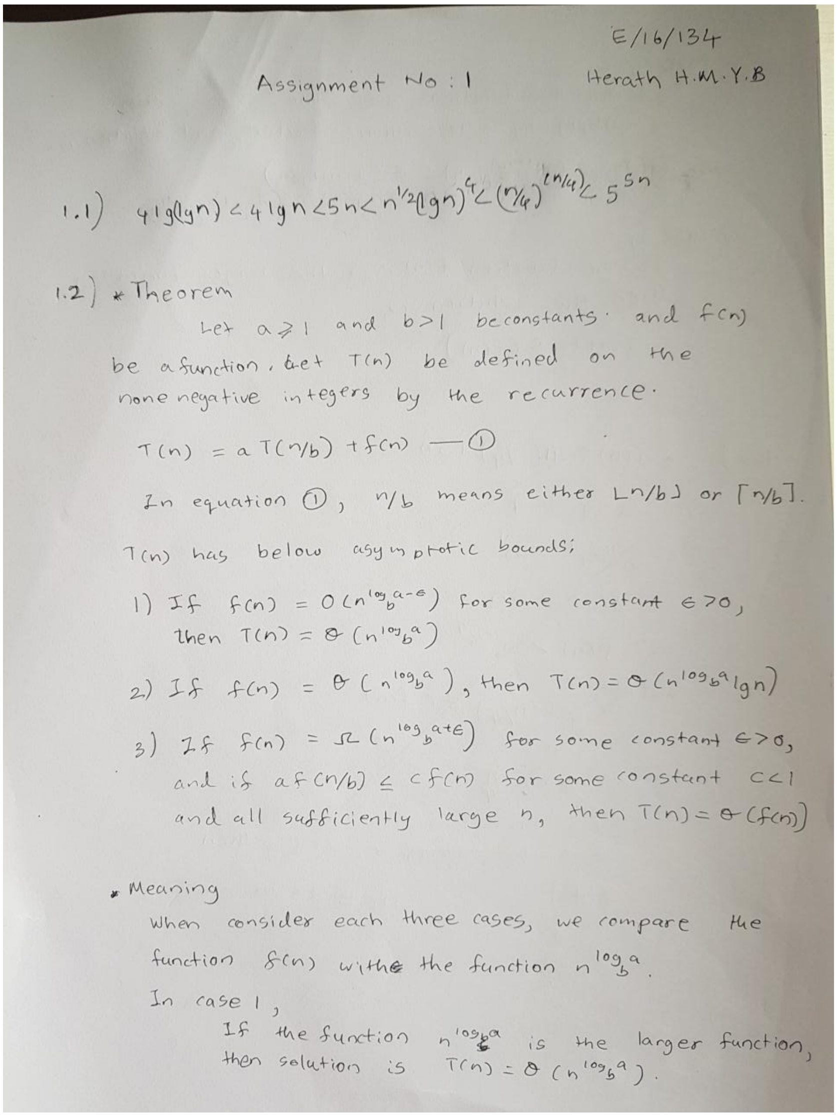


E/16/134

Assignment No : 1 Herath H.M.Y.B

1.1) $4\lg(\lg n) < 4\lg n < 5n < n^{1/2}(\lg n)^4 < (n/4)^{(n/4)} < 5^{5n}$

1.2) * Theorem

Let $a \geqslant 1$ and $b > 1$ be constants. and $f(n)$ be a function, Let $T(n)$ be defined on the none negative integers by the recurrence.

