AOA

RAYYAN SHABBIR

BITF20M535

SEM #05

ANALYSIS OF ALGORITHM

↳ To efficient design algorithm.

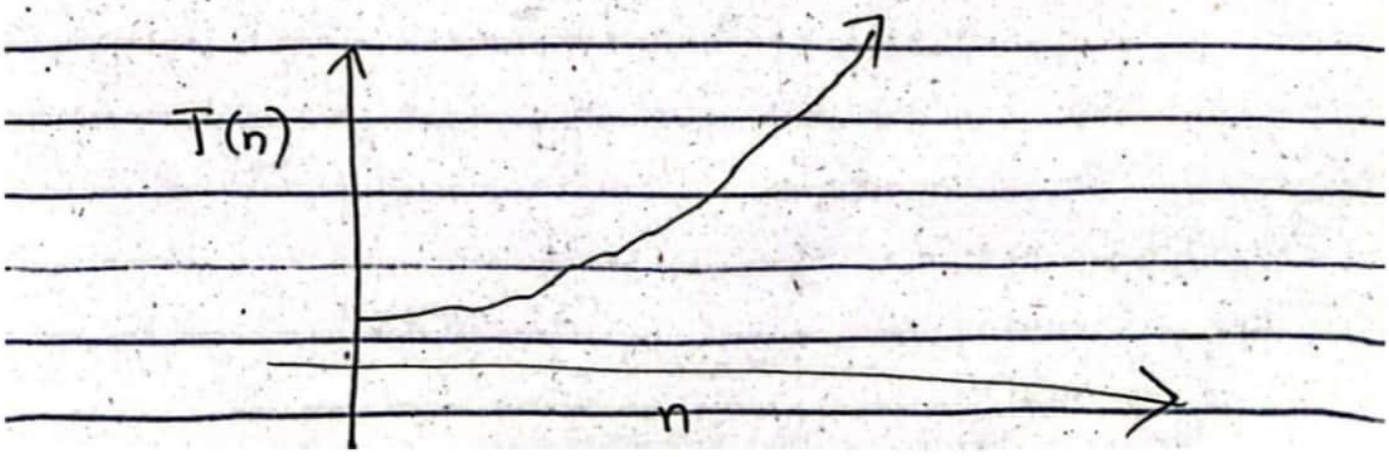
ALGORITHM: sequence of computational steps to solve a problem.

* input is necessary for algorithm.
  ↳ (atleast one)
* output is also necessary for algorithm.

ANALYSIS: Figure out efficiency of algorithms.

- time → } primary
- space
- resources } secondary
- correctness

"Time depends on size of input".
"T(n)"



→ We consider a machine which executes any instruction in constant time.

Example

→ here analysis is independent of language syntax in analysis.

if (i < j)

{

★ conditional

1- if i < j than

2- =

3- =

4- ―

we no. every statement

★ 1D 2D

Array; A[i] A[i, j]

★ Loop For

for i = 1 to k

S1

S2

S3

★ While loop

while i < j

=

★ Function

sum (A)

S1

S2

S3

return j, i, A

Array ↓

Example

In this machine for array length A.length.

★ Index of array starts from '1'.

Example: which takes any input and return length for it.

Maximum (A)

			Cost (no. of steps)	time (frequency)
1-	n = A.length	→ Any length.	C_1	1
2-	max = A[1]		C_2	1
3-	for i=2 to n		C_3	n
4-	if {A[i] > max than		C_4	n-1
5-	→ max = A[i]		C_5	n-1
6-	return max	worst case ←	C_6	1

☆ We mostly interested in worst case analysis.

1	2	3	4	5	6
11	4	15	3	9	7

	Total
1-	c_1
2-	c_2
3-	nc_3
4-	$(n-1)c_4$
5-	$(n-1)c_5$
6-	c_6

$\sum = T(n) =$

$T(n) = c_1 + c_2 + nc_3 + (n-1)c_4 + (n-1)c_5 + c_6$

$= c_1 + c_2 + nc_3 + nc_4 - c_4 + nc_5 - c_5 + c_6$

$= n(c_3 + c_4 + c_5) + (c_1 + c_2 - c_4 - c_5 + c_6)$

all are constants

$= an + b$

$t(n) = an + b$

↳ it means $t(n)$ increases (linearly) w.r.t $an + b$.

↳ It is linear line algorithm.

1 2 3 — X

5 4 3 — ✓

18 > 20

Lecture #02

Analysis of Algorithm

Example: To insert value '4' in a sorted list.

