28/09/20 # Design & Analysis of Algorithms

(10-12 M)

Text-books:

→ Introduction to Algorithms
- CLR (Coremen)

→ Fundamentals of Algorithms
- Sahani

Syllabus:

1. Analysis of Algorithms

→ Algorithm concept & lifecycle

→ Need for analysis

→ Methodology of analysis

→ Types of analysis

**→ Asymptotic Notations (ASN)

- Definitions & properties
- Problem solving

*→ Framework for analyzing recursive & non-recursive algorithms.

*→ Analysis of Program segments with loops.

→ Space complexities

2. Design Strategies:

(i) Divide & Conquer

→ Control Abstraction

→ Max Min

→ Merge Sort

→ Quick sort

→ Binary Search

→ Matrix Multiplication

→ long integer multiplication

→ Master's theorem

(ii) * Greedy Method

→ Control Abstraction

→ Job seq with deadlines

→ knapsack problem

→ Merge patterns

→ Huffman coding

→ Spanning trees

→ Shortest paths

(iii) ** Dynamic Programming

→ General method

→ DP vs GM v D&C

→ Multistage Graphs

→ 0/1 knapsack

→ All pairs shortest path

→ Travelling salesperson.

→ Matrix chain Product

→ Longest common subsequence

→ Bellman Ford algorithm.

→ OBST

→ Reliable System Design

→ kadane Algorithm.

(iv) Graph Techniques

→ Traversals

→ Parenthesization theorem

→ Components; Articulation points.

(v) Heap algorithms

(vi) Sorting techniques

→ Classification

→ Case studies

———o———

Algorithm: Consists of finite set of steps/stmts to solve a given problem.

→ Each step/stmt may involve one/more fundamental operation.

→ Every operation must be

(i) Definite (i.e., clear) (unambiguous)

(ii) Effective (i.e., finite time)

→ Every algorithm must halt in finite time.

→ Every algorithm may accept 0 (or) more inputs and must produce atleast one output.

Lifecycle steps:

1. Problem definition
2. Requirements.
3. Design [logic]
4. Develop the algorithm (Pseudo code)
5. Validation (Proving its correctness)
6. Analysis
7. Implementation
8. Testing & Debugging

Analysis

need for analysis

- to determine resource consumption <time, memory, power, cost (registers etc.)>
- Performance Analysis (Comparision blw diff algorithms)

Methodology of analysis :

Aposteriori Analysis (Experimentation analysis)

→ This analysis done by converting algorithm into a program and we analyze by running it on a platform.

This analysis depends on the processors, memory speed, OS, programing language (compiler) i.e., platform dependent.

Advantages:

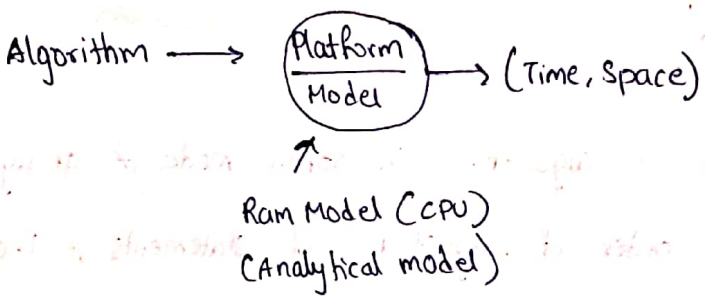
* Gives exact values in units of time/space.

Drawback:

* reports different values from platform to platform
* difficult to make performance comparision.
* Difficult to determine the exact times (∵ OS & other programs could run in parallel).

Hence

Apriori Analysis: (Platform Independent Analysis)



This is a hypothetical model with infinite memory and we assume that it takes one unit of time for every fundamental operation.

→ This gives estimated time.

Adv:
→ This analysis do help in performance comparision.
→ Easy to carry out.

Components of analytic framework used in apriori analysis

→ A language for describing algorithm (Pseudocode)
→ A computational model (ram model) that algorithms execute within it.
→ A metric for measuring running time of algorithm.
i.e., basic / fundamental operation
→ An approach for characterizing running times of algorithms (i.e., Asymptotic notation)

Algorithm Test — time-apriori mode (Step count method)

```
{
1. x ← y+z ..... 2
2. for i ← 1 to n ..... 1 + (n+1) + n
       x ← y+z ..... n+n
3. for i ← 1 to n ..... 1+(n+1)+n
       for j ← 1 to n ..... n + n(n+1) + n.n
           a ← b+c; ..... n.n + n.n
}
```

(2 → assign, → comparisions, → increaments)

$5 + 8n + 4n^2$

∴ time required is

$$T(n) = 4n^2 + 8n + 5$$

↓

i/p size

The running time of algorithm in apriori mode of analysis is determined by order of magnitude of statements of the algorithm.

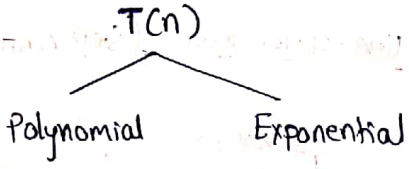
Here, order of magnitude refers to the frequency of the fundamental operation in the statement.

```
Algorithm Test                          order of magnitude
{
  1. x ← y+z  ----------                    1
  2. for i ← 1 to n }
       x ← y+z      } ----------           n
  3. for i ← 1 to n }
       for j ← 1 to n } ----------          n²
         a ← b+c;
}
```

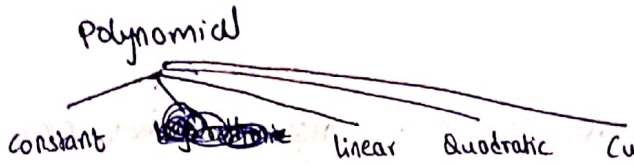
$$T(n) = n^2 + n + 1$$

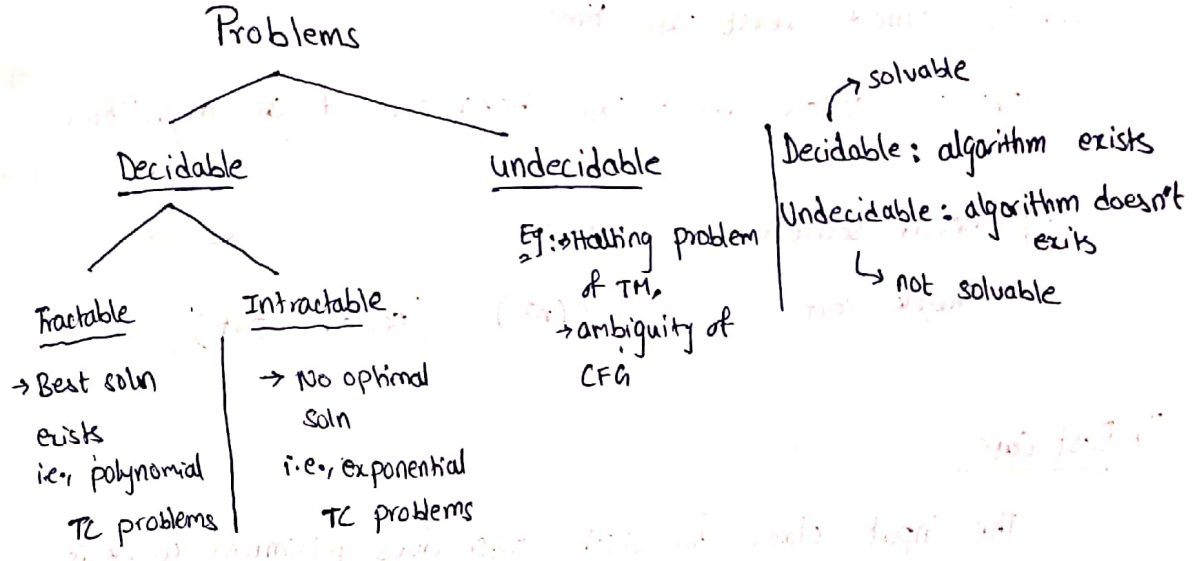
The objective of Apriori analysis representing running time of algorithm is to represent the time of an algorithm as a mathematical function of input size.

↳ derived or expressed using step count (or) order of magnitude



→ polynomial func have lesser rate of growth and exponential functions have faster rate of growth.





→ Tractable problems fall under the class of P (Polynomial)

→ NP Problems (Non-Deterministic Polynomial)

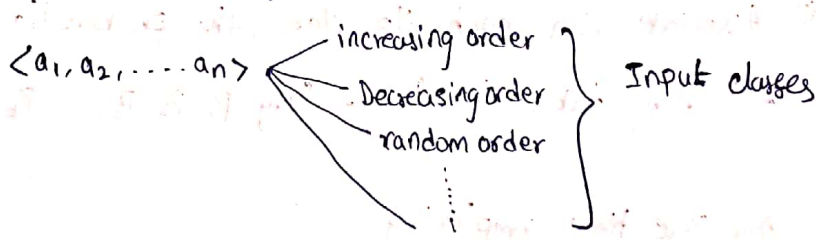
Those are problems that are solvable in polynomial time on non-deterministic TM.

However they take exponential time on deterministic TM.

→ Most of the intractable problems are NP.

Types of Analysis / Behaviours of algorithm:

→ Input of fixed size (say 'n')



Let no of input classes be k.

Based on this we broadly classify the input classes into 3 categories

(i) Worst-Case:

The input class for which algorithm does maximum work is called worst-case input. The corresponding time taken is called worst case time.

This is called worst case behaviour of an algorithm.

Eg: linear search ..... O(n)

Quick sort ..... $O(n^2)$ (sorted input)

(ii) Best Case:

The input class for which algo does minimum work is best case & corresponding time is best case time.

Eg: Linear search .... O(1)

Quick Sort ..... O(nlogn)

(iii) Average Case Analysis: (Not needed for gate)

This is carried out in 3 steps:

(i) Determine all the input classes.

$\langle I_1, I_2, I_3, \ldots I_k \rangle$ (say size 'n')

(ii) Determine the time taken by the algorithm for each input class. (say $t_1, t_2, \ldots t_k$)

(iii) Associate with each i/p class, the probability with which algo may take i/p from. (say $P_1, P_2, P_3, \ldots P_k$)

Avg case time complexity

$$A(n) = \sum_{i=1}^{k} t_i * P_i$$

→ Let $w(n)$, $B(n)$, $A(n)$ be worst case, best case, avg case timens.

⇒ $\boxed{B(n) \leq A(n) \leq w(n)}$

Note:

→ $B(n) = A(n) = w(n)$ ⇒ Uniform Behaviour

Eg: Merge Sort ... $O(n \log n)$.

→ $[B(n) = A(n)] < w(n)$

Eg: Quick Sort .... $B(n) = A(n) = O(n \log n)$

$w(n) = O(n^2)$

→ $B(n) < [A(n) = w(n)]$

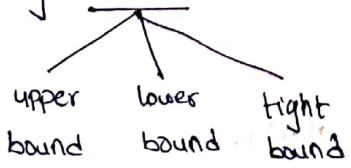
Eg: linear search .... $B(n) = O(1)$

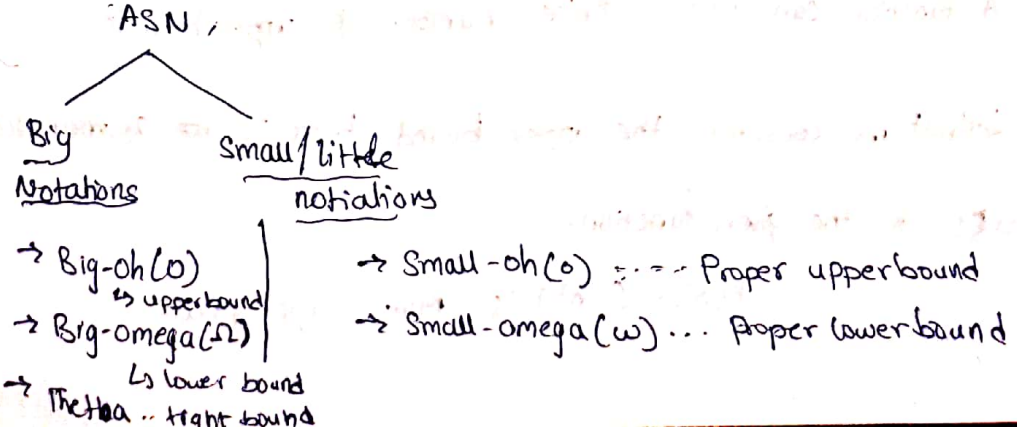
$A(n) = w(n) = O(n)$

→ $B(n) < A(n) < w(n)$

Asymptotic Notations (ASN):

Asymptotic notation is a mathematical tool which is used for representing bounds of a function.





Let 'f' & 'g' be functions from set of real numbers to real numbers.

(i) Big-Oh (O): (upper bound)

$f(x)$ is $O(g(x))$

iff $f(x) \leq c \cdot g(x)$ for some $c > 0$ whenever $x > k$

(k is a constant)

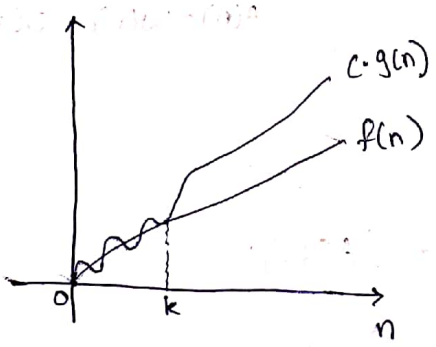
Eg:- $f(n) = 1 + n + n^2 = O(n^2)$

$1 + n + n^2 < n^2 + n^2 + n^2$

$\therefore f(n) < 3 \cdot n^2, \quad n > 1$

$c \quad g(n)$

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



also $f(n) = 1 + n + n^2 < 3n^3$

$\Rightarrow f(n) = O(n^3)$

$\therefore f(n)$ is $O(n^2), O(n^3), O(n^4) \ldots$

$\therefore$ A function can have infinite number of upper bounds

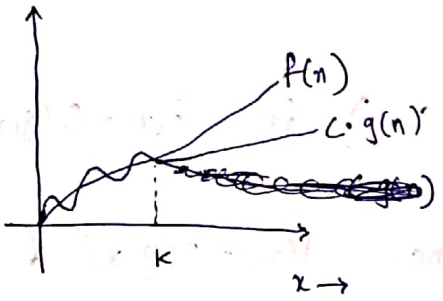
By default we consider the upper bound that is asymptotically closest to the given function.

$\therefore f(n) = O(n^2)$ is more appropriate.

(ii) Big-omega (Ω): (Lower bound of a function)

$f(x)$ is called $\Omega(g(x))$

iff $f(x) \geq c \cdot g(x)$ for some $c > 0$ & whenever $x > k$



$f(n) = 1+n+n^2 \geq n^2$

$\Rightarrow f(n) = \Omega(n^2)$ —

$\therefore f(n) = 1+n+n^2 > 1 \Rightarrow f(n) = \Omega(1)$.

$f(n) = 1+n+n^2 > n \Rightarrow f(n) = \Omega(n)$

$\therefore$ A function can have several lower bounds

$\therefore \Omega(n^2)$ is asymptotically closer to $f(n)$

$\therefore f(n) = \Omega(n^2)$ is more appropriate.

Eg: $f(n) = n \rightarrow \begin{cases} O(n) \\ \Omega(n) \end{cases}$

$f(n) = 2^{100} \rightarrow \begin{cases} O(1) \\ \Omega(1) \end{cases}$

$f(n) = n + \log n \rightarrow \begin{cases} O(n) \\ \Omega(n) \end{cases}$ .... $\because n + \log n > n$

(iii) Theta(θ) : (Tight bound)

$f(x)$ is equal to $\theta(g(x))$

iff $c_1 \cdot g(x) \le f(x) \le c_2 \cdot g(x)$ $\quad c_1 > 0 \;\&\; c_2 > 0$

whenever $x > k$

In other words

$f(x) = \theta(g(x))$ iff $f(x) = O(g(x)) \;\&\; f(x) = \Omega(g(x))$

∴ In the cases where $f(x) = O(g(x)) \;\&\; f(x) = \Omega(g(x))$ it is more appropriate representation would be

$$f(x) = \theta(g(x))$$

Eg :

$f(n) = 10^{10} \begin{cases} O(1) \\ \Omega(1) \end{cases} \quad \therefore \theta(1)$

$f(n) = n^2 + n\log n + 2^{100} \begin{cases} O(n^2) \\ \Omega(n^2) \end{cases} \quad \therefore \theta(n^2)$

$f(n) = \sum_{i=1}^{n} 1 \begin{cases} O(n) \\ \Omega(n) \end{cases} \quad \therefore \theta(n)$ $\qquad \because \sum_{i=1}^{n} 1 = n$

$f(n) = \sum_{i=1}^{100} 1 \begin{cases} O(1) \\ \Omega(1) \end{cases} \quad \therefore \theta(n)$

$f(n) = \sum_{i=1}^{n} n \cdot 1 \begin{cases} O(n^2) \\ \Omega(n^2) \end{cases}$

$\sum_{i=1}^{n} n = n \cdot n = n^2$ $\qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$f(n) = \sum_{i=1}^{n} i \begin{cases} O(n^2) \\ \Omega(n^2) \end{cases}$

$f(n) = \sum_{i=1}^{n} O(n) \begin{cases} O(n^2) \\ \Omega(n^2) \end{cases}$ $\qquad \because \sum_{i=1}^{n} O(n) = O(n) \sum_{i=1}^{n} 1 = O(n^2)$

$f(n) = \sum_{i=1}^{100} O(n) \begin{cases} O(n) \\ \Omega(n) \end{cases}$

→ $f(x) = \sum_{i=1}^{n} i^2 \begin{cases} O(n^3) \\ \Omega(n^3) \end{cases}$

→ $f(n) = \sum_{i=1}^{n} i^3 \begin{cases} O(n^4) \\ \Omega(n^4) \end{cases}$

*→ → $f(n) = \sum_{i=1}^{n} \frac{1}{x} \begin{cases} O(\log n) \\ \Omega(\log n) \end{cases}$

$\because \sum_{i=1}^{n} \frac{1}{x} = \int_{1}^{n} \frac{1}{x} dx$

$= (\log x)_1^n = \log n$

→ $f(n) = \sum_{i=1}^{n} 2^i$

$= 2^1 + 2^2 + \cdots + 2^n$

$= 1 + 2 + 2^2 + \cdots + 2^n - 1$

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

$= 2^{n+1} - 1$

$O(2^n)$

$\leq 2^2 \cdot 2^n$

$= O(2^n)$

$\therefore f(n)$ is $O(2^n)$

$f(n)$ is $\Omega(2^n)$

Note:

$\sum_{i=1}^{n} O(1) \cong$ for i=1 to n
a=a+b

$\sum_{i=1}^{n} O(n) \cong \sum_{i=1}^{n} \sum_{j=1}^{n} O(1)$

$\cong$ for i=1 to n
for j=1 to n
a=a+b

→ $f(n) = \sum_{i=1}^{n} i \cdot 2^i$

$f(n) = 1 \cdot 2 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + n \cdot 2^n$

$2 \cdot f(n) = 1 \cdot 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \cdots + n \cdot 2^{n+1}$

$2f(n) - f(n) = -2 - 2^2 - 2^3 - \cdots - 2^n + n \cdot 2^{n+1}$

$\Rightarrow f(n) = 1 - (1+2+2^2+\cdots+2^n) + n \cdot 2^{n+1}$

$f(n) = 1 + n \cdot 2^{n+1} - \frac{2^{n+1}-1}{2-1}$

$f(n) = 2 + n \cdot 2^{n+1} - 2^{n+1} = \underline{(n-1)2^{n+1} + 2}$

$\Rightarrow f(n) = O(n \cdot 2^n)$

$f(n) = \Omega(n \cdot 2^n)$

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

$* \rightarrow f(n) = \sum_{i=1}^{n} 1/2^i$

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

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

$$= \frac{2^n - 1}{2^n} = 1 - \frac{1}{2^n}$$

$\therefore f(n) = O(1)$ $\quad \because 1 - \frac{1}{2^n} \leq 1$

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

$\rightarrow f(x) = \sum_{i=0}^{\infty} x^i$ where $x < 1$

$\Rightarrow f(x) = \frac{1}{1-x}$ $\quad \left(\because x < 1 \Rightarrow 1-x > 1, \ \frac{1}{1-x} < 1\right)$

$\therefore f(x) = O(1)$ $\quad$ (~~$\because x$ is a constant~~)

$\rightarrow f(n) = \sum_{i=j}^{n} a^i = \frac{a^{n+1} - a^j}{a-1}$

$\Rightarrow f(n) = O(a^n)$

$\rightarrow f(n) = \prod_{i=1}^{n} 1 = O(1)$

→ $f(n) = \prod_{i=1}^{n} i$

⇒ $f(n) = n!$

$= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n < n \cdot n \cdot n \cdots n$ (n times)

⇒ $f(n) < n^n$

$\therefore f(n) = n! = O(n^n)$

However $f(n) \neq \Omega(n^n)$

$n! = \Omega(n)$
$= \Omega(n^2)$
$= \Omega(2^n)$
$= \Omega(an)$

→ $f(n) = \log n!$

$\log n! < \log n^n$

⇒ $\log n! < n \log n$

⇒ $\log n! = O(n \log n)$

$f(n) = n! = O(n^n)$
$n! \neq \Omega(n^n)$
$n! = \Omega(n)$
$= \Omega(n^2)$
$= \Omega(2^n) = \Omega(3^n)$
$= \Omega(an)$

→ $f(n) = \sum_{i=1}^{n} \log i$

$= \log 1 + \log 2 + \cdots + \log n$

$= \log n!$

$= O(n \log n)$

However $f(n) \neq \Omega(n \log n)$

$a^{\log_c b} = b^{\log_c a}$

→ $f(n) = (\log n)!$

$< (\log n)^{(\log n)}$

⇒ $f(n) = O((\log n)^{\log n})$

• also $(\log_e n)^{\log_e n} = n^{\log_e (\log n)}$

⇒ $f(n) = O(n^{\log(\log n)})$ $\quad (\because a^{\log_c b} = b^{\log_c a})$

$\sum_{i=1}^{n} \sqrt{i} = \sqrt{1} + \sqrt{2} + \cdots + \sqrt{n}$

$\int_{1}^{n} \sqrt{x} \cdot dx = \frac{2}{3} x^{3/2}$

$\therefore O(n^{3/2})$

→ Smaller functions are in the order of bigger functions

i.e., $f(x) = O(g(x))$

↙ smaller function

↳ bigger function

→ Bigger functions are in the omega of smaller functions.

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

↙ bigger function

↳ smaller function

Eg: if $f(x) = O(g(x))$ then $g(x) = \Omega(f(x))$

Eg: Consider $f(n) = n^2$ & $g(n) = n \log n$

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

&

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

Small / Little Notations:

The bounds provided by big-oh Notations (O, Ω) may or may not be asymptotically tight.

Eg: $f(n) = n = O(n)$ is a tight upper bound

where as $O(n^2)$ is a loose upper bound

Sly $f(n) = n = \Omega(n)$ is a tight lower bound

$f(n) = n = \Omega(\log n)$ is a loose lower bound.

However the bound provided by small notations are always loose.

(iv) Small-oh (o) : (Proper upper bound)

$f(x)$ is $o(g(x))$

iff $f(x) < c \cdot g(x)$, for all $c > 0$ & $x > k$

Eg: $f(n) = n^2 = O(n^2)$

$\neq o(n^2)$

$f(n) = n^2 = o(n^3)$ $\quad \because n^2 < c \cdot n^3$, $\forall c$, $n > 1$

$= o(n^4)$

$\vdots$

(v) Small-omega (ω) : (Proper lower bound)

$f(x)$ is $\omega(g(x))$

iff $c \cdot g(x) < f(x)$ $\quad \forall c > 0$ & $x > k$

$f(n) = n^2$

$\Rightarrow f(n) = \Omega(n^2), \Omega(n \log n), \Omega(n), \ldots$

$f(n) = \omega(n \log n), \omega(n) \ldots$

Analogy b/w ASNs & real numbers:

→ Let f & g be functions ; a & b be real numbers.

$f(n)$ is $O(g(n)) \iff a \leq b$

$f(n)$ is $\Omega(g(n)) \iff a \geq b$

$f(n)$ is $\Theta(g(n)) \iff a = b$

$f(n)$ is $o(g(n)) \iff a < b$

$f(n)$ is $\omega(g(n)) \iff a > b$

Note:

→ $f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$

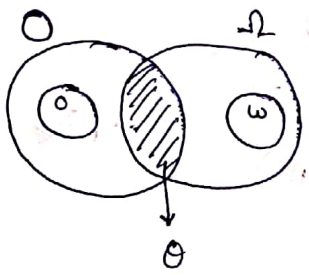
reverse need not to be true.

$\Rightarrow o \subseteq O$

→ $f(n) = \omega(g(n)) \Rightarrow f(n) = \Omega(g(n))$

reverse need not to be tree

$\Rightarrow \omega \subseteq \Omega$



Properties of Asymptotic Notations:

i) Discrete Properties:

	Big-Oh	Big-Omega	Thetha	Small-oh	Small-Omega
Reflexive	✓	✓	✓	X	X
Symmetric	X	X	✓	X	X
Transitive	✓	✓	✓	✓	✓
Transpose Symmetry	f(n) is O(g(n)) iff g(n) is Ω(f(n))		—	f(n) is o(g(n)) if g(n) is ω(f(n))	

Reflexive:

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

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

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

$f(n) \neq o(f(n))$

$f(n) \neq \omega(f(n))$

Symmetric:

If $f(n) = O(g(n))$ then it not neccessary that $g(n) = O(f(n))$

∴ Big-oh is not symmetric

Silly Ω is not symmetric

Transitivity:

$f(n) = O(g(n))$ & $g(n) = O(h(x)) \Rightarrow f(n) = O(h(x))$

$\therefore$ ~~transpose~~ transitive

Sily for Ω & theata, o, ω

(iii) General Properties:

→ if $f(n) = O(g(n))$ then

$a \cdot f(n) = O(g(n))$

where a is a constant.

→ If $f(n) = O(g(n))$ and $d(n) = O(e(n))$ then

(i) $f(n) + d(n) = O(\max(g(n), e(n)))$

(ii) $f(n) * d(n) = O(g(n) \cdot e(n))$

→ ⊛ $\log n^c = O(\log n)$, where c is constant

*→ $(\log n)^x = O(n^y)$ $(x>0, y>0)$

Proof:

$(\log n)^x$ ⊛ n^y

log on both size

$\log((\log n)^x)$ $\log n^y$

$x \log(\log n)$ $y \log n$

⊛ $x \log(\log n) < y \log n$ $(\because x, y$ ax constants$)$

$\therefore$ ~~$O((\log n)^x)$~~

$(\log n)^x = O(n^y)$

→ n^2 is $O(a^n)$, $a>1$

(x & a are constants)

Dominance Relation:

Constants $\underset{A}{<}$ logarithmic $\underset{A}{<}$ polynomial $\underset{A}{<}$ Exponential

$\underset{A}{<}$ ... Asymptotically lesser.

Eg: Which of the below is asymptotically larger.

$f = \sqrt{n}$ $g = \log n$

Sol:

applying log

$\log(f) = \log\sqrt{n} = \frac{1}{2}\log n$

$\log(g) = \log(\log n)$

$\Rightarrow \log(f) > \log(g)$

$\Rightarrow$ f is asymptotically larger than g

$\Rightarrow g = O(f)$

Tricotomy Property:

tricotomy property of real numbers:

For any two real numbers a & b one of the below 3 conditions must hold.

(i) $a > b$

(ii) $a < b$

(iii) $a = b$

However, for any two functions it is not needed that they satisfy tricotomy property

Eg: $f(n) = n$; $g(n) = n^{1+\sin n}$

f & g here doesn't fall under any case

$(f \underset{A}{<} g,\ f \underset{A}{>} g,\ f \underset{A}{=} g)$

(Q1) Consider the equality $\sum_{i=1}^{n} i^3 = x$ & the following choices for x

a) $\theta(n^4)$ b) $\theta(n^5)$ c) $O(n^5)$ d) $\Omega(n^3)$

Sol:

$$\sum_{i=1}^{n} i^3 = x = \frac{n^2(n+1)^2}{4} = \frac{n^4+2n^3+n^2}{4}$$

$$= O(n^4)$$

(or)

$$\Omega(n^4)$$

(or)

$$\theta(n^4)$$

$\therefore$ opt (a) & (b) & (c)

(Q2) Let $w(n)$ & $A(n)$ represent worst case and average case time of an algorithm of input size 'n'. which is always true?

a) $A(n) = O(w(n))$ b) $A(n) = \Omega(w(n))$

c) $A(n) = \theta(w(n))$ d) $A(n) = o(w(n))$

Sol:

$w(n)$ could be sometime equal to $A(n)$

$\therefore$ opt (d) is not possible.

$\therefore$ opt (a)

Eg: Check the correctness of below relations

(i) $100n\log n = O(n\log n)$ --- True

(ii) $2^{n+1} = O(2^n)$ ---- True.

(iii) $2^{2n} = O(2^n)$ ..... False

$\because 2^{2n} = (2^2)^n = 4^n > 2^n$ $\therefore$ False

Note!

$\rightarrow \log_b a = \frac{1}{\log_a b}$

$\rightarrow \log_b a = \frac{\log_x a}{\log_x b}$

(iv) $(n+k)^m \neq \Theta(n^m)$ ..... False.

$(k.m) > 0$

(v) $\sqrt{\log n} = O(\log \log n)$ ..... ~~True~~ False

$\sqrt{\log n}$ $\log \log n$

$\frac{1}{2}(\log * \log n)$ $\log(\log \log n)$

$\Rightarrow \sqrt{\log n} \leq \log(\log n)$

$\therefore$ ~~$\sqrt{\log n} = O(\log \log n)$~~

(vi) $\log n = \Omega(1/n)$ ..... True

(vii) $2^{n^2} = O(n!)$ ---------- False

if $2^{n^2} = O(n!)$ then

$2^{n^2} = O(n^n)$ must be true too

$\Rightarrow 2^{n^2} < n^n$

$\log 2^{n^2} < \log n^n$

$n^2 < n \log n$

which is false

(viii) $n^2 = O(2^{2\log n})$ ..... True

~~B~~ n^2	$2^{2\log n}$
$\log n^2$	$\log 2^{2\log n}$
$2\log n$	$2\log n (\log 2)$

(or)

$2^{2\log_2 n} = 2^{\log_2 n^2} = n^2 (\log_2 2) = n^2$

$\therefore$ true

(ix) $a^n \neq O(n^x)$ $a>1, x>0$ ..... True

29/09/20

Asymptotic Comparisions

(11/02) (ii) $f(n) = n^2 \log n$, $g(n) = n \log^{10} n$ ~~$\log^2 n = \log(\log n)$~~

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

$\log[f(n)] = 2\log n + \log(\log n)$

$\log[g(n)] = \log n + \log^{11} n$

~~$\log(f(n)) \geq \log(g(n))$~~

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

(or)

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

take common factors aside

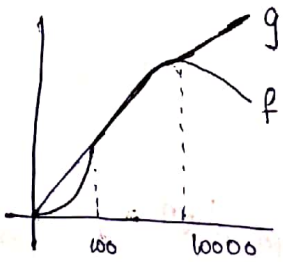
$f(n) = (n\log n)(n)$ $g(n) = (n\log n)(\log n)^9$

So we compare

$f(n) = n$ $g(n) = (\log n)^9$

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

(iii) $f(n) = \begin{cases} n^3, & 0 < n \leq 10000 \\ n, & n > 10000 \end{cases}$ $g(n) = \begin{cases} n, & 0 \leq n \leq 100 \\ n^3, & n > 100 \end{cases}$



$\therefore f(n) = O(g(n))$ $\forall n \geq 100$

also $f(n) = o(g(n))$ $\forall n \geq 10000$

H1/03 $10n\log n < 0.0001 n^2$

$10 \cdot \log n < 10^{-4} n$

$\log n < 10^{-5} n$

let $n = 10^x$

$\log_{10} 10^x < 10^{-5} 10^x$

$x < 10^{x-5}$

put $x=1 \Rightarrow 1 < 10^{-4}$ ✗

$x=2 \Rightarrow 2 < 10^{-3}$ ✗

$\Rightarrow x=5 \Rightarrow 5 < 10^0$ ✗

$\Rightarrow x=6 \Rightarrow 6 < 10^1$ ✓

∴ Min value of $x=6$

H#/04 (i) n^2; $n\log n$; $n\sqrt{n}$; e^n; n; 2^n; $1/n$

Sol:

$n\sqrt{n}$ & $n\log n$

$\sqrt{n} \gtrsim \log n$

$\Rightarrow n\sqrt{n} \gtrsim n\log n$

$\frac{1}{n} = O(1)$

∴ $\frac{1}{n} < n < n\log n < n\sqrt{n} < n^2 < 2^n < e^n$

(ii) 2^n; $n^{3/2}$; $n\log n$; $n^{\log n}$

Sol:

2^n is exponential

$n^{3/2}$, $n\log n$, $n^{\log n}$ are polynomial

$n^{3/2} > n\log n$

$n\log n$ & $n^{\log n}$

log on both sides

$\log(n\log n)$	$\log(n^{\log n})$	$\log(n^{3/2})$
$\log n + \log(\log n)$	$(\log n)(\log n)$	$\frac{3}{2}\log n$
$O(\log n)$	$O((\log n)^2)$	

$\therefore n^{\log n} \geq_A n\log n$ & $n^{\log n} \geq_A n^{3/2}$

$\therefore$ ~~$n\log n < n^{\log n}$~~

$n\log n < n^{3/2} < n^{\log n} < 2^n$

(iii) $n^{1/3}$; e^n; $n^{7/4}$; $n\log^9 n$; 1.001^n

exponential

$n^{7/4}$	$n\log^9 n$
$\log(n^{7/4})$	$\log(n\log^9 n)$
$\frac{7}{4}\log n$	$\log n + 9\log(\log n)$
$O(\log n)$	$O(\log n)$

Now we cancel out common terms & compare

$n^{3/4}$	$\log^9 n$
$\log(n^{3/4})$	$\log(\log^9 n)$
$\frac{3}{4}\log n$	$9\log(\log n)$

$(\log n)^9$

$\Rightarrow n^{7/4} \geq_A n\log^9 n$

$\Rightarrow n^{1/3} < n\log^9 n < n^{7/4} < 1.001^n < e^n$

(H1/06)

$g(n)$ ——— $f(n) = O(g(n))$
$f(n)$ ——— $g(n) \neq O(f(n))$
$\Rightarrow$ $g(n)$ is loose upper bound of $f(n)$

$g(n) = O(h(n))$
$h(n) = O(g(n))$
$\Rightarrow$ $g(n)$ is right upper bound of $h(n)$ and vice versa
i.e., $g = \Theta(h)$

$\therefore f < (g = h)$

$\therefore f(n) = O(h(n))$ ✓ (transivite property)

$f(n) + h(n) = O(g + h)$ ✓

$h(n) \neq O(f(n))$ ✓

$f(n) \cdot g(n) \neq O(g(n) \cdot h(n))$ ×

Framework for analysis of non-recursive & recursive algorithm

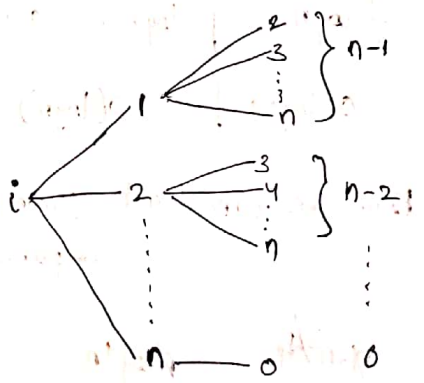
↓ general problems without design strategies.

(H1/07) Naive/Brute force

```
Algo Leaders1(A,n)
{
    for i=1 to n
    {
        for j=i+1 to n
        {
            if(A[i]<A[j])
                break;
        }
        if(j=n+1)
            print(A[i]);
    }
}
```

TC: $O(n^2)$

No of Comparisions



$T(n) = 0 + 1 + 2 + \cdots + n-1$

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

$\Rightarrow T(n) = O(n^2)$

Optimized algorithm:

```
Algo LEADER (A,n)
{
    L ← A[n];
    for i ← n-1 downto 1
    {
        if (A[i] > L)
        {
            print(A[i]);
            L ← A[i];
        }
    }
}
```

TC: O(n)

∴ Time complexity of efficient algorithm = O(n)

(H1/08) Let no of levels = k

∴ height = k

$1+2+3+\cdots+k$ = no of nodes = n.

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

$$k^2+k-2n=0$$

$$k = \frac{-1 \pm \sqrt{1+8n}}{2}$$

$$\Rightarrow k = O(\sqrt{n})$$

(H1/09)

n MultiQueue operations

$\Rightarrow O(1)+O(1)+\cdots+O(1)$ (n times)

$\therefore O(n)$

Assuming Queue is not initially empty.

(H1/10) Out of linear & circular queue circular is more efficient.

and in circular insertion & deletion can be done in O(1) time

∴ opt @

Calculate TC for below algorithm

(H1/11)

```
Algorithm what(n)
{
    if(n==1) return;
    else
    {
        what(n-1);
        B(n)
    }
}
```

a) Assume TC of B(n) = O(1):

TC of recursive algorithm is computed by deriving a recurrence relation.

Let T(n) represent TC of what(n)

$$T(n) = \begin{cases} c, & n=1 \\ a + T(n-1) + b, & n>1 \end{cases}$$

a: time for comparision; b: B(n)

$$T(n) = \begin{cases} c, & n=1 \\ d + T(n-1), & n>1 \end{cases}$$

$$T(n) = T(n-1) + d \quad \text{---} ①$$

$$T(n-1) = T(n-2) + d \quad \text{---} ②$$

② in ① ⇒ $T(n) = T(n-2) + 2d$

$= T(n-3) + 3d$

$\vdots$

$T(n) = T(n-k) + kd$

To terminate the condition

$n-k=1$

$\Rightarrow k=n-1$

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

$\boxed{T(n) = c + dn - d}$

$\therefore T(n) = O(n)$
$T(n) = \Omega(n)$ $\Big\}\ \Theta(n)$

b) Assume TC of $B(n) = O(n)$

$$T(n) = \begin{cases} c, & n=1 \\ a + T(n-1) + n, & n>1 \end{cases}$$

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

$= a + [a + T(n-2) + n] + n$

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

$\vdots$

$= ka + T(n-k) + (n + (n-1) + (n-2) + \cdots (n-(k-1)))$

$n-k=1 \Rightarrow k=n-1$

$T(n) = (n-1)a + T(1) + (2+3+4+\cdots+n)$

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

$= n^2 - n + an - a + c$

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

$\therefore T(n) = O(n^2)$

c) Assume TC of $B(n) = O(1/n)$

$$T(n) = \begin{cases} c, & n=1 \\ a + T(n-1) + \frac{1}{n}, & n>1 \end{cases}$$

$T(n) = a + T(n-1) + \frac{1}{n}$

$= a + \left(a + T(n-2) + \frac{1}{n-1}\right) + \frac{1}{n}$

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

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

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

$\vdots$

$= ka + T(n-k) + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{n-(k-1)}$

$n-k=1 \Rightarrow k = n-1$

$= (n-1)a + T(1) + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{2}$

$= (n-1)a + c + \underbrace{1 + \frac{1}{2} + \cdots + \frac{1}{n}} - 1$

$T(n) = (n-1)a + c + \log n - 1$

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

Note:

→ If $T(n) = a + T(n-1) + \frac{1}{n}$ then $T(n) = O(n)$

→ If $T(n) = T(n-1) + \frac{1}{n}$ then $T(n) = O(\log n)$

} So constant is also matters.

