DAA

Algorithm

An algorithm can be defined as a finite set of steps which are to be carried out to solve a particular problem.

characteristic / criteria of an Algorithm

- Input
- output
- Effectiveness
- Finiteness
- Definiteness.

Procedure for writing Algo

Algorithm Name of the Algo (parameters)

// Define parameters.

Input:-

output:-

BEGIN or {

1.

2.

3.

END or }

- The frequency count method in an algorithm depicts the relation between the time requirement of a program based on the number of inputs.

$$T(P) \propto n$$

$$T(n) \propto n$$

where, $T(n)$ is the time required for n input program.

$$T(n) = Kn$$

↓

proportionality constant.

Asymptotic Notations

- Asymptotic notations are mathematical tools used to represent the time complexity of algorithms for asymptotic analysis.

- There are mainly 3 asymptotic notations:-
  1) Big-O Notation
  2) Big-omega "
  3) Theta " .

Apart from this, there are ~~little~~ little-o, little omega notation.

Here, Big-O Notation } depicts the maximum bound
Little-O "

Big-omega notation } minimum
Little-omega "

θ - Theta } depicts the bounded value.

1) Big-O

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

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

where c is the constant,

$n \geq n_0$

and n_0 is the initial input.

ex:- $f(n) = 2n + 5$ Given

since, $f(n) \leq c(g(n))$

$2n+5 \leq 2n+n$

$2n+5 \leq 3n$

Here, $c = 3$.

and $n_0 = 5$. and $g(n) = n$.

Hence, $f(n) = O(n)$

- We generalize it as,

If $f(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots$

Then, $f(x) = O(x^n)$

Q. Given, i) $f_1(n) = O(g_1(n))$
ii) $f_2(n) = O(g_2(n))$

Find a) $f_1 f_2$ b) $f_2 + f_1$ c) $\frac{f_1}{f_2}$

Soln a) $f_1 f_2 = O(g_1(n) \times g_2(n))$

$f_1(n) = O(n)$
and $f_2(n) = O(n^2)$

a) $f_1 f_2 = O(n^3)$

b) $f_1 + f_2 = O(\max(n, n^2))$
$= O(n^2)$

c) $\frac{f_1}{f_2} = O(n) \times O\left(\frac{1}{n^2}\right) = O\left(\frac{1}{n}\right)$

Q. Find the order of eqn
$f(n) = n\log n^3 + n^2 \log\log n^2 + 5n^2 + 8n + 8$.

Soln $f(n) = n\log n^3 + n^2\log\log n^2 + 5n^2 + 8n + 8$

f_1 f_2 f_3 f_4 f_5

$f(n) = O(\max(f_1, f_2, f_3, f_4, f_5))$

$f_5 = O(1)$
$f_4 = O(n)$
$f_3 = O(n^2)$

$f_2 = n^2 \log\log n^2$
$f_1 = n \log n^3$

f_6 f_7

$f_6 = O(n)$
$f_7 = \log n^3$
$= 3\log n$
$f_7 = O(\log n)$

$f_1 = O(f_6 * f_7)$
$f_1 = O(n) \times O(\log n)$
$f_1 = O(n\log n)$

$$f_2 = \underbrace{n^2}_{f_8} . \underbrace{\log \log n^2}_{f_9}$$

$f_8 = O(n^2)$

$f_9 = \log(\log n^2)$

$f_9 = O(\log \log n)$

$f_2 = O(n^2) \times O(\log \log n)$

$= O(n^2 \log \log n)$

$f = O(\max(n \log n, n^2 \log \log n, n^2, n, 1))$

upon checking with values of n,

$f = O(n^2)$

* <u>ALGORITHM</u>

(1) <u>Algorithm sum N (a, n)</u>

// a is an array consisting of 'n' number of elements

Input:- 'n' no. of inputs

output:- sum of n numbers.

		s/e	f/e
BEGIN			
1.	sum = 0	1	1
2.	for (i=0; i<n; i++)	1	n+1
3.	{	0	-
4.	sum = sum + a[i];	1	n
5.	}	0	-
6.	printf sum	1	1
END			

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

$= 2n + 3$

$T_{sum}(n) = O(n)$

(2) Algorithm Factorial (n)

// n is the number

I/P: n is the number whose factorial is to be calculated

O/P: Factorial of n

	s/e	f/e
BEGIN or {		
1. fact = 1	1	1
2. for (i = n; i ≥ 1; i--)	1	n+1
3. {	0	-
4. fact = fact * i;	1	n
5. }	0	-
6. print fact;	1	1
7. END or }		

$$T_{fact}(n) = 1+n+1+n+1$$
$$= 3+2n$$
$$= 2n+3$$

$$T_{fact}(n) = O(n)$$

· RECURSION-USED ALGORITHMS

(3) Algorithm rsumN(a,n)

// a is the array, n is the input in the array

I/P: a is the array of number n

O/P: sum of n numbers.

	s/e $n \le 0$	s/e $n > 0$	f/e $n \le 0$	f/e $n > 0$
{				
1. sum = 0	1	1	1	1
2. if ($n \le 0$)	1	1	1	1
3. return sum	1		1	
4. else				
5. return (sum + a[n] + rsumN(a,n-1))		1		1 + rsum(n-1)
}				

$$T_{rsumN}(n) = \begin{cases} 2 & (n \le 0) \\ 2 + T_{rsumN}(n-1) & (n > 0) \end{cases}$$

- For each and every recursive program there must be a base value or terminate value otherwise program will be infinite.

$$\begin{aligned} T_{rsumN} &= 2 + T_{rsumN}(n-1) \\ &= 2+2+T_{rsumN}(n-2) \\ &= 4 + T_{rsumN}(n-2) \\ &= 2\times 2 + T_{rsumN}(n-2) \\ &= 2\times 3 + T_{rsumN}(n-3) \end{aligned}$$

At n step,

$$\begin{aligned} &= 2\times n + T_{rsumN}(n-n) \\ &= 2n + T_{rsumN}(0) \\ &= 2n+2 \\ &= 2(n+1) \\ &= O(n) \end{aligned}$$

④ Algorithm rfact(n)

		s/e n=0	s/e n>0	f/e n=0	f/e n>0
	{				
1.	if(n=0)	1	1	1	1
2.	return 1	1		1	
3.	else				
4.	return(rfact(n-1)*n)		1		1+rfact(n-1)

$$T_{rfact}(n) = \begin{cases} 2 & n=0 \\ 2+T_{rfact}(n-1) & o/w \end{cases}$$

RECURRENCE RELATION

There are following methods to solve recurrence relation:-

→ Iterative method
→ Recursive tree method
→ substitution method
→ master method
→ n^{α} method
→ changing variable method.