	1	2	3	4	5	6
A :	2	5	8	11	15	18

A[6] = 18

4 < 10 A[7] = 18

* To increase / decrease size of array is not an issue here (bcz we are language independent)

SOL:-

```
Insert( A , 4)
n = A.length
for i=n to 1
    if A[n] > 4
        swap(A[n] , 4)
```

1	2	3	4	5	6	7
2	5	8	10	11	15	18

A[i+1] = 4

SOL:-

Insert(A, 4)

		Cost	Time
1-	i = n = A.length	C_1	1
2-	while A[i] > 4 and i > 0	C_2	n+1
3-	A[i+1] = A[i]	C_3	n
4-	i = i-1	C_4	n
5-	A[i+1] = 4	C_5	1

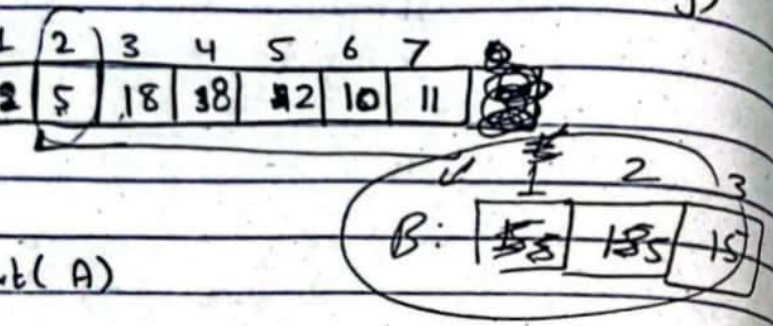
	Total
1-	C_1
2-	$C_2(n+1)$
3-	$C_3 n$
4-	$C_4 n$
5-	C_5

$$T(n) = C_1 + C_2 n + C_2 + C_3 n + C_4 n + C_5$$
$$= C_1 + C_2 + C_5 + C_2 n + C_3 n + C_4 n$$
$$= C_1 + C_2 + C_5 + n(C_2 + C_3 + C_4)$$
$$T(n) = a + bn \rightarrow \text{worst}$$

* Best and Worst average is mostly same as worst case.

E → So, Avrage case is mostly equal to worst.

Example: Arrange arry in increasing order (sort array)



Sol:-

MySort(A)

1- n = A.length
2- B[1] = A[1] ▷ B is an arry of size = 1
3- for i=2 to n — n
4- Insert(B , A[i]) — c_4
5 return B

("▷" is used for commets here.

for To it cost is 3 ... means its cost depends on 'i'.

(1) Insert is function call, so its costs cannot be constant → To insert an element in Insert function it requires 'n' cost. (generally)

Sol:- MySort(A) n = A.

	Cost	Time	Total
1-	c_1	1	c_1
2-	c_2	1	c_2
3-	c_3	n	$c_3 n$
4- depends on i ← c_4 OR	i	n-1	OR $\frac{n(n+1)}{2}-1$ c_2
5-	c_5	1	c_5

* both of them equals to = $1+2+3+4+\cdots+n = \frac{n(n+1)}{2}$

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

$\Sigma = T(n)$

$=$

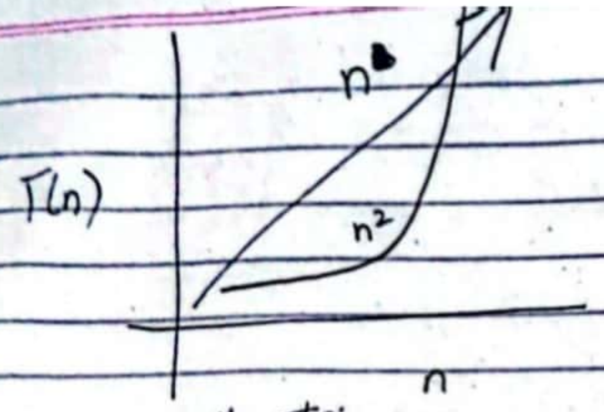
$T(n) = c_1 + c_2 + c_3 n + \frac{c_4 n^2}{2} + \frac{c_4}{2} n - c_4 + c_5$

$= \frac{c_4 n^2}{2} + c_3 n + \frac{c_4}{2} n + c_1 + c_2 - c_4 + c_5$

$= n\left(c_3 + \frac{c_4}{2}\right)$

$T(n) = an^2 + bn + c$ → Wost Case → running time is quadratic

$T(n) = an + b$ → best Case (when arry is already



Growth rate:

~~log < n < n² < n³~~

linear, quad etc are class of function. Like for linear : 10n + 3 or 1n + 5 are some... we only called them linear.

(1) $f(n) = 10n$ → linear function

→ after $n >= 10$ n^2 is bigger than n. but for $n < 10$ linear is best then quadratic.

(2) $f(n) = 2n^2$ → quadratic function is better than linear

* but generally Growth rate,

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

** We can compare algo's on basis of their inputs.

* These growth rates are also called "rate of change."

We can also write these growth rate in a compact way by using NOTATIONS.

For ① $2n+3$ or $an+b$

~~we can simply write~~

big-Oh : $O(n)$

② an^2+bn+c

big-Oh : $O(n^2)$

bcz here dominating term is 'n^2'.