$T(n) = aT(n/b) + f(n)$ —①

In equation ①, n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

$T(n)$ has below asymptotic bounds;

1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$,
then $T(n) = \Theta(n^{\log_b a})$

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

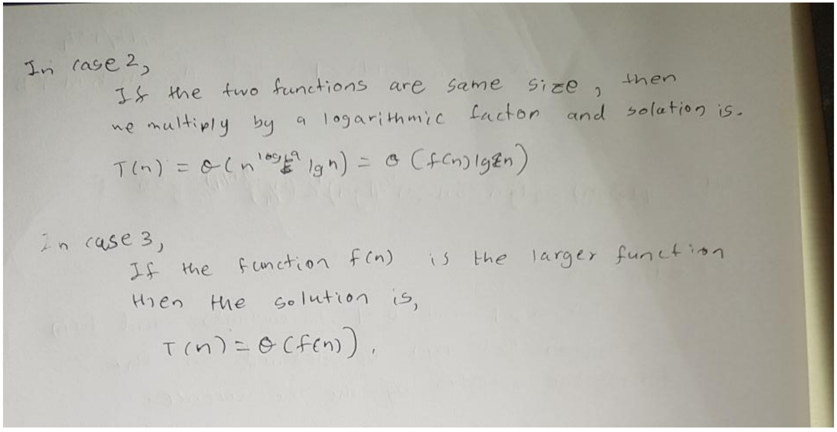
3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$,
and if $af(n/b) \leq cf(n)$ for some constant $c < 1$
and all sufficiently large n, then $T(n) = \Theta(f(n))$

* Meaning

When consider each three cases, we compare the function $f(n)$ with ~~the~~ the function $n^{\log_b a}$.

In case 1,

If the function $n^{\log_b a}$ is the larger function, then solution is $T(n) = \Theta(n^{\log_b a})$.



In case 2,

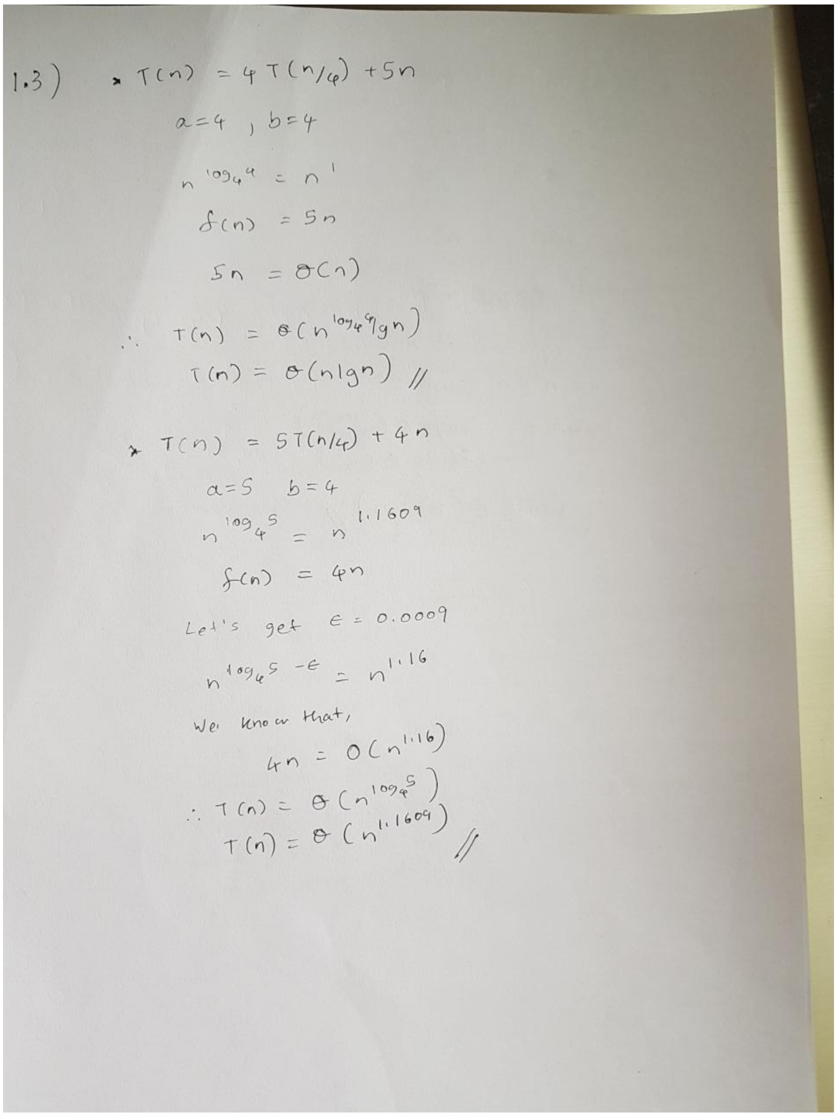
If the two functions are same size, then we multiply by a logarithmic factor and solation is.