① RECURSIVE TREE METHOD

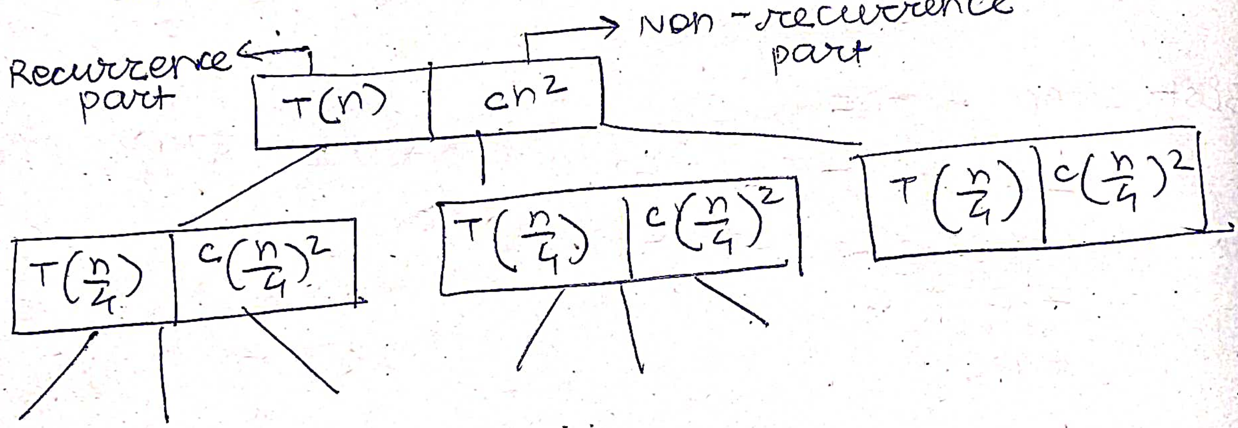
- It is a pictorial representation of iterative method using trees.

ex:-

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2$$

$\left(\frac{n}{4}\right)$ → no. of each subpart

3 ↓ denotes the no. of child



and so on

$$i^{th} \text{ level} = 3^i c\left(\frac{n}{4^i}\right)^2$$

$T(n) = O(n^2)$ after solving.

② ITERATIVE METHOD

- In this case, we keep upon repeating the terms and substituting.

ex:-

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n), & n > 1 \\ 2, & n = 1 \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$
$$= 2T\left(\frac{n}{2}\right) + cn$$
$$= 2\left[2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right] + cn$$
$$= 4T\left(\frac{n}{4}\right) + 2cn \quad \text{and so...on.}$$

ith step $\rightarrow$ $2^i T\left(\frac{n}{2^i}\right) + icn$

$$2^i = n$$
$$\Rightarrow i = \log_2 n$$

The equation becomes,

$$nT\left(\frac{n}{n}\right) + \log_2 n \; cn$$
$$= nT(1) + cn \log_2 n$$
$$= 2n + cn \log_2 n$$
$$= O(\max(n, n\log n))$$
$$= O(n\log n)$$

This is an example of iterative process / method of solving recurrence relation.

③ MASTER METHOD

when we have a recurrence relation in the form,

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

where, a is the no. of divisors (subparts)
b is the size of each subpart.

f(n) indicates the total time required to divide & ~~conquer~~ / combine each subproblem.

also, the values of a and b must satisfy $a \geq 1$ and $b > 1$.

- There are 3 cases in this method:-

case I

If $f(n) = O(n^{E-\epsilon})$
then $T(n) = \theta(n^E)$

case II

If $f(n) = O(n^E)$, then $T(n) = \theta(f(n)\log n)$

case III

If $f(n) = \Omega(n^{E+\epsilon})$
and $f(n) = \theta(n^{E+\epsilon})$
then $T(n) = \theta(f(n))$

ex

$T(n) = 4T\left(\frac{n}{2}\right) + n$

given, $a = 4$
$b = 2$
$f(n) = n$

$f(n) = n^{\log_b a}$
$= n^{\log_2 4}$
$= n^2$

calculated value > given value
so, $T(n) = \theta(n^2)$
where $E = 2$

Methods to design an algorithm

Following are the methods to design an algorithm:-

→ Divide and conquer
→ Greedy method
→ Dynamic programming
→ Backtracking.

Generalised Algorithm for Divide & conquer

Algorithm DANDC (P)

1. If small (P) then return S(P)
2. else
3. Divide the P into P1, P2, P3 ... PK K>1
   and find S(P1), S(P2), ... S(PK)
4. Return (Combine (S(P1)), S(P2), ... S(PK))

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ T(n_1) + T(n_2) + \dots + T(n_K) + f(n) & \text{o/w} \end{cases}$$

$$T(n) = \begin{cases} g(n) \\ aT\left(\frac{n}{b}\right) + f(n) \end{cases}$$

⑥ QUICKSORT

Algorithm Quicksort (A, p, r)

// A is the array, p is the lower bound,
r is the upper bound

{
1. If (p<r)
2. q = partition (A, p, r)
3. Quicksort (A, p, q-1)
4. Quicksort (A, q+1, r)

}

Algorithm Partition (A, p, r)

```
x = A[r]                              1    1
i = p-1                               1    1
for j = p to r-1                      1    r
{
    if (A[j] ≤ x)                     1    r-1
    {                                 1
        i = i+1                            r-1
        exchange A[i] ⇔ A[j]          1
    }
}
exchange A[i+1] ⇔ A[r]                1
```

Best case: $T(n) = O(n \log n)$
Worst case: $T(n) = O(n^2)$
Average case: $T(n) = O(n \log n)$

MERGE SORT

⑦ Algorithm Mergesort (A, p, r)
// A is an array where p and r are the lower and upper bound of the array respectively.