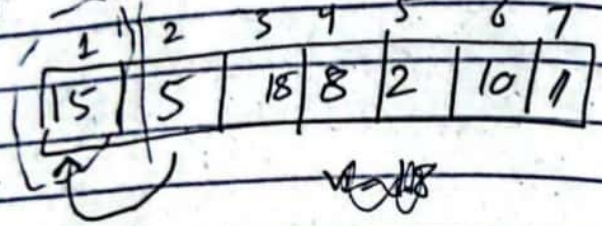
→ In place algo's which solve problem without any additional variables/space

→ "Insert" function is not an Inplace algo ; bcz it requires additional memory.

* We can also make this insert fuction In place.

by doing any into two halves.

Simple Insertion Sort

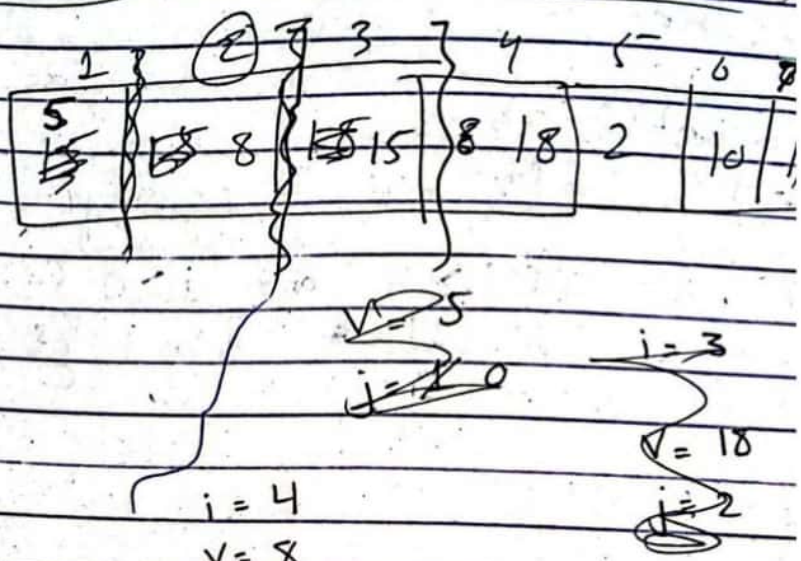


To make insert fuction In place which will became insertion sort.

```
Insertion-Sort (A)
1-  n = A.leyth
2-  for i=2 to n
3-      v = A[i]
4=      j = i-1
5-      while A[j] > v and j > 0
            A[j+1] = A[j]
7-          j = j-1
5-      A[j+1] = v
```

Insertion sort → worst case → $O(n^2)$

best case → $O(n)$



~~Insert~~ 4 = 28

①

Insert(A, 4)

1- n = A.length

Insert(A, 4)

		Cost	Time
1-	i = n = A.length	C_1	1
2-	while A[i] > 4 and i > 0	C_2	n+1
3-	A[i+1] = A[i]	C_3	n
4-	i = i-1	C_4	n
5-	A[i+1] = 4	C_5	1

$= C_1 + C_2 n + C_2 + C_3 n + C_4 n + C_5$

$= C_1 + C_2 + C_4 + C_5 + n(C_2 + C_3 + C_4)$

$= a + bn \rightarrow$ worst

$= c \rightarrow$ best

A: | 7 | 1 | 9 | 5 | 0 |

1

B: | 7 |

②

MySort(A)

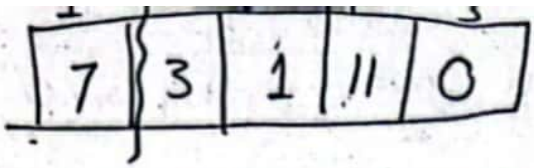
		cost	time	total
1-	n = A.length	C_1	1	C_1
2-	B[1] = A[1]	C_2	1	C_2
3-	for i = 2 to n	C_3	n	$C_3 n$
4-	Insert(B, A[i]) C_4 = i	$\frac{n(n+1)}{2} - 1 = n - 1$		$C_4 \frac{n(n+1)}{2}$
5	return B	C_5	1	C_5

$T_a(n) = C_1 + C_2 + C_5 + C_3 n + C_4 \left(\frac{n^2}{2} + \frac{n}{2} - 1\right)$

$= C_1 + C_2 + C_5 - C_4 + C_3 n + \frac{C_4 n^2}{2} + \frac{C_4 n}{2}$

$= an^2 + bn + c \rightarrow$ worst

$= an + b \rightarrow$ best



③ Insertion - Sort (A)

```
1-   n = A.length

2-   for i = 2 to n
3-       v = A[i]
4-       j = i-1

5-       while A[j] > v and j >= 0

6-           A[j+1] = A[j]
7-           j = j-1

8-       A[j+1] = v
```

Lecture#03

AofA

Incremental way to solve problem → minimum find one by one.