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

In case 3,

If the function $f(n)$ is the larger function then the solution is,

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



1.3) * $T(n) = 4T(n/4) + 5n$

$a=4$, $b=4$

$n^{\log_4 4} = n^1$

$f(n) = 5n$

$5n = \Theta(n)$

$\therefore T(n) = \Theta(n^{\log_4 4} \lg n)$

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

* $T(n) = 5T(n/4) + 4n$

$a=5$ $b=4$

$n^{\log_4 5} = n^{1.1609}$

$f(n) = 4n$

Let's get $\epsilon = 0.0009$

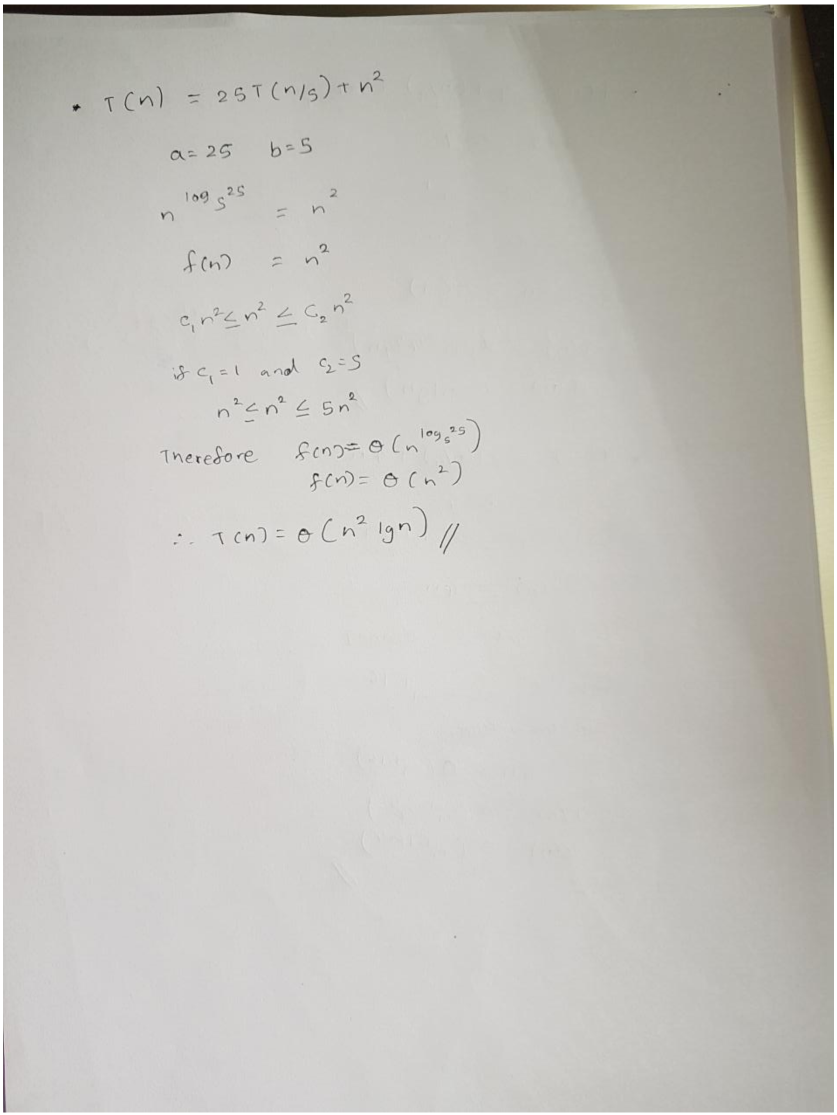
$n^{\log_4 5 - \epsilon} = n^{1.16}$

We know that,

$4n = O(n^{1.16})$

$\therefore T(n) = \Theta(n^{\log_4 5})$

$T(n) = \Theta(n^{1.1609})$ //



* $T(n) = 25T(n/5) + n^2$

$a = 25 \quad b = 5$

$n^{\log_5 25} = n^2$

$f(n) = n^2$

$c_1 n^2 \le n^2 \le c_2 n^2$

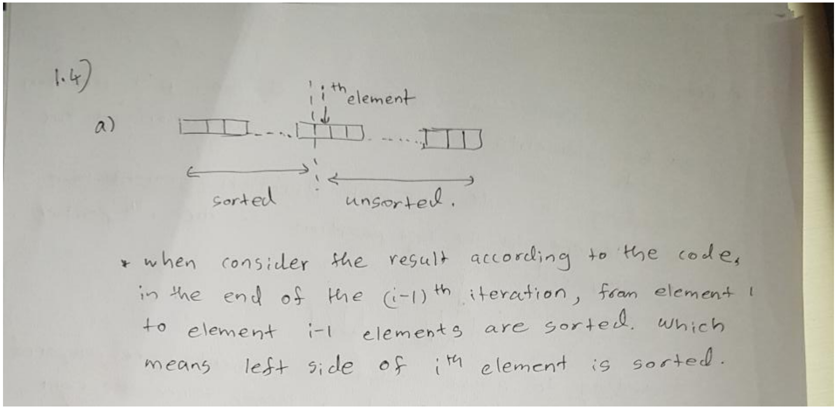
if $c_1 = 1$ and $c_2 = 5$

$n^2 \le n^2 \le 5n^2$

Therefore $f(n) = \Theta(n^{\log_5 25})$

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

$\therefore T(n) = \Theta(n^2 \lg n)$ //



1.4)

a)

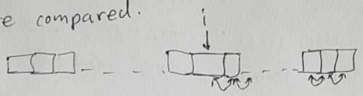
i^{th} element
sorted
unsorted.

* when consider the result according to the code, in the end of the $(i-1)^{th}$ iteration, from element 1 to element $i-1$ elements are sorted. which means left side of i^{th} element is sorted.

(*) Progress.

- So we know that in the end of the $(i-1)^{th}$ iteration all the elements from position 1 to position $i-1$ are sorted.

According to code, in each iteration ~~elements posit~~ from element in the last position to the i^{th} position are compared.



According to above adjacent elements are compared. If the left side element is larger than the right side elements, those two elements are swaped. So in this final comparison is done to i^{th} element and $(i+1)^{th}$ element. Therefore in the end of the each iteration, i^{th} element is sorted. ~~Therefore~~ So in the end of $(i-1)^{th}$ iteration all the element from position 1 to $(i-1)$ position are sorted, in ascending order.

