Algorithm	Worst Case	Average Case
Insertion Sort	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n^2)$	$O(n \lg n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \lg n)$	—
Counting Sort	$O(k+n)$	$O(k+n)$
Radix	$O(d(n+k))$	$O(d(n+k))$
Bucket	$O(n^2)$	$O(n)$

Outline 1

Algorithm

An algorithm is any well-defined computational procedure that takes values or set of values as input and produces value, or set of values as output

=> Input sequence for an algo is called instance of problem

=> Algo is said to be correct if it produces correct output for every instance

Algorithm as Technology

=> Algorithms that ~~are~~ efficient uses less memory space and consumes small time are considered to be more efficient algorithms

Efficiency

↳ Diff algos for solving same problem

↳ Efficiency depends on running time and space

↳ Ex 2 algos for sorting

Insertion : Merge

$c_1 \cdot n^2$: $c_2 \cdot n \lg n$

n^2 : $n \lg n$

Best for small i/p : Best for large i/p

Insertion Sort

Algorithm

ex array

40	20	60	10	50	30

for j=2 to A.length
  K=arr[j]
  i=j-1

while i>0 and arr[i]>K
  arr[i+1]=arr[i]
  i=i-1

arr[i+1]=K

Dry Run

j=2

40 20 60 10 50 30
i=1 j=2 K=20

i>0(T) & 40>20

arr(2)=40
i=0

arr[i]=20

j=3

20 40 60 10 60 30
i=2 j=3 K=60

i>0 T & 40>60 X
i=2
X

i=1
i>0 T & 20>60 X
X

Analyzation

∵ Analyzation of insertion sort depends upon the comparisons and total no. of swaps

⇒ Best Case
↳ Ascending order

comparisons = n-1
swaps = 0
$O(n)$

⇒ Worst Case
↳ array is in descending order

comparisons : 1, 2, 3, ... n-1
swaps : 1, 2, 3, ... n-1

$$O\left(\frac{n(n-1)}{2}\right)$$

$$O\left(\frac{n^2-n}{2}\right)$$

$$O(n^2)$$

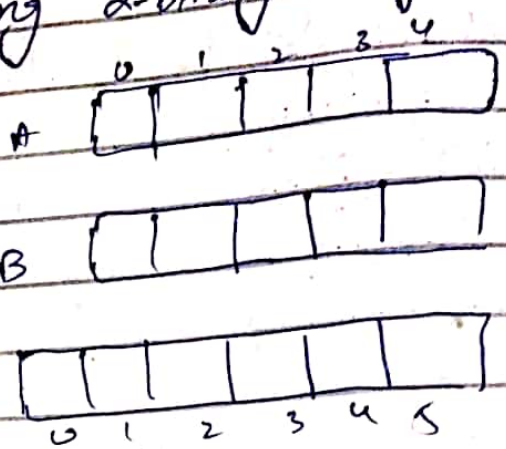
⇒ Average Case

$$O(n^2)$$

Advantages of Insertion Sort
(i) Stable ⇒ don't change order of same element in which they appear in array
(ii) inplace ⇒ don't consume extra space.

Exercise 2.1-4

Algo for adding 2-binary arrays



carry = 0

for i = n to 1

$C[i+1] = A[i] + B[i] + carry \% 2$

$carry = A[i] + B[i] + carry / 2$

$C[0] = carry$

Analyzing Algorithms

↳ means predicting the resources that algorithm requires

eg:
- memory
- communication bandwidth
- hardware
- computational time

↳ Model for algo implementation
- > RAM model (random-access machine)
- > Generic one processor

=> RAM model: instructions are executed one after the other

=> Analyzation of Insertion Sort
$O(n^2)$

Depends upon n of input
Max input -> Max time
Allread closely sorted - min time

So time depends upo nature of input.
Consider constant time to exe each statement
S = 1,2,3, .. i time $c_1, c_2, c_3 \ldots c_i$
Time taken when a statement of loop is executed i.e test statement denoted by t_j

		cost	Times
1	j=2 to A.length	c_1	n
2	key = Arr[j]	c_2	$n-1$
3	i = j-1	c_3	$n-1$
4	while i>0 & Arr[i] > key	c_4	$\sum_{j=2}^{n} 2^{t_j}$
5	Arr[i+1] = Arr[i]	c_5	$\sum_{j=2}^{n} 2^{(t_j-1)}$
6	i = i-1	c_6	$\sum_{j=2}^{n} 2^{(t_j-1)}$
7	Arr[i+1] = key	c_7	$n-1$

Total Time = $c_1 n + c_2(n-1) + c_3(n-1)$

$T(n)$ $+ c_4 \sum_{j=2}^{n} 2^{t_j} + c_5 \sum_{j=2}^{n} 2^{(t_j-1)}$

$+ c_6 \sum_{j=2}^{n} 2^{(t_j-1)} + c_7(n-1)$

=> For Best Case

array is already sorted

$T(n)$

comparision at statement 4 = 0

for every $j = 2, 3, 4, \ldots n$

$t_j = 1$

$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1)$

$+ c_7(n-1)$

$\therefore c_5 \sum_{j=2}^{n} (t_j - 1) = 0$

$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$

⇒ For worst Case

array is in descending order

we need to compare each element with entire array for every $j = 2, 3, \ldots n$

so

$t_j = j$

$j = 2 + 3 + \cdots + (n-1) + n$

$S = 1 + 2 + 3 + \cdots + (n-1) + n \quad -1$

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

so

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} t_{j-1} = \sum_{j=2}^{n} j-1 = \frac{n(n-1)}{2}$$

$T(n) = c_1 n + c_2(n-1) + \cdots$

$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_8}{2}\right) n^2 + \left(\cdots\right) n + \left(\cdots\right)$

Growth of function

We deal with the asomptotic efficiency of algorithms.

This is concerned with how the running time of an algorithm increases with the size of the input.

=> Usually, the algorithm that is asompototically more efficient will be the best choice for all but very small inputs.

Example

Insertion sort

worst case $O(n^2)$

Merge sort

Worst case $O(n \lg n)$

when size of n increases at a point merge sort beats insertion sort

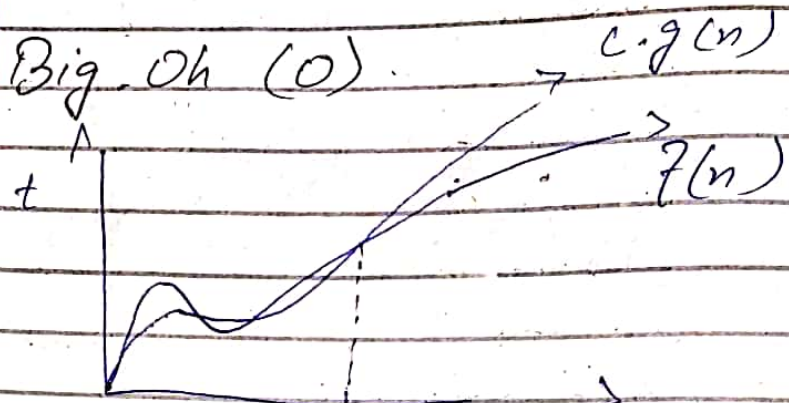
i.e

Insertion sort running time > Merge sort running time

Asymptotic Notations

=> Mathamatical way of representing Time Complexity

1) Big Oh (O)



=> Upper Bound

=> Worst Case

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

$c > 0$
$n \geq k$
$k \geq 0$

Ex

$f(n) = 2n^2 + n$

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

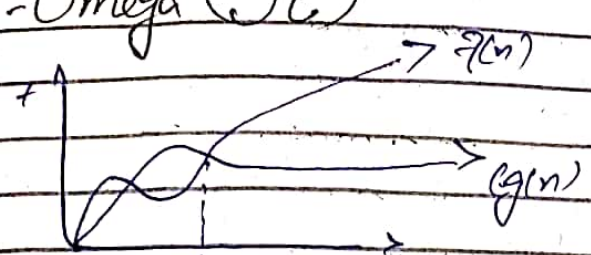
$2n^2 + n \leq c \cdot g(n^2)$ ← dominant value of function i.e n^2

$2n^2 + n \leq c \cdot n^2$

$2n^2 + n \leq 3n^2$

∵ c should be equal to a value that satisfy function inequality

Big-Omega (Ω)



=> Best Case
=> greatest lower Bound

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

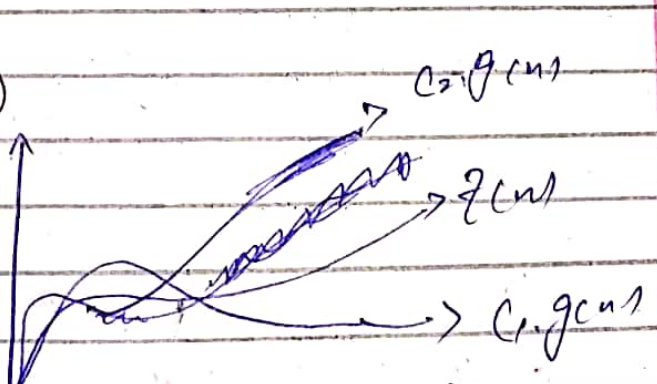
e.g

$2n^2+n \geq c \cdot g(n^2)$ ← dominant last closest

$2n^2+n \geq cn^2$ ← c should be such value that satisfy inequality

$2n^2+n \geq 2n^2$

Theta (Θ)



=> Average Case

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

ex

$$2n^2 \leq 2n^2+n \leq 3n^2$$

Little-Oh(o)

$f(n) < c \cdot g(n)$

Little Omega (ω)

$f(n) > c \cdot g(n)$

Properties of Asymptotic Notations

Big Oh (O) $f(n) \leq c \cdot g(n)$

Big Omega (Ω) $f(n) \geq c \cdot g(n)$

Big Theta (Θ) $c_1 g(n) \leq f(n) \leq c_2 g(n)$

(i) Reflexive Property

↳ self checking i.e $a = a$

e.g Big Oh (O) $f(n) \leq c \cdot g(n)$

$n^2 \quad n^2$

Reflexive Property holds

(ii) Symmetric Property

if $a > b$ then $b > a$

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

$a \leq b$

For symmetric $\Rightarrow n^2 \leq n^3$ and $n^2 \geq n^3$

but this is false so

Symmetric property doesn't hold in Big Oh

⇒ It hold in only Big Theta $c_1 g(n) \leq f(n) \leq c_2 g(n)$

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

$f(n) = g(n)$

	Reflexive $a \geq b, b \geq a$	Symmetric $a \geq b \& b \geq a$	Transitive $a \geq b \geq c$
Big O $f(n) \leq c \cdot g(n)$ eg $n^2 \leq n^2$	✓	✗	✓
Big (Ω) $f(n) \geq c g(n)$ $n^2 \geq n^2$	✓	✗	✓
Big (Θ) $c_1 g(n) \leq f(n) \leq c_2 \cdot g n$ $1n^2 \quad 2n^2 \quad 3n^2$	✓	✓	✓
Small (o) $f(n) < c \cdot g(n)$ $n^2 \quad n^3$	✗	✗	✓
Small (ω) $f \cdot c g(n) < f(n)$ $n^2 \quad n^3$	✗	✗	✓