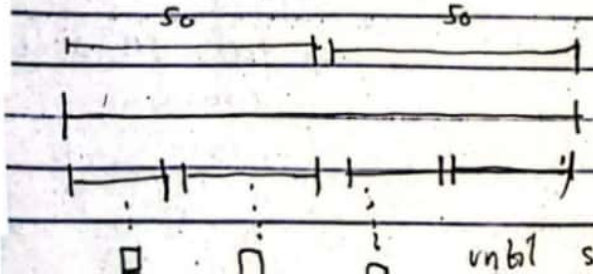
Against it the algo's are called "Iterative".

↳ Elegant way to solve problem → Divide and Conquer Rule

Recursion base. → ↓

To solve bigger problem into small problems and then solve them.

Finding Smallest
↳ ~~Sorting~~ using "Divide and Conquer."



Problem: Finding Smallest Element from array using divide and Conquer.

↳ for this first we have to decide where we stop. ← (base condition) condition termination.

Smallest (A, S, e)

1- if s = e then → this step depend on our self we can also stop there where we have S elements.
2- return A[S]
3- if s < e then
4- m = ⌊(s+e)/2⌋ → Divide step
↳ we take floor function bcz maybe ans will not be integer so, we convert it to integer.
5- SL = Smallest(A, s, m)
6- SR = Smallest(A, m+1, e) → this also depend on ourselfs.
7- if SL < SR then
8- return SL
9- else return SR → Conquer

So, its cost depends on

cost of first So elems + cost of last So elems + constant 'if' times

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

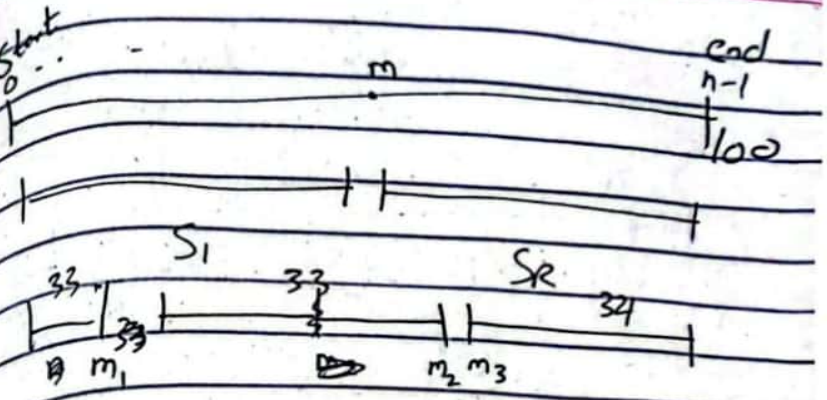
$$\boxed{T(n) = 2T(n/2) + C}$$

↳ As algo is recursive So, its running time will also be recursive.

↳ So, to calculate time of Divide and Conquer technique we have to develop/calculate vector. time and then add them.

↳ The subproblem development depends on our logic.

↳ We can make even (2, 3, 4) division as well as uneven distribution



★ Divide and Conquer has also 3rd step → "merge step"

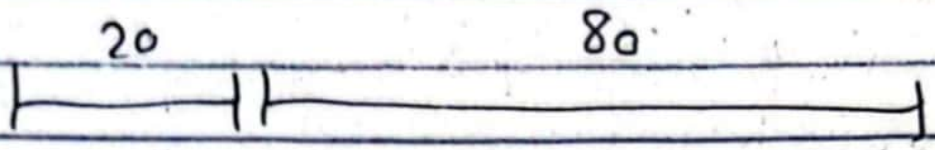
↳ which will merge the individuals elements.

So D & C has 3 steps

1- Divide
2- Recursively solve
3- Merge

→ time of sol this kind of problems, depends on only these steps.

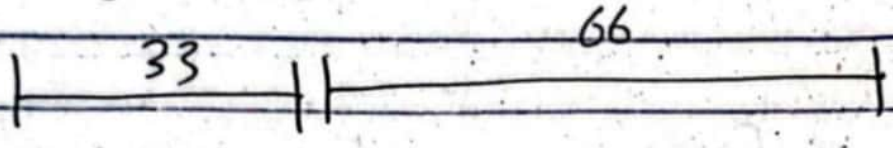
↳ We can also develop



its rector will be : $T(n) = T\left(\frac{2n}{10}\right) + T\left(\frac{8n}{10}\right)$

OR

↳ We can also develop

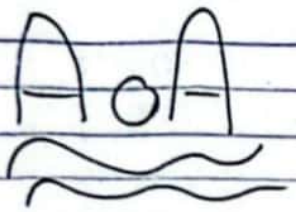


So, its rector will be : $T(n) = \left(\frac{n}{3}\right) + \left(\frac{2n}{3}\right)$

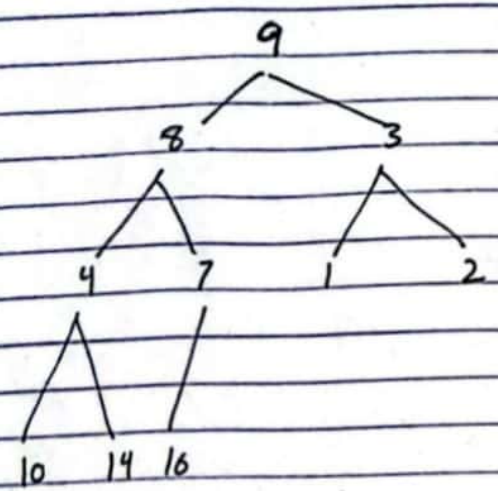
↳ So, we can do any kind of prob by Divide and Conquer rule

↳ But first we need to understand which problem to do or which to not?