(H/W)

$$T(n) = \begin{cases} c, & n=2 \\ a + T(\sqrt{n}), & n>2 \end{cases}$$

$T(n) = a + T(n^{1/2})$

$= a + a + T(n^{1/4})$

$= 2a + T(n^{1/4})$

$= 2a + T(n^{1/2^2})$

$= T(n^{1/2k}) + ka$

$= T(2) + \left(\frac{1}{2}\log n\right)a$

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

$n^{1/2k} = 2$

$\frac{1}{2k} = \log_n 2$

$2k = \frac{1}{\log_n 2} \Rightarrow 2k = \log_2 n$

$k = \frac{1}{2}\log n$

01/10/20

(#/12)

$$T(n) = \begin{cases} c, & n=2 \\ a + T(\sqrt{n}), & n>2 \end{cases}$$

$T(n) = T(n^{1/2}) + a$

$T(n) = T(n^{1/4}) + 2a$

$T(n) = T[(n^{1/4})^{1/2}] + 3a$

$T(n) = T(n^{1/8}) + 3a$

$T(n) = T(n^{1/2^k}) + ka$

$T(n) = T(2) + a \cdot \log_2 \log_2 n$

$T(n) = c + a \cdot \log_2 \log_2 n$

$\Rightarrow T(n) = O(\log_2 \log_2 n)$

$n^{1/2^k} = 2$

$\frac{1}{2^k} = \log_n 2$

$2^k = \log_2 n$

$\Rightarrow k = \log_2 (\log_2 n)$

Eg: Find TC for below algorithm.

```
Algo A(n)
{
    if(n==2) return
    else
        return A(√n) + n;
}
```

$$T(n) = \begin{cases} c, & n=2 \\ a + T(\sqrt{n}), & n>2 \end{cases}$$

$\Rightarrow T(n) = O(\log_2 \log_2 n)$

H/14

$$T(n) = \begin{cases} c, & n=2 \\ a + 2T(\sqrt{n}), & n>2 \end{cases}$$

$$T(n) = 2T(n^{1/2}) + a$$

$$= 2\left[2 \cdot T(n^{1/2^2}) + a\right] + a$$

$$= 2^2 T(n^{1/2^2}) + 2a + a$$

$$= 2^3 T(n^{1/2^3}) + 2^2 a + 2a + a$$

$$\vdots$$

$$= 2^k T(n^{1/2^k}) + (a + 2a + \cdots + 2^{k-1}a)$$

$n^{1/2^k} = 2$

$\log n^{1/2^k} = \log_2 2$

$\frac{1}{2^k} \log_2 n = 1$

$2^k = \log_2 n$

$k = \log\log n$

~~$= 2^{\log(\log n)} T(2) + a(\log\log n)$~~

~~$T(n) = a \log_2 n + a(\log_2 \log_2 n)$~~

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

$$= 2^k T(n^{1/2^k}) + a(2^k - 1)$$

$$= 2^{\log\log n} T(2) + a(2^{\log\log n} - 1)$$

$$= c \cdot \log n + a \cdot \log n - a$$

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

H/13

$$T(n) \begin{cases} c, & n=1 \\ 2T(n-1) + a, & n>1 \end{cases}$$

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

$$= 2\left[2 \cdot T(n-2) + a\right] + a$$

$$= 2^2 T(n-2) + 2a + a$$

$$= 2^2\left[2 \cdot T(n-3) + a\right] + 2a + a$$

$= 2^3 \cdot T(n-3) + 2^2 a + 2a + a$

$\vdots$

$= 2^k T(n-k) + (2^{k-1} + 2^{k-2} + \cdots + 2 + 1)a$

$= 2^k T(n-k) + (2^k - 1)a$ $\quad n-k = 1 \Rightarrow k = n-1$

$= 2^{n-1} T(1) + (2^{n-1} - 1)a$

$= c \cdot 2^{n-1} + a \cdot 2^{n-1} - a$

$= (a+c)2^{n-1} - a$

$\therefore T(n) = O(2^n)$

(H/15) i) Assume $B(n)$ is ~~O(n)~~ $O(1)$

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

comparision $B(n)$

$T(n) = 2T\left(\frac{n}{2}\right) + a$ $\quad (\because a+1$ is const$)$

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

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

$= 2^2\left[2T\left(\frac{n}{2^3}\right) + a\right] + 2a + a$

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

$\vdots$

$= 2^k T\left(\frac{n}{2^k}\right) + a(2^{k-1} + 2^{k-2} + \cdots + 2 + 1)$

$= 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)a$ $\quad \frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log n$

$= 2^{\log_2 n} T(1) + (2^{\log_2 n} - 1)a$

$= nc + an - a$

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

Assume B(n) is O(n)

(ii) $$T(n) = \begin{cases} c, & n = 1 \\ 2T(n/2) + a + n, & n > 1 \end{cases}$$

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

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

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

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

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} + a\right] + 2n + 2a + a$$

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

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n + (2^2 + 2 + 2^0)a$$

$$\vdots$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn + (2^{k-1} + 2^{k-2} + \cdots + 2 + 1)a$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn + (2^k - 1)a$$

$$= 2^{\log_2 n} T(1) + (\log n)\left(\frac{n}{n}\right) + (2^{\log n} - 1)a$$

$\frac{n}{2^k} = 1$
$\Rightarrow 2^k = n$
$k = \log_2 n$

~~$= \log n\,(c) + a \log n$~~

$$= c \log n + n \log n + an - a$$

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

4/16

$$T(n) \begin{cases} 2 & , n=2 \\ \sqrt{n}\,T(\sqrt{n})+n & , n>2 \end{cases}$$

$$T(n) = n^{1/2}T(n^{1/2}) + n$$

$$= n^{1/2}\left[n^{1/4}T(n^{1/4}) + n^{1/2}\right] + n$$

$$= n^{\frac{1}{2}+\frac{1}{4}}T(n^{1/4}) + n+n$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}}T(n^{1/4}) + 2n$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}}\left[n^{\frac{1}{8}}T(n^{\frac{1}{8}}) + n^{1/4}\right] + 2n$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}}T(n^{1/2^3}) + n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^2}} + 2n$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}}T(n^{1/2^3}) + 3n$$

$$\frac{1}{2}+\frac{1}{2^2}+\cdots+\frac{1}{2^k} = \frac{\frac{1}{2}\left(1-\frac{1}{2^k}\right)}{1/2} =$$

$$= n^{\frac{1}{2}+\frac{1}{2^2}+\cdots+\frac{1}{2^k}}T(n^{1/2^k}) + kn$$

$$n^{1/2^k} = 2$$

$$\frac{1}{2^k} = \log_n 2$$

$$2^k = \log_2 n$$

$$k = \log\log n$$

$$= n^{1-1/2^k}T(n^{1/2^k}) + kn$$

$$= n \cdot n^{-1/2^k}T(n^{1/2^k}) + kn$$

$$= n \cdot \frac{1}{n^{1/2^k}}T(n^{1/2^k}) + kn$$

$$= n\frac{1}{2}T(2) + n\log\log n$$

$$= n + n\log\log n$$

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

(4.17) Best case:

Best case is when less no of steps executed

∴ 2 recursive calls per one call will be called

$\Rightarrow T(n) = 2T(n/2) + a$ ... $n/2^k = 1$

$\Rightarrow$ ... $2^k = n$

$= 2^k T(n/2^k) + (2^k - 1)a$ ... $k = \log n$

$= n\,T(1) + (n-1)a$

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

Worst case:

Worst case is when ~~less~~ more no of recursive calls are executed per each call

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

$= 8^2 T(n/2^2) + 8a + a$

$= 8^3 T(n/2^3) + 8^2 a + 8a + a$

$\Rightarrow 8^k T(n/2^k) + a(8^{k-1} + 8^{k-2} + \cdots + 8a + a)$

$= 8^{\log_2 n} T(1) + a\left(\frac{8^k - 1}{8 - 1}\right)$ ... $\frac{n}{2^k} = 1$, $k = \log n$

$= 8^{\log_2 n} T(1) + \frac{a}{7}\left(8^{\log_2 n} - 1\right)$ ... $8^{\log_2 n} = 2^{3\log_2 n} = n^3$

$= n^3(1) + \frac{a}{7}(n^3 - 1)$

$\therefore T(n) = O(n^3)$

Note:

From previous problem, we can say

if $T(n) = aT(n/2) + c$ then

$$T(n) = O(n^{\log_2 a}) \quad \left.\right\} \; a \neq 1$$

Sily if $T(n) = aT(n/b) + c$ then

$$T(n) = O(n^{\log_b a})$$

if $a=1$, then $T(n) = O(\log_b n)$ (in both cases)

(H/W) 's' is a string of length n

$$T(n) = \begin{cases} c_1 & n=0 \\ 2T(n-1)+a, & n>1 \end{cases}$$

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

$$= 2[2T(n-2)+a] + a$$

$$= 2^2 T(n-2) + (2^2-1)a$$

$$\vdots$$

$$= 2^k T(n-k) + (2^k-1)a \qquad n-k=0 \Rightarrow k=n$$

$$= 2^n T(1) + (2^n-1)a$$

$$= 2^n \cdot c + 2^n \cdot a - a$$

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

Note:

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

2) $T(n) = T(n-1) + n$ ------ $O(n^2)$

3) $T(n) = T(n-1) + \frac{1}{n} + a$ ------ $O(n)$

4) $T(n) = T(n-1) + \frac{1}{n}$ ------ $O(\log n)$

5) $T(n) = 2T(n-1)+c$ ------ $O(2^n)$

6) $T(n) = 2T(n-1)+n$ ------ $O(2^n)$ --- solve it

7) $T(n) = T(\sqrt{n})+c$ ------ $O(\log\log n)$

8) $T(n) = 2T(\sqrt{n})+c$ ------ $O(\log n)$

9) $T(n) = 2T(n/2)+c$ ------ $O(n)$

10) $T(n) = 2T(n/2)+n$ ------ $O(n\log n)$

11) $T(n) = T(n/2)+c$ ------ $O(\log n)$

Analysing running time of program segments with loops:

→ for i ← 1 to n
  c = c+1; ---- repeats n times
  ⇒ T.C : O(n)

→ for i ← 1 to n ---------- n times
  for j ← 1 to n/2 ----- n/2 time
  c = c+1; ---------- n · n/2 time

T.C = $(n)\left(\frac{n}{2}\right) = \frac{n^2}{2} = O(n^2)$

If initial value of c is 0, then value of c in the end will be $\frac{n^2}{2}$

→ for i ← 1 to n ---------- n times
  for j ← 1 to n/4 ------ n/4 times
  for k ← 1 to n ------- 1 time (i.e., n · n/4 times)
  break; ---------- n · n/4 times

∴ TC : $n \cdot \frac{n}{4}$ ⇒ TC: $O(n^2)$

→ for (i=1; i<=n; i++) ——— n

for (j=1; j<=n; j++) ——— n

for (k=n/2; k<=n; k+=n/2) ——— 2

c=c+1;

TC: n*n*2 = $O(n^2)$

value of c in the end = $2n^2$

→ i=1;

while (i<=n)

{

i=i*2;

}

Assume that above loop is repeated k times

$\Rightarrow 2^{k+1} > n$

$k+1 > \log n$

$k > \log n - 1$

$\therefore$ TC: $O(\log n)$

we repeat the loop until below condition is violated

$2^k \leq n$

$k \leq \log_2 n$

$\Rightarrow$ Time ... $O(k) = O(\log n)$

→ i=n;

while(i>0)

{

i=i/2;

}

TC: $O(\log n)$

→ for (i=1; i<=n; i++) ——— n

for(j=1; j<n; j=2*j); ——— log n

c=c+1;

TC: $O(n \log n)$

→ $m = 2^n$

```
for(i=1; i<=n; i++)  ------ n
    for(j=1; j<=m; j=2*j) ------ log2 m = n
        c=c+1;
```

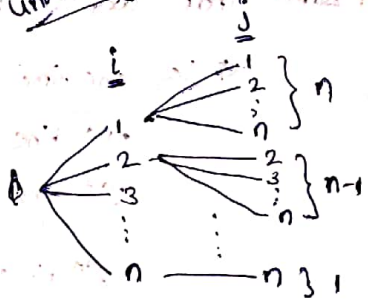
TC: $O(n^2)$

unfolding has to done if loop variables depend on each other

→ for i ← 1 to n ------ n

for j ← i to n ----- n-i

c=c+1;



$TC = 1+2+3+\cdots+n$

$= \frac{n(n+1)}{2} = O(n^2)$

value of $c = \frac{n(n+1)}{2}$.

→ i=n;

while (i>0) ------ $\log_2 n$

{

j=1;

while (j<=n) ------ $\log_2 n$

{

j=2*j; ------ $\therefore (\log_2 n)(\log_2 n)$

}

i=i/2;

}

j=2*j has max frequency

i.e., it is executed $(\log_2 n)^2$ times

$\therefore$ TC: $O(\log^2 n)$

02/10/20

→
```
k=1; i=1;
while (k<=n)
{
    i++;
    k= k+i;
}
```

Assume that loop is repeated x times

i.e., one time for k=1
2nd time for k=3
3rd time for k=6
⋮
x^{th} time for $k = 1+2+\dots+x$

$\Rightarrow 1+2+3+\dots+x \leq n$

$\frac{x^2+x}{2} \leq n$.

~~$x^2+x-2n \geq 0$~~

~~$x^2+x \geq n$~~ $x^2+x \leq 2n$

$\therefore x$ is $O(\sqrt{n})$

(Pg2)

→
```
int fun(int n)
{
    int i,j,p,q=0;
    for(i=1; i<=n; ++i)
    {
        p=0;
        for(j=n; j>1; j=j/2)
            ++p;
        for(k=1; k<p; k=k*2)
            ++q;
    }
    return(q);
}
```

For the function given,
⇒ find order of value returned by q.

$q = O(?)$

j loop runs $\log_2 n$ times
$\Rightarrow p = \log_2 n$

k loop runs $\log_2 p$ times
$\Rightarrow$ i.e., $\log\log n$ times
$\Rightarrow$ for every outer loops iteration q valued is incremented by loglogn.

Outer loop runs 'n' times.
$\therefore$ order of value q is $n \log\log n$

→ int fun(int n)
{
int i, j, p, q=0;
p=0;
for(i=1; i<=n; i++)
{
for(j=n; j>1; j=j/2)
++p;
for(k=1; k<p; k=k*2)
++q;
}
return q;
}

Find order of value returned.

TC of the program on the left side is $O(n \log n)$
↳ Solve it

for every 'j' loop execution p is incremented by logn

i=1	i=2	i=3	----	i=n
p=logn	p=2logn	p=3logn	----	p=nlogn
q=logp = loglogn	q=logp = log(2logn)	q=logp = log(3logn)	---	q=logp = log(nlogn)

$\Rightarrow q = \log(\log n) + \log(2\log n) + \log(3\log n) + \cdots + \log(n\log n)$
$= \log(\log n) + \log 2 + \log\log n + \log 3 + \log\log n + \cdots + \log n + \log\log n$
$= n\log(\log n) + \log n!$ $\Rightarrow$ order $q = O(\log n!) = O(n\log n)$

→ Find TC of below code

```
for (i=1; i<=n; i++) ---------n
{
        j=1; --  --------- 1
        while (j<=n) --------- logn
            j=2*j;
        for(k=1; k<=n; k++)---- n
            c=c+1;
}
```

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

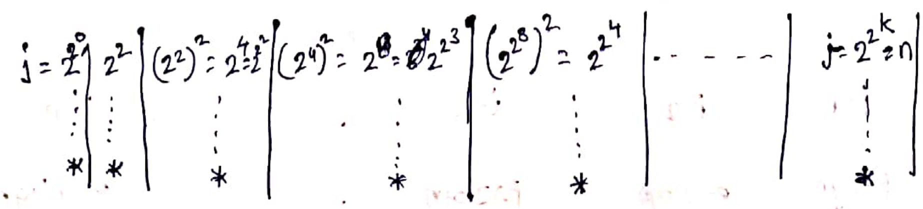
$= n+n\log n+n^2$

$\Rightarrow$ TC: $O(n^2)$

→ ~~$n=2^{2k}$~~ $n=2^{2^k}$

```
for (i=1; i<=n; i++)
{
    j=2;
    while (j<=n)
    {
        j=j*j;
        Print("*");
    }
}
```

for one iteration of outer loop



while loop runs k+1 times

$\Rightarrow$ k+1 "*"'s are printed

$\therefore$ total no of stars printed = $n(k+1)$

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

$n=2^{2^k}$

$2^k=\log n$

$k=\log\log n$

→ A: Array [1....n] of binary

```
f(m)= O(m)
count : integer;
count = 1;
for i=1 to n
{
    if ( A[i]==1) count ++;
    else
    {
        f(count);
        count = 1; // line 1
    }
}
```

(i) What is best case & worst case TC for above program?

Sol:

Case I:

all bits are 1

TC: O(n)

Case II: all bits are 0

TC: O(n)

Case III: A[1 to n-1] = 1 & A[n] = 0

TC: (n-1) + O(n) = O(n)

Case IV: ~~first half~~ A[1 to n/2] = 1 A[n/2+1 to n] = 1

~~time~~: i = 1 to n/2 i = n/2 + 1 i = n/2 + 2 to n

time: n/2 n/2 + 1 O(n/2 - 1)

=> TC: O(n)

Case V:

Alternative 1's & 0's

TC: O(n)

* Here we can observe that irrespective of I/P sequence the TC is $O(n)$.

$\therefore$ Best case: $O(n)$

Worst case: $O(n)$

(ii) Deleting line 1, determine the best & worst case TC.

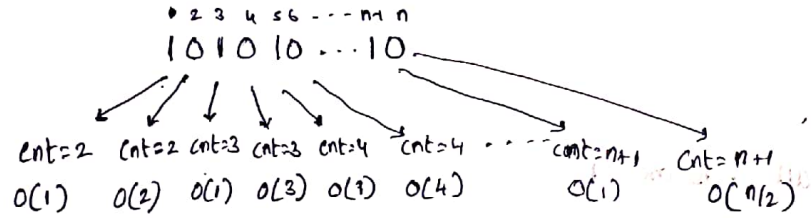
Sol:

If all inputs are 1 then

$O(n)$ --- Best case.

Worst case:

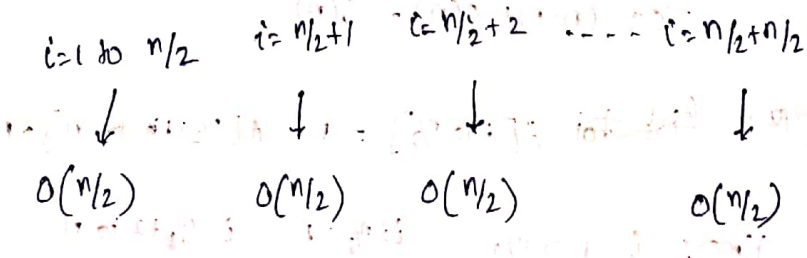
i) Consider case with alternate 1's & 0's



$\Rightarrow O(1)\frac{n}{2} + O(2+3+4+\cdots+n/2)$

$\Rightarrow O(n^2)$ ------ worst case

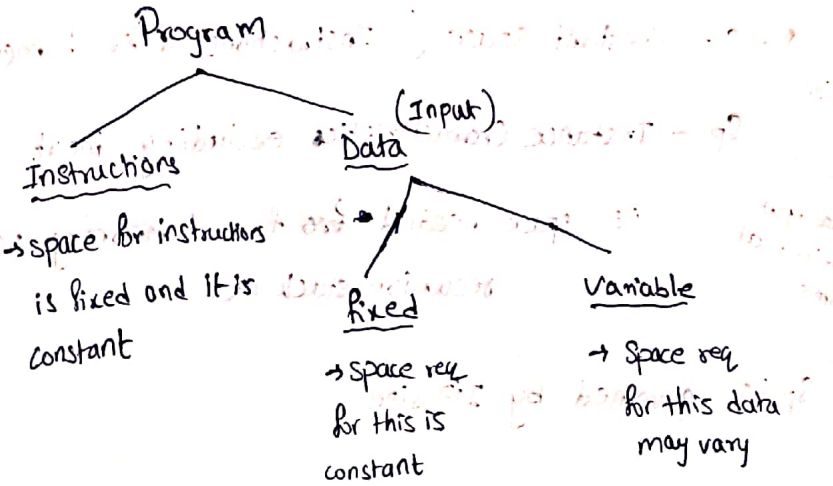
ii) Consider case with first half are 1's and next half are 0's



$\therefore$ time $= O(n/2) + \frac{n}{2} \cdot O(n/2)$

$= O(n^2)$ --- worst case.

Space Complexity



→ Space complexity is not for program instructions or not for data.

Defn:

The space used by an algorithm is defined as the number of memory words needed to carry out the computational steps required to solve an instance of the problem, excluding the space allocated to hold the input. In other words, it is only work space required by the algorithm.

→ All definitions of order of growth & asymptotic bounds pertaining to time complexity equally applies to space complexity.

→ The work space (space complexity) cannot exceed the running time of an algorithm, since writing into each memory cell requires atleast a constant amount of time.

i.e., if $T(n)$ = time complexity &

$S(n)$ = space complexity then

$$S(n) = O(T(n))$$

Space Complexity (P) = C + S_p

C — Constant space (Instructions & fixed memory)
(determined prior to the execution)

S_p — Instance Characteristics excluding input space.
(depends on input and determined at runtime)

i.e., space needed for temporary variable, recursion stack etc.

→ S_p is governed by I/P size.

Eg: void swap (int a, int b)
{
int t;
t = a;
a = b;
b = t;
}

a, b — input
t — additional variable

TC: O(1)

∴ now 't' contributes to space complexity

size(t) : workspace : time complexity

size(t) : 2 bytes = O(1)

∴ S = C + S_p
= 2 + 0
= O(1)

(I)

Eg: Algorithm Sum (A, n)
A[1...n] , n : int
{
int i, sum = 0;
for i ← 1 to n
sum = sum + A[i];
}

TC: O(n)

A & n are inputs
∴ A & n doesn't contribute is Space complexity

variables i, sum contribute to space complexity

S = C + S_p = 4 bytes + 0
= O(1)

Eg: Algo Rsum(A,n)

(II)

```
{
    if(n==1)
        return A[1];
    else
        return (Rsum(A,n-1)+A[n]);
}
```

Since recursion is used we need consider size of stack used.

stack constains activation records.

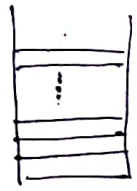
Activation record has formal parameters, local variables, return address etc.

∴ size of activation ~~vector~~ record is constant.

Here we need O(n) activation records.

∴ amount of space used in stack

$= O(n) * C = O(n)$



workspace of this algorithm

∴ Space complexity = C + Sp

$= 0 + O(n)$

$= O(n)$

Time complexity = O(n)

Observation:

Both algo I & algo II does the same job of computing sum of array elements.

Algo I < TC: O(n), SC: O(1)

Algo II < TC: O(n), SC: O(n)

∴ Algo I is space efficient.

Algo II is space inefficient.

General rule:

An algorithm is said to be space efficient, if its space requirement is O(1) or atmost O(log n) in case of recursive algorithm.

Eg:
```
Algo Test (A,n)
A[1..n, 1..n] , n: int ← input
{
    integer B[1..n], i;
    for i←1 to n
        B[i] = A[i,i]
}
```

TC: O(n)

B, i are considered workspace.

$S = C + S_p$

$= \text{size}(i) + \text{size}(B)$

$= k + O(n)$

$\therefore SC = O(n)$

Eg:
```
Algo A(n)
{
    if (n=1) return;
    else
    {
        A(n/2);
    }
}
```

TC: O(log n)

Space complexity is calculated by amount of space used by stack.

⇒ no of recursive calls = log n

⇒ * space = (log n)(size of activation record)

$S = C + S_p = 0 + \log n * c = O(\log n)$

Eg: A(n)

```
{
    if(n=2) return;
    else
        return (A(√n));
}
```

Space complexity in this case is order of no of rec. calls.

let S(n) denote no of recursive calls.

$$\Rightarrow S(n) = 1 + S(\sqrt{n})$$
$$= 1 + S(n^{1/2})$$
$$= 1 + 1 + S(n^{1/2^2})$$
$$= 2 + S(n^{1/2^2})$$
$$\vdots$$
$$= k + S(n^{1/2^k})$$

$n^{\frac{1}{2^k}} \leq 2$

If k is no of recursive calls.

then $n^{1/2^k} = 2$

$\Rightarrow 2^k = \log_2 n$

$k = \log \log n$

$\therefore$ Space Complexity $= O(\log \log n)$

Eg: recur(n)

```
{
    if(n=1) return;
    else
    {
        recur(n/2);
        recur(n/2);
        B(n);
    }
}
```

let S(n) denote no of recursive calls.

~~S(n) = order of no of recursive calls~~

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

$= 2[2 \cdot S(n/2^2) + 1] + 1$

$= 2^2 S(n/2^2) + 2^2 - 1$

$= k\,2^k S(n/2^k) + 2^k - 1$

~~If k is no of recursive calls then~~

~~$n/2^k = 1$~~

$\frac{n}{2^k} = 1$

$\Rightarrow k = \log_2 n$

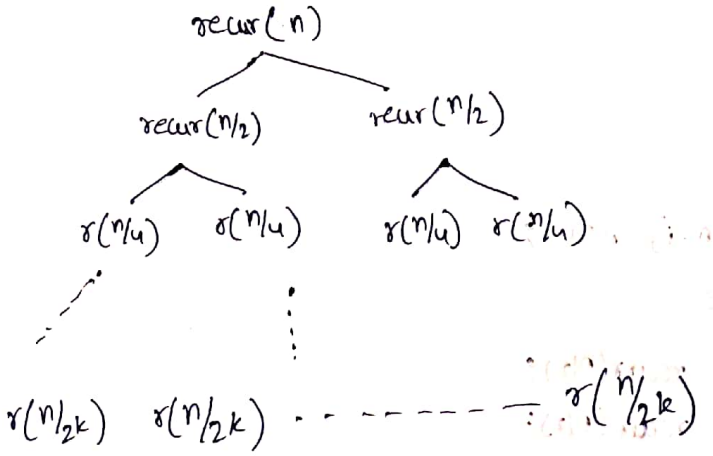
$S(n) = 2^{\log_2 n} S(1) + 2^k - 1$

$= n + n - 1$

$= 2n - 1$

$\therefore$ no of recursive calls $= 2n - 1$

However observing the recursion stack, the depth of recursion stack is only k..



$\therefore$ space complexity = depth of recursion stack

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

~~Design Strategies~~

~~Divide & Conquer Tech~~

→ Find value of c & time complexity for below code segment.

```
for i←1 to n-1
    for j←i+1 to n
        for k←1 to j
            c=c+1;
```

TC: $O(n^3)$

$$c = \frac{(n-1)(n)(n+1)}{3}$$

→ Find value of c & TC for below code segment

```
for(i=1; i<=n; i++)
    for(j=1; j<=i*i; j++)
        if((j%i)==0)
            for(k=1; k<=j; k++)
                c=c+1;
```

TC: $O(n^4)$

$$c = \frac{n(n+1)(3n^2+7n+2)}{24}$$

04/10/20

Divide & Conquer

→ This strategy is used when the problem (its input) is large / complex.

→ So we divide it into subproblems until the subproblem is small.

→ A subproblem is said to be small, if it can be solved in one / two fundamental (basic) operations.

→ Combine (conquer) the results of smaller problems to get the result of the original problem.

This step of combining is optional: It is done only if it is needed.

Eg: quicksort uses only dividing but doesn't perform combining.

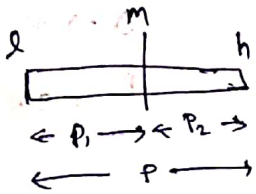
→ Division can be done into 2 or more problems and each subproblem can be of different sizes.

Control Abstraction: → A procedure whose flow of control is clear but the inner details are abstracted.

```
Algorithm DandC(P)
{
    if (SMALL(P)) then
        return(S(P)); // S(P) is soln of smaller problem.
    else
    {
        Divide P into smaller problems P1, P2, ... Pk (k>=1) // Mandatory
        Apply DandC to each subproblems
        return(COMBINE(DandC(P1), ... DandC(Pk))); //optional
    }
}
```

→ The below ~~pro~~ procedure is a general procedure for dividing problem into 2 subproblems.

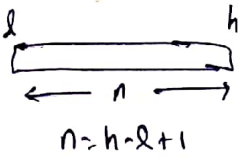
```
Procedure DandC(P,l,h)
{
    if (SMALL(l,h))
        return G(P,l,h);
    else
    {
        m ← Divide(l,h);
        Soln1 ← DandC(P,l,m);
        Soln2 ← DandC(P,m+1,h);
        return COMBINE(soln1, soln2);
    }
}
```



→ The TC of D&C problems is expressed in the form of DandC recurrence equations.

let T(n) represent time complexity of DandC(n) ← size of the problem (i/p)

Consider we divide the problem into 2 subproblems



$$T(n) \begin{cases} f(n) & , n \text{ is small} \\ 2T(n/2)+g(n) & , n \text{ is large} \end{cases}$$

f(n) → time for SMALL & function's (in most of the cases this is O(1))

g(n) → time for SMALL, dividing & combining

The general form of DandC Recurrence: (for symmetric DandC) i.e., equal size subproblems.

$$T(n) \begin{cases} f(n) & , n \text{ is small} \\ a \cdot T(n/b) + g(n) & , n \text{ is large} \end{cases}$$

$a \rightarrow$ no of subproblems $\therefore a \geq 1$

$\frac{n}{b} \rightarrow$ size of each subproblem. $\therefore b > 1$ (∵ problem size has to reduce when it is divided)

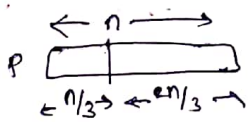
$g(n) \rightarrow$ time for SMALL divide, combine

$\therefore g(n) > 0$

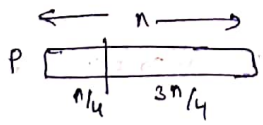
$\therefore a \geq 1 ; b > 1 ; g(n) > 0$

Variation of DandC reccurence: (asymmetric)

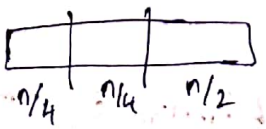
↳ unequal sized subproblems



$\Rightarrow T(n) = T(n/3) + T(2n/3) + g(n)$



$\Rightarrow T(n) = T(n/4) + T(3n/4) + g(n)$



$\Rightarrow T(n) = 2 \cdot T(n/4) + T(n/2) + g(n)$

~~Sum of sizes of subproblems~~

$=$ ~~size of original problem.~~

→ sometimes its possible that sum of size of subproblems not equal to size of original ~~program~~ problem.

Eg: $T(n) = T(n/2) + T(n/4) + g(n)$

Eg: $T(n) = T(n/2) + g(n)$

general form: $T(n) = T(\alpha n) + T(\beta n) + T(\gamma n) + \cdots + g(n)$

where $0 < \alpha, \beta, \gamma, \ldots < 1$

Note:

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

The above eq is valid even ~~thor~~ though size of subproblem is $n/2$ and no of subproblems are 8.

Reason: It could the case the a subproblem is solved multiple times

(or)

It is possible when the input data ~~struc~~ structure is non-linear

For example, for input being $n \times n$ matrix & dividing ~~it do~~ into 4 subproblems where each is $\frac{n}{2} \times \frac{n}{2}$ matrix.

i) Max-Min:

The problem is finding minimum and maximum element in an array of size 'n' elements.

Naive approach (Non-DC):

```
Procedure Non-DC (A,n, fmax, fmin)
{
    1. fmax ← fmin ← A[1];
    2. for i ← 2 to n
       {
           if (A[i] > fmax)
               fmax ← A[i]
           if (A[i] < fmin) // line 1
               fmin ← A[i];
       }
}
```

TC: O(n)

→ Here the metric based on which we measure the time is comparision. (Also we consider comparisions involving array element but we ignore comparision of index in for loop)

Here total no of comparisions = 2(n-1)

↳ (best, avg, worst cases)

→ Here by replacing line 1 with "else", for certain elements we can reduce no of comparisions.

best case ⟶ n-1 (increasing order)

worst case ⟶ 2(n-1) (decreasing order)

Avg. case → on an ~~avg~~ average assum that the first comparision fails for 1/2 of given elements.

It is given that comparision fails for 1/2 elements

⇒ for $n/2$ elements comparision fails

⇒ $n/2$ comparisions out of n-1 comparisions fails

In this case no of comparisions

req = $n/2 * 2 = n$

⇒ for $n-1-n/2$ comparision are success

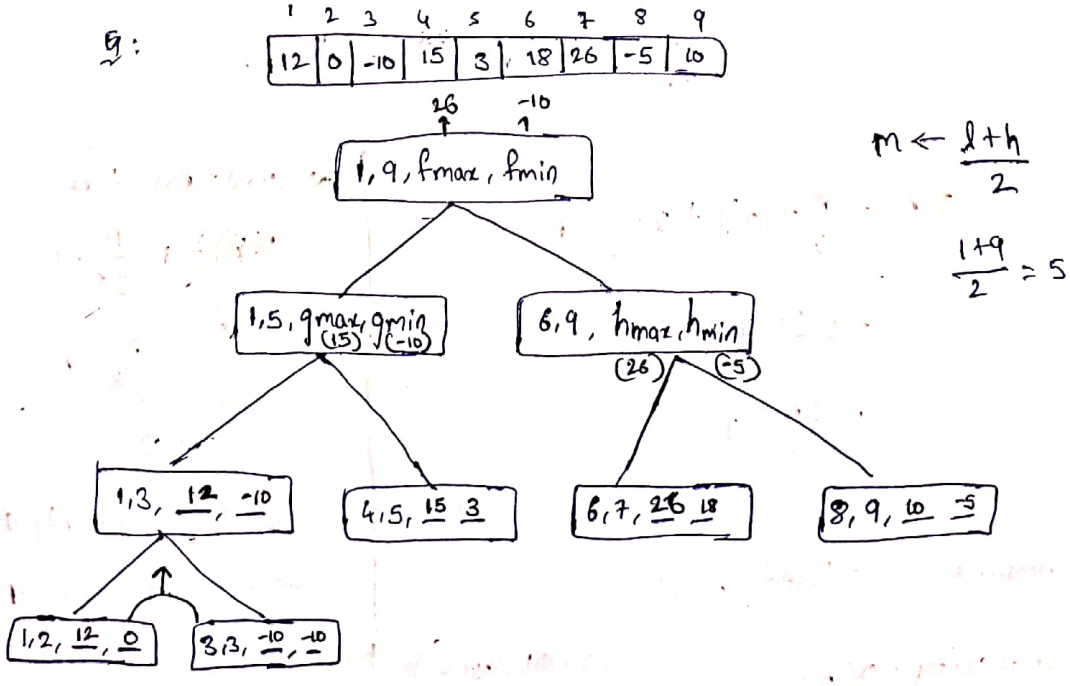
⇒ total comparisions → $n + n-1-n/2$

$$\frac{3n}{2} - 1$$

D and C approach:

If n=1 or n=2 then we say the problem is small. Since in that case no of comparisions will be either 0 (or) 1 respectively.

Eg:



$m \leftarrow \frac{l+h}{2}$

$\frac{1+9}{2} = 5$

∴ every level of combine requires 2 comparisions

Let $T(n)$ represent the no of comparisions made in DandC Maximum(n)

$$T(n) = \begin{cases} 0 & , n=1 \\ 1 & , n=2 \\ 2T(n/2)+2 & , n>2 \end{cases}$$

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

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

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

$$= 2^3 T(n/2^3) + 2^3+2^2+2$$

$$= 2^k T(n/2^k) + 2^k + 2^{k-1} + \cdots + 2$$

$$= 2^k T(n/2^k) + 2\frac{(2^k-1)}{2-1}$$

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

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

terminating condition

$$\frac{n}{2^k} = 2 \Rightarrow n = 2^{k+1} \Rightarrow 2^k = n/2$$

$$T(n) = \frac{n}{2} T(2) + 2 \cdot \frac{n}{2} - 2$$

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

$$= \frac{3n}{2} - 2$$

→ no of comparision in DandC $= \frac{3n}{2} - 2$ ↳ (All cases)

→ Time complexity : $O(n)$

we should not take

$T(n/2^k) = 1$ $\frac{n}{2^k} = 1$

$\therefore \frac{n}{2^k} = 1$

$\Rightarrow \frac{n}{2^{k-1}} = 2$

which was already a terminating condition

In most of the cases it terminates with $n=2$.

Performance Comparision

I/P class	Non-DC	DC	
Inc order	$(n-1)$	$\frac{3n}{2}-2$	} naive approach is better
Dec order	$2(n-1)$	$\frac{3n}{2}-2$	} D and C is better here
random order	$\frac{3n}{2}-1$	$\frac{3n}{2}-2$	
TC	$O(n)$	$O(n)$	
Space-C	$O(1)$	$O(\log n)$	

Space Complexity

Non-DC: $O(1)$ --- memory for i

DC:

Here we need consider the space for stack.

The stack grows upto a depth of $\log n$

$\therefore O(\log n)$

Merge Sort

	1	2	3	4	5	6	7	8	9	10
A:	310	285	179	652	351	423	861	254	450	520

Merging Process (conquer)

→ The below is about 2-way merging i.e., we merge 2 sorted lists.

Problem stmt: Given two sorted lists L_1 & L_2, it is req to merge them into single sorted list.

Lists

L_1: ⟨4, 5, 8, 12, 20⟩ L_2: ⟨3, 7, 11, 18, 25, 30⟩

Assume size(L_1) ≤ size(L_2)

let combined list be L.

let i, j be indices pointing to the start of the list.

L_1: ⟨4, 5, 8, 12, 20⟩ L_2: ⟨3, 7, 11, 18, 25, 30⟩

↑ i ↑ j

$L_1[i] > L_2[j]$

$\therefore L[] = L_2[j]$; j++

L_1: ⟨4, 5, 8, 12, 20⟩ L_2: ⟨3, 7, 11, 18, 25, 30⟩

↑ i ↑ j

L: ⟨3, . . .⟩

$L_1[i] < L_2[j]$

$\therefore L[2] = L_1[i]$; i++

L_1: ⟨4, 5, 8, 12, 20⟩ L_2: ⟨3, 7, 11, 18, 25, 30⟩

↑ i ↑ j

L: ⟨3, 4, . . .⟩

Continuing this process, we obtain

L: ⟨3, 4, 5, 7, 8, 11⟩

one of the two array will be completely traversed.

Now append the elements of array that is not completely traversed to the end of L.

Finally we obtain

L: ⟨3, 4, 5, 7, 8, 11, 12, 20, 25, 30⟩

~~Let i be size of L_1 & j be size of L_2~~

Analysis of Merge process:

Finding no of comparisions:

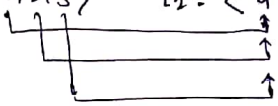
→ Here we consider comparision only b/w array elements

Let n be size(L_1) & m be size(L_2)

Case I: (best case)

Every element of L_1 < starting element of L_2

Eg: L_1: ⟨1,2,3⟩ L_2: ⟨4,5,6,7,8⟩



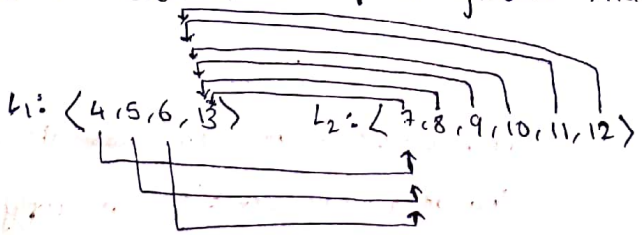
∴ no of comparision req = 3 = size of L_1

Case II: (worst case)

~~every L_1 except last element of~~

n-1 of list L_1 are less than 1st element of L_2.

and nth element of L_1 is greater than all elements of L_2



∴ no of comparisions = (n-1) + m

= n+m-1

∴ ~~Time complexity~~

no of comparisions in best case = min(n,m)

no of comparisions in worst case = n+m-1

∴ TC: O(n+m)

Now we see how mergesort works

	1	2	3	4	5	6	7	8	9	10
A:	310	285	179	652	351	423	861	254	450	520

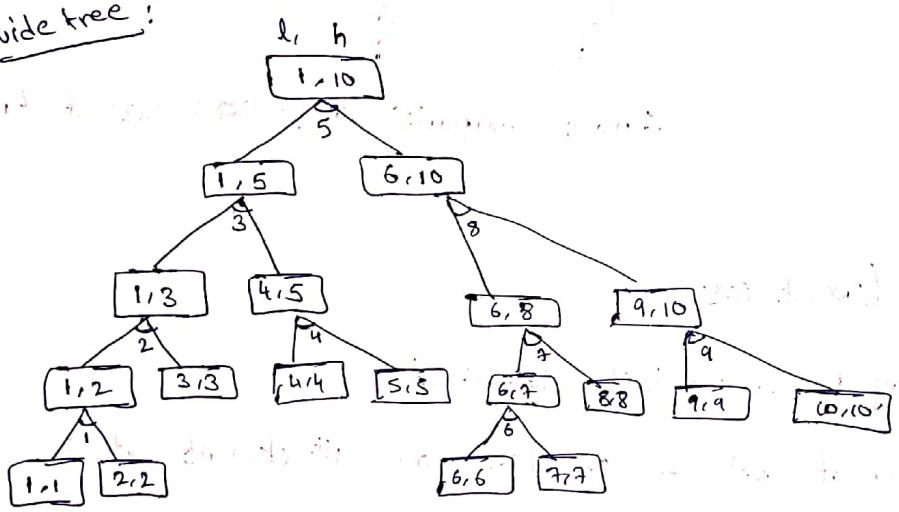
Merge sort uses a temporary array which is of the same size of A. Let this temporary array be B.

B:

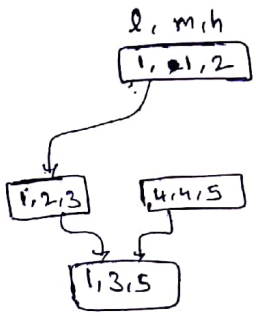
→ we say the problem is small when there is 1 element in the list.

Division: mid ← $\frac{l+h}{2}$

Divide tree:



Conquer tree:



while combining we store ~~the~~ the elements in B. And after every combine, we copy those combined elements from B to A.

B[1]=285 B[2]=310

Now copy them into A

A: <285, 310, 179, 652, ...>

Combine(1,2,3):

~~B[1]=179~~

<285, 310> <179>

B: <179, 285, 310 ....>

A: <179, 285, 310, ...>

Combine (4,4,5) :

B: ~~<351> <652>~~ <17>

4 5

<652> <351> ⇒ <351,652>

B: <179, 285, 310, 351, 652, ...>

~~A:~~ A: <179, 285, 310, 351, 652, ...>

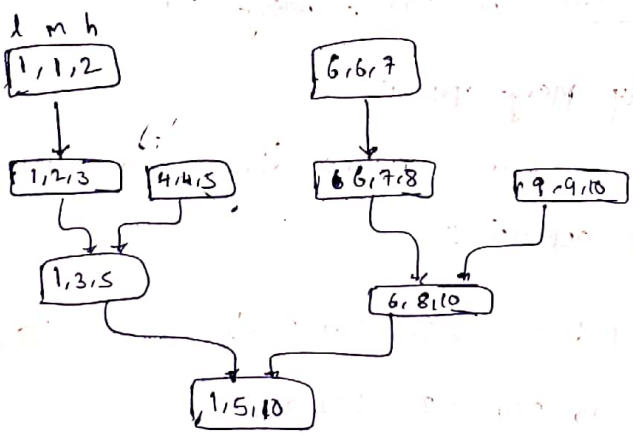
Combine (1,3,5):

<179, 285, 310> <351, 652>

B: <179, 285, 310, 351, 652, ...>

A: <179, 285, 310, 351, 652, ...>

Combine tree:



Performance Analysis:

Let T(n) represent time complexity of DandC-MS(n)

$$T(n) = \begin{cases} c, & n=1 \\ 2T(n/2) + bn, & n>1 \quad (b>0) \end{cases}$$

for the process of merging, checking if the problem is small etc.

bn in best case → $n/2$

worst case → $n/2 + n/2 - 1 = n-1$

we can also have

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

∴ time complexity of merge sort

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

Space Complexity:

$S = c + S_p$

c is some constants and

$S_p \rightarrow$ sum of stack size + additional array B

$\rightarrow O(\log n) + O(n)$

$\therefore S = O(n)$

∴ MergeSort is not space efficient.

