Algorithm

It is sequence of steps to be followed to solve some particular problem in finite amount of time. (or)

It is a finite sequence of instructions, each of which has a clear meaning & can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite no. of instructions.

Requirements of an Algorithm

1) Input :

An algorithm may take '0' or any no. of inputs

2) Output :

An algorithm may have minimum '1' & max any no. of outputs.

3) Definiteness

Every statement in an algorithm must be clear and unambiguous.

Eg: a:=5 or 10;

This is an ambiguous statement in which variable 'a' has been assign more than one value such statements may be removed.

4) Finiteness

statements in the algorithm must be executed in finite amount of time (once converted to program).

```
i = 5;
while (i>2) do
{
  print i;
  i = i+1;
}
```

Though we have 6 lines of code in algorithm but once executed it enter into infinite loop.

So, the above code is not possessing property of finiteness. so, such statements should be moved.

5) Effectiveness:

Every statement must have some impact on the outcome of the program.

Ex:

Algorithm to perform addition of 2 numbers;

1) Read a; // By removing the statement, o/p is garbage
2) Read b; // By removing the statement, o/p is garbage
3) c = a+b; // By removing the statement, o/p is garbage
4) print c; // By removing the statement, we won't get o/p

All the above statements are having some impact on the outcome of the program.

→ Issues for Algorithm

There are various issues in the study of algorithm.

1) How to devise an algorithm ?

choosing the correct solution or method for solving the problem.

2) How to validate an algorithm?

By testing with all possible inputs.

3) How to analyse an algorithm?

using time complexity, space complexity algorithm can be analyzed.

4) How to debug the algorithm?

Verify the statements line by line to avoid errors.

→ Pseudocode for expressing Algorithms

Algorithm specification

Algorithm can be described in three ways.

1) Natural language like english :

when this way is choosed care should be taken, we should ensure that each & every statement is definite.

2) Graphic representation called flowchart:

This method will work well when the

→ Issues for Algorithm

There are various issues in the study of algorithm.

1) How to devise an algorithm?
choosing the correct solution or method for solving the problem.

2) How to validate an algorithm?
By testing with all possible inputs.

3) How to analyze an algorithm?
using time complexity, space complexity algorithm can be analyzed.

4) How to debug the algorithm?
Verify the statements line by line to avoid errors.

→ Pseudocode for expressing Algorithms

Algorithm specification

Algorithm can be described in three ways.

1) Natural language like english:
when this way is choosed care should be taken, we should ensure that each & every statement is definite.

2) Graphic representation called flowchart:
This method will work well when the algorithm is small & simple.

3) pseudo-code method:
In this method, we should typically describe algorithm as program, which resembles language like pascal & cobol.

→ pseudo-code Conventions

1) Comments begin with // & continue till end of line.
2) Blocks are indicated with matching braces { }
3) An identifier begins with a letter. The data types of variables are not explicitly declared.
4) Compound datatypes can be formed with records.

Ex:
```
Node. Record
{
    data type -1 data -1;
        :
    data type -n data -n;
    node *link;
}
```

Here link is a pointer to the record node. Individual data items of a record can be accessed with → and period.

5) Assignment of values to variables is done using the assignment statement.

<variable> := <expression>;

Ex:
```
a := 10;
b := a;
```

6) there are two Boolean values True, False

→ logical operators AND, OR, NOT

→ Relational operators <, <=, >, >=, =, !=

7) Conditional statements :

i) simple if :

```
if (condition) then
  Begin
    statements;
  end
    (or)
if (condition) then
  {
   statements;
  }
```

ii) else if ;

```
if (condition) then
  begin
    statement1;
  end
 else
   begin
     statement 2;
   end
```

iii) switch ;

```
switch : variable
 begin
    : case variable 1 : ≡
    : case variable 2 : ≡
    : default :      ≡
```


iv) else-if ladder ;

```
if (condition) then
  begin
    statements;
  end
else if (condition) then
  begin
    statements;
  end
else
  begin
    statements;
  end
```

8) Control statements :

i) for :

for variable := value1 to value 2 (inclusion) in step values (default) do

```
  {
   statements;
  }
```

ex: for i:=1 to 5 in step 2 do

```
  {
   print i;
  }
```

- for incrementation use positive step value, & value1 <= value 2.

- for decrementation, step value must be a negative number and value1 >= value 2.

2) while :-

```
while (condition) do
  {
   statements;
  }
```

Ex:
```
i:=0;
while (i<2) do
  {
   print i;
   i:= i+1;
  }
```

3) repeat until:

```
repeat
  {
   statements;
  } until (condition);
```

this loop runs until the condition becomes true

Ex:
```
i:=0;
repeat
  {
   print i;
   i:=i+1;
  } until (i<5);
```

output: 0

8) functions

```
Algorithm | procedure  name of algo (parameters)
begin
  ≡
end;
```

def { calls name of algo (actual parameters)

9) Input and output are done using the "instructions read & write.

Ex: Algorithm for sum of n numbers;

```
algorithm sumn(n)
{ begin
    declare sum, i;
    sum:=0;
    for i:=1 to n in step 1 do
    {
      sum:=sum+i;
    }
    Print sum;
} end;
```

Ex: Algorithm to print series of prime numbers

```
Algorithm series of primes()
{
  declare n,i,j;
  read n;
  for i:=2 to n in step 1 do
  {
    for j:=2 to √i instep 1 do
    {
      if (i%j=0) then
        break;
    }
    if (j>√i) then
      print i;
  }
}
```

→ Asymptotic Notations

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm.

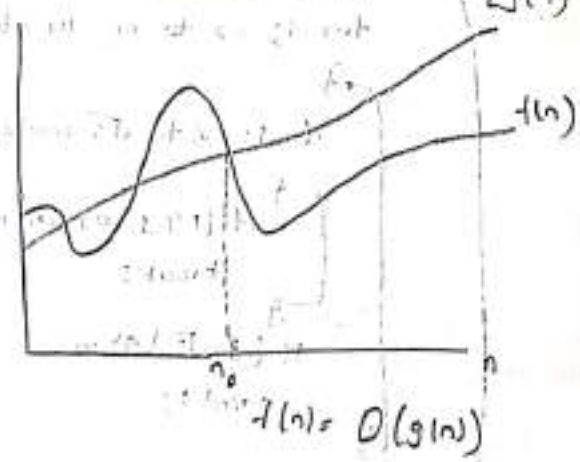
1) Big-OH (O)
2) Big-OMEGA (Ω)
3) Big-THETA (Θ)

Big-OH (O) (UpperBound)

let $f(n), g(n)$ are 2 functions then $f(n)$ can be written as $f(n) = O(g(n))$ if there exists 2 positive constants c, n_0 such that

$$f(n) \leq c * g(n) \quad \forall \quad n \geq n_0$$

$f(n) = O(g(n))$ says that the growth rate of $f(n)$ is less than or equal ($\leq$) that of $g(n)$.



Ex1: let $f(n) = 2n+3$

$g(n) = n$

$c = 4$

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

$2n+3 \leq 4 * n$

$n_0 = 1$	$5 \leq 4$	F
$n_0 = 2$	$7 \leq 8$	T
$n_0 = 3$	$9 \leq 12$	T

$\therefore$ $2n+3 = O(n)$ $\forall n \geq 2$

'O' this represents upper boundary and worst case time complexity.

Ex2: let $f(n) = 2n+2$

$g(n) = n^2$

$c = 4$

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

$2n+2 \leq 4 * n^2$

$n_0 = 1$	$4 \leq 4$	T
$n_0 = 2$	$6 \leq 16$	T

$\therefore$ $2n+2 = O(n^2)$ $\forall n \geq 1$

In Big-oh notation the RHS polynomial must be greater compared to LHS ie

$2n+5 = O(\log n)$

can never be true.

$\log n$ can never become the upper bound of n.

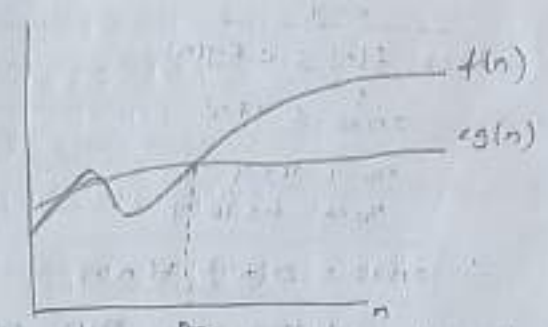
ii) Omega notation (Ω) (lower Bound)

Let $f(n), g(n)$ are 2 functions then $f(n)$ can be written as $f(n) = \Omega(g(n))$ if there exists 2 positive constants c, n_0 such that

$$f(n) \geq c * g(n) \quad \forall \; n \geq n_0$$

Ω represents lower bound or best case time complexity.

$f(n) = \Omega(g(n))$ says that the growth rate of $f(n)$ is greater than or equal to ($\geq$) that of $g(n)$



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

Ex 1: let $f(n) = 3n+3$

$g(n) = n$

$c = 3$

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

$3n+3 \geq 3 * n$

$n_0 = 1$	6	≥ 3
$n_0 = 2$	9	≥ 6

$3n+3 = \Omega(n) \; \forall \; n \geq 1$

c, n_0 should starts with 1

Ex 2: let $f(n) = 3n^2+4$

$g(n) = n$

$c = 10$

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

$3n^2+4 \geq 10 * n$

$n_0 = 1$	$\rightarrow$	≥ 10
$n_0 = 2$	16	≥ 20
$n_0 = 3$	31	≥ 30

$\therefore 3n^2+4 = \Omega(n) \; \forall \; n \geq 3$

In omega notation the RHS polynomial must be lesser compared to LHS i.e

$5+2\log_2 n = \Omega(n)$ can never be true

$2n+5 = \Omega(n^2)$ can never be true

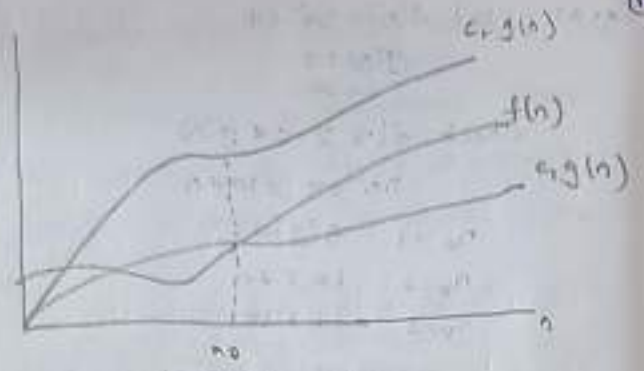
iii) Theta notation (Θ) (same order)

Let $f(n), g(n)$ are 2 functions then $f(n)$ can be written as $R(n) = \Theta(g(n))$ if there exists 3 constants c_1, c_2, n_0 such that

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

$f(n) = \Theta(g(n))$ says that the growth rate of $f(n)$ equals ($=$) growth rate of $g(n)$

[if $f(n) = O(g(n))$ and $T(n) = \Theta(g(n))$]



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

Ex:1 let $f(n) = 3n+2$, $g(n) = n$, $c_1 = 3$, $c_2 = 4$

check $c_1 g(n) \le f(n) \le c_2 g(n)$

$3n \le 3n+2 \le 4n$

$n_0 = 1$ $3 \le 5 \le 4$ F

$n_0 = 2$ $6 \le 8 \le 8$ T

$n_0 = 3$ $9 \le 11 \le 12$ T

$\therefore$ $3n+2 = \Theta(n)$ $\forall\, n \ge 2$

Ex:2 Let $f(n) = n^2+4$, $g(n) = n^2$, $c_1 = 1$, $c_2 = 3$

check $c_1 g(n) \le f(n) \le c_2 g(n)$

$n^2 \le n^2+4 \le 3n^2$

$n_0 = 1$ $1 \le 5 \le 3$ F

$n_0 = 2$ $4 \le 8 \le 12$ T

$n_0 = 3$ $9 \le 13 \le 27$ T

$\therefore$ $n^2+4 = \Theta(n^2)$ $\forall\, n \ge 2$

→ Performance Analysis

The performance of an algorithm can be arrived using 2 measures

1) Time Complexity
2) Space Complexity

1) Time Complexity

The amount of time taken by program (algorithm) to complete the execution.

The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. the time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

The Running time of a program

when solving a problem we are faced with a choice among algorithms. The basis for this can be anyone of the following.

i) we would like an algorithm that is easy to understand code and debug.

ii) we would like an algorithm that makes efficient use of the computer's resources, especially one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

1) The input to the program
2) The quality of code generated by the compiler used to create the object program.
3) The nature and speed of the instructions on the machine used to execute the program.
4) The time complexity of the algorithm underlying the program.

	statement	s/e	frequency	total
1.	Algorithm sum(a,n)	0	-	0
2.	{	0	-	0
3.	s=0.0;	1	1	1
4.	for I=1 to n do	1	n+1	n+1
5.	s=s+a[I];	1	n	n
6.	return s;	1	1	1
7.	}	0	-	0

Total time will be 2n+3

Ex:1 Algorithm factorial()

```
{
    declare n,i,fact; — 2
    read n; — 1
    fact:=1; — 1
    for i:=1 to n ;in step 1 do — n+1
    {
        fact:=fact*i; — n
    }
    print fact; — 1
}
```

2n+7 ≃ O(n)

Time Complexity = O(n)

Ex:2 Algorithm for matrix addition

Algorithm matrixAdd()

```
{
    :
    for i:=1 to m in step 1 do — m+1
    {
        for j:=1 to n in step 1 do — m(n+1)
        {
            c[i,j]:=a[i,j]+b[i,j]; — mn
        }
    }
    :
}
```

2mn+2m+1 ≃ O(

if (n=n) ≃ O(n²)

Ex: 3: Algorithm for matrix multiplication

```
Algorithm matrixMul( )
{
    for i:=1 to n in step 1 do ——— n+1
    {
        for j:=1 to n in step 1 do ——— n(n+1)
        {
            for k:=1 to n in step 1 do ——— n²(n+1)
            {
                c[i,j]= c[i,j]+ a[i,k]*b[k,j]; ——— n³
            }
        }
    }
}
```

$2n^3+2n^2+2n+1 \cong O(n^3)$

The order of time Complexities

$O(1) < O(\log n) < O(n) < O(nt) < O(n^3) \ldots < O(2^n) < O(n!)$

- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(nt)$: Quadratic
- $O(n^3)$: cubic
- $O(2^n)$: Exponential

Space Complexity

The amount of space taken by program (algorithm) to complete the execution.

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space

Is the space needed to store the compiled version of the program instruction.

The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation.
- The target computer.

Data space

It is the space needed to store all constant and variable values. It has 2 compon

- space needed by constants and simple variables in program.
- space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space

It is used to save information needed to resume execution of partially completed functions

The space requirement $S(P)$ of any algorithm P may therefore be written as,

$$S(P) = c + Sp(\text{Instance characteristics})$$

where 'c' is constant

Eg: Algorithm for sum of n natural numbers

```
Algorithm SumNatural( )
{
    declare n,i,sum; — 3
    read(n); — 0
    sum:=0; — 0
    for i:=1 to n in step 1 do — n
    {
        sum:=sum+i; — 0
    }
    print sum; — 0
}
```

3 Constant

∴ space complexity = O(1)

Eg: Algorithm Sumof Array( )

```
{
    declare i, n, sum; — 3
    read(n);
    declare a[n]; — n
    read(array); — 0
    sum:=0; — 0
    for i:=1 to n in step 1 do — n
    {
        sum:=sum+a[i]; — 0
    }
    print(sum); — 0
}
```

n+3 ≥ n

The order of space complexities

$1 < \log n < n < n \log n < n^2 < n^3 \ldots < 2^n < 3^n < \ldots < n^n$

Eg 3:

```
Algorithm sum(a,n)
{
    s=0.0;
    for I=1 to n do
        s=s+a[I];
    return s;
}
```

— the problem instances for this algorithm are characterized by n, the number of elements to be summed. The space needed by 'n' is one word, since it is of type integer.

- the space needed by 'a' as the space needed by variables of type array of floating point numbers.
- This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed
- So, we obtain Ssum(n) = (n+3) [n for a[], one each for n, 2 args]

Complexity of Algorithms (M)

M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of size 'n' of input data.

The storage space required by an algorithm is simply a multiple of data size 'n', Complexity shall refer to running time of algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data the complexity function f(n) for certain cases are:

1) Best case: The min possible value of f(n) is called the best case.
2) Average case: The expected value of f(n).
3) Worst case: The maximum value of f(n) for any key possible input.

little-o notation

A theoretical measure of the execution of an algorithm, usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation $f(n) = o(g(n))$ means $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity. The notation is read: "f of n is little oh of g of n".

formal definition

$f(n) = o(g(n))$ means for all $c > 0$ there exists some $k > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge k$. The value of k must not depend on n, but may depend on c.

- let f(n), g(n) are 2 non-negative functions if $\lim [f(n)/g(n)] = 0$ then we say that $f(n) = o(g(n))$

Ex: $f(n) = 2n+3$, $g(n) = n^2$

$$\lim f(n)/g(n) = 0$$

$$\lim (2n+3)/n^2$$

$$= \lim n(2+(3/n))/n^2$$

$$= \lim (2+(3/n))/n$$

$$= 2/$$

$$= 0$$

$$\therefore f(n) = o(n^2)$$

→ Divide and Conquer

→ General Method

It is a design strategy which is well known to breaking down efficiency barriers. when the method applies, it often leads to a large improvement in time complexity.

For ex from $O(n^2)$ to $O(n \log n)$ to sort the elements.

Divide and Conquer strategy is as follows: divide the problem instance into two or more smaller instances of same problem, solve the smaller instances recursively and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. when dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

It consist of two parts:

Divide: Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer: The solution to the original problem is then formed from the solutions to the sub problems.