Comparision of different type of functions

$O(c) = O(1)$ e.g. $O(10^3) = O(10^{99}) = O(1)$

$O(\log\log n)$ (log base 2) e.g. $\log(\log(1000)) = \log 10 = 3.3\ldots$

$O(\log n)$ e.g. $\log 1000 = 10$

$O(n^{1/2})$ e.g. $(1000)^{1/2} = 33.3\ldots$

$O(n)$ e.g. $O(1000) = 1000$

$O(n \log n)$ e.g. $O(1000 \log(1000)) = 10 \times 1000$

$O(n^2)$ e.g. $O(1000^2)$

$O(n^3)$

$O(n^k)$ e.g. $k \geq 4$. e.g. (1000^4)

$O(2^n)$ e.g. (2^{1000})

$O(n^n)$ e.g. $(1000)^{1000} < O(2^n)$

Divide and Conquer Approach

```
DAC(P)
{
  if(small(P))
  {
    S(P)
  }
  else
  {
    divide P into P1, P2, P3 ... Pk
    Apply DAC(P1), DAC(P2), ... DAC(Pk)
    Combine(DAC(P1), DAC(P2), DAC(P3) ... DAC(Pk))
  }
}
```

Application of Divide and Conquere

(i) Binary Search
(ii) Find Max and Minimum number
(iii) Quick Sort
(iv) Merge Sort
5) Strassen's Matrix Multiplication

Quick Sort
↳ DAC method.

35 50 15 25 80 20 90 45

Qs expelled

How it works

Time Complexity

Ω , O , Θ

Recurrence Relation

Solving Reivrence Relation using Substitution Method

=> For Binary Search

$$T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ 1 & \text{for } n = 1 \end{cases}$$

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

$$T(n/2) = T\left(\frac{n}{2} \times \frac{1}{2}\right) + c$$

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

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

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

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

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

:

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

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

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

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

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

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

$$= T(n/16) + c + 3c$$

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

$$\vdots$$

$$= T(n/2^k) + kc$$

$\Rightarrow$ Now Terminate $T(n/2^k) = T(1) = 1$

$$n/2^k = 1 \implies T(1) + kc$$

$$n = 2^k \qquad \boxed{1 + kc}$$

$n = 2^k$

$\log n = k \log 2$

$k = \log n$

for Complexity in Term of $n \Rightarrow O(\log n)$ Ans log base c

Adj of Substitution Meth.

=> Give recurrence relation of any kind of problem

=> Master theorem solves only specific kind of problems

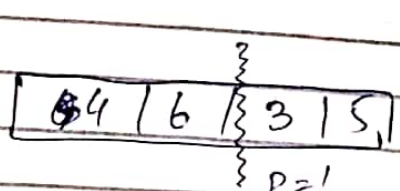
=> Always give correct answer

=> Slow due to more computation than master theorem

Example $T(n) = \begin{cases} 2T(\frac{n}{2}) + n & \text{otherwise} \\ 1 & \text{if } n = 1 \end{cases}$

Pseudo Code for Merging

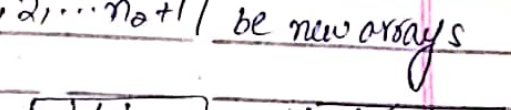
Merge (A, p, q, r)



(i) $n_1 = q - p + 1$ $\quad n_1 = 2$

(ii) $n_2 = r - q$ $\quad n_2 = 2$

(iii) Let $L[1, 2, \dots, n_1+1]$ & $R[1, 2, \dots n_2+1]$ be new arrays



(iv) for $i = 1$ to n_1

$L[i] = A[p+i-1]$

(v) for $j = 1$ to n_2

$R[j] = A[q+j]$

(vi) $L[n_1+1] = \infty$

(vii) $R[n_2+1] = \infty$

(viii) $i = 1$

(ix) $j = 1$

(x) for $k = p$ to r

if $L[i] \leq R[j]$

$A[k] = L[i]$

$i = i+1$

else $A[k] = R[j]$

$j = j+1$

Master Theorem

=> Faster than substitution method bcz it includes less mathematical calculations

=> Can be applied to only a specific type of problem

i.e

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

where $a \geq 1$ & $b > 1$

examples

$T(n) = 8T\left(\frac{n}{2}\right) + n^2$ ⇒ $a = 8, b = 2$

$T(n) = T\left(\frac{n}{2}\right) + C$ ⇒ $a = 1, b = 2$

Solution

$$T(n) = n^{\log_b a} [U(n)]$$

Here

$U(n)$ depends on $h(n)$

and

$$h(n) = \frac{f(n)}{n^{\log_b a}}$$

Relation b/w $h(n)$ and $U(n)$

if $h(n)$ is	then $U(n)$
n^r, $r > 0$	$O(n^r)$
n^r, $r < 0$	$O(1)$
$(\log n)^i$, $i \geq 0$	$\frac{(\log_2 n)^{i+1}}{i+1}$

Example

Q8 $T(n) = 8T(n/2) + n^2$

$a = 8, \; b = 2, \; f(n) = n^2$

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

$= n^{\log_2 8} \cdot U(n)$

$= n^{\log_2 2^3} \cdot U(n)$

$= n^{3 \log_2 2} \cdot U(n)$ $\quad \log_2 8 = 3$

$T(n) = n^3 \cdot U(n)$ —(1)

$U(n) \Rightarrow$ depends on $h(n)$

$h(n) = \dfrac{f(n)}{n^{\log_b a}}$

$= \dfrac{n^2}{n^3} = n^{-1}$

for $h(n) = n^x$ and $x < 0$

then

$U(n) = O(1)$

put $U(n) = O(1)$ in eqn (A)

$= n^3 \cdot 1$

$= n^3$

Example #2

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

$a = 1,\ b = 2,\ f(n) = C$

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

$$= n^{\log_2 1}\ U(n) \qquad \log_2 1 = 0$$

$$= n^0 \cdot U(n)$$

$$= U(n)$$

$U(n) \Rightarrow$ depends upon $h(n)$

$$h(n) = \frac{f(n)}{n^{\log_b a}} = \frac{C}{1} = C$$

$h(n)$ doesn't satisfy 1st and 2nd case so we have to use 3rd case

$$h(n) = C = (\log_2 n)^0 \cdot C = (\log_2 n)^0 \cdot C$$

now it satisfies 3rd case

i.e

$$(\log n)^i = \frac{(\log_2 n)^{i+1}}{i+1}$$

so

$$U(n) = \frac{(\log_2 n)^{0+1}}{0+1} \cdot C$$

$$T(n) = (\log_2 n) \cdot C$$

Strassen's Matrix Multiplication

=> Multiplaition of Matrix (2×2).
=> Theory Book

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}_{2\times 2} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}_{2\times 2} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}_{2\times 2}$$

Formulas:

P: $(A_{11} + A_{22}) \times (B_{11} + B_{22})$ => Sum of Diagonals of matrix (A) × (B)

Q: $B_{21}(A_{11} + A_{22})$

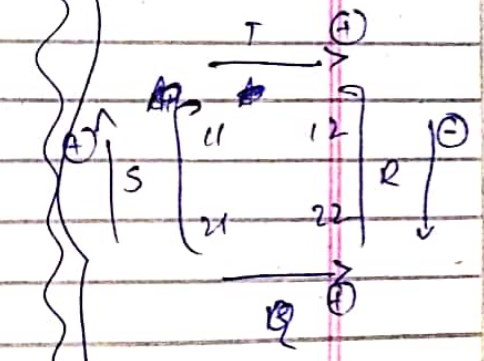
R: $A_{11}(B_{12} - B_{22})$

S: $A_{12}(B_{11} + B_{21})$

T: $B_{22}(A_{11} + A_{12})$

U: $(B_{21} + B_{11}) \times (A_{11} + A_{12})$

V: $(A_{21} + A_{22}) \times (B_{12} - B_{22})$



B_{21} A_{12} A_{22} B_{22}

U => By combining S & T

V => By combining Q & R

P Q R S I U V

C_{11} : P + S − T + V

C_{12} : R + T

C_{21} : Q + S

C_{22} : P + R − Q + V

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

Example

$A = \begin{bmatrix} 5 & 6 \\ -4 & 3 \end{bmatrix}$ $B = \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix} = \begin{bmatrix} \end{bmatrix}$

$A_{11} = 5$ $B_{11} = -7$

$A_{12} = 6$ $B_{12} = 6$

$A_{21} = -4$ $B_{21} = 5$

$A_{22} = 3$ $B_{22} = 9$

8×2

$P = 16$

$Q = 5 \times (-1) = -5$

$R = 5 \times (-3) = -15$

$S = 6 \times (-2) = -12$

$T = 9 \times (11) = 99$

$U = (-2) \times (11) = -22$

$V = (-1) \times (-3) = 3$

$C_{11} = 16 - 12 - 99 + 3 = -92$

$C_{12} = -15 + 99 = 84$

$C_{21} = -5 - 12 = -17$

$C_{22} = 16 - 15 + 5 + 3 = 9$

$$\begin{bmatrix} -92 & 84 \\ -17 & 9 \end{bmatrix}$$

16
-12
4
-99
-95
+3
-92

-96
-12
-108
+16
-92

Radix Sort

=> Non-comparison based algorithm for sorting array

Array : 904, 46, 5, 74, 62, 1

=> Relative position should not be changed i.e stable sort algo

Example

A = { 904, 46, 5, 74, 62, 1 }

Step 1 : make equal no of bit i.e 904 has 3

904
046
005
074
062
001

Step 2 : sort according to LSB (least significant bit) i.e bit on most right side.

=> stable sorting

001
062
904
074
005
046

Step3 : Sort according to next bit on left of LSB

001 → 001
062 → 904
904 → 005
074 → 046
005 → 062
046 → 074

Step3 : Sort accord to MSB (most significant bit)

001 → 001
904 → 005
005 → 046
046 → 062
062 → 074
074 → 904

Counting Sort Algorithm

=> Non-Comparison Based Algorithm
=> Given input size = n
=> Given rang is 'k'

∴ Drawback
=> Restricted Rang ie (1—10) (100 elements in this range)
=> Input size must be known

Example
array of 7 elements having range (1-4)
2 1 2 3 1 2 4

Step 1 : take a new array of size 1 greater than range
In this case it will be 5
and initilize it to 0

1	0
2	0
3	0
4	0
5	0

Step 2:
Traverse given array one by one and add, the no of items an element repeats, in new array

1	0
2	0
3	0
4	0
5	0

→

1	2
2	3
3	1
4	1
5	0

Step 3 : Now write array according to above result
ie
[1, 1, 2, 2, 2, 3, 4] => sorted.