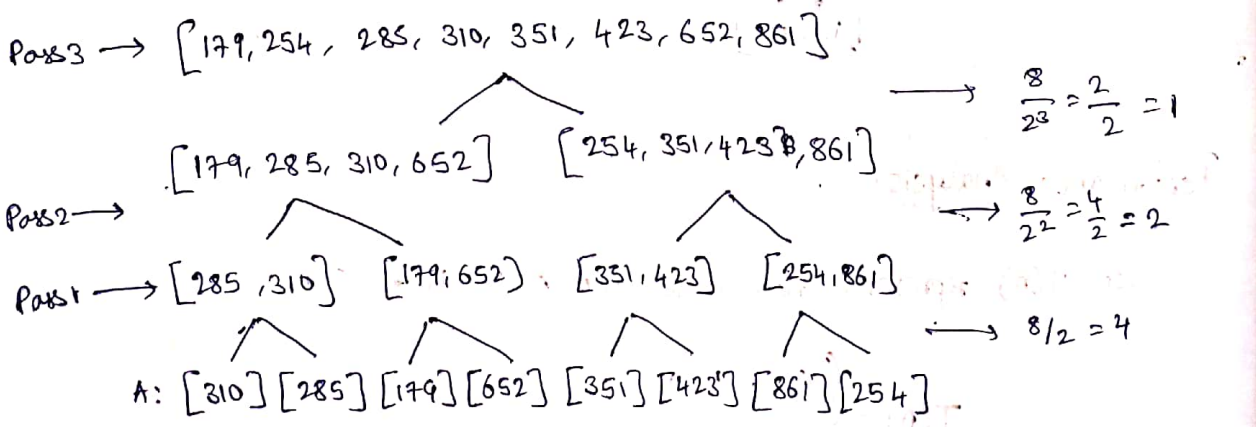
MergeSort is not inplace sorting.

2-way / bottom-up MergeSort (variation of merge sort)

→ This sorting uses only merge operations.

→ Here we consider array is already divided and hence we perform only merge operations

A: ⟨310, 285, 179, 652, 351, 423, 861, 254⟩



Time Complexity : $O(n \log n)$

↳ ∵ log n passes & every pass has comparisions less than n.

#185 Time complexity of merge sort = $n \log n$

$n = 64$
- Aposteriori time = 30s
- Apriori time = $64 \log_2 64 = 64 \times 6$ units

$\Rightarrow$ 64×6 units $\longrightarrow$ 30s

1 units $\longrightarrow \frac{30}{64 \times 6}$ s

$\Rightarrow$ 360s ~~$\longrightarrow$ 64~~

$\Rightarrow$ 1s $\longrightarrow \frac{64 \times 4}{30}$

6 min $\Longrightarrow$ 360 s $\longrightarrow \frac{64 \times 4 \times 360}{30} = 4608$

Let n be no of elements sorted in 360s

then $n \log n = 4608$

let $n = 2^k$

$\Rightarrow 2^k \cdot k = 4608$

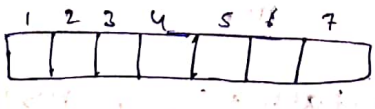
$k = 9$ satisfies the equation

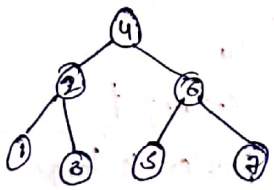
$\therefore n = 2^k = 512$ elements.

Binary Search:

→ Binary search requires that the list is sorted

Consider a list of size 7



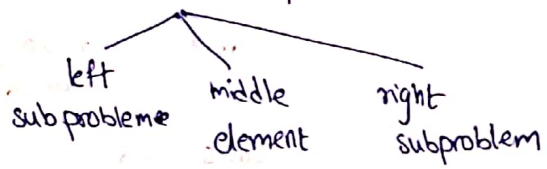


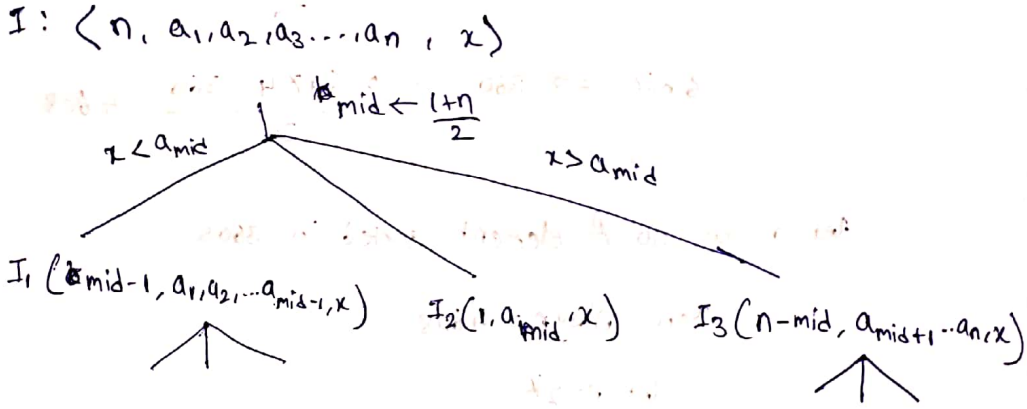
Search path ~~is~~ using binary search corresponds to one of the path from root to leaf in the above tree.

DandC-BS(n,x)

→ we say a problem is small when there is only one element

→ In binary search, ~~the binary~~ when problem is large (n>1), the problem is divide into 3 subproblems





Out of these 3 subproblems, we solve only 2

i.e., I_2 and/or (I_1 (or) I_3)

→ Binary search ~~doesn't~~ doesn't perform combine operation.

Performance Analysis:

time Complexity: Let T(n) represent time complexity of DandC-BS(n)

$$T(n) = \begin{cases} c, & n=1 \\ T(n/2) + a + b + c, & n>1 \end{cases}$$

solving I_2 — checking if problem is large — time for divide

(or)

$$T(n) = T(n/2) + k, \quad n>1, k>0$$

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

→ Best case TC of binary search = O(1)

→ Space Complexity:

$S = C + Sp$

$= c_1 +$ size of stack used

$= c_1 + \log n$

$S = O(\log n)$

However for non-recursive implementation of binary search

Space Complexity, $S = O(1)$

Linear Search:

→ Best Case $TC = O(1)$

→ worst case & avg case $TC = O(n)$

→ The avg no of key comparisions involved in a successful sequential search with n-elements is $\frac{n+1}{2}$

i.e., $\frac{1+2+3+\cdots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$

Matrix Multiplication [square matrices]

→ $C = A \times B$

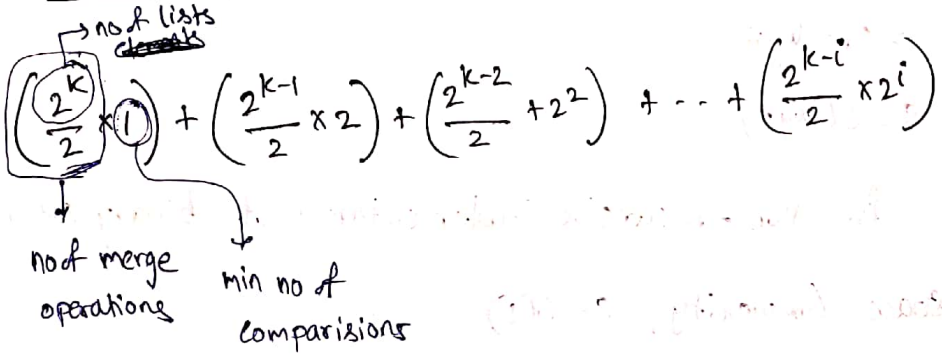
naive approach:

```
for i ← 1 to n
    for j ← 1 to n
    {
        C[i,j] = 0;
        for k ← 1 to n
    }       C[i,j] += A[i,k] * B[k,j];
```

(a) Calculate the total no of minimum & maximum no of element comparisions involved in 2-way / bottom up merge-sort with $n=2^k$ elements.

Sol:

Minimum Comparision:



At the termination, no of merge operations will be one.

$\Rightarrow$ no of lists $=2$

i.e., $2^{k-i}=2$

$\Rightarrow k-i=1$

$\Rightarrow i=k-1$

$$\Rightarrow \frac{2^k+2^k+\cdots+2^k \text{ (k times)}}{2} = \frac{k\cdot 2^k}{2} = k\cdot 2^{k-1}$$

$\therefore$ Min no of comparisions $= \frac{n\log n}{2}$

Maximum Comparisions:

$$\left(\frac{2^k}{2}\times 1\right)+\left[\frac{2^{k-1}}{2}\times(2^2-1)\right]+\left[\frac{2^{k-2}}{2}+(2^3-1)\right]+\cdots+\left[\frac{2^{k-i}}{2}\times(2^{i+1}-1)\right]$$

$$\Rightarrow\left[\frac{2^k}{2}\times 2^1-1\right]+\left[\frac{2^{k-1}}{2}\times(2^2-1)\right]+\cdots+\left[\frac{2^1}{2}\times(2^k-1)\right]$$

$2^{k-i}=2$

$k-i=1$

$i=k-1$

$$\frac{1}{2}\left[\underbrace{(2^{k+1}+2^{k+1}+\ldots+2^{k+1})}_{k \text{ times}} - (2^k + 2^{k-1} + \ldots + 2)\right]$$

$$= \frac{1}{2}\left[k \cdot 2^{k+1} - 2(2^k - 1)\right] = 2^k(k-1) + 1$$

$$= n\log n - n + 1$$

$\therefore$ Max no of comparision $= n\log n - n + 1$

05/10/20

Consider

$$A = \left[\begin{array}{cc|cc} 4 & 5 & 6 & 7 \\ 8 & 9 & 1 & 2 \\ \hline 3 & 9 & 1 & 5 \\ 6 & 7 & 2 & 3 \end{array}\right] \quad \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix}$$

$$\times \; B = \left[\begin{array}{cc|cc} 9 & 6 & 5 & 2 \\ 4 & 1 & 3 & 9 \\ \hline 8 & 6 & 3 & 2 \\ 4 & 1 & 3 & 5 \end{array}\right] \quad \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}$$

$$= \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] \quad C$$

Here

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

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$

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

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

$A, B, C \rightarrow n \times n$

$A_{ij}, B_{ij}, C_{ij} \rightarrow \frac{n}{2} \times \frac{n}{2}$

→ Here we say problem is small when the matrix is 2×2 or 1×1

$$T(n) = \begin{cases} c & , n <= 2 \quad (c > 0) \\ 8T(n/2) + bn^2 & , n > 2 \end{cases}$$

8 multiplications where each multiplication is blw two $n/2 \times n/2$ matrices

bn^2 → time for addition

$$T(n) = 8T(n/2) + bn^2$$

$$= 8\left[8T(n/2^2) + b\left(\frac{n}{2}\right)^2\right] + bn^2$$

$$= 8^2 T(n/2^2) + 8\frac{bn^2}{4} + bn^2$$

$$= 8^2 T(n/2^2) + \frac{8^2}{7} bn^2$$

$$= 8^k T(n/2^k) + \frac{8^k - 1}{7} bn^2$$

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow 8^k = n^3$$

$$= n^3 T(1) + n^3 - 1$$

$$= 8^2\left[8T(n/2^3) + b(n/2^2)^2\right] \quad \because 2bn^2 + bn^2$$

$$= 8^3 T(n/2^3) + 2^2 bn^2 + 2bn^2 + bn^2$$

$$= 8^3 T(n/2^3) + (2^3 - 1)bn^2$$

$$= 8^k T(n/2^k) + (2^k - 1)bn^2$$

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow 8^k = n^3$$

$$= n^3 T(1) + (n-1)bn^2$$

$$= cn^3 + bn^3 - bn^2$$

$\therefore T(n) = O(n^3)$

Space Complexity $= O(\log n)$

$\therefore$ So this approach of matrix multiplication didn't reduce the time complexity.

Strassen's Matrix Multiplication

→ The idea behind strassen's matrix multiplication is reducing no of multiplication by increasing addition & subtraction operations.

Let $A, B, C \longrightarrow n \times n$

$A_{ij}, B_{ij}, C_{ij} \longrightarrow \frac{n}{2} \times \frac{n}{2}$

$P, Q, R, S, T, U, V \longrightarrow \frac{n}{2} \times \frac{n}{2}$

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) \cdot B_{11}$$

$$R = (A_{11}) \cdot (B_{12} - B_{22})$$

$$S = (A_{22}) \cdot (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) \cdot (B_{22})$$

$$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$

$$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

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

Here to compute $C_{11}, C_{12}, C_{21}, C_{22}$ we perform.

7 multiplications & 18 add/sub

$$T(n) = \begin{cases} c, & n <= 2 \quad (c > 0) \\ 7T(n/2) + bn^2, & n > 2 \end{cases}$$

$$T(n) = 7T(n/2) + bn^2$$

$$= 7\left(7T(n/2^2) + b(n/2)^2\right) + bn^2$$

$$= 7^2 T(n/2^2) + 7b(n/2)^2 + bn^2$$

$$= 7^2\left[7T(n/2^3) + b(n/2^2)^2\right] + 7b(n/2)^2 + bn^2$$

$$= 7^3 T(n/2^3) + 7^2(n/2^2)^2 + 7^1 b(n/2)^2 + bn^2$$

$$\vdots$$

$$= 7^k T(n/2^k) + 7^{k-1}(n/2^{k-1})^2 + 7^{k-2} b(n/2^{k-2})^2 + \cdots + 7^0 b(n/2^0)^2$$

$$= 7^k T(n/2^k) + n^2\left(\left(\frac{7}{2^2}\right)^{k-1} + \left(\frac{7}{2^2}\right)^{k-2} + \cdots + 1\right)$$

$$= 7^k T(n/2^k) + n^2\left[\frac{(7/4)^k - 1}{\frac{7}{4} - 1}\right]$$

$n/2^k = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$

$\Rightarrow 7^k = 7^{\log_2 n}$

$$= 7^{\log_2 n} T(1) + \frac{4}{3} n^2\left[(7/4)^{\log_2 n} - 1\right]$$

$$= n^{\log_2 7} T(1) + \frac{4}{3} n^2\left[n^{\log_2 7 - \log_2 4} - 1\right]$$

$$= n^{\log_2 7} \cdot c + \frac{4}{3} n^2\left[n^{\log_2 7 - 2} - 1\right]$$

$$= c \cdot n^{\log_2 7} + \frac{4}{3} n^{\log_2 7} - 1$$

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

$\Rightarrow T(n) = O(n^{2.81})$

Space Complexity, $S = O(n^2)$

↳ space for matrices P, Q, R, S, T, U, V

Quick Sort [Partition - Exchange Sort]

→ Quick sort is based on the principle of partitioning (pivoting)

1	2	3	4	5	6	7	8	9
65	70	75	80	85	60	55	50	45

A: [65 70 75 80 85 60 55 50 45]

Partitioning:

(i) Select a pivot (partitioning element)

(ii) Pivot the partitioning element at its correct place (fix)

(iii) All elements that are less than pivot has to be on left of the pivot.

(iv) All elements that are greater than pivot has to on right of the pivot.

```
int PARTITION (A, l, h)
{
    i ← l;  j ← h; // h is n+1
    v ← A[l]; // v contains partitioning element.
    loop
    {
        loop i ← i+1 until (A[i] ≥ v);
        loop j ← j-1 until (A[j] ≤ v);
```

```
if (i<j)
    SWAP(A[i], A[j]);
else
    break;
} until (false);
A[l] = A[j]; A[j] = v; return j;
                          ↳ position of pivot
} until (false);
```

(l) (h)

	1	2	3	4	5	6	7	8	9	10
A:	65	70	75	80	85	60	55	50	45	∞
	↑ i									↑ j

↳ Having infinity here helps when pivot is the largest element. Sometimes, if we don't use ∞ the program may go to infinite loop

$v \leftarrow A[l]$

$v \leftarrow 65$

$i \leftarrow i+1 \Rightarrow i = 2$

$A[i] \geq v \Rightarrow A[2] \geq 65$

↓

true ∴ exit loop

$j \leftarrow j-1 \Rightarrow j = 9$

$A[j] \leq v \Rightarrow A[9] \leq 65$

↓

true ∴ exit loop

if (i<j) .. true

swap(A[i], A[j])

∴ swap(A[2], A[9])

	1	2	3	4	5	6	7	8	9	10
A:	65	45	75	80	85	60	55	50	70	
		↑ i							↑ j	

$i \leftarrow i+1 \Rightarrow i = 3$

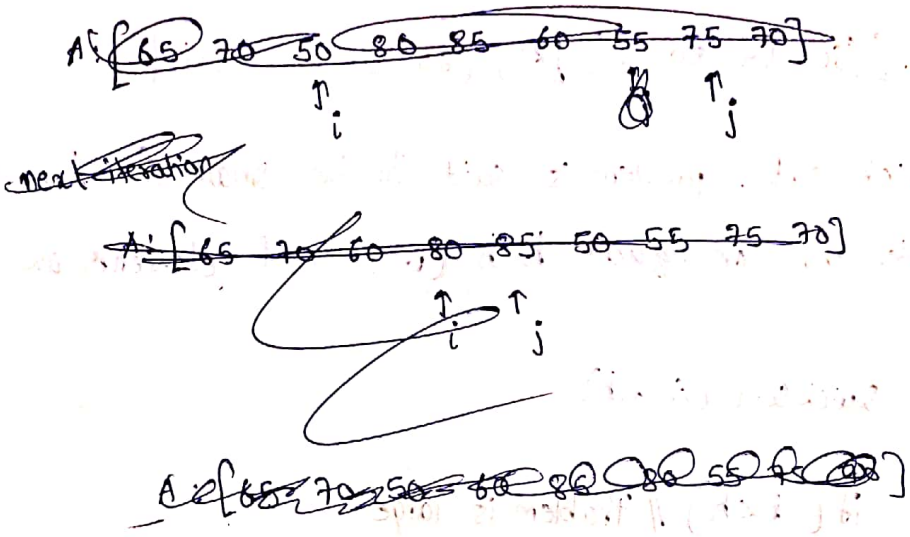
$A[3] \geq 65$ ✓

$j \leftarrow j-1 \Rightarrow j = 8$

$A[8] \leq 65$ ✓

if (i<j) .. true

∴ SWAP(A[3], A[8])



A: [65 45 50 80 85 60 55 75 70]

i j

Next iteration:

A: [65 45 50 55 85 60 80 75 70]

i j

next iteration:

A: [65 45 50 55 60 85 80 75 70]

i j

i 5 j 6

$i \leftarrow i+1 \Rightarrow i$ 6

$A[i] \geq 65$ ✓

$j \leftarrow j-1$ j 5

$A[j] \leq 65$

$60 \leq 65$ ✓

if $(i < j)$ ... false

Now we need to put partitioning element in its position.

This position is present in j.

A: [(60 45 50 55) 65 (85 80 75 70)]

→ In quick sort, problem is said to be small if low is greater than or equal to h (i.e., no of elements are 1)

```
void QuickSort (A, l, h)
{
    if( l < h) // Problem is large
    {
        m ← PARTITION (A, l, h);
        QuickSort (A, l, m-1);
        QuickSort (A, m+1, h);
    }
}
```

Analysis:

TC of PARTITION:

→ In both loops which are inside loop-until loop, we perform a total 'n+1' comparision blw array elements and v(pivot).

∴ TC of partition = O(n)

TC of quicksort:

A: <a1, a2, a3, ... an>

Partitioning : O(n)

<(...) a1 (...)> <(...) a1> <a1 (...)>

(I) (II) (III)

Case I:

$$T(n) = \begin{cases} c, & n=1 \\ 2T(n/2)+O(n), & n>1 \end{cases}$$

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

Case II & III:

$$T(n) = \begin{cases} c, & n=1 \\ T(n-1)+n, & n>1 \end{cases}$$

$T(n) = O(n^2)$

Note:

→ avg case TC of Quick sort $= O(n \log n)$

∴ for QS,

best case TC = avg case TC = $O(n \log n)$

worst case TC = $O(n^2)$

→ worst case of TC happens when elements are already sorted (i.e., either increasing or decreasing)

Space Complexity:

best case space complexity → $O(\log n)$

worst case space complexity → $O(n)$

↳ sorted list

H/76

Partitioning has to start choosing pivot element from one of the end elements of array.

So we swap central element with any other end elements of array & finally the worst case TC is $O(n^2)$

H/77

time to find median $O(n)$

time to partition w.r.to median $O(n)$

$\therefore$ total time for partitioning $O(n)$

Here every time pivot goes to middle of list and hence forms best case.

$\therefore O(n \log n)$

H/78

$$T(n) = O(n) + O(n) + T(n/4) + T(3n/4)$$

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

Note:

$$T(n) = T(\alpha n) + T((1-\alpha)n) + O(n) \quad , \quad 0 < \alpha < 1$$

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

H/79

$$T(n) \leq T(n/5) + T(4n/5) + n$$

H1/85 both are worst case

the question talks abt realistic time. (∵ the word program is given)

$\therefore t_1 < t_2$

H1/86 $t_1 \rightarrow$ worst case

$t_2 \rightarrow$ best case

$\therefore t_1 > t_2$

H1/87 Both are worst case & they are symmetric.

$\therefore c_1 = c_2$

Master Theorem for solving Dand C Recurrences:

$$T(n) = a\,T(n/b) + f(n) \quad , n > d \qquad a \geq 1;\ b > 1,\ f(n) \text{ is +ve}$$

$$= c \quad , n \leq d$$

Case I:

If $f(n)$ is $O(n^{\log_b a - \epsilon})$, $\epsilon > 0$ then

$$T(n) = \theta(n^{\log_b a})$$

Case II:

If $f(n)$ is $\theta(n^{\log_b a} \cdot \log^k n)$ for some 'k'.

a) $k \geq 0$ then $T(n) = \theta(n^{\log_b a} \cdot \log^{k+1} n)$

b) $k = -1$ then $T(n) = \theta(n^{\log_b a} \cdot \log\log n)$

c) $k < -1$ then $T(n) = \theta(n^{\log_b a})$

Case III :

If $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$ and

$a \cdot f(n/b) \leq \delta \cdot f(n)$ whenever $\delta < 1$ then

$$T(n) = \theta(f(n))$$

Examples on Master theorem :

→ $T(n) = 4T(n/2) + n$

$a = 4$; $b = 2$; $f(n) = n$;

$\log_2 4 = 2$

Case (i) : $n = O(n^{2-\epsilon})$

for $\epsilon > 0$

say $\epsilon = 0.5$

the equation satisfies

$\therefore \Rightarrow T(n) = O(n^{\log_b a})$

$= O(n^2)$

→ $T(n) = 2T(n/2) + n\log n$

$a = 2$ $b = 2$ $f(n) = n\log n$

$\log_a b = 1$

Case (i) : $n\log n = O(n^{\log_b a - \epsilon})$

$n\log n = O(n^{1-\epsilon})$

$\because$ for $\epsilon > 0$

the above equation is false

Case(ii) :

$n \log n = \theta(n \log^k n)$

for $k=1$, the above equation is satisfied.

also $k \geq 0$

$\therefore T(n) = \theta(n^{\log_b a} \cdot \log^{k+1} n)$

$= \theta(n^1 \cdot \log^2 n)$

$= \theta(n \cdot \log n \cdot \log n)$

→ $T(n) = T(n/3) + n$

$a=1, \; b=3; \; f(n)=n$

$\log_b a = \log_3 1 = 0$

Case I : $n = O(n^{0-\epsilon})$

false

Case II : ~~$n = \theta(\log n \log^k n)$~~

$n = \theta(n^{\log_b a} \cdot \log^k n)$

$n = \theta(n^0 \cdot \log^k n)$

$n = \theta((\log n)^k)$ ∝

Case III : $n = \Omega(n^{\log_b a + \epsilon})$

$n = \Omega(n^\epsilon), \; \epsilon > 0$

& $0 < \epsilon \leq 1$

above eqn satisfies

Now $af(n/b) \leq \delta f(n)$, $\delta < 1$

i.e. $n/3 \leq \delta \cdot n$

$\frac{n}{3} \leq \delta n$ ⓒ

~~for $\delta \geq \frac{1}{3}$~~

for $\frac{1}{3} \leq \delta < 1$ the above inequality holds

$\Rightarrow T(n) = \theta(f(n))$

$\Rightarrow T(n) = \theta(n)$

→ $T(n) = 9T(n/3) + n^{2.5}$

$a = 9 \quad b = 3 \quad f(n) = n^{2.5}$

$\log_b a = 2$

Case I:

$f(n) = n^{2.5} = O(n^{\log_b a - \epsilon})$

$n^{2.5} = O(n^{2-\epsilon})$ ✗

Case II:

$f(n) = n^{2.5} = \theta(n^{\log_b a} \log^k n)$

$n^{2.5} = \theta(n^2 \log^k n)$

For no k, the above eqn will be satisfied ($\because \log^k n = O(n^a)$ as $a > 0$)

Case III:

$n^{2.5} = \Omega(n^{\log_b a + \epsilon})$

$= \Omega(n^{2+\epsilon})$ ✓

now consider

$$a \cdot f(n/b) \leq \delta \cdot f(n)$$

$$9 f(n/3) \leq \delta\, n^{2.5}$$

$$9 \cdot (n/3)^{2.5} \leq \delta\, n^{2.5}$$

$$\frac{n^{2.5}}{\sqrt{3}} \leq \delta\, n^{2.5}$$

$$\Rightarrow \frac{1}{\sqrt{3}} \leq \delta$$

$$\Rightarrow \delta \geq \frac{1}{1.732}$$

$$\delta \geq 0.-$$

∴ The above eqn satisfies for $\delta < 1$

$$\therefore T(n) = \Theta(f(n))$$

$$= \Theta(n^{2.5})$$

→ MaxMin :

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

Case (i) : $2 = O(n^{\log_2 2 - \epsilon})$.

$$2 = O(n^{1-\epsilon})$$

true for $\epsilon = 1$

$$\therefore T(n) = O(n^{\log_2 b}) = O(n)$$

→ MergeSort :

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

Case (i) : $bn = O(n^{1-\epsilon})$ ×

Case (ii) : $bn = \Theta(n^1 \log^k n)$

true for $k > 0$

$k \geq 0$ $\therefore T(n) = \theta(n^{\log_a b} \cdot \log^{k+1} n)$

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

→ Matrix multiplication:

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

$n^2 = O(n^{3-\epsilon})$ ✓

$\therefore T(n) = \theta(n^{\log_2 8}) = \theta(n^3)$

Strassen's Matrix mul:

$T(n) = 7T(n/2) + bn^2$

$bn^2 = O(n^{2.81-\epsilon})$ ✓

$\therefore T(n) = \theta(n^{\log_2 7}) = \theta(n^{2.81})$

→ Binary Search:

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

(i) $c = O(n^{\log_2 1 - \epsilon})$

$c = O(n^{-\epsilon})$ ✗

(ii) $c = \theta(n^0 \log^k n)$

for $k = 0$ eqn is satisfied

now $k \geq 0$

$\Rightarrow T(n) = \theta(n^{\log_b a} \log^{k+1} n)$

$= \theta(n^0 \log n)$

$\therefore T(n) = \theta(\log n)$

$T(n) = 2 \cdot T(\sqrt{n}) + \log n$

Here we cannot directly apply master theorem.

so we perform appropriate transformations.

Let $n = 2^k$

$\Rightarrow T(2^k) = 2 \cdot T(2^{k/2}) + k$

Let $T(2^k) \rightarrow S(k)$

$S(k) = 2 \cdot S(k/2) + k$

while transform only T has to be changed to S. Remaining has to be kept same.

$\Rightarrow S(k) = \Theta(k \log k)$

$\Rightarrow T(n) = \Theta(\log n \cdot \log\log n)$

$T(n) = 2T(n/2) + \frac{n}{\log n}$

Case (i): $\frac{n}{\log n} = O(n^{1-\epsilon})$ ✗

Case (ii) $\frac{n}{\log n} = \Theta(n \cdot \log^k n)$

$\Rightarrow k = -1$

$\therefore T(n) = \Theta(n^{\log_b a} \cdot \log\log n)$

$\Rightarrow T(n) = \Theta(n \cdot \log\log n)$

→ $T(n) = T(n/2) + T(n/4) + n$ → Recursion tree method gives $O(n)$

$\because T(n/4) < T(n/2)$

$\Rightarrow T(n) < 2T(n/2) + n$

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

Long Integer Multiplication:

We represent long integers using 1-D array

→ In general time for addition or multiplication is $O(1)$.

However representing with array ~~use~~ time complexity for addition or multiplication will not be $\Theta(1)$.

→ TC for add/sub of two long integers = $O(n)$

→ TC for mul of two long integers = $O(n^2)$ (naive approach)

$\because$ for every digit of one number we multiply it with the other number.

D&C approach:

Let u, v be n digit numbers

$m \leftarrow \lfloor \frac{n}{2} \rfloor$

Now we divide u as follows into w & x ($2(n/2)$ integers)

$u = w * 10^m + x$

where $w = u / 10^m$

$x = u \% 10^m$

similarly we divide v into 2 integers

$v = y * 10^m + z$

$y = v / 10^m$

$z = v \% 10^m$ — w, x, y, z are $n/2$ sized integers

⇒ now $uv = (w \times 10^m + x)(y \times 10^m + z)$

$= (wy \times 10^{2m}) + (wz + xy)10^m + xz$

4 multiplications of size $n/2$

$$T(n) = \begin{cases} c & , n = 1 \\ 4T(n/2) + bn & , n > 1 \end{cases}$$

↓ additions

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

Case (i): $bn = O(n^{2-\epsilon})$

$\therefore T(n) = \theta(n^2)$

$\therefore$ time complexity $\rightarrow O(n^2)$

space complexity

Anatolli karatsuba's optimization: for LIM:

→ The ideology behind the optimization is reducing no of multiplication & operations

$uv = (wy \times 10^{2m}) + (wz + xy)10^m + xz$

Let t be an $n/2$ sized integer

$\therefore t = (w+x)(y+z)$

$t = wy + wz + xy + xz$

$\Rightarrow wz + xy = t - (wy + xz)$

from this we define

$P_1 = wy$

$P_2 = xz$

$P_3 = (w+x)(y+z)$

$\Rightarrow wz + xy = P_3 - (P_1 + P_2)$

Now we write

$uv = (P_1 \times 10^{2m}) + (P_3 - (P_1 + P_2)) \times 10^m + P_2$

This approach requires only 3 multiplication.

However no of add/sub operations are increased.

$\therefore T(n) = 3T(n/2) + O(n) \quad , n > 1$

$= C \quad , n = 1$

$\Rightarrow T(n) = (n^{\log_2^3}) = O(n^{1.58})$

Toom & Cook optimization for LIM:

→ This approach uses multi-way split.

3-way split:

$m \leftarrow \lfloor n/3 \rfloor$

~~$u = \text{base}^{2m}$~~

$u = a \times 10^{2m} + b \times 10^{m} + c$

$v = d \times 10^{2m} + e \times 10^{m} + f$

$uv = (a \times 10^{2m} + b \times 10^{m} + c)(d \times 10^{2m} + e \times 10^{m} + f)$

⇒ no of multiplication req $= 3^2 = 9$

each on $n/3$ sized integers

$$\Rightarrow T(n) = \begin{cases} 9T(n/3) + O(n) & n > 1 \\ c & n = 1 \end{cases}$$

$\Rightarrow T(n) = O(n^2)$

using anatolli's optimization we get

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

In anatolli's optimization value of 'a' is always reduced by 1.

$\Rightarrow T(n) = O(n^{1.89})$

Ideology behind Toom & Cook optimization

let x be no of subproblems solved

$\Rightarrow T(n) = xT(n/3) + O(n)$

Now target is to have $T(n) < O(n^{1.58})$

Here $T(n) = O(n^{\log_3 x}) < O(n^{1.58})$

$\Rightarrow \log_3 x < 1.58$

$\Rightarrow x = 5$

Now the conclusion is that if we solve only 5 subproblems then we can have TC which is less than $O(n^{1.58})$.

→ Toom & Cook were successful in devising equation in which we require to perform only 5 multiplication operations.

$\Rightarrow T(n) = 5T(n/3) + O(n)$

$\Rightarrow T(n) = O(n^{\log_b a})$

$\Rightarrow T(n) = O(n^{1.46})$

4-way split:

normal DAC approach

$T(n) = 16\,T(n/4) + O(n^2)$

Anatolli's optimization

$T(n) = 15\,T(n/4) + O(n)$

$\Rightarrow T(n) = O(n^{1.9})$

Toom & Cook's Optimization:

$\Rightarrow T(n) = xT(n/4) + O(n) \doteq O(n^{\log_4 x}) < O(n^{1.46})$

$\Rightarrow x = 7$

$\Rightarrow T(n) =$

$\Rightarrow T(n) = 7T(n/4) + O(n)$

$\Rightarrow T(n) = O(n^{1.40})$

k-way split:

DC: $T(n) = k^2 T(n/k) + O(n) : O(n^2)$

AK: $T(n) = (k^2-1) T(n/k) + O(n) : O(n^{\log_k (k^2-1)})$

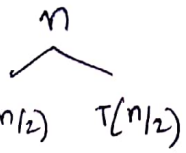
TFC: $T(n) = (2k-1) T(n/k) + O(n) : O(n^{\log_k (2k-1)})$

07/10/20

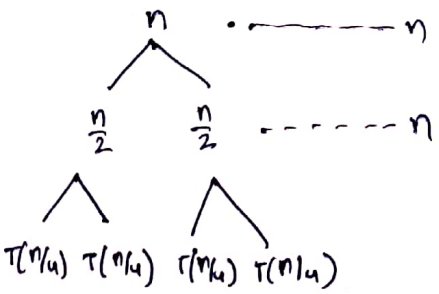
Solving Dand C Recurrences using Recursion tree method:

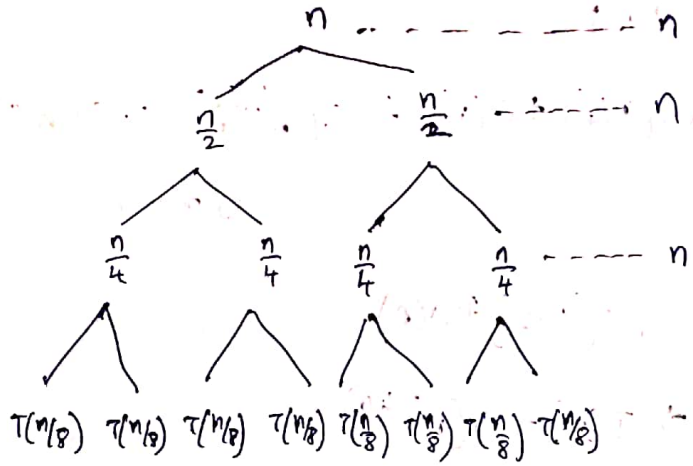
→ Recursion tree can be used to solve asymetric recurrences.

Eg: $T(n) = 2T(n/2) + n$



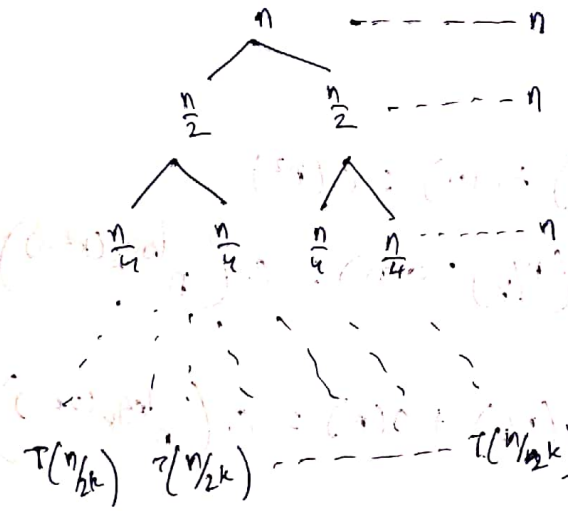
$\Rightarrow T(n/2) = 2T(n/4 + n/2)$





At every level it is n.

So we continue to max possible depth



This last level containing 2^k subproblems.

$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log n$

no of levels are $\frac{n}{2^0}, \frac{n}{2^1}, \frac{n}{2^2} \cdots \frac{n}{2^{k-1}}, \frac{n}{2^k}$

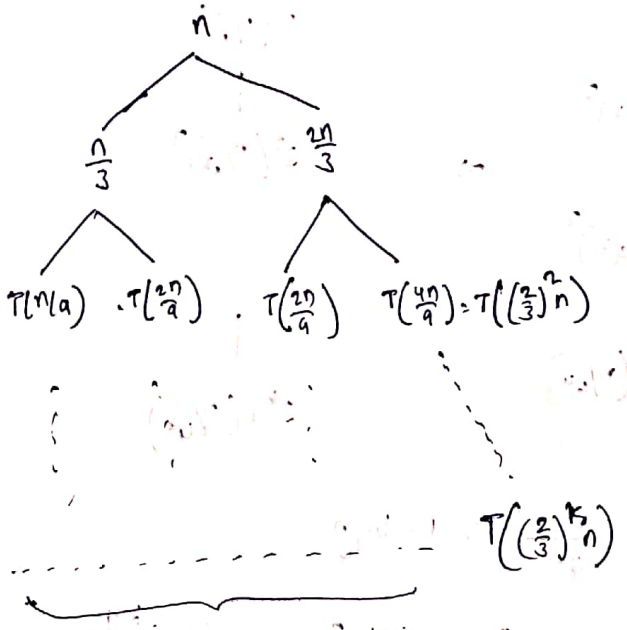
$\therefore$ k+1 level.

$\therefore T(n) = n(k+1)$

$= n(\log n + 1)$

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

Eg: $T(n) = T(n/3) + T(2n/3) + n$; $T(1) = 1$



→ This node goes to maximum depth

→ 2^k subproblem are present in this level.

we assume all nodes have reach this depth and get an approximate answer

$\therefore \left(\frac{2}{3}\right)^k n = 1$

$\left(\frac{3}{2}\right)^k = n$

$k = \log_{3/2} n$

total of $k+1$ levels.

$\therefore T(n) = n(k+1)$

$= n(\log_{3/2} n + 1)$

$\therefore T(n) = O(n \log_{3/2} n) = O(n \log n)$

Eg: $T(n) = 3T(n/4) + n^2$

n^2

$3T(n/4)$

$\Rightarrow$

n^2

$3(n/4)^2$

$3^2 T(n/4^2)$

$\Rightarrow$

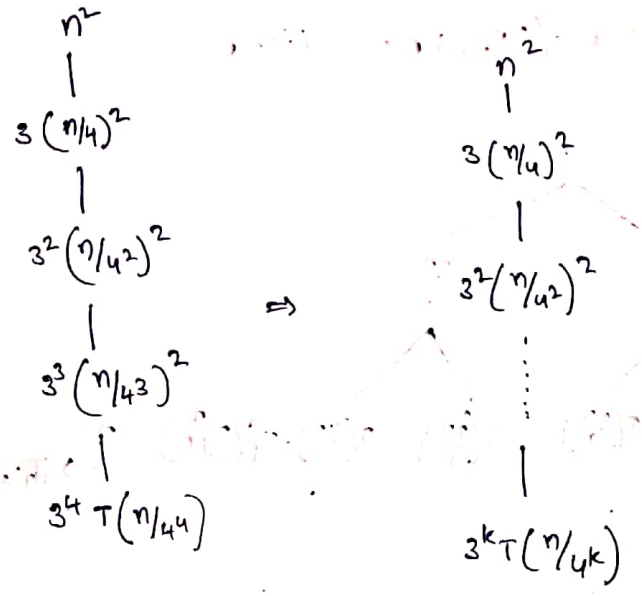
$T(n/4) = 3T(n/4^2) + (n/4)^2$

n^2

$3(n/4)^2$

$3^2 (n/4^2)^2$

$3^3 T(n/4^3)$



$\frac{n}{4^k} = 1 \Rightarrow k = \log_4 n$

$\therefore n^2 + \frac{3}{4^2} n^2 + \left(\frac{3}{4^2}\right)^2 n^2 + \left(\frac{3}{4^2}\right)^3 n^2 + \cdots + \left(\frac{3}{4^2}\right)^{k-1} n^2 + 3^k$

$= n^2 \left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{k-1}\right) + 3^k$

$= n^2 \left(\frac{1}{1-3/16}\right) + 3^{\log_4 n}$

$= n^2 \left(\frac{16}{13}\right) + n^{\log_4 3}$

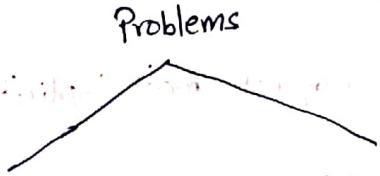
$\Rightarrow T(n) = O(n^2)$.

→ $T(n) = T(n/2) + T(n/4) + n$

Ans : $O(n)$ → solve

Greedy Method:

→ generally used for optimization problem.



Decision problems

→ Characterized by the fact that its result/ outcome is either yes/no.

Eg: Divisibility, prime number etc.

Optimization → by optimization, it never means optimizing time complexity.

→ characterized by the fact that it tries to optimize a given criterian of the problem.

Eg: knapsack, Job sequencing, etc.

Eg: Graph coloring ⟶ optimization problem

Travelling salesman ⟶ optimization problem

shortest path ⟶ optimization problem

n-Queens ⟶ Decision (whether queens can be placed in non-attacking position or not)

Sorting ⟶ Decision

Banker's algo ⟶ Decision

CPU scheduling ⟶ Optimization

Memory allocation ⟶ Optimization.

Terminology:

i) **Problem Definition** : Defining problem Eg: n-queens
↳ we define what the problem is.

ii) **Constraints**
- Implicit (local constraints to the problem. (Feasibility Criteria) Eg: no two queens should in attacking positions)
- Explicit (depends on instance of problem. For example solving 4-queens, so array size is 4)

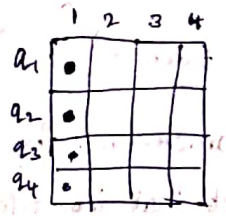
2.

(iii) Solution space:

All possible ways of organizing inputs but satisfying explicit constraints.

i.e., It may or may not meet implicit constraint.

Eg:



⇒ This part of soln space

i.e., ⟨1,2,3,4⟩ is part of soln space.

However ⟨1,3,5,2⟩ is not a part of soln space because 4-queens problems explicit constraints is that queens could be place in locations from 1 to 4.

∴ Having 5 doesn't meet explicit constraint.

(iv) So for n-queens problems size of soln space = n!

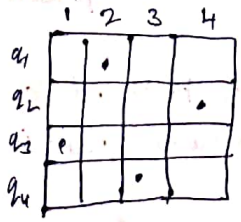
If problem is constructing binary search tree with 'n' nodes, then size of soln space = $\frac{1}{n+1} {}^{2n}C_n$

→ Working with entire soln space is called brute fore method.

(iv) Feasible solution:

The solutions in solutions space, that satisfy the implicit constraints of the problem is/are called feasible solutions.