Input:- array A.
Output:- sort of element in A

```
if (p < r)
{
    q = (p+r)/2
    mergesort (A, p, q)
    Mergesort (A, q+1, r)
    Merge (A, p, q, r)
}
```

Algorithm Merge (A, p, q, r)

```
{
n1 = q-p+1
n2 = r-q
L1 [1 ... n1] and R [1 ... n2]
for i=1 to n1
  {
   L[i] = A[p+r-1]
  }
for j=1 to n2
   {
    R[j]= A [q+j]
   }
L [n1+1] = ∞
R [n2+1] = ∞
i = 1
j = 1
for K=p to r
     {  if (L[i] ≤ R[j])
             A[K] = L[i]
             i = i+1

        else A[K] = R[j]
             j = j+1

}
```

merge = $O(n)$

Dividing = $2T\left(\frac{n}{2}\right)$

$T(n) = O(n \log n)$

HEAP

⑧ Algorithm Build Heap (A)

BEGIN

1. heapsize = length [A]
2. for i = heapsize /2 to 1.
3. max heapify (A, i)

END

Algorithm max Heapify (A, i)

// i is the index where max heap property violates.

BEGIN

1. l = left (i)
2. R = right (i)
3. If $l \leq$ A.heapsize and A[l] > A[i]
4. largest = A[l]
   else largest = A[i]
5. If R ≤ A.heapsize and A[R] > largest
   largest = A[R]
6. If largest ≠ i
7. exchange A[i] with A [largest]
8. max heapify (A, largest)

END

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

$$\frac{n}{2}(\log n)$$

$$T(n) = O(n \log n)$$

⑨ Algorithm Heapsort (A)

// A is the array

Input :- A is an array

output :- sorted array

1. MAX BUILDHEAP (A)
2. p = heap length
3. for i = p to 2
4. exchange A[i] with A[1]
5. heapsize = heap - 1
6. maxheapify (A, 1)

$T(n) = O(n \log n)$

Following are the applications of heapsort :

- maintaining priority queue
- extract max

⑩ Algorithm Insert (A, x)

// x is the new element to be inserted

1. size = length (A)
2. size = size + 1
3. A[size] = x
4. while i > 1 and A (parent(i)) < A (i))
5. max heapify (A, parent(i))

⑪ Algorithm Increase key Heap (A, i, key)

1. if (key < A[i])
2. print " can't be exchanged "
3. A[i] = key
4. while (i > 1 and A (parent(i)) < A[i])

{ exchange A[i] ⟺ A [parent(i)]

5. i = parent(i)

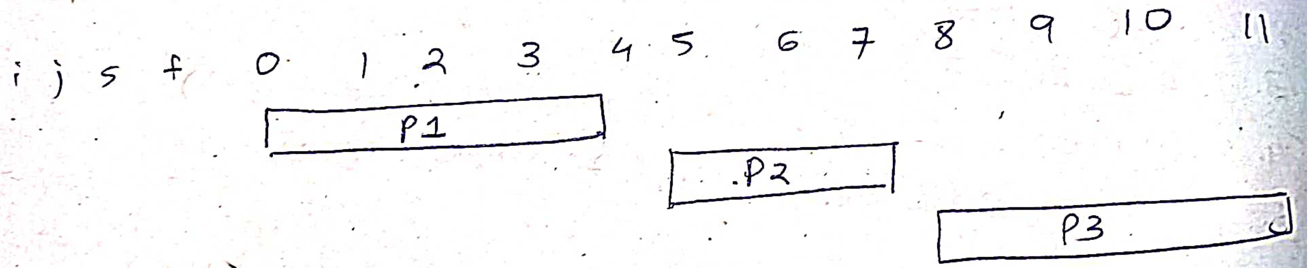
}

GREEDY ALGORITHM

a1	a2	ai	aj	an	] → process
s1	s2	si	sj	sn	→ start time
f1	f2	fi	fj	fn	→ finish time

Here, a_i and a_j are mutually exclusive

process	start time		finish time	
				4
P1	1		4	5
P2	5	3	7	6
P3	3	0	9	7
P4	6	5	10	9
P5	5	3	9	9
P6	8	5	11	10
P7	2	6	14	11
P8	12	8	16	12
P9	3	8	5	14
P10	0	2	6	16
P11	8	12	12	



and so-on.

(13) Algorithm Recursive ~~Active~~ Active selection (s, f, i, j)

// s is the start array if is the finish array

{

1. $m = i+1$
2. while $m \leq j$ and $s_m \leq f_i$
3. {
4. $m = m+1$
5. }
6. if $(i \neq j)$
7. return $(P_m \cup$ recursive active selection $(s, f, i, j))$

}

KNAPSACK

- Consider a bag or knapsack whose capacity is m and n number of objects. and select an object x_i whose weight is w_i is placed into the knapsack its price is P_i in such a manner that it will earn maximum profit $P_i w_i$.

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

subject to condition

$$\sum_{i=1}^{n} w_i x_i \leq m \quad \text{and} \quad 0 \leq x_i \leq 1$$

- If object can be expressed in terms of fraction then it is fractional knapsack.

- And if object cannot be fraction it is called 0-1 knapsack.

ex:-

Given $n = 3, m = 20$

If there are 3 objects whose prices and weights are:

$(P_1\ P_2\ P_3) = (25, 24, 15)$

$(w_1\ w_2\ w_3) = (18, 15, 10)$

Item	weight	price
I_1	18	25
I_2	15	24
I_3	10	15

capacity, $m = 20$

Possible combinations

	x_1	x_2	x_3	Profit $x_1P_1 + x_2P_2 + x_3P_3$
i)	1	$\frac{2}{15}$	0	$= 28.2$
ii)	0	1	$\frac{5}{10}$	cost $= 31.5$
iii)	0	$\frac{10}{15}$	1	cost $= 31$

and so on are the feasible solutions - out of all these feasible solutions, we need to find the most optimal one.