Time Complexity

↳ depends upon no of elements and range of array

$O(n+k)$

Space Complexity

$O(k)$

Adj

Works in linear type.

Dis Adj

⇒ can occupy very large space

ex:

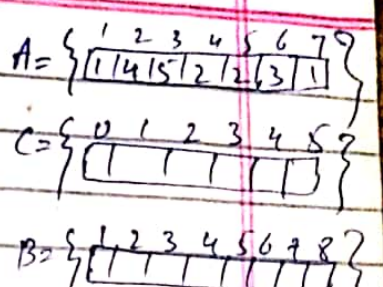
$A = \{2, 2000, 3\}$

⇒ n should be known

⇒ k " " "

Algorithm

Counting Sort (A, B, k)



(i) let C[0 ... k] be a new array

(ii) for i = 0 to k

(iii) C[i] = 0

(iv) for j = 1 to A.length

(v) C[A[j]] = C[A[j]] + 1

(vi) for i = 1 to k

(vii) C[i] = C[i] + C[i-1]

(viii) for j = A.length to 1

(ix) B[C[A[j]]] = A[j]

(x) C[A[j]] = C[A[j]] - 1

Step 2,3 → initializing array
step 4,5 → updating value in C, how much time a number repeated
Step 6,7 → adding 1st and 2nd index and updating in 2nd index
Step 8,9,10 → calculations and finalizing results ☺

Heap Tree

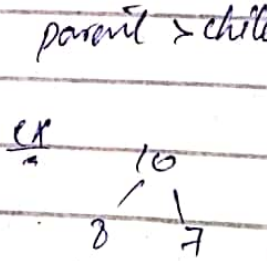
Topics
insertion, deletion, heap sort, heapify
array rep, priority queue.

=> every heap tree must follow 2 properties

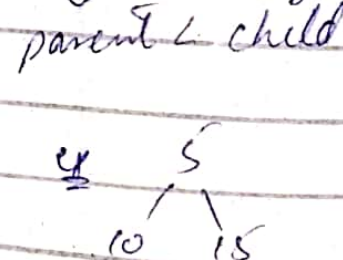
(i) Structural property
↳ must be ACBT

(ii) Ordering property
↳ Max heap or Min heap

↓ parent > child



↓ parent < child



Heap Tree Construction

2 Methods

=> insert key one by one in given order

$O(n \log n)$

=> Heapify Method (vip)

$O(n)$

➡ Insert key one by one

example 14, 24, 12, 11, 25, 8, 35

max-heap

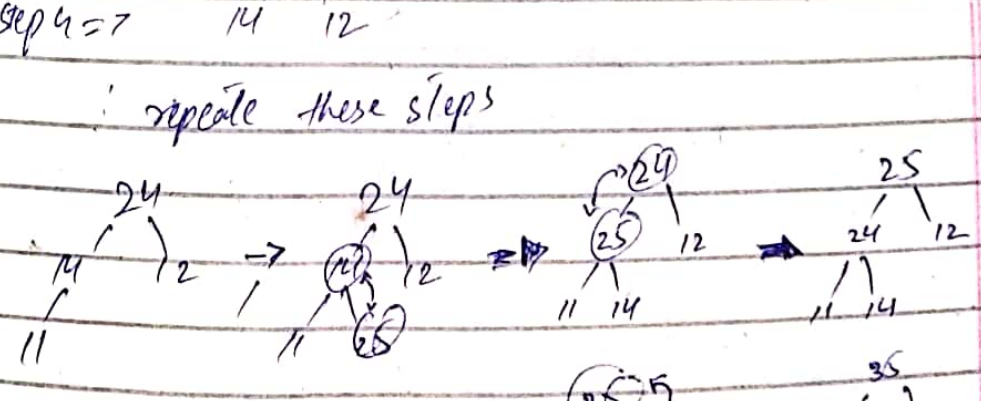
step 1 => 14

step 2 =>

step 3 => compare and check if satisfy heap property and swap

step 4 =>

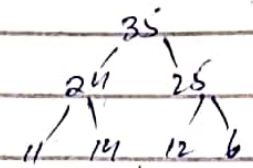
repeate these steps

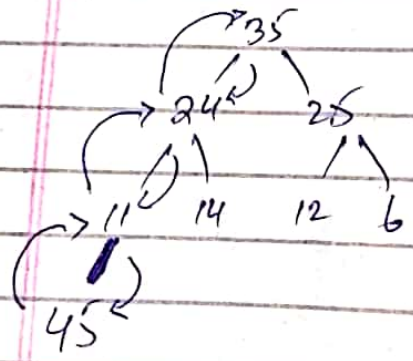


Algo Analysis

(i) Inserting key one by one
to insert a key in empty heap takes O(1) time

(ii) In worst case
consider example of adding 45 in





We have to compare new element to 'height of tree' no of times

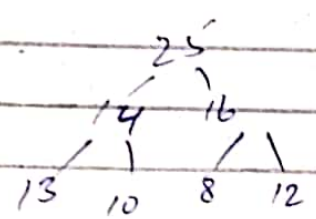
Height of ACBT is given by $\log n$

So time taken to add a new element in heap will be $\log n + \log n = 2 \log n$ and swaps will be so $O(\log n)$

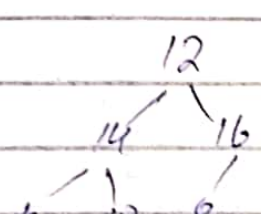
=> For adding n no of element in heap time taken will be
$O(n \log n)$

Deletion Process:

A = { 25, 14, 16, 13, 10, 8, 12 }

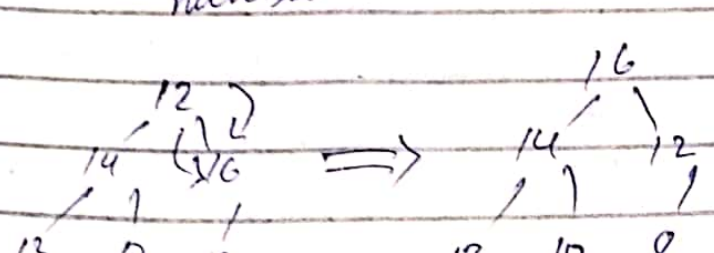


=> Step 1 delete root node and replace the right most child to root position



25

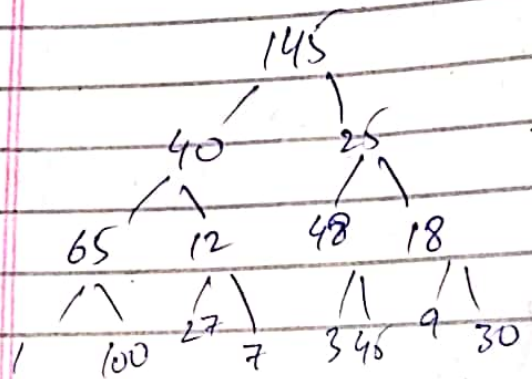
=> Step 2 now check heap property and call heapify function and replace replaced root node with max number



Heapify Method

Ans 145, 40, 25, 65, 12, 48, 18, 1, 100, 27, 7
3, 45, 9, 30

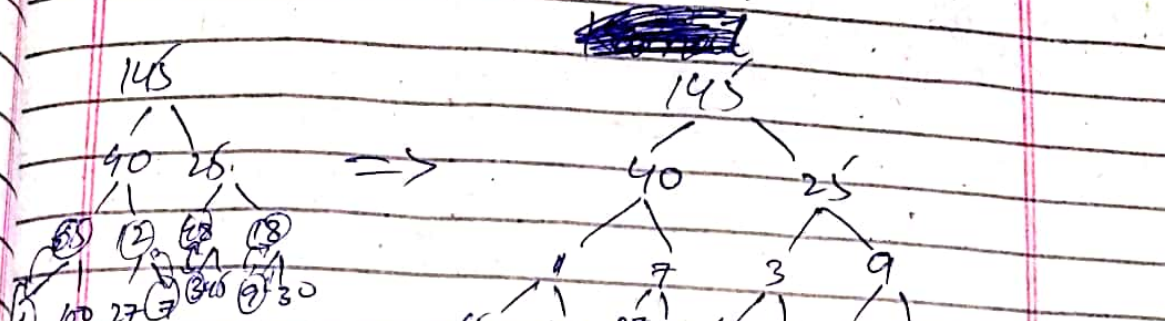
Step 1: Create a simple ACBT



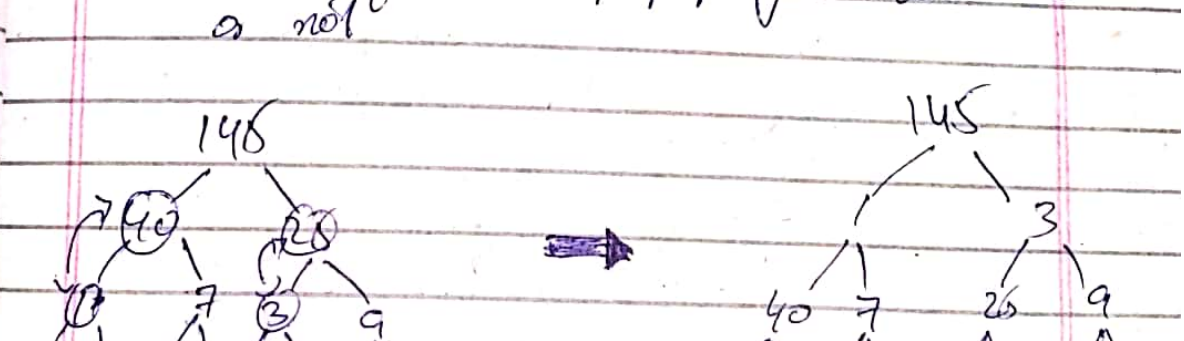
∴ height of tree = $\log n$

∵ leaf nodes will not require any comparison or swapping so we begin heapify from internal nodes that are above leaf nodes.

