p_l and p_r differ by at most Δ.

II. The y-coordinates of p_l and p_r differ by at most Δ.



The combine stage can be implemented by:

1. Finding all points in P_{LEFT} whose x-coordinate is at least $x_{n/2}-\Delta$.
2. Finding all points in P_{RIGHT} whose x-coordinate is at most $x_{n/2}+\Delta$.
3. Call the set of points found in (1) and (2) P_{strip}. and sort the s points in this in order of increasing y-coordinate. letting $(q_1, q_2, \ldots, q_s)$ denote the sorted set of points.

Then the combine stage of the algorithm consists of two nested **for** loops:


```
for i in 1..s loop
  for j in i+1..s loop
    exit when ( | x_i − x_j | > Δ or
                | y_i − y_j | > Δ);
    if distance( q_i, q_j ) < Δ then
      Δ := distance ( q_i, q_j );
    end if;
  end loop;
end loop;
```

Example 3

Integer Multiplication

Divide-and-Conquer may also be applied to problems other than those involving searching. The following problem should be familiar:

Input:

$\underline{x} = x_{n-1} x_{n-2} \cdots x_1 x_0$

$\underline{y} = y_{n-1} y_{n-2} \cdots y_1 y_0$

2 n-digit non-negative numbers represented in decimal.

Output:

$\underline{z} = z_{2n-2} z_{2n-3} \cdots z_1 z_0$

The $(2n)$-digit decimal representation of the product $x * y$.

Note: The algorithm below works for *any* number base, e.g. binary, decimal, hexadecimal, etc. We use decimal simply for convenience.

The classical primary school algorithm for multiplication requires $O(n^2)$ steps to multiply two n-digit numbers.

A *step* is regarded as a *single operation* involving two *single digit* numbers, e.g. 5+6, 3*4, etc.

In 1962, A.A. Karatsuba discovered an *asymptotically faster* algorithm for multiplying two numbers by using a divide-and-conquer approach.

The *values* (x and y) represented by

$\underline{x} = x_{n-1} x_{n-2} \cdots x_1 x_0$

$\underline{y} = y_{n-1} y_{n-2} \cdots y_1 y_0$

are clearly given by,

$$x = \sum_{i=0}^{n-1} (x_i) * 10^i \;\; ; \;\; y = \sum_{i=0}^{n-1} (y_i) * 10^i$$

So that, the result $z = x*y$ is

$$z = \sum_{i=0}^{2n-2} (z_i) * 10^i \;=\; \left(\sum_{i=0}^{n-1} x_i * 10^i\right) * \left(\sum_{i=0}^{n-1} y_i * 10^i\right)$$

For example:

$$581 = 5*10^2 + 8*10^1 + 1*10^0$$

$$602 = 6*10^2 + 0*10^1 + 2*10^0$$

$$581*602 = 349762 = 3*10^5 + 4*10^4 + \cdots$$

Now, suppose we let

$\underline{a} = x_{n-1} x_{n-2} \cdots x_{(n/2)+1} x_{n/2}$

$\underline{b} = x_{(n/2)-1} x_{(n/2)-2} \cdots x_1 x_0$

$\underline{c} = y_{n-1} y_{n-2} \cdots y_{(n/2)+1} y_{n/2}$

$\underline{d} = y_{(n/2)-1} y_{(n/2)-2} \cdots y_1 y_0$

Then if a, b, c, and d are the numbers whose decimal representations are $\underline{a}$, $\underline{b}$, $\underline{c}$, and $\underline{d}$, then

$$x = a*10^{(n/2)} + b \;\; ; \;\; y = c*10^{(n/2)} + d$$

For example, if $n=4$, $x = 1026$ and $y = 7329$ then a=10, b=26, c=73 and d=29. And,

$$x = 1026 = 10*10^2 + 26 = a*10^2 + b$$

$$y = 7329 = 73*10^2 + 29 = c*10^2 + d$$

From this we also know that the result of multiplying x and y (i.e. z) is

$$\begin{aligned} z = x*y &= (a*10^{n/2} + b) * (c*10^{n/2} + d) \\ &= (a*c)*10^n + (a*d + b*c)*10^{n/2} \\ &\quad + (b*d) \end{aligned}$$

e.g. 1026*7329 is

$$\begin{aligned} &= (10*73)*10^4 + (10*29 + 26*73)*10^2 \\ &\quad + (26*29) \\ &= 730*10^4 + 2188*10^2 + 754 = 7{,}519{,}554 \end{aligned}$$

The terms $(a*c)$, $(a*d)$, $(b*c)$, and $(b*d)$ are each products of

2 (n/2)-digit numbers.