Eg:



4-queens has 2 feasible solns

(v) Objective Function:

Refers to min/max of given criterian of the problem.

Problems may have objective function or may not have.

↳ n-queens problem doesn't have objective function.

Eg: Consider shortest path problem

→ soln space has all paths blw any two vertices

→ feasible soln has only those paths that connect source and destination vertex.

→ Here objective function is to minimize the cost of path.

(vi) Optimal Solution:

It is the feasible solution that satisfies objective function.

→ optimal soln is always unique.

Note:

→ If a problem doesn't have objective function then we find only feasible soln.

→ Decision problems doesn't have objective function, hence for decision problems we find only feasible solutions.

→ Greedy method is an algorithm design strategy, used for solving problems, whose outcomes are seen as a result of making a sequence of decisions.

→ Greedy method makes these decision in a stepwise manner by applying the principle of local optimality (greedy choice property)

→ Principle of local optimality says that whatever the initial state & available options, greedily select that option that satisfies the objectivity function (this give feasible soln)

Eg: SJF algorithm, Shortest seek time First

(i) Knapsack Problem:

→ given a knapsack of capacity: M

→ given n-objects (O_i) & each object is associated with weight (w_i) and profit (p_i)

→ Maximizing the profit such that the total weight that put into knapsack should not exceed its capacity (M).

→ For every object we make decision to find x_i (fraction of O_i put into knapsack)

$\updownarrow$

$0 \le x_i \le 1$

for x_i the weight put into knapsack $= w_i x_i$

the profit object $= p_i x_i$

∴ we need to maximize $\sum_{i=1}^{n} p_i \cdot x_i$ such that

$\sum_{i=1}^{n} w_i \cdot x_i \le M$ → x_i can be real value

This type of knapsack problem is called → real knapsack / fractional knapsack / greedy knapsack

→ Size of soln space of real knapsack problem infinite.

→ for 0/1 knapsack the size of soln space is 2^n.

Consider below instance of knapsack problem.

$n=3;\ m=20;\quad \langle P_1, P_2, P_3 \rangle = \langle 25, 24, 15 \rangle$

$\langle w_1, w_2, w_3 \rangle = \langle 18, 15, 10 \rangle$

$\langle x_1, x_2, x_3 \rangle = ?$

Explicit constraint of knapsack problem is

$$\sum_{i=1}^{n} w_i > M$$

(i) Greedy about profit:

P_1 has highest profit

$\Rightarrow x_1 = 1$

remaining capacity = $20 - 18 = 2$

P_2 has next highest profit

$\Rightarrow x_2 = \frac{2}{15}$

Here profit $P = P_1x_1 + P_2x_2 + P_3x_3$

$= 25(1) + 24\left(\frac{2}{15}\right) = 28.3$

(ii) Greedy about weight:

w_3 is least.

$\therefore x_3 = 1$

remaining capacity = $20 - 10 = 10$

w_2 has next least weight

$\Rightarrow x_2 = \frac{10}{15} = \frac{2}{3}$

$\therefore$ profit = $P_1x_1 + P_2x_2 + P_3x_3$

$= 0 + \frac{2}{3}(24) + 1(15)$

$= 31$

(iii) Greedy about object that has highest profit per weight ratio:

$\frac{P_1}{\omega_1} = 25/18 = 1.3 \Rightarrow x_1 = 0$

$\frac{P_2}{\omega_2} = 24/15 = 1.6 \Rightarrow x_2 = 1$

$\frac{P_3}{\omega_3} = 15/10 = 1.5 \Rightarrow x_3 = \frac{5}{10}$

Profit $P = P_1x_1 + P_2x_2 + P_3x_3$

$= 0 + 24(1) + 15\left(\frac{5}{10}\right)$

$= 24 + 7.5 = 31.5$

→ 3rd approach is optimal. ✓

→ So, arrange the objects in decreasing order of P/w ratio & keep on including the objects, until a kth object cannot be placed and for this object we take proportionate fraction.

Time complexity after sorting → O(n)

Time complexity including sorting → O(nlog n)

Eg: n=7 m=15; $\langle P_1 - P_7 \rangle = \langle 10, 5, 15, 7, 6, 18, 3 \rangle$

$\langle \omega_1 - \omega_7 \rangle = \langle 2, 3, 5, 7, 1, 4, 1 \rangle$

$\left\langle \frac{P_1}{\omega_1} - \frac{P_7}{\omega_7} \right\rangle = \langle 5, 1.66, 3, 1, 6, 4.5, 3 \rangle$

$x_1 = 1$, $x_2 = 2/3$, $x_3 = 1$, $x_5 = 1$, $x_6 = 1$, $x_7 = 1$

$P = 10 + \frac{2}{3}(5) + 15 + 0 + 6 + 18 + 3$

$= 55.33$

(H/36) It is clear the optimal profit is 60.

$\langle w_i \rangle : \langle 10, 7, 4, 2 \rangle$

$\langle P_i \rangle : \langle 60, 28, 20, 24 \rangle$

$\langle \frac{P_i}{w_i} \rangle : \langle 6, 4, 5, 12 \rangle$

② ④ ③ ①

Select $O_4 \Rightarrow$ remaining weight $= 11-2 = 9$ $P = 24$

select $O_2 \Rightarrow w_2 > 9$

∴ not possible.

select $O_3 \Rightarrow$ remaining weight $= 9-4 = 5 \Rightarrow P = 24 + 20 = 44$

Select $O_4 \Rightarrow w_2 > 5$

∴ not possible

$\therefore V_{opt} = 60$ $V_{greedy} = 44$

∴ Ans: 16

Job Sequencing with Deadlines (JSD)

→ n-jobs / programs / Process

→ Each job (J_i) arrives at time 0 and needs one unit time.

→ Every job is associated with a deadline d_i & profit P_i. The profit is obtained only if the job is completed within its deadline.

Problem stmt:

Select a subset of given n-jobs such that the jobs in the subset are completable within their deadlines and generate maximum profit.

→ Here the trivial case is when all the jobs can be scheduled. 04

→ size of soln space is 2^n.

It means working out this problem using brute force, the time complexity is $O(2^n)$.

Eg: $n=4$; $(J_1 - J_4)$;

$(P_1 - P_4) = \langle 100, 15, 10, 40 \rangle$

$\langle d_1, d_2, d_3, d_4 \rangle = \langle 2, 1, 2, 1 \rangle$

feasible soln for this problem are those subsets having all the jobs which can be scheduled within the deadline (Implicit constraint)

Let J be soln subset. Initially $J = \emptyset$ ← feasible soln

I. $|J| = 0$

feasible soln

profit, $P = 0$

II: $|J| = 1$

$J = \{J_1\}$

feasible soln; $P = 100$

$J = \{J_2\}$

feasible soln; $P = 15$

III: $|J| = 2$

$J = \{J_1, J_2\}$

J_2	J_1

feasible soln

profit → 100+15 = 115

schedule: J_2, J_1

$J_3 = \{J_1, J_3\}$

this is not a feasible soln

Profit = 110

schedule: J_1, J_3 (or) J_3, J_1

$J = \{J_2, J_4\}$

This is not a feasible soln.

∵ we cannot schedule J_2 & J_4 within their deadlines.

IV - $|J| = 3$

no three jobs can be scheduled

∴ there is no feasible soln with $|J| = 3$.

Note:

The maximum no of jobs that a feasible subset can have = maximum deadline.

(This rule applies only when each job takes exactly 1 unit of time)

Greedy approach for JSD:

$n = 4$; $\langle j_1 - j_4 \rangle$; $\langle P_1 - P_4 \rangle = \langle 100, 15, 10, 40 \rangle$; $\langle d_1 - d_4 \rangle = \langle 2, 1, 2, 1 \rangle$

Let J be the soln subset

→ The job with highest profit will always be part of optimal soln

Step(i): Arrange the jobs in decreasing order of profits.

i.e., j_1, j_4, j_2, j_3

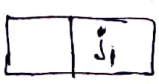
one by one

Schedule the jobs of sorted order such that the job goes into maximum possible slot below its deadline.

∴ Pick j_1:

deadline is 2

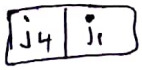
slot 2 is free

∴ 

Pick 4:

slot deadline → 1

slot '1' is empty



pick j_2:

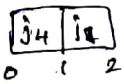
slot 1 is not empty.

pick j_3:

slot 2 is not empty. Now check slot 1.

slot 1 is not empty.

∴ final schedule:



∴ profit = 100 + 40 = 140

Eg: n = 7

$\langle J_1 - J_7 \rangle$;

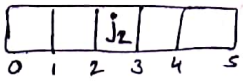
$\langle d_1 - d_7 \rangle = \langle 2, 3, 4, 5, 2, 4, 4 \rangle$

$\langle P_1 - P_7 \rangle = \langle 54, 92, 30, 10, 15, 60, 45 \rangle$

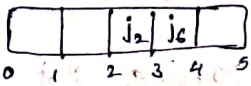
decreasing order of profits

$j_2, j_6, j_1, j_7, j_3, j_5, j_4$

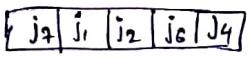
Pick j_2:



Pick j_6:



pick j_7:



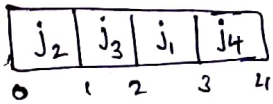
profit = 54 + 92 + 10 + 60 + 45

= 261

Eg: $n=6$; $\langle j_1 - j_6 \rangle$;

$\langle d_1 - d_6 \rangle = \langle 4, 4, 3, 4, 2, 3 \rangle$

$\langle p_1 - p_6 \rangle = \langle 105, 38, 46, 160, 20, 19 \rangle$

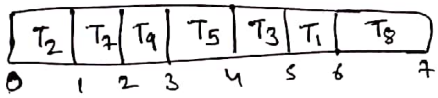


$P = 105 + 38 + 46 + 150$

$= 339.$

H/37

a)



∴ opt (d)

b) $P = 15 + 20 + 30 + 18 + 23 + 16 + 25$

$= 147$

Control Abstraction of Greedy method

```
Procedure GREEDY (A, n)        ↳ i/p size
{
    1. solution ← ∅
    2. for i ← 1 to n
       {
           x ← SELECT(A);
           if (FEASIBLE (x, solution)) // checks whether including x in soln
                                       is feasible or not.
               ADD(x, solution);
       }
    3. return (solution);
}
```

From this control abstraction,

we say minimum time complexity is $O(n)$

i.e., when, SELECT(), FEASIBLE(), ADD() are $O(1)$

High-level Pseudocode for JSD:

```
Algorithm JSD (d, P, J, n)
// we assume that jobs are arranged in decreasing order of profits
// P1 ≥ P2 ≥ P3 ... ≥ Pn
{
    1. J ← {1};
    2. for i ← 2 to n ——→ O(n)
       {
           if (all job in J ∪ {i} can be completed by their ——→ O(n)
               deadlines) then
               J ← J ∪ {i};
       }
}
```

Time Complexity of JSD: $O(n^2)$

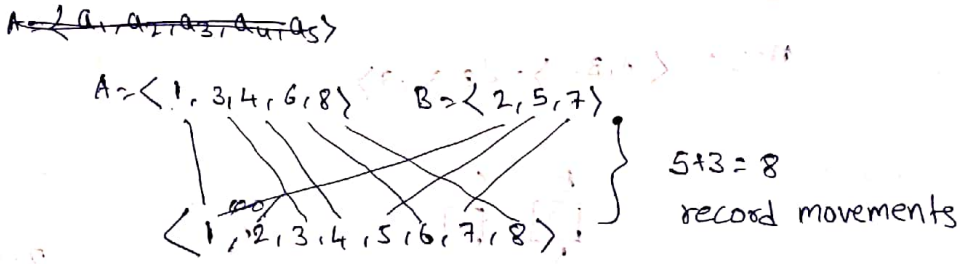
→ The working of JSD is similar to the worst case behaviour of insertion sort.

Optimal Merge Patterns

→ Merging of files: Files have records in a sorted order.

→ Given n files, it is required to merge them using 2-way merging to get a single sorted file with an objective of minimizing the total no of record movements.

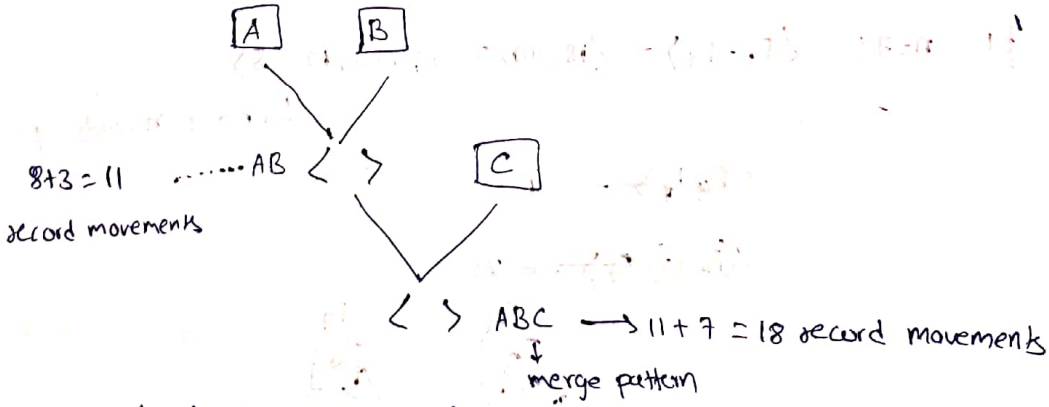
Eg: Consider merging two file A, B with 5, 3 records respectively



Eg 2: n=3;

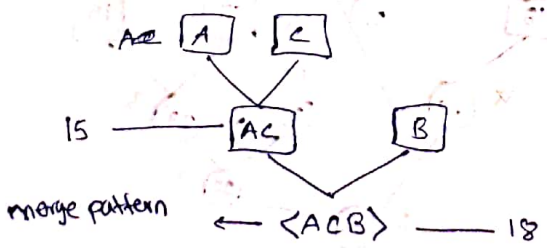
⟨A, B, C⟩ = ⟨8, 3, 7⟩
↳ no of records

→ Consider we merge A, B first



∴ 11 + 18 = 29 records movements.

→ Now consider we merge A, C first



∴ 33 record movements.

→ Here size of soln space = $n!$

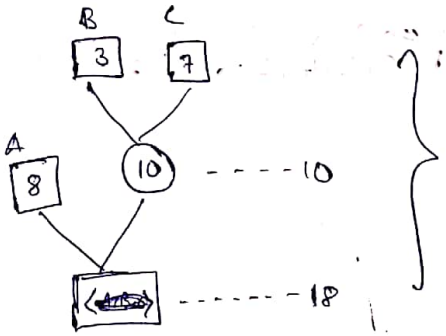
∴ TC with brute force = $O(n^n)$

→ Also here every soln in the soln space is a feasible soln.

Greedy strategy for OMP

At each step (of every (n-1) steps), select the two files with least no of records, merge them and add to the list.

$n=3$; $\langle A,B,C\rangle = \langle 8,3,7\rangle$



→ 2-way Optimal binary merge tree (reverse of the figure)

∴ Min record movements = 10+18 = 28

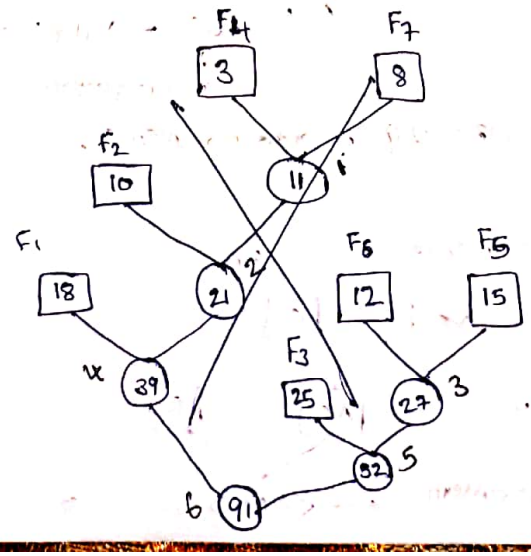
Optimal merge pattern = $\langle B,C,A\rangle$

Eg: $n=7$; $\langle F_1 - F_7\rangle = \langle 18,10,25,3,15,12,8\rangle$

↳ no of records per each file

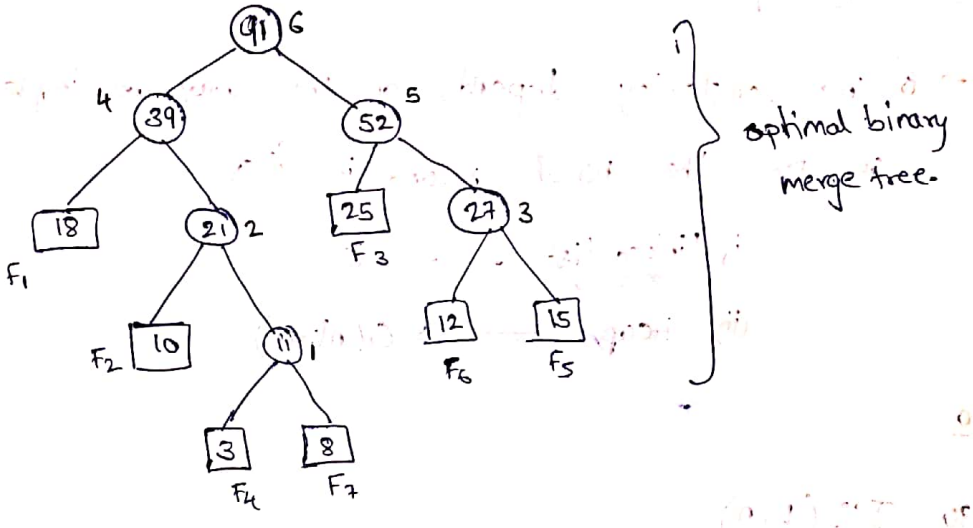
~~$\langle F_4 F_7\rangle$~~ 11

~~$\langle (F_4 F_7) F_2\rangle$~~ 21



∴ no of record movements = 11+21+39+~~25~~+27+52+91

= 241



→ Sum of internal nodes gives min no of record movements.

Note:

Total no of record movements involved during OMP is given by weighted external path length.

i.e., $\sum_{i=1}^{n} q_i * d_i$

d_i → distance from root to F_i

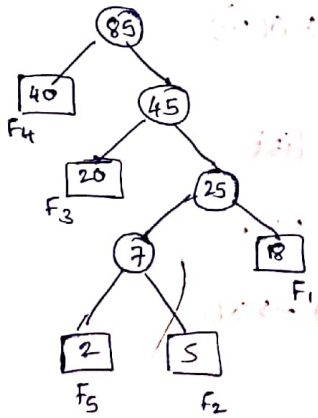
q_i → size of F_i

= (2×18) + (3×10) + (2×25) + (4×3) + (3×15) + (3×12) + (4×8)

= 36 + 30 + 50 + 12 + 45 + 36 + 32

= 241

Ex: n=5; ⟨F_1-F_5⟩ = ⟨18, 5, 20, 40, 2⟩



Min record movement

= 85 + 45 + 25 + 7

= 162

Performance Analysis:

Time Complexity:

→ Time complexity depends on the way we implement or store the no of record in files

(i) linked list (or) array $\longrightarrow O(n^2)$

(ii) Heap $\longrightarrow O(n \log n)$

11/10/20

Algo JSD(L, n)

// L is a list of n single node binary tree as described below

{

1. for i ← 1 to n-1 do
2. call GETNODE(T) // creates a node T
3. LCHILD(T) ← LEAST(L)
4. RCHILD(T) ← LEAST(L)
5. WEIGH(T) ← WEIGHT(LCHILD(T)) + WEIGHT(RCHILD(T))
6. call INSERT(L, T)
7. repeat
8. return (LEAST(L))

}

Here time complexity depends on the way we implement list L.

(i) If L is ordered linear list

LEAST(L) $\longrightarrow O(1)$

INSERT(L, T) $\rightarrow O(n)$

(ii) If L is unordered linear list

LEAST(L) $\longrightarrow O(n)$

INSERT(L, T) $\rightarrow O(1)$

Thus if the algorithm is implemented with a list

∴ time complexity ⟶ $O(n^2)$

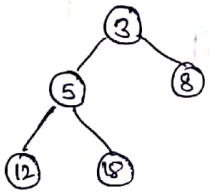
Space complexity ⟶ $O(n)$

↳ for tree created by nodes T

(iii) If L is implemented as a heap (min heap)

first we build the heap with O(n) time

$\langle F_1 - F_5 \rangle = \langle 5, 12, 8, 3, 18 \rangle$



LEAST(L) takes $O(\log n)$ time

INSERT(L, T) takes $O(\log n)$ time

⇒ Time complexity ⟶ $O(n \log n)$

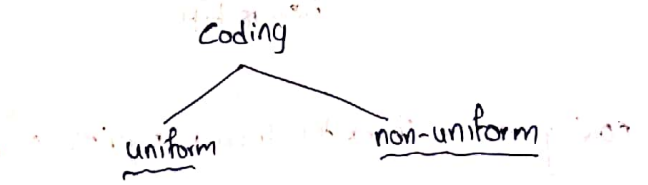
Space Complexity ⟶ $O(n)$

↳ space for heap

Huffman Coding

→ It is an application of optimal merge pattern

→ Huffman coding is a data encoding technique.



→ Each character is represented with equal no of bits

Eg: ASCII

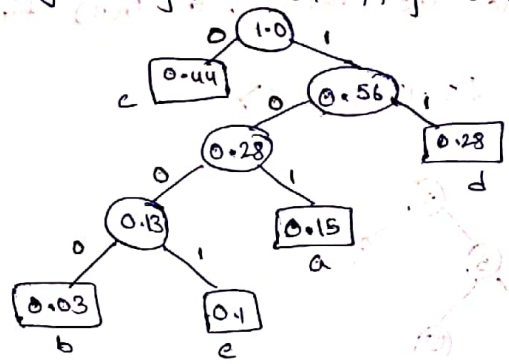
→ Characters may be represented with different no of bits.

→ Huffman coding is a non-uniform coding.

→ Characters / elements having higher frequencies / probabilities of occurences must be encoded with less no of bits.

Eg : $L = \langle a, b, c, d, e \rangle = \langle 0.15; 0.03; 0.44; 0.28; 0.1 \rangle$

Huffmancoding : to get codes, apply OMP algorithm.



Mark every left edge as 0 and right edge as 1
(reverse can also be followed)

⇒ a → 101
b → 1000
c → 0
d → 11
e → 1001

Now consider the text ccdccddade

then corresponding binary stream is 0 0 11 0 0 0 11 101 11 1001

i.e., 18 bits

using uniform coding every character would require 3 bits and hence the text would require 30 bits.

Ex: Consider binary stream 10101011111

Find text. Considering previous example

Sol:

For i/p traverse the tree until you reach the leaf. After reach leaf, print that character and traverse again from the root using remaining i/p.

∴ In above stream after reading 10 we arrive at leaf 'a'

after reading 0 we arrive at leaf 'c'.

⋮

∴ 10 1 0 11 0 11 11 → acdedd

→ Find avg no of bits used per character in huffman coding.

$(0.15)(3) + (0.08)(4) + (0.44)(1) + (0.28)(2) + (0.1)(4) = 1.97$ bits

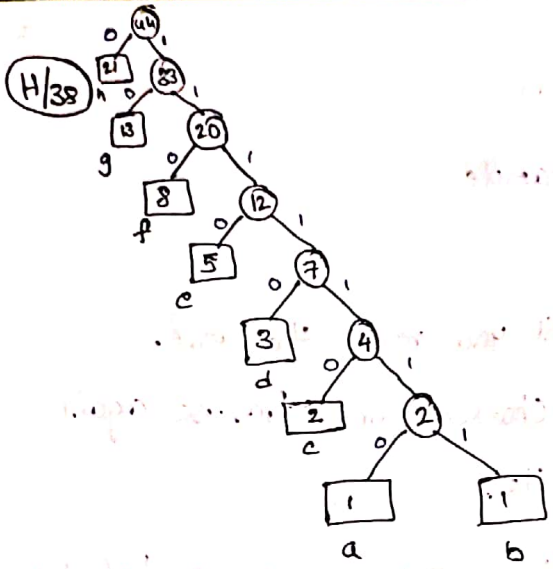
- i.e., $\sum_{i=1}^{n} d_i q_i$

i.e., avg no of bits is given by ~~external~~ weighted external path length = sum of internal nodes.

→ Disadvantage of huffman coding is that even a single bit error can make the whole text corrupted.

→ If there are 2^n characters and if each character has equal probability (i.e., $\frac{1}{2^n}$) then avg no of bits req in huffman coding is n. Also the tree formed will be a complete binary tree. with all the character nodes at leaf position.

H/38



110 111 00 1110 10
f d h e g

∴ fdheg

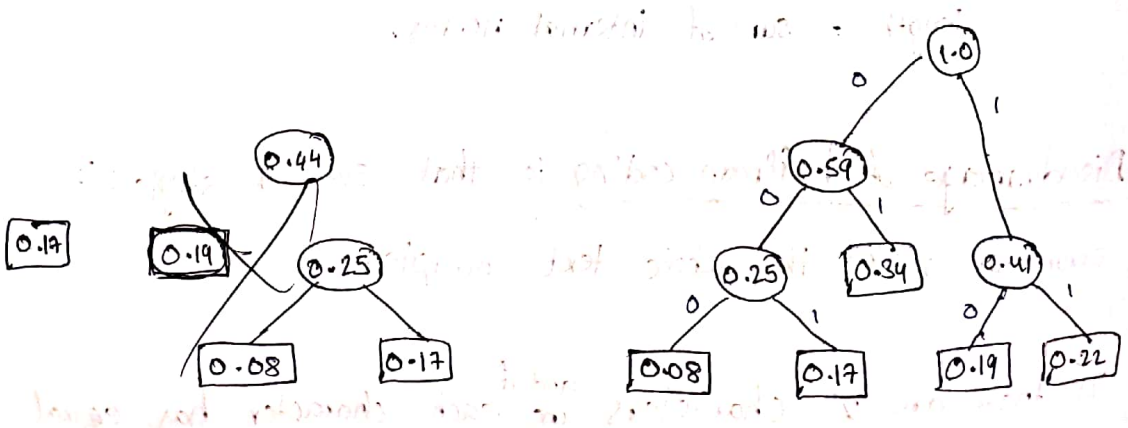
Find avg no of ~~characters~~ bits per characters.

$$= \frac{\sum f_i d_i}{\sum f_i} \quad \Sigma f_i \rightarrow \text{frequencies}$$

$$= \frac{\text{Sum of internal nodes}}{1+1+2+3+5+8+13+21}$$

$$= \frac{122}{54}$$

H/39



avg no of bits = 3(0.08) + 3(0.17) + 2[0.34 + 0.19 + 0.22]

= 0.75 + 2(0.75)

~~= 1.5 bits~~ = 2.25 bits

∴ 225 bits

Spanning Trees:

$G = (V, E)$; $|V| = n$; $|E| = e$;

G is weight undirected graph

Spanning tree: A subgraph $T(V, E')$ of a given graph $G = (V, E)$ where $E' \subseteq E$ is a spanning tree iff T is connected acyclic graph (tree)

Solution space:

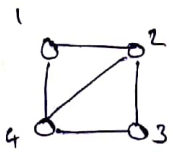
i.e., Maximum no of spanning trees $= n^{n-2}$ → kaley's formula for a complete graph of n vertices.

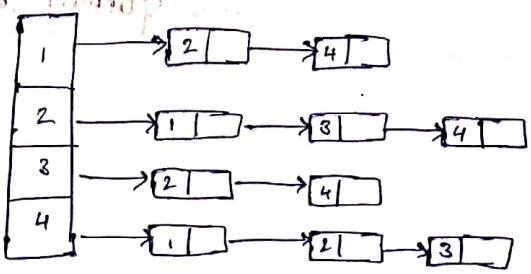
→ ~~No of spanning trees possible for any graph is given by cofactor of any element of the matrix representing the graph~~

Note:

→ Any graph based problem that use adjacency matrix or) cost adjacency matrix has time complexity no less than $O(n^2)$

→





undirected graph → ~~n+2e~~ n+2e nodes.

directed graph → n+e nodes

→ Thus any graph problem which represent graph with adjacency list has atleast $O(n+e)$

→ Max value possible for e is $O(n^2)$

↳ In some cases $O(n+e) = O(n^2)$

∴ For complete graphs / dense graphs, adjacency matrix representation is desirable.

For sparse graphs adj. list representation is desirable.

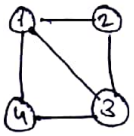
Calculating no of spanning trees for a non-complete graph:

(i) Create adjacency matrix for given graph.

(ii) Replace all diagonal elements with degrees of node and replace all non-diagonal 1's (edges) with -1.

(iii) Now calculate co-factor of any element of the matrix and this gives no of spanning trees.

Eg:

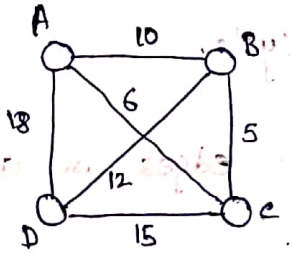


$$\rightarrow \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

Cofactor of 3 = $\begin{vmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{vmatrix} = 8$

∴ 8 spanning trees.

Minimum cost spanning Tree (MCST):

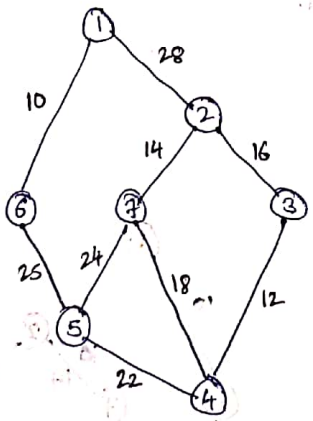


Applications of spanning tree:

→ Used in multicast, broadcast

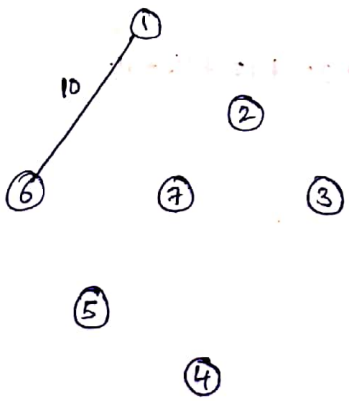
→ MCST is used in electronic & eletrical applications (in circuits)

Construction of MCST:



Prims Algorithm:

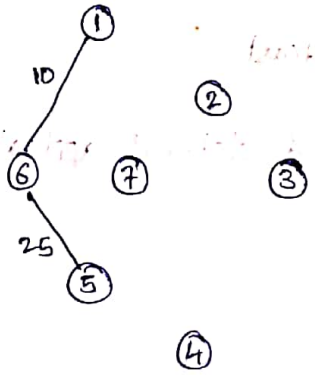
i) choose the least cost edge and add to the solution.



(ii) The next edge to be added, must be adjacent to the vertices included so far such that it has least cost and it does not form a cycle.

Repeat step (ii) until a total (n-1) edges are added

So add edge 1-6

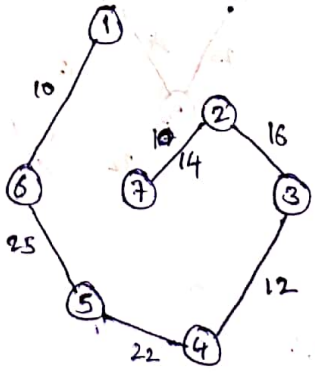


Add 5-4 ... 22

Add 4-3 ... 12

Add 2-3 16

Add 2-7 14



6 edges are added and we finish the process here

∴ Cost =

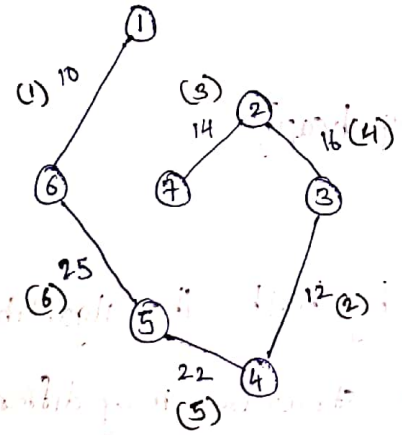
Cost of MCST = 10+25+22+12+16+14

= 99

Kruskal's algorithm

1. Arrange the edges of the graph in increasing order. (min heap)
2. Delete the least cost edge from the list. (delete root of min heap)
3. Add the edge to the soln iff it does not lead to cycle.
4. Repeat (2) & (3) until a total of (n-1) edges are added.

edges → 10, 12, 14, 16, 18, 22, 24, 25, 28



Add 10
Add 12
Add 14
Add 16
Adding 18 leads to a cycle
Add 22
Add 24 forms a cycle
Add 25

The numbering shows the order in which the edges are added.

→ At the time of construction edges 18, 24 are discarded whereas 28 is not considered.

Difference b/w Prim's & kruskal's algorithms

(i) Prim's method always maintains tree structured property at every step (connectedness) where as kruskal's algorithm may or may not.

(ii) Time complexities:

prim's algorithm:

- adjacency matrix & linear list : $O(n^2)$
- adjacency list & heap : $O((n+e)\log n)$

kruskal's algorithm

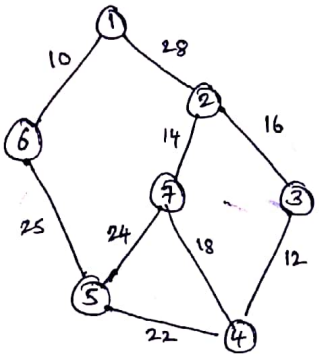
- $O(e \log e)$ [using heap]

(iii) The cost of MCST generated by both the algorithms is same. However, the tree structure may differ.

* The tree structure may differ when there are multiple edges of same weight (or) ~~when there are two or more spanning trees with same minimum cost~~

* Tree structure generated by prim's and kruskal will if all the edge costs are distinct. ~~and there is only one spanning tree with minimum cost~~.

Dijkstra's algorithm:

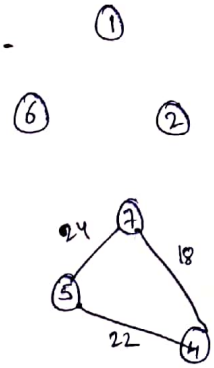


(This algorithm doesn't come totally under greedy)

1. Starting adding any edge and continue adding any edges one by one until a cycle is formed. Once a cycle is formed remove the maximum cost edge from the cycle.

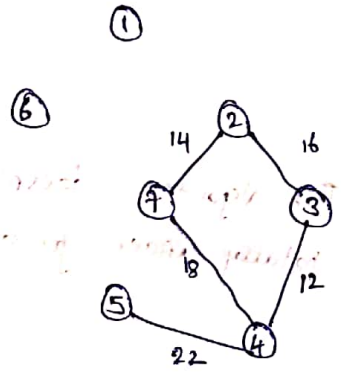
Repeat the above step until n-1 edges are added or all the edges are exhausted. (note that no edge should be added 2nd time after its removal)

Eg: Consider adding 24, 22, 18.

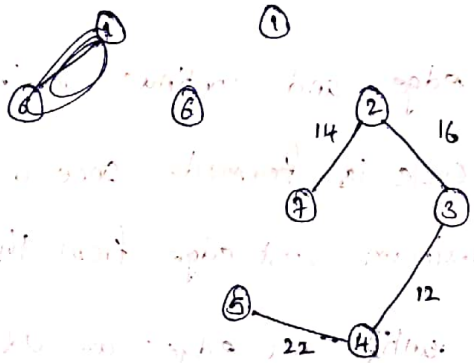


Remove 24 from cycle.

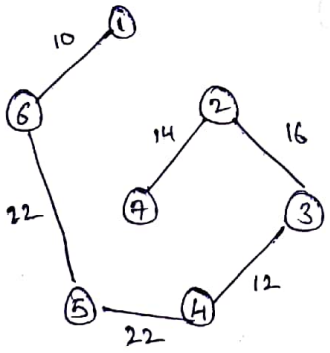
Add 12, 16, 14



Remove 18



Add 25, 10



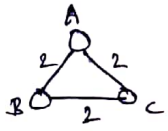
6 edges are added

∴ stop the process

Calculating no of MCSTs:

→ If all edges costs are distinct the graph will have a unique MCST.

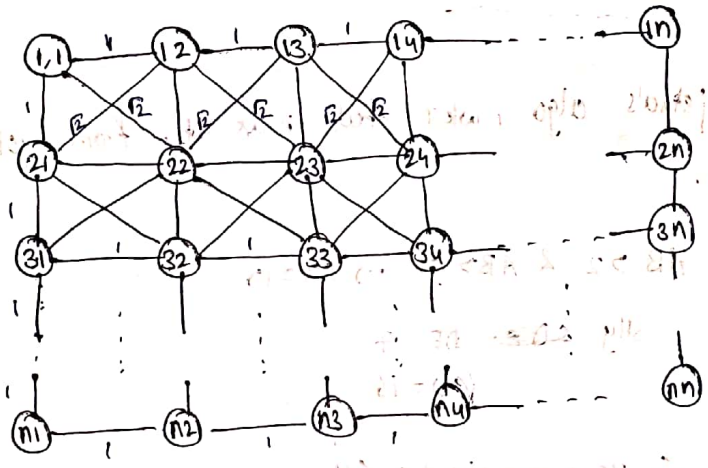
→ If there exist multiple edges of same cost then there may exist more than one MCST with same cost.

Eg: 

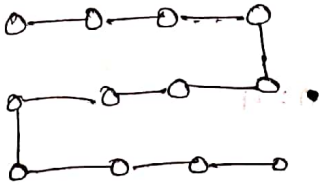
No of MCSTs possible = 3

Note: There is no standard formula for calculating no of MCSTs. we just need to calculate from the graph given using permutations & combinations.

H/40

(i) 

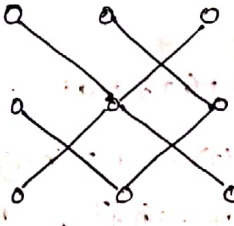
So the min cost spanning tree is



∴ Cost is $(n-1)(n) + (n-1) = (n+1)(n-1) = n^2-1$

(ii) Find cost of max cost spanning tree.

take $n=3$



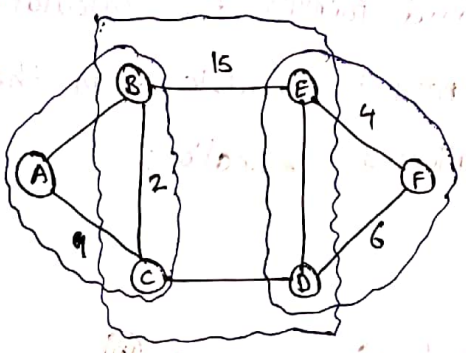
we have added 7 edges to each of cost 2.

Now we can no more add any edge of '2'.

∴ we add one edge of cost 1.

∴ Cost is $(n^2-2)(2) + 1$

H/41



using dikjstra's algo (wkt that max wt. from cycle has to removed.

∴ AB > 2 & AB > 9 ⇒ AB = 10

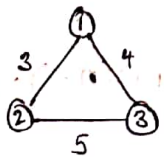
silly DE = 7

CD = 16

∴ 36 + 10 + 16 + 7 = 69

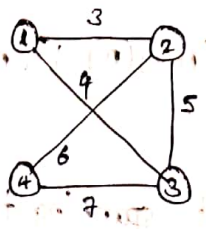
H/42

n = 3



i.e., 3+4 = 7

n = 4



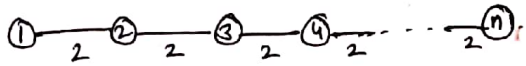
i.e., 3+4+6

∴ the sequence is 3 + 4 + 6 + 8 + 10 + ...

i.e., $3 + 2(2) + 2(3) + \cdots + 2(n-1)$

$3 - 2 + 2[1 + \cdots + (n-1)] = 1 + 2\left(\frac{n(n-1)}{2}\right) = n^2 - n + 1$

(H/43) MCST is



$\therefore 2(n-1)$

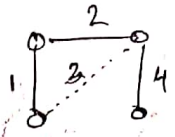
(H/44) $\{1,2,3,4,5,6\}$

'1' is always parts of MCST

Using kruskal's algorithm, next we add 2 and this doesn't lead to cycle no matter where the edge is

Choosing '3' in first case it could be forming a cycle

So '4' will be part of MCST



~~∴ MIA COS~~

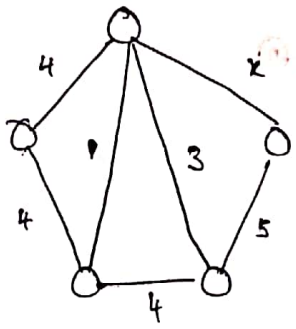
Max possible cost of MCST = 1+2+4 = 7

(H/45) After modification,

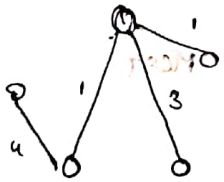
MCST's structure remains same with weights increased.

$\therefore 500 + 99(5) = 995$

H/46



$x=1$ ⇒



(or)

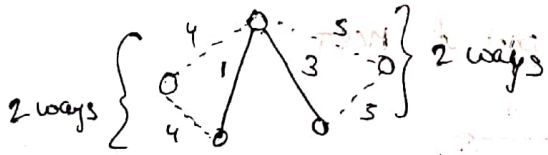
Silly for $x=2$,

2 MCSTs.

for $x=3$, 2 MCSTs

for $x=4$, 2 MCSTs

for $x=5$



$\therefore 2 \times 2 = 4$ ways

For this kind of problems use dijkstra's algo.

Having $x=5$ will add 2 possibilities of removing max cost edge from its corresponding cycle

H/47

(i) ~~False for w-regular graph~~.

(ii) ~~true~~

(iii)

(i) True, since we can start construction of MCST for that edge (kruskal's algo)

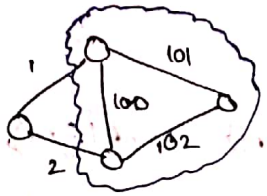
(ii) true

(iii) False in the case of w-regular graph

(iv) true

H/48

I.