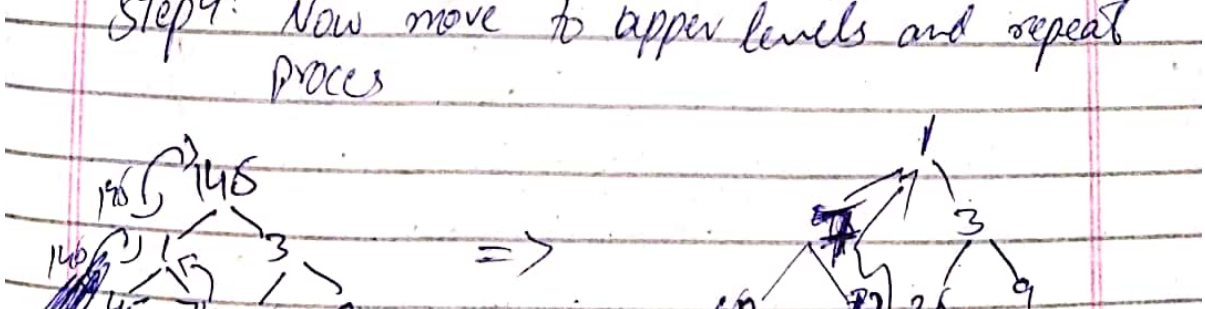
Step 2: Let's make min heap start checking from its right most internal node if it satisfies heap property and perform swaps.



Step3: Now move to upper level and check if H don't satisfy heap property then swap and check in lower levels if their heap property is disturbed or not

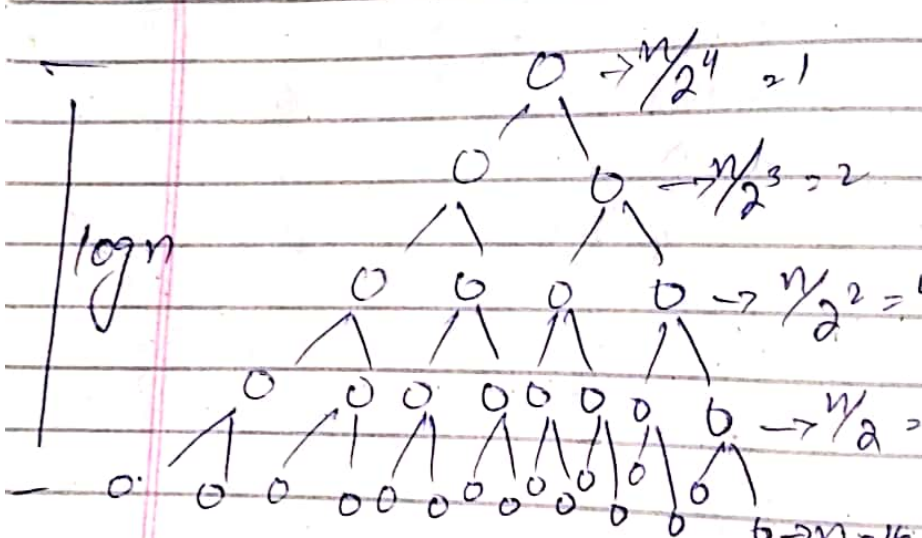


Step4: Now move to upper levels and repeat proces



Note:
in heapify we ignore leaf nodes
The total no of leaf nodes = n/2
so by not performing comparisons
operations on n/2 elements,
the time complexity becomes smaller

Analysis



if there are 'n' nodes in leaf level, then upper levels will have n/2 nodes

Calculation

Total swaps = S

=> For n'

n no of nodes in leaf

total swaps = 0

=> For n/2'

n/2 no of nodes in interal node

total no of swaps for each node = 1

so

this level will have n/2 * 1 swaps

=> For n/2²

n/2² no of nodes in internal level

so total no of swaps for n/2² nodes will be = $\frac{n}{2^2} * 2$

$= \frac{n}{2^{\log n}} * \log n$

Total swaps

$$S = \frac{n}{2^0} * 0 + \frac{n}{2^1} * 1 + \frac{n}{2^2} * 2 \cdots + \frac{n}{2^{\log n}} * \log n$$

$$= n \left[\frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{\log n}{2^{\log n}} \right] \quad (i)$$

Multiplying eqn (i) with $\frac{1}{2}$

$$\frac{S}{2} = n\left[\frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \cdots + \frac{\log n}{2^{\log n+1}}\right]$$

$$\frac{S}{2} = n\left[\frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{\log n - 1}{2^{\log n}} + \frac{\log n}{2^{\log n+1}}\right]$$

Subtract (2) from (1)

$$S - \frac{S}{2} = n\left[\left[\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{\log n}}\right] - \frac{\log n}{2^{\log n+1}}\right]$$

Using GP formula $R < 1$

$$S = a\left(\frac{1-r^n}{1-r}\right)$$

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

$$\frac{S}{2} = n\left[\left(\frac{\frac{1}{2}\left(1 - \frac{1}{2^{\log n}}\right)}{\frac{1}{2}}\right) - \frac{\log n}{2^{\log n+1}}\right]$$

$$= \frac{1}{2}\left[\frac{1}{2^2} - \frac{1}{2^2}\right]$$

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

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

$$S = n\left[\left(1 - \frac{1}{2^{\log n}}\right) - \frac{\log n}{2^{\log n + 1}}\right]$$

$$S = n\left[\left(\frac{2^{\log n} - 1}{2^{\log n}}\right) - \frac{\log n}{2^{\log n} \cdot 2^1}\right]$$

$$= n\left[\left(\frac{n-1}{n}\right) - \frac{\log n}{2n}\right] \quad \because 2^{\log n} = n^{\log 2} = n$$

$$= n\left[\frac{2n - 2}{2n} - \frac{n(\log n)}{2n}\right]$$

$$S = \left(n - 1 - \frac{\log n}{2}\right) \quad \boxed{O(n)}$$

Algorithm for Max-Heapify

A = array of node in which new node is added in
i = index
l = left of i index
r = right of i index

MAX-HEAPIFY (A, i)

1. l = Left (i)
2. r = Right (i)

∵ Left(i) and Right(i) are functions, return index of left and right of 'i' index

3. if l ≤ A.heap-size & A[l] > A[i]
4. largest = l
5. else largest = i
6. if r ≤ A.heap-size and A[r] > A[largest]
7. largest = r
8. if largest ≠ i
9. exchange A[i] with A[largest]
10. MAX-HEAPIFY (A, largest)

Heap Sort Algorithm

for max-heap

HEAP SORT (A)

1. BUILD-MAX-HEAP (A)
2. for i = A.length to 2
3. exchange A[1] with A[i]
4. A.heapsize = A.heapsize - 1
5. MAX-HEAPIFY (A, 1)

BUILD-MAX-HEAP (A)

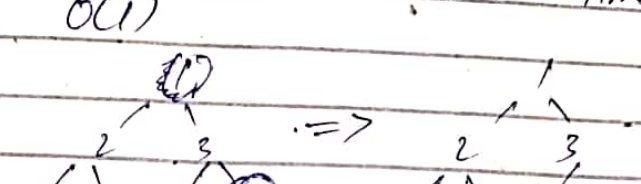
1. A.heapsize = A.length
2. for i = ⌈A.length/2⌉ to 1
3. MAX-HEAPIFY (A, i)

Deletion in Heap Tree

=> Best Case

we delete element on the lowest right node

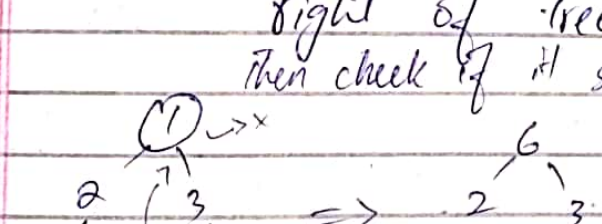
so it will take linear time complexity O(1)



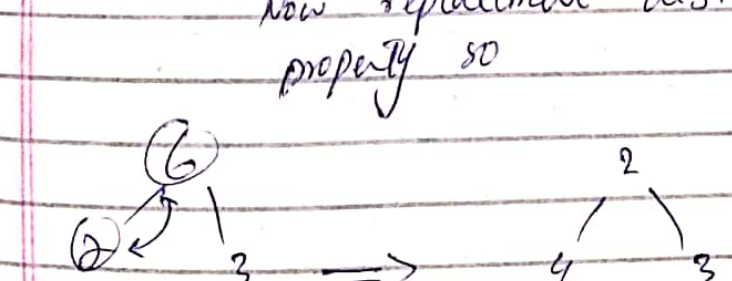
=> Worst Case

We delete element on the root node and then replace with the last node on right of tree

Then check if it satisfies heap property



Now replacement disturbed heap property so



O(nlgn)

Heap Sort

=> 2 methods are executed to sort a heap (i) Build-Heap (ii) Deletion

Steps

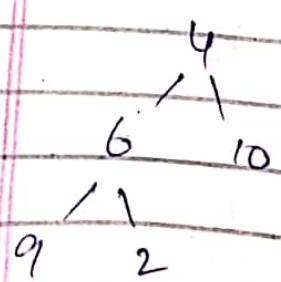
① First we make a simple tree using elements in array

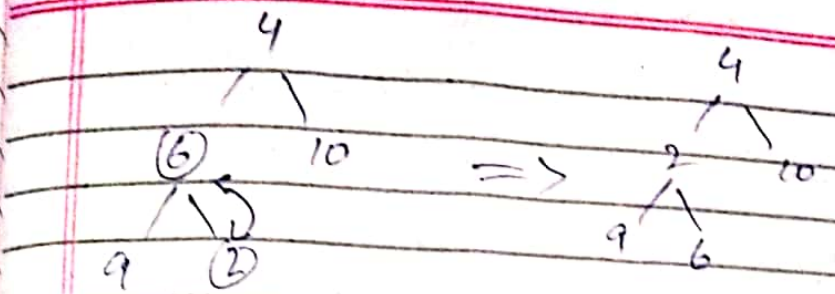
② Call heapify method to attain heap property

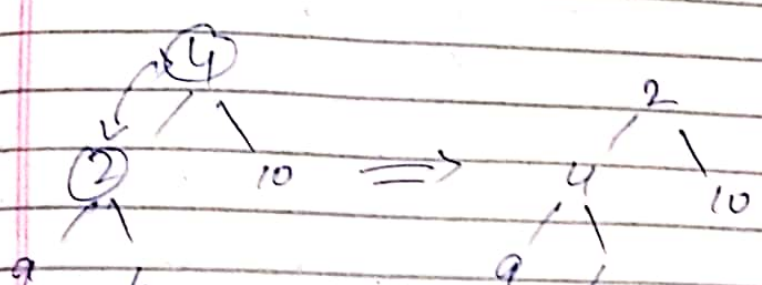
③ delete the root element and replace the right most leaf element with root

④ After replacement check if it satisfy heap property,
If it doesn't then call heapify proces