Thus the expression for the multiplication of x and y in terms of the numbers a, b, c, and d tells us that:

1) Two single digit numbers can be multiplied immediately. (Recursive base: 1 step)

2. If $n > 1$ then the product of 2 n-digit numbers can be expressed in terms of *4 products of 2 (n/2)-digit numbers* (Divide-and-Conquer stage)

3. To calculate the result of multiplying x and y given the four products returned involves only *addition* (can be done in $O(n)$ steps) and multiplying by a *power of 10* (also can be done in $O(n)$ steps, since it only requires placing the appropriate number of 0s at the end of the number). (Combine stage).

(1-3) therefore describe a Divide-&-Conquer algorithm for multiplying two n-digit numbers represented in decimal.

However,

Moderately difficult: How many steps does the resulting algorithm take to multiply two n-digit numbers?

Karatsuba discovered how the product of 2 n-digit numbers could be expressed in terms of **three** products each of 2 $(n/2)$-digit numbers - instead of the **four** products that a naive implementation of the Divide-and-Conquer schema above uses.

This saving is accomplished at the expense of slightly increasing the number of steps taken in the ‘combine stage’ (Step 3) (although, this will still only use $O(n)$ operations).

Suppose we compute the following 3 products (of 2 $(n/2)$-digit numbers):

$$U = a * c$$
$$V = b * d$$
$$W = (a+b) * (c+d)$$

Then

$$a*d + b*c = W - U - V$$

and so

$$\begin{aligned} z &= x*y \\ &= (a*10^{n/2} + b) * (c*10^{n/2} + d) \\ &= (a*c)*10^{n} + (a*d + b*c)*10^{(n/2)} + b*d \\ &= U*10^{n} + (W - U - V)*10^{n/2} + V \end{aligned}$$

function *Karatsuba* ($\underline{x}$, $\underline{y}$: **n-digit integer**;
n : **integer**)
return (2n)-digit integer is
$\underline{a}$, $\underline{b}$, $\underline{c}$, $\underline{d}$: **(n/2)-digit integer**
$\underline{U}$, $\underline{V}$, $\underline{W}$: **n-digit integer**;
begin
if $n = 1$ **then**
return $x_0 * y_0$;
else
$\underline{a} := x_{n-1} \cdots x_{(n/2)}$;
$\underline{b} := x_{(n/2)-1} \cdots x_0$;
$\underline{c} := y_{n-1} \cdots y_{(n/2)}$;
$\underline{d} := y_{(n/2)-1} \cdots y_0$;
$\underline{U}$:= *Karatsuba* ($\underline{a}$, $\underline{c}$, $n/2$);
$\underline{V}$:= *Karatsuba* ($\underline{b}$, $\underline{d}$, $n/2$);
$\underline{W}$:= *Karatsuba* ($\underline{a}+\underline{b}$, $\underline{c}+\underline{d}$, $n/2$);
return $\underline{U}*10^{n} + (\underline{W}-\underline{U}-\underline{V})*10^{n/2} + \underline{V}$;
end if;
end *Karatsuba*;

Performance Analysis

It was observed earlier that one reason for examining algorithmic paradigms was the fact that their running time could often be precisely determined.

This is useful in allowing comparisons between the performances of two algorithms to be made.

For Divide-and-Conquer algorithms the running time is mainly affected by 3 criteria:

1. The **number of sub-instances** (α) into which a problem is split.
2. The **ratio of initial problem size to sub-problem size**. (β)
3. The **number of steps** required to **divide** the initial instance and to **combine** sub-solutions, expressed as a function of the input size, n.

Suppose, P, is a divide-and-conquer algorithm that instantiates α sub-instances, each of size n/β.

Let $T_p(n)$ denote the number of steps taken by P on instances of size n. Then

$$T_P(n_0) = Constant \quad (Recursive-base)$$
$$T_P(n) = \alpha T_P(n/\beta) + \gamma(n)$$

In the case when α and β are both constant (as in all the examples we have given) there is a general method that can be used to solve such *recurrence relations* in order to obtain an asymptotic bound for the running time $T_p(n)$.

In general:

$$T(n) = \alpha T(n/\beta) + O(n^{\gamma})$$

(where γ is constant) has the solution

$$T(n) = \begin{cases} O(n^{\gamma}) & \text{if } \alpha < \beta^{\gamma} \\ O(n^{\gamma}\log_{\beta} n) & \text{if } \alpha = \beta^{\gamma} \\ O(n^{\log_{\beta}\alpha}) & \text{if } \alpha > \beta^{\gamma} \end{cases}$$

Dynamic Programming

This paradigm is most often applied in the construction of algorithms to solve a certain class of

Optimisation Problem

That is: problems which require the *minimisation* or *maximisation* of some measure.