Lecture # last (pre mid)



Sorby Heap Tree



* We run loop till '2' bcz root will be automatically sorted.

prob #01

Algo Heap Sat

HeapSort (A)

```
1-  Build_Max_Heap(A)                    → O(n)
2-  n = A.length
3-  for i=n down to 2                     → n
4-      exchange (A[1] , A[A.heap_size]
5-      A.heap_size = A.heap_size - 1     } → lgn
6-      Max_Heapify (A, 1)                }
```

(n lgn)

Time Complexity = O(n lgn)

prob#02 Most frequent occuring element in Array

solution#01

range 0-k

A:

2	5	3	0	2	3	0	3

C:

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

C:

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

Counting Sort

↳ time to initialise it $O(k)$

Overall: $O(n+k)$

here 'n' is for the 'for loop'.

↳ This is the problem here.

↳ if element not exist for index then that index will be wastage.

↳ Real numbers not deal here.

* This sol is only not for some scenarios.

* This 'C' gives us a lot of info like

↳ sorted the array

↳ how many student have 5 marks? → C[5] ↳ constant time

↳ how many student have marks less than 10?

↳ add 0-9

کتنے students مارکس 0 سے 5 تک -ہیں do it in less time.

$C[i] = C[i] + C[i-1]$

C:

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

→ now by this we can give range like question.

→ It will also give position of any element

position of '5' in sorted array: 8

C: | 2 | 2 | 4 | 7 | 7 | 8 |
index: 1 2 3 4 5 6 7 8

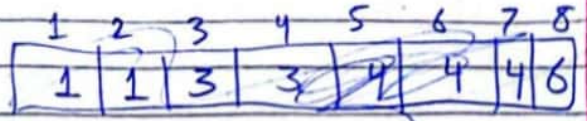
B: | 0 | 0 | 2 | 2 | 3 | 3 | 3 | 5 |
index: 1 2 3 4 5 6 7 8

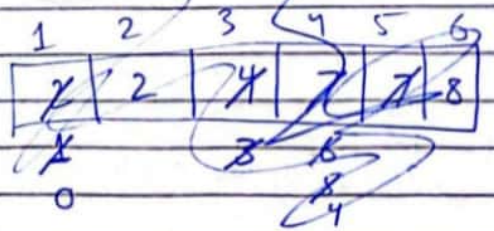
→ COUNTING SORT

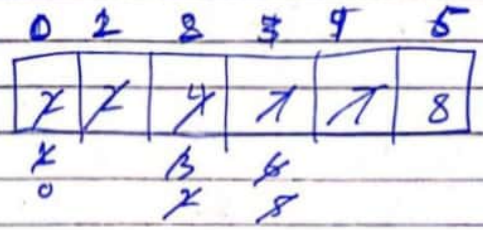
* check element in C and place that element in B array at its index value
  ↳ Then decrement the index val by '1'.

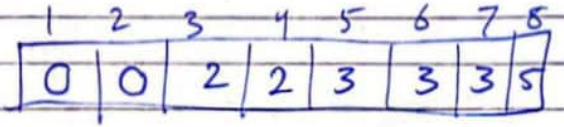
B[C[A[i]]]

* Counting sort is unstable sorting algorithm bcz first '3' comes at position → which is last index for that position i.e 3.
  ↳ To make stable counting sort start traversing array from









B[C[

1. → Find largest elem from any
2. → Insert elem in sorted arry.
3. → Sort the array
4. → Insertion Sort
5. → Smallest (Divide and conquer) *
6. → Factorial
7. → Fibnaccoi serics
8. → Peak value
9. ~~→ [illegible]~~
10. → Binay Search
11. → Merge (while + for)
12. → MergeSort
13. → asymptotic n
14. → methods
15. → partition
16. → Quick_Sort

Sort large number of array?

17. → Max Heapify
18. → Insertion in Heap
19. → Build_Max Heap
20. → Priority Queue
21. → Heap Sort
22. → Counting Sort.

lecture# 09

AOA

SUBSTITUTION METHOD

↳ use to solve recursion.

↳ guess based method

steps:

↳ guess the solution.

↳ then verify the guess

↳ Check

Solve: ~~Recursion~~ Recurrence

Example → ~~Guess the Solution.~~

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

∴ running time atleast cn.

guess:

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

$$T(n) \leq cn^2$$

Verification:

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

(left: $T(n)$; right: $2T(n/2)$ → substitute)

put value of $T(n) = cn^2$ in $T(n) = 2T(n/2) + cn$

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

$= \frac{1}{2} cn^2 + cn$

Check:

$\frac{1}{2} cn^2 + cn \leq (cn^2)$?