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. It is very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{
  if SMALL(P) then return S(P);
  else
  {
    divide P into smaller instances P1, P2 ... PK, k≥1;
    apply DANDC to each of these sub problems;
    return (COMBINE (DANDC(P1), DANDC(P2), ...
                     DANDC(PK));
  }
}
```

SMALL(P) is a Boolean valued function which determines whether the input size is small

enough so that the answer can be computed without splitting. If this is so function 's' is invoked otherwise the problem 'p' into smaller sub problems. These sub problems $P_1, P_2, \ldots, P_k$ are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the Computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2T(n/2) + f(n) & \text{otherwise} \end{cases}$$

where, $T(n)$ is the time for DANDC on 'n' inputs

$g(n)$ is the time to complete the answers directly for small inputs and

$f(n)$ is the time for Divide and Combine.

The complexity of many divide and conquer algorithms is given by recurrences of the form.

$$T(n) = \begin{cases} T(1) & n=1 \\ aT(n/b) + f(n) & n>1 \end{cases}$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b ($i.e.\ n = b^k$)

One of the methods for solving any such recurrence relation is called substitution method this method repeatedly makes substitution for each occurrence of the function T in the

right-hand side until all such occurrences disappear.

Ex: $a=2$, $b=2$, let $T(1)=2$ and $f(n)=n$, we have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\left[2T(n/4) + n/2\right] + n \\ &= 4T(n/4) + 2n \\ &= 4\left[2T(n/8) + n/4\right] + 2n \\ &= 8T(n/8) + 3n \end{aligned}$$

In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$.

→ Applications

→ Binary search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$ when we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that $a[j] = x$ (successful search). If 'x' is not in the list then 'j' is to set to zero (unsuccessful search).

In Binary Search we jump into the middle of the file, where we find key a[mid], and Compare 'x' with a[mid]. If $x = a[mid]$ then the desired record has been found.

If $x < a[mid]$ then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if $a[mid] > x$, then further search is only necessary in that part of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every unsuccessful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparision between 'x' and a[mid] and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

low and high are Integer variables such that each time through the loop either 'x' is found or low is increased by at least one or high is decreased by at least one. Thus we have two sequences of Integers approaching each other & eventually low will become greater than high causing termination in a finite number of steps if 'x' is not present.

Ex: Recursive binary search

```
Algorithm BinSrch(a,i,l,x)
// Given an array a[i:l] of elements in nondecreasing
//order, 1≤i≤l, determine whether x is present, &
// if so, return j such that x=a[j]; else return 0
{
  if (l=i) then // if Small(P)
  {
    if (x=a[i]) then return i;
    else return 0;
  }
  else
  { // Reduce P into a smaller subproblem.
    mid := [(i+l)/2];
    if (x=a[mid]) then return mid;
    else if (x<a[mid]) then
      return BinSrch(a,i,mid-1,x);
    else return BinSrch(a,mid+1,l,x);
  }
}
```

Iterative Binary Search

```
Algorithm BinSearch(a,n,x)
// Given an array a[1:n] of elements in nondecreasing
// order, n≥0, determine whether x is present, &
// if so, return j such that x = a[j]; else return 0
{
    low := 1; high := n;
    while (low ≤ high) do
    {
        mid := ⌊(low+high)/2⌋;
        if (x < a[mid]) then high := mid-1;
        else if (x > a[mid]) then low := mid+1;
             else return mid;
    }
    return 0;
}
```

Ex:

Let us illustrate Binary Search on the following 9 elements

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101

The no of comparisions required for searching different elements is as follows

1. searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

No of Comparisions = 4.

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

No of Comparisions = 3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	Not found

No of Comparisions = 4

Continuing in this manner the no of element Comparisions needed to find each of 9 elements is

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101
Comparision	3	2	3	4	1	3	2	3	4

No. of element requires more than 4 comparisons to be found summing the comparisons needed to find all 9 items & dividing by 9, yielding 25/9 or appr. 2.77 comparisons per successful search on the average.

There are 10 possible ways that an un-successful search may terminate depending upon the value of x.

If $x < a[1]$, $a[1] < x < a[2]$, $a[2] < x < a[3]$, $a[5] < x < a[6]$, $a[6] < x < a[7]$ or $a[7] < x < a[8]$ the algorithm requires 3 element comparisions to determine that 'x' is not present. for all of the remaining possibilities BINSECH requires 4 elements comparisons. Thus the avg number of element comparisons for an unsuccessful search is:

$$(3+3+3+4+4+3+3+3+4+4)/10 = 34/10 = 3.4$$

The time Complexity for a successful search is $O(\log n)$ & for an unsuccessful search is $\Theta(\log n)$.

Successful Searches			Un-successful Searches
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
Best	avg	worst	best, avg & worst

Analysis for worst case

Let $T(n)$ be the time Complexity of Binary Search

The algorithm sets mid to $\lceil (n+1)/2 \rceil$

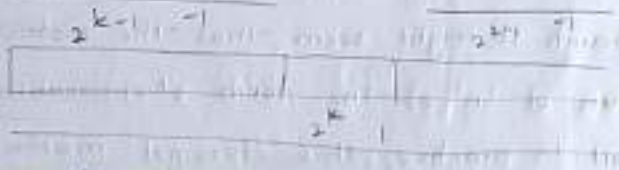
$\therefore$ $T(0) = 0$

$T(n) = 1$ if $x = a[mid]$

$= 1 + T(\lceil (n+1)/2 \rceil - 1)$ if $x < a[mid]$

$= 1 + T(n - \lceil (n+1)/2 \rceil)$ if $x > a[mid]$

Let us restrict 'n' to values of the form $n = 2^k - 1$, where 'k' is a non-negative integer. the array always breaks symmetrically into two equal pieces plus middle element



Algebraically this is

$$\lceil n+1 \rceil \cdot \lceil 2^k - 1 + 1 \rceil = 2^{k-1} \text{ for } k > 1$$

$$\lceil 2 \sqrt{\ } \rceil \lceil 2 \ 1 \rceil$$

Giving

$T(0) = 0$

$T(2^k - 1) = 1$ if $x = a[mid]$

$= 1 + T(2^{k-1} - 1)$ if $x < a[mid]$

$= 1 + T(2^{k-1} - 1)$ if $x > a[mid]$

in the worst case the test x < a[mid] always fails, so

$w(0) = 0$

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

this is now solved by repeated substitution:

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

$= 1 + [1 + w(2^{k-2} - 1)]$

$= 1 + [1 + [1 + w(2^{k-3} - 1)]]$

$= \ldots$

$= i + w(2^{k-i} - 1)$

for $i \le k$, letting $i = k$ gives $w(2^k - 1) = k + w(0) = k$

But as $2^k - 1 = n$, so $k = \log_2(n+1)$, so

$w(n) = \log_2(n+1) = O(\log n)$

for $n = 2^k - 1$, concludes this analysis of binary search

Although it might seem that the restriction of values of 'n' of the form $2^k - 1$ weakens the result. In practice, this does not matter very much, $w(n)$ is a monotonic increasing function of 'n', & hence the formula given is a good approximation even when 'n' is not of the form $2^k - 1$.

→ Merge sort

It is a classic example of divide & conquer. To sort an array, recursively, sort its left and right halves separately & then merge them. The time complexity of merge sort in the best case, worst case and average case is $O(n \log n)$ and the no of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge 2 adjacent sorted arrays together in place.

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

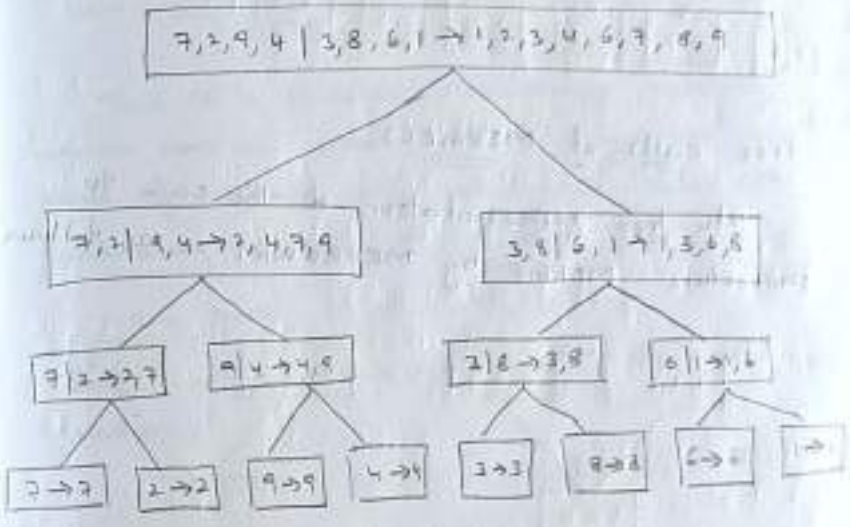
Algorithm

```
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
    if (low < high)
    {
        mid := ⌊(low + high)/2⌋;   // finds where to split set
        MERGESORT (low, mid);      // sort one subset
        MERGESORT (mid+1, high);   // sort the other subset
        MERGE (low, mid, high);    // combine the results
    }
}
```

```
Algorithm MERGE (low, mid, high)
// a[low:high] is a global array containing two
                                    sorted subsets
// in a(low:mid) and    in a(mid+1:high)
// the objective is to merge those sorted sets
                                    into single sorted
// set residing in a (low:high). An auxiliary
                                    array B is used.
{
    h:=low; i:=low; j:=mid+1;
    while((h≤mid) and (j≤high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i]:=a[h]; h:=h+1;
        }
        else
        {
            b[i]:=a[j]; j:=j+1;
        }
        i:=i+1;
    }
    if (h>mid) then
        for k:=j to high do
        {
            b[i]:=a[k]; i:=i+1;
        }
    else
        for k:=h to mid do
        {
            b[i]:=a[k]; i:=i+1;
        }
    for k:=low to high do
        a[k]:=b[k];
}
```

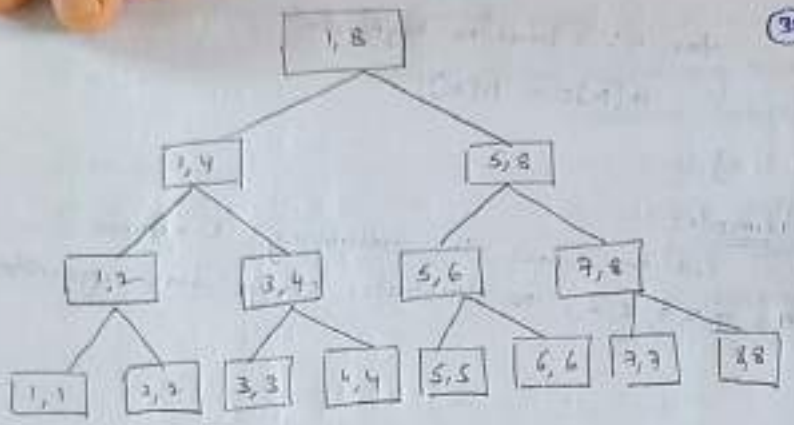

Example:

Let us select the following 8 entries 7,2,9,4,3,8,6,1 to illustrate merge sort algorithms.



Tree calls of MERGESORT (1,8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree calls of MERGE()

The tree representation of the calls to Procedure MERGE by MERGESORT is as follows:

Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

for n=1, the time to merge sort is constant, which we will be denote by 1, otherwise, the time to merge sort 'n' numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says this exactly:

$$T(1) = 1$$
$$T(n) = 2T(n/2) + n$$

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right-hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this main equation

$$2T(n/2) = 2(2(T(n/4)) + n/2)$$
$$= 4T(n/4) + 4$$

We have,

$$T(n/2) = 2T(n/4) + n$$
$$T(n) = 4T(n/4) + 2n$$

Again, by substituting n/4 into main eqn, we get that

$$4T(n/4) = 4(2T(n/8)) + n/4$$
$$= 8T(n/8) + n$$

So we have,

$$T(n/4) = 2T(n/8) + n$$

$$T(n) = 8T(n/8) + 3n$$

Continuing in this manner, we obtain:

$$T(n) = 2^k T(n/2^k) + k \cdot n$$

As $n = 2^k$, $k = \log_2 n$. sub. this in above eqn.

$$T(n) = 2^{\log_2 n} T\left(\frac{2^k}{2^k}\right) + \log_2 n \cdot n$$

$$= nT(1) + n\log n$$

$$= n\log n + n$$

Representing this in O notation:

$$\boxed{T(n) = O(n \log n)}$$

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is $O(n \log n)$, it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. The Best and worst case time complexity of Merge sort is $O(n \log n)$.

→ Quick sort

The main reason for the slowness of algorithms is that all comparisons & exchanges between keys in a sequence $w_1, w_2, \ldots, w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

The quick sort algorithm partitions the original array by rearranging it into 2 groups. The first group contains those elements less than some arbitary chosen value taken from set, Second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the pivot element. once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. when all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of 2 pointers 'i' & 'j' which are moved toward each other in following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot

- Repeatedly decrease the pointer 'j' until a[j] <= pivot.

- If j > i, interchange a[j] with a[i]

- Repeat the steps 1, 2, 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found & Place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' & 'high'.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

- Here we choose the first element as 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], ... x[high].

- It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 and high.

```
- Algorithm QuickSort(p,q)
// sorts the elements a[p], ..., a[q] which reside in global
// array a[1:n] into ascending order; a[n+1] is considered to
// be defined & must be ≥ all the elements in a[1:n]
{
    if (p < q) then // If there are more than one element
    {
        // divide p into two subproblems.
        j := partition (a, p, q+1);
            // j is the position of the partitioning element
        // solve the subproblems
        QuickSort (p, j-1);
        QuickSort (j+1, q);
        // There is no need for combining solutions
    }
}
```

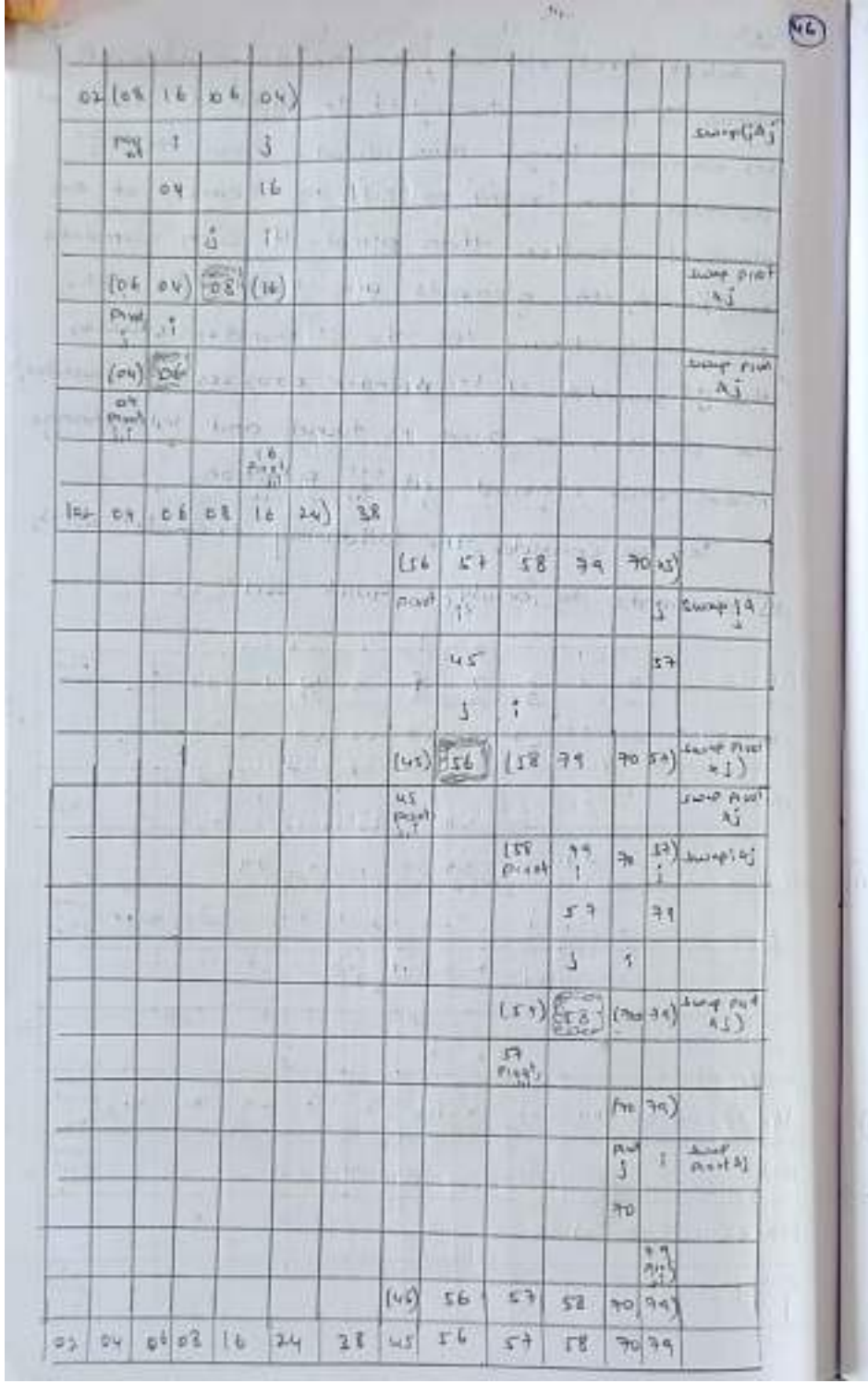
```
Algorithm partition (a,m,p)
// within a[m], a[m+1] ... a[p-1] the elements are
// rearranged in such a manner that if
                    initially t = a[m],
// then after completion a[q] = t for some q
                                between m
// and p-1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p, q is returned, set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i+1;
        until (a[i] ≥ v);
        repeat
            j := j-1;
        until (a[j] ≤ v);
        if (i < j) then Interchange (a,i,j);
    } until (i ≥ j);
    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange (a,i,j)
// Exchange a[i] with a[j]
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```


Ex:

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	2	3	4	5	6	7	8	9	10	11	12	13	Exchange with
38	08	16	06	79	57	24	56	02	58	04	70	45	
Pivot				i						j			Exchange i & j
				04						79			
					i			j					Exchange i & j
					02			57					
						j	i						
38	08	16	06	04	02	38	56	57	58	79	70	45	Exchange Pivot with j
Pivot					j								Exchange Pivot with a
02	08	16	06	04	24								
Pivot, j													Exchange Pivot & j



Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take $T(0) = T(1) = 1$, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (the pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n-i-1) + c_n \quad (1)$$

where, $i = |S_1|$ is the no of elements in S_1.

Worst Case Analysis

The pivot is the smallest element, all the time. Then $i = 0$ and if we ignore $T(0) = 1$, which is insignificant, the recurrence is:

$$T(n) = T(n-1) + c_n \qquad n > 1 \quad (2)$$

Using equation - (1) repeatedly, thus

$$T(n-1) = T(n-2) + c(n-1)$$

$$T(n-2) = T(n-3) + c(n-2)$$

$$\vdots$$

$$T(2) = T(1) + c(2)$$

Adding up all these equations yields.