One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the **same computations being performed repeatedly** since *identical* sub-instances may arise.

The idea behind *dynamic programming* is to avoid this pathology by obviating the requirement to calculate the same quantity twice.

The method usually accomplishes this by maintaining a *table of sub-instance results*.

Dynamic Programming is a

Bottom-Up Technique

in which the smallest sub-instances are *explicitly* solved first and the results of these used to construct solutions to progressively larger sub-instances.

In contrast, Divide-and-Conquer is a

Top-Down Technique

which *logically* progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.

We can illustrate these points by considering the problem of calculating the *Binomial Coefficient*, *"n choose k"*, i.e.

$$\left[\begin{matrix} n \\ k \end{matrix}\right] = \frac{n\,!}{k\,!\,(n-k)!} \quad (\,0 \le k \le n\,)$$

It is straightforward to show that

$$\left[\begin{matrix} n \\ k \end{matrix}\right] = \begin{cases} \left[\begin{matrix} n-1 \\ k-1 \end{matrix}\right] + \left[\begin{matrix} n-1 \\ k \end{matrix}\right] & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

Using this relationship, a rather crude Divide-and-Conquer solution to the problem of calculating the Binomial Coefficient *'n choose k'* would be:


```
function bin_coeff (n : integer;
                    k : integer)
                    return integer is
  begin
    if k = 0 or k = n then
      return 1;
    else
      return
      bin_coeff (n-1, k-1) + bin_coeff (n-1, k);
    end if;
  end bin_coeff;
```


By contrast, the Dynamic Programming approach uses the same relationship but constructs a **table of all the (n+1)*(k+1) binomial coefficients** *'i choose j'* for each value of *i* between 0 and *n*, each value of *j* between 0 and *k*.

These are calculated in a particular order:

1. First the table entries corresponding to the coefficients 'i choose 0' and '1 choose 1' are fixed to the value 1.
2. The remaining table entries corresponding to the binomial coefficient 'i choose j' are calculated in *increasing order of the value of i+j*.

It should be noted that since the coefficient '*i choose j*' requires only the values of '*i-1 choose j-1*' and '*i-1 choose j*', computing the table entries in the order of increasing $i+j$ ensures that the table entries needed for ''*i choose j*' have already been calculated, i.e.

$$(i-1)+(j-1) < (i-1)+j < i+j$$

The Dynamic Programming method is given by:


```
function bin_coeff (n : integer;
                    k : integer)
                    return integer is
type table is array (0..n, 0..k) of integer;
bc : table;
i, j, k : integer;
sum : integer;
begin
  for i in 0..n loop
    bc (i, 0) := 1;
  end loop;
  bc (1,1) := 1;
  sum := 3; i := 2; j := 1;
  while sum ≤ n+k loop
    bc (i,j) := bc (i-1, j-1)+bc (i, j-1);
    i := i-1; j := j+1;
    if i < j or j > k then
      sum := sum + 1;
      if sum ≤ n+1 then
        i := sum-1; j := 1;
      else
        i := n; j := sum-n;
      end if;
    end if;
  end loop;
  return bc (n,k);
end bin_coeff;
```


The section of the function consisting of the lines:

```
if i < j or j > k then
  sum := sum + 1;
  if sum ≤ n+1 then
    i := sum-1; j := 1;
  else
    i := n; j := sum-n;
  end if;
end if;
```

is invoked when all the table entries '*i choose j*', for which $i+j$ equals the current value of *sum*, have been found.

The **if** statement increments the value of *sum* and sets up the new values of *i* and *j*.

Now consider the differences between the two methods:

The **Divide-and-Conquer** approach recomputes values, such as $\left[\begin{matrix}2\\1\end{matrix}\right]$, a very large number of times, particularly if n is large and k depends on n, i.e. k is not a constant.

It can be shown that the running time of this method is

$$\Omega\left[\left[\begin{matrix}n\\k\end{matrix}\right]\right]$$

In the worst-case ($k=n/2$) this is, asymptotically, $\Omega(2^n/n)$.

Despite the fact that the algorithm description is quite simple (it is just a direct implementation of the relationship given) it is **completely infeasible** as a practical algorithm.

The **Dynamic Programming** method, since it computes each value $\left[\begin{matrix}i\\j\end{matrix}\right]$ **exactly once** is far more efficient. Its running time is $O(n*k)$, which is $O(n^2)$ in the worst-case, (again $k=n/2$).

It will be noticed that the dynamic programming solution is rather more involved than the recursive Divide-and-Conquer method, nevertheless its running time is practical.

The binomial coefficient example illustrates the key features of dynamic programming algorithms.