↳ equal to $\frac{1}{2} cn^2 + \frac{1}{2} cn^2$

Thus, it means our guess is correct.

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

*Maybe this guess is not tight upper bound.

So, to check tight upper bound we check guess with another guess

Guess:

$T(n) = O(n)$

$T(n) \leq cn$

Verification

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

$= 2c\frac{n}{2} + cn$

$= 2cn$

Check:

$2cn \not\leq cn$?

False,

Therefore, our guess is wrong.

So, we check by seeing mid guess b/w

$O(n)$, $O(\lg n)$, $O(n^2)$

× ✓

Guess:

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

$\Rightarrow T(n) \leq cn \lg n$

Verification,

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

↳ substitute

$= 2c\frac{n}{2}\log\frac{n}{2} + cn$

↳ bc2 ↳ *2 is divided by n not by $T(n)$ like this: $\frac{T(n)}{2}$

$= cn \lg \frac{n}{2} + cn$

$= cn[\lg n - \lg 2] + cn$ $\quad \because (\lg_2 2 = 1)$

$= cn \lg n - cn + cn$

$= cn \lg n$

Check:

$cn \lg n \leq cn \lg n$ (guess)

⇒ Thus, $T(n) = O(n \lg(n))$ is exact equal. So, we can say that it is exact upper bound.

Means

$T(n) = O(n \lg(n))$ is tight upper bound of $T(n)$.

* We stop our iteration at doing this, until we get the tight bound.

Master Method

We can apply master method, only when our recurrence is in form of ↓

$T(n) = aT(n/b) + f(n)$ } can apply master method.

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

* We cannot apply master method

on:

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

cannot apply master method.

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

here size of sub problem is different so we cannot apply master method.

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

↳ leaves: $n^{\log_2 4} = n^2$

$T(n) = aT(n/b) + f(n)$

↳ leaves: $n^{\log_b a}$

a would be $a \geq 1$ → because subproblems/branches/child can be 1 or >1 not equal to '0'.

↳ $b > 1$ } → because if $b=1$ problem cannot be divided.

* By using leaves we can identify cost.

* In master method for general recursion we have three methods.

Case #01 E → epsilon → which is 0.1 or more.

$E > 0$

- if $f(n) = O\left(n^{\log_b a - E}\right)$, then $T(n) = \Theta(n^{\log_b a})$

$f(n) \leq O(n^{\log_b a})$, then solution ↑

if Agar

(case #01)

Example

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

Solution: $a = 4$

$b = 2$

$f(n) = n$

$n^{\log_b a} = n^{\log_2 4}$

$= n^2$

Since,

$f(n) = O(n^2)$ is true,

therefore, case#01 of master method is True →

↳ which means its solution of our the recurrence is:

$T(n) = \Theta(n^2)$

* if it does not return "True":

next page

Exple #02:

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

Sol: $a = 2, \quad b = 2, \quad f(n) = n$

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$

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

↳ but here Case#02 satisfied. not but Case#02 satisfied.

Second case of MM is True.

$\Rightarrow T(n) = \Theta(n \lg n)$

* Agr bara hojaye to case #01

* Agr equal hojaye to case#02

* Agr less hojaye to case#03

Case#02:

2- if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$

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

case #03

E value can be any.

3- if $f(n) = \Omega(n^{\log_b a + E})$ and

$af(n/b) \leq cf(n)$ $\therefore c < 1$ then $E > 0$

condition 1 condition 2

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

Expl #03

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

Soln:

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

$n^{\log_b a} = n^{\log_8 2}$

$= n^{1/3} = \sqrt[3]{n}$

$f(n) \geq \Omega$

↳ condition 1 satisfied.

Now check condition #02

$a f(n/b) = 2 \times n/8$

$= \frac{1}{4} f(n) \qquad c = \frac{1}{4} < 1$

$T(n) = \Theta(n)$ since all conditions of case#03 satisfied. So, it is true.

* $f(n)$ is any term other than leading terms.

$\therefore T(n) = 4T(n/2) + \underbrace{O(n)}_{\hookrightarrow f(n)}$

Expl

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

Soln:

$a = 4$

$b = 3$

$f(n) = n^2$

$n^{\log_b a} = n^{\log_3 4}$

$= n^{1.3} = n^{4/3} = \sqrt[3]{n^4}$

$f(n) > \sqrt[3]{n^4}$

↳ case#03 → condition 1 satisfied.

$a f\left(\frac{n}{b}\right) \quad 4 f\left(\frac{n}{3}\right) = 4 \times \left(\frac{n}{3}\right)^2 = \frac{4n}{9}$

$= \frac{4}{9}(f(n))$

$c < 1$

↳ condition 2 satisfied

So, it is "TRUE"

Q#01

$$T(n) = 5T\left(\frac{n}{5}\right) + cn$$

Sol:-

$a = 5$

$b = 5$

$f(n) = cn$

$$n^{\log_b a} = n^{\log_5 5}$$

$$= n$$

Since,

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

↳ Here case#01 not satisfied.
Case#02 satisfied.