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

(3)

$$\boxed{T = O(n^2)}$$

Best and Average case Analysis

The no of comparisons for first call on Partition: Assume left_to_right moves over k smaller element and thus k comparisons. so when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

$T(n)$ = Comparisons for first call on quicksort.

$$+ \left\{ \sum_{1 <= n_{left}, n_{right} <= n} \left[T(n_{left}) + T(n_{right}) \right] \right\} n + (n+1) + 2\left[T(0) + T(1) + T(2) + \cdots + T(n-1) \right] / n$$

$$nT(n) = n(n+1) + 2\left[T(0) + T(1) + T(2) + \cdots + T(n-2) + T(n-1) \right]$$

$$(n-1)T(n-1) = (n-1)n + 2\left[T(0) + T(1) + T(2) + \cdots + T(n-2) \right]$$

Substracting both sides:

$$nT(n) - (n-1)T(n-1) = \left[n(n+1) - (n-1)n \right] + 2T(n-1) = 2n + 2T(n-1)$$

$$nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)$$

$$T(n) = 2 + (n+1)T(n-1)/n$$

the recurrence relation obtained is:

$$T(n)/(n+1) = 2/(n+1) + T(n-1)/n.$$

using the method of substitution:

$$T(n)/(n+1) = 2/(n+1) + T(n-1)/n$$

$$T(n-1)/n = 2/n + T(n-2)/(n-1)$$

$$T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)$$

$$T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)$$

$$\vdots$$

$$T(3)/4 = 2/4 + T(2)/3$$

$$T(2)/3 = 2/3 + T(1)/2 \quad T(1)/2 = 2/2 + T(0)$$

Adding both sides:

$$T(n)/(n+1) + \left[T(n-1)/n + T(n-2)/(n-1) + \cdots + T(2)/3 + T(1)/2 \right]$$

$$= \left[T(n-1)/n + T(n-2)/(n-1) + \cdots + T(2)/3 + T(1)/2 \right] + T(0) + \left[2/(n+1) + 2/n + 2/(n-1) + \cdots + 2/4 + 2/3 \right]$$

cancelling the common terms:

$$T(n)/(n+1) = 2\left[1/2 + 1/3 + 1/4 + \cdots + 1/n + 1/(n+1) \right]$$

$$T(n) = (n+1)2 \left[\sum_{2 \le k \le n+1} 1/k \right]$$

$$= 2(n+1)[\ -\ ]$$

$$= 2(n+1)\left[\log(n+1) - \log 2 \right]$$

$$= 2n \log(n+1) + \log(n+1) - 2n \log 2 - \log 2$$

$$\boxed{T(n) = O(n \log n)}$$

→ Strassen's Matrix Multiplication:

The matrix multiplication algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two n×n matrices A and B, yielding result matrix 'c' as follows:

```
for i:=1 to n do
    for j:=1 to n do
        c[i,j]:=0;
        for k:=1 to n do
            c[i,j]:=c[i,j]+a[i,k]*b[k,j];
```

The algorithm requires n^3 scalar multiplications (i.e multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. for example let us consider three multiplication like this:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Then C_{ij} can be found by the usual matrix multiplication algorithm,

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

this leads to a divide-and-conquer algorithm, which performs n×n matrix multiplication by partitioning the matrices into quarters and performing eight $(n/2)\times(n/2)$ matrix multiplication & 4 $(n/2)\times(n/2)$ matrix additions.

$$T(1) = 1$$
$$T(n) = 8T(n/2)$$

which leads to $T(n) = O(n^3)$, where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring 7 $(n/2)\times(n/2)$ matrix multiplications & 8 $(n/2)\times(n/2)$ matrix additions & subtractions:

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22})B_{11}$$
$$R = A_{11}(B_{12} - B_{22})$$
$$S = A_{22}(B_{21} - B_{11})$$
$$T = (A_{11} + A_{12})B_{22}$$
$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$C_{11} = P + S - T + V$$

$c_{12} = R + T$

$c_{21} = Q + S$

$c_{22} = P + R - Q + U$

This method is used recursively to perform the 7 $(n/2) \times (n/2)$ matrix multiplications, then the recurrence equation for the no. of scalar multiplicatio Performed is:

$$T(1) = 1$$
$$T(n) = 7\,T(n/2)$$

Solving this for case of $n = 2^k$ is easy:

$$T(2^k) = 7\,T(2^{k-1})$$
$$= 7^2\,T(2^{k-2})$$
$$= \cdots$$
$$= 7^i\,T(2^{k-i})$$

Put $i = k$

$$= 7^k\,T(1)$$
$$= 7^k$$

That is, $T(n) = 7^{\log_2 n}$

$$= n^{\log_2 7}$$
$$= O(n^{\log_2 7}) = O(n^{2.81})$$

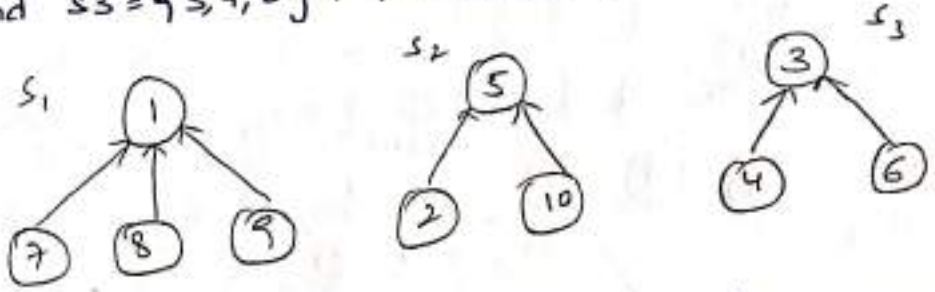
So, Concluding that strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

UNIT - II

Sets and Disjoint set union

Disjoint set union:

Considering a set $S = \{1, 2, 3, \ldots 10\}$ (when n=10), then elements can be partitioned into three disjoint sets $S1 = \{1, 7, 8, 9\}$, $S2 = \{2, 5, 10\}$ and $S3 = \{3, 4, 6\}$. Possible tree representations are:



In this representation each set is represented as a tree. Nodes are linked from the child to parent rather than usual method of linking from parent to child.

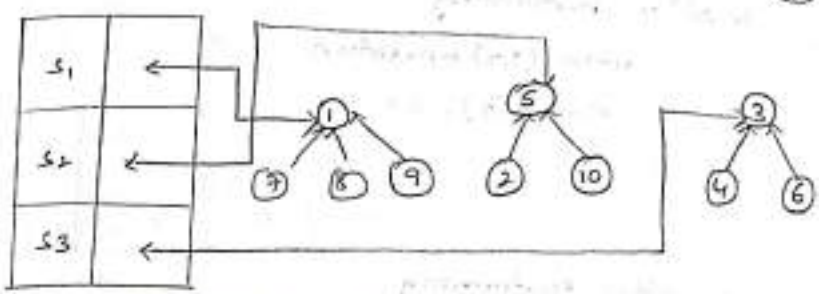
The operations we wish to perform on these sets are:

1) Disjoint set union:

If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = all the elements x such that x is in S_i or S_j. Thus $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$

2) Find (i):

Given the element i, find the set containing i. Thus, 4 is in set S_3, 9 is in S_1.



The transition to

If we determine that element i is in a tree with root i, and j has a pointer to entry k of

→ simple Algorithm for Union:

```
Algorithm Union(i,j)
{
  // replace the disjoint sets with roots i & j,
                i not equal to j by their
                union integer i,j;
  P[j]: = i;
}
```

Ex: Implement following sequence of operations union (1,8), union (2,5) union (1,2)

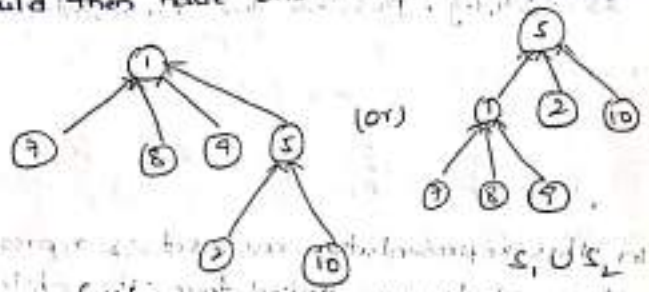
Solution

Initially parent array contains zeros.

0	0	0	0	0	0	
1	2	3	4	5	6	

→ Union and Find operations

Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2. Since we have linked the nodes from children to parent, we simply make one of the trees a subtree of the other. $S_1 \cup S_2$ Could then have one of the representations.



To obtain the union of 2 sets, all that has to be done is to set the parent field of one of the roots to the other root; this can be accomplished easily if with each set name, we keep a pointer to the root of the tree representing that set. Each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree & use the pointer to the set name.

In presenting the union & find algorithms, we ignore the set names & identify sets just by the roots of the trees representing them.

1. After performing union(1,3) operation

parent[3]:=1

0	0	0	0	0	0
1	2	3	4	5	6

2. After performing union(3,5) operation

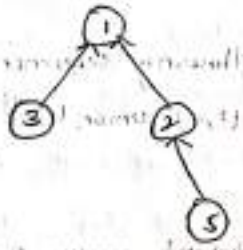
parent[5]:=2

0	0	1	0	2	0
1	2	3	4	5	6

1. After performing union(1,2) operation

parent[2]:=1

0	1	1	0	2	0
1	2	3	4	5	6



Process the following sequence of union operations union(1,2) union(2,3) union(n-1,n)

Degenerate Tree:

The time taken for n-1 union is O(n).

Find(i) operation:

determines the root of the tree containing element i. simple Algorithm for 'Find':

```
Algorithm Find(i)
{
    j:=i;
    while (p[j]>0) do
    j:=p[j]; return j;
}
```

Find operation:

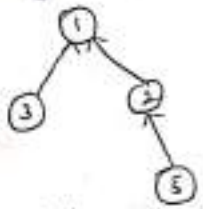
Find(i) implies that it finds the root node of ith node, in other words it returns the name of the set i.

Ex: consider the union(1,3)

Find(1) = 0

Find(3) = 1, since its Parents is 1. (i.e root is 1)

Ex: Considering



Array Representation

P[i]	0	1	1	2
i	1	2	3	5

find(5) = 1

find(2) = 1

find(3) = 1

The root node represents all the nodes in the tree. Time Complexity of 'n' find operations is $O(n^2)$.

To improve the performance of union & find algorithms by avoiding the creation of degenerate tree. To accomplish this, we use weighting rule for Union(i,j).

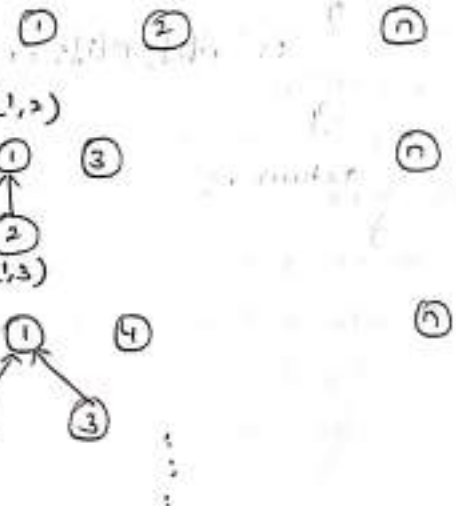
Weighting Rule for Union(i,j)

Algorithm WeightedUnion(i,j)

// union sets with roots i & j, i ≠ j, using the
// weighting rule, p[i] = -count[i] and p[j] = -count[j]

```
{
    temp := p[i] + p[j];
    if (p[i] > p[j]) then
    {  // i has fewer nodes.
        p[i] := j; p[j] := temp;
    }
    else
    {  // j has fewer or equal nodes.
        p[j] := i; p[i] := temp;
    }
}
```

Tree obtained with weighted Initially



Union (i,n)

→ Find algorithm with collapsing rule

```
Algorithm collapsingFind(i).
// find the root of the tree containing
                    element i. use the
// collapsing rule to collapse all nodes
                    from i to the root.
{
    r := i;
    while (p[r] > 0) do r := p[r]; // find
                        the root.

    while (i ≠ r) do // collapse nodes from
                        i to root r.
    {
        s := p[i]; p[i] := r; i := s;
    }
    return r;
}
```


→ Back Tracking

→ General Method

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple $(x_1, \ldots, x_n)$ where each $x_i \in S_i$, S_i being a finite set.

The solution is based on finding one or more vectors that maximize, minimize or satisfy a criterion function $P(x_1, \ldots, x_n)$. form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into 2 categories; explicit, implicit constraints.

Definition 1: Explicit constraints are rules that restrict each x_i to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

Definition 2: Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the

criterion function. Thus, implicit constraints describe the way in which the x_i's must relate to each other.

- for 8-queens problem:

Explicit constraints using 8-tuple formation, for this problem are $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$

The implicit constraints for this problem are that no two queens can be the same (i.e all queens must be on different columns) & no two queens can be on the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

It is the procedure whereby, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent & proceed with the search on the next child.

A Backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way

we implement 'backtracking algorithm. we say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

state space is set of paths from root node to other nodes. state space tree is the tree organizing of the solution space. the state space trees are called state trees. this terminology follows from the observation that the tree organizations are independent of the problem instance being solved. for some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

Terminology:

Problem state is each node in the dfs tree.

Solution states are the problem states 's' for which the path from the root node to 's' defines a tuple in the solution space.

Answer states are those solution states for which the path from root node to s defines a tupple that is a member of the set of solutions.

Live node is a node that has been generated but whose children have not yet been generated.

E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Dead node is a generated node that is not be expanded or explored any further. All children of a dead node have already been expanded.

Branch and Bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Depth first node generation with bounding functions is called backtracking. state generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

→ Applications

→ N-Queens Problem:

Let us consider, N=8. Then 8-queens problem is to place 8 queens on an 8x8 chessboard so that no two "attack", that is, no two of them are on the same row, column or diagonal.

All solutions to the 8-queens problem can be represented as 8-tuples $(x_1, \ldots, x_8)$, where x_i is the column of the i^{th} row where i^{th} queen is placed.

The explicit constraints using this formulations are $S_i = \{1,2,3,4,5,6,7,8\}$, $1 \le i \le 8$. Therefore the solution space contains of 8^8 8-tuples.

The implicit constraints for this problem are that no two x_i's can be the same (i.e, all queens must be on diff columns) & no 2 queens can be on the same diagonal.

This realization reduces the size of the solution space from 8^8 tuples to 8! tuples.

The promising function must check whether two queens are in the same

Column or diagonal:

Suppose 2 queens are placed at positions (i,j) and (k,l). Then:

- Column conflicts: Two queens conflict if their j values are identical.
- Diag 45 conflict: 2 queens i & j are on the same 45° diagonal if:

$$i-j = k-l.$$

This implies, $j-l = i-k$

- Diag 135 conflict:

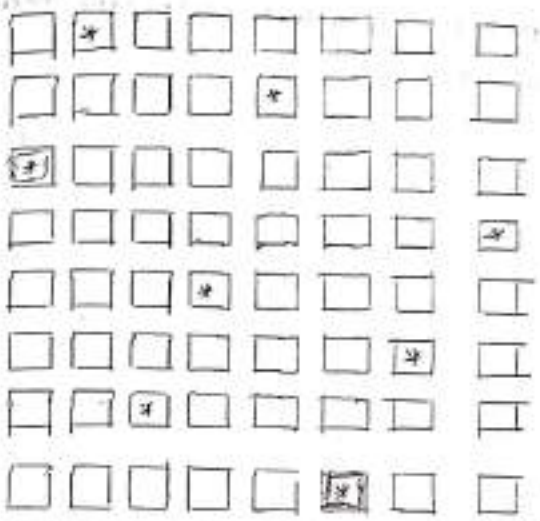
$$i+j = k+l$$

This implies, $j-l = k-i$

Therefore, 2 queens be on the same diagonal if and only if: $|j-l| = |i-k|$

where, j be the column of object in row i for the i^{th} queen & l be the column of object in row k for the k^{th} queen.

To check the diagonal clashes, let us take the following tile configuration;



In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

Let us consider for the 3rd row & 8th row case whether the queens on are conflicting or not. In this case $(i,j) = (3,1)$ and $(k,l) = (8,6)$.

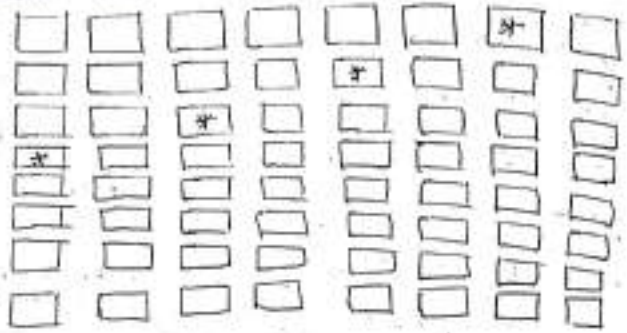
Therefore: $|j-l| = |i-k| \Rightarrow |1-6| = |3-8|$

$\Rightarrow 5 = 5$

In the above example we have, $|j-l| = |i-k|$, so the 2 queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.



Step 1:

Add to the sequence the next number in the sequence 1, 2, ---, 8 not yet used.

Step 2:

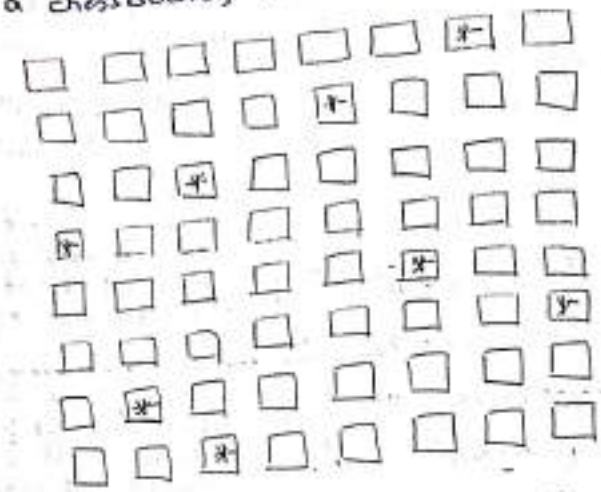
If this new sequence is feasible & has length 8 then STOP with a solution. If the new sequence is feasible and has length less than 8, repeat step 1.

Step 3:

If the sequence is not feasible, then backtrack through the sequence until we find the most recent place at which we can exchange a value. Go back to step 2.

1	2	3	4	5	6	7	8	Remarks
7	5	3	1					
7	5	3	1*	2*				|j-1| = |1-2| = 1 |i-k| = |4-5| = 2
7	5	3	1	4				
7*	5	3	1	4	2*			|j-1| = |9-2| = 5 |i-k| = |1-6| = 5
7	5	3*	1	4	6*			|j-1| = |3-6| = 3 |i-k| = |3-6| = 3
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		|j-1| = |4-1| = 3 |i-k| = |5-2| = 3
7	5	3	1	4*	8	6*		|j-1| = |4-6| = 2 |i-k| = |5-4| = 2
7	5	3	1	4	8			Back track
7	5	3	1	4				Back track
7	5	3	1	6				
7*	5	3	1	6	2*			|j-1| = |1-2| = 1 |i-k| = |7-6| = 1
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	|j-1| = |3-8| = 5 |i-k| = |3-8| = 5
7	5	3	1	6	4	2		Back track
7	5	3	1	6	4			Back track
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	SOLUTION
		*	Indicates	conflicting	queens.			