1. A **table** of **all** sub-instance results is constructed.
2. The entries corresponding to the **smallest** sub-instances are initiated at the start of the algorithm.
3. The remaining entries are filled in following a precise **order** (that corresponds to **increasing sub-instance size**) using only those entries that have already been computed.
4. Each entry is calculated **exactly once**.
5. The **final** value computed is the solution to the initial problem instance.
6. Implementation is by **iteration** (*never* by recursion, even though the analysis of a problem may naturally suggest a recursive

solution).

Example: Shortest Path

Input: A directed graph, $G(V,E)$, with nodes

$V = \{1, 2, \ldots, n\}$

and edges $E \subseteq V \times V$. Each edge in E has associated with it a non-negative length.

Output: An $n \times n$ matrix, D, in which $D_{i,j}$ contains the *length* of the *shortest path* from node i to node j in G.

Informal Overview of Method

The algorithm, conceptually, constructs a *sequence of matrices*:

$$D_0, D_1, \ldots, D_k, \ldots, D_n$$

For each k (with $1 \le k \le n$), the (i, j) entry of D_k, denoted $D_k(i,j)$, will contain the

Length of the shortest path from node i to node j when only the nodes

$$\{1, 2, 3, \ldots, k\}$$

can be used as intermediate nodes on the path.

Obviously $D_n = D$.

The matrix, D_0, corresponds to the '*smallest sub-instance*'.

D_0 is initiated as:

$$D_0(i,j) = \begin{cases} 0 & \text{if } i=j \\ \infty & \text{if } (i,j) \notin E \\ Length(i,j) & \text{if } (i,j) \in E \end{cases}$$

Now, suppose we have constructed D_k, for some $k < n$.

How do we proceed to build D_{k+1}?

The shortest path from i to j with **only**

$$\{ 1, 2, 3, \ldots, k, k+1 \}$$

available as *internal nodes*

Either: *Does not* contain the node $k+1$.

Or: *Does* contain the node $k+1$.

In the former case:

$$D_{k+1}(i, j) = D_k(i, j)$$

In the latter case:

$$D_{k+1}(i, j) = D_k(i, k+1) + D_k(k+1, j)$$

Therefore $D_{k+1}(i, j)$ is given by

$$\textit{minimum} \begin{cases} D_k(i, j) \\ D_k(i, k+1) + D_k(k+1, j) \end{cases}$$

Although these relationships suggest using a recursive algorithm, as with the previous example, such a realisation would be extremely inefficient.

Instead an *iterative* algorithm is employed.

Only *one* $n \times n$ matrix, D, is needed. This is because after the matrix D_{k+1} has been constructed, the matrix D_k is no longer needed. Therefore D_{k+1} can overwrite D_k.

In the implementation below, L denotes the matrix of *edge lengths* for the set of edges in the graph $G(V, E)$.


```
type matrix is array (1..n, 1..n) of integer;
L : matrix
function shortest_path_length (L : matrix;
                               n : integer)
                               return matrix is
D : matrix; -- Shortest paths matrix
begin
  -- Initial sub-instance
  D (1..n, 1..n) := L (1..n, 1..n);
  for k in 1..n loop
    for i in 1..n loop
      for j in 1..n loop
        if D (i,j) > D (i,k) + D (k,j) then
          D ( i,j ) := D (i,k) + D (k,j);
        end if;
      end loop;
    end loop;
  end loop;
  return D (1..n, 1..n);
end shortest_path_length;
```


This algorithm, discovered by Floyd, clearly runs in time

$$O(n^3)$$

Thus $O(n)$ steps are used to compute each of the n^2 matrix entries.

Greedy Algorithms

This is another approach that is often used to design algorithms for solving

Optimisation Problems

In contrast to dynamic programming, however,

A. Greedy algorithms **do not always** yield a genuinely optimal solution. In such cases the greedy method is frequently the basis of a *heuristic approach*.

B. Even for problems which can be solved exactly by a greedy algorithm, establishing the **correctness** of the method may be a non-trivial process.

In order to give a precise description of the greedy paradigm we must first consider a more detailed definition of the environment in which typical optimisation problems occur.

Thus in an optimisation problem, one will have, in the context of greedy algorithms, the following:

1. A collection (set, list, etc) of **candidates**, e.g. nodes, edges in a graph, etc.
2. A set of candidates which have already been 'used'.
3. A **predicate** (*solution*) to test whether a given set of candidates give a *solution* (not necessarily optimal).
4. A predicate (*feasible*) to test if a set of candidates can be **extended** to a (not necessarily optimal) solution.
5. A **selection function** (*select*) which chooses some candidate which has not yet been used.
6. An **objective function** which assigns a *value* to a solution.