- In this case,

x_1	x_2	x_3
1	$\frac{5}{10}$	0

profit $= 1 \times 24 + \frac{1}{2} \times 15$

$= 24 + 7.5$

$= 31.5$ is the best optimal solution.

Algorithm Knapsack (m,n)

// m is the capacity of the knapsack having n no. of objects.

// P(1:n) and W(1:n) contain profit and weight ratio.

// Arrange the ratio of profit and weight in the form of $P(i)/W(i) > P(i+1)/W(i+1)$

```
BEGIN
    for i=1 to n
      {
          x[i] = 0
      }
      U = m
      for i = 1 to n
        {
            if (w[i] ≤ U)
              {
                  x[i] = 1.0
                  U = U - w[i]
              }
            else
                break;
        }
      if (i ≤ n)
          x[i] = U/wi
```

HUFFMAN CODING

Steps

1) Find the frequency of each character
Here, the leaf nodes provide the frequency of character

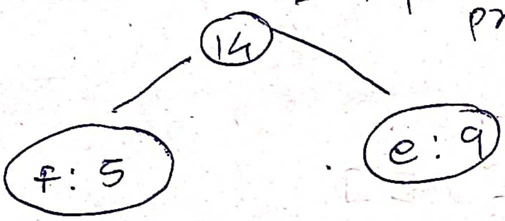
11) Build a heap tree from it.

ex Given, f = 5. b = 13
e = 9 d = 16
c = 12 a = 45.

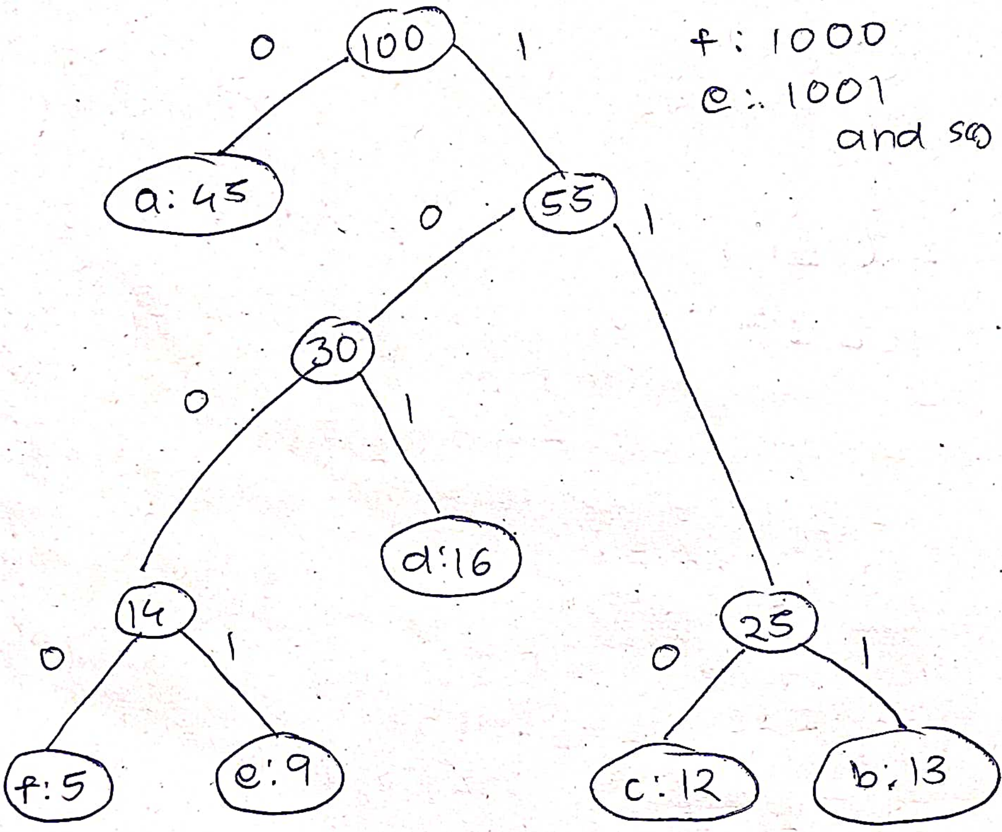
soln: constructing Huffman tree,

f: 5 e: 9 c: 12 b: 13 d: 16 a: 45
14 25 55

sequentially increasing order.
select 2 nodes. → put this in the previous sequence.



codefor
a: 0
f: 1000
e: 1001
and so... on.



(14) Algorithm Huffman (C)

// C is the set of characters
// n is the no. of characters present
// Q is the queue.

BEGIN

1. $n = |C|$
2. $Q = C$
3. for $i = 1$ to $n-1$
4. {
5. LC (Z) $\leftarrow x \leftarrow$ Extract (Q)
   // LC is the left child
6. RC (Z) $\leftarrow y \leftarrow$ Extract (Q)
   // RC is the right child
7. $f(Z) = x + y$
8. Insert min (Q, Z)
9. }
10. Return (min (Q))

END

Time complexity
$T(n) = O(n \log n)$

DYNAMIC PROGRAMMING

- The dynamic programming method
  - uses divide & conquer
  - divides the problem into dependent subproblems.
  - always provides an optimal solution
  - parent is dependent on the various of leaf nodes.