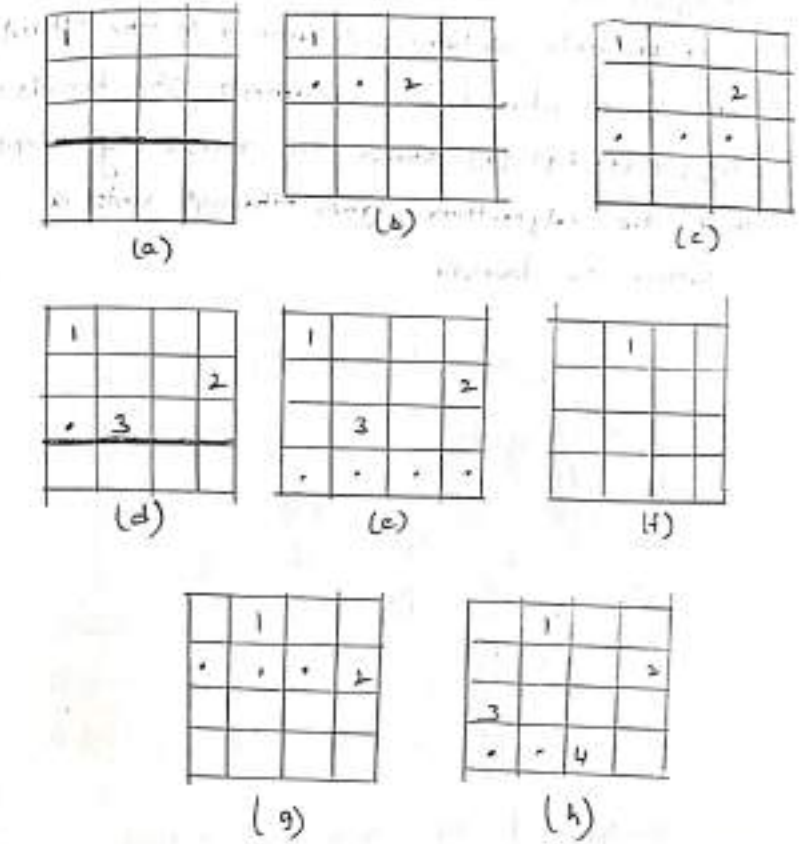
on a chessboard, the solution will look like:



4- Queens problem:

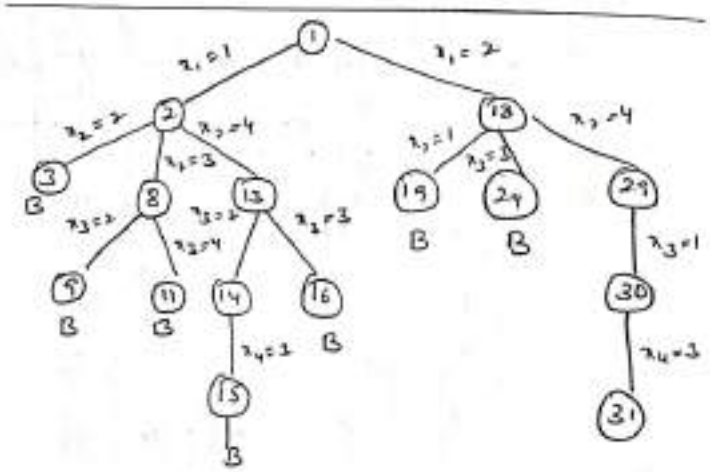
Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure 1.5 is generated & the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated & immediately killed. The next node generated is node 8 & the path becomes (1,3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that

cannot lead to an answer node. We backtrack to node 2 & generate another child, node 13. The path is now (1,4). The board configurations as backtracking proceed is as follows:



The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried & rejected because another queen was attacking.

In fig(b) the second queen is placed on columns 1 & 2 and finally settles on column 3. In the figure(c) the algorithm tries all four columns & is unable to place the next queen on a square. Backtracking now takes place. In fig(d) the second queen is moved to the next possible column, column 4 & the third queen is placed on column 2. The boards in fig (e), (f), (g), (h) show the remaining steps that the algorithm goes through until a solution is found.



Portion of the tree generated during backtracking.

→ Algorithm Place(k,i)

// Returns true if a queen can be placed in k^{th} row & // i^{th} column. otherwise it returns false. x[] is a // global array whose first (k-1) values have been set. // Abs(r) returns the absolute value of r.

```
{
    for j := 1 to k-1 do
        if ((x[j]=i) // two in the same column
            or (Abs(x[j]-i) = Abs(j-k)))
                // or in the same diagonal.
            then return false;
    return true;
}
```

the Above one is about the algorithm can a new queen be placed?

→ Algorithm NQueens(k,n)

// using backtracking, this procedure prints all // possible placements of n queens on an nxn // chessboard so that they are non attacking

```
{
    for i := 1 to n do
    {
        if Place(k,i) then
        {
            x[k] := i;
```

```
if(k==n) then write (x[1:n]);
else NQueens(k+1,n);
}
}
}
```

This algorithm is about all solutions to the n-queens problem.

Complexity Analysis:

$$1+n+n^2+n^3+\cdots+n^n = \frac{n^{n+1}-1}{n-1}$$

for the instance in which n=8, the state space tree contains:

$$\frac{8^{8+1}-1}{8-1} = 19,193,961 \text{ nodes.}$$

→ Sum of subsets problem

Given positive number w_i, $1 \le i \le n$, & m, this problem requires finding all subsets of w_i whose sums are 'm'.

All solutions are k-tuples, $1 \le k \le n$.

Explicit Constraints:

- $x_i \in \{j \mid j \text{ is an integer and } 1 \le j \le n\}$.

Implicit Constraints:

- No two x_i can be the same.
- the sum of the corresponding w_i's be m.
- $x_i < x_{i+1}$, $1 \le i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for ex. (1,2,4) & (1,4,2) represent the same subset).

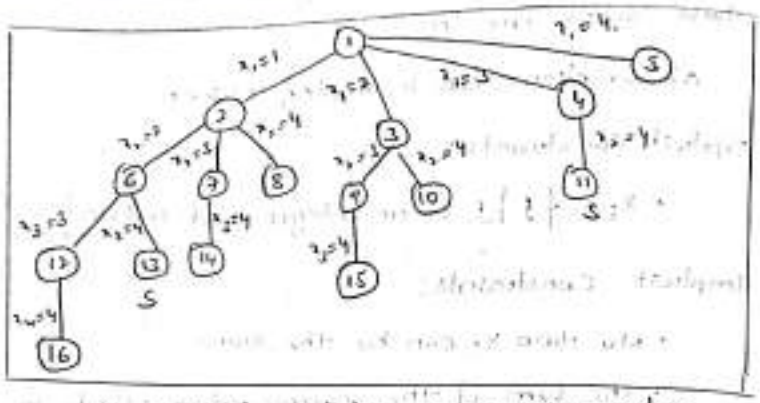
A better formulation of the problem is where the solution subset is represented by an n-tuple $(x_1, \ldots, x_n)$ such that $x_i \in \{0,1\}$.

The above solutions are then represented by (1,1,0,1) and (0,0,1,1)

for both the above formulations, the solution space is 2^n distinct tuples.

for ex. n=4, w = (11,13,4,7) and m=31, the desired subsets are (11,13,7) and (24,7)

The following figure shows a possible tree organization for a possible formulations of the solution space for the case n=4.



A possible solution space organisation for the sum of the subset problem.

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level $i+1$ node represents a value for x_i. At each node, the solution space is partitioned into sub-solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

The possible paths are (1), (1,2), (1,2,3), (1,2,3,4), (1,2,4), (1,3,4), (2), (2,3) & so on. Thus, the left most sub-tree defines all subsets

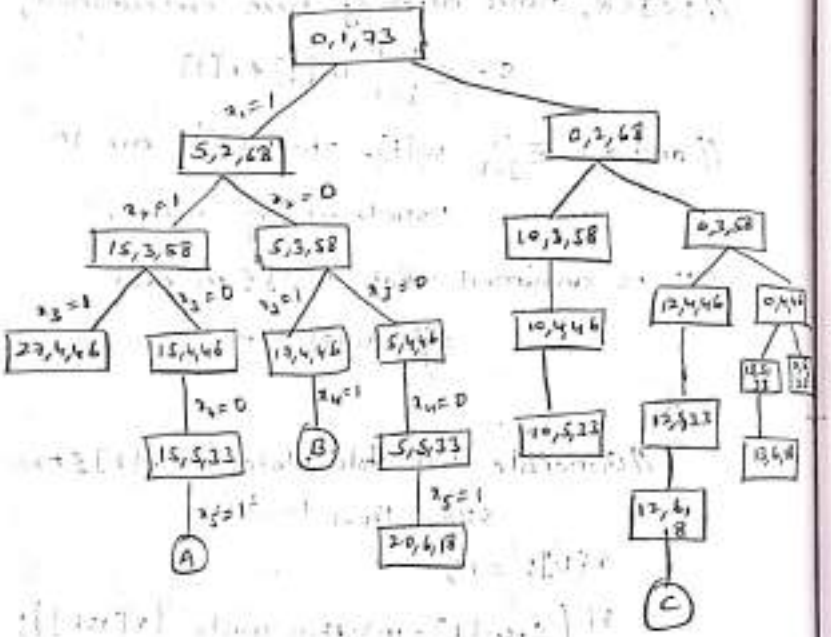
containing w_1, the next sub-tree defines all subsets containing w_2 but not w_1, & so on.

→ Recursive backtracking algorithm for sum of subsets problem.

```
Algorithm SumOfSub (s,k,r)
// find all subsets of w[1:n] that sum to m.
                The values of x[j],
// 1≤j<k, have already been determined,
                s = Σ_{j=1}^{k-1} w[j]*x[j]
// and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in
                nondecreasing order.
// It is assumed that w[1] ≤ m and
                Σ_{i=1}^{n} w[i] ≥ m.
{
  //Generate left child. Note: s+w[k] ≤ m
        since B_{k-1} is true.
  x[k]:=1;
  if (s+w[k]=m) then write (x[1:k]);
                //subset found
        //There is no recursive call here
          as w[j] > 0, 1≤j≤n.
  else if (s+w[k]+w[k+1]≤m)
        then SumOfSub (s+w[k],k+1,
                r-w[k]);
```

// Generate right child & evaluate B_k.

```
if ((s+r-w[k] ≥ m) and (s+w[k+1] ≤ m)) then
{
    x[k] := 0;
    SumOfSub(s, k+1, r-w[k]);
}
}
```



Portion of state space tree generated by SumOfSub.

→ Graph Coloring

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m-Colorability decision problem. The m-Colorability optimization problem asks for the smallest integer m for which the graph G can be colored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that indeed four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x[] to zero. The colors are represented by the integers 1, 2, ... m & the solutions are given by the n-tuple $(x_1, x_2, \ldots, x_n)$ where x_i is the color of node i.

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

→ Algorithm mcoloring(k)

// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G[1:n, 1:n]. All assignments of
// 1, 2, ..., m to the vertices of the graph such that adjacent vertices are assigned
// distinct integers are printed. k is the index of the next vertex to color.

{
  repeat
  {
    // Generate all legal assignments for x[k].
    NextValue(k); // Assign to x[k] a legal color.
    if (x[k] = 0) then return; // No new color possible
    if (k = n) then // at most m colors have been
                    // used to color the n vertices.
      write (x[1:n]);
    else mcoloring(k+1);
  } until (false);
}

→ Algorithm NextValue(k)

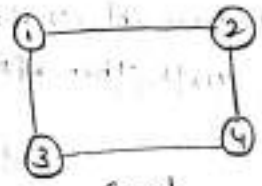
// x[0], ... x[k-1] have been assigned integer values in the range [1, m] such that
// adjacent vertices have distinct integers. A value for x[k] is determined in range
// [0, m]. x[k] is assigned the next highest numbered color while maintaining distinctness
// from the adjacent vertices of vertex k. If no such color exists, then x[k] is 0

{
  repeat
  {
    x[k] := (x[k] + 1) mod (m+1) // Next highest color
    if (x[k] = 0) then return; // All colors have been used.

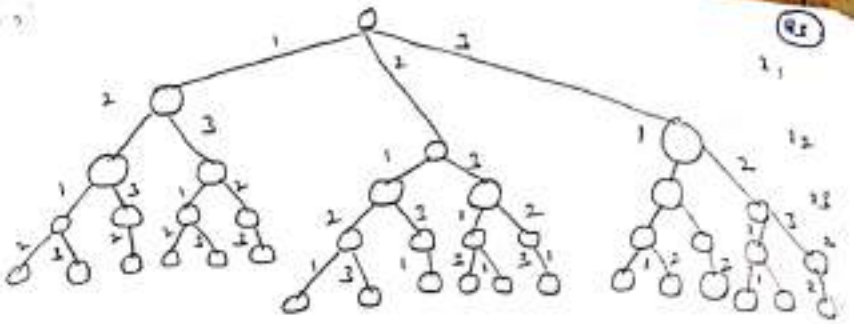
```
for j:=1 to n do
{
  // check if this color is distinct from
                adjacent colors
  if ((G[k,j] ≠ 0) & (x[k] = x[j]))
  // if (k,j) is and edge & if adj. vertices
                have the same color.
  then break;
}
if (j=n+1) then return;  // New color found
} until (false);  // otherwise try to find
                another color.
}
```

Example:

Color the graph given below with minimum number of colors by backtracking using state space tree.



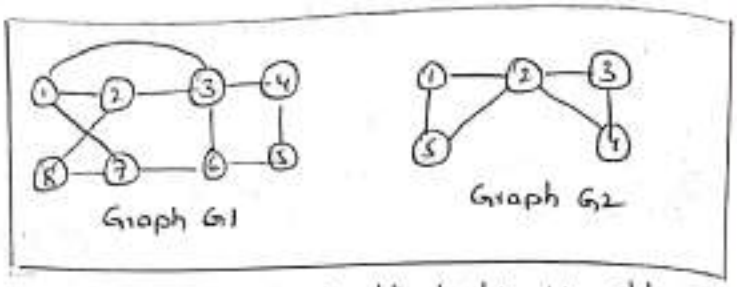
Graph.



A 4-node graph and all possible 3-colorings.

Hamiltonian Cycles:

Let $G=(V,E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other vertices of G are visited in the order $v_1, v_2, \ldots v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$, and the v_i are distinct expect for v_1 and v_{n+1}, which are equal. The graph G_1 contains the Hamiltonian cycle 1,2,8,7,6,5,4,3,1. The Graph G_2 contains no Hamiltonian cycle.



Two Graphs to Illustrate Hamiltonian cycle.

The backtracking solution vector $(x_1, \ldots x_n)$ is defined so that x_i represents the i^{th} visited vertex of the proposed cycle. If $k=1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to x_{k-1}. The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1.

Using NextValue algorithm, we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix $G[1:n, 1:n]$, then setting $x[2:n]$ to zero and $x[1]$ to 1, & then executing Hamiltonian(2).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. for the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

```
→ Algorithm Hamiltonian(k)
// this algorithm uses the recursive formulation
    of backtracking to find all the Hamiltonian
// cycles of a graph. The graph is stored as an
    adjacency matrix G[1:n, 1:n]. All cycles
                                        begin
// at node 1.
{
repeat
{       // Generate values for x[k].
    NextValue(k); // Assign a legal Next value
                     to x[k].
    if (x[k]=0) then return;
        if (k=n) then write (x[1:n]);
            else Hamiltonian(k+1)
}until(false);
} else return mincost;
```

→ Algorithm NextValue(k)

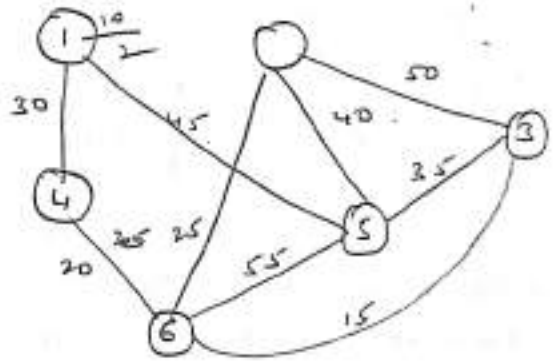
```
// x[1:k-1] is a path of k-1 distinct vertices.
//    If x[k]=0, then no vertex has as yet been
// assigned to x[k]. After execution, x[k] is
//    assigned to next highest numbered vertex
// which does not already appear in x[1:k-1] &
//    is connected by an edge to x[k-1].
// otherwise x[k]=0. If k=n, then in addition
//    x[k] is connected to x[1].
{
  repeat
  {
    x[k]:=(x[k]+1) mod (n+1); // Next Vertex.
    If (x[k]=0) then return;
    If (G[x[k-1],x[k]]≠0) then
    {  // is there any edge?
      for j:=1 to k-1 do if (x[j]=x[k]) then
          break; // check for distinctness.
      If (j=k) then // If true, then vertex
                    is distinct.
      If ((k<n) or ((k=n) & G[x[n],x[1]]≠0))
      then return;
    }
  } until(false);
}
```


Running time:

- The no. of finds is at most 2e, and the no. of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O(n+e).
- We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

Summing up the various components of the computing times, we get O(n+eloge) as asymptotic Complexity.

Example:



UNIT-III

→ Dynamic Programming

→ General Method

It is a name, coined by Richard Bellman in 1955. Dynamic programming as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision sequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called dynamic-programming recurrence equations, that enable us to solve the problem in an efficient way.

Dynamic programming is based on the Principle of optimally (also coined by Bellman). The principle of optimality states that no matter whatever the initial state & initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from

the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the Principle of optimality may not hold for some formulations of some problems. It is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations.
- Solve the dynamic-programming recurrence Equ. for the value of optimal solution
- Perform a trace back step in which the solution itself is constructed.

Dynamic programming differs from the greedy method since the greedy method produce only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal. optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every

time a sub-problem is encountered.

The divide & conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independly. In this principle is carried to an extreme; when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems. Care is to be taken to avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity. In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Two difficulties may arise in any application of dynamic programming:

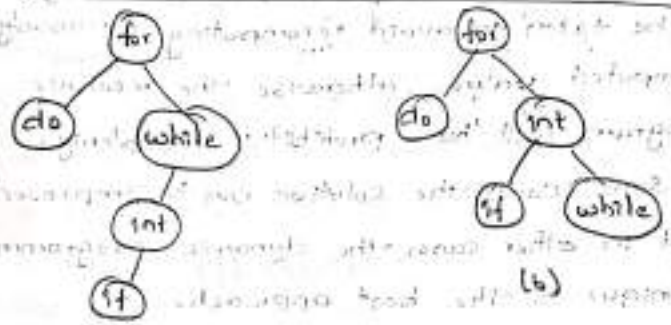
1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.
2. The no. of small problems to solve may be un-acceptably large.

There is no characterized precisely which Problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seem to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

→ Applications

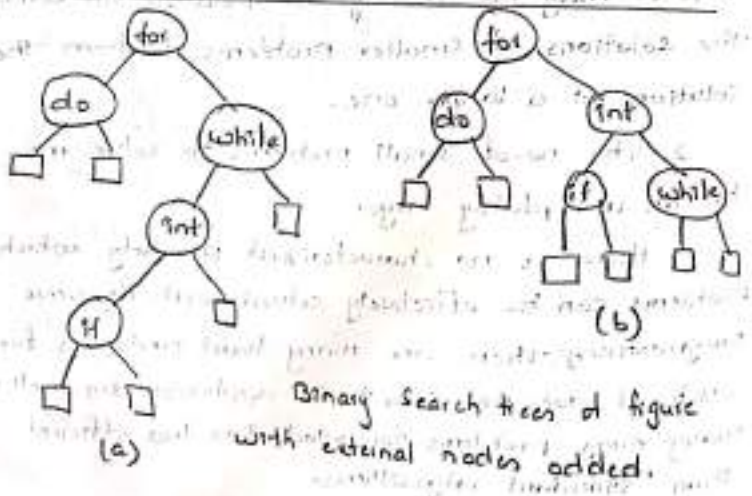
⇒ Optimal Binary Search Tree:

In Computer Science, an optimal binary search tree (optimal BST), sometimes called a weight-balanced binary tree, is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities).