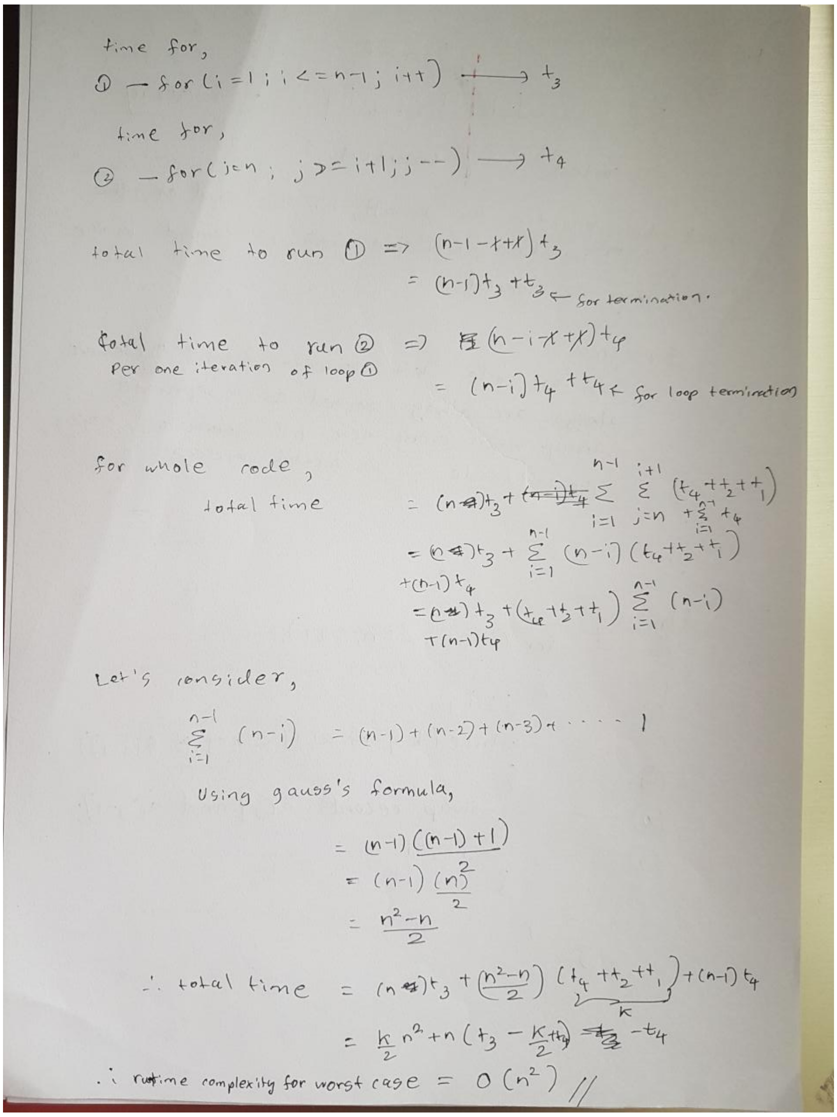
b) * In the worst-case, the given array is in reverse sorted order. Which means array is sorted in decending order. In this we have to do always swaps in each iteration of the inner loop.

amount of time to do a single comparison $= t_1$

" " time to do a swap $= t_2$

for outer loop $= \sum_{i=1}^{n-1}$

for inner loop $= \sum_{j=n}^{i+1}$



time for,

① − for(i=1; i<=n-1; i++) $\longrightarrow t_3$

time for,

② − for(j=n; j>=i+1; j--) $\longrightarrow t_4$

total time to run ① $\Rightarrow (n-1-\cancel{1}+\cancel{1})t_3$

$= (n-1)t_3 + t_3 \leftarrow$ for termination.

total time to run ② per one iteration of loop ① $\Rightarrow \cancel{\Sigma}(n-i-\cancel{1}+\cancel{1})t_4$

$= (n-i)t_4 + t_4 \leftarrow$ for loop termination

for whole code,

total time $= (n\cancel{-1})t_3 + \cancel{(n-i)t_4} \sum_{i=1}^{n-1} \sum_{j=n}^{i+1} (t_4 + t_2 + t_1) + \sum_{i=1}^{n-1} t_4$

$= (n\cancel{-1})t_3 + \sum_{i=1}^{n-1} (n-i)(t_4 + t_2 + t_1) + (n-1)t_4$

$= (n\cancel{-1})t_3 + (t_4 + t_2 + t_1) \sum_{i=1}^{n-1} (n-i) + (n-1)t_4$

Let's consider,

$\sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + (n-3) + \cdots 1$

Using gauss's formula,

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

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

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

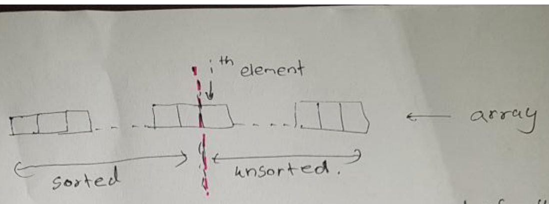
$\therefore$ total time $= (n\cancel{-1})t_3 + \left(\frac{n^2-n}{2}\right)\underbrace{(t_4 + t_2 + t_1)}_{k} + (n-1)t_4$