CRITERIA

i) Characterize the structure of an optimal solution

ii) Recursively divide the value

iii) compute the value using bottom up approach

iv) construct an optimal solution from computed information

MATRIX MULTIPLICATION

15 Algorithm matrix multiplication

// A and B are matrices of size $A_{p \times q}$, B of size $B_{r \times s}$

```
BEGIN.
    If (q ≠ r)
      {
         print " Not possible"
      }
      else
        {
          for (i=1; i<=p; i++)
            {
              for (j=1; j<= s; j++)
                {
                  C[i] [j] = 0
                  for (K=1; K<=q; K++)
                   {
                     c[i][j] = (C[i] [j]
                                + A[i][k]
                                * B[k] [j])
                   }
            }
        }
```

computational cost
$= p \times s \times (q+1)$
$= O(psq)$
This is the cost when 2 matrices are multiplied.

CHAIN MATRIX MULTIPLICATION

- when sequencing of matrix is required for the minimum computational cost. This is known as chain matrix multiplication.

$((A_1 A_2) A_3)$
$(A_1 (A_2 A_3))$.

$((A_1 A_2) A_3)$

A_1	5×10
A_2	10×25
A_3	25×7

$5 \times 10 \times 25 = 1250$
$c_1 = 5 \times 25$.
$1250 + 5 \times 25 \times 7$
$= 1250 + 975$
$= 2225$.

$(A_1 (A_2 A_3))$

A_1	5×10
A_2	10×25
A_3	25×7

$c_1 = 1750$

OPTIMAL PARENTHESIS

- parenthesis provides break.
- No. of parenthesis required for a matrix A is

$$P(n) = \sum_{K=1}^{n-1} P(K)\,P(n-K)$$

ex. $P(3)$

$$P(3) = \sum_{K=1}^{3-1} P(1) \cdot P(2)$$

$$= P(1)\,P(3-1) + P(2)\,P(3-2)$$

$$= P(1)\,P(2) + P(2)\,P(1)$$

↑
Dependent problems.

- To find the cost we use

$$m(i,j) = \{\min\{m(i,k) + m(k+1,j) + p_{j-1} p_k p_j\}\}, \; i \neq j)$$

(16) Algorithm Matrix chain multiplication (p)

```
// P is the sequence
n = length(p)
construct two tables m[1...n] and
    s[1...n]
for i = 1 to n
        m[1,1] = 0
for l = 2 to n
        for i = 1 to n-l+1
                j = i+l-1
                m[i,j] = ∞
        for k = i to j-1
            q = m[i,k] + m[k+1,j] + p_{i-1} p_j p_k
        if  q < m[i,j]
            m[i,j] = q
            s[i,j] = k
return m and s
```

Time complexity $= O(n^3)$

(17) Algorithm Print optimal Parenthesis(s, i, j)

```
{
1.  if i = j
2.      print "Ai"
3.  else print "("
4.      print-optimal parenthesis (s, i, s[i,j])
5.      print-optimal parenthesis (s, s[i,j]+1, j)
6.  print ")"
}
```

ELEMENTS IN DYNAMIC PROGRAMMING

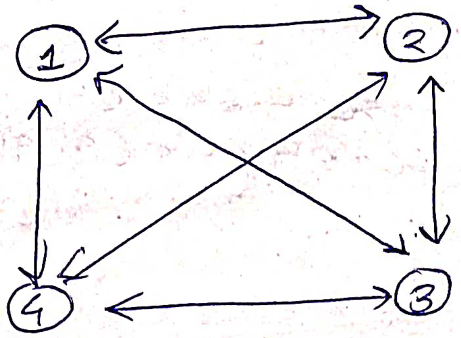
~~There are 3 main elements in DP :-~~

There are 3 main elements
1) optimal substructuring
11) overlapping subproblems
111) variant (memoization)

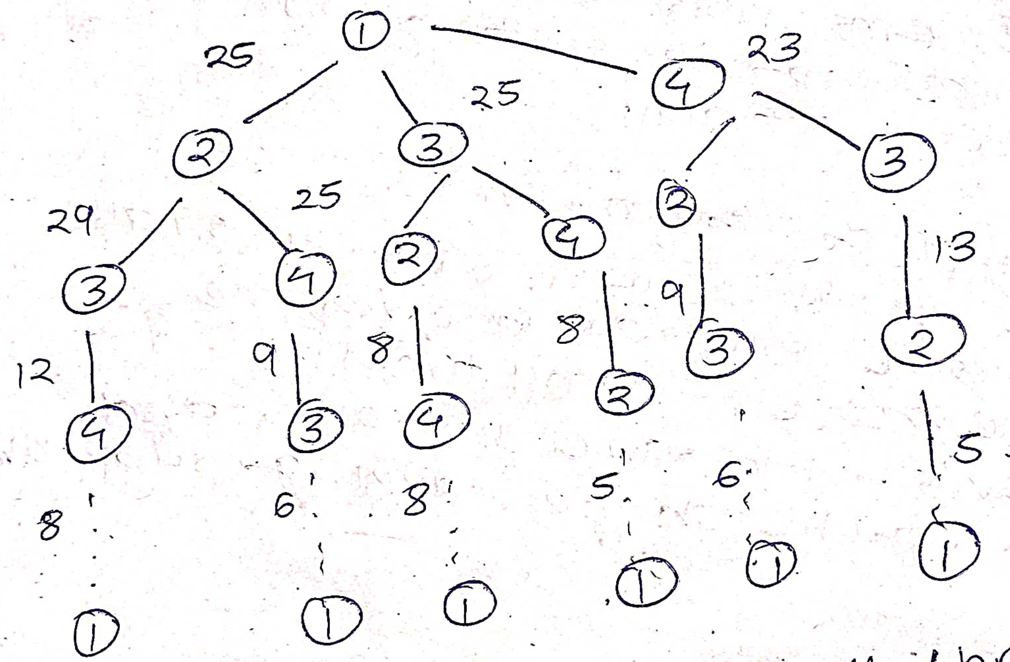
1) optimal substructuring
- split the problem into subproblems
- subproblems must be optimal
- otherwise, optimal splitting wouldn't have been optimal.