(a) (b)

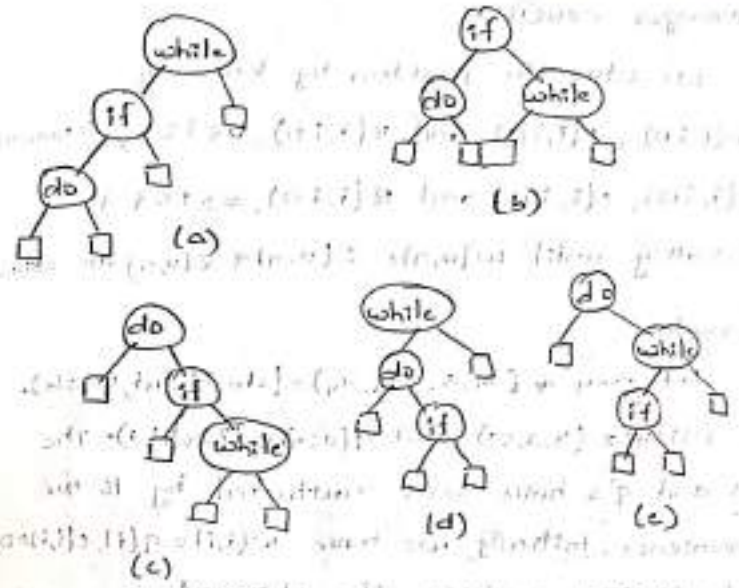
Two possible binary search trees.



(a) (b)

Binary search trees of figure with external nodes added.

The no. of external nodes are same in both sides.



possible binary search trees for the identifier set {do, if, while}

The $C(i,j)$ can be computed as:

$$C(i,j) = \min_{i<k\le j} \{C(i,k-1) + C(k,j) + P(k) + w(i,k-1) + w(k,j)\}$$

$$= \min_{i<k\le j} \{C(i,k-1) + C(k,j)\} + w(i,j) \quad \text{— (1)}$$

where $w(i,j) = P(j) + Q(j) + w(i,j-1)$ — (2)

Initially $c(i,i) = 0$ and $w(i,i) = Q(i)$ for $0 \le i < n$.

$c(i,j)$ is the cost of the optimal binary search tree 'T_{ij}' during computation we record the root $R(i,j)$ of each tree 'T_{ij}'. Then an optimal binary search tree may be constructed from these.

$R(i,j)$, $R(i,j)$ is the value of 'k' that minimizes equ①.

We solve the problem by knowing $w[i,i+1)$, $c[i,i+1)$ and $R[i,i+1)$, $0 \le i < 4$; knowing $w[i,i+2)$, $c[i,i+2)$ and $R[i,i+2)$, $0 \le i < 3$ & repeating until $w[0,n)$, $c[0,n)$ & $R[0,n)$ are obtained.

Example:

Let $n=4$ & $(a_1, a_2, a_3, a_4) = (do, if, int, while)$. Let $p[1:4) = (3,3,1,1)$ and $q[0:4) = (2,3,1,1,1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w[i,i) = q[i]$, $c[i,i) = 0$ and $r[i,i) = 0$, $0 \le i \le 4$. The observation $w[i,j) = p[j] + q[j] + w[i,j-1)$, we get

$w[0,1) = p[1] + q[1] + w[0,0) = 8$

$c[0,1) = w[0,1) + \min\{c[0,0) + c[1,1)\} = 8$

$r[0,1) = 1$

$w[1,2) = p[2] + q[2] + w[1,1) = 7$

$c[1,2) = w[1,2) + \min\{c[1,1) + c[2,2)\} = 7$

$r[0,2) = 2$

$w[2,3) = p[3] + q[3] + w[2,2) = 3$

$c[2,3) = w[2,3) + \min\{c[2,2) + c[3,3)\} = 3$

$r[2,3) = 3$

$w[3,4) = p[4] + q[4] + w[3,3) = 3$

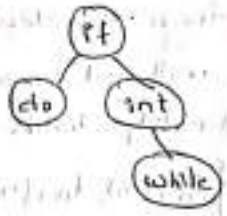
$c[3,4) = w[3,4) + \min\{c[3,3) + c[4,4)\} = 3$

$r[3,4) = 4$

Knowing $w[i,i+1)$ and $c[i,i+1)$, $0 \le i < 4$, we can again use Equ #12 to compute $w[i,i+2)$, $c[i,i+2)$ & $r[i,i+2)$, $0 \le i < 3$. This process can be repeated until $w[0,4)$, $c[0,4)$ & $r[0,4)$ are obtained.

	0	1	2	3	4
0	$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
1	$w_{01}=8$ $c_{01}=8$ $r_{01}=1$	$w_{12}=7$ $c_{12}=7$ $r_{12}=2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=4$	
2	$w_{02}=12$ $c_{02}=19$ $r_{02}=1$	$w_{13}=9$ $c_{13}=12$ $r_{13}=2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=3$		
3	$w_{03}=14$ $c_{03}=25$ $r_{03}=2$	$w_{14}=11$ $c_{14}=19$ $r_{14}=2$			
4	$w_{04}=16$ $c_{04}=32$ $r_{04}=2$				

computation of $c(0,4)$, $w(0,4)$ and $r(0,4)$



optimal search tree for example.

→ 0/1 knapsack problem

We are given n objects and a knapsack. Each object i has a positive weight w_i & a positive value v_i. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1, x_2 \dots x_n$. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it.

Let us assume that decisions on the x_i are made in the order $x_n, x_{n-1}, \dots x_1$ following a decision on x_n, we may be in one of two possible states: the capacity remaining in m - w_n & a profit of p_n has accured. It is clear that the remaining decisions $x_{n-1}, \dots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n. otherwise x_n, x_1 will not be optimal. Hence, the principal of optimality holds.

$$F_n(m) = \max\{f_{n-1}(m),\ f_{n-1}(m-w_n)+p_n\} \quad -①$$

for arbitary $f_i(y)$, $i \geq 0$ this equ generalizes to:

$$F_i(y) = \max\{f_{i-1}(y),\ f_{i-1}(y-w_i)+p_i\} \quad -②$$

Eq② can be solved for $f_n(m)$ by beginning with the knowledge $f_0(y)=0$ for all y and $f_i(y)=-\infty$, $y<0$. Then $f_1, f_2, \dots f_n$ can be successively computed using equ②.

When the w_i's are integer, we need to compute $f_i(y)$ for integer y, $0 \leq y \leq m$. since $f_i(y) = -\infty$ for $y<0$, these function values need not be computed explicitly. since each f_i can be computed from f_{i-1} in $\Theta(m)$ time, it takes $\Theta(m,n)$ time to compute f_n. when the w_i's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. so f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit $\Theta(m,n)$ computation of f_n may not be the most efficient computation. so, we explore an alternative method for both cases.

The $f_i(y)$ is an ascending step function; i.e there are a finite number of y's, $0 = y_1 < y_2 < \dots < y_k$, such that $f_i(y_1) < f_i(y_2) < \dots < f_i(y_k)$; & $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f_i(y_k)$, $y \geq y_k$; & $f_i(y) = f_i(y_j)$, $y_j \leq y \leq y_{j+1}$. so we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. we use the

ordered set $S^i = \{(f_i(y_j), y_j) \mid 1 \le j \le k\}$ to represent $f_i(y)$. Each number of S^i is a pair (P, w), where $P = f_i(y_j)$ and $w = y_j$. Notice that $S^0 = \{(0,0)\}$. We can compute S^{i+1} from S^i by first computing:

$$S^i_1 = \{(P, w) \mid (P - P_i, w - w_i) \in S^i\}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S^i_1 together. Note that if S^{i+1} contains two pairs (P_j, w_j) & (P_k, w_k) with the property that $P_j \le P_k$ and $w_j \ge w_k$, then the pair (P_j, w_j) can be discarded because of eqn (3). Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above (P_k, w_k) dominates (P_j, w_j).

Example 1:

Consider the knapsack instance $n=3$, $(w_1, w_2, w_3) = (2,3,4)$, $(P_1, P_2, P_3) = (1,2,5)$ and $M = 6$.

solution:

Initially $f_0(x) = 0$, for all x and $f_i(x) = -\infty$ if $x < 0$

$$F_n(M) = \max\{f_{n-1}(M), f_{n-1}(M - w_n) + P_n\}$$

$$F_3(6) = \max\{f_2(6), f_2(6-4) + 5\} = \max\{f_2(6), f_2(2) + 5\}$$

$$F_2(6) = \max\{f_1(6), f_1(6-3) + 2\} = \max\{f_1(6), f_1(3) + 2\}$$

$$F_1(6) = \max\{f_0(6), f_0(6-2) + 1\} = \max\{0, 0+1\} = 1$$

$$F_1(3) = \max\{f_0(3), f_0(3-2) + 1\} = \max\{0, 0+1\} = 1$$

$\therefore F_2(6) = \max(1, 1+2) = 3$

$$F_2(2) = \max\{f_1(2), f_1(2-3) + 2\} = \max\{f_1(2), -\infty + 2\}$$

$$F_1(2) = \max\{f_0(2), f_0(2-2) + 1\} = \max\{0, 0+1\} = 1$$

$$F_2(2) = \max\{1, -\infty + 2\} = 1$$

finally, $f_3(6) = \max\{3, 1+5\} = 6$

other solution:

for the given data we have:

$S^0 = \{(0,0)\}$; $S^0_1 = \{(1,2)\}$

$S^1 = (S^0 \cup S^0_1) = \{(0,0), (1,2)\}$

$x - 2 = 0 \Rightarrow x = 2$ $\quad y - 3 = 0 \Rightarrow y = 3$

$x - 2 = 1 \Rightarrow x = 3$ $\quad y - 3 = 2 \Rightarrow y = 5$

$S^1_1 = \{(2,3), (3,5)\}$

$S^2 = (S^1 \cup S^1_1) = \{(0,0), (1,2), (2,3), (3,5)\}$

$x-5=0 \Rightarrow x=5$ $\quad y-4=0 \Rightarrow y=4$

$x-5=1 \Rightarrow x=6$ $\quad y-4=2 \Rightarrow y=6$

$x-5=2 \Rightarrow x=7$ $\quad y-4=3 \Rightarrow y=7$

$x-5=3 \Rightarrow x=8$ $\quad y-4=5 \Rightarrow y=9$

$S^2_1 = \{(5,4), (6,6), (7,7), (8,9)\}$

$S^3 = (S^2 \cup S^2_1) = \{(0,0), (1,2), (2,3), (3,5), (5,4), (6,6), (7,7), (8,9)\}$

By applying Dominance rule,

$S^3 = (S^2 \cup S^2_1) = \{(0,0), (1,2), (2,3), (5,4), (6,6)\}$

from (6,6) we can infer that the maximum profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$

→ All pairs shortest path

In this problem, we are to find a shortest path between every pair of vertices in a directed Graph G. That is, for every pair of vertices (i,j) we are to find a shortest path from i to j as well as one from j to i. these two paths are the same when G is undirected.

when no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.

the all pairs shortest path problem is to determine a matrix A such that A(i,j) is the length of a shortest path from i to j. the matrix A can be obtained by solving n single-source problems using the algorithm shortest paths. since each application of this procedure requires $O(n^2)$ time, the matrix A can be obtained in $O(n^3)$ time.

The dynamic programming solution, called Floyd's algorithm, runs in $O(n^3)$ time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).

the shortest i to j path in G, $i \neq j$ originates at vertex i and goes through some intermediate vertices (possibly none) & terminates at vertex j. If k is an intermediate vertex on this shortest path, then the subpaths from i to k & from k to j must be shortest paths from i to k and k to j, respectively. otherwise, the i to j path is not of minimum length. so, the principle of optimality holds. Let $A^k(i,j)$ represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:

$$A^k(i,j) = \min\{\min_{1 \le k \le n}\{A^{k-1}(i,k) + A^{k-1}(k,j)\}, c(i,j)\}$$

→ Algorithm All paths (Cost, A, n)

// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices, A[i,j] is the cost of a shortest path from vertex
// i to vertex j. cost[i,j] = 0.0, for $1 \le i \le n$

{

for i := 1 to n do

for j := 1 to n do

A[i,j] := cost[i,j]; // copy cost into A.

for k := 1 to n do

for i := 1 to n do

for j := 1 to n do

A[i,j] := min(A[i,j], A[i,k] + A[k,j]);

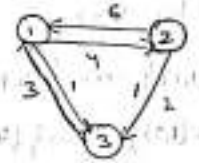
}

Complexity Analysis

A Dynamic programming algorithm based on this recurrence involves in calculating n+1 matrices, each of size n x n. therefore, the algorithm has a complexity of $O(n^3)$.

Example 1

Given a weighted digraph $G = (V, E)$ with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.



Cost Adjacency matrix $(A^0) = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$

General formula: $\min_{1 \le k \le n} \{ A^{k-1}(i,k) + A^{k-1}(k,j) \}, c(i,j) \}$

Solve the problem for diff values of k=1,2 & 3.

Step 1: Solving the equation for, k=1;

$A^1(1,1) = \min\{(A^0(1,1) + A^0(1,1)), c(1,1)\} = \min\{0+0, 0\} = 0$

$A^1(1,2) = \min\{(A^0(1,1) + A^0(1,2)), c(1,2)\} = \min\{(0+4), 4\} = 4$

$A^1(1,3) = \min\{(A^0(1,1) + A^0(1,3)), c(1,3)\} = \min\{(0+11), 11\} = 11$

$A^1(2,1) = \min\{(A^0(2,1) + A^0(1,1)), c(2,1)\} = \min\{(6+0), 6\} = 6$

$A^1(2,2) = \min\{(A^0(2,1) + A^0(1,2)), c(2,2)\} = \min\{(6+4), 0\} = 0$

$A^1(2,3) = \min\{(A^0(2,1) + A^0(1,3)), c(2,3)\} = \min\{(6+11), 2\} = 2$

$A^1(3,1) = \min\{(A^0(3,1) + A^0(1,1)), c(3,1)\} = \min\{(3+0), 3\} = 3$

$A^1(3,2) = \min\{(A^0(3,1) + A^0(1,2)), c(3,2)\} = \min\{(3+4), \infty\} = 7$

$A^1(3,3) = \min\{(A^0(3,1) + A^0(1,3)), c(3,3)\} = \min\{(3+11), 0\} = 0$

$$A^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 2: Solve the equation for k=2;

$A^2(1,1) = \min\{(A^1(1,2) + A^1(2,1)), c(1,1)\} = \min\{(4+6), 0\} = 0$

$A^2(1,2) = \min\{(A^1(1,2) + A^1(2,2)), c(1,2)\} = \min\{(4+0), 4\} = 4$

$A^2(1,3) = \min\{(A^1(1,2) + A^1(2,3)), c(1,3)\} = \min\{(4+2), 6\} = 6$

$A^2(2,1) = \min\{(A(2,2) + A(2,1)), c(2,1)\} = \min\{(0+6, 6\} = 6$

$A^2(2,2) = \min\{(A(2,2) + A(2,2)), c(2,2)\} = \min\{(0+0, 0\} = 0$

$A^2(2,3) = \min\{(A(2,2) + A(2,3)), c(2,3)\} = \min\{(0+2), 2\} = 2$

$A^2(3,1) = \min\{(A(3,2) + A(2,1)), c(3,1)\} = \min\{(7+6) + 3\} = 3$

$A^2(3,2) = \min\{(A(3,2) + A(2,2)), c(3,2)\} = \min\{(7+0), 7\} = 7$

$A^2(3,3) = \min\{(A(3,2) + A(2,3)), c(3,3)\} = \min\{(7+2), 0\} = 0$

$$A^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

Step 3: Solving the equation for k=3

$A^3(1,1) = \min\{A^2(1,3) + A^2(3,1), c(1,1)\} = \min\{(6+3), 0\} = 0$

$A^3(1,2) = \min\{A^2(1,3) + A^2(3,2), c(1,2)\} = \min\{(6+7), 4\} = 4$

$A^3(1,3) = \min\{A^2(1,3) + A^2(3,3), c(1,3)\} = \min\{(6+0), 6\} = 6$

$A^3(2,1) = \min\{A^2(2,3) + A^2(3,1), c(2,1)\} = \min\{(2+3), 6\} = 5$

$A^3(2,2) = \min\{A^2(2,3) + A^2(3,2), c(2,2)\} = \min\{(2+7), 0\} = 0$

$A^3(2,3) = \min\{A^2(2,3) + A^2(3,3), c(2,3)\} = \min\{(2+0), 2\} = 2$

$A^3(3,1) = \min\{A^2(3,3) + A^2(3,1), c(3,1)\} = \min\{(0+3), 3\} = 3$

$A^3(3,2) = \min\{A^2(3,3) + A^2(3,2), c(3,2)\} = \min\{(0+7), 7\} = 7$

$A^3(3,3) = \min\{A^2(3,3) + A^2(3,3), c(3,3)\} = \min\{(0+0), 0\} = 0$

$$A^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

→ Travelling Salesperson Problem

Let $G=(V,E)$ be a directed graph with edge costs c_{ij}. The variable c_{ij} is defined such that $c_{ij}>0$ for all i and j and $c_{ij}=\infty$ if $\langle i,j\rangle \notin E$. Let $|V|=n$ and assume $n>1$. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The travelling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts & ends at vertex 1.

Let $g(1,S)$ be the length of shortest path starting at vertex 1, going through all vertices in S, and terminating at vertex 1. The function $g(1,V-\{1\})$ is the length of an optimal salesperson tour. From the principal of optimality it follows that:

$$g(1,V-\{1\}) = \min_{2\le k\le n} \{c_{1k} + g(k, V-\{1,k\})\} \quad -(1)$$

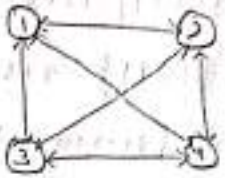
Generalizing eqn (1), we obtain (for $i \notin S$)

$$g(i,S) = \min_{j\in S} \{c_{ij} + g(j, S-\{j\})\} \quad -(2)$$

The eqn can be solved for $g(1,V-1)$ if we know $g(k, V-\{1,k\})$ for all choices of k.

Example:

For the following graph find minimum cost tour for the travelling salesperson problem.