'100' is lightest, still it is excluded cuz it is heaviest of another cycle

∴ false

II. true from dijkstra's algo

∴ true

12/10/20

Algo PRIM (E, COST, n, T, mincost)

// E is set of edges in G

// COST(n,n) is cost adjacency matrix such that COST(i,j) is either positive or ∞ if there is no edge b/w i,j

// T[1...n-1, 1..2] is store set of edges of MCST

{

1. real COST(n,n), mincost;
2. integer NEAR(n), n, i, j, k, l, T(n-1,2);
3. (k,l) ← edge with min cost ------ O(e)
4. mincost ← cost(k,l)
5. (T(1,1), T(1,2)) ← (k,l)
6. for i←1 to n do ------ ------ O(n)
   if COST(i,l) < cost(i,k) then
   NEAR(i) ← l
   else
   NEAR(i) ← k
7. for

7. NEAR(k) ← NEAR(l) ← 0

8. for i ← 2 to n-1 do ---- O(n),
{
a) let j be an index such that NEAR[j] ≠ 0 and COST[j, NEAR[j]] is minimum. ---------- O(n)

b) (T[i,1], T[i,2]) ← (j, NEAR[j])

c) mincost ← mincost + COST[j, NEAR[j])

d) NEAR[j] ← 0

e) for k ← 1 to n do ---------- O(n)
{
if NEAR(k) ≠ 0 and COST(k, NEAR(k)) > COST[k,j) then
NEAR(k) ← j
}
}

9. if mincost ≥ ∞ then
print ("no spanning tree")

}

Time complexity

using adjacenty matrix : $O(n^2)$

using heap / Red black tree:

(i) Creating heap with e edges —— O(e)

(ii) Step 3 —— O(log e)

(iii) Step 8 —— O(n)

Step 8.a) —— O(log n) as using heap with cost(j, NEAR(j)) values.

Step 8.e) . NEAR(k) ← j corresponds to decrease key operation and it takes O(log n)

In total se occurs $n^2 \log n$ times (in worst case
However in general it is $e \log n$

$\therefore TC = O(e) + O(\log e) + \cdots + n\log n + e\log n$

(8.a) (8.e)

i.e., $O((n+e)\log n) \cong O(e \log n)$

↳ in worst case it is $O(n^2 \log n)$

Hence for dense graphs we use only adjacency matrix but if the graph is sparse graph using heap is better.

Space Complexity:

without heap:

S.C = size(T) + size(near)

= O(n)

with heap:

S.C = size(heap for edges) + size(heap for near)

S.C = O(e+n)

Shortest Paths:

I. Single pair shortest path

II. Single source shortest path

Dijkstra's	Bellman-Ford
→ Greedy	→ Dynamic
→ Always works good for +ve wt edges	→ Always works correctly with -ve wt. edges but not having -ve cycle reachable from source.
→ May or may not work with -ve edges	

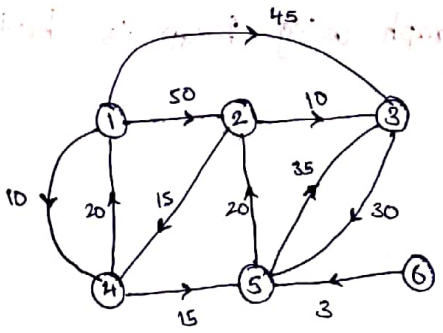
III. All Pair shortest Paths

→ one way is
- we can use single source shortest paths for all vertices

→ second approach is

Floyd-warshall's algorithim

Single Source Shortest Paths : Dijkstra's Algorithm:



This approach works for both directed & undirected graphs

Consider finding shortest path values from vertex '1'

i) Matrix form:

d(x): The value of vertex 'x' is defined as cost of shortest path known so far

vertex selected	d-values 1	2	3	4	5	6
{1}	-	50	45	(10)	∞	∞
{1,4}	-	50	45	(10)	(25)	∞
{1,4,5}	-	(45)	45	(10)	(25)	∞
{1,4,5,2}	-	(45)	(45)	(10)	(25)	∞
{1,4,5,2,3}	-	(45)	(45)	(10)	(25)	(∞)

→ This process of reducing distance is called relaxation.

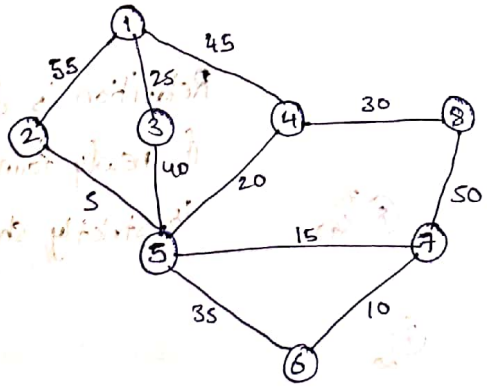
↳ cost of shortest paths.

Consider finding shortest path from vertex 6

Vertex selected	d-values					
	1	2	3	4	5	6
{6}	∞	∞	∞	∞	(3)	-
{6,5}	∞	(23)	38	∞	(3)	-
{6,5,2}	∞	(23)	(33)	38	(3)	-
{6,5,2,3}	∞	(23)	(33)	(38)	(3)	-
{6,5,2,3,4}	58	23	33	28	3	-

The limitation of this matrix form is that it shows only the cost of the shortest path but not the path.

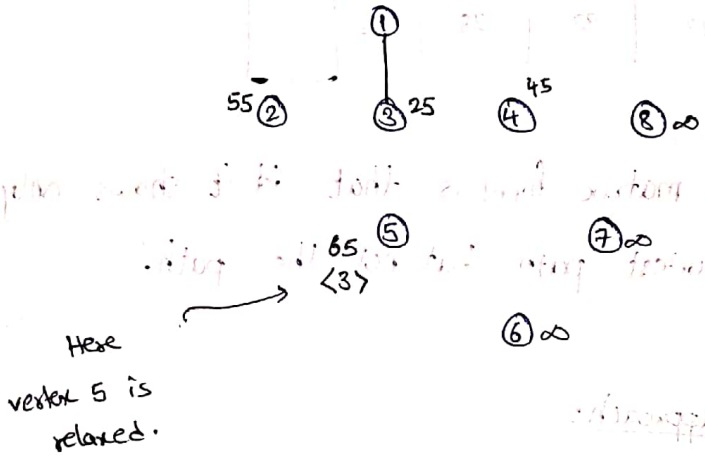
(ii) Spanning tree approach:



Consider finding shortest path from vertex 1.

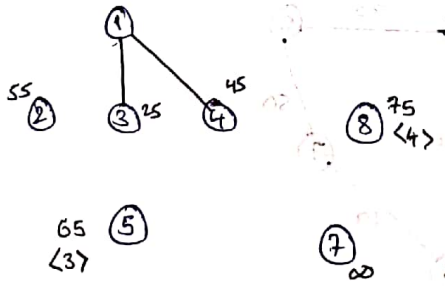
∴ write the edges of all nodes from node 1. as shown in the figure.

Since '3' is nearsest so far, choose vertex 3



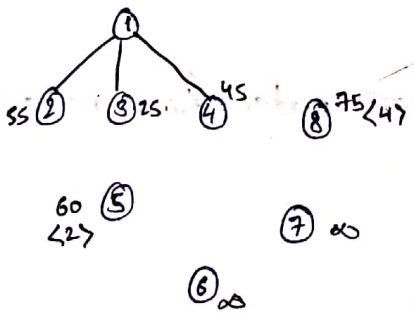
Here vertex 5 is relaxed.

Now choose vertex 4

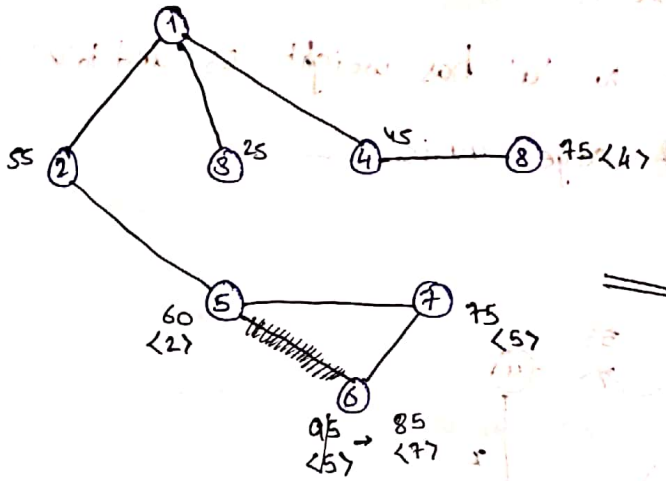


Relaxition is done if newly found path is strictly shorter.

Choose vertex '2'



Choose vertex 5 (via 2)



⟹ Single Source Shortest Paths Spanning Tree.

∴ Final costs are

1	2	3	4	5	6	7	8
-	55	25	45	60	85	75	75

1-2 1-3 1-4 1-2-5 1-2-5-7-6 1-2-5-7 1-4-8

Note:

→ MCST and single source shortest paths spanning tree need not to be same.

→ The working/implementation of dijkstra's algorithm is similar to prim's algorithm.

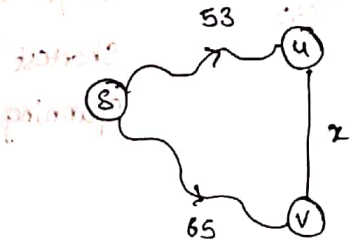
Time Complexity

- without heap: $O(n^2)$
- with heap: $O((n+e)\log n)$

∴

Q) Consider weighted undirected graph. Let uv be an edge in the graph. It is known that the shortest path from the source vertex 's' to 'u' has weight 53 and to 'v' is 65. Then weight of edge uv is ——

Sol:

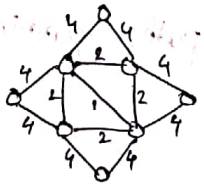


$\Rightarrow 53 + x \geq 65$

$\Rightarrow x \geq 12$

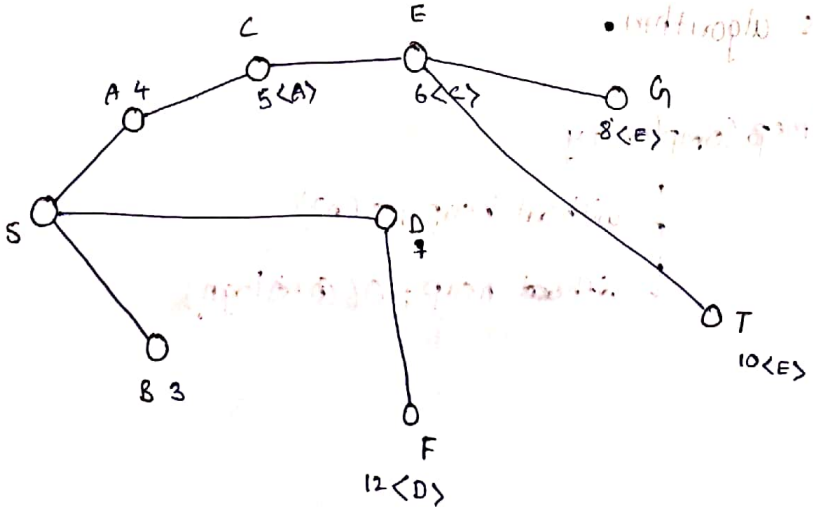
H/49

Q) Find no of MCST possible for below graph



Ans: 64

H/49



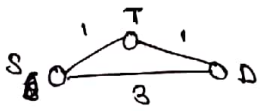
∴ SACET

(H/50)

1. True (∵ doing so will inc cost of every spanning tree by $(n-1)c$) ↳ increased cost.

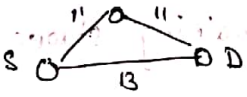
2. False

Eg:



shortest path from S to D is S-T-D

inc weight by 10

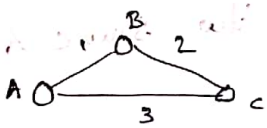


Shortest path : S-D

(H/52)

I. True (concept)

II.



from A to C we have 2 shortest paths

i.e., A - B - C

A - C

∴ False

Dynamic Programming:

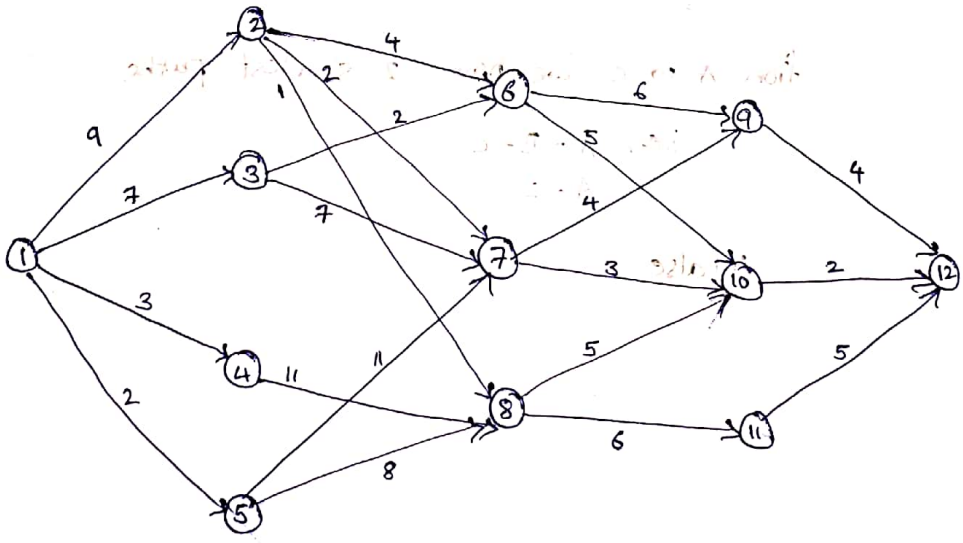
↳ Here programming means act of tabulating the results of subproblems.

Multistage Graph:

→ A k-stage multistage graph has k stages where 1st & last stages have exactly one vertex and remaining stages have atleast one vertex.

→ Every edge is from ~~stage i to~~ a vertex of stage i to a vertex of stage i+1

→ The problem is to find distance btw source & destination



Applying greedy method (choosing least cost at every stage)

1-5-8-10-12 ⇒ 17 ↳ so using greedy method for single pair shortest path may fail

Applying Dijkstra's algorithm

1-3-6-10-12 ⇒ 16 ↳ greedy method works when we need to find shortest paths to all vertices.

→ DP is an algorithm design technique used for solving problems whose solutions are viewed as a result of making a set decisions.

→ One way of solving these problems is by making decisions at every step based on local information available at that step. This is true for problems solved by greedy method.

→ But for many problems, like multistage graphs, optimal solutions cannot be acheived by making decisions based on local information in a stepwise manner.

→ one way of solving the problems for which optimal solutions cannot be made based on local information, is to enumerate all possible decision sequences and pick up the best. This is known as brute force approach (enumeration strategy). However for this strategy, the drawback is that it has very large time & space complexities.

→ DP is based on enumeration, but it often tries to curtail the amount of enumeration by removing/avoiding those sequences that are not feasible or sub optimal.

Due this there may be a significant improvement on time complexities.

→ DP makes the sequence of decisions by applying principle of optimality. (global optimality) (Optimality substructure)

→ principle of optimality states that whatever the initial state and decision are, the remaining sequence of decision must be optimal with regard to state resulting from the current decision.

→ Any problem that obeys principle of optimality and overlapping subproblems can be solved by DP.

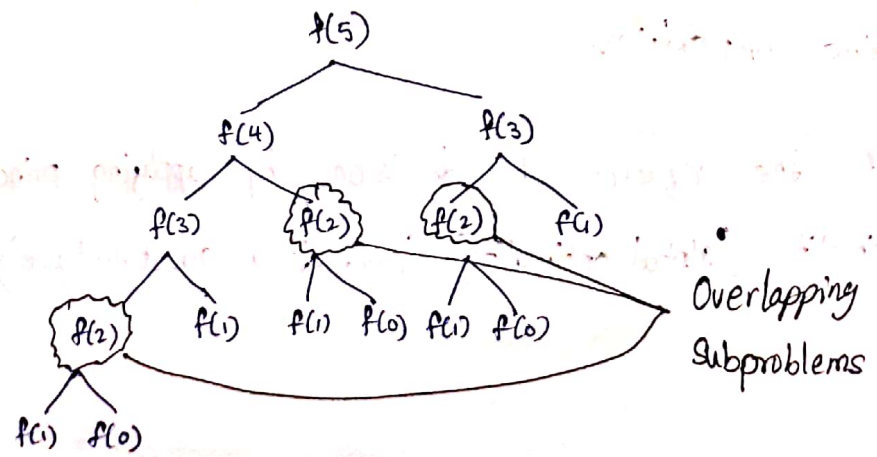
→ The fundamental difference b/w greedy method and DP is that GM generates only one decision sequence whereas DP may generate multiple decision sequences.

→ In DP, the results of subproblems can be tabulated and can be reused. (Memoization)

→ Thus DP optimizes time by curtailing decision that lead to non-feasible are sub optimal soln and by tabulated results of subproblems.

Eg: Consider computing nth Fibonacci number

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

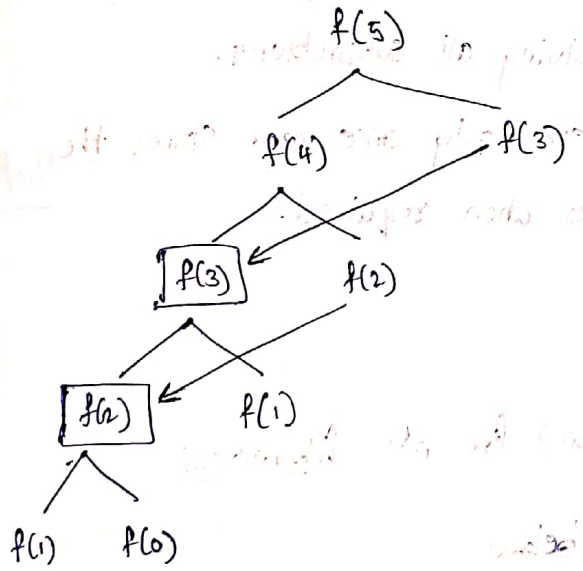


→ Regular recursive implementation

```
int fib(n)
{
    if(n<=1) return n;
    else
        return fib(n-1)+fib(n-2);
}
```

TC: $O(2^n)$. Space. Comp: $O(n)$

→ Top down DP with memoization:



Rather than tree, beare a closed graph is formed without cycle.

→ DP problems can be solved in 2 ways:

Top down approach with memoization (implemented with recursion)

Bottom up approach with tabulation (Non-recursive)
→ Generally this is more efficient in terms of T.C

Note:

Greedy: Builds up a solution increamentally by optimizing at each step some local criteria.

Dfc: Break up a problem into seperate subproblems and solve each subproblem independently and combining the solutions to subproblems to get soln of original problem.

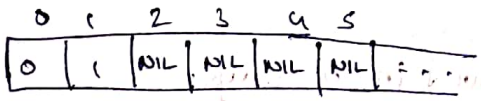
DP: Break up a problem into a series of overlapping subproblems and buildup solns to larger and larger ~~sup~~ subproblems.

- Unlike ~~Unlike~~ Dfc, DP involves solving all subproblems.
- DP solves each subproblem only once and stores the results & uses them later when required.

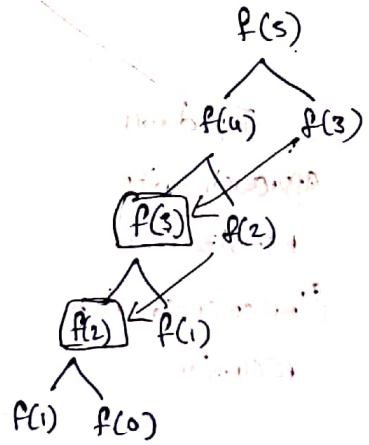
13/10/20

Top down approach (Memoization) for nth fibanocci:

initialize arry f[ ] as shown below




```
int fib(n)
{
    if(f[n]==NIL)
        f[n]=fib(n-1)+fib(n-2)
    return (f[n]);
}
```



TC: O(n)

∵ for every number fib(n) is computed only once.

SC: O(n)

i.e., size(array) + size(stack)

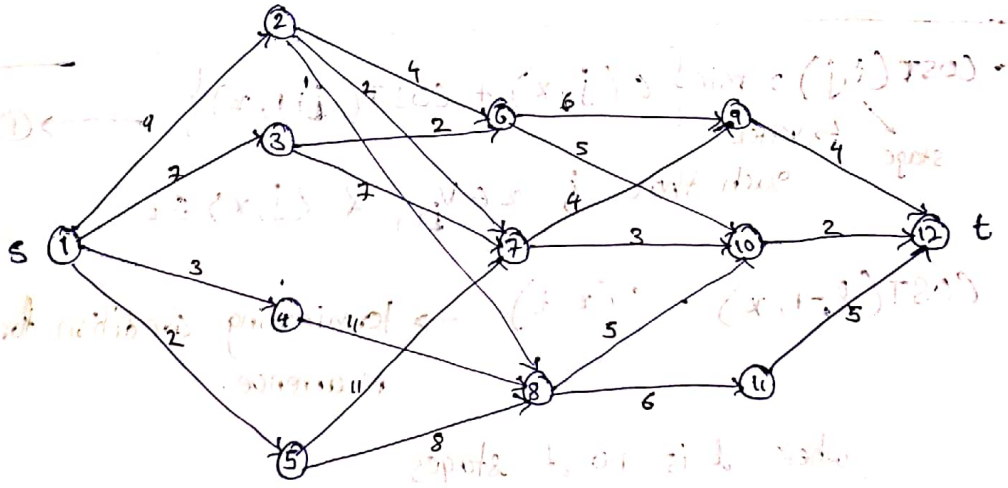
Bottom up approach (Tabulation) for nth fibonacci number

```
f[1000]: int array.
f[0]=0; f[1]=1;
int fib(n).
{
    for i=2 to n
        f[i] = f[i-1]+f[i-2];
    return (f[i]);
}
```

TC: O(n)
SC: O(n) [i.e., size of array]

Multistage Graph

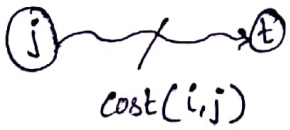
V1 V2 V3 V4 V5 ... stages



Let C[1..n, 1..n] represent cost adjacency matrix.

Let cost(i,j) represent cost of path from vertex 'j' that is present in stage i to the destination vertex 't'.

i.e., stage i



cost(i,j)

So the target is to find cost(1,1).

from ① we go to vertex x such that $x \in V_2$ & $\langle 1,x \rangle \in E$

↓

i.e., there are edge blw 1 & x.

we choose this x such that (c(1,x) + remaing cost of path from x to t) is minimum.

i.e., c(1,x) + cost(x,t) is minimum

∴ $COST(1,1) = \min\{c(1,x) + COST(2,x)\}$

such that $x \in V_2$ & $\langle 1,x \rangle \in E$

∴ In general

$COST(i,j) = \min\{c(j,x) + COST(i+1,x)\}$ ——→ ①

↓ stage ↳ vertex

such that $x \in V_{i+1}$ & $\langle j,x \rangle \in E$

optimal substructure property

$COST(l-1,x) = c(x,t)$ → terminating condition for the recurrence.

where l is no of stages

$D(i,j) = x'$

such 'x' is the value which minizes COST(i,j) in eq①

↳ used to obtain the vertices in the shortest path.

$COST(1,1) = \min\{\ 9+COST(2,2),\ 7+COST(2,3),\ 3+COST(2,4),\ 2+COST(2,5)\}$

$2 = \{2,3,4,5\}$

so we start from back

$Cost(4,9) = c(9,t) = 4$

$Cost(4,10) = c(10,t) = 2$

$Cost(4,11) = c(11,t) = 5$

Now we compute costs of stage 3 vertices

$COST(3,6) = \min\{c(6,9)+COST(3+1,9),\ c(6,10)+COST(3+1,10)\}$

$= \min\{6+COST(4,9),\ 5+COST(4,10)\}$

$= \min\{6+4,\ 6+2\} = 7$

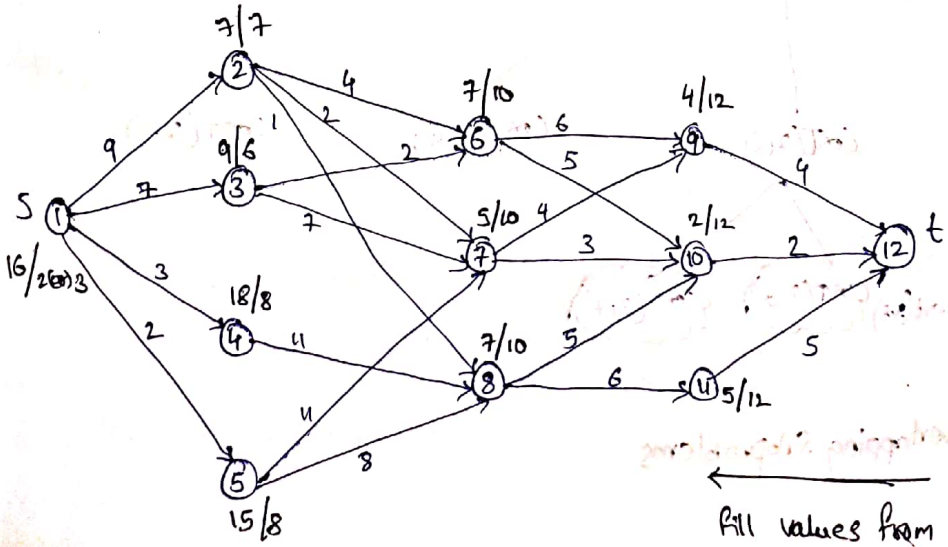
also $D(3,6) = 10$

↳ the vertex which gives shortest path from vertex 6.

$COST(3,7) = 5$

$D(3,7) = 10$

} These value can be found directly from the graph



∴ Min cost path's cost = 16

path can be found by using array D.

From graph we find path using the labels assigned to each vertex

∴ path: 1 - 2 - 7 - 10 - 12

1 - 3 - 6 - 10 - 12

Formal way for path construction:

For an l-stage graph we need to make l-2 decisions.

path: 1 -

D(1,1) = 2

path: 1 - 2 -

↑ 1st decision

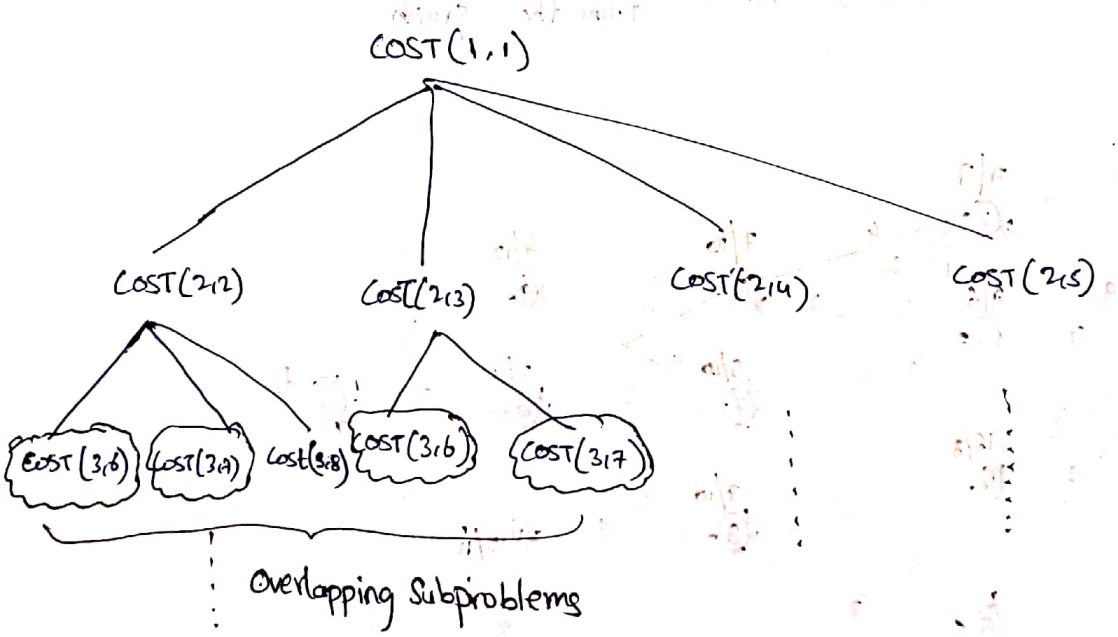
D(2, D(1,1)) = D(2,2) = 7

path: 1 - 2 - 7 -

↑ 2nd decision

D(3, D(2, D(1,1))) = D(3, D(2,2)) = D(3,7) = 10

path: 1 - 2 - 7 - 10 - 12

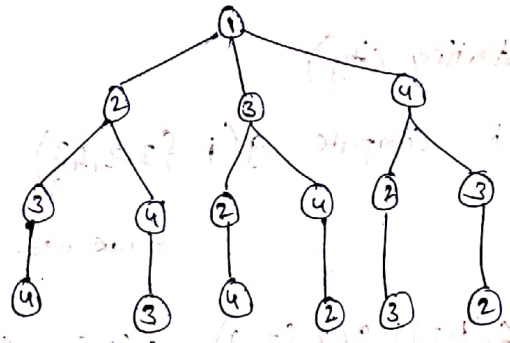


Travelling Salesman Problem

Problem Defn:

The TSP has to organize a tour in such a way that all the cities (except the starting city) must be visited exactly once and return to the home (starting city) with an objective of minimizing the cost of the tour.

⑧ Thus if there are 4 cities (including starting city) brute force approach will be



$\longrightarrow (n-1)!$

$\therefore$ TC of brute force $= (n-1)! = O(n^n)$

The path is like 1 —— x —— 1

optimality substructure.

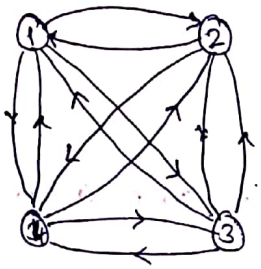
From tree, we can show overlapping subproblems.

Recurrence formula for TSP:

Let $c[i,j]$ be a cost adjacent matrix

let $g(i,s)$ represent cost of the tour from vertex i, visiting all the vertices in the set s exactly once and terminating the tour at v_0 (home city)

Consider



Let cost adjacency matrix C be

C	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

for $v_0 = 1$ (starting city)

we need to compute $g(1, \{2,3,4\})$

↳ we need to minimize this.

$\therefore g(1,\{2,3,4\}) = \min\{c(1,2)+g(2,\{3,4\}),\ c(1,3)+g(3,\{2,4\}),\ c(1,4)+g(4,\{2,3\})\}$

∴ In general,

$$g(i,S) = \min\{c(i,x)+g(x,S-\{x\})\}$$

such that $x \in S$ & $\langle i,x\rangle \in E$

$g(i,S) = c(i,v_0)$ if $S=\phi$ → termination condition for recursion.

↓ $i \neq v_0$

we use $J(i,S)$ to compute the path

$J(i,S) = x$

Now let us calculate $g(1,\{2,3,4\})$

$|S|=0$:

$g(2,\phi)=5$

$g(3,\phi)=6$

$g(4,\phi)=8$

$|S|=1$:

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

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

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

$g(3,\{4\}) = 12+8=20$, $J(3,\{4\})=4$

$g(4,\{2\}) = 8+5=13$, $J(4,\{2\})=2$

$g(4,\{3\}) = 9+6=15$ $J(4,\{3\})=3$

$|S|=2$:

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

$= \min\{9+20, 10+15\}$

$= 25$

$J(2,\{3,4\})=4$

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

$= 25$

$\Rightarrow J(3,\{2,4\})=4$

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

$= 23$

$J(4,\{2,3\})=2$

$|S| = 1$:

$g(1, \{2,3,4\})$

$= \min\{ c(1,2) + g(2,\{3,4\}),\ c(1,3) + g(3,\{2,4\}),$

$c(1,4) + g(4,\{2,3\})\}$

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

$= 35$

~~$J(1,\{3,3$~~ $J(1,\{2,3,4\}) = 2$

Path construction :

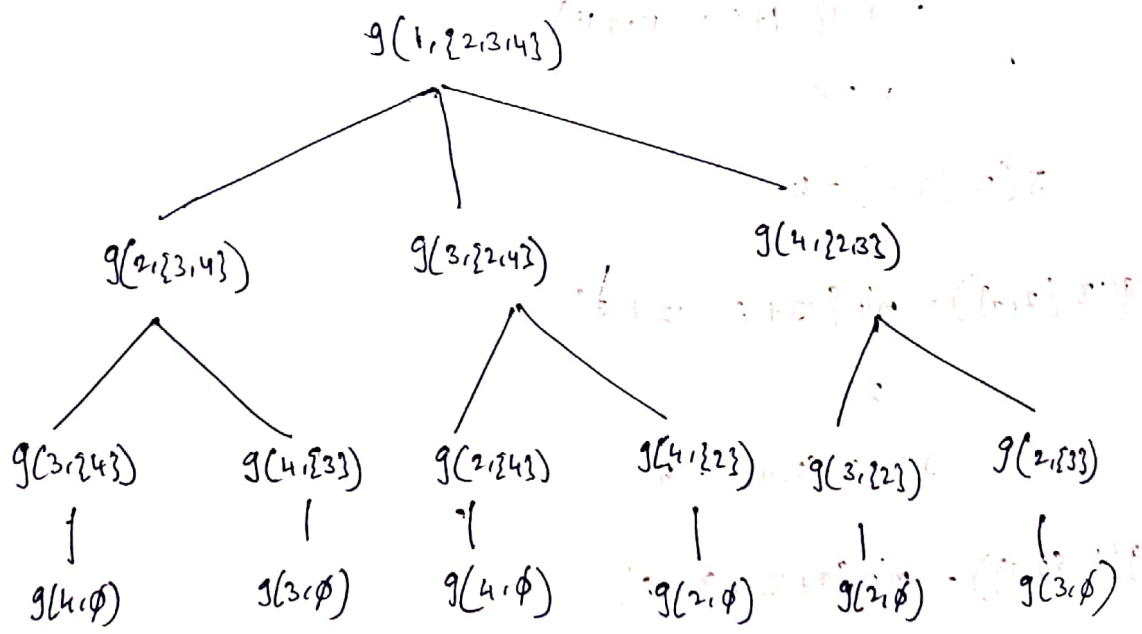
If $J(i,s) = x$ then from x we go to $J(x, S-x)$

$J(1,\{2,3,4\}) = 2$

$J(2,\{3,4\}) = 4$

$J(4,\{3\}) = 3$

$\therefore$ path : 1 - 2 - 4 - 3 - 1



For larger problems we can see overlapping subproblems.

→ The algorithm for TSP is Held & Carp Algorithm

Time Complexity : $O(n^2 \cdot 2^n)$

∵ we have 2^n subsets and for every subsets we need to check for possible edges (max possible edges = n^2)

Space complexity : $O(n \cdot 2^n)$

↳ To store values of g

because for every possible subset S and for every vertex except start vertex we calculate and store

$g(v, s)$ ∴ $O(n \cdot 2^n)$

n 2^n

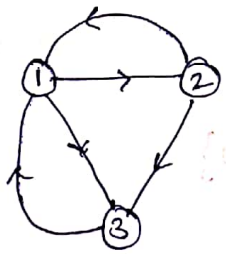
→ TSP is an intractable Problem

∵ it doesn't have a polynomical TC algorithm so far.

∴ TSP is an NP problem (NP-complete).

16/10/20

All Pairs Shortest Paths (Flyod Warshall's Algorithm)



c	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

$c(i,j) \longrightarrow$ edge cost

Let $A^k(i,j)$ represent the cost of the path from vertex 'i' to vertex 'j' NOT going through the intermediate vertex greater than k.

Assume vertices are sequentially numbered from 1 to n $\langle 1,2,3,\ldots n \rangle$

$\therefore A^k(i,j) = \min\{$ path goes through k, path doesn't go through k$\}$

If path goes through k

right now here highest intermediate vertex possible = k-1

$\therefore$ If path goes through k, it will be

$$A^{k-1}(i,k) + A^{k-1}(k,j)$$

$$\therefore A^k(i,j) = \min\{A^{k-1}(i,k) + A^{k-1}(k,j),\ A^{k-1}(i,j)\}, \quad k \neq 0$$

$$A^0(i,j) = c(i,j)$$

Eg:

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

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

$= 4$

$A^1(1,3) = 11 \quad (1 \to 1 \to 2)$

$A^1(2,1) = 6 \quad (2 \to 1 \to 1)$

$A^1(2,3) = \min\{A^0(2,1) + A^0(1,3),\ A^0(2,3)\} = \min\{17, 2\} = 2$

$A^1(3,1) = 3$

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

$A^1(3,3) = 0$

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

$\Rightarrow$

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

Algo Floyd-Warshall (G, n, e, c, A)

// c[1..n, 1..n] --- cost adjacency matrix

// A[1..n, 1..n] --- path cost adjacency matrix

```
{
1. for i ← 1 to n
       for j ← 1 to n
           A[i,j] = c[i,j];

2. for k ← 1 to n
       for i ← 1 to n
           for j ← 1 to n
               A[i,j] = min{A[i,j], A[i,k] + A[k,j]}
}
```

TC: $O(n^3)$

SC: $O(n^2)$

~~→ The transi~~

Transitive closure:

→ Let A be the adjacency matrix of a directed / undirected graph. The transitive closure A^* of A be a matrix with the property that $A^*(i,j) = 1$, iff G has a directed path / path containing atleast atleast one edge from vertex i to vertex j, otherwise $A^*(i,j) = 0$.

→ Thus, transitive closure of a matrix, represent graph with n vertices, can be computed in $O(n^3)$ time.

Reflexive Transitive Closure:

Let 'A' be adjacency matrix of graph G and matrix A^+ be reflexive transitive closure of graph G defined as.

$A^+(i,j) = 1$ iff G has a path containing 0 or more edges from i to j

$A^+(i,j) = 0$, otherwise

0/1 knapsack:

knapsack capacity : m

n-object: $\langle o_1, o_2, \ldots o_n \rangle$

weight (w_i) — Profit (P_i) — Decision (x_i)

Explicit Constraint : $\sum w_i \leq m$

objective func: Maximize the total profit ~~that we~~ subjected to the condition that total weight being put into the knapsack doesnot exceed its capacity.

i.e., Maximize $\sum_{i=1}^{n} P_i x_i$ subjected to $\sum_{i=1}^{n} w_i x_i \leq M$

where $x_i = 0/1$

Approach I:

Let $f_n(m)$ represent the profit with n-objects and a knapsack capacity of 'M'.

$f_n(m) = \max\{$ profit by including the object, profit by excluding the object $\}$

$$\therefore f_n(m) = \max\left\{ \underset{(x_n=1)}{P_n + f_{n-1}(m-w_n)}, \underset{(x_n=0)}{f_{n-1}(m)} \right\}$$

$f_0(x) = 0$;

Eg: $n=3$; $M=6$; $\langle P_1, P_2, P_3 \rangle = \langle 1, 2, 5 \rangle$

$\langle w_1, w_2, w_3 \rangle = \langle 2, 3, 4 \rangle$

$\langle x_1, x_2, x_3 \rangle = ?$

Tuple method:

Let f_n be represented by $S^n = \{(P, w)\}$

$P \rightarrow$ Profit

$w \rightarrow$ weight put into the knapsack.

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

S_i^1 is obtained by adding $P_i \& w_i$ to S^0

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

Now we merge S^0 & S_1^1 to obtain S^1.

$$S^1 = \{\underset{x_1=0}{(0,0)}, \underset{x_1=1}{(1,2)}\}$$

S_1^2 is obtained by ~~action~~ adding (P_2, w_2) to S^1

$$S_1^2 = \{(2,3)(3,5)\}$$

S^1 $\quad$ $x_1 x_2$: 00 $\quad$ 01 $\quad$ 10 $\quad$ 11

$S_1^2 \rightarrow S^2 = \{(0,0)\ (1,2)\ (2,3)\ (3,5)\}$

Now

$S_1^3 = \{(5,4)\ (6,6)\}$ $\quad$ not feasible ∴ not to add (5,4) $\quad$ not feasible to add (5,4)

S^2

$S_1^3 \rightarrow S^3 = \{(0,0)\ (1,2)\ (2,3)\ (5,4)\ (6,6)\}$

↳ Here we have discarded (3,5) since (5,4) gives max. profit with less weight.
This is called Purging rule.
∵ (3,5) will never lead to optimal soln.

Purging rule:

Let (P_i, w_i) & (P_j, w_j) be tuple to be merged
if $(P_i < P_j)$ and $(w_i > w_j)$ then the tuple (P_i, w_i) may be purged (removed).