11) overlapping ~~subproblems~~ subproblems
- space of ~~subproblem~~ must be small
- recursive solution resolves the same problem many times.
- we revisit the same ones over and over again, for overlapping subproblems

TRAVELLING SALESMAN PROBLEM



$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 1 \\
2 & 1 & 0 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 1 & 0
\end{array}
\qquad
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 10 & 15 & 20 \\
2 & 5 & 0 & 9 & 10 \\
3 & 6 & 13 & 0 & 12 \\
4 & 8 & 8 & 9 & 0
\end{array}
$$



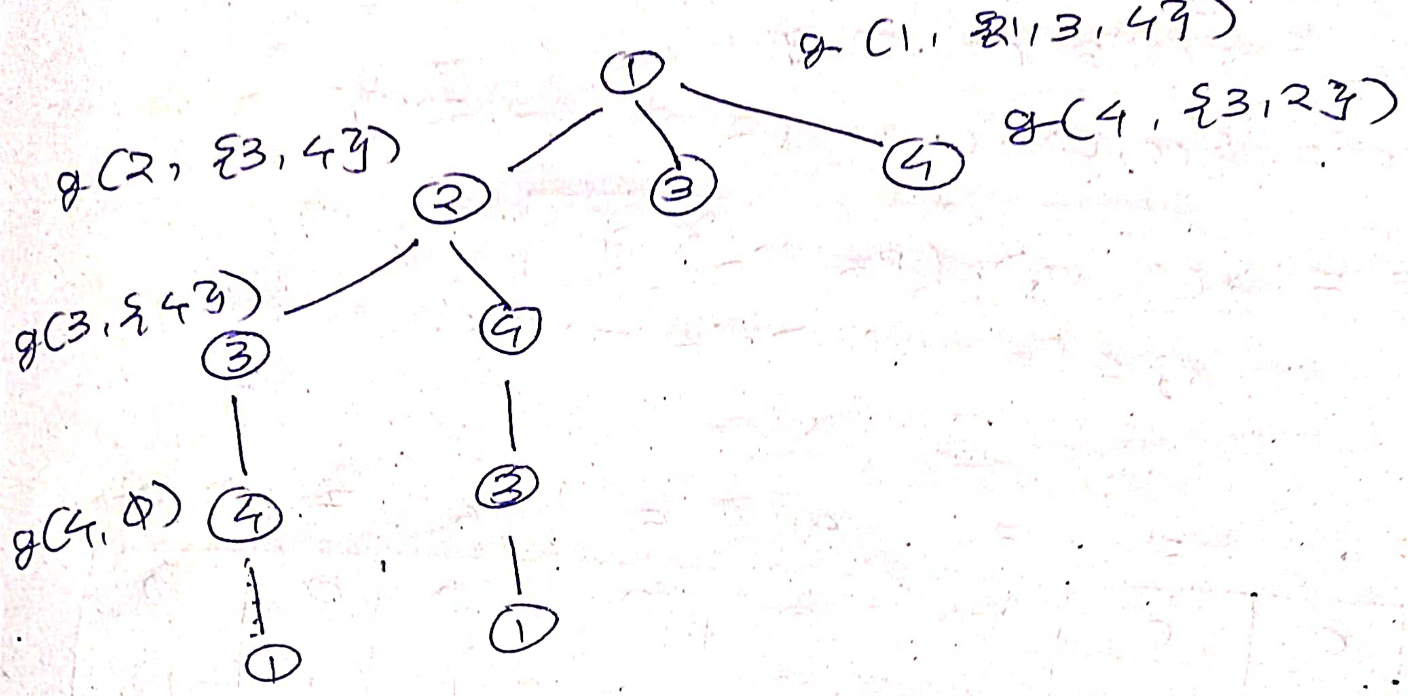
- The aim of the problem is to visit the nodes, complete the tour with minimum cost.

so, $g = \{1, \{2,3,4\}\}$

$= \min(g(2, \{3,4\}), g(3, \{2,4\}), g(4, \{2,3\}))$

- upon generalizing, we get

$$g(1, S) = \min_{k \in S} \{ c_{1k} + g(k, S - \{k\}) \}$$



and so on.

Time complexity $= O(2^n)$

- ∞ means no path.
- Hence, cost is infinite.

0/1 KNAPSACK

Given, $P = \{2, 3, 1, 4\}$
$W = \{3, 4, 6, 5\}$
$m = 8$

the given data of a 0/1 knapsack.

ex:- $P = \{1, 2, 5, 6\}$
$W = \{2, 3, 4, 5\}$
$m = 8$

$S^0 \rightarrow$ means initialization of putting objects in the knapsack.

$S^0 = \{(0,0)\}$

$S_1^0 = \{(1,2)\}$

$S_1^1 = \{(0,0), (1,2)\}$

$S_0^1 = \{(2,3), (3,5)\}$

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

$\boxed{S^i = \{(P_j\ W_j) \cdot (P_K\ W_K)\}}$ then discard

$P_j \le P_K, \quad W_j > W_K \rightarrow$

	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
2	0	1	2	2	3	3	3	3
3	0	1	2	5	5	6	7	7
4	0	1	2	5	6	6	7	8.

Here,

$m[i, w_i] = \max\{m(i-1, w_i), m(i-1, w-w_i) + P_i\}$

$m(2,3) = \max\{m(1,3), m(1,0)+2\}$

$= \max\{1, 2\}$

and so-on.

LONGEST COMMON SUBSEQUENCE

P = <K L O K M K N K L O K>

Q = <X K L L K N K L L K N Y Y>

S = <K L K N . K L K>

P = <L M N>

Q = <M L N

S = <. L N>

S = <M N>.