$= \frac{k}{2}n^2 + n\left(t_3 - \frac{k}{2} + t_4\right) \cancel{t_3} - t_4$

$\therefore$ runtime complexity for worst case $= O(n^2)$ //

1.5)

c)



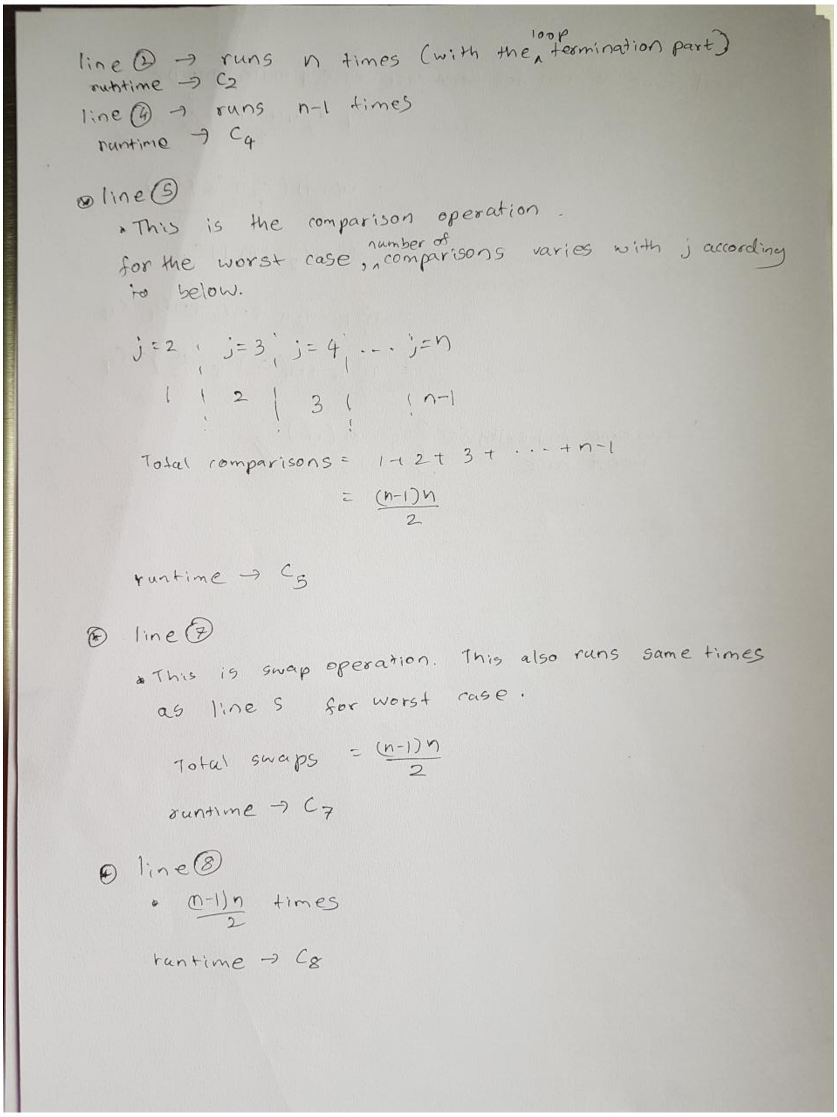
- when consider the result, in the end of the $(i-1)^{th}$ iteration, left side from i^{th} element in the array is sorted.

⊕ Progress

In this code, left elements from i^{th} element, ~~always~~ are always sorted as a separate array. When every new i^{th} iteration begins, the left side is in sorted order.

d)

```
{
  for (i=2; i<=n; i++)
  {
    j=i;
    while (key of a[j] < key of a[j-1])
    {
      swap records a[j] and a[j-1];
      j=j-1
    }
  }
}
```



line ② → runs n times (with the loop termination part)
runtime → C_2
line ④ → runs $n-1$ times
runtime → C_4

⊛ line ⑤

- This is the comparison operation.

for the worst case, number of comparisons varies with j according to below.

j=2	j=3	j=4	...	j=n
1	2	3		n-1

Total comparisons $= 1 + 2 + 3 + \cdots + n-1$

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

runtime → C_5

⊛ line ⑦

- This is swap operation. This also runs same times as line 5 for worst case.