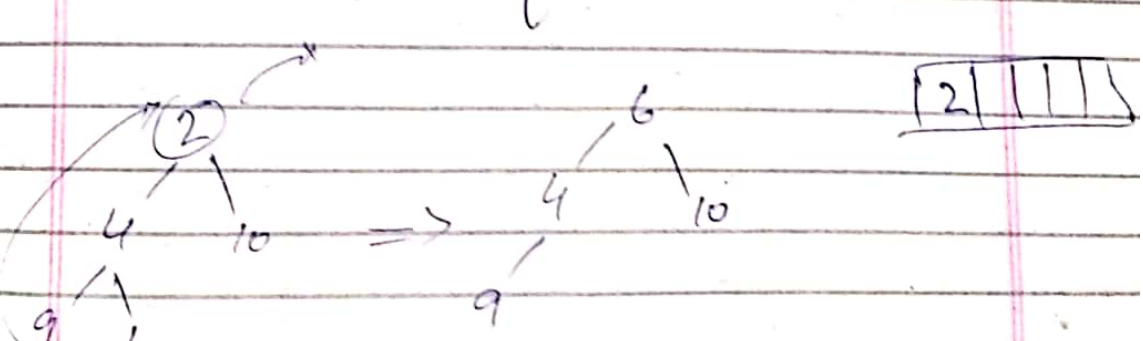
Example (min heap)
4, 6, 10, 9, 2



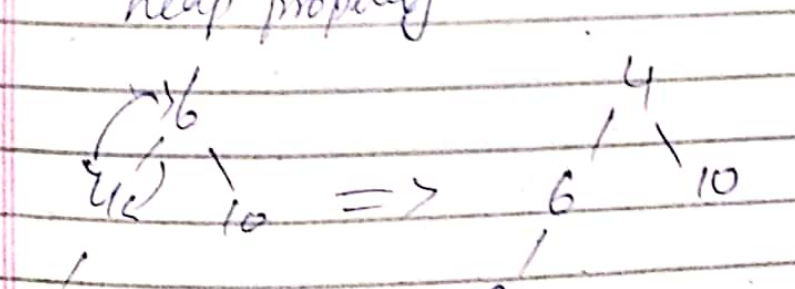


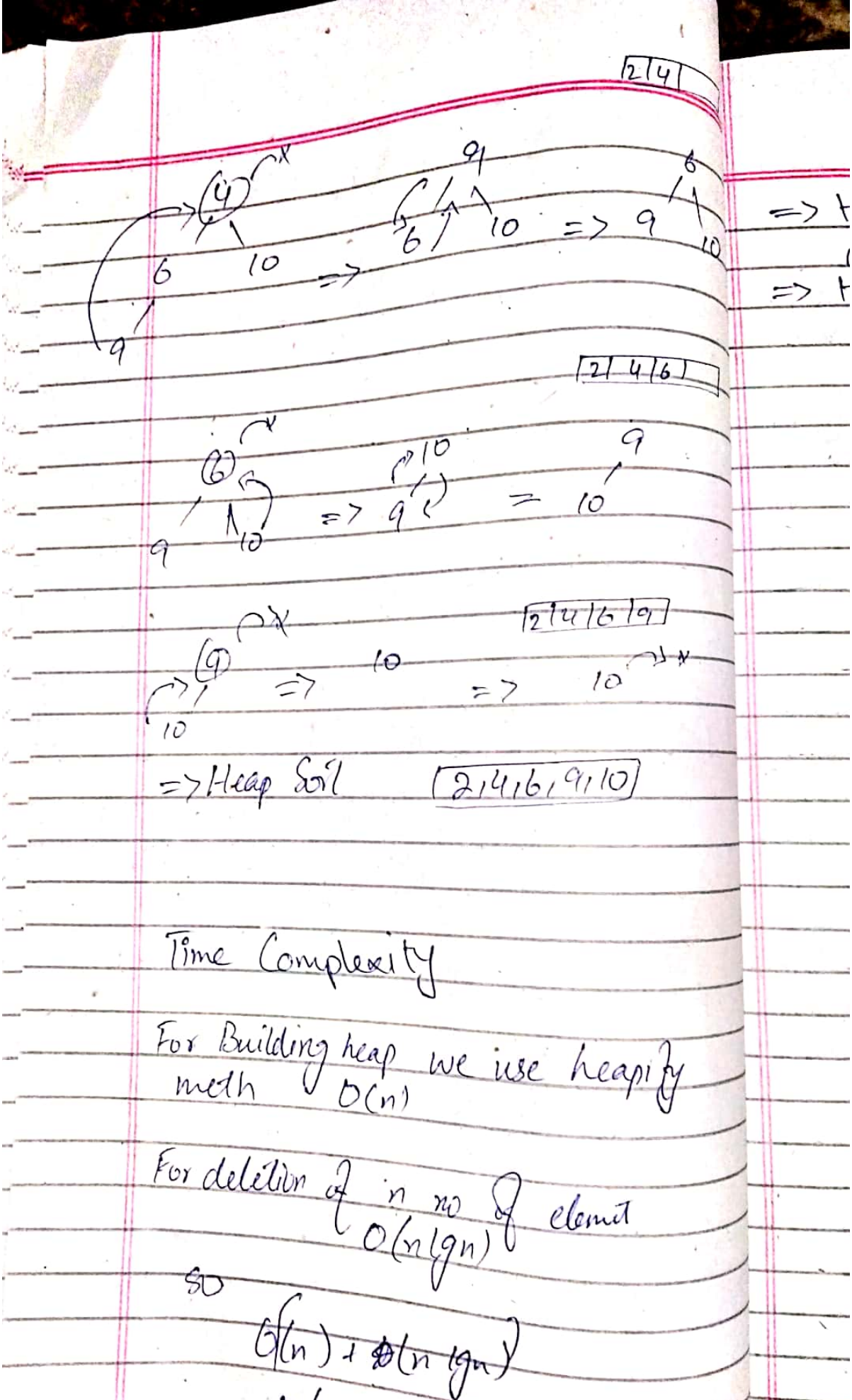


Now heap is built, and we called deletion proces from root index



After replacement and deletion check for heap property





=> Heap Sort [2,4,6,9,10]

Time Complexity

For Building heap we use heapify meth $O(n)$

For deletion of n no of elemt $O(n \lg n)$

So

$$\Theta(n) + \Theta(n \lg n)$$

$$\Theta(n \lg n)$$

=> HeapSort is in place
as it don't consume extra space
=> Heap Sort is unstable sort

Greedy Algorithms

↳ Follows local optimal choice at each stage with intend of finding global optimum.

∴ At each stages, choses best option to reduce the overall cost.

↳ Fesible Solution
   ⇒ based on selection criteria we chose fesible solutions

↳ Optimal solution
   ⇒ Minimum cost
   ⇒ Maximize Profit

Applications:
- Knapsack Problem
- Job Sequencing
- Min Spaning Tree
- Huffman Coding
- Dijkstra's Algo

Knapsack Problem

↳ capacity give
↳ greedy algo
=> We consider greediness about both aspects either weight and profit

Example

objects	ob_1	ob_2	ob_3
Profit	25	24	15
Weight	18	15	10

Knapsack capacity (W) = 20

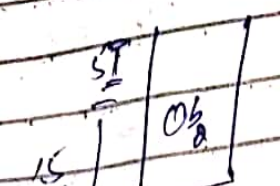
=> For knapsack problem we took both cost/weight and profit in consideration for optimal solution

Step 1: - We take profit/weight ratios of all the objects

ob_1	ob_2	ob_3
1.3	1.6	1.5

Step 2: Fill in knapsack bag one by one in ascending ratios.

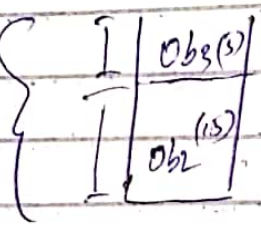
=> 1.6



1.6 ratio obj is filled in bag
rem cap = 5
profit = 24

=> Now fill 5th position of 1.5 ratio object

$$P = \frac{5}{10_2} \times 15 = 7.5$$



So total profit will be
24 + 7.5 = 31.5

Note:
If we are greedy about profit or weight
=> For profit
After calculation total profit = 31
=> For weight
" " " weight = 28.2
=> So Profit/weight ratio is Optimal.

Algo for KnapSack

(1) for i=1 to n
(2) calculate profit/weight ration } O(n)
(3) Sort obj's in ascending order using some sorting algo } O(n lg n)
(4) for i=1 to n
(5) if M>0 & Wi≤M
(6) M = M - Wi;
(7) Profit = Profit + Profit(i);
(8) else
(9) break;
(10) if (M>0)
(11) Profit = Profit + Profit(i)(M/Wi);

O(n)

= O(n) + O(n lg n) + O(n)

so

O(n lg n)

Huffman Coding

=> Greedy Algorithm
=> Compression techniques to encode chars.

Ex.

a=50	e=3
b=20	f=2
c=30	
d=5	

M=100

=> If we use ASCII technique we will require 7 bits to encode
i.e 0-127 require 7 bits $= 2^7 = 128$
ASCII encoding uses 128 characters
so for 100 characters => Total Bits $= 7 \times 100 = 700$

=> If we use binary digits i.e 0,1s

a 000
b 001
c 010
d 011
e 100
f 101

we will req 3 bits to rep each char
so
Total no of bits $= 3 \times 100 = 300$ bits