Computing values of x_i's:

we got (6,6) as answer

$(6,6) \in S^3$ but $(6,6) \notin S^2$

⇒ we added O_3 i.e., (5,4) is added

$\therefore x_3 = 1$

$(6,6) - (5,4) = (1,2)$

Now $(1,2) \in S_2^2$ & $(1,2) \in S_1^1$ & $(1,2) \notin S^0$

$\Rightarrow x_1 = 1$

$x_2 = 0$

$\therefore (x_1 x_2 x_3) = (1,0,1)$

∴ If $(P,w) \in S^i$ & $(P,w) \notin S^{i-1}$ then $x_i = 1$ & $(P,w) \leftarrow (P-P_i, w-w_i)$

if $(P,w) \in S^i$ & $(P,w) \in S^{i-1}$ then $x_i = 0$

Eg: $n=4$; $\langle P_1, P_2, P_3, P_4 \rangle = \langle 60, 28, 20, 24 \rangle$

$M = 11$; $\langle w_1, w_2, w_3, w_4 \rangle = \langle 10, 7, 4, 2 \rangle$

I. Greedy fractional knapsack:

$P_i/w_i : \langle 6, 4, 5, 12 \rangle$

∴ $x_4 = 1 \Rightarrow m = 11-2 = 9$

$x_1 = \frac{9}{10} = 0.9$

∴ $P = (0.9)(60) + (1)(24)$

$= 54 + 24 = 78$

II. 0/1 knapsack with greedy approach

$P_i/w_i : \langle 6, 4, 5, 12 \rangle$

$x_4 = 1 \Rightarrow m = 11-2 = 9$

$w_1 = 10$ not possible to include

∴ $x_1 = 0$

$x_3 = 1 \Rightarrow m = 9-4 = 5$

$w_2 > m$ ∴ $x_2 = 0$

∴ $P = 0 + 20 + 24 = 44$

III · 0/1 knapsack using DP

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

$S^1_1 = \{(60,10)\}$

$S^1 = \{(0,0)\ (60,10)\}$

$S^2_1 = \{(28,7)\}$ — Merge → $S^2 = \{(0,0)\ (28,7)\ (60,10)\}$

$S^3_1 = \{(20,4)\ (48,11)\}$ — (+) —

$S^3 = \{(0,0)\ (20,4)\ (28,7)\ (60,10)\}$

$S^4_1 = \{(24,2)(44,6)(52,9)\}$

$\Rightarrow S^4 = \{(0,0)(24,2)(44,6)(52,9)(60,10)\}$

$\therefore (60,10)$ is optimal

$(60,10) \in S^4$; $(60,10) \in S^3$; $(60,10) \in S^2$; $(60,10) \in S^1$; $(60,10) \in S^0$

$x_4 = 0$; $x_3 = 0$; $x_2 = 0$; $x_1 = 1$

$\therefore P = (1)(60) = \underline{60}$

Approach II : (Tabulation / bottom up approach)

KNAP → $k(n, m)$ — n: no of obj, m: knapsack capcity

$k(n,m) = k(n-1,m)$, $w[n] > m$

$= \max\{k(n-1,m),\ k(n-1, m-w_n)+P_n\}$, $w[n] \leq m$

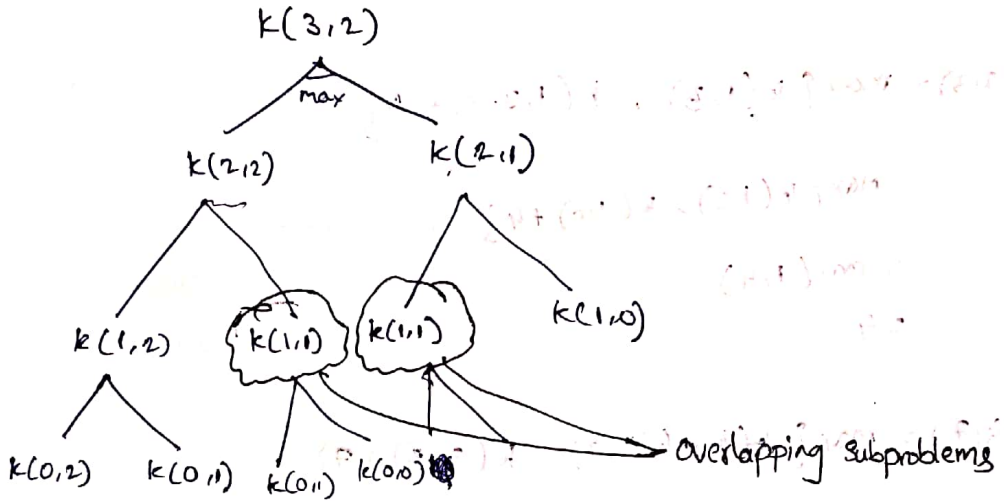
($x_n = 0$) ($x_n = 1$)

optimality substructure.

Eg: n=3; m=2;

w[ ] = {1, 4}

P[ ] = {5, 10, 15}



Boundary conditions:

$$k(0,x) = 0; \quad k(y,0) = 0;$$

Eg: m=7; n=4; $P_i = \langle 1,4,5,7 \rangle$ $w_i = \langle 1,3,4,5 \rangle$

we can solve this problem using a 2-dimensional matrix.

k[n+1, m+1]

m=7, n=4 ⇒ k[5,8]

~~Tabular~~

k	m→ 0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5	5	5	5
3	0	1	1	4	5	6	6	9
4	0	1	1	4	5	7	8	9

n ↓

The cell is k[4,7] i.e., soln

$k(1,1) = \max\{k(0,1), k(0,0)+1\} = 1$

$k(1,2) = 1$

$k(2,1) = \max\{ k(0,1), k(1,-2)\}$

(not possible)

$= k(1,1)$

$= 1$

$k(2,3) = \max\{ k(1,3), k(1,3-3)+4\}$

$= \max\{ k(1,3), k(1,0)+4\}$

$= \max(1,4)$

$= 4$

$k(4,7) = \max\{ k(3,7), k(3,2)+7\}$

$= \max\{9,8\}$

$= 9$

Calculating x_i values from table

$k(n,m)$ is obtained from either $k(n-1,m)$ cell or $k(n-1,m-w_n)$ cell.

If obtained from $k(n-1,m)$ then $k(n,m) = k(n-1,m)$

If obtained from $k(n-1,m-w_n)$ then $k(n,m) = k(n-1,m-w_n)+P_n$

∴ If $k(n,m) = k(n-1,m)$ then

$x_i = 1$ & $(n,m) \leftarrow (n-1,m)$

If $k(n,m) \neq k(n-1,m)$ then

$x_i = 0$ & $(n,m) \leftarrow (n-1, m-w_n)$

and we perform this recursively.

```
Algo KNAP(n, m, w[], P[])
{
    declare k[n+1, m+1];
    for i ← 0 to n
        for j ← 0 to m
            if (i == 0 or j == 0)
                k[i, j] = 0
            else if (w[i] ≤ j) then
                k[i, j] = max{ k[i-1, j], k[i-1, j-w[i]] + Pi }
            else
                k[i, j] = k[i-1, j];
}
```

→ Time Complexity : $O(n*m)$

→ Time complexity with brute force : $O(2^n)$

However for large values of m, $O(n*m)$ is too high.

For example if $m = 2^n$

then TC of tabulation method = $O(n \cdot 2^n)$

and in this case brute force would be better.

Thus tabulation method is good for smaller values of n.

→ Space Complexity with tabulation method : $O(nm)$

17/10/20

Longest Common Subsequence:

Substring : grp of one or more characters taken from the string that are contiguous :

Subsequence : group of one or more characters may not be contiguors nevertheless the relative order is same.

→ Every substring is a subsequence

→ For a string of length n,

no of substrings → $O(n^2)$

no of subsequences → $O(2^n)$

→ A subsequence that is common to given two string is called common subsequence.

The longest of all these common subsequences is called longest common subsequence.

Applications of LCS :

(i) Web search

(ii) DNA matching

(iii) Software matching

(iv) Plagiarism

Solution :

Given two strings X & Y of length 'n' & 'm' respectively. It is required to determine a subsequence of longest length that is common to both X & Y

$X = \langle x_1 x_2 \dots x_n \rangle$ $Y = \langle y_1 y_2 y_3 \dots y_m \rangle$

→ Let 'i' & 'j' be indices into the strings X & Y as shown.

→ Let L[i,j] represent the length of common subsequence of string X & Y.

Case I : Last characters are same

Eg: X = <G T T C C T A A T A>
0 1 2 3 4 5 6 7 8 9

Y = <C G A T A A T T G A G A>
0 1 2 3 4 5 6 7 8 9 10 11

so we need to find L[9,11].

if X[i] = Y[j] then

$$L[i,j] = 1 + L[i-1, j-1]$$

Case II : Last characters don't match

X = <G T T C C T A A T A>

Y = <C G A T A A T T G A G>

If x[i] ≠ Y[j]

∴ This means that LCS may terminate with last character of X or last character of Y or neither.

i.e., $L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$

∴ if x[i] = Y[j]

$$L[i,j] = 1 + L[i-1, j-1]$$

if x[i] ≠ Y[j]

$$L[i,j] = \max\{L[i-1,j], L[i,j-1]\}$$

$$L[-1,j] = 0$$

$$L[i,-1] = 0$$

Tree approach to solve LCS:

X = ⟨ABBAB⟩ Y = ⟨ACBAB⟩

("ABAB")

↑ u ("ABB")

LCS("ABBAB", "ACBAB")

("ABB") = ↖ (i+)(B)

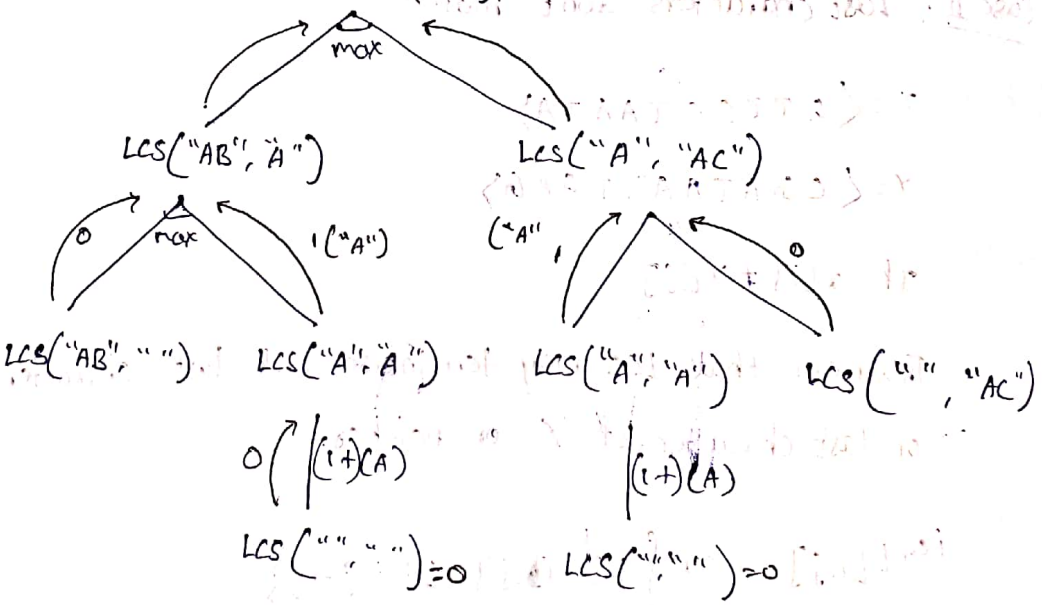
LCS("ABBA", "ACBA")

("AB") = ↖ (i+)(A)

LCS("ABB", "ACB")

("A") = ↖ (i+)(B)

LCS("AB", "AC")



Tabulation method (bottom up):

X = ⟨A B C B D A B⟩ (0 1 2 3 4 5 6)

Y = ⟨B D C A B A⟩ (0 1 2 3 4 5)

		-1	0 (B)	1 (D)	2 (C)	3 (A)	4 (B)	5 (A)
-1		0	0	0	0	0	0	0
0	A	0	0 ←	0 ←	0 ←	↖ 1	← 1	↖ 1
1	(B)	0	↖ 1	1 ←	← 1	← 1	↖ 2	← 2
2	C	0	↑ 1	← 1	↖ 2	← 2	← 2	← 2
3	B	0	↑ 1	← 1	↑ 2	← 2	↖ 3	← 3
4	(D)	0	↑ 1	↖ 2	← 2	← 2	↑ 3	← 3
5	(A)	0	↑ 1	↑ 2	← 2	↖ 3	← 3	↖ 4
6	(B)	0	↖ 1	↑ 2	← 2	↑ 3	↖ 4	← 4

→ This is the soln i.e., L[6,5]

$L[0,0] = \max\{L[-1,0], L[0,-1]\} = 0$

$L[0,1] = \max\{L[-1,1], L[0,0]\} = 0$

If case "II" applies then we take max {upper cell, left side cell}

If case I applies we take 1 + upper left diagonal cell's value.

$L[0,3] = 1 + L[-1,2] = 1 + 0 = 1$

$\therefore L[6,5] = 4$

To get the subsequence we backtrack while backtracking add corresponding symbol whenever you move diagonally upward.

$\therefore$ LCS: B D A B

```
Algorithm LCS (x, y, n, m)
L[n+1, m+1] : array
{
  1. for i ← 0 to n-1
       ~~for j ← 0 to m-1~~
         L[i, -1] = 0
  2. for j ← 0 to m-1
       L[-1, j] = 0;
  3. for i ← 0 to n-1
       for j ← 0 to m-1
         if(x[i] = y[i])
           L[i, j] = 1 + L[i-1, j-1];
         else
           L[i, j] = max{L[i-1, j], L[i, j-1]}
}
```

T.C: $O(n * m)$

S.C: $O(n * m)$

TC of LCS with brute force is $O(2^n \cdot m)$ or $O(n \cdot 2^m)$

Matrix Chain Product:

→ Given two square matrices of order n

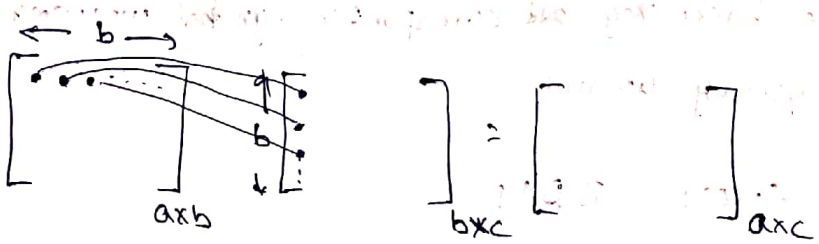
time to multiply: $O(n^3)$

operation: scalar multiplication

i.e., for product of two square matrices n^3 multiplications are required.

→ However two multiply two non-square matrices, the matrices have to be compatible.

i.e., no of colums of 1st matrix = no of rows of 2nd matrix



we need to compute axc element and computations of each element require b scalar multiplications.

⇒ we nee to perform axbxc no of multiplications

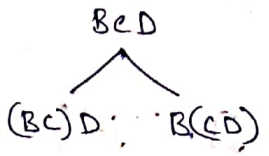
$\therefore TC = O(axbxc)$

~~Problem Statement:~~

Given a chain of matrices, it is required to multiply them to get a resultant product matrix.

- Eg: Consider ⇒ A = BCD

B: 2x10; C: 10x50; D: 50x20;



(BC)D ⟨ BC : 2×10×50 = 1000 ; (BC)D : 2×50×20 = 2000 } = 3000

(BC) D
2×50 50×20

B(CD) ⟨ CD : 10×50×20 : 10000 ; B(CD) : 2×10×20 : 400 } = 10400

Objective function:

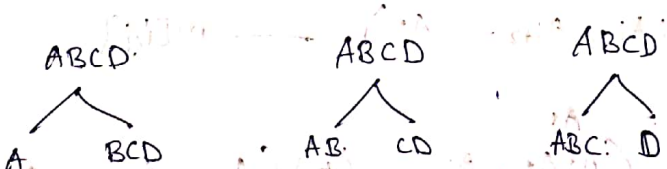
Minimizing the number of scalar multiplications.

Problem stmt:

Given a chain of matrices $\langle A_1 A_2 A_3 \ldots A_n \rangle$ where matrix A_i is of order $P_{i-1} \times P_i$. It is required to fully parantesize the given chain of matrices such that the no of scalar multiplications is minimum.

ABC can be paranthesized in 2 ways
i.e., (AB)C & A(BC)

ABCD can be paranthesized in 5 way



∴ (A(B(CD))) ((AB)(CD)) ((AB)C)D

A((BC)D) (A(BC))D

→ Observing this we can say brute force method will have exponential time complexity.

Applications

(i) In mathematics
(ii) Satellite communication
(iii) Physics

Eg: $A_1 \rightarrow 3 \times 5 \longrightarrow P_0 \times P_1$

$A_2 \rightarrow 5 \times 8 \longrightarrow P_1 \times P_2$

$A_3 \rightarrow 8 \times 6 \longrightarrow P_2 \times P_3$

$A_4 \rightarrow 6 \times 2 \longrightarrow P_3 \times P_4$

$A_5 \rightarrow 2 \times 3 \longrightarrow P_4 \times P_5$

$A_6 \rightarrow 3 \times 7 \longrightarrow P_5 \times P_6$

$\langle A_1 A_2 \cdots A_6 \rangle \longrightarrow P_0 \times P_6$

→ Let $A_{i..j}$ be the matrix that results from multiplication of matrices $\langle A_i A_{i+1} A_{i+2} \cdots A_j \rangle$

Eg: $A_{1..6} = \langle A_1 A_2 A_3 A_4 A_5 A_6 \rangle$

→ Let $m[i,j]$ be the no of scalar multiplications needed to get the matrix $A_{i..j}$

→ Any optimal parenthesization must split the given chain of matrix about matrix A_k such that the no of scalar multiplications is minimum

$(A_i A_{i+1} A_{i+2} \cdots A_j) \longrightarrow m[i,j]$

$= (A_i A_{i+1} A_{i+2} \cdots A_k)(A_{k+1} A_{k+2} \cdots A_j)$

$P_{i-1} \times P_k \qquad P_k \times P_j$

Here k value ranges from i to j-1
i.e., j-i splits are possible.

∴ If split at A_k

$\therefore m[i,j] = m[i,k] + m[k+1,j] + (P_{i-1} * P_k * P_j)$

$$\therefore m[i,j] = \min_{i \le k < j} \{ m[i,j] + m[k+1,j] + (P_{i-1} * P_k * P_j) \}$$

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

Tabulation method: (bottom up)

Eg: $A_1 \rightarrow 3 \times 5$

$A_2 \rightarrow 5 \times 8$

$A_3 \rightarrow 8 \times 6$

$A_4 \rightarrow 6 \times 2$

So we need to compute $m[1,4]$

So in tabulation method ie compute

j-i=0

$m[1,1] = m[2,2] = m[3,3] = m[4,4] = 0$

j-i=1

$m[1,2] = 3 \times 5 \times 8$

$m[2,3] = 5 \times 8 \times 6$

$m[3,4] = 8 \times 6 \times 2$

j-i=2

$m[1,3] = \min\{ m[1,2] + m[3,3] + 3*8*6 ,\ m[1,1] + m[2,3] + 3*5*8 \}$

Silly compute $m[2,3]$

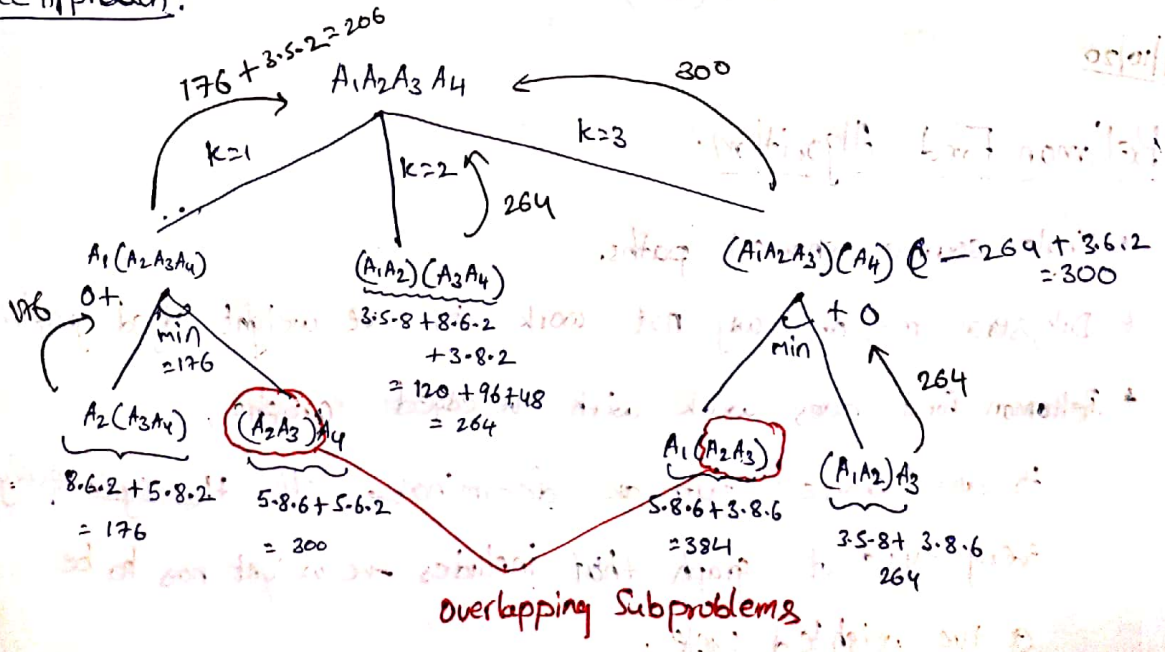
j-i=3

$m[1,4] = \min\{ m[1,1] + m[2,4] + 3*5*2,$

$m[1,2] + m[3,4] + 3*8*2,$

$m[1,3] + m[3,4] + 3*6*2 \}$

Tree Approach:



∴ Minimum cost = 206

optimal paranthesization = $(A_1 (A_2(A_3 A_4)))$

Given a chain of n matrices $\langle A_1, A_2 \cdots A_n \rangle$.

The time complexity to multiply them with optimal paranthesization

$= O(n^3)$

Proof:

In tabulation method

$j-i = n-1 \longrightarrow$ no of comparisions $= n-1$

no of instances $= 1$

$j-i = n-2 \longrightarrow$ no of comparisions $= n-2$

no of instances $= 2$

$j-i = n-3 \longrightarrow$ no of comparisions $= n-3$

no of instances $= 3$

⋮

no of comparisions =

$$(n-1)(1) + (n-2)(2) + \cdots + (n-k)(k) + \cdots + (n-(n-1))(n-1)$$

$= O(n^3)$

19/10/20

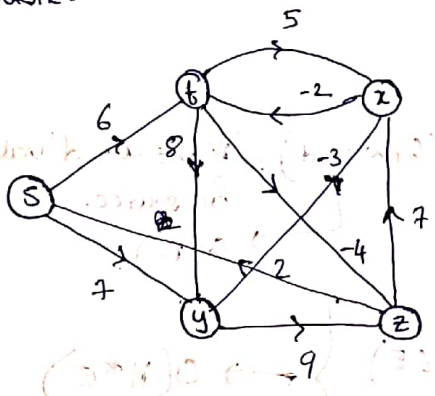
Bellman Ford Algorithm:

* Single source shortest paths.
* Dikjstra's may or many not work for -ve weight edged graph.
* Belloman Ford does work with -ve edged graphs, however, shortest path are determinable. If every every cycle of graph that includes -ve weight has to be a +ve weighted cycle.

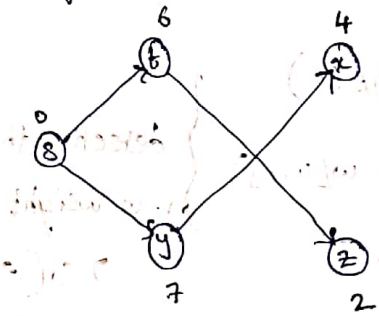
If the cycle containing negative edge is negative weighted then shortest path is not determinable.

Infact is not determinable with any algorithm.

→ The shortest path of all the vertices that are reachable through negative weighted cycle is not determinable. there could vertices that are not part of negative weighted cycle but still have their shorted path not determinable.



Using Dikjstra's algorithm



Here shortest path from s to t is reported as 6..

However the path s-y-x-t has a cost of 2.

Thus we say dijkstra ~~doesn't work~~ may or may not have correct soln for -ve weight graph.

(∵ ~~Re~~ re-relaxation of path s-t (6) is never done in dijkstra's algorithm)

Bellman ford approach:

An unrestricted shortest path blw a pair of vertices in a graph having n-vertices cannot have more than (n-1) edges assuming cycle/loop free path.

Algo BellmanFord(G, V, E, w, s)

// w[1...n][1...n] – cost adjacency matrix

// d[1...n] – result array

{

1. initialize – single_src(G, w, s, d) // Make all d value to ∞ except for source.
   → O(n)

2. for i ← 1 to (n-1)
   for (each edge (u,v) ∈ E)
   relax(u, v, w, d);
   → O(n*e)

3. for (each edge (u,v) ∈ E)
   if [d[v] > d[u] + w[u,v]]
   return FALSE
   → detecting the presence of -ve weight cycle.
   → O(e)

4. return TRUE // No -ve weight cycle reachable from source.

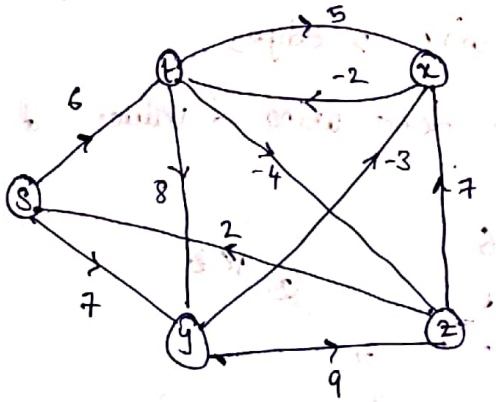
}

∴ Bellman ford algo detects -ve weight cycle if ~~exists from src~~ it is reachable from source.

→ Time Complexity: O(n*e) → adjacency list
adjacency matrix → O(n³)

→ Space Complexity: O(n)

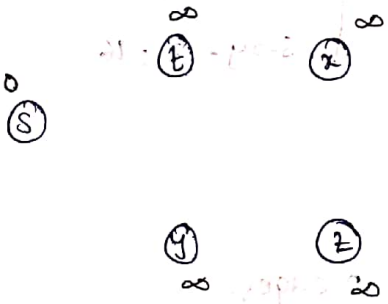
Consider the previous example again



~~Trace of point (2) in algo~~

Step 1:

Initialize d values:

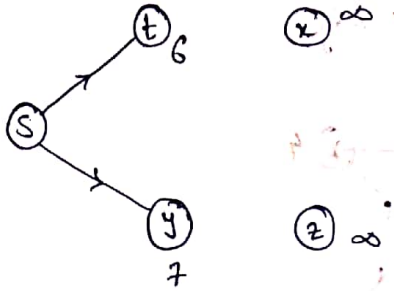


Refer Sahani for optimization on Bellmanford

Step 2: (point '2' in algo - 1st iteration) (~~edge~~ paths with 1 edge)

→ no relaxation can be done using ~~value of~~ d values of t, x, y, z (∵ it is infinite)

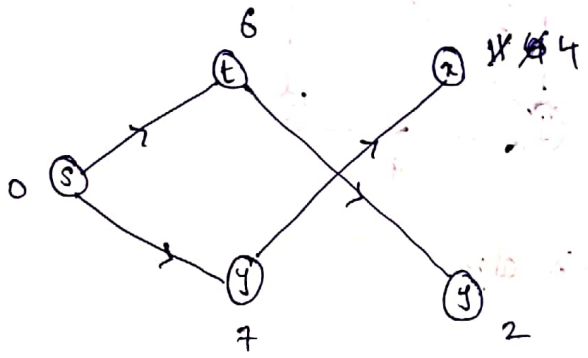
→ In this step relaxation can be done only from vertex s.



⇒ we perform this relaxation for every edge.

Step 3 :

* finding shortest paths with 2 edges
* Now relaxation can be done using d values of s, t, y

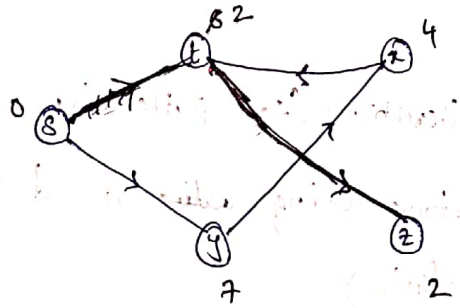


$s \to t \to x : 11$ | $s \to t \to z : 2$

$s \to y \to x : 4$ | $s \to y \to z : 16$

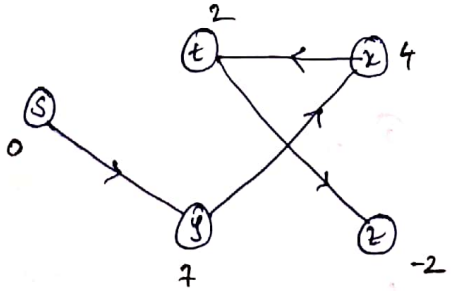
Step 4 :

→ Finding shortest paths with 3 edges.



Step 5 :

→ Find shortest paths with 4 edges



Derivation of formula:

Let c[i,j] be edge cost

d[x] = cost of the path from source 's' to vertex 'x' with atmost 'l' edges.

$$\therefore d^l[x] = \min\left\{ d^{l-1}[x],\ \min_{\forall k \in v}\{ d^{l-1}[k] + c[k,x] \} \right\}$$

$d^{l-1}[x]$ → without l-1 edges

$d^l[x] = c[s,x]$

S ..... k — x

l-1 edges, 1 edge

d[k]

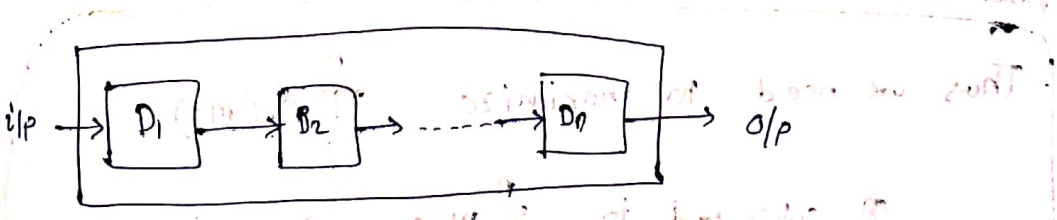
~~Reliability Design~~:

Reliable System Design

Design of an n-stage system

Here design is in cascade

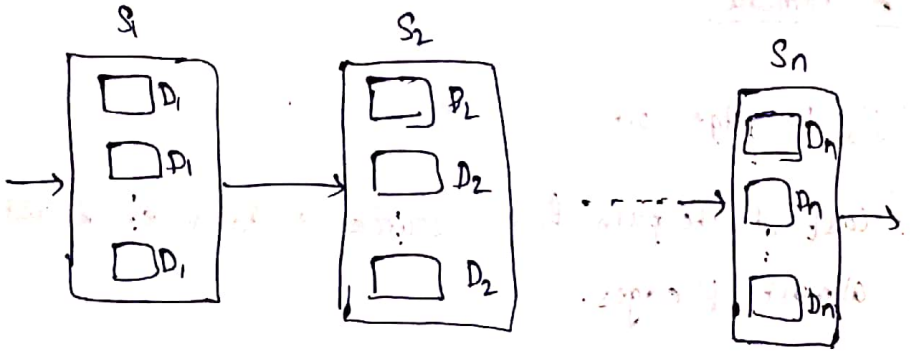
i.e., one design take o/p of another device as i/p



Every device Di has a cost c_i and reliability r_i

→ So the problem is to maximize the ~~not~~ reliability of the system such that cost of the design is less than given cost.

→ we increase reliability by adding more devices at each stage.



Let each stage 'Si' has m_i copies of device D_i such that

$1 \le m_i \le u_i$

where u_i is an upper bound.

→ Total cost of the system = $\sum_{i=1}^{n} m_i c_i$

→ Reliability of the system :

reliability of S_1

$= 1 - (1-r_1)^{m_1}$

reliability of S_2

$= 1 - (1-r_2)^{m_2}$

⋮

∴ Reliability of the design = $\prod_{i=1}^{n} \left[1 - (1-r_i)^{m_i}\right]$

Problem Defn:

∴ Thus we need to maximize $\prod_{i=1}^{n} \phi_i(m_i)$

subjected to $\sum_{i=1}^{n} m_i c_i \le C$ → given cost

where $\phi_i(m_i) = 1 - (1-r_i)^{m_i}$ & $1 \le m_i \le u_i$

$$u_i = \left\lfloor \frac{C - \sum c_j + c_i}{c_i} \right\rfloor$$

Eg: Consider 3 stage system with

$$c=105 ; \quad \langle c_1 c_2 c_3 \rangle = \langle 30, 15, 20 \rangle ; \quad \langle r_1 r_2 r_3 \rangle = \langle 0.9, 0.8, 0.5 \rangle$$

→ Let $f_n(c)$ represent reliability of an n-stage system with a total project cost of c;

$$f_n(c) = \max_{1 \le m_n \le u_n} \{ \phi_n(m_n) * f_{n-1}(c - c_n m_n) \}$$

$$f_0(x) = 1$$

Eg: $n=3$; $c=105$;

$\langle c_1 c_2 c_3 \rangle = \langle 30, 15, 20 \rangle$

$\langle r_1 r_2 r_3 \rangle = \langle 0.9, 0.8, 0.5 \rangle$

$$u_1 = \left\lfloor \frac{105 - 15 - 20}{30} \right\rfloor = 2$$

$$u_2 = \left\lfloor \frac{105 - 30 - 20}{15} \right\rfloor = 3$$

$$u_3 = \left\lfloor \frac{105 - 30 - 15}{20} \right\rfloor = 3$$

Let $f_n \sim S^n = \{(\text{reliability}, \text{cost})\}$

∴ initially we have

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

For calculation S^1 we computes S_1^1 & S_2^1

where S_j^i represents i^{th} stage & j^* copies of stage i devices.

$S_1' = \{(0.9, 30)\}$

$S_2' = \{(0.99, 60)\}$

$1-(1-0.9)^2$

$1-(0.1)^2$

$= 0.99$

using S_1' & S_2', we compute S'

$S' = \{(0.9, 30)\ (0.99, 60)\}$

Computing S^2:

we compute S^2 by computing S_1^2, S_2^2, S_3^2

$S_1^2 = \{(0.72, 45), (0.792, 75)\}$ → removed in purging rule.

$\phi_2(m_2=2) = 1-(1-0.8)^2 = 1-0.04 = 0.96$

$S_2^2 = \{(0.864, 60), (0.9504, 90)\}$

(X) ↳ from here we cannot add even 1 copy of D_3. so this path is not explored.

$\phi_2(3) = 1-(1-0.8)^3 = 1-(0.2)^3$

$= 1-0.008 = 0.992$

$S_3^2 = \{(0.8928, 75), (\quad)\}$ ↳ infeasible

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

Computing S^3:

→ First we calculate S^3_1, S^3_2, S^3_3

$S^3_1 = \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

$\phi_3(2) = 1 - (1-0.5)^2 = 1 - 0.25 = 0.75$

$S^3_2 = \{(0.54, 85), (0.648, 100), (\quad)\}$
↳ infeasible

$\phi_3(3) = 1 - (1-0.5)^3 = 1 - 0.125 = 0.875$

$S^3_3 = \{(0.63, 105)\}$
↳ eliminated using purging rule

$\therefore S^3 = \{(0.36, 65), (0.432, 80), (0.54, 80), (0.648, 100)\}$

$\therefore$ Ans: $(0.648, 100)$

Computing no of copies at each stage

$(0.648, 100)$ is present in S^3_2

$\Rightarrow m_3 = 2$

Now subtract $2 * C_3 = 40$ from 100

$100 - 40 = 60$

Now search for tuple $(-, 60)$ in S^2

i.e., $(0.864, 60)$

It is found in $S^2_2 \Rightarrow m_2 = 2$

Now search for $(-, 60-(2)(15)) = (-, 30)$ in S^1

i.e., found in S_1^1

$\therefore m_1 = 1$

$\therefore (m_1, m_2, m_3) = \langle 1, 2, 2 \rangle$

$\therefore$ reliability $= (0.9) * (1-(1-0.8)^2) * (1-(1-0.5)^2)$

$= 0.648$

Optimal Cost Binary Search Tree:

→ BST is used in compilers to maintain the keyword table. (contains all the keywords)

~~Cost of~~

Computing cost of BST:

Let T be a BST

Cost(T) = cost(successful searches) + cost(unsuccessful searches)

Cost(successful searches) $= \sum_{i=1}^{n} cost(a_i)$

where $cost(a_i)$ is cost for searching a_i in T.

$\therefore cost(a_i) \propto level(a_i)$

where $level(a_i)$ is the level of a_i in T

$cost(a_i) = P_i * level(a_i)$

where P_i is probability of using a_i in the application.

$\therefore$ Cost(succ. searches) $= \sum_{i=1}^{n} P_i * \underbrace{level(a_i)}_{\text{↳ element comparisions}}$

→ Now let us compute the cost of unsuccessful searches.

Unsuccessful searches are at null links.

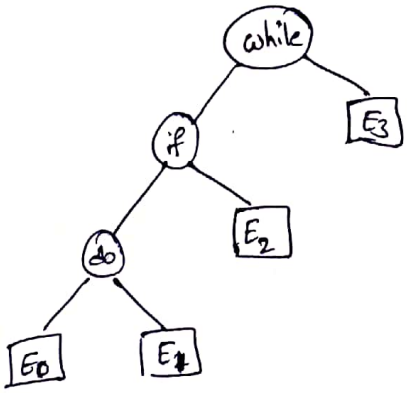
In a binary tree of 'n' nodes

total links = 2n

no of used links = n-1

∴ no of null links = 2n-(n-1) = n+1

Eg: In BST with 3 nodes we have 4 null links.



whenever there it unsuccessful search it ends at E0/E1/E2/E3

Unsuccessful search ends at E0 if search word is < do

unsuccessful search ends at Ei if search word is (< if) and (> do)

∴ cost(unsuccessful search) = $\sum_{i=0}^{n}$ cost(E_i)

$$cost(E_i) \propto [level(E_i) - 1]$$

$$Cost(E_i) = q_i * (level(E_i) - 1)$$

↳ no of element comparisions.

where q_i is probability of searching a string from set E_i

∴ Cost(unsuccessful search) = $\sum_{i=1}^{n} q_i * (level(E_i) - 1)$

$$\therefore \text{Cost}(T) = \sum_{i=1}^{n} P_i * \text{level}(a_i) + \sum_{i=0}^{n} q_i * (\text{level}(E_i) - 1)$$

$\sum_{1}^{n} p(i) + \sum_{0}^{n} q(i) = 1$

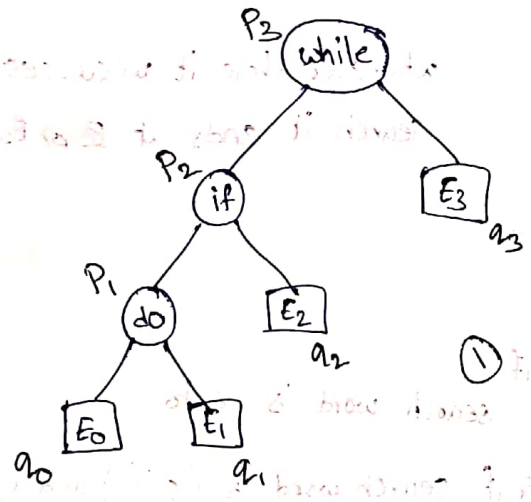
Formally $q(i)$ is probability of searching key x such that $a_i < x < a_{i+1}$

Eg: $n = 3$; $\langle a_1 a_2 a_3 \rangle = \langle$ do, if, while $\rangle$

$\langle P_1 P_2 P_3 \rangle = \langle 0.5, 0.1, 0.05 \rangle$

$\langle q_0, q_1, q_2, q_3 \rangle = \langle 0.15, 0.1, 0.05, 0.05$

Compute the cost below BST



Each E_i corresponds to class of string x such that $a_i < x < a_{i+1}$

also we assume $a_0 = -\infty$; $a_{n+1} = +\infty$

Sol:

The level of root is 1.

$$\text{Cost(suc. search)} = \sum_{i=1}^{n} \text{level}(a_i) * P_i$$

~~$= P_1*1 + P_2*2 + P_3*3$~~ $= P_1*3 + P_2*2 + P_3*1$

~~$= 0.5 + 0.2 + 0.15$~~ $= (0.5)(3) + (0.1)(2) + (0.05)(3)$

~~$= 0.85$~~ $= 1.5 + 0.2 + 0.05$

$= 1.75$

$$\text{Cost(unsuc. search)} = \sum_{i=0}^{n} (\text{level}(E_i - 1)) * q_i$$

$= (0.15)(3) + (0.1)(3) + (0.05)(2) + (0.05)(1)$

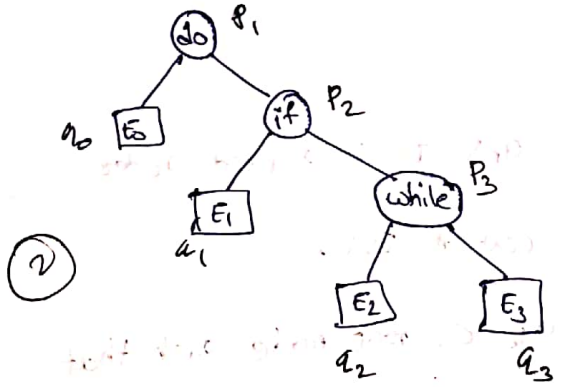
$= 0.45 + 0.3 + 0.1 + 0.05$

$= 0.9$

1.75
~~1.85~~

$\therefore$ Cost(BST) = ~~0.85~~ + 0.9 = ~~0.95~~ ~~1.75~~ = 2.65

Eg: Compute cost of below tree



(2)

~~pole the diran~~
If Tree in previous example is considered, then ~~order of~~ arrangement of E_0, E_1, E_2, E_3 has to fixed in 2nd tree

Cost(suc) = $1 * 0.5 + 2 * 0.1 + 3 * 0.05$

= $0.5 + 0.2 + 0.15$

~~= 1.35~~ = 0.85

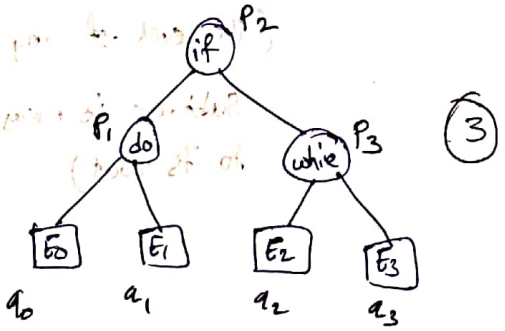
Cost(unsuc) = $(0.15)(1) + (0.1)(2) + (0.05)(3) + (0.05)(3)$

= $0.15 + 0.2 + 0.15 + 0.15$

= 0.65

$\therefore$ Cost(T) = 0.85 + 0.65 = 1.5

Eg: Compute cost of below tree



(3)

Cost(suc) = $(0.5)(2) + (0.1)(1) + (0.05)(2)$ = $1 + 0.1 + 0.1 = 1.2$

Cost(unsuc) = $(0.15)(2) + (0.1)(2) + (0.05)(2) + (0.05)(2)$

= $0.3 + 0.2 + 0.1 + 0.1 = 0.7$

Cost(T) = 1.9

even through Tree 3 is Complete it has higher cost than Tree 2.

Construction of BST of optimal cost:

Let us consider n identifiers ~~with~~ $\langle a_1 \cdots a_n \rangle$ associated with probabilities $\langle P_1 \cdots P_n \rangle$. Let probabilities of external nodes be $\langle q_0 \cdots q_n \rangle$.

Assume $\{a_1\ a_2\ a_3 \cdots a_n\}$ are in sorted order.

Let cost(T) denote cost of BST.