The cost adjacency matrix =
$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Let us start the tour from vertex 1:

$$g(1,V-\{1\}) = \min_{2\le k\le n} \{c_{1k} + g(k, V-\{1,k\})\} \quad -(1)$$

More generally writing:

$$g(i,S) = \min\{c_{ij} + g(j, S-\{j\})\} \quad -(2)$$

clearly, $g(i,\phi) = c_{i1}$, $1\le i\le n$. so,

$$g(2,\phi) = c_{21} = 5$$
$$g(3,\phi) = c_{31} = 6$$
$$g(4,\phi) = c_{41} = 8$$

using eqn (2) we obtain:

$$g(1,\{2,3,4\}) = \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\}$$

$g(2,\{3,4\}) = \min\{c_{23}+g(3,\{4\}),\ c_{24}+g(4,\{3\})\}$

$= \min\{9+g(3,\{4\}),\ 10+g(4,\{3\})\}$

$g(3,\{4\}) = \min\{c_{34}+g(4,\phi)\} = 12+8 = 20$

$g(4,\{3\}) = \min\{c_{43}+g(3,\phi)\} = 9+6 = 15$

Therefore, $g(2,\{3,4\}) = \min\{9+20, 10+15\} = \min\{29,25\} = 25$

$g(3,\{2,4\}) = \min\{(c_{32}+g(2,\{4\}),\ (c_{34}+g(4,\{2\}))\}$

$g(2,\{4\}) = \min\{c_{24}+g(4,\phi)\} = 10+8 = 18$

$g(4,\{2\}) = \min\{c_{42}+g(2,\phi)\} = 8+5 = 13$

Therefore $g(3,\{2,4\}) = \min\{13+18, 12+13\} = \min\{41,25\} = 25$

$g(4,\{2,3\}) = \min\{c_{42}+g(2,\{3\}),\ c_{43}+g(3,\{2\})\}$

$g(2,\{3\}) = \min\{c_{23}+g(3,\phi)\} = 9+6 = 15$

$g(3,\{2\}) = \min\{c_{32}+g(2,\phi)\} = 13+5 = 18$

Therefore $g(4,\{2,3\}) = \min\{8+15, 9+18\} = \min\{23,27\} = 23$

$g(1,\{2,3,4\}) = \min\{c_{12}+g(2,\{3,4\}),\ c_{13}+g(3,\{2,4\}),$

$c_{14}+g(4,\{2,3\})\}$

$= \min\{10+25,\ 15+25,\ 20+23\}$

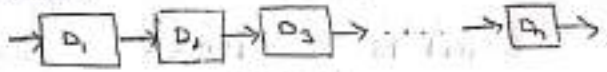
$= \min\{35, 40, 43\} = 35$

The optimal tour for the graph has length = 35

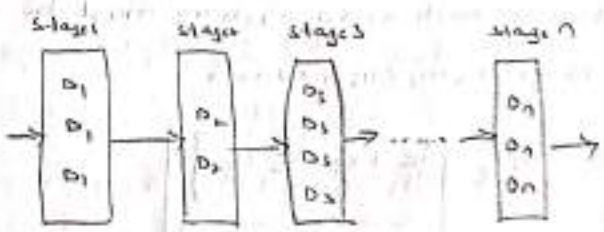
the optimal tour is: 1, 2, 4, 3, 1

→ Reliability Design:

The problem is to design a system that is composed of several devices connected in series. let r_i be the reliability of device D_i (that is r_i is the probability that device i will function properly) then the reliability of the entire system is πr_i. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good. for example, if $n=10$ & $r_i = 0.91$, $1 \le i \le 10$, then $\pi r_i = .904$. Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel.



n devices D_i, $1 \le i \le n$ connected in series



Multiple devices connected in parallel in each stage.

If stage i contains m_i copies of device D_i. Then the probability that all m_i have a malfunctions is $(1-r_i)^{m_i}$. Hence the reliability of stage i becomes $1-(1-r_i)^{m_i}$.

The reliability of stage 'i' is given by a function $\phi_i(m_i)$.

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed

We wish to solve:

$$\text{Maximize} \prod_{1\le i\le n} \phi_i(m_i)$$

$$\text{subject to} \sum_{1\le i\le n} c_i m_i \le c$$

$m_i \ge 1$ and integer, $1\le i\le n$

Assume each $c_i > 0$, each m_i must be in the range $1\le m_i\le u_i$, where

$$u_i = \left\lfloor \frac{\left(c + c_i - \sum_{1}^{n} c_j\right)}{c_i} \right\rfloor$$

The upper bound u_i follows from the observation that $m_i \ge 1$

An optimal solution $m_1, m_2, \ldots, m_n$ is the result of a sequence of decisions, one decision for each m_i.

Let $f_i(x)$ represent the max value of

$$\prod_{1\le j\le i} \phi(m_j)$$

subject to the constraints:

$$\sum_{1\le j\le i} c_j m_j \le x \quad \text{and} \quad 1\le m_j\le u_j,\ 1\le j\le i$$

Example:

Design a three stage system with device types D_1, D_2 & D_3. the costs are \$30, \$15 & \$20. respectively. the cost of the system is to be no more than \$105. the reliability of each device is 0.9, 0.8 and 0.5 respectively.

solution:

We assume that if stage i has m_i devices of type i in parallel, then ϕ

$$\phi_i(m_i) = 1-(1-r_i)^{m_i}$$

Since, we assume each $c_i > 0$, each m_i must be in the range $1\le m_i\le u_i$. where:

$$u_i = \left[\frac{\left[C + c_i - \sum_{1}^{n} c_i \right]}{c_i} \right]$$

using the above equ Compute $u_1, u_2, \& u_3$.

$$u_1 = \frac{105+30-(30+15+20)}{30} = \frac{70}{30} = 2$$

$$u_2 = \frac{105+15-(30+15+20)}{15} = \frac{55}{15} = 3$$

$$u_3 = \frac{105+20-(30+15+20)}{20} = \frac{60}{20} = 3$$

we use $s_j^i \rightarrow i$: stage number and

j: no. of devices in stage $i = m_i$

$s^0 = \{ t_0(x), x \}$ initially $t_0(x) = 1$ and

$x > 0$, so $s^0 = \{1, 0\}$

Compute s^1, s^2 and s^3 as follows:

s^1 = depends on u_1 value as $u_1 = 2$, so

$s^1 = \{ s_1^1, s_2^1 \}$

s^2 = depends on u_2 value as $u_2 = 3$, so

$s^2 = \{ s_1^2, s_2^2, s_3^2 \}$

s^3 = depends on u_3 value, as $u_3 = 3$, so

$s^3 = \{ s_1^3, s_2^3, s_3^3 \}$

Now, find $s_1^1 = \{ (f_1(x), x) \}$

$f_1(x) = \{ \phi_1(1) f_0(1), \phi_1(2) f_0(1) \}$ with devices

$m_1 = 1$ and $m_2 = 2$

Compute $\phi_1(1)$ and $\phi_1(2)$ using the formula:

$$\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$$

$$\phi_1(1) = 1 - (1 - r_1)^{m_1} = 1 - (1 - 0.9)^1 = 0.9$$

$$\phi_1(2) = 1 - (1 - 0.9)^2 = 0.99$$

$s_1^1 = \{ f_1(x), x \} = (0.9, 30)$

$s_2^1 = \{ 0.99, 30+30 \} = (0.99, 60)$

Therefore, $s^1 = \{ (0.9, 30), (0.99, 60) \}$

Next find $s_1^2 = \{ (f_2(x), x) \}$

$f_2(x) = \{ \phi_2(1) * f_1(), \phi_2(2) * f_1(), \phi_2(3) * f_1() \}$

$\phi(1) = 1 - (1 - r_1)_{m_1} = 1 - (1 - 0.8) = 1 - 0.2 = 0.8$

$\phi(2) = 1 - (1 - 0.8)^2 = 0.96$

$\phi(3) = 1 - (1 - 0.8)^3 = 0.992$

$S^2_1 = \{(0.8(0.9), 30+15), (0.8(0.99), 60+15)\} =$

$= \{(0.72, 45), (0.792, 75)\}$

$S^2_2 = \{(0.96(0.9), 30+15+15), (0.96(0.99), 60+15+15)\}$

$= \{(0.864, 60), (0.9504, 90)\}$

$S^2_3 = \{(0.992(0.9), 30+15+15+15), (0.992(0.99), 60+15+15+15)\}$

$= \{(0.8928, 75), (0.98208, 105)\}$

$S^2 = \{S^2_1, S^2_2, S^2_3\}$

By applying Dominance rule to S^2:

$\therefore S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

Dominance Rule:

If S^i contains two pairs $(t_1, r_1), (t_2, r_2)$ with the property that $t_1 \geq t_2$ and $r_1 \leq r_2$, then (t_1, r_1) dominates (t_2, r_2) hence by dominance rule (t_2, r_2) can be discarded. Discarding or pruning rules such as the one above is known as dominance rule. Dominating tuples will be present in S^i & Dominated tuples has to be discarded from

case 1: if $t_1 \leq t_2$ and $r_1 > r_2$ then discard (t_1, r_1)

case 2: if $t_1 \geq t_2$ and $r_1 < r_2$ then discard (t_2, r_2)

case 3: otherwise simply write (t_1, r_1)

$S_2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

$\phi_3(1) = 1 - (1-r_1)^{m_1} = 1 - (1-0.5)^1 = 1 - 0.5 = 0.5$

$\phi_3(2) = 1 - (1-0.5)^2 = 0.75$

$\phi_3(3) = 1 - (1-0.5)^3 = 0.875$

$S^3_1 = \{(0.5(0.72), 45+20), (0.5(0.864), 60+20), (0.5(0.8928), 75+20)\}$

$S^3_1 = \{(0.36, 65), (0.437, 80), (0.4464, 95)\}$

$S^3_2 = \{(0.75(0.72), 45+20+20), (0.75(0.864), 60+20+20), (0.75(0.8928), 75+20+20)\}$

$= \{(0.54, 85), (0.648, 100), (0.6696, 115)\}$

$S^3_3 = \{(0.875(0.72), 45+20+20+20), (0.875(0.864), 60+20+20+20), (0.875(0.8928), 75+20+20+20)\}$

$$S^2_3 = \{(0.63, 105), (1.756, 120), (0.7812, 135)\}$$

If cost exceeds 105, remove that tuples.

$$S^3 = \{(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)\}$$

The best design has a reliability of 0.648 and a cost of 100. Tracing back for the solution through S^1 s we can determine that $m_3 = 2$, $m_2 = 2$ and $m_1 = 1$

UNIT - IV

→ Greedy Method

→ General Method

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. this is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure.

Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called subset paradigm.

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is ordering paradigm. Some problems

Control Abstraction

```
Algorithm Greedy(a,n)
// a[1:n] contains the 'n' inputs
{
    solution := φ;  // initialize the soln to empty
    for i:=1 to n do
    {
        x := select(a);
        if feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```


Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from 'a', removes it and assigns its value to 'x'. Feasible is a Boolean valued function, which determines if 'x' can be included into the solution vector. The function Union combines 'x' with solution and updates the objective function.

→ Applications

→ Job sequencing with Deadlines:

When we are given a set of 'n' jobs. Associated with each job i, deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit P_i is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array $d[1:n]$ is used to store the deadlines of the order of their P-values. The set of jobs $J[1:k]$ such that $J[r], 1 \leq r \leq k$ and the jobs in 'J' and

$d(J[1]) \le d(J[2]) \le \dots \le d(J[k])$. To test whether $J \cup \{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d[J[r]] \le r$, $1 \le r \le k+1$.

Example:

Let $n=4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$. The feasible solution and their values are:

S. No	Feasible solution	Processing sequence	Value	Remarks
1	1, 2	2, 1	110	
2	1, 3	1, 3 or 3, 1	115	
3	1, 4	4, 1	127	optimal
4	2, 3	2, 3	25	
5	3, 4	4, 3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

The algorithm constructs an optimal set of J of jobs that can be processed by their deadlines.


```
Algorithm GreedyJob (d, J, n)
// J is a set of jobs that can be completed
                by their deadlines.
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
                        by their deadlines)
        then J := J ∪ {i};
    }
}
```

→ knapsack problem:

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight w_i and the knapsack has a capacity 'm'. If a fraction x_i, $0 < x_i < 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'. The problem is stated as:

maximize $\sum_{i=1}^{n} p_i x_i$

subject to $\sum_{i=1}^{n} a_i x_i \leq M$

where, $0 \leq x_i \leq 1$ and $1 \leq i \leq n$

The profits & weights are positive numbers. If the objects are already been sorted into non-increasing order of $p[i]/w[i]$ then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack(m,n)

```
// p[1:n] and w[1:n] contains the profits & weights respectively of
// objects ordered so that p[i]/w[i] > p[i+1]/w[i+1]
// m is the knapsack size and x[1:n] is the solution vector.
{
    for i:=1 to n do x[i]:=0.0  // initialize x
    u:=m;
    for i:=1 to n do
    {
        if (w(i)>u) then break;
        x[i]:=1.0; u:=u-w[i];
    }
    if (i<=n) then x[i]:=u/w[i];
}
```


Running time:

The objects are to be sorted into non-decreasing order of p_i/w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only $O(n)$ time.

Example

Consider the following instance of the knapsack problem: $n=3$, $m=20$, $(p_1,p_2,p_3)=(25,24,15)$ and $(w_1,w_2,w_3)=(18,15,10)$.

1. First, we try to fill the knapsack by selecting the objects in some order:

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
$1/2$	$1/3$	$1/4$	$18\times1/2+15\times1/3+10\times1/4$ $=16.5$	$25\times1/2+24\times1/3+15\times1/4=24.25$
		1		

2. Select the object with the max profit first ($p=25$). so $x_1=1$ and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit ($p=24$). so, $x_2=2/15$.

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
1	$\frac{2}{15}$	0	$18\times1+15\times\frac{2}{15}=20$	$25\times1+24\times\frac{2}{15}=28.2$

3. Considering the objects in the order of non-decreasing weights w_i:

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	$\frac{2}{3}$	1	$15\times 2/3 + 10\times 1 = 20$	$24\times 2/3 + 15\times 1 = 31$

4. Consider the objects in order of ratio p_i/w_i.

p_1/w_1	p_2/w_2	p_3/w_3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i/w_i. Select the object with max p_i/w_i ratio, so $x_2=1$ & Profit earned is 24. Now, only 5 unit of space is left, select the object with next largest p_i/w_i ratio, so $x_3 = 1/2$ and the Profit earned is 7.5

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	1	1/2	$15\times 1 + 10\times 1/2 = 20$	$24\times 1 + 15\times 1/2 = 31.5$

this solution is the optimal solution.

→ minimum-cost spanning Trees: Prim's Algorithm

A given graph can have many spanning trees. from these many spanning trees, we have to select a cheapest one. this tree is called minimal cost spanning tree.

It is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). It is a spanning tree for which the sum of the edge labels is as small as possible.

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In this algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

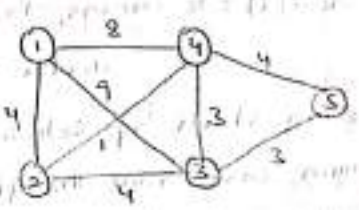
```
Algorithm prim(E, cost, n, t)
// E is the set of edges in G. Cost [1:n, 1:n]
                              is the cost
// adjacency matrix of an n vertex graph
                    such that cost [i,j] is
// either a positive real number or ∞ if
                    no edge (i,j) exists.
// A minimum spanning tree is computed &
                              stored as a set of
// edges in the array t[1:n-1, 1:2].
                    (t[1,1], t[1,2]) is an edge in
// the minimum-cost spanning tree. the
                    final cost is returned.
{
  Let (k,l) be an edge of minimum cost in E;
  mincost := cost[k,l];
  t[1,1] := k; t[1,2] := l;
  for i := 1 to n do // Initialize near
    if (cost[i,l] < cost[i,k]) then
                    near[i] := l;
    else near[i] := k;
  near[k] := near[l] := 0;
  for i := 2 to n-1 do // find n-2 additional
                    edges for t.
  {
    Let j be an index such that
                    near[j] ≠ 0 and
```



```
    cost[j, near[j]] is minimum;
    t[i,1] := j; t[i,2] := near[j];
    mincost := mincost + cost[j, near[j]];
    near[j] := 0
    for k := 1 to n do      // update near[].
      if ((near[k] ≠ 0) and (cost[k, near[k]] >
                    cost[k,j]))
        then near[k] := j;
  }
  return mincost;
}
```

Example:

Considering the following graph, find the minimal spanning tree using prim's algorithm.



the cost adjacency matrix is

$$\begin{bmatrix} \infty & 4 & 9 & 8 & \infty \\ 4 & \infty & 4 & 1 & \infty \\ 9 & 4 & \infty & 3 & 3 \\ 8 & 1 & 3 & \infty & 4 \\ \infty & \infty & 3 & 4 & \infty \end{bmatrix}$$

The minimal spanning tree obtained as:

Vertex 1	Vertex 2
2	4
3	4
5	3
1	2

The cost of minimal spanning tree = 11.

The steps as per the algorithm are as follows:

Algorithm near[j] = k means, the nearest vertex to j is k.

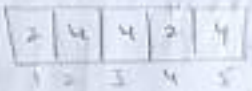
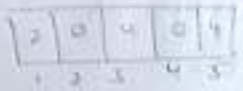
The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (3,4).

k=3, j=4

min cost = cost (3,4) = 1

T[1,1] = 2

T[1,2] = 4

	Near matrix	Edges added to min spanning tree:
for i=1 to 5 Begin j=1 is cost (1,4) < cost (1,2) 8<4, NO than near(1)=2		T[1,1]=2 T[1,2]=4
i=2 is cost (2,4) < cost (2,2) 1<2, yes so near[2]=4		
i=3 is cost (3,4) < cost (3,2) 1<4, yes so near[3]=4		
i=4 is cost (4,4) < cost (4,2) 2<1, no so near[4]=2		
i=5 is cost (5,4) < cost (5,2) 4<2, yes so near[5]=4		
end near[k]=near[1]=0 near[3] near[4]=0		

for i=2 to n-1(4) do

i=2

for j=1 to 5

j=1

near(1) ≠ 0 and

cost(1, near(1)) ≠ 0

and cost(1,2)=4

j=2

near(2)=0

j=3

is near(3) ≠ 0 4≠0

& cost(3,4)=3

j=4

near(4)=0

j=5

is near(5)≠0 4≠0

and cost(4,5)=4

select min cost from the above obtained costs, which is 3 & corresponding j=3

min cost = 1 + cost(3,4)

= 1+3 = 4

T(2,1)=3

T(2,2)=4

Near[j]=0

i.e. near(3)=0

2	0	0	0	4
1	2	3	4	5

T(2,1)=3

T(2,2)=4

for(k=1 to n)

k=1

is near(1) ≠ 0, yes

1 ≠ 0

& cost(1,2) > cost(1,3)

4 > 9, NO

k=2

is near(2) ≠ 0, NO

k=3

is near(3) ≠ 0, NO

k=4

is near(4) ≠ 0, NO

k=5

is near(5) ≠ 0 4≠0,

yes

& is cost(5,4) > cost(5,3) 4>3, yes

than near(5)=3

2	0	0	0	3
1	2	3	4	5

i=3

for(j=1 to 5)

j=1

is near(1) ≠ 0 2 ≠ 0

cost(1,2) = 4

j=2

is near(2) ≠ 0, NO

j=3
Is near(3)≠0, no near(3)=0

j=4
Is near(4)≠0, no near(4)=0

j=5
Is near(5)≠0 near(5)=2→
3≠0; yes
and cost(5,3)=3

choosing the min cost from the above obtaining costs which is 3 & corresponding

j=5
Min cost = 4 + cost(5,3)
= 4+3=9

r[3,1]=5
r[3,2]=3
Near(3)=0 → near(5)=0

for (k=1 to 5)

k=1
is near(1)≠0 yes &
cost(1,2) > cost(1,5)
4 > ∞, NO

k=2
Is near(2)≠0 no

k=3
Is near(3)≠0 no

r[3,1]=5
r[3,2]=3

3	0	0	0	0
1	2	3	4	5

k=4
Is near(4)≠0 no

k=5
Is near(5)≠0 no

i=4

for j=1 to 5

j=1
Is near(1)≠0
2≠0, yes
cost(1,2)=4

j=2
Is near(2)≠0, NO

j=3
Is near(3)≠0, NO
near(3)=0

j=4
Is near(4)≠0, no
near(4)=0

j=5
Is near(5)≠0, NO
near(5)=0

choosing min cost from the above it is only '4' and correspond
j=1

Min cost = 7 + cost(1,2)
= 7 + 4 = 11

T(4,1) = 1
T(4,2) = 2

Near(1) = 0 & Near(i) = 0

for(k = 1 to 5)

k = 1
Is near(1) ≠ 0, NO

k = 2
Is near(2) ≠ 0, NO

k = 3
Is near(3) ≠ 0, NO

k = 4
Is near(4) ≠ 0, NO

k = 5
Is near(5) ≠ 0, NO

End.

0	0	0	0	0
1	2	3	4	5

T(4,1) = 1
T(4,2) = 2

→ The single source shortest path problem:

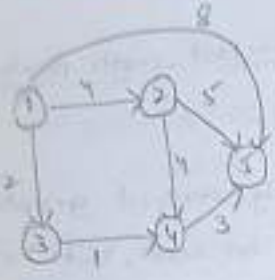
DIJKSTRA'S ALGORITHM:

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. It takes a labeled graph & a pair of vertices p and q, & finds the shortest path between them (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

It does not work for negative edges at all. The figure lists the shortest paths from vertex 1 to 5 vertex weighted digraph.



Graph

1	①
2	① → ②
3	① → ③ → ④
4	① → ②
6	① → ③ → ④ → ⑤

Shortest path

Algorithm shortest-paths(v, cost, dist, n)

// dist[j], 1≤j≤n, is set to the length of the shortest path

// from vertex v to vertex j in the digraph G with n vertices.

// dist[v] is set to zero. G is represented by its

// cost adjacency matrix cost[1:n, 1:n]