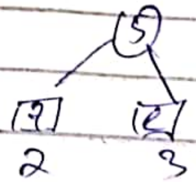
=> We use huffman coding technique

Step 1

take the charcter having lowest fs and right on left and on right tithin node of tree

f = 2, e = 3

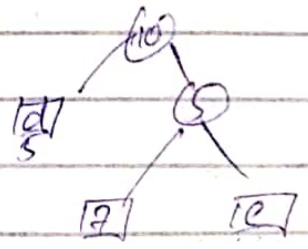
and write root node = Sum of both

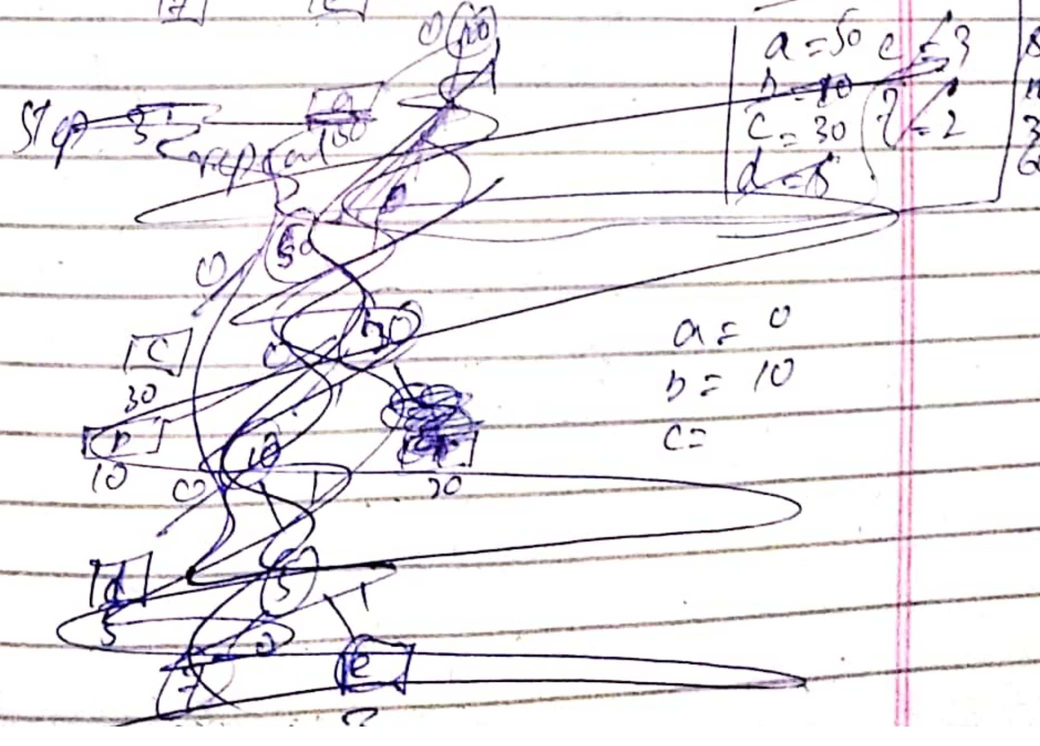


Step 2:

Now take frequnce greater than summed freq

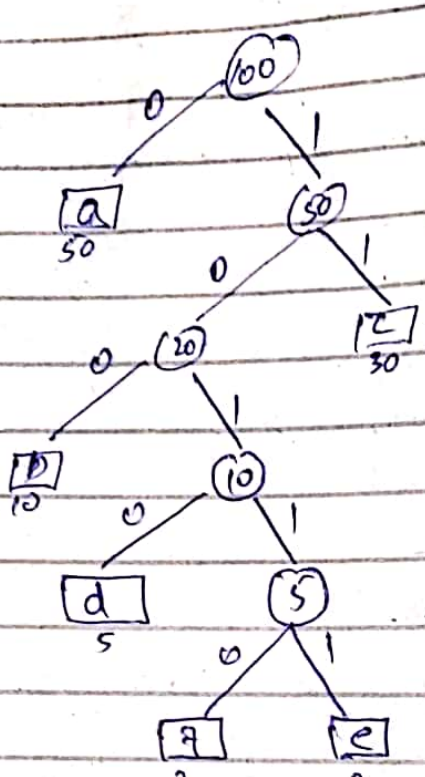
i.e d = 5





Step 3: repeat

a=50
b=10
c=30
d=5
e=3
f=2



Step 4 : Now assign bits to the edges of tree
0 on left => smaller side
1 on right => bigger side

Now Bits used to rep

a = 0 = 1 x 50 = 50
b = 100 = 3 x 10 = 30
c = 11 = 2 x 30 = 60
d = 1010 = 4 x 5 = 20
e = 10111 = 5 x 3 = 15
f = 10110 = 5 x 2 = 10
= 185 bits

=> So average bits used to represent 1 character will be $= \frac{185}{100} = 1.85$ bits

which is lesser than

ASCII $= \frac{700}{100} = 7$ bits

&

Binary $= \frac{300}{100} = 3$ bits.

=> So total 185 bits used to rep 100 characters in Huffman code.

Job/Task Scheduling Algo

Greedy algorithm

=> Assumptions:

(i) Uniprocess
i.e one process at a time

(ii) No preemption

(iii) Every job will cost 1 unit time

Algorithm

Step 1: Arrange all jobs in decreasing order of profit

Step 2: For each job (me), do linear search to find particular slot in array of size 'n'

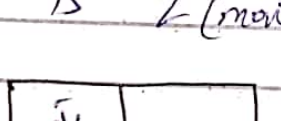
∴ n = max deadline date
∴ m = total jobs

Example

	J_1	J_2	J_3	J_4
Profit	50	15	10	25
Deadline	2	1	2	1

Let us add jobs according to maximum profit in gant chart

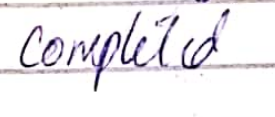
Gant chart will have max value upto 2 because the max deadline in for job to be done is 2 (max/day/duration)



P=50

then add J_3 because J_2 & J_4 are supposed to be done with 1 unit time.

Now we only have J_3 that can be completed upto 2 unit time So

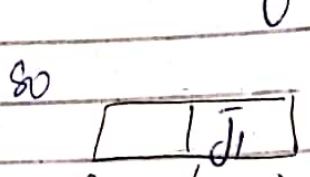


50+10
= 60 profit

=> This technique doesn't give optimal solution.

To get optimal solution

Step 1:
add job having larger profit in slot near to the dead line of job
J_1 has $50 profit and a deadline of 2 unit time



P=50

Step 2:
Repeat

P = 25 + 50
P = 75

So this will be optimal solution

Analysis
Step 1 = $O(n \log n)$
sorting n no of jobs
Step 2 = $n \times m$ or $n \times n = n^2$
Linear search for slot for every element
$O = n \log n + n^2$
$O(n^2)$

Spanning Tree

Spanning Tree

A connected subgraph 'S' of Graph (V, E) is said to be a spanning tree

if and only if

=> 'S' should contain all vertices of G

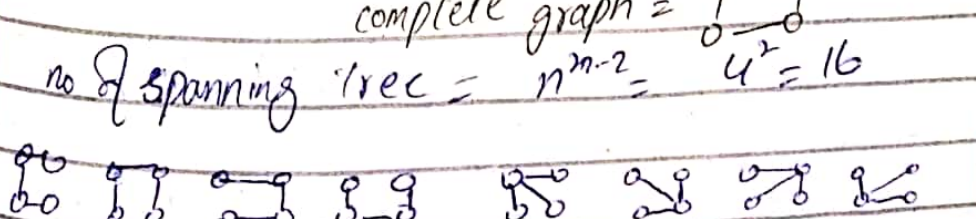
=> 'S' should contain (|V| - 1) edges

↳ Spanning Tree is acyclic graph
↳ Its all vertices are/must be connected
↳ Total no of edges would by = (V-1).
↳ Spanning Trees are formed by complete graphs in which all vertices are connected.
↳ No of spanning trees that can be formed by a complete graphs are equal to = n^{n-2}
∴ n is no of vertices.

Example

complete graph = 

no of spanning tree = $n^{n-2} = 4^2 = 16$

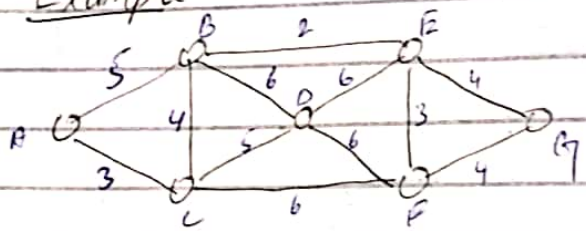


Kruskal's Algorithm

↳ used to find minimum cost spanning tree from a connected graph

↳ in intermediate level of process spanning tree is disconnected in Kruskal's Algorithm but provide accurate result and propper spanning tree after completion

Example



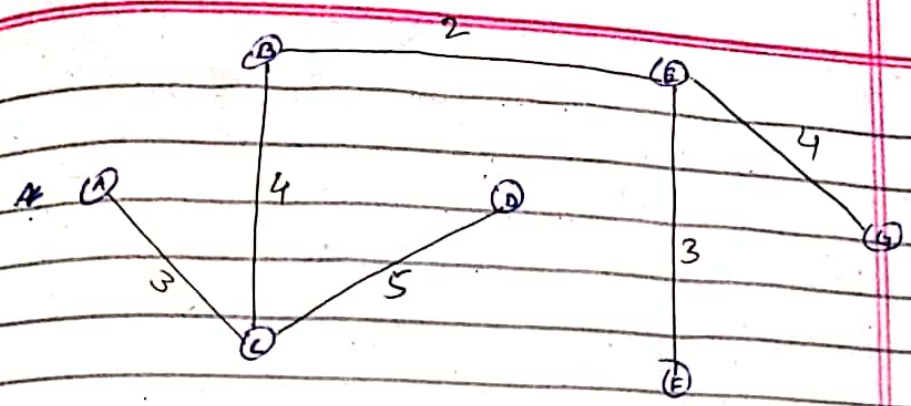
Step 1:

We first arrange all the edges according to increasing order of edge cost

The connect vertices using these sorted edges one by one

Avoid loop creation

=> No of edges will be = V-1 = 8-1 = 7



Increasing order of edges: BF, AC, EF, BC, EG, FG, CD, AB, DB - -
2 3 3 4 4 4 5 5 6 6 6 -

↑ avoid as it may cause loop

Algorithm

Construct min heap with 'e' edges
$O(e)$ => using heapify function

Add edges one by one in spanning tree and cycle should not be create

Best Case $O(n-1)$ (n-1) edges

Worst Case $O(\log e)$ e edges

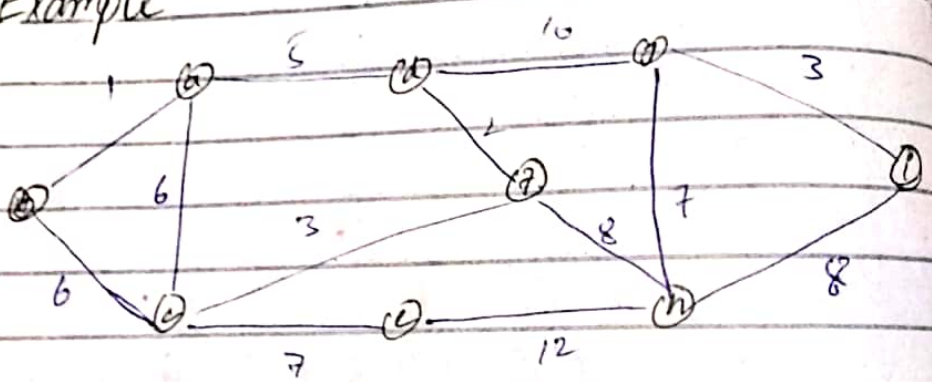
So for n=e edges
$O(e \log e)$

Prim's Algorithem

↳ used to find minimum spanning tree from a complete connected graph

↳ it in intermediate level, the minimum spanning tree in Prim's algo is always connected