① If $X_m = Y_n$
then $Z_K = X_m = Y_n$
Z_{K-1} is an LCS of X_{m-1} and Y_{n-1}

② If $X_m \neq Y_n$
then $Z_K \neq X_m$
Z from LCS X_{m-1} and Y_{n-1}

③ If $X_m \neq Y_n$
then $Z_K \neq Y_n$
Z from LCS X_{m-1} and Y_{n-1}

So, the recurrence relation is

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

18 Algorithm LCS (x, y)

```
{
  m = length (x)
  n = length (Y)
  for i = 1 to m
     {
        c [i, 0] = 0
     }
  for j = 1 to n
     {
        c [0, j] = 0
     }
  for i = 1 to m
     { for j = 1 to n
         {  if (xi = yi)
              {
                c [i, j] = c [i-1, j-1] + 1
                b [i, j] ← "↖"
              }
            else if (c [i-1, j] ≥ c [i, j-1])
              { c [i, j] ← c [i-1, j]
                b [i, j] ← "↑"
              }
            else if (c [i, j] ← c [i, j-1])
                b [i, j] ← "←"

         }
     }
  return c and b
}
```

Depth First Search

(19) Algorithm DFS (G)

// G is the graph consisting of set of vertices and edges.

```
BEGIN
for each vertex u ∈ G.V.
    {
        u.color = W
        u.π = NIL
    }
time = 0
    for each vertex u ∈ G.V.
    {
        if u.color = W
            DFS_VISIT (G, u)
    }
END
```

Total $= 6n - 4$
$= O(n)$

(20) Algorithm DFS_VISIT (G, u)

```
BEGIN
time = time + 1
u.d = time
u.color = G
for each vertex v ∈ G, adj [u]
    {
        if v.color = W
            v.π = u
        DFS_VISIT (G, v)
    }
u.color = B
time = time + 1
u.f = time
END
```

$T = O(n+e)$

SPANNING TREE

- Spanning tree form from undirected graph.
- A graph without forming a cycle is called spanning tree.
- It should include all the vertices.

$$G = (V, E)$$
$$ST = (V', E') \quad \text{where } V' = V, \; E' \subset E.$$

- For a single cycle,

No. of ST formed $= {}^{V}C_{V-1}$

- For more than one cycles,

No. of ST formed $= {}^{|E|}C_{|V|}$ − No of cycles.

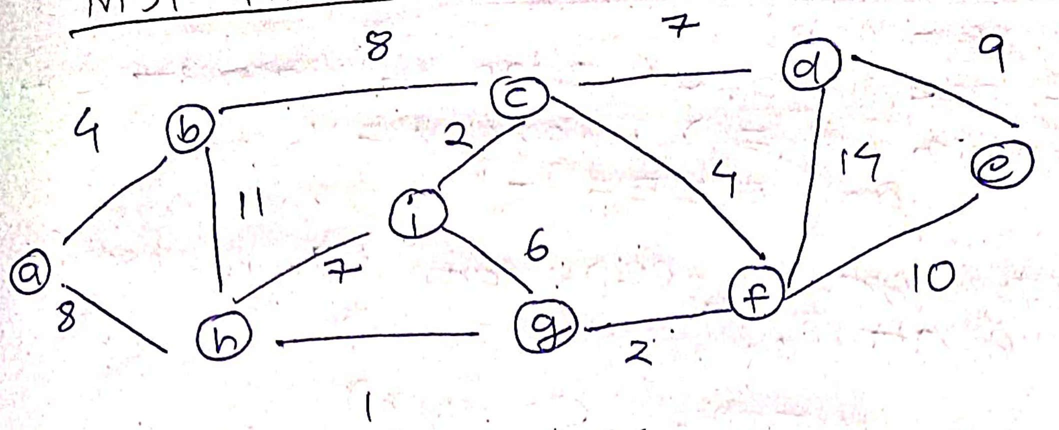
Minimum spanning Tree

- Graph must be i) weighted
  ii) undirected
- Thus, we apply greedy strategy.

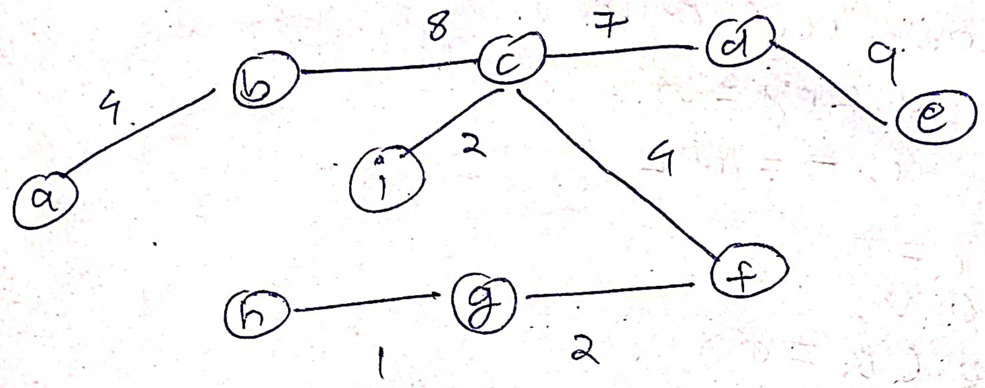
(21) **Algorithm spanning Tree (G, w)**

1. $A \leftarrow \phi$
2. while A doesnot form a spanning tree
3. do find an edge (u, v) that is safe in A.
4. $A \leftarrow A \cup (u, v)$
5. return A.

MST- Kruskal Method



i) write all the vertices
ii) sort the edges in ascending order of its cost.
iii) connect the vertices



Minimum cost = 37

(22) Algorithm MST-Kruskal (G, w)
// G is a graph consisting of vertices & edges
// w is the cost matrix

1. A ← Φ
2. for each vertex v ∈ V[G]
3. do MAKE-SET (v)
4. sort the edges of E into ascending order by weight w
5. for each edge (u, v) ∈ E taken in ascending order
6. do if FIND-SET (u) ≠ FIND-SET (v)
7. A = A ∪ {(u, v)}
8. return A.

PRIM'S ALGORITHM

- Remove self loops
- Remove one parallel edge of highest cost.
- A graph G' (V', E') where V' = V
  and E' ⊂ E = V − 1

- A connected graph keeps forming, in case of Prim's algorithm.