In other words:

An optimisation problem involves finding a subset, *S*, from a collection of candidates, *C*; the subset, *S*, must satisfy some specified criteria, i.e. be a solution and be such that the *objective function* is optimised by *S*.

'Optimised' may mean

Minimised or **Maximised**

depending on the precise problem being solved.

Greedy methods are distinguished by the fact that the **selection function** assigns a *numerical value* to each candidate, *x*, and chooses that candidate for which:

$SELECT(x)$ is **largest**

or

$SELECT(x)$ is **smallest**

All Greedy Algorithms have exactly the same general form. A Greedy Algorithm for a particular problem is specified by describing the predicates '*solution*' and '*feasible*'; and the selection function '*select*'.

Consequently, Greedy Algorithms are often very easy to design for optimisation problems.

The General Form of a Greedy Algorithm is


```
function select (C : candidate_set) return candidate;
function solution (S : candidate_set) return boolean;
function feasible (S : candidate_set) return boolean;
--***************************************************
function greedy (C : candidate_set) return candidate_set is
x : candidate;
S : candidate_set;
begin
  S := ∅;
  while (not solution (S)) and C /= ∅ loop
    x := select ( C );
    C := C − {x};
    if feasible( S ∪ {x} ) then
      S := S ∪ { x };
    end if;
  end loop;
  if solution( S ) then
    return S;
  else
    return ∅;
  end if;
end greedy;
```


As illustrative examples of the greedy paradigm we shall describe algorithms for the following problems:

1. Minimal Spanning Tree.
2. Integer Knapsack.

For the first of these, the algorithm **always** returns an optimal solution.

Minimal Spanning Tree

The inputs for this problem is an (undirected) graph, $G(V, E)$ in which each edge, $e \in E$, has an associated positive edge length, denoted $Length(e)$.

The output is a **spanning tree**, $T(V, F)$ of $G(V,E)$ such that the *total edge length*, i.e. $\sum_{e \in F} Length(e)$ is **minimal** amongst all the possible spanning trees of $G(V,E)$.

Note: An *n*-node **tree**, *T* is a *connected n*-node graph with *exactly n-1 edges*.

$T(V,F)$ is a **spanning tree** of $G(V,E)$ if and only if *T* is a tree and the edges in *F* are a **subset** of the edges in *E*.

In terms of general template given previously:

1. The **candidates** are the **edges of G(V,E)**.
2. A subset of edges, *S*, is a **solution** if the graph $T(V,S)$ is a spanning tree of $G(V,E)$.
3. A subset of edges, *S*, is **feasible** if there is a spanning tree $T(V,H)$ of $G(V,E)$ for which $S \subseteq H$.
4. The **objective function** which is to be **minimised** is the sum of the edge lengths in a **solution**.
5. The **select** function chooses the candidate (i.e. edge) whose **length is smallest** (from the remaining candidates).

The full algorithm, discovered by Kruskal, is:


```
function min_spanning_tree (E : edge_set)
                    return edge_set is
S : edge_set;
e : edge;
begin
  S := ∅;
  while (H (V,S) not a tree)
          and E /= ∅ loop
    e := Shortest edge in E;
    E := E − {e};
    if H (V, S ∪ {e}) is acyclic then
      S := S ∪ {e};
    end if;
   end loop;
   return S;
end min_spanning_tree;
```

Before proving the correctness of this algorithm, we give an example of it running.

The algorithm may be viewed as dividing the set of nodes, V, into n parts or *components*:

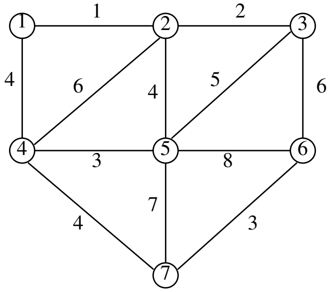
$$\{1\}\,;\{2\}\,;\;\cdots\;;\{n\}$$

An edge is added to the set S if and only if it joins two nodes which belong to *different* components; if an edge is added to S then the two components containing its endpoints are coalesced into a single component.

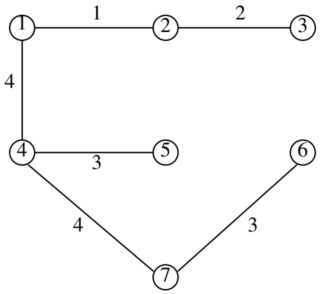
In this way, the algorithm stops when there is just a single component

$$\{\,1,2,\ldots,n\,\}$$

remaining.