Example

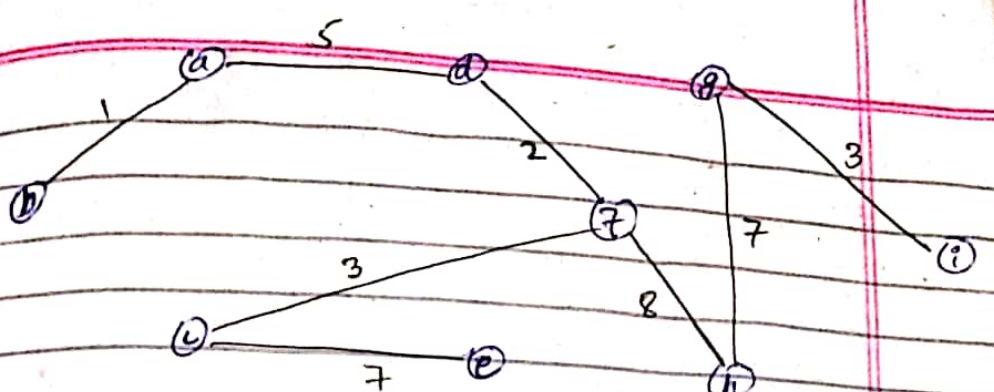


Steps:

We can start from any ~~edge~~ vertex we want

At each vertex we check for min edge from all the available edges

Avoid selecting edge that can cause loop creation



select 'b'

Available edges : ~~ab~~, ~~bc~~, ~~ad~~, ~~ae~~, df, dg, fe, fh, ch, eh, hg
~~prim's~~ array 1 6 9 6 2 10 3 8 7 12 4

hi, gi
8 3

ab → ad → bf → fc

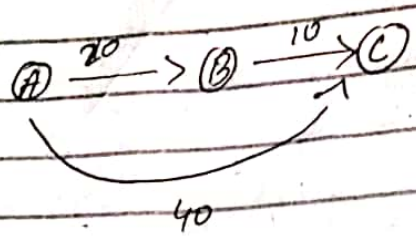
we don't chose ae & bc as they can form loop

fh → ce → hg → gi

So our total vertices were 9

Total no of edges = 9-1 = 8 edges.

Dijikstra's Algorithm



Relaxation's:

if $d(u) + c(u,v) < d(v)$

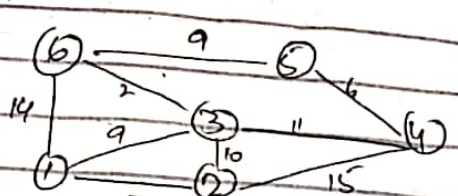
~~then~~ $d(v) = d(u) + c(u,v)$

We consider distance from a single source as (∞)

A	B	C
∞	∞	∞
AB	(20)	40
	(20)	10

We start from a singl selected source and move to its connected nodes.
After visting all nodes and cal distance from starting source ~~we~~ selet the minium distance value and visit its adjacent nodes.

Example :



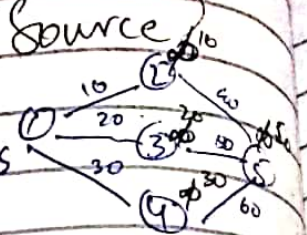
Source	Destination				
1	2	3	4	5	6
	∞	∞	∞	∞	∞
	(7)	9	∞	∞	14
1,2	(7)	(9)	22	∞	14
1,2,3	(7)	(9)	20	∞	(11)
1,2,3,6	(7)	(9)	20	20	(11)
1,2,3,6,4					

Algorithm

```
Dijkstra's (Graph, source)
  Create vertex of Q
  for each vertex v in graph        } Building Heap
      dist[v] = ∞                   } O(v)
      add v to Q
  dist[source] = 0
  while Q is not empty
      v = Extract min[Q]
      for each neighbour V of v
          Relax (V, v)
```

Algorith Analysis

Dijkstra's (Graph, Source)



Create vertex set Q

For each vertex u in graph

dist[u] = ∞] O(V)

add u to Q] Building heap using heapify O(V)

dist[source] = 0

while Q is not empty

v = Extract-min[Q]] V.log V

for each neighbour u of v

Relax(v, u)] E.log V

= O(V) + O(V) + O(V.log V) + O(E log V)

= O(E log V)

∵ V.log V => extracting value and heapifying function for every vertex (V)

∵ E log V => relaxing every v and checking every edge (E) attach with (V)

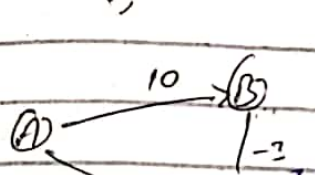
A | O | A,C | A,C,B

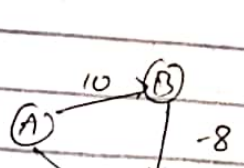
∴ This wors alg we an ed hig

Scanned with CamScanner

Dijkstra's Algo for negative edges

-ve weighted edge





	A	B	C
0	0	∞	∞
		10	(5) → relaxed
A,C		(10)	(5) can't be change
A,C,B		(10)	(5)

∴ This algo graph is working for dijkstra's algo bcz, even if we don't relax C and go to C via B edge the cost is high.

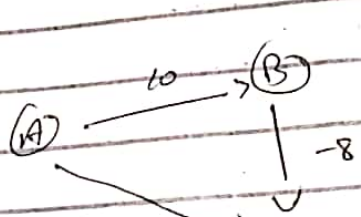
	A	B	C
0	0	∞	∞
B		10	(5)
A,C		(10)	(5)
A,C,B		(10)	

∴ This graph isn't working bcz C is relaxed already and if we go to C via B edge the cost is low and we can't change already relaxed vertex's cost

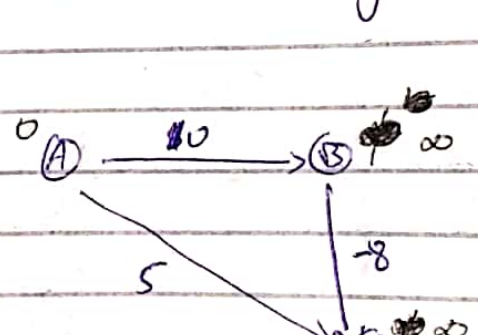
Bellman Ford Algo

Key Point
Relax every edge (V-1) no of times

Example

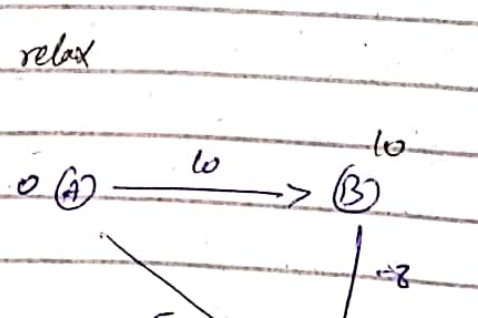


This graph has 3 vertices
so we relax every edge 2 times



∵ ~~B->C~~

1st time relax

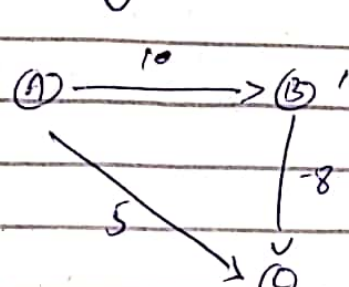


∵ B->C acc to Q's graph
B->C = ∞-8 = ∞ so B->C will be 5 ~~in~~ 5

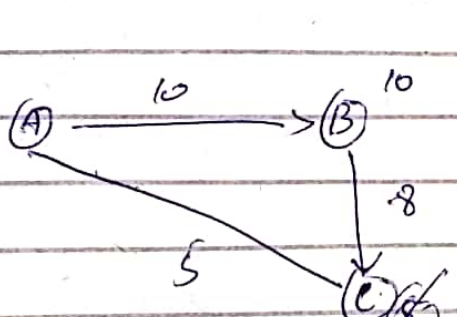
2nd Time Relax

This graph would be relaxed according to 1st time relaxed graph

i.e

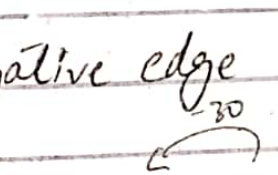


&



For Negative edge cycle

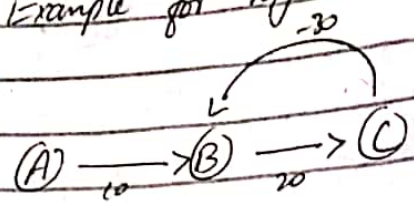
Like



Bellman Ford Algorithm relaxes

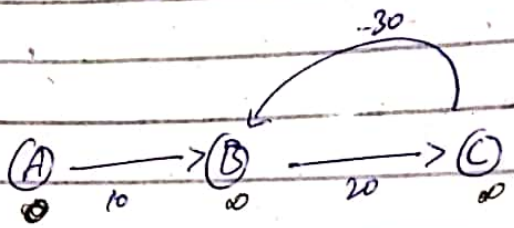
$(V-1)+1$ times

Example for negative edge cycle

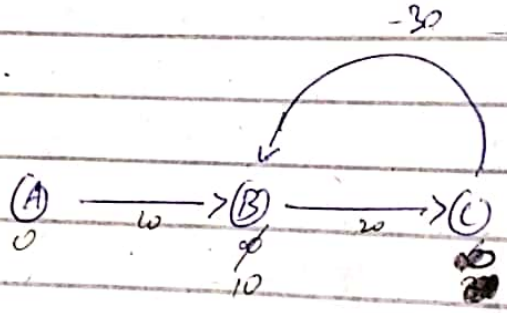


Acc to Bellman ford, we relax graph's edge (v-1) time

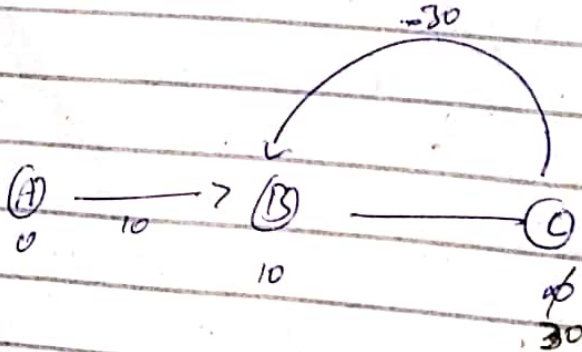
3-1 = 2 times



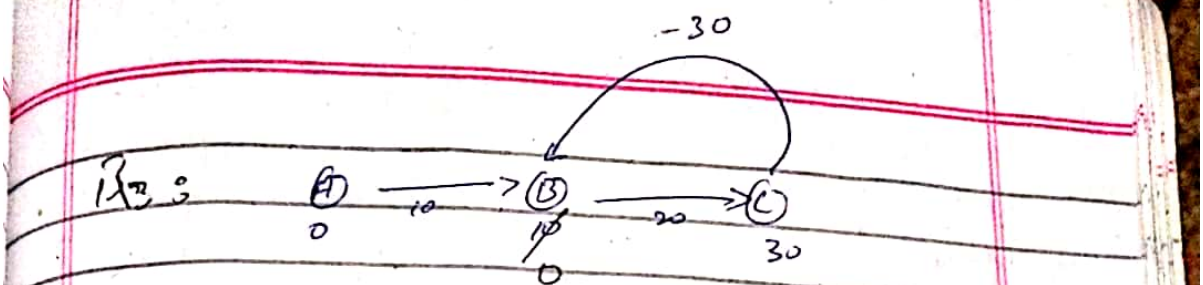
R1:



R2:



∴ Graph is relaxed (v-1) times but to check negative edge cycle, bellman ford algo proposed to check {(v-1)+1} times



=> so due to negative cycle from C to B the cost of B becomes 0 so bellman algo ~~ex~~ stops further relaing and generate error

Note: If we further Relax, then the cost continues to decrease in negative and halts error.