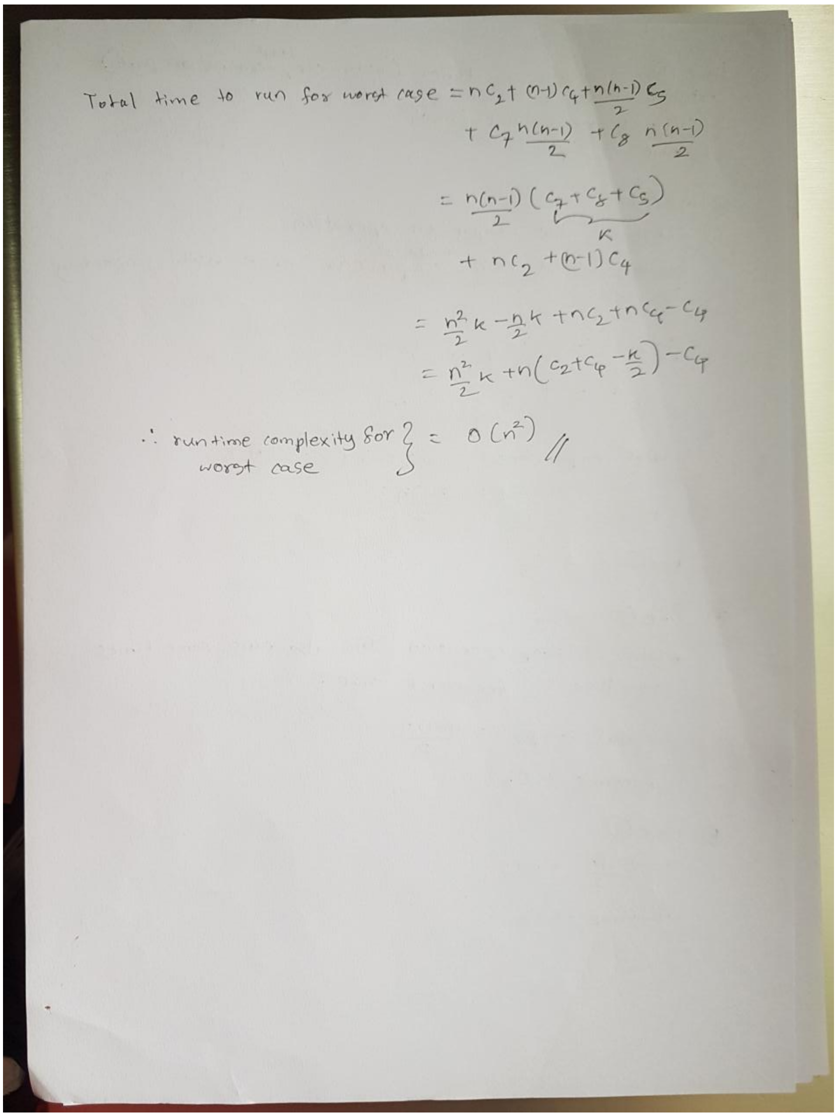
Total swaps $= \frac{(n-1)n}{2}$

runtime → C_7

⊛ line ⑧

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

runtime → C_8



Total time to run for worst case $= nC_2 + (n-1)C_4 + \frac{n(n-1)}{2}C_5$

$$+ C_7\frac{n(n-1)}{2} + C_8\frac{n(n-1)}{2}$$

$$= \frac{n(n-1)}{2}(\underbrace{C_7 + C_8 + C_5}_{K})$$

$$+ nC_2 + (n-1)C_4$$

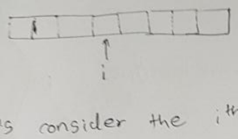
$$= \frac{n^2}{2}K - \frac{n}{2}K + nC_2 + nC_4 - C_4$$

$$= \frac{n^2}{2}K + n\left(C_2 + C_4 - \frac{K}{2}\right) - C_4$$