Iteration	Edge	Components
0	-	{1}; {2}; {3}; {4}; {5}; {6}; {7}
1	{1,2}	{1,2}; {3}; {4}; {5}; {6}; {7}
2	{2,3}	{1,2,3}; {4}; {5}; {6}; {7}
3	{4,5}	{1,2,3}; {4,5}; {6}; {7}
4	{6,7}	{1,2,3}; {4,5}; {6,7}
5	{1,4}	{1,2,3,4,5}; {6,7}
6	{2,5}	Not included (adds cycle)
7	{4,7}	{1,2,3,4,5,6,7}



Question: How do we know that the resulting set of edges form a

Minimal

Spanning Tree?

In order to prove this we need the following result.

For $G(V,E)$ as before, a subset, F, of the edges E is called *promising* if F is a subset of the edges in a minimal spanning tree of $G(V,E)$.

Lemma: Let $G(V,E)$ be as before and $W \subset V$. Let $F \subseteq E$ be a **promising** set of edges such that no edges in F has *exactly one* enpoint in W.

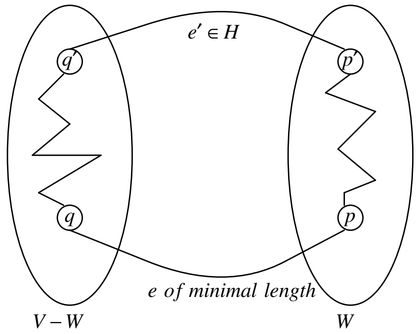
If $\{p,q\} \in E-F$ is a shortest edge having exactly one of p or q in W then: the set of edges $F \cup \{\,\{p,q\}\,\}$ is promising.

Proof: Let $T(V,H)$ be a minimal spanning tree of $G(V,E)$ such that $F \subseteq H$. Note that T exists since F is a promising set of edges.

Consider the edge $e = \{p,q\}$ of the Lemma statement.

If $e \in H$ then the result follows immediately, so suppose that $e \notin H$. Assume that $p \in W$ and $q \notin W$ and consider the graph $T'(V,\,H \cup \{e\})$. Since T is a tree the graph T' (which contains one extra edge) must contain a cycle that includes the (new) edge $\{p,q\}$.

Now since $p \in W$ and $q \notin W$ there must be *some* edge, $e' = \{p',q'\}$ in H which is also part of this cycle and is such that $p' \in W$ and $q' \notin W$.



Now, by the choice of e, we know that

$$Length\,(\,e\,)\;\le\;Length\,(\,e'\,)$$

Removing the edge e' from T' gives a new spanning tree of $G(V,E)$. The cost of this tree is exactly

$$cost\,(\,T\,) - Length\,(e') + Length\,(e)$$

and this is $\le cost\,(T)$.

T is a *minimal* spanning tree so either e and e' have the same length or this case cannot occur. It follows that there is a minimal spanning tree containing $F \cup \{e\}$ and hence this set of edges is promising as claimed. □

Theorem: Kruskal's algorithm always produces a minimal spanning tree.

Proof: We show by induction on $k \geq 0$ - the number of edges in S at each stage - that the set of edges in S is always promising.

Base $(k=0)$: $S=\emptyset$ and obviously the empty set of edges is promising.

Step: ($\leq k-1$ ☞ k): Suppose S contains $k-1$ edges. Let $e=\{p,q\}$ be the next edge that would be added to S. Then:

1. p and q lie in different components.
2. $\{p,q\}$ is a shortest such edge.

Let C be the component in which p lies. By the inductive hypothesis the set S is promising. The Inductive Step now follows by invoking the Lemma, with $W = \mathit{Set\ of\ nodes\ in}\ C$ and $F=S$. □.

Integer Knapsack

In various forms this is a frequently arising optimisation problem.

Input: A set of items

$$U=\{u_1, u_2, \ldots, u_n\}$$

each item having a given *size* $s(u_i) \in \mathbf{N}$ and **value** $v(u_i) \in \mathbf{N}$.
A **capacity** $K \in \mathbf{N}$.

Output: A subset B of U such that $(\sum_{u \in B} s(u) \leq K)$ and $\sum_{u \in B} v(u)$ is maximised.

No fast algorithm *guaranteed* to solve this problem has yet been discovered.

It is considered

extremely improbable

that such an algorithm exists.

Using a greedy approach, however, we can find a solution whose value is at worst 1/2 of the optimal value.

1. The items, U, are the **candidates**.
2. A subset, B, is a **solution** if $\sum_{u \in B} s(u) \leq K$ but $(\sum_{u \in B} s(u)) + s(v) > K$ for all $v \notin B$, i.e. the total size of B fits within the given capacity, but adding any other item will exceed the capacity.
3. The **objective function** which is to be **maximised** is $\sum_{u \in B} v(u)$.