"Case#02 of mm is True".

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

Q#02

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

guess:

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

$T(n) = cn^2$

with:

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

$$= \frac{2c\,n^2}{(n-1)^2} + c \leq cn^2$$

Guess:

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

$= c \lg n$

Guess: Verification:

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

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

$$= 2c\left[\lg n - \frac{\lg n}{\lg 1}\right] + c$$

$$2c \lg \frac{n}{n-1} + c \leq c \lg n$$

Q3)

fun (A, s, e)
if s > e then return failure
n = A.length
m = (s+e)/2

if (m-1) > 1 && A[m] == A[m-1]) then } C
return m-1
else if (m+1) < n A[m] == A[m+1] then
return m+1

return fun (A, s, m) } n/2

return fun (A, m+1, e) } n/2

1 2 3

func (s, e, A)

if s ≥ e then
return FAILURE

n = A.length
m = (s+e)/2

C { if ((m-1) > 1 && A[m] == A[m-1]) then
return m-1

else if ((m+1) < n A[m] == A[m+1]) then
return m+1

return fun (s, m, A) } T n/2

return fun (m+1, e, A) } T n/2

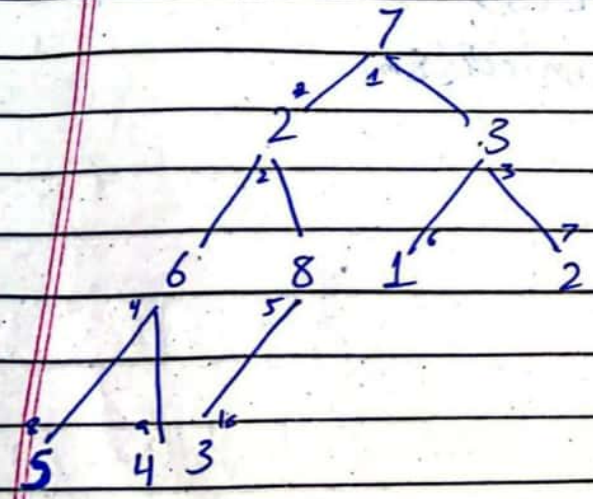
$T(n) = 2T\left(\frac{n}{2}\right) + C$

Lecture # 11

AOA

Heap Data Structure

↳ complete bny Tree.
↳ Heap is an Arry Data Structure.
↳ Evry level is complete and except last one last is complete from left side
↳ ~~Arrange~~ Arrange level by left to right.



1	2	3	4	5	6	7	8	9	10
7	2	3	6	8	1	12	5	4	3

To find parent of a node

Formula,

right child : $\frac{n}{2}$

left child : $\lfloor \frac{n}{2} \rfloor$

→ Parent (i)
1- return $\lfloor \frac{i}{2} \rfloor$

~~Left~~

To Find child of a node

For left child : left(i)
1- return 2xi

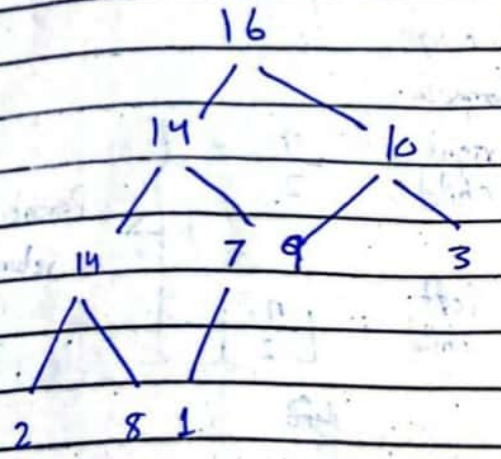
For right child: Right(i)
1- return 2xi+1

Heap types

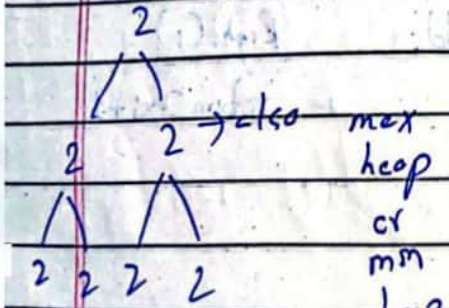
① Max Heap

↳ Every parent greater than or equal child.

↳ root will be the largest value.