Now we select a_k as root node such that

$$Cost(T) = \min_{1 \le k \le n} \{ P_k + COST(L) + COST(R) \}$$

Mathematical modelling:

$$C(0,n) = \min_{k=1 \text{ to } n} \{ P_k + C(0,k-1) + C(k,n) \}$$

$+\Delta_1$ $+\Delta_2$

~~However the~~

The value calculated are calculate for subtree considering it as a tree. So level number will be less. So we add $\Delta_1 + \Delta_2$ (The level of any identifier within subtree is measured with respect to its root)

Heap Algorithms:

Heap is used in the implementation of priority queue.

Defn: It is a complete binary tree with the property that the value of a node is greater(smaller) than the values of the nodes in their left & right subtrees.

→ In general it is implemented as binary heap. However we may use 3-ary or 4-ary or k-ary heap.

Operations:

(i) Create
(ii) Insert
(iii) Delete
(iv) Inc/Dcr key

Insertion:

(i) Inserting elements one ~~after~~ after other.

Time complexity:

- Best case : O(n) (Dec order)
- worst case : O(nlogn) (Inc order)

** Framework:

→ The max no of nodes at any level 'i' is 2^{i-1}. (root at level 1)

→ The no of level comparisions/movements of a node being inserted at level 'i' = i-1

→ The total no of level comparision for all nodes at level 'i'

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

→ Time for all nodes at all levels

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

where k is last level

$$= \frac{1}{2}\left[\sum_{i=1}^{k} i \cdot 2^i - \sum_{i=1}^{k} 2^i\right]$$

$$= \frac{1}{2}\left[(k-1)2^{k+1} + 2 - (2^{k+1} - 2)\right]$$

$$= \frac{1}{2}\left[k \cdot 2^{k+1} - 2^{k+1} + 2 - 2^{k+1} + 2\right]$$

$$= \frac{1}{2}\left[2^{k+1}(k-2) + 4\right]$$

$$= 2^k(k-2) + 2$$

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

For a full binary tree / complete binary tree

if 'n' is no of nodes

$$2^k = O(n) \;\&\; k = O(\log n)$$

$$\Rightarrow n(\log n - 2) + 2$$

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

22/10/20

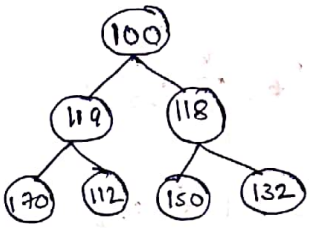
Build Heap (Heapify):

Heapify: The process of transforming a given complete binary Tree (with n elements) to a heap using method of adjust.

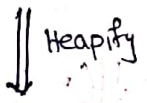
Adjustment procedure is carried out level by level starting from level k-1 then k-2 upto level 1.

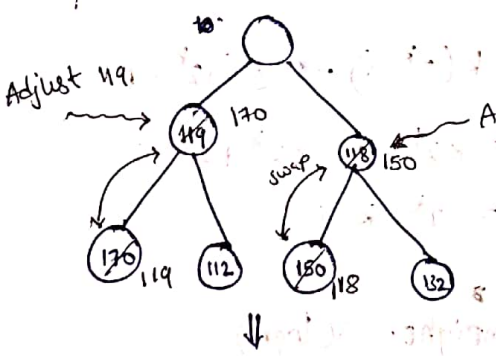
where k is Last level

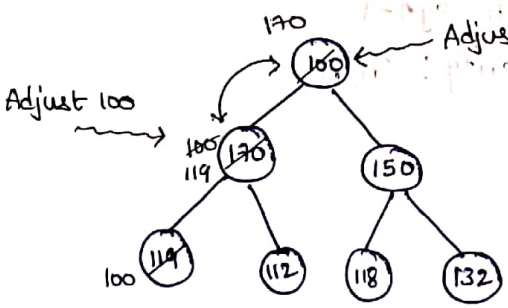
Eg: A: <100, 119, 118, 170, 112, 150, 132>

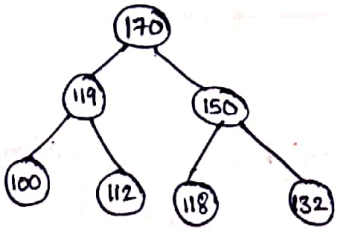


100	119	118	170	112	150	132









170	119	150	100	112	118	132

Time Complexity:

→ Max no of nodes at any level is 2^{i-1}

→ Max no of level comparisions for a node that is getting adjusted at level i is $k-i$

where k is height (or) last level

→ for all node at level i, total comparisions $= (k-i)\,2^{i-1}$

→ Time Complexity = total comparision for all nodes at all levels

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

$$= k\sum_{i=1}^{k} 2^{i-1} - \sum_{i=1}^{k} i\cdot 2^{i-1}$$

$$= k\cdot\frac{2^k-1}{2-1} - \frac{1}{2}\sum_{i=1}^{k} i\cdot 2^{i}$$

$$= k(2^k-1) - \frac{1}{2}\left[(k-1)\cdot 2^{k+1} + 2\right]$$

$$= k(2^k-1) - (k-1)2^k - 1$$

$$= k2^k - k - k2^k + 2^k - 1$$

$$= 2^k - k - 1$$

k = height $= O(\log n)$

$= n - \log n - 1$

∴ time complexity : $O(n)$

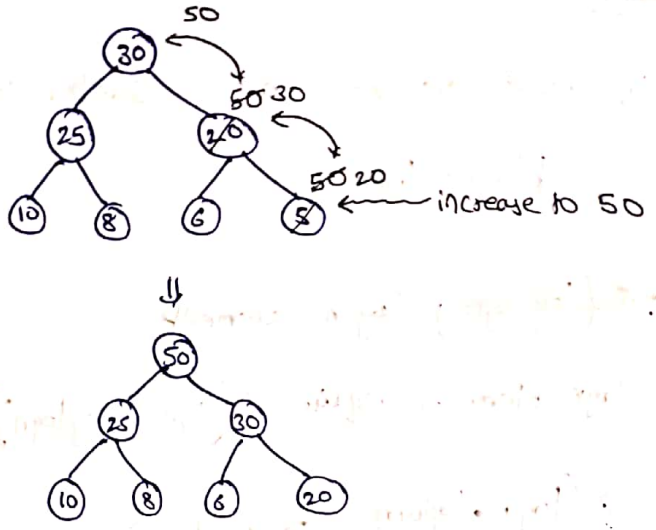
Heap Sort:

```
Algo Heap_Sort(n)
{
   1. Heapify(n)  ⟶ O(n).
   2. for i ← n downto 2
      {
         swap(A[1], A[i]); O(1)
      }  Adjust(A[1]); ⟶ O(log n)   } O(n log n)
}
```

∴ Time complexity of heapsort : O(nlogn)

Increasing/ Decreasing value of a Node :

Consider Max heap



Thus if it is increase operation, the node may go from leaf to root in worst case and if it is delete operation, the node may go from root to leaf

∴ TC for inc/dec operation : O(logn)

(H/61) 1st smallest element cannot be below 1st level

2nd smallest element cannot be below 2nd level

Silly 7th smallest cannot be below 7th level.

∴ we need to search all 1st 7 levels.

∴ $2^7-1 = 128-1 = 127$ nodes have to be searched

∴ O(1)

(H/63) ⓓ

options are

a) $\log n$ b) $\sqrt{\log n}$ c) $O(1)$ d) $\frac{\log n}{\log \log n}$

option verification :

If k is no of element can be sorted in $O(\log n)$ then

$$k \log k \geq \log n$$

a) ~~$\log n (\log \log n)$~~ $\log n$ elements

$\log n$ elements require $O(\log n (\log \log n))$ time.

$O(\log n \cdot \log \log n) > O(\log n)$

∴ false

b) $\sqrt{\log n}$ elements

$\sqrt{\log n}$ elements require $O(\sqrt{\log n} (\log \sqrt{\log n}))$ time

$O(\frac{1}{2} \sqrt{\log n} \cdot \log \log n) \neq O(\log n)$

∴ false.

c) ~~Constant time is~~.

$O(1)$ --- constant elements.

to sort constant no of elements constant time is required

$O(1) \neq O(\log n)$

d) to sort $\frac{\log n}{\log \log n}$ we need $O(\frac{\log n}{\log \log n} \cdot \log (\frac{\log n}{\log \log n}))$

$$= O\left(\frac{\log n}{\log\log n}\left(\log\log n - \log(\log\log n)\right)\right)$$

$$= O\left(\log n\left(1 - \frac{\log(\log\log n)}{\log\log n}\right)\right)$$

$$= O\left(\log n - \underbrace{\frac{(\log n)(\log\log\log n)}{\log\log n}}_{< \log n}\right)$$

$$= O(\log n)$$

$\log\log\log n < \log\log n$

$\Rightarrow \frac{\log\log\log n}{\log\log n} < 1$

$\Rightarrow \log n \cdot \frac{(\log\log\log n)}{\log\log n} < \log n$

H/64 (i) Take 1st element from each list and build a help heap. H this takes $\log\log n$ time.

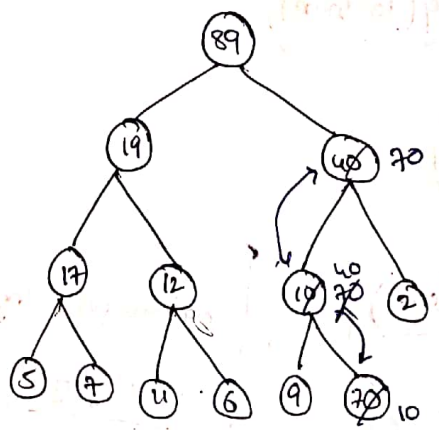
(ii) Delete an element from this heap and put it in the array (the array in which we want to store the sorted result) --- $O(\log\log n)$.

(iii) Now take and element from ~~array~~ list ~~foo~~ whose element got added in the array and insert into heap H --- $O(\log\log n)$.

Repeat ~~step (i)~~ step (ii) & (iii) until all the elements are from the list are finished i.e., n times

~~i.e., $n\log\log n$~~

$\therefore$ TC : $O(n\log\log n)$



∴ 2 interchanges

Graph Techniques (Traversals):

Traversal: Visiting all the nodes of the tree/graph in an order & processing the information only once is traversal.

Tree traversal vs Graph traversal:

→ The traversal of tree (inorder or preorder or postorder) is unique where as graph traversal need not to be unique.

Framework:

1. **Status of a node:**

During the traversal, any node must be in one of the 3 states

a) **E-Node:**

The node that is currently under exploration.

There is only one E-node during traversal.

b) Live - Node :

The Node which has not yet been explored fully or all of whose children has not yet been explored.

- stack in DFS & queue in BFS are used to store these node

c) Dead - Node :

The node which has been explored fully.

2. Timing values during traversal :

→ every node is associated with 2 two timing value

d = discovery time (Prev_value)

f = finishing time (Post_value)

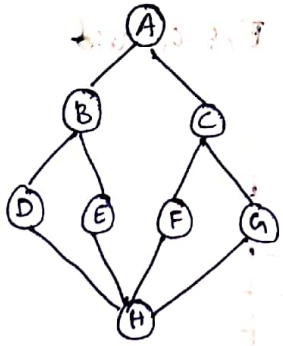
d & f > 0

d: the time at which node is first visited.

f: the time at which node becomes dead.

DFS :

1) DFS in undirected connected graph :

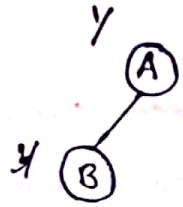


Let us start DFS from A

So initially we give discovery time of A = 1

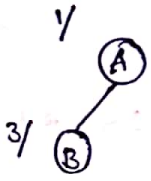
(However we can have any value greater than 0)

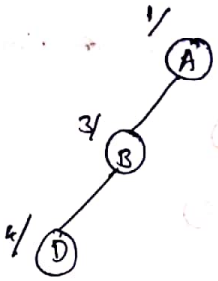
Now current A is E-Node.



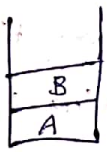
A ; B

Now B becomes E-node, & A becomes live node and A is stored in stack

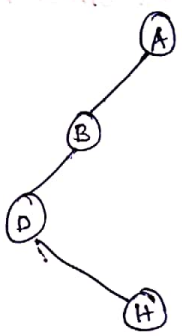




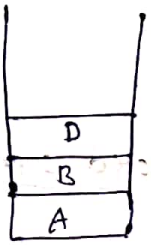
A ; B ; D



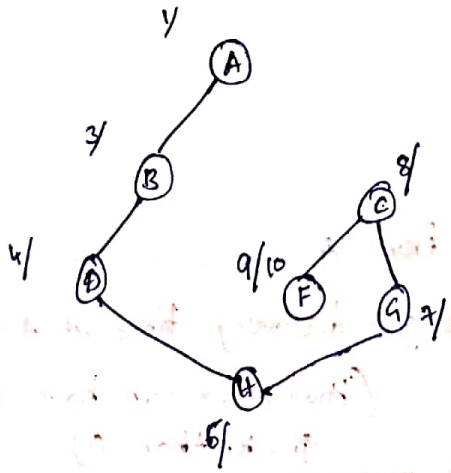
} live nodes



A ; B ; D ; H



Here D is parent of H and E, F, G, are children of H



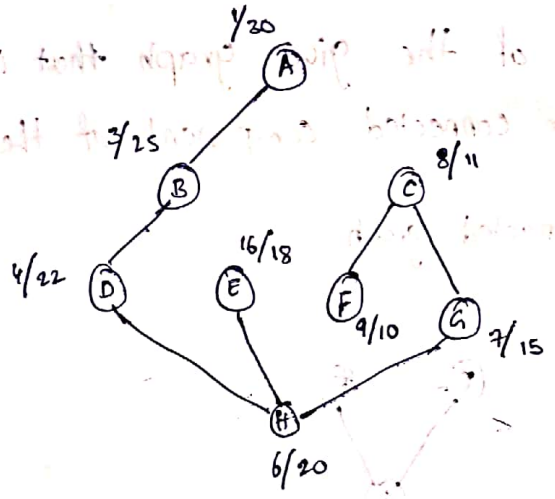
A ; B ; D ; H ; G ; C ; F

Now F 'b' has no adjacent unexplored nodes.

So we assign it finish time and 'F' becomes dead node.

Now pop off 'c' from the stack.

Now c becomes the E-node



In this traversal 'H' becomes E-node thrice

∴ traversal : A, B, D, H, G, C, F, E

Note :

→ ~~The graph~~ is

→ The undirected graph is connected

⇔ finish time of first node is the highest

⇔ traversal finishes when the stack become empty for first time.

⇔ traversal finishes when the first node becomes dead node

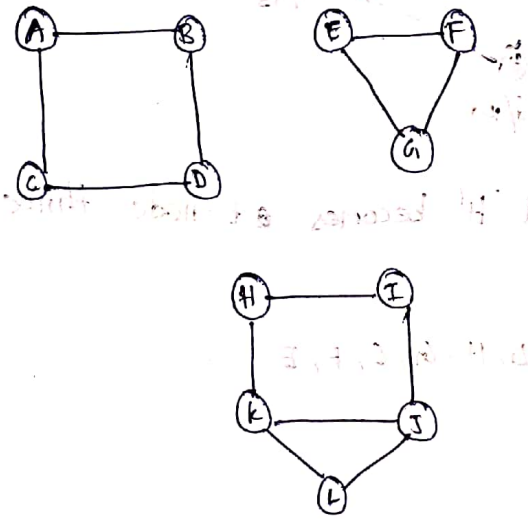
2) DFS in undirected disjoint (disconnected graph):

→ DFS when carried out on disconnected graph is called Depth First Tree (DFT).

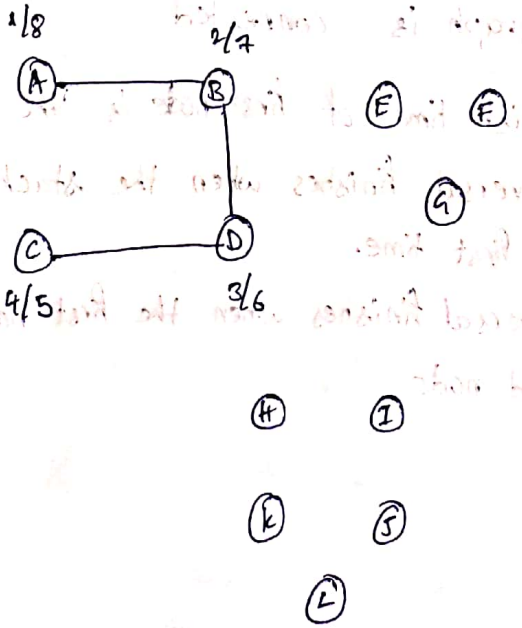
Connected Component:

Maximal subgraph of the given graph that is connected is called ~~com~~ connected component of the graph.

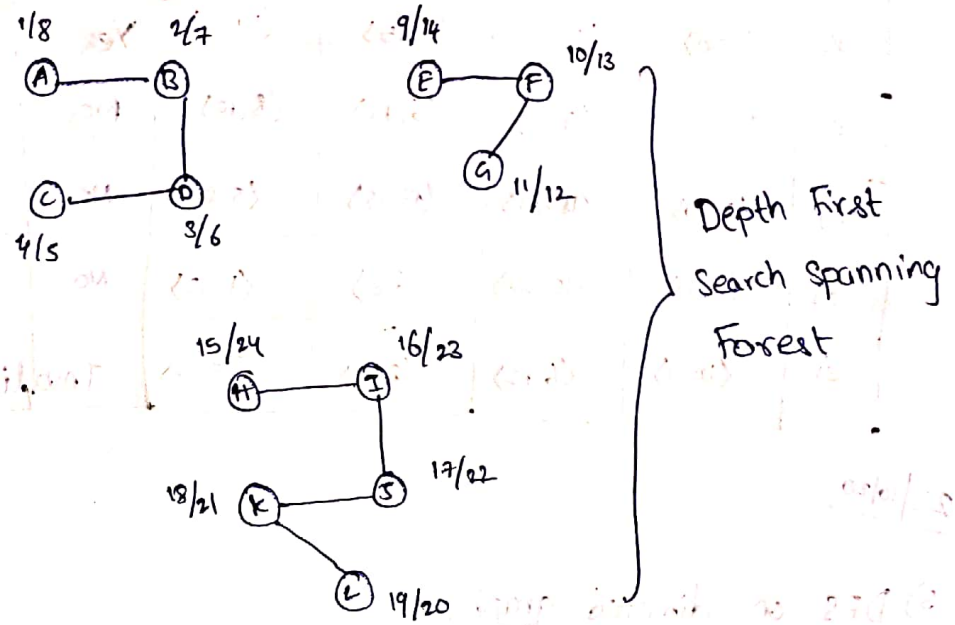
Consider below disconnected graph



Start traversal from any of the unexplored node.



Now stack has become empty but still traversal is not finished since we have unexplored nodes.



Here finishing time of first node (A) is not the highest finishing time

∴ disconnected

Also graph is disconnected if there are more than one spanning trees.

no of disconnected components = no of spanning trees in depth first traversal.

Q. Given an undirected graph with 4 vertices (P, Q, R, S) and their (d,f) values as follows. Indicate for each option whether the graph is connected or disconnected and no of connected components.

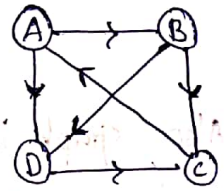
	Vertices				Connected	no of connected components
	P	Q	R	S		
	⟨d,f⟩	⟨d,f⟩	⟨d,f⟩	⟨d,f⟩		
a)	⟨5,20⟩	⟨6,18⟩	⟨8,15⟩	⟨10,12⟩	Yes	1
b)	⟨13,14⟩	⟨7,11⟩	⟨6,12⟩	⟨8,10⟩	No	2 (QRS) (P)
c)	⟨3,20⟩	⟨8,20⟩	⟨13,15⟩	⟨5,8⟩	No	3 PS R Q
d)	⟨9,10⟩	⟨15,20⟩	⟨6,8⟩	⟨11,12⟩	No	4 R P S Q
e)	⟨10,11⟩	⟨6,12⟩	⟨8,13⟩	⟨15,14⟩	Invalid	

23/10/20

3) DFS on directed graph:

→ DFS when carried out over a directed graph leads to the following types of edges

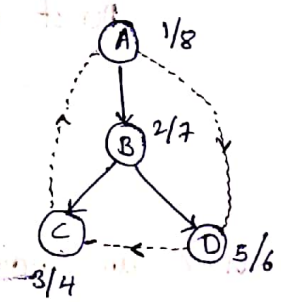
a) Tree edge:



The edges that are part of depth first search tree / forest

Eg: AB, BC, BD

b) Forward Edge:



It is an edge that leads from a node to its non-child descendent. (This edge is part of graph but not of spanning tree / forest)

Eg: AD is the only forward in the spanning tree

(∵ D is non-child descendent of A)

c) Back Edge :

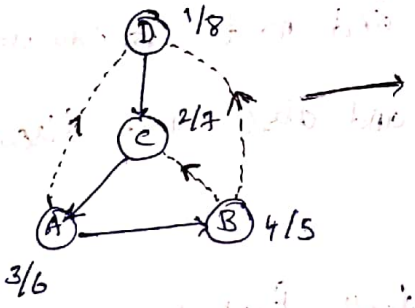
It is an edge that leads from a node to its ancestor (This edge is present in graph but not in spanning tree). Eg : CA

d) Cross Edge :

It is an edge that leads from a node to another node which is neither ancestor nor descendent.

Eg : DC (This edge is also part of graph but not of tree)

Eg : Consider below below DFS on node 'D'

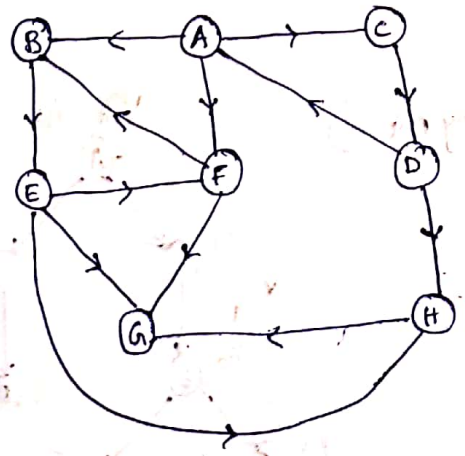


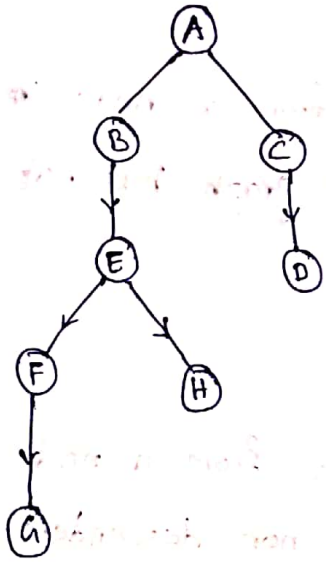
→ Fully decorated DFS spanning tree...
(The DFS spanning tree in which all the types of edges are drawn)

DC, CA, AB are tree edges

AD, BC, BD are back edges

Eg :





For above given graph and a spanning tree, check whether the given ~~spann~~ DFT tree is valid or not.

If it is valid find no of forward edges, backward edges and cross edges. and also mark discovery & finishing times.

Sol:

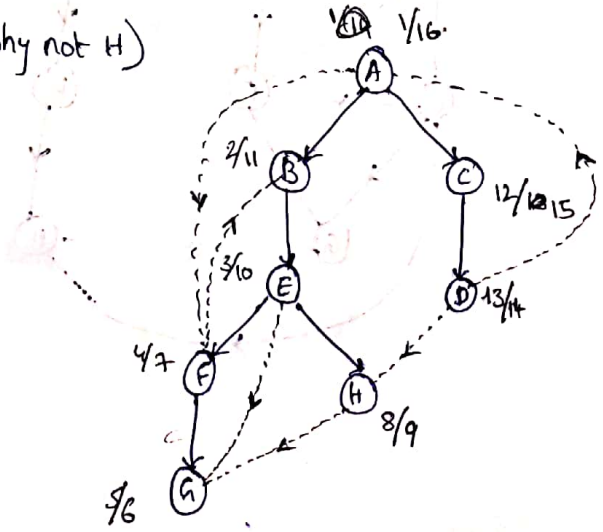
It is a valid depth first traversal spanning tree.

The order in which nodes are visited is

A; B; E; F; G; H; C; D

(Think why not C)

(Think why not H)



Forward Edges: AF; EG;

Back Edges: DA; FB;

Cross edges: DH; HG;

Parenthesis Theorem:

In any DFS of a directed graph $G=(V,E)$, for any two vertices 'u' & 'v' having an directed edge from u to v. exactly one of the condition holds

inclusive

I. `[ [ ] ]` → The (d/f) time interval of vertex 'v' is enclosed within (d/f) time interval of vertex 'u'; Then edge uv is either a forward edge (or) a tree edge

d[u] d[v] f[v] f[u]

II. `[ [ ] ]` → Same as previous, case

d[v] d[u] f[u] f[v]

Here uv is formes a back edge.

exclusive

III. `[ ] [ ]` → uv is a cross edge

d[u] f[u] d[v] f[v]

(or)

d[v] f[v] d[u] f[u]

4) DFS on DAG (Directed Acyclic Graph) (Precedence Graph)

Every DAG is charecterized by two kinds of vertices

i) Source vertices:

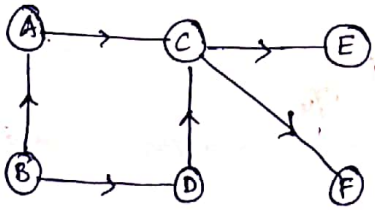
Vertices that doesn't have any precedences.

i.e., vertices that don't have any incoming edges

ii) Sink vertices:

Vertices that don't have any out bounded (outgoing) edges.

Eg:



B → source vertex

E, F → Sink vertices.

→ One of application of DAG is to perform topological sort (or) topological ordering.

→ Topological sort:

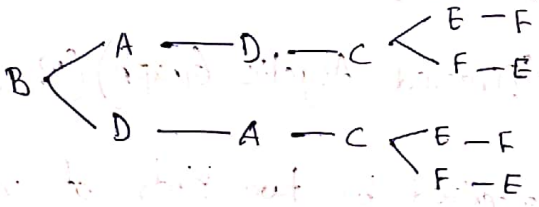
It is a linear ordering that exhibits the precedences of the vertices in the graph.

DAG can be viewed as task where edges represents dependencies or precedences and vertices represent activities.

Eg: In the example graph

- Activity C should be done only after finishing activity A & D.

Eg: For example graph taken topological order is



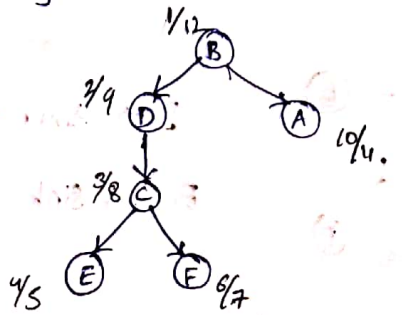
∴ topological sorts possible are

B A D C E F
B A D C F E
B D A C E F
B D A C F E

∴ no of topological orders possible = 4

Finding topological sort from DFS:

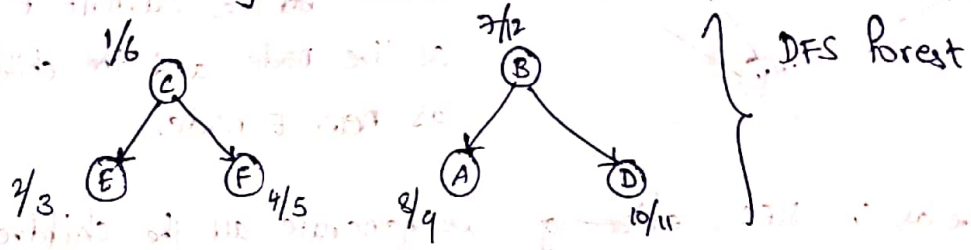
Consider starting DFS from 'B'



The decreasing order of finishing times gives topological sort.
i.e., BADCFE for this traversal

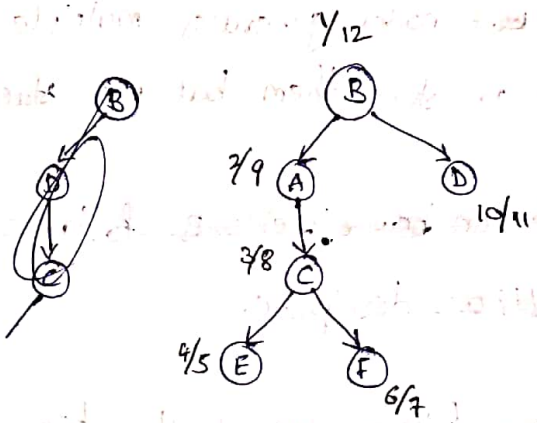
Note that to obtain topological sort we can start from any vertex.

Consider starting from vertex 'c'



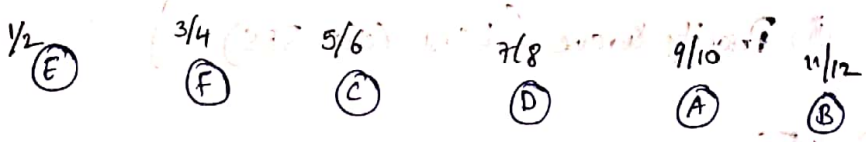
∴ Topological sort : BDACFE

Consider another traversal



∴ topological sort : BDACFE

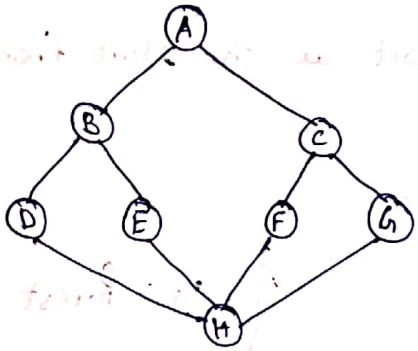
Consider another traversal



topological sort : BADCFE

```
Algo topological_sort()
{
    DFS(v); // v can be any vertex.
    Print nodes in descending order of their finishing times
}
```

Breadth First Search (BFS)



In DFS, when we explore a node, we generate a child node of current E-node and make the current E-node as live node and the child nodes as new E-node.

whereas in BFS, we generate all the children of current E-node and the unexplored nodes of these nodes are made live nodes (stored in the queue)

Since a single E-nodes generates multiple live nodes, we must use queue to store them but not stack.

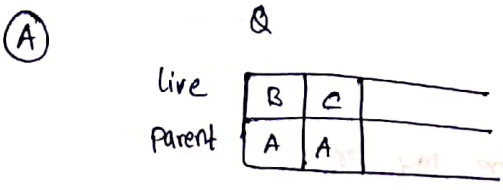
~~→ The BFS state can be either follow either a FIFO discipline or LIFO discipline.~~

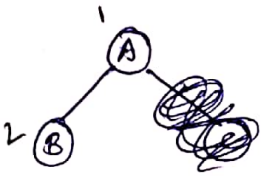
→ The Queue may follow any of the below 3 disciplines.

(i) FIFO (FIFO BFS) → taken as default
(ii) LIFO (LIFO BFS or Dsearch)
(iii) Priority Queue (Least Cost BFS)

} Corresponding BFS

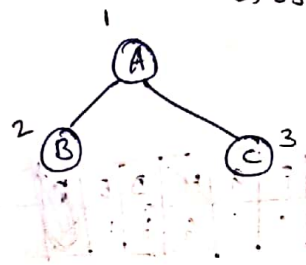
FIFO BFS





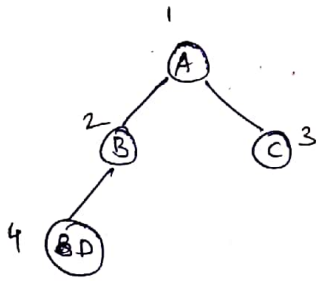
live	~~B~~	C	~~B~~ D	E	
parent	A	A	B	B	

Now obtain C from queue and add to tree
↳ as E-Node



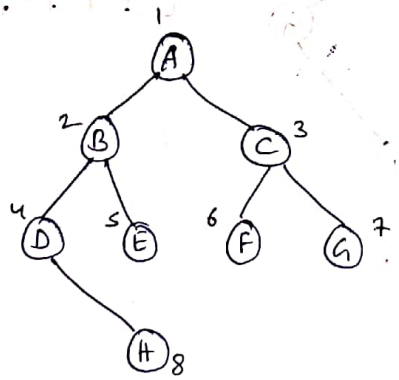
~~B~~	~~C~~	D ~~D~~	E	F	G
A	A	B	B	C	C

Now ~~cho~~ obtain D as next E-Node



~~B~~	~~C~~	~~D~~	E	F	G	H
A	A	B	B	C	C	D

Choose E as next E-Node

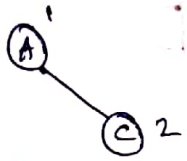


~~B~~	~~C~~	~~D~~	~~E~~	~~F~~	~~G~~	~~H~~
A	A	B	B	C	C	D

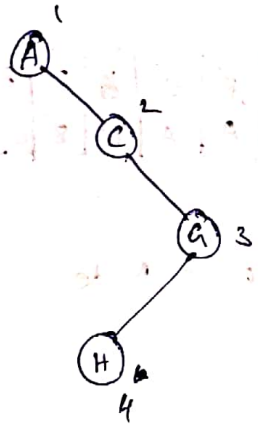
Breadth First Search - Spanning tree

This is FIFO BFS and the order of visiting will be the order in which they are pushed in queue

LIFO BFS:

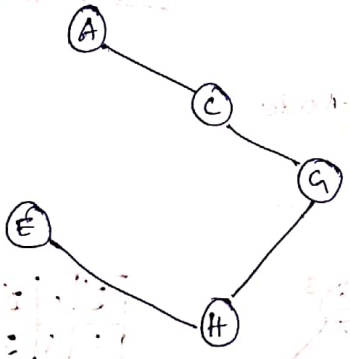


B	~~C~~	F	G
A	A	C	C



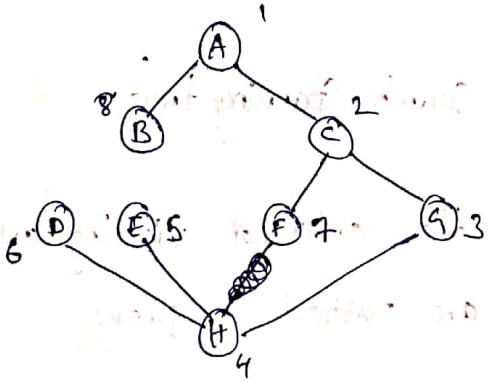
~~B~~	~~C~~	F	~~G~~	~~H~~	D	E	~~F~~
A	~~A~~	C	C	G	H	H	~~H~~

Choose E



B	~~C~~	F	~~G~~	~~H~~	D	~~E~~
A	A	C	C	G	H	H

Choose D



B	~~C~~	~~F~~	~~G~~	~~H~~	~~D~~	~~E~~
A	A	C	C	G	H	H

At every node we move to depth.

Hence we call it D-search (depth-search)

D-search sequence is A C G H F D F B

↳ (This is invalid DFS)

Note:

D-search need not to be a depth first search.

Priority Queue (LC-BFS)

→ This is used in gaming algorithms

Eg: Consider 15-puzzle problem

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Initial arrangement

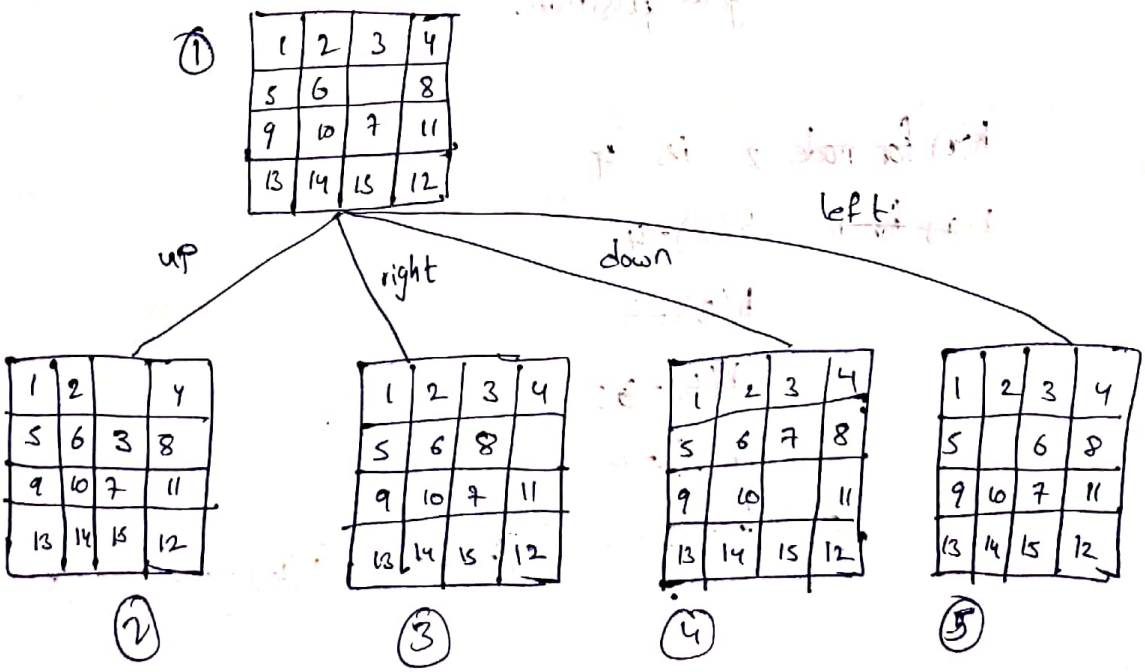
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal Arrangement

The objective is to reach goal arrangement with min possible moves.

We LC-BFS for this

Based empty slot we may have 2 (or) 3 (or) 4 possible moves at each step



→ Using FIFO or LIFO here is a blind search.

→ So we need to prioritize one of the live nodes and make it as E-node.

Thus to perform every LC-BFS, every live node is associated with a cost.

We select a node with minimum cost and hence the priority queue is min-heap

→ Cost function of live node is indicated as $\hat{c}(x)$

$$\hat{c}(x) = f(x) + h(x)$$

$f(x)$ → cost of reaching node 'x' from root.

$h(x)$ → estimated cost of reaching goal state from node x.

Formulating $h(x)$ may vary from node to node.

→ One way to estimate $h(x)$ in this problem is

$h(x)$ = no of non-blank tiles that are not in goal position.

~~$h(x)$ for node 2 is 4~~

~~$h(x)$ for n~~ $h(2) = 4$.

$h(3) = 4$

$h(4) = 2$

$h(5) = 4$

$\Rightarrow \hat{c}(2) = 5;\ \hat{c}(3) = 5;\ \hat{c}(4) = 3;\ \hat{c}(5) = 5;$

Thus we explore ~~only~~ node 4.

i.e., ~~that~~ make 4 as E-node.

→ After exploring node '4' we get 3 more live node and total live nodes at this point = 6.

Now we choose least cost node and continue the process.

Time Complexities:

→ The time complexity of both DFS & BFS with

(i) adjacency matrix ---- $O(n^2)$

(ii) Adjacency list ---- $O(n+e)$

Applications:

→ Both BFS & DFS can be used to determine the presence of cycle in graph.

→ either using DFS or BFS

The presence of backedge in directed (back/forward in undirected) confirms the presence of cycle.

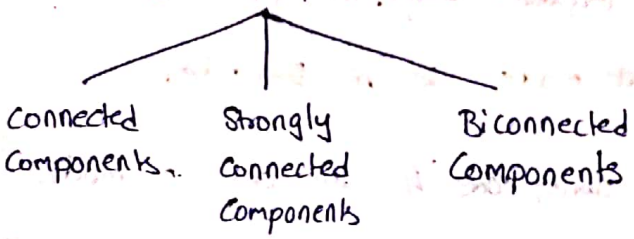
→ DFS & BFS can be used to know existence of a path btw a given pair of vertices.

→ DFS is used as a searching strategy in backtracking design strategy.

→ BFS is used as a searching strategy in branch & bound strategy.

→ For undirected graph, the BFS spanning tree will act as the solution for the single source shortest path problem by considering unit wt. cost graph. Optimal ~~graph~~ algorithm

→ DFS is used to find ~~some~~ Components & Articulation points



Components:

(i) Connected Components:

Maximal subgraph that is connected is called connected component of given graph.

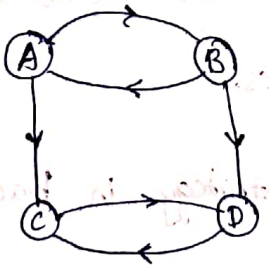
This concept applies only of undirected graphs.

(ii) Strongly Connected Components (This concept applied to directed graphs)

Two vertices 'u' & 'v' in a directed are said to be connected iff there is a path from 'u' to 'v' and 'v' to 'u'.

→ This relation of being connected, partitions the vertex set 'V' of the graph into maximal disjoint sets known as strongly connected components.

Eg:

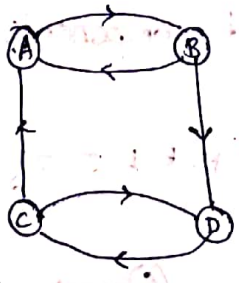


G=(V,E)

The above graph has 2 strongly connected components

i.e., {A,B} {C,D}

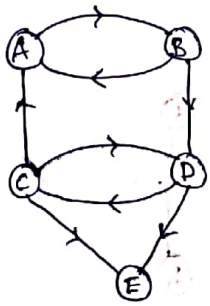
Eg:



no of strongly connected components = 1

i.e., {A, B, C, D}

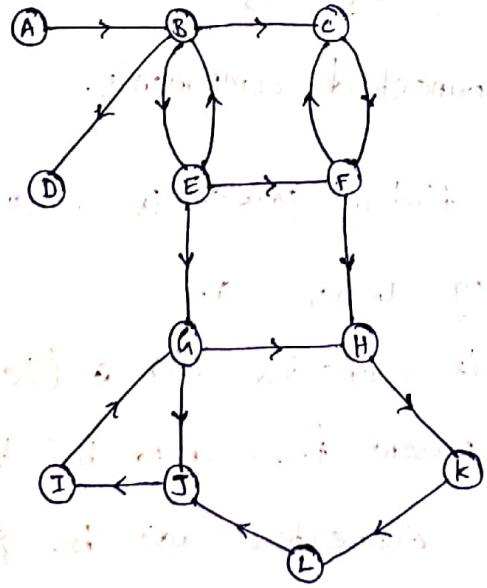
Eg:



no of strongly connected components = 2.

i.e., {A, B, C, D} & {E}

Eg:



Sol:

A has 1 out going edge & D has only 1 incoming edge

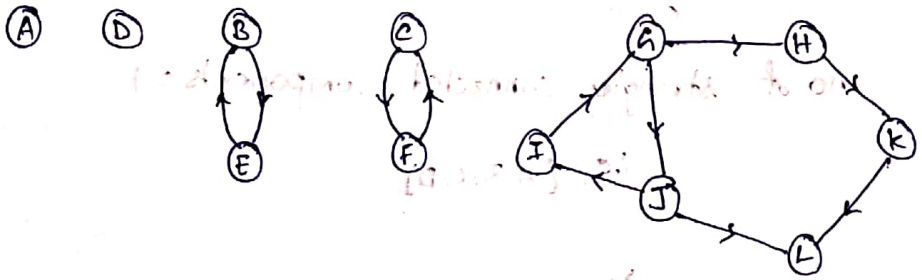
∴ {A} {D} are 2 strongly connected components

G H K L J I is a cycle.