The **selection function** chooses that item, u_i for which

$$\frac{v(u_i)}{s(u_i)}$$

is **maximal**

These yield the following greedy algorithm which *approximately* solves the integer knapsack problem.


```
function knapsack (U : item_set;
                   K : integer )
                   return item_set is
C, S : item_set;
x : item;
begin
  C := U; S := ∅;
  while C /= ∅ loop
    x := Item u in C such that
         v(u)/s(u) is largest;
    C := C − {x};
    if ( Σ_{u ∈ S} s(u) ) + s(x) ≤ K then
      S := S ∪ {x};
    end if;
  end loop;
  return S;
end knapsack;
```


A very simple example shows that the method can fail to deliver an optimal solution. Let

$$U=\{u_1, u_2, u_3, \ldots, u_{12}\}$$

$s(u_1)=101$; $v(u_1)=102$

$s(u_i)=v(u_i)=10 \quad 2 \leq i \leq 12$

$K=110$

Greedy solution: $S=\{u_1\}$; Value is 102.

Optimal solution: $S=U-\{u_1\}$; Value is 110.

Backtracking and Searching

In a number of applications graph structures occur.

The graph may be an *explicit* object in the problem instance as in:

Shortest Path Problem
Minimal Spanning Tree Problem

Graphs, however, may also occur *implicitly* as an abstract mechanism with which to analyse problems and construct algorithms for these.

Among the many areas where such an approach has been used are:

Game Playing Programs
Theorem Proving Systems
Semantic Nets
Hypertext
. . .

Whether a graph is an explicit or implicit structure in describing a problem, it is often the case that *searching* the graph structure may be necessary.

Thus it is required to have methods which

- Can 'mark' nodes in a graph which have already been 'examined'.
- Determine which node should be examined next.
- Ensure that every node in the graph **can** (but not necessarily **will**) be visited.

These requirements must be realised subject to the constraint that the search process respects the structure of the graph. That is to say,

(With the exception of the first node inspected)

Any new node examined must be adjacent to some node that has previously been visited.

So, *search methods* implicitly describe an **ordering** of the nodes in a given graph.

One search method that occurs frequently with implicit graphs is the technique known as

backtracking

Suppose a problem may be expressed in terms of detecting a particular class of subgraph in a graph.

Then the *backtracking* approach to solving such a problem would be:

Scan each node of the graph, *following a specific order*, until

1. A subgraph constituting a solution has been found.

   or

2. It is discovered that the subgraph built so far cannot be extended to be a solution.

If (2) occurs then the search process is '*backed-up*' until a node is reached from which a solution might still be found.

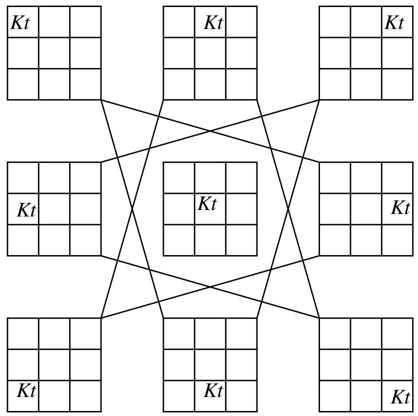
Simple Example

Knight's Tour

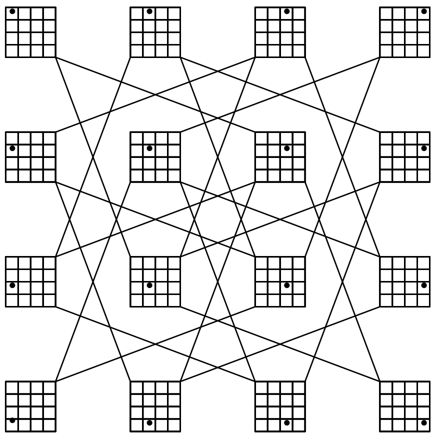
Given a natural number, n, describe how a Knight should be moved on an $n \times n$ chessboard so that it visits every square exactly once and ends on its starting square.

The *implicit graph* in this problem has n^2 nodes corresponding to each position the Knight must occupy. There is an edge between two of these nodes if the corresponding positions are 'one move apart'.

The sub-graph that defines a solution is a cycle which contains each node of the implicit graph.



Knight's Move Graph (n=3)



Knight's Move Graph (n=4)

Of course it is not necessary to construct this graph explicitly, in order to to solve the problem.

The algorithm below, recursively searches the graph, labeling each square (i.e. node) in the order in which it is visited.