$A[parent(i)] \geq A[i]$

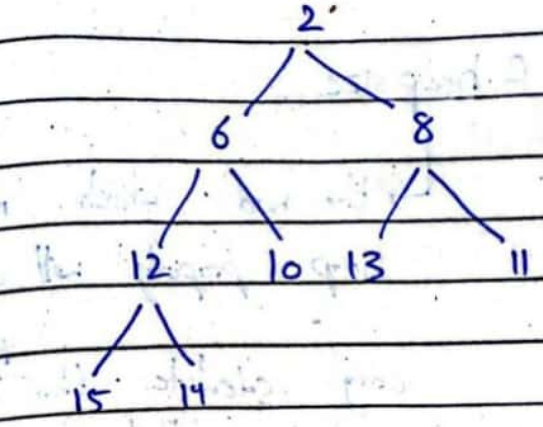


→ also max heap or min heap

② Min Heap:

↳ Every child greater than or equal its parent.

↳ root will be the smallest.



$A[Parent(i)] \leq A[i]$

* Heap Available Attributes:

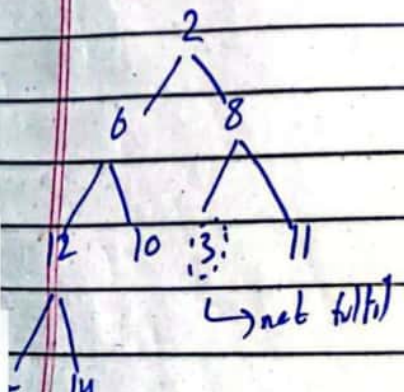
(1) A.length

(2) A.heapsize

↳ the node which notfulfil heap property will discarded previous.

only calculate, the ↑ nodes which fullfil heaps property

Exple:

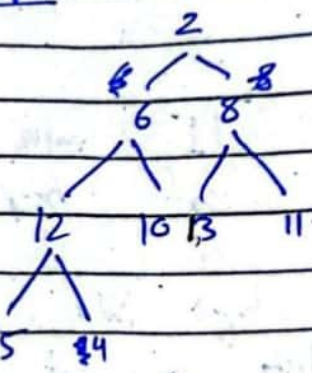


↳ not fulfil

A.length = 9

A.heapsize = 5

Expl



A.length = 9

A.heapsize = 8

Scenario:

change '1' value of a node of heap tree.

↳ The chage value may satisfy or not max heap property.

↳ To solve this we use function which solve / remove this problem.

↳ Max Heapify.

↳ check node with its children and swap with layest child

↳ Do this recursively until all child are less from it or no child exists

Function:

```
Max_Heapify ( A, i )
1-  l = left(i)
2-  r = Right(i)
3-  if A[l] > A[i] and l ≤ A.heap-size
4-      largest = l
5-  else largest = i
6-  if A[r] > A[largest] and r ≤ A.heap-size
7-      largest = r
8-  if largest ≠ i then
9-      exchange (A[i], A[largest])
10-     Max_Heapify (A, largest)
```

height of binay Tree = $\lg_2 n$

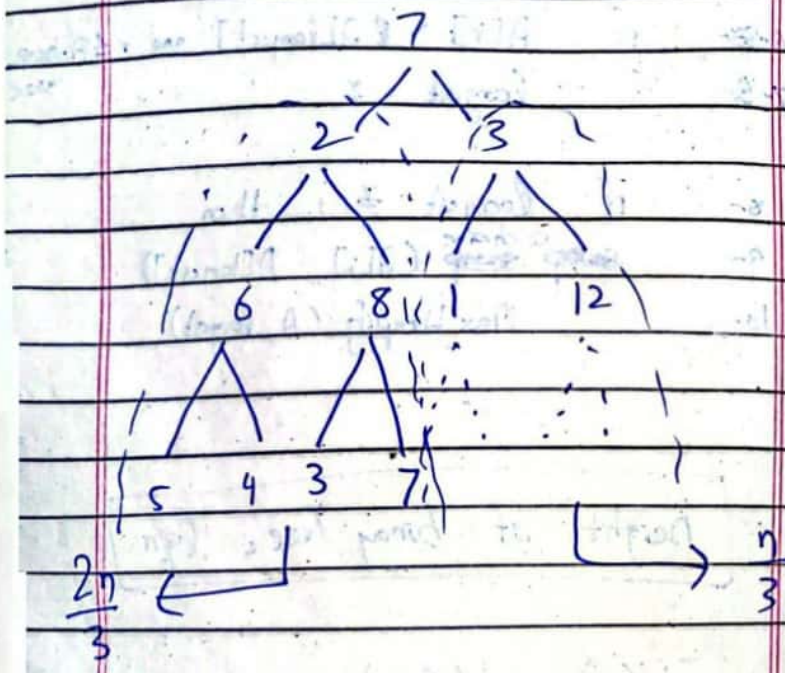
$T(n) = O(\lg n)$ ← Worst Case

If sub-prob become left side of tree.
For ~~right~~ left side tree.

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

↳ recurrence of Max-Heapify

↳ which equals to $O(\lg n)$



→ If sub-prob become right side of tree

→ For right side tree

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

Note:

Max heap

* root will be on the largest value but

* Min heap → smallest will be on top

↳ but largest will be somewhere in leaf.

↳ not exactly we know where on leaf it is.