```
{
  for i:=1 to n do
  {
    s[i] := false;        // Initialize S.
    dist[i] := cost[v,i];
  }
  s[v] := true; dist[v] := 0.0;   // put v in S.
  for num := 2 to n-1 do
  {
    Determine n-1 paths from v.
    choose u from among those vertices not
        in S such that dist[u] is min;
    s[u] := true;                 // put u in S.
```



```
    for(each w adjacent to u with s[w] = false) do
      if(dist[w] > (dist[u] + cost[u,w])) then
                                  // update distance
        dist[w] := dist[u] + cost[u,w];
  }
}
```

Running time:

Depends on implementation of data structures for dist.

- Build a structure with n elements — A
- at most m = |E| times decrease the value of an item — mB
- 'n' times select the smallest value — nC
- for array A = O(1); B = O(1); C = O(n) which gives $O(n^2)$ total
- for heap A = O(n); B = O(log n); C = O(log n) which gives O(n + m log n) total

UNIT - 4

Branch and Bound

- General Method

It is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.
2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

It refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node.

It is the generalization of both graph search strategies, BFS and D-search.

- A BFS like state space search is called as FIFO search as the list of live nodes in a first in first out list (or queue).

• A D search like state space search is called as LIFO search as the list of live nodes in a last in first out (or stack).

Def 1: live node is a node that has been generated but whose children have not yet been generated.

Def 2: E-node is live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Def 3: Dead node is generated node that is not be expanded or explored any further. All children of a dead node have already been expanded.

Def 4: Branch-and-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Def 5: The adjective "heuristic" means "related to improving problem solving performance". As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving Person Problem". But imperfect methods are not necessarily heuristic

or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy trick simplification or any other kind of device which drastically limits search for solutions, they do not guarantee any solution at all. A useful heuristic offers solution which are good enough most of the time.

Least Cost (LC) search:

→ Applications

→ Travelling Sales Person problem

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. this can be solved by branch & bound technique using efficient bounding function. The time complexity of traveling sale Person problem using LC Branch & bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than Previous method.

We start at a particular node & visit all nodes exactly once & come back to initial node with minimum cost.

Let $G=(V,E)$ is a connected graph. let $c(i,j)$ be the cost of edge $\langle i,j\rangle$ $c_{ij}=\infty$ if $\langle i,j\rangle \notin E$ and let $|V|=n$, the number of vertices, every tour starts at vertex 1 and ends at the same vertex. so the solution space is given by $S=\{1,\pi,1\}$ π is a permutation of $(2,3,\cdots,n)$ and $|S|=(n-1)!$. The size of S can be reduced by restricting S so that $(1,i_1,i_2,\cdots,i_{n-1},1)\in S$ iff $\langle i_j,i_{j+1}\rangle \in E$, $0\le j\le n-1$ and $i_0=i_n=1$

Procedure for solving travelling sale person

Problem:

1. Reduce the given cost matrix. A matrix is reduced if every row & column is reduced. A row(column) is said to be reduced if it contain at least one zero and all remaining entries are non-negative. This can be done as follows:

(a) Row reduction: Take the minimum element from first row, subtract it from all elements of first row, next take min. element from the second row & subtract it from second row. similarly apply the same procedure for all rows.

(b) find the sum of elements, which were substracted from rows.

(c) Apply column reductions for the matrix obtained after row reduction.

Column reduction: Take the min element from the first column, subtract it from all elements of first column, next take min element from the second column & subtract it from second column. similarly apply the same procedure for all columns.

(d) find the sum of elements, which were subtracted from columns.

(e) obtain the cumulative sum of row wise reduction & column wise reduction.

Cumulative reduced sum: Row wise reduction sum + Column wise reduction sum.

Associate the cummulative reduced sum to the starting state as lower bound & ∞ as upper bound.

2. calculate the reduced cost matrix for every node R. let A is the reduced cost matrix for node R. let S be a child of R such that the tree edge (R,S) corresponds to including

edge <i,j> in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

(a) change all entries in row i & column j of A to ∞.

(b) set A[j,1] to ∞.

(c) Reduce all rows & columns in the resulting matrix except for rows & column containing only ∞. Let r is the total amount subtracted to reduce the matrix.

(d) Find c(S) = c(R) + A[i,j] + r

where 'r' is the total amount subtracted to reduce the matrix, c(R) indicates the lower bound of the T^{th} node in (i,j) Path and c(S) is called cost function

3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch & bound solution for the traveling Sales person problem whose cost matrix is as follows:

The cost matrix is

$$\begin{pmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{pmatrix}$$

Step 1: Find the reduced cost matrix.

Apply row reduction method:

Deduct 10 (which is min) from all values in the 1st row.

Deduct 2 (which is min) from all values in 2nd row.

Deduct 2 (which is min) from all values in 3rd row.

Deduct 3 (which is min) from all values in 4th row.

Deduct 4 (which is min) from all values in 5th row.

The resulting row wise reduced cost matrix

$$\begin{pmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 0 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{pmatrix}$$

Row wise reduction sum = 10+2+2+3+4 = 21

Now apply column reduction for the above matrix:

Deduct 1 (which is min) from all values in the 1st column.

Deduct 3 (which is min) from all values in the 3rd column.

The resulting column wise reduced

Cost matrix (A) = $\begin{pmatrix} \infty & 10 & 19 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{pmatrix}$

Column wise reduction sum =

$1+0+3+0+0 = 4$.

Cumulative reduced sum = row wise reduction + column wise reduction sum

$= 21+4 = 25$.

This is the cost of a root i.e node 1, because this is the initially reduced Cost matrix.

The lower bound for node 1 is 25 and upper bound is ∞.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. so, consider to explore the paths (1,2), (1,3), (1,4) and (1,5).

The tree organization upto this point is as follows:

variable 'i' indicates the next node to visit.

step 2: Consider the path (1,2):

change all entries of row 1 & column 2 of A to ∞ and also set A(2,1) to ∞.

$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{pmatrix}$

Apply row and column reduction for the row & columns whose rows & columns are not completely ∞.

Then the resultant matrix is $\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{pmatrix}$

Row reduction sum = $0+0+0+0 = 0$

Column reduction sum = $0+0+0+0 = 0$

Cumulative reduction (r) = $0+0 = 0$

Therefore, as $c(S) = c(R) + A(1,2) + r$

$c(S) = 25+10+0 = 35$

Consider the path (1,3):

Change all entries of row 1 & column 3 of A to ∞ & also set A(3,1) to ∞.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{pmatrix}$$

Apply row & column reduction for the rows & columns whose rows & columns are not completely ∞.

Then the resultant matrix is

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{pmatrix}$$

Row reduction sum = 0

column reduction sum = 11

cumulative reduction (r) = 0 + 11 = 11,

therefore, as $c(s) = c(R) + A(1,3) + r$

$c(s) = 25 + 17 + 11$

$= 53$

Consider the path (1,4):

Change all entries of row 1 & column 4 of A to ∞ & also set A(4,1) to ∞.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{pmatrix}$$

Apply row & column reduction for the rows & columns whose rows & columns are not completely ∞.

Then the resultant matrix is

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{pmatrix}$$

Row reduction sum = 0

Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

therefore, as $c(s) = c(R) + A(1,4) + r$

$c(s) = 25 + 0 + 0$

$= 25$

Consider the path (1,5):

change all entries of row 1 & Column 5 of A to ∞ and also set A(5,1) to ∞.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{pmatrix}$$

Apply row & column reduction for the rows & columns whose rows & columns are not completely ∞.

then the resultant matrix is

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{pmatrix}$$

Row reduction sum = 5

Column reduction sum = 0

cumulative reduction (r) = 5 + 0 = 0

Therefore, as $c(S) = c(R) + A(1,2) + r$

$$c(S) = 25 + 1 + 5$$
$$= 31$$

the tree organization up to this point is as follows:

the cost of the paths between (1,2) = 35, (1,3) = 53, (1,4) = 25 and (1,5) = 31.

The cost of the path between (1,4) is min. Hence the matrix obtained for path (1,4) is considered as reduced cost matrix.

$$A = \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{pmatrix}$$

The new possible paths are (4,2), (4,3) & (4,5)

Consider the path (4,2):

change all entries of row 4 & Column 2 of A to ∞ & also set A(2,1) to ∞.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{pmatrix}$$

Apply row and column reduction for the rows & columns whose rows & columns are not completely ∞.

Then the resultant matrix is

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{pmatrix}$$

Row reduction sum = 0

Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A[4,3] + r$

$c(S) = 25 + 3 + 0$

$= 28.$

Consider the path (4,3):

Change all entries of row 4 & column 3 of A to ∞ and also set A[3,1] to ∞.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{pmatrix}$$

Apply row & column reduction for the rows & columns whose rows & columns are not completely ∞.

Then the resultant matrix is

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{pmatrix}$$

Row reduction sum = 2

column reduction sum = 11

Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $c(S) = c(R) + A[4,3] + r$

$c(S) = 25 + 12 + 13 = 50$

Consider the path (4,5)

change all entries of row 4 & column 5 of A to ∞ and also set A[5,1] to ∞.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{pmatrix}$$

Apply row & column reduction for the rows & columns whose rows & columns are not completely ∞.

Then the resultant matrix is

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ \infty & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{pmatrix}$$

Row reduction sum = 11

Column reduction sum = 0

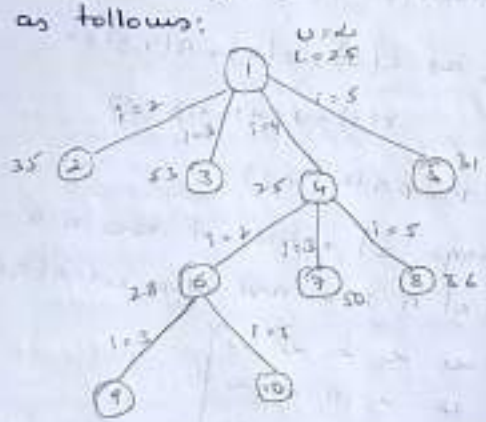
cumulative reduction (r) = 11 + 0 = 11

Therefore, as $c(s) \geq c(R) + A(4,5) + r$

$c(s) = 25 + 0 + 11$

$= 36$

The tree organization up to this point is as follows:



The cost of the paths between (4,2) = 28, (4,3) = 50 and (4,5) = 36. The cost of the path between (4,2) is min. Hence the matrix obtained for path (4,2) is considered as reduced cost matrix

$$A = \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{pmatrix}$$

The new possible paths are (3,3) and (2,5).

consider the path (3,3):

change all entries of row 3 and column 3 of A to ∞ and also set A(3,1) to ∞.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{pmatrix}$$

Apply row and column reduction for the rows and columns whose rows & columns are not completely ∞.

Then the resultant matrix is

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{pmatrix}$$

Row reduction sum = 2

column reduction sum = 11

cumulative reduction (r) = 2 + 11 = 13

Therefore, as $c(s) = c(R) + A(3,5) + r$

$c(s) = 28 + 11 + 13$

$= 52$

Consider the path (3,5):

change all entries of row 3 & column 5 of A to ∞ and also set A(5,1) to ∞.

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{pmatrix}$$

Apply row & column reduction for the rows & columns whose rows & columns are not completely ∞.

Then the resultant matrix is

$$\begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{pmatrix}$$

Row reduction sum = 0

column reduction sum = 0

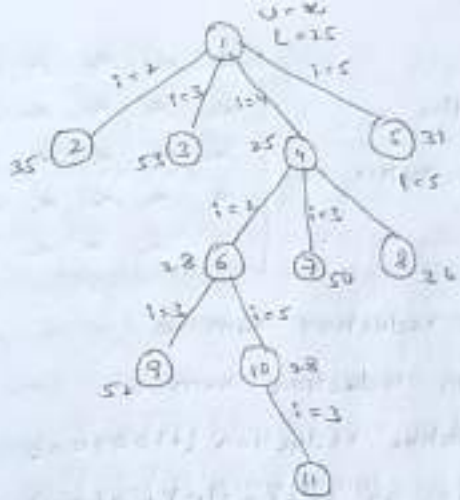
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A(3,5) + r$

$c(S) = 28 + 0 + 0$

$= 28$

The tree organization up to this point is as follows:



The cost of the paths between (3,5) = 52 and (2,5) = 28. The cost of the path between (3,5) is minimum. Hence the matrix obtained for path (3,5) is considered as reduced cost matrix

$$A = \begin{pmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{pmatrix}$$

the new possible path is (5,3).

Consider the path (5,3):

change all entries of row 5 & column 3 of A to ∞ & also set A(3,1) to ∞. Apply row & column reduction for the rows & columns whose rows & columns are not completely ∞.

Then the resultant matrix is

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Row reduction sum = 0

Column reduction sum = 0

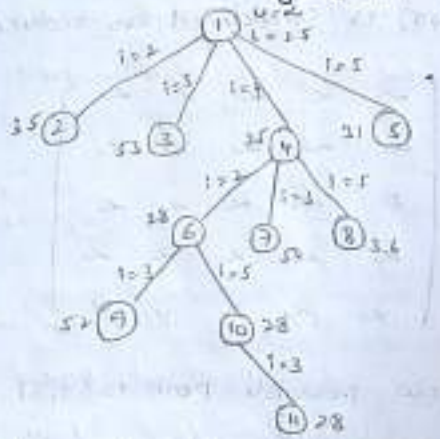
Cumulative reduction (r) = 0+0 = 0

Therefore, as $c(s) = c(R) + A(s,j) + r$

$c(s) = 28 + 0 + 0$

$= 28$

The overall tree organization is as follows



The path of traveling sale person problem is :

$1 \to 4 \to 2 \to 5 \to 3 \to 1$

The min cost of the path is : $10+6+2+3+3$

$= 28.$

→ 0/1 knapsack problem

Consider the instance : $M=15, n=4$

$(P_1, P_2, P_3, P_4) = (10, 10, 12, 18)$ & $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$

0/1 knapsack problem can be solved by using branch & bound technique. In this problem we will calculate lower bound & upper bound for each node.

place first item in knapsack. Remaining weight of knapsack is $15-2=13$. place next item w_2 in knapsack & the remaining weight of knapsack is $13-4=9$. place next item w_3 in knapsack then the remaining weight of knapsack is $9-6=3$. No fractions are allowed in calculation of upper bound so w_4 cannot be placed in knapsack.

Profit $= P_1 + P_2 + P_3 = 10+10+12$

So, upper bound $= 32$

To calculate lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

lower bound : $10+10+12+\left(\frac{3}{9} \times 18\right)$

$= 32 + 6$

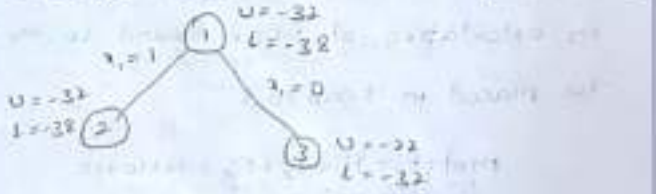
$= 38$

knapsack problem is max. problem but branch & bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore, upperbound (U) = -32

lower bound (L) = -38

we choose the path, which has minimum difference of upper bound & lower bound. If the difference is equal then we choose the path by comparing upper bounds & we discard node with maximum upper bound



Now we will calculate upper and lower bound for nodes 2,3

for node 2, $x_1=1$, means we should place first item in the knapsack.

U = 10+10+12 = 32, make it as -32

L = 10+10+12+$\frac{3}{9}$ x 18 = 32+6 = 38

make it as -38.

for node 3, $x_1=0$, means we should not place first item in the knapsack.

U = 10+12 = 22, make it as -22

L = 10+12+$\frac{5}{9}$ x 18 = 10+12+10 = 32,

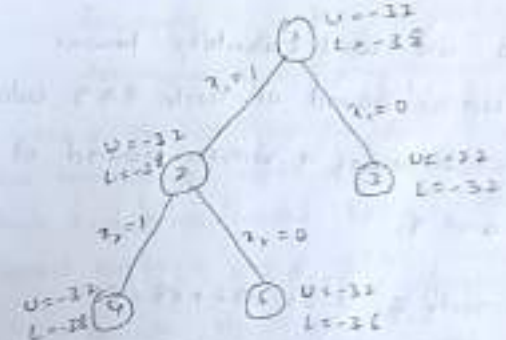
make it as -32

Next, we will calculate difference of upper bound and lower bound for nodes 2,3

for node, 2, U-L = -32+38 = 6

for node 3, U-L = -22+32 = 10

choose node 2, since it has minimum difference value of 6.

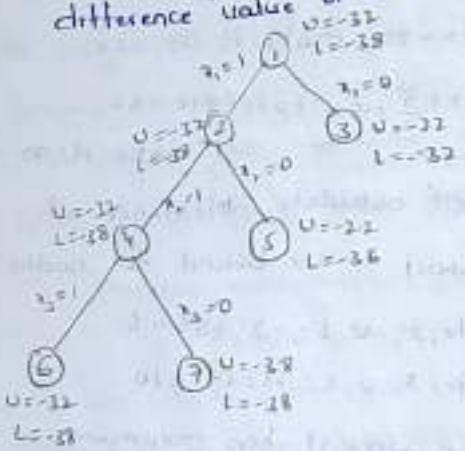


Now we will calculate lower bound & upper bound of node 4 and 5 calculate difference of lower and upper bound of nodes 4 and 5.

for node 4, U-L = -32+38 = 6

for node 5, U-L = -32+36 = 14

Choose node 4, since it has minimum difference value of 6.

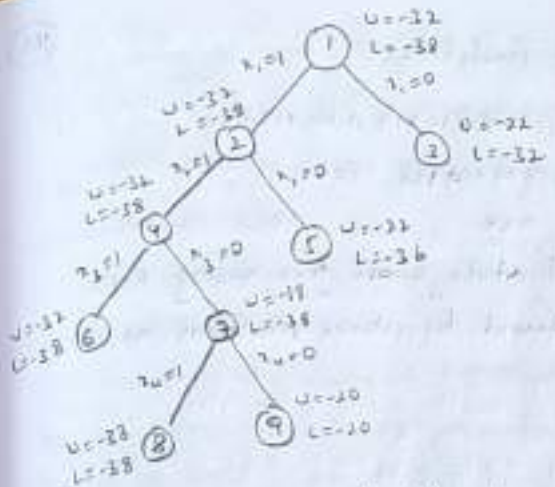


Now we will calculate lower bound & upper bound of node 8 & 9. Calculate difference of lower & upper bound of nodes 8 and 9.

for node 6, U-L = -32+38 = 6

for node 7, U-L = -38+38 = 0

Choose node 7, since it is minimum difference value of 0.



Now we will calculate lower bound & upper bound of node 4 & 5. Calculate difference of lower & upper bound of nodes 4 and 5.

for node 8, U-L = -38+38 = 0

for node 9, U-L = -20+20 = 0

Here the difference is same, so compare upper bounds of nodes 8 & 9. Discard the node, which has max upper bound. choose node 8, discard node 9 since, it has max upper bound.

Consider the path from

$1 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 8$

$x_1 = 1$

$x_2 = 1$

$x_3 = 0$

$x_4 = 1$

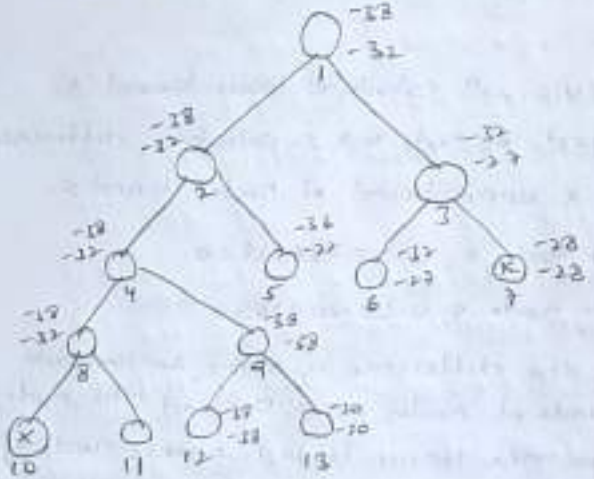
The solution for 0/1 knapsack problem is

$(x_1, x_2, x_3, x_4) = (1,1,0,1)$

Maximum profit is:

$$EP_{[7]} = 10\times1 + 10\times1 + 12\times0 + 18\times1$$
$$= 10 + 10 + 18$$
$$= 38.$$

Portion of state space tree using FIFO Branch & Bound for above problem as follows:



upper number = 2

lower number = 4

LC-Branch & bound tree.

→ LC Branch and Bound solution

An LC Branch-and Bound search of the tree (#3) will begin with upper = ∞ & node 1 as the first E-node.

When node 1 is expanded, nodes 2, 3, 4, 5 are generated in that order.

As in the case of FIFO&upper is updated to 14 when node 3 is generated as the nodes 4 and 5 are killed as c(4) > upper and c(5) > upper.

Node 2 is the next E-node as c(2) = 0 and c(3) = 5. Nodes 6, 7, 8 are generated & upper is updated to 9 when node 6 is generated. So, node 7 is killed as c(7) = 10 > upper. Node 8 is infeasible & so killed. The only live nodes now are nodes 3 and 6.

Node 6 is the next E-node as c(6) = 0 < c(3) Both its children are infeasible.

Node 3 becomes the next E-node when node 9 is generated, upper is updated to 8 as u(9) = 8. So, node 10 with c(10) = 11 is killed on generation.

Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node.

The path = $1 \xrightarrow{2} 3 \xrightarrow{3} 9$ = 5+3 = 8

→ FIFO Branch and Bound:

A FIFO branch & bound algorithm for the job sequencing problem can begin with upper=∞ as an upper bound on the cost of a minimum-cost answer node.

Starting with node 1 as the E-node & using the variable tuple size formulation, nodes 2, 3, 4 and 5 are generated. Then u(2)=19, u(3)=14, u(4)=18, u(5)=21.

The variable upper is updated to 14 when node 3 is generated. Since c(4) & c(5) are greater than upper, nodes 4 & 5 get killed. only nodes 2 & 3 remain alive.

Node 2 becomes the next E-node. Its children, nodes 6, 7, 8 are generated.

Then u(6)=9 & so upper is updated to 9. The cost c(7)=10 > upper & node 7 gets killed. Node 8 is infeasible & so it is killed.

Next, node 3 becomes the E-node. Nodes 9 & 10 are now generated. Then u(9)=8 & so upper becomes 8. The cost c(10)=11 > upper, & thus node 10 is killed.

The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The min-cost answer node is node 9. It has cost of 8.

When implementing a FIFO branch-&-bound algorithm, it is not economical to kill live nodes with c(x) > upper each time upper is updated. This is so because live nodes are in the queue in the order in which they are generated. Hence, nodes with c(x) > upper are distributed in some random way in the queue. Instead, live nodes with c(x) > upper can be killed when they are about to become E-nodes.

The FIFO-based branch-&-bound algorithm with an appropriate c(·) and u(·) is called FIFOBB.

NP Hard and NP- Complete Problems

Basic Concepts

Deterministic & non-deterministic algorithm

Deterministic

The algorithm in which every operation is uniquely defined is called deterministic algorithm.

Non-Deterministic

The algorithm in which the operations are not uniquely defined but are limited to specific set of possibilities for every operation, such algorithm is called non-deterministic algorithm.

The non-deterministic algorithm use the following functions:

1. choice: Arbitarily chooses one of the element from given set.
2. Failure: Indicates an unsuccessful completion.
3. success: Indicates a successful Completion.

A non-deterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal. whenever, there is a set of choices that leads to a successful completion, then one such set of choices is selected & the algorithm terminates successfully.

In case the successful completion is not possible, then the complexity is O(1). In case of successful signal completion then the time required is the minimum number of steps needed to reach a successful completion of O(n) where n is the number of inputs.

The problems that are solved in polynomial time are called tractable problems and the problems that require super polynomial time are called non-tractable problems. All deterministic polynomial time algorithms are tractable and the non-deterministic polynomials are intractable.

- Algorithm Nsort (A,n)

//sort n positive integers