(23) Algorithm MST_Prim (G, w, r)

// G is the graph
// r is the starting vertex
// w is the weight

```
for each u ∈ G.V.
        u.key = ∞
        u.π = NIL
r.key = 0
Q = G.V.
while (Q ≠ φ)
        u = EXTRACT_MIN (Q)
        for each v ∈ G.adj (u)
        {
          if v ∈ Q and w (u, v) < v.key
                v.π = u
                v.key = w (u, v)

        }
```

TC = E log V

Algorithm DIJKSTRA (G, w, s)

// G is the graph, s is the single source
// A is a set, Q has all the vertices and is a priority queue.

```
Initialize single source (G, s)
A = Φ
  Q = G.V.
    while Q ≠ Φ
        u = EXTRACT-MIN (Q)
        A = A ∪ {u}
    for each vertex v ∈ G.adj (u)
        RELAX (u, v, w)
```

Algorithm Initialize single source (G, s)

```
for each vertex v ∈ G.v
        v.d = ∞
        v.π = NIL
s.d = 0.
```

Algorithm Relax (u, v, w)

```
if d(u) + w(u, v) < d[v]
        d[v] = d(u) + w(u, v)
        v.π = u
```

- If the cost is negative, this algorithm cannot work
- It works only for positive weights.

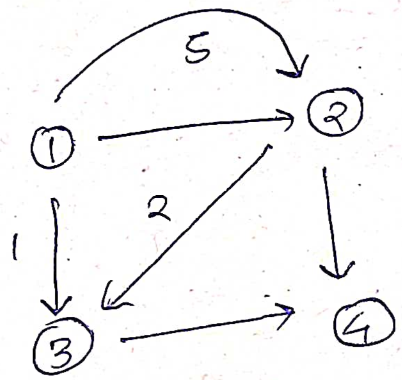
Algorithm Bellman Ford (G, w, s)

```
Initialize single source (G, s)
for i = 1 to |V| - 1
      for each edge (u, v) ∈ E
            RELAX (u, v, w)
      for each edge (u, v) ∈ E
            if d(v) > d(u) + w(u, v)
                  return false
```

$TC = O(V \cdot E)$

10/11/22

- All pair shortest path.



- minimum cost has to be considered

From the graph, create adjacent matrix

Note down all the cost

'n' vertices → matrix size = n

cost to form matrix = $O(n^2)$

diagonal elements = 0.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \end{array}$$

(26) Algorithm Floyd-Warshall (W)

1. $n = W.row$
2. $A^0 = W$
3. for $k = 1$ to n
4. Let $A^k = d_{ij}^{k}$ be a matrix of size $n \times n$.
5. for $i = 1$ to n
6. for $j = 1$ to n
7. $(d_{ij}^{k} = (d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$

- A^1 indicates the cost of vertices by considering 1 as the intermediate vertex.

since, there are 3 loops, $TC = O(n^3)$

STRING MATCHING ALGORITHM

(27) Algorithm Rabin karp (T, P)

// T is the text array
// P is the pattern array

1. $n = length(T)$
2. $m = length(P)$
3. $h = d^{m-1} \bmod q$
4. $p = 0$
5. $t_0 = 0$
6. for $i = 1$ to m
7. $p = (dp + P[i]) \bmod q$
8. $t_0 = (dt_0 + T[i]) \bmod q$
9. for $s = 0$ to $n - m$
10. if $p == t_s$
11. if $P[1..m] = T[s+1 .... s+m]$ — $(n-m)$
12. print pattern occurs with shift 's' — $(n-m)$
13. if $s < n-m$ — $(n-m)$
14. $t_{s+1} = (d(t_s - T(s+1)h + T[s+m+1]) \bmod q$ $(n-m)$

$O(n-m+1)$

(28) Algorithm Prefix Function (P)

1. $m = p.\log \pi$
2. compute $\pi[1 \ldots m]$
3. $\pi[1] = 0$
4. $K = 0$
5. for $q = 2$ to m
6. while $(K > 0$ and $p[K+1] \neq p[q])$
7. $K = \pi[K]$
8. if $(p[K+1]) = p[v]$
9. $K = K+1$
   $\pi[q] = K$
10. return π
11.

ex T: ababaca

1	2	3	4	5	6	7
a	b	a	b	a	c	a
0	0	1	2	3	0	1

(29) Algorithm KMP (T, P)

1. $n = T.length$
2. $m = P.length$
3. π = compute_PREFIX-FUNCTION
4. $q = 0$
5. for $i = 1$ to n
6. {
7. while $q > 0$ and $p[q+1] \neq T[i]$
   $q = \pi[q]$
8. if $p[q+1] = T[i]$
9. $q = q+1$
10. if $q = m$
11. print "pattern occur with shift $(i-m)$
12. $q = \pi[q]$

$T = O(n)$

BOYER MOORE

T: welcome_to_VSSUT

p: come

Bad numbers

c	3
o	2
m	1
e	4

Good suffix table

come	4
come	4
come	4
come	4

ajiji	2
ajiji	2
ajiji	5
ajiji	5
ajiji	5

barber	3
barber	3
barber	3
barber	6
barber	6
barber	6

POLYNOMIAL EVALUATION

polynomial addition - $O(n)$
polynomial multiplication - $O(n^2)$

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$$

$$f(x) = \sum_{k=0}^{n} Y_K \frac{\prod_{j \neq K} (x - x_j)}{\prod_{j \neq K} (x_K - x_j)}$$

is called Lagrange's equation.

DFT (Discrete fast fourier Transform)

complex nth root of unity (1)

$$\omega^n = 1$$

$$\omega \text{ value} \leftarrow e^{2\pi i k/n}$$

$$e^{iu} = \cos(u) + i\sin(u)$$

$$\Rightarrow (e^{2\pi i k/n})^n = 1$$

$$\Rightarrow e^{2\pi i k} = 1$$

$$e^{2\pi i k} = \cos(2\pi k) + i\sin(2\pi k)$$
$$= 1$$

- A polynomial $A(x)$ of degree bound n at the nth complex nth root of unity. i.e. $\omega_n^0, \omega_n^1, \ldots \omega_n$ and assume n is the power of

$$y_k = A(\omega_n^k)$$

$$y_k = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$