So:

We check upto (v-1) edge and if after checking upto (v-1) +1 time any decrease in cost will prove that there is negative edge cycle

Analysis of Bellman Ford Algo

Bellman Ford (G, V, E, S)

Step 1:
for every vertex $v \in G$ } $O(v)$
    dist [v] = ∞
    dist [source] = 0
∴ perform's for every vertex v

Step 2:

for i = 1 to v-1 } $O(V \cdot E)$
    for each edge $(u, v) \in G$
        Relax (u, v, w)
∴ perform for every edge (E) of every 'v'

Step 3:

for edge $(u, v) \in G$
    if (dist [v] + w(u,v) < dist [v]) $O(1)$
    S.O.P "Graph contain -ve edge cycle"
    else
    retur address

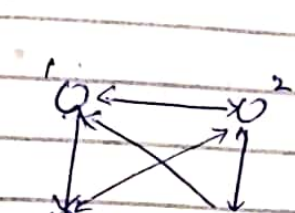
=> So $O(VE)$

Dynamic Programing

Travelling Salesman Problem / Hamiltonian Cycle Problem

Starting and Ending Point should be same.

Ex

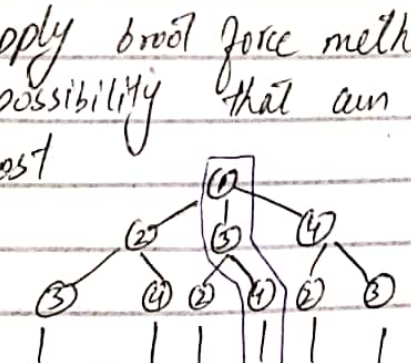


	1	2	3	4
1	0	10	15	20
2	5	0	25	10
3	15	30	0	5
4	15	10	20	0

∴ If we apply greedy approch and select lowest cost of travelling then path would be $① \xrightarrow{10} ② \xrightarrow{10} ④ \xrightarrow{20} ③ \xrightarrow{15} ①$

so the total cost would be = 55

∴ Let's apply brute force method and check for every possibility that can occur and find min cost



Total cost for every possibly: 255 255 265 35 270 75

1 2 3 4 1
1 3 2 4 1
1 4 3 2 1
1 . . . 1
1 . . . 1

so (n-1) combinations

$O((n-1)!) = O(n!)$

∴ If we apply dynamic approach its Time Complexity will remain exponential $O(n^n)$

∴ So ~~optimal~~ solution/path = ① → ③ → ④ → ② → ①

But Time complexity = $O(n-1)!$ or ~~$O(n!)$ which is bigger so we use dynamic approach~~

Graph Traversal

BFS and DFS :

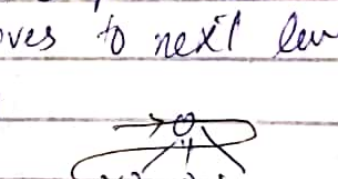
Traversal: The process of visiting and exploring a graph for processing is called graph traversal

- Breadth First Search
- Depth First Search

Breadth First Search :

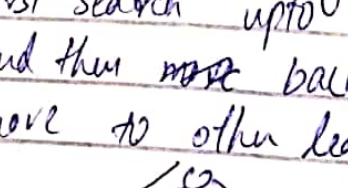
Searches in width of tree. First complete one level and then moves to next level

eg



Depth First Search :

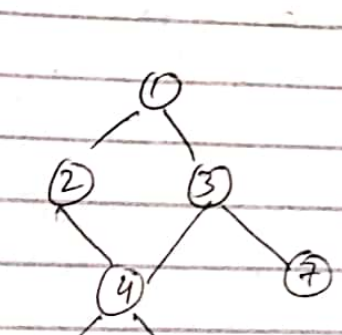
Search in depth of tree First search upto the leaf node and then back tracks and move to other leaf nodes



BFS

↳ use Queue data str.
↳ add node in Queue, visit it, explore it and add its connected nodes in Queue
↳ Now, dequeue the explored node and visit the next node in queue
↳ Repeat process

Example



Queue : 1 2 3 4 7 5 6

Traversal : 1, 2, 3, 4, 7, 5, 6

→

There can be multiple traversal like if we writ 3,2 in Queue instead of 2,3

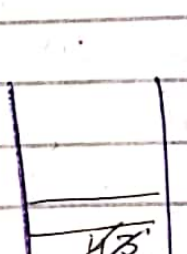
DFS

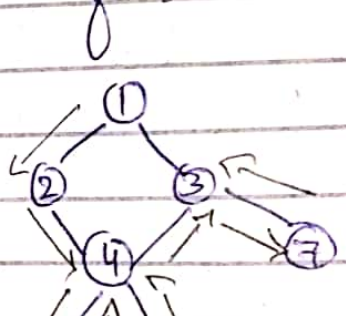
↳ We use stack data structure

↳ Add 1st node in stack and explore its child.

↳ if its child have further childrens then add it in stack and explore its child.

↳ if there are not further child nodes do a back stack and check for other child of node in top index of stack





Traversed : 1, 2, 4, 5, 6, 3, 7

∵ 5 has no child, so we visited it and do a back stack

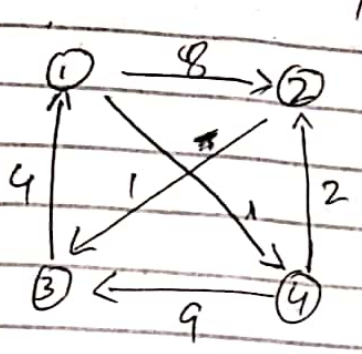
∵ the node is 4 after back stack found using stack

∵ at 6, we again do back stack to 4

First visited all nodes of 4,
Then visited node of 3, when all nodes are visited
3 is poped and went to 2 in stack
2 is poped and went to 1
1 " " .

Floyd Warshall Algo

⇒ All pair shortest path



$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

D^0 = we write distance in matrix if 2 nodes have a direct edge b/w them otherwise the distance will be infinity

$$D' = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & (12) & 0 & (5) \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{array}$$

1-1 : 1-1 + 1-1 = 11
1-2 : 1-1 + 1-2 = 12
2-1 : 2-1 + 1-1 = 2-1
4-1 : 4-1 + 1-1 = 4-1
so we can write row and column same as it is n-1 row and 1 column

D' = we can use (1) node to move to other edges

example 3→2 = 3→1 + 1→2

For : 3→2 = 3→1 + 1→2

$\infty > 4 + 8$

$\infty > 12$

D^2 =

	1	2	3	4
1	0	8	[9]	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	[3]	0

1→3 : 1→2+2→3
∞ > [8+1=9]
so 1→3 = 9

4→3 : 4→2+2→3
4→3 > 3 (2+1)

D^2 : 2nd row and 2nd column will be written as it is nodes can be visited via 2nd node and as well as using via one 1. node.

Example

3→4 : 3→1 + 1→2 + 2→4

~~∞~~ 4 + ~~∞~~ 8 + ∞

~~so 3→4 = ∞~~

OR

3→4 = 3→1 + 1→4

= 4 + 1 = 5

D^3 =

	1	2	3	4
1	0	8	9	1
2	[5]	0	1	[6]
3	4	12	0	5
4	[7]	2	3	0

2→4 : 2→3+3→1+1→4

4→1 : 431 or 423
↓ 13 ↓ 7
so 4→1 = 7

D^3 : We can visit nodes via 3rd node as well as using 1 & 2 node also

Example : 2→1 : 2→3 + 3→1

∞ > 1 + 4

so ∞ > 5
2→1 = 5

D^4

D^4 :

D^4 =

	1	2	3	4
1	0	3	9	1
2	5	0	1	6
3	4	7	0	5
4	7	2	3	0

D^4: We can traverse to all other nodes via 4th node.
also can use 1, 2, 3 node for traversing.

Example: 1->2: 1->2 or 1->4->4->2
(1->2 : 8; 1->4->4->2 : 1 + 1 + 2)

so 1->2 = 3 3

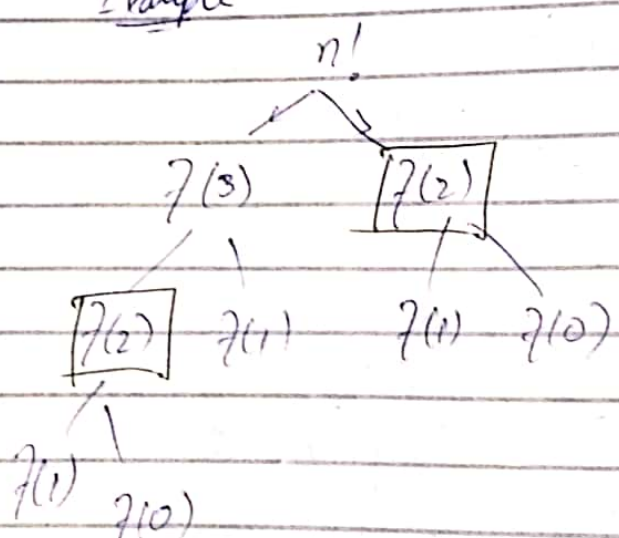
Dynamic Programing:

It divides the problem into series of overlapping subproblem

Two feature: Optimal Substructure

Overlapping Subproblem.

Example



f(2) = overlapping subproblem repeating

f(2)

f(0), f(1) = optimal substr.