```
{
    for i:=1 to n do B[i]:=0; // Initialize
                                 B[]
    for i:=1 to n do
    {
        j:= choice(1,n);
        if B[j]≠0 then Failure();
        B[j]:= A[i];
    }
    for i:=1 to n-1 do // verify order
        if B[i] > B[i+1] then Failure();
    write (B[1:n]);
    Success();
}
```

Non-deterministic Sorting

- Algorithm DKP(p,w,n,m,r,x)

```
{
    w:=0; p:=0;
    for i:=1 to n do
    {
        x[i]:= choice(0,1);
        w:= w+x[i] * w[i];
        p:= p+x[i] * p[i];
    }
    if ((w>m) or (p<r)) then Failure();
    else Success();
}
```


Non-deterministic knapsack algorithm

Satisfiability problem:

It is a boolean formula that can be constructed using the following literals & operations.

1. A literal is either a variable or its negation of the variable.
2. The literals are connected with operators $\vee, \wedge, \rightarrow, \Rightarrow, \Leftrightarrow$
3. parenthesis.

This problem is to determine whether a Boolean formula is true for some assignment of truth values to the variables. In general, formulas are expressed in Conjunctive Normal form (CNF).

A Boolean formula is in Conjunctive normal form if it is represented by

$(x_1 \vee x_2 \vee x_3') \wedge (x_1 \vee x_2' \vee x_3)$

A Boolean formula is in 3CNF if each clause has exactly 3 distinct literals.

Example:

The non-deterministic algorithm that terminates successfully iff a given formula $E(x_1, x_2, x_3)$ is satisfiable.

```
— Algorithm Eval(E,n)
  // Determine whether the propositional
                          formula E is
  // satisfiable. The variables are
                          x1, x2, ... xn.
  {
    for i:=1 to n do // choose a truth
                          value assignment
      xi := choice(false, true);
    if E(x1, ..., xn) then Success();
    else Failure();
  }
```

Nondeterministic Satisfiability

Reducability:

A problem Q_1 can be reduced to Q_2 if any instance of Q_1 can be easily rephrased as an instance of Q_2. If the solution to the problem Q_2 provides a solution to the problem Q_1, then these are said to be reducable problems.

Let L_1 and L_2 are two problems. L_1 is reduced to L_2 iff there is a way to solve L_1 by a deterministic polynomial time and is denoted by $L_1 \propto L_2$.

If we have a polynomial time algorithm for L_2 then we can solve L_1 in polynomial time. Two problems $L_1 \propto L_2$ are said to be polynomially equivalent iff $L_1 \propto L_2$ and $L_2 \propto L_1$.

Example:

Let P_1 be the problem of Selection & P_2 be the problem of sorting.

Let the input have n numbers. If the numbers are sorted in array A[] the i^{th} smallest element of the input can be obtained as A[i]. Thus P_1 reduces to P_2 in $O(1)$ time.

Decision problem:

Any problem for which the answer is either yes or no is called decision problem. the algorithm for decision proble is called decision algorithm.

Example:

Max clique problem, sum of subsets problem.

optimization problem:

Any problem that involves the identification of an optimal value (max or min) is called optimization problem.

Ex: knapsack problem, travelling salesper

In decision problem, the output statemen is implicit & no explicit statements are permit

The olp from a decision problem is uniquely defined by the input parameters and algorithm specification.

Many optimization problems can be reduced by decision problems with the property that a decision problem can be solve in polynomial time if corresponding optimiz problem can be solved in polynomial time.

If the decision problem cannot be solved in polynomial time then the optimization problem cannot be solved in polynomial time.

class P's

The class of decision problems that are solvable in $O(p(n))$ time, where $p(n)$ is a polynomial of problem's input size n

Examples:

- Searching
- element uniqueness
- graph connectivity
- graph acyclicity
- primality testing

class NP

NP (nondeterministic polynomial): class of decision problems whose proposed solutions can be verified in polynomial time = solvable by a nondeterministic polynomial algorithm.

A nondeterministic polynomial algorithm is an abstract two-stage procedure that:

- generates a random string purported to solve the problem.
- checks whether this solution is correct in polynomial time.

By definition, it solves the problem if it is capable of generating & verifying a solution on one of its tries.

Example: CNF satisfiability.

Problem: Is a boolean expression in its conjunctive normal form (CNF) satisfiable? i.e are there values of its variables that makes it true? This problem is in NP.

Non deterministic algorithm:

- Guess truth assignment
- Substitute the values into the CNF formula to see if it evaluates to true.

What problems are in NP?

- Hamiltonian circuit existence
- partition problem: Is it possible to partition a set of n integers into two disjoint subsets with the same sum?
- Decision versions of TSP, knapsack problem, graph coloring & many other combinatorial optimization problems. (Few exceptions include: MST, shortest paths)
- All the problems in P can also be solved in this manner (but no guessing is necessary), so we have:

$$P \subseteq NP$$

- Big question: $P = NP$?

Commonly believed relationship between P and NP.

- NP Hard and NP Complete

Polynomial Time algorithms

Problems whose solutions times are bounded by polynomials of small degree are called polynomial time algorithms.

Example: Linear search, quick sort, all pairs shortest path etc.

Non-polynomial time algorithms

Problems whose solutions times are bounded by non-polynomials are called non-polynomial time algorithms.

Examples: Travelling salesman problem, 0/1 knapsack problem etc.

It is impossible to develop the algorithms whose time complexity is Polynomial for non-polynomial time problems, because the computing times of non-polynomial are greater than polynom

A problem that can be solved in polynomial time in one model can also be solved in Polynomial time.

NP-Hard and NP-Complete Problems

Let P denote the set of all decision problems solvable by deterministic algorithm in polynomial time. NP denotes set of decision problems solvable by nondeterministic algorithms in polynomial time. Since, deterministic algorithms are a special case of nondeterministic algorithms, $P \subseteq NP$. The nondeterministic polynomial time problems can be classified into two classes. They are

1. NP Hard and
2. NP Complete

NP-Hard:

A problem L is NP-Hard iff satisfiability reduces to L i.e, any nondeterministic polynomial time problem is satisfiable & reducable then the Problem is said to be NP-Hard.

Example: Halting Problem, Flow shop scheduling problem.

NP-Complete:

A problem L is NP-Complete iff L is NP-Hard and L belongs to NP(nondeterministic polynomial).

A problem that is NP-Complete has the property that it can be solved in polynomial time iff all other NP-Complete problems can also be solved in polynomial time (NP=P).

If an NP-hard problem can be solved in polynomial time, then all NP-Complete problems can be solved in Polynomial time. All NP-Complete problems are NP-hard, but some NP-hard problems are not known to be NP-Complete.

Normally the decision problems are NP-Complete but the optimization problems are NP-Hard.

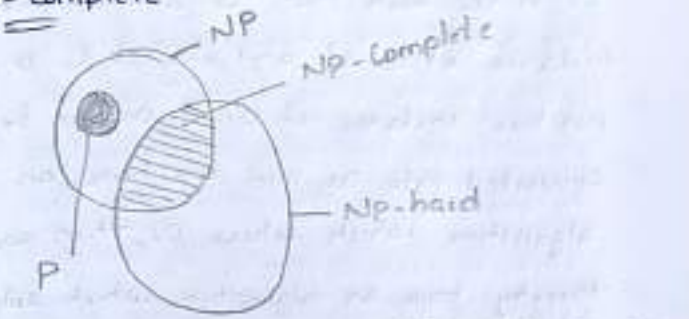
However if problem L1 is a decision problem and L2 is an optimization problem, then it is possible that L1 ∝ L2.

Example:

Knapsack decision problem can be reduced to knapsack optimization problem.

There are some NP-hard problems that are not NP-Complete.

Relationship between P, NP, NP-hard, NP-Complete



Commonly believed relationship among P, NP, NP-Complete and NP-hard problems.

Let P, NP, NP-hard, NP-Complete are the sets of all possible decision problems that are solvable in polynomial time by using deterministic algorithms, non-deterministic algorithms, NP-Hard and NP-Complete respectively. Then the relationship between P, NP, NP-hard, NP-Complete can be expressed using Venn diagram as:

Problem Conversion

A decision problem D_1 can be converted into a decision problem D_2 if there is an algorithm which takes as input an arbitary instance I_1 of D_1 and delivers as output an instance I_2 of D_2 such that I_2 is a positive instance of D_2 if and only if I_1 is a positive instance of D_1. If D_1 can be converted into D_2, and we have an algorithm which solves D_2, then we thereby have an algorithm which solves D_1. To solve an instance I of D_1, we first use the conversion algorithm to generate an instance of I_0 of D_2, and then use the algorithm for solving D_2 to determine whether or not I_0 is a positive instance of D_2. If it is, then we knows that I is a positive instance of D_1, and if it is not, then we know that I is a negative instance of D_1. Either way, we have solved D_1 for that instance.

Moreover, in this case, we can say that the computational complexity of D_1 is at most the sum of the computational complexities of D_2 and the conversion algorithm. If the conversion algorithm has polynomial complexity, we say that D_1 is at most polynomially harder than D_2. It means that the amount of computational work we have to do to solve D_1, over and above whatever is required to solve D_2, is polynomial in the size of the problem instance. In such a case the conversion algorithm provides us with a feasible way of solving D_1, given that we know how to solve D_2.

Given a problem X, prove it is in NP-Complete.

1. Prove X is in NP.
2. Select problem Y that is known to be in NP-Complete.
3. Define a polynomial time reduction from Y to X.
4. Prove that given an instance of Y, Y has a solution iff X has a solution.

→ Cook's theorem:

It implies that any NP problem is at most polynomially harder than SAT. This means that if we find a way of solving SAT in polynomial time, we will then be in a position to solve any NP problem in polynomial time. This would have huge practical repercussions, since many frequently encountered problems which are so far believed to be intractable are NP. This special property of SAT is called NP-Completeness. A decision problem is NP-Complete if it has the property that any NP problem can be converted into it in polynomial time. SAT was the first NP-Complete problem to be recognized as such (the theory of NP-completeness having come into existence with the proof of Cook's Theorem), but it is by no means the only one. There are now literally thousands of problems

cropping up in many different areas of computing, which have been proved to be NP-Complete.

In order to prove that an NP problem is NP-Complete, all that is needed is to show that SAT can be converted into it in polynomial time. The reason for this is that the sequential composition of two polynomial-time algorithms is itself a polynomial-time algorithm, since the sum of two polynomials is itself a polynomial.

Suppose SAT can be converted to problem D in polynomial time. Now take any NP problem D_0. We know we can convert it into SAT in polynomial time, & we know we can convert SAT into D in polynomial time. The result of these two conversions is a polynomial-time conversion of D_0 into D. Since D_0 was an arbitrary NP problem, it follows that D is NP-Complete.

Boolean satisfiability problem (SAT/B-SAT)

In logic and computer science, the Boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated SATISFIABILITY, SAT or B-SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula.

In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE. If this is the case, the formula is called satisfiable.

On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments & the formula is unsatisfiable.

for example, the formula "a AND NOT b" is satisfiable because one can find the values a = TRUE and b = FALSE, which make (a AND NOT b) = TRUE. In contrast, "a AND NOT a" is unsatisfiable.

SAT is the first problem that was proven to be NP-Complete.