In this algorithm:

- *board* is an $n \times n$ representation of the board; initiated to 0.
- (x,y) are the coordinates (row, column) of the current square.
- *move* is the number of squares visited so far.
- *ok* is a Boolean indicating success or failure.


```
type chess_board is array (1..n,1..n) of integer;
procedure knight (board : in out chess_board;
                  x,y,move : in out integer;
                  ok : in out boolean) is
w, z : integer;
begin
  if move = n^2+1 then
    ok := ( (x,y) = (1,1) );
  elsif board (x,y) /= 0 then
    ok := false;
  else
    board (x,y) := move;
    loop
      (w,z) := Next position from (x,y);
      knight (board, w, z, move+1, ok );
      exit when (ok or No moves remain);
    end loop;
    if not ok then
      board ( x,y ) :=0; -- Backtracking
    end if;
  end if;
end knight;
```


Depth-First Search

The Knight's Tour algorithm organises the search of the implicit graph using a **depth-first** approach.

Depth-first search is one method of constructing a **search tree** for *explicit graphs*.

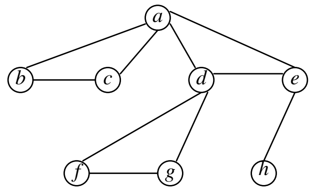
Let $G(V,E)$ be a connected graph. A **search tree** of $G(V,E)$ is a **spanning tree**, $T(V, F)$ of $G(V,E)$ in which the nodes of T are labelled with unique values k $(1 \le k \le |V|)$ which satisfy:

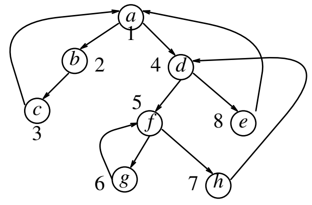
1. A distinguished node called the **root** is labelled 1.
2. If (p,q) is an edge of T then the label assigned to p is less than the label assigned to q.

The labelling of a search tree prescribes the *order* in which the nodes of G are to be scanned.

Given an undirected graph $G(V,E)$, the depth-first search method constructs a search tree using the following recursive algorithm.


```
procedure depth_first_search (G(V,E) : graph;
                              v : node;
                              λ : integer;
                              T : in outsearch_tree) is
begin
  label (v) := λ;
  λ:= λ+1;
  for each w such that {v,w} ∈ E loop
    if label (w) = 0 then
      Add edge {v,w} to T;
      depth_first_search(G (V,E),w,λ,T);
    end if;
  end loop;
end depth_fist_search;
--*******************************
-- Main Program Section
--******************************
begin
  for w ∈ V loop
    label ( w ) :=0;
  end loop;
  λ:= 1;
  depth_first_search ( G (V,E), v, λ, T);
end;
```



If $G(V,E)$ is a *directed graph* then it is possible that not all of the nodes of the graph are reachable from a single root node. To deal with this the algorithm is modified by changing the *'Main Program Section'* to

```
begin
  for each w ∈ V loop
    label ( w ) := 0;
  end loop;
  λ:= 1;
  for each v ∈ V loop
    if label ( v ) = 0 then
      depth_first_search ( G (V,E),v,λ,T );
    end if;
  end loop;
end;
```

The running time of both algorithms, input $G(V,E)$ is $O(|E|)$ since each edge of the graph is examined only once.

It may be noted that the recursive form given, is a rather inefficient method of implementing depth first search for explicit graphs.

Iterative versions exist, e.g. the method of Hopcroft and Tarjan which is based on a 19th century algorithm discovered by Tremaux.

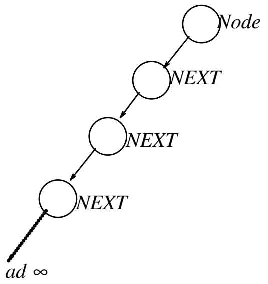
Depth-first search has a large number of applications in solving other graph based problems, for example:

Topological Sorting

Connectivity Testing

Planarity Testing

One disadvantage of depth first search as a mechanism for searching implicit graph structures, is that expanding some paths may result in the search process never terminating because no solution can be reached from these. For example this is a possible difficulty that arises when scanning proof trees in Prolog implementations.



Breadth-first Search is another search method that is less likely to exhibit such behaviour.

```
λ := 1; -- First label
Current_Level := {v}; -- Root node
while Current_Level /= ∅ loop
 for each v ∈ Current_Level loop
  Next_Level := Next_level ∪
         Unmarked neighbours of v;
  if label(v) = 0 then
   label(v) := λ;
   λ := λ + 1;
  end if;
 end loop;
 Current_Level := Next_Level;
 Next_Level := ∅;
end loop;
```

Thus each vertex labelled on the k'th iteration of the outer loop is 'expanded' before any vertex found later.