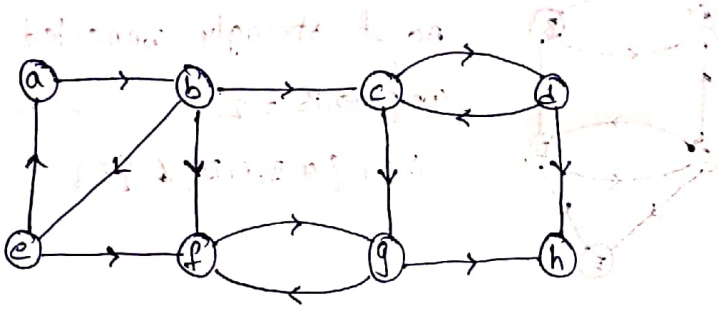
∴ {G, H, K, L, J, I} is a strongly connected component

∴ we have 5 strongly connected components

{A} {D} {B,E} {C,F} {G,H,K,L,J,I}



Eg:



{a,b,e} {f,g} {c,d} {h}

∴ 4 strongly connected components.

Approach for this kind of problem is create a set initially containing vertex a.

Now check if 'b' is connected with 'a' (i.e, path ~~present b to from~~ present from a to b & b to a.)

Since b & a are connected add b. {a,b}.

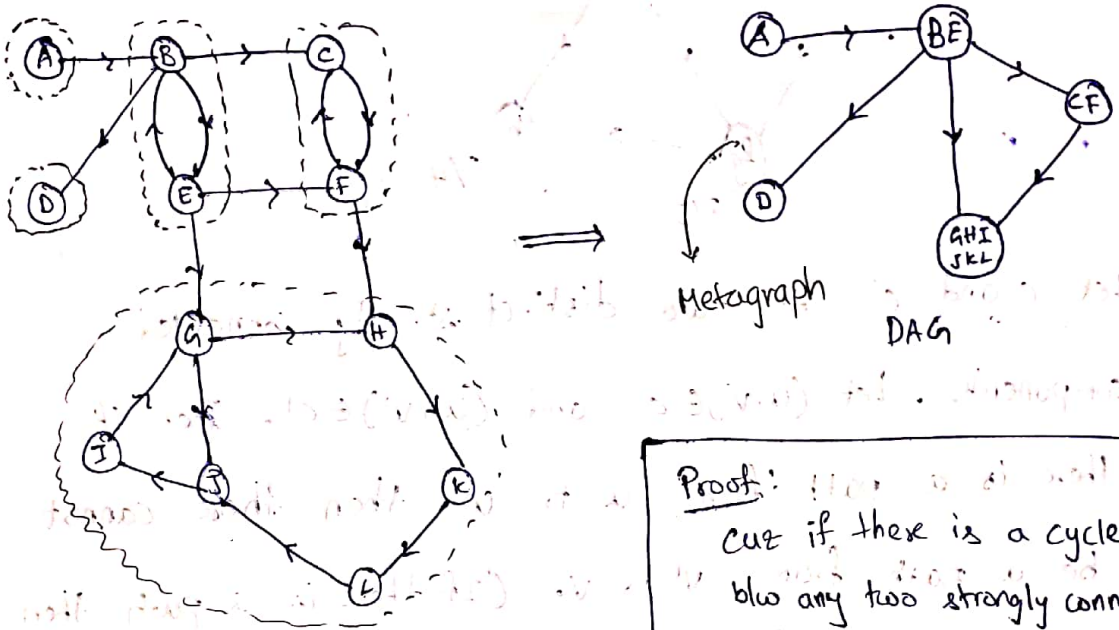
Now check with c.

~~C~~ C is not connected with 'a'. So create a new set containing c.

Continue this process for all the vertices.

Properties:

1) Every directed graph is a DAG of its strongly connected components.

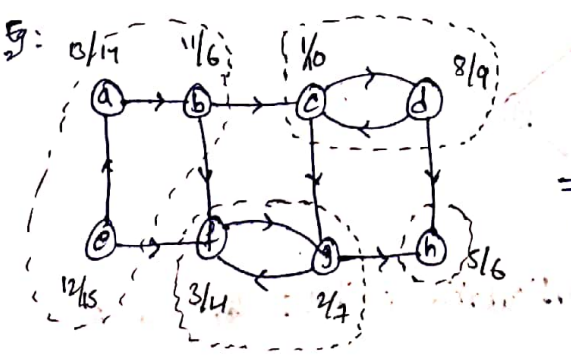


Metagraph: If $G=(V,E)$ is graph then $G'=(V,E')$ such that $E' \subseteq E$ is called metagraph. i.e. Metagraph is a subgraph with all the vertices in graph

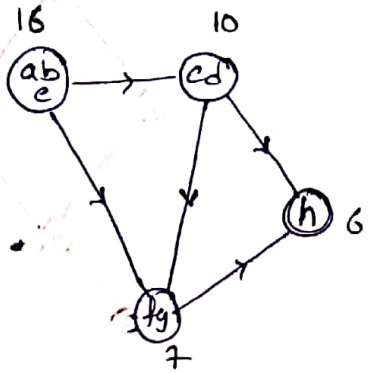
Proof:
Cut if there is a cycle blw any two strongly connected components then the two components would become one.

2) Let c & c' be two strongly connected components. If there is a path from a vertex in c to a vertex in c', then the highest finishing time of c' will always be greater than the highest finishing time (of any vertex) of c'. (Think why)

Eg:

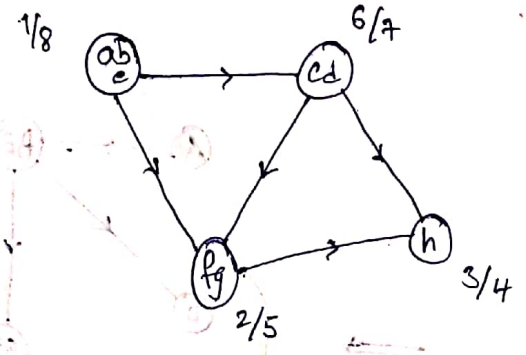


Metagraph



The property can be verified from above graph

Consider carrying DFS on DAG of connected components of a directed graph. The same property holds again :



3) Let c and c' be two distinct strongly connected components. Let $(u,v) \in c$ and $(u',v') \in c'$. Then if there is a path from u to u', then there cannot be a path from v' to v. (If there is a path then the whole thing forms a single connected component)

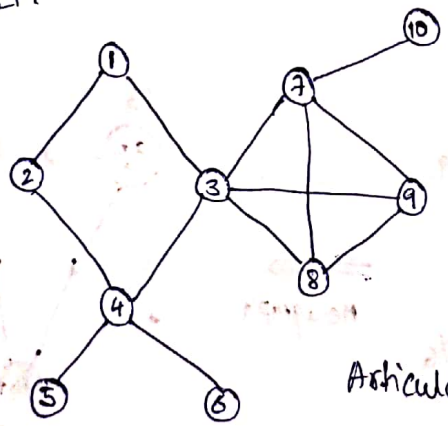
24/10/20

Articulation Point (Cut vertex): & Biconnected Components

The vertex whose removal makes a connected undirected graph disconnected is called articulation point.

→ The graph is said to be biconnected if it does not have articulation point.

Eg:



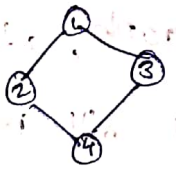
Articulation points: 3, 4, 7

Biconnected Components:

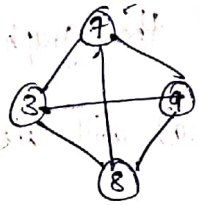
Maximal subgraph that is biconnected is called biconnected component of graph G.

→ To find no of biconnected ~~graph~~ components in a given graph note that biconnected points form at the point where articulation points are present.

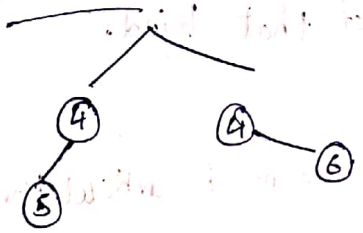
i.e., for 3 & 4 draw all the vertices adjacent to 3 & 4



Now choose 3 again 7 also gets added since it is adjacent to 3.



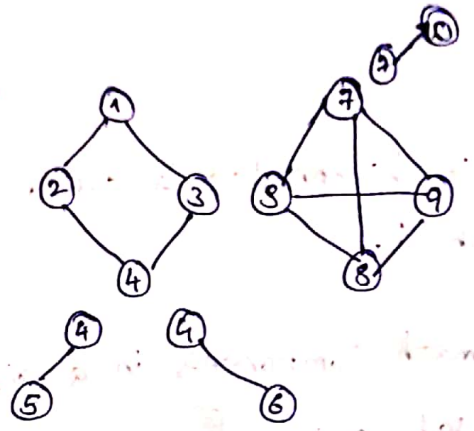
Choose '4'



Choose '7':

∴ no of biconnected components for this graph = 5

i.e.,



→ Minimum no of edges that required to be added to make a graph biconnected ≤ no of articulation points.

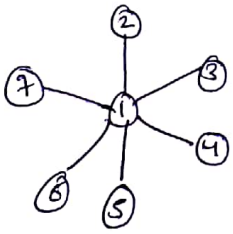
Eg: In the previous graph, adding edges (1,10) (2,5) (8,6) will make the graph biconnected.

However adding only two edges (5,10) & (6,8) will ~~not~~ will also make the graph biconnecte.

(or) adding (1,5) & (6,10) will also make the graph biconnected.

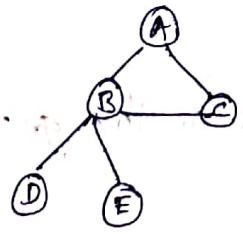
→ This formula generally applies to all graph except to star graph or graph of that kind.

Eg:



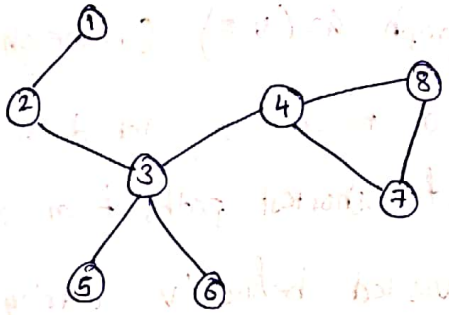
Here no of articulation points = 1
However adding (≤1) edges won't make the graph biconnected.

Eg:



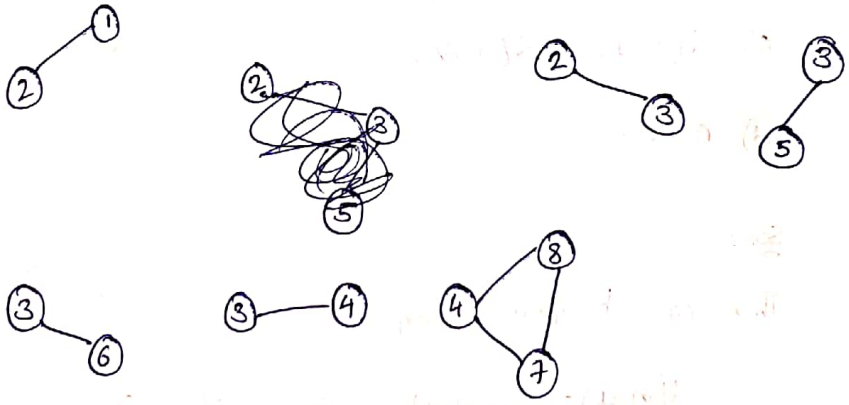
B is the articulation point.
no of articulation points = 1.
However min no of edges req to make the graph biconnected = 2

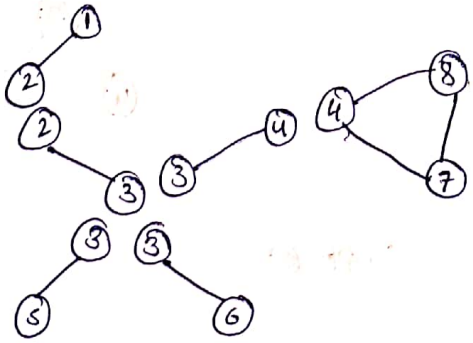
Eg: Identify the articulation points and draw the biconnected components of graph and find min of edges to be added to make the graph biconnected.



Articulation points → 2, 3, 4

Biconnected Components





∴ 6 biconnected Components

Min no of edges to be added to make the graph biconnected = 3

i.e., Add (2,5) (6,7) (1,4)

Bridge:

The edge whose removal makes the graph disconnected.

Q) Consider an undirected graph $G=(V,E)$ (unweighted). If BFS of G is done from a node 'r', let $d(r,u)$ and $d(r,v)$ be the lengths of shortest paths from r to u & v respectively. If 'u' is visited before 'v' during the traversal then which is true.

a) $d(r,u) \leq d(r,v)$

b) $d(r,u) > d(r,v)$

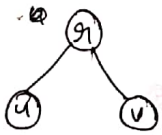
c) $d(r,u) \leq d(r,v)$

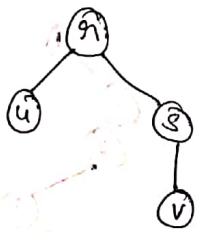
d) none.

Sol:

This can happen when

$d(r,u) = d(r,v)$ (or) $d(r,u) < d(r,v)$





∴ opt ©

Q) In a DF-traversal of a graph 'G' with n-vertices, 'k' edges are marked as tree edges, the no of connected components of 'G' is ____

a) k b) k+1 c) n-k+1 d) n-k

If a forest of n vertices have k edges then no of trees in (components) in forest = n-k.

(Q) A DFS is performed on a DAG. d(x) is discovery time and f(x) is finishing time of a vertex 'x'. which is always true for all edges (u,v) in the graph?

a) d[u] < d[v] b) d[u] < f[v]

c) f[u] < f[v] d) f[u] > f[v]

Sol:

opt (a) & (b) & (c) fails if we start traversal from v.

opt (d) will be successful in any case.

Sorting Techniques:

Sorting techniques are classified ~~into two types~~ as

(i) stable vs unstable

(ii) Internal vs external

(iii) Inplace vs not inplace

(iv) Comparision based vs non-comparision based.

(v) Recursive vs non-recursive.

Comparision based:

The sorting algorithm that use comparision b/w elements is called comparision based sort.

Eg: Quick sort, heapsort etc.

Non-Comparision based:

The sorting technique that doesn't used comparision.

Eg: radix sort, counting sort, address calculation sort.

→ No comparision based sorting takes time less than $O(n \log n)$ for any best case.

Inplace vs Not-inplace:

Sorting technique is said to be inplace if its space complexity is $O(1)$. However for recursive sorting technique it can be $O(\log n)$ for recursion stack.

Otherwise it is said to be not inplace.

Eg: Inplace → Quicksort

not inplace → Mergesort

Internal vs External:

The sorting technique which puts a constraint that all elements must be present in memory before the sorting begins.

Where as external sorting technique allows partial no of elements to be present in memory or allows

· new elements to be added to the list of the numbers to be sorted.

Eg: Insertion sort behaves like external sorting. But it is not completely external sorting.

Stable vs Unstable

A sorting method is said to be stable iff the relative order of non-distinct elements is maintained in the sorted list.

Otherwise it is unstable.

Eg: Merge sort → stable

Quick sort, heap sort → unstable.

Generally stable sorting techniques are more desirable.

Note:

→ Time complexity of a comparision based sort depends on no of compasions and no of swaps.

∴ time complexity = $O(\max\{\text{no of comparision}, \text{no of swaps}\})$

→ No of swaps depends on inversions in the list to be sorted.

if $(i<j)$ and if $A[i] > A[j]$ then the pair (i,j) is known as inversion of the array.

~~Thus no of inversions forms a lower bound for no of swaps.~~

Bubble Sort

```
Algo BS (A,n)
// A[1...n]: array
{
    for pass ← n downto 1
    {
    for i ← 1 to (pass-1)
    {
        if (A[i] > A[i+1])
            swap(A[i], A[i+1]);
    }
    }
}
```

Eg: A: 85 65 25 15 40 5

1st pass:

i=1 : 65 85 25 15 40 5

i=2 : 65 25 85 15 40 5

i=3 : 65 25 15 85 40 5

i=4 : 65 25 15 40 85 5

i=5 : 65 25 15 40 85 85

2nd pass:

i=1 : 25 65 15 40 5 85

i=2 : 25 15 65 40 5 85

i=3 : 25 15 40 65 5 85

i=4 : 25 15 40 5 65 85

3rd pass:

i=1 : 15 25 40 5 65 85

i=2 : 15 25 40 5 65 85

i=3 : 15 25 5 40 65 85

4th pass:

i=1 : 15 25 5 40 65 85

i=2 : 15 5 25 40 65 85

5th pass:

i=1 : 5 15 25 40 65 85

Note:

→ No of comparision in bubble sort = $\frac{n(n-1)}{2}$ (irrespective of data set)

→ No of swaps = no of inversions

∴ Time complexity : $O(n^2)$

→ Even if the elements are in increasing order, we still make $\frac{n(n-1)}{2}$ comparision.

To overcome this we keep track of whether swap has occured in a pass or not. If no swap has occured then the algorithm terminates. The below is the modified algorithm:

```
for pass ← n downto 1
{
    flag=0;
    for i←1 to (pass-1)
    {
        if (A[i] > A[i+1])
        {
            swap(A[i], A[i+1]);
            flag=1;
        }
    }
    if(flag==0) break;
}
```

→ Now for this modified B.S

best case TC = O(n)

worst case TC = $O(n^2)$

The disadvantage is that is requires flag and this is overcome by insertion sort.

→ The recurrence relation for ~~no of comparisions~~ TC of BS is given by

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

$$\Rightarrow T(n) = O(n^2)$$

Selection Sort:

```
Algo ss (A,n)
{
    for i←1 to n-1
    {
        min ← i;
        for j ← (i+1) to n
        {
            if (A[j] < A[min])
                min ← j;
        }
        swap(A[min], i);
    }
}
```

This algorithms selects ith smallest element in ith pass and places it in ith place. So we repeat this for n-1 times.

Eg: A: 80 60 20 15 40 10

i=1: 10 60 20 15 40 80

i=2: 10 15 20 60 40 80

i=3: 10 15 20 60 40 80

i=4: 10 15 20 40 60 80

i=5: 10 15 20 40 60 80

Time Complexity:

- Time = no of comp + no of swaps

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

$$T.C = O(n^2)$$

→ Selection sort requires least no of swaps compared to any other comparision based sorting technique.

→ Rec relation:

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

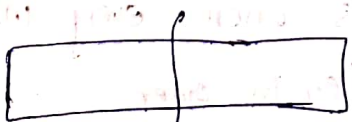
$$\Rightarrow T(n) = O(n^2)$$

Insertion Sort:

working; Insert one element at a time from an unsorted list to a partially sorted list.

→ It works in (n-1) passes

→ At every pass, we have the below scenerio



Take an element from unsorted list and put it in the sorted list in its position

Eg : A: 4 3 8 6 9 2

initially size of sorted list = 1

A: ⟨4 | 3 8 6 9 2⟩

Now insert 3 into partially sorted list

pass 1 → A: ⟨3 4 | 8 6 9 2⟩

pass 2 → A: ⟨3 4 8 | 6 9 2⟩

pass 3 → A: ⟨3 4 6 8 | 9 2⟩

pass 4 → A: ⟨3 4 6 8 9 | 2⟩

pass 5 → ~~A: ⟨2 3 4 6~~

A: ⟨2 3 4 6 8 9⟩

From this we say insertion sort works as external sorting.

Since at any point of time insertion sort doesn't both about where the unsorted list is stored.

Time Complexity.

(i) Best Case:

Best case happens when: every insertion finishes with one comparision. i.e., Inc order

⇒ Time = no of comp + no of swaps

= 0 (n-1) + 0

⇒ T.C = O(n)

(ii) Worst Case:

Worst case happens when elements are in descending order.

time = no of comp + no of swaps = $O(n^2)$

→ In general, it is found that TC of insertion sort depends on no of inversions.

⇒ TC = $O(n+d)$

where d is no of inversions.

Thus if d=0 (i.e., no inversions)

TC = $O(n)$

Eg: Consider a list with n inversions. Then TC for insertion sort is _____?

Sol:

TC = $O(n+d)$

$= O(n+n) = O(n)$

25/10/20

Non-Comparision based sorting techniques

(i) Radix Sort:

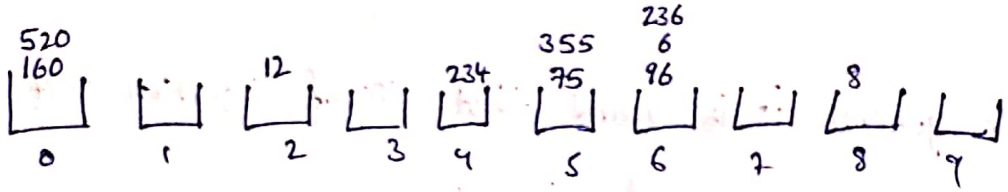
→ Here the term radix indicates base of the system.

→ If base is 'b' we take 'b' no of buckets.

Eg: A: 234 60 96 12 8 75 6 355 520 236

Pass 1:

Distribute the numbers into buckets based on unit digit.

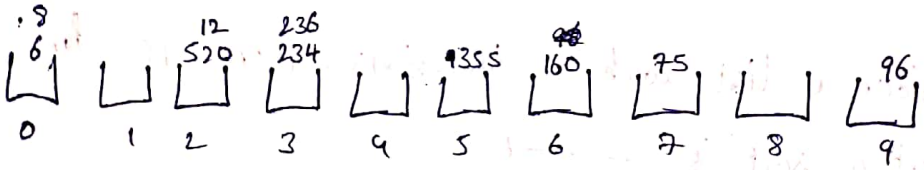


∴ Elements after pass 1

160 520 12 234 75 355 96 6 236 8

Pass 2:

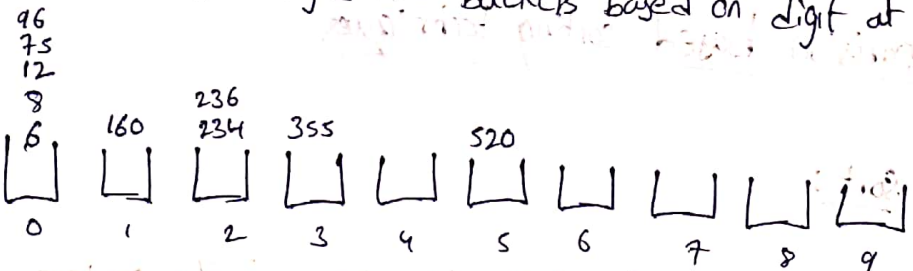
Distribute the digits into buckets based on digit at 10's place



∴ 0 6 8 520 12 234 236 355 160 75 96

Pass 3:

Distribute the digits into buckets based on digit at 100's place.



∴ 6 8 12 75 96 160 234 236 355 520

Drawback:

→ Special implementation is needed if we need to sort negative numbers.

→ Sorting fractional numbers is not possible.

Time complexity:

$T.C = O(d(n+b))$ → since 'b' is const we can also write $TC = O(dn)$

$d \rightarrow$ max digits in a number

$n \rightarrow$ size of list

$b \rightarrow$ base

If 'x' is max number possible in the base, then

$$d = \log_b x$$

$$TC = O((n+b)\log_b x)$$

$$= O(n \log_b x)$$

let $x \leq n^c$

c is a constant

$$T.C = O(n \log_b n)$$

Thus radix sort performs good when $b \geq n$

If $b \geq n$

$$O(n \log_b n) = O(n)$$

(H/82) Method 1:

we use heap for this (similar to problem (H/64))

we have mn elements.

size of heap = no of lists = n

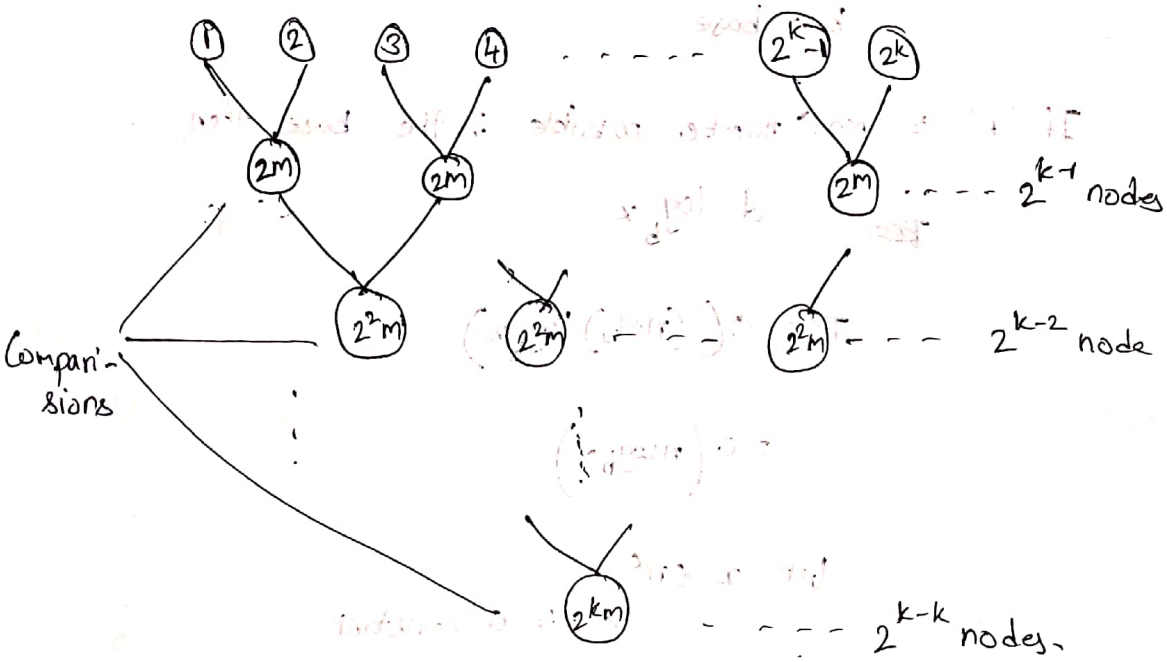
∴ del & insert on heap → log n

$$\therefore O(mn \log n)$$

Method 2:

This can also be solved by considering optimal merge pattern.

let $n = 2^k$



∴ total comparisions requiied

$$(2^1m)(2^{k-1}) + (2^2m)(2^{k-2}) + \cdots + (2^km)(2^{k-k})$$

$$= (2^km + 2^km + \cdots + 2^km)k$$

$$= nm\log n$$

(#83) Let b be a base.

no of digits $d = \log_b n^k = k\log n$

TC of radix sort $= O(d(n+b))$

$= O(dn)$

$= O(n \cdot k\log n)$

$= O(n\log n)$

H/88 If assume base is n

then TC of radix sort = O(n)

Disjoint Set Data structure

Operations:

(i) Create(): Create the set will required element

(ii) Union(): Combining the two sets into one.

(iii) Find(): returns the representative element of the set.

Every set has a representative element

Representation:

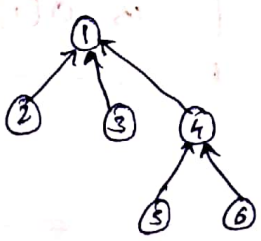
i) Forest:

Eg: $S_1 = \{1,2,3\}$

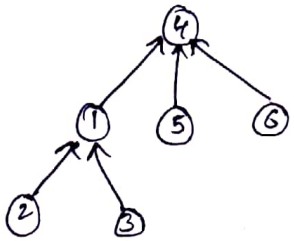
$S_2 = \{4,5,6\}$

$S_3 = \{7,8,9\}$

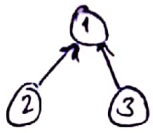
$S_1 \cup S_2$: This can be done by choosing one of the representative element as new representative element for $S_1 \cup S_2$



(or)

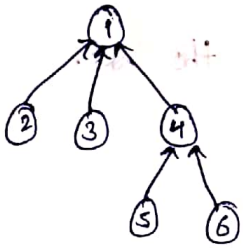


Find operation:



Here Find(1) = 1
Find(2) = 1
Find(3) = 1

Here Find(4) = 4
Find(5) = 4
Find(6) = 4



Here Find(1) = 1
Find(2) = 1
~~Find(3)~~ Find(3) = 1
Find(4) = 1
Find(5) = 1
Find(6) = 1

TC of find depends on ~~highest~~ height of the tree.

(ii) Array based Representation:

$S_1 = \{1,2,3\}$ $S_2 = \{4,5,6,7\}$ $S_3 = \{8,9\}$

Let 1, 4, 8 be corresponding representative elements.

Array:

	1	2	3	4	5	6	7	8	9
parent	-1	1	1	-1	4	4	4	-1	8
element	1	2	3	4	5	6	7	8	9

With amortized analysis it is found that TC of both union & find is : O(1)

Kruskal's algorithm for MCST

```
Algo kruskal(E, COST, n, T, mincost)
{
    //E is edge set in G. G has n vertices
    //COST[1..n, 1..n] is cost matrix
    //T is set of edges in MCST

    1. real mincost COST(1..n, 1..n);
    2. integer Parent(1:n), T(1:n, 1:2), n;
    3. Construct a heap out of edge cost using heapify  ——— O(e)
    4. Parent ← -1
    5. i ← mincost ← 0
    6. while i<n-1 and heap not empty  ——— O(e)
        a) delete a min cost edge (u,v) from the heap
           and reheapify using ADJUST                  } ——→ O(loge)
        b) j ← FIND(u); k ← FIND(v);
        c) if j≠k then i ← i+1
               T(i,1) ← u; T(i,2) ← v
               mincost ← mincost + COST(u,v).          } ——→ O(1)
               call UNION(j,k)
           endif
       repeat
    7. if i≠n-1 then print ("no spanning tree") endif
    8. return;
}
```

∴ TC = O(e log e)

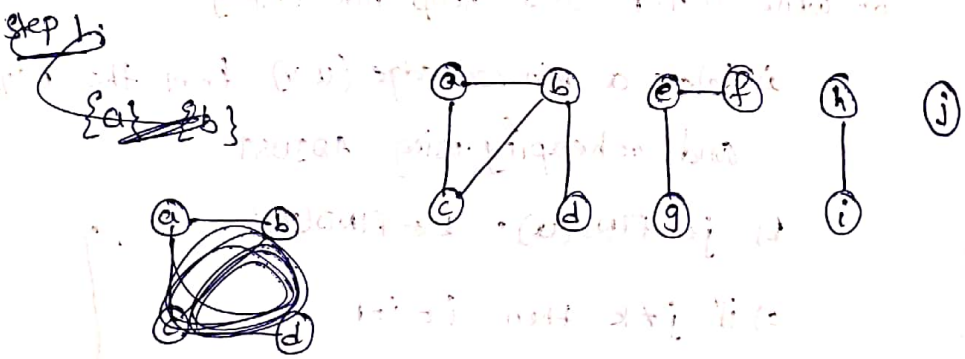
S.C = O(n) [i.e., space for parent array]

Algorithm to find connected components:

```
CONNECTED COMPONENTS(G)
{
    1. for each vertex v ∈ V[G]
           do MAKE-SET(v)
    2. for each edge (u,v) ∈ E[G]
           if find(u) ≠ find(v) then
               union(u,v)
}
```

Eg:

Eg: Consider below graph & its tracing



Step 1. -

{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}

Step 2.

edge (a,b)

{a,b} {c} {d} {e} {f} {g} {h} {i} {j}

edge (b,c)

{a,b,c} {d} {e} {f} {g} {h} {i} {j}

edge {b,d}

{a,b,c,d} {e} {f} {g} {h} {i} {j}

edge (a,c):

find(a) = find(c)

∴ no union will be perform

edge (e,f) & edge (e,g)

{a,b,c,d} {e,f,g} {h} {i} {j}

edge (h,i)

{a,b,c,d} {e,f,g} {h,i} {j}

∴ 4 connected components.

TC = O(e) [assuming that find & union can be done in O(1)]

Sum of Subsets (SOS):

→ Given a set of n-elements and another positive integer 'M' It is required to determine whether there exists a subset of given numbers wher sum equal M.

→ TC of brute force — $O(2^n)$

$$SOS(n,M) = True \quad \text{if } M=0, n\geq 0$$

$$= False \quad \text{if } n=0, M>0$$

$$= [SOS(n-1,M) \;||\; SOS(n-1, M-a[n])]$$

↳ only if $A[n] \leq M$

Tabulation Method for SOS

n=5 ; A = ⟨2, 8, 4, 11, 9⟩ ; M = 6

S[0...n, 0...M]

	0	1	2	3	4	5	6
0	T	F	F	F	F	F	F
1	T	F	T	F	F	F	F
2	T	F	T	F	F	F	F
3	T	F	T	F	T	F	T
4	T	F	T	F	T	F	T
5	T	F	T	F	T	F	T

→ soln i.e., sos(5,6)

sos[1,1] = sos(0,1) || sos(0,-1)

= F

sos[1,2] = sos(0,1) || sos(0,0)

= F || T

= T

∴ sos[5,6] = T

Think how to backtrack and obtain the subset required

Note:

If s(i,j) is true then s(k,j) is true ∀ k ≥ i

```
Algo sos(A, n, M)
{
    for i ← 0 to n
    {
        for j ← 0 to m
        {
            sos(i,j) ← implement code for recursive eqn
        }
    }
}
```

TC: O(nM)

However if value of n is too large like 2^n (or) n^n then

TC will be $O(n \cdot 2^n)$ (or) $O(n \cdot n^n)$

-and this case brute time complexity is better.

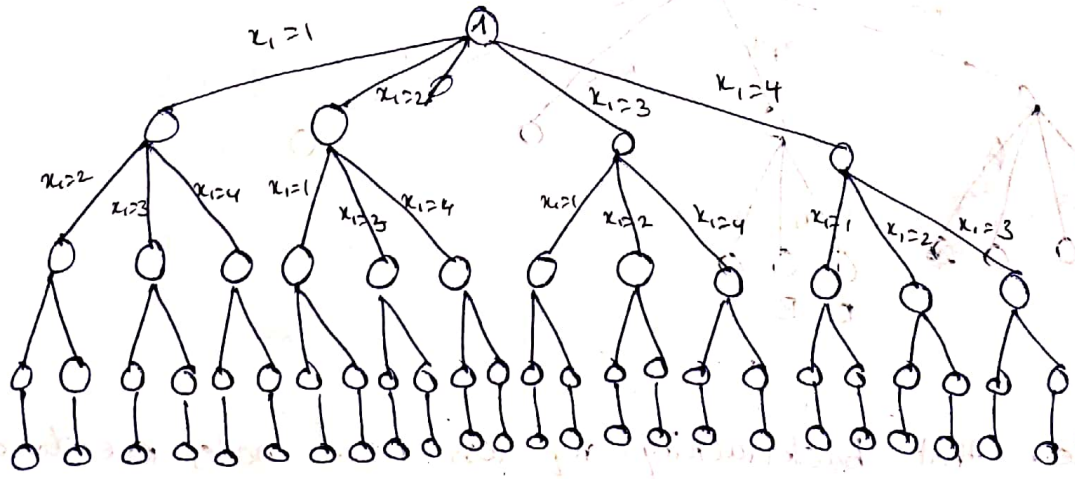
→ The time complexities like $O(n \cdot M)$ are called psuedo-polynomical TCs.

Introduction to Backtracking and Branch & Bound

Backtracking (BFS) is algorithm design strategies that uses depth first searching (BFS) & to find feasible/optimal soln in solution space (state space tree)

Eg: n-queens

Consider 4 queens problem

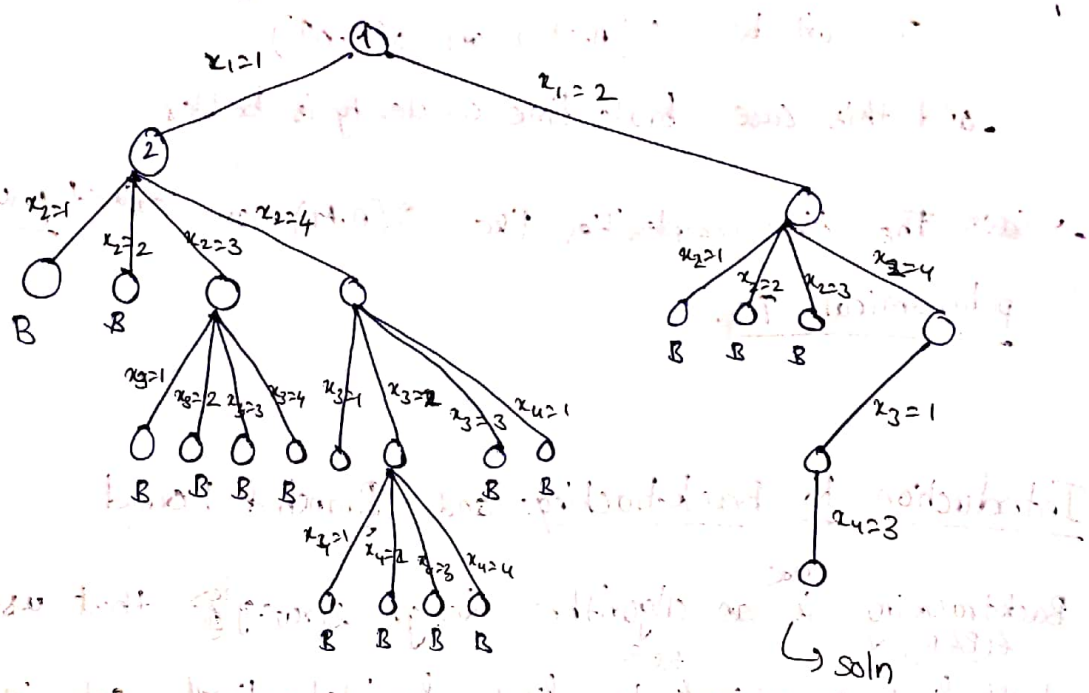


State space tree

If DFS is used to search for soln in state space tree then it is called Backtracking.

If BFS is used then it is called Branch & Bound.

DFS : Backtracking

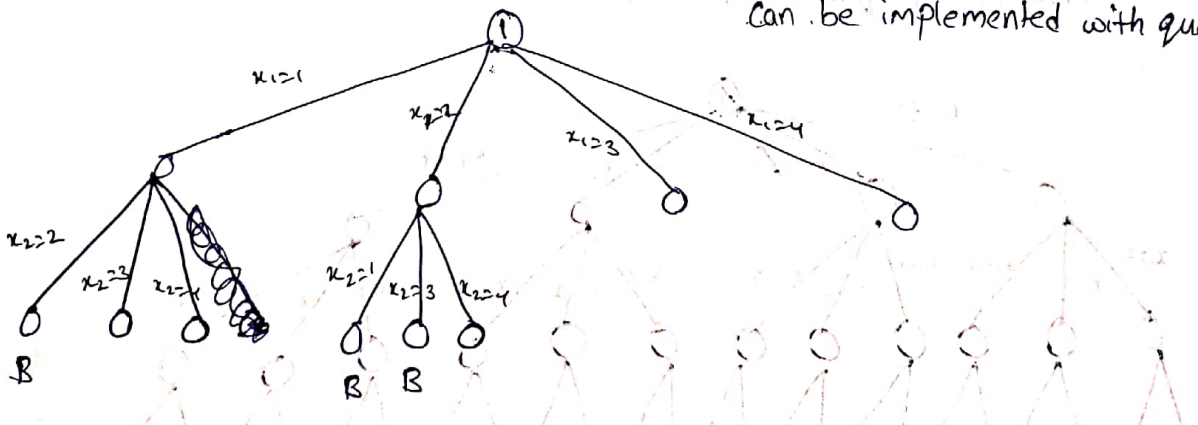


B → Bound

(Soln is not ~~feasible~~

BFS : Branch & Bound :

Can be implemented with queue



→ It clear that 'backtracking' req less no of nodes ~~to~~ explored compared to branch & bound.

Thus n-queens can be efficiently implemented using backtracking.

→ similarly we can also solve backtracking in this approach.

→ Also TSP can be more efficiently solved using branch & bound.

Kadane's algorithm for max sum sub array

→ Bruteforce TC : $O(n^2 * n) = O(n^3)$

all subarray, computing sum

→ DP approach (Kadane's algo)

~~we maintain 3 variable~~

we perform n iterations (n is no of elements)

we maintain 3 variables max_sum_so_far, current_sum, index.

~~In index says from which iteration we start summing~~

~~Thus n iterations correspond to index value from 1 to n.~~

Index says till which index max sum is obtained.

```
Algo kadane ( )
{
    // start_index, end_index // solun.
    // MaxSum

    for  i=1  to  n
    {
        temp_start = i;  Max_sum_sofar=0;  curr_sum=0;  index=-1;
        for j=i to n
        {
            curr_sum += a[j];
            If (curr_sum > max_sum_so_for)
            {
                Max_sum_so_far = cur_sum;
                index = j;
            }
        }
    if ( max_sum_sofar > MaxSum)
    {   MaxSum = Max_sum_so_far;
```

```
            start_index = i;
            end_index = index;
        }
    }
    return [start_index, end_index];
}
```

TC: $O(n^2)$

LIS:

→ one approach is solving this using LCS.

→ Let A be given ~~arrg~~ array and store the sorted array of A in another array B.

Now LCS(A, B) will be the LIS(A).

→ Also we can use pure DP approach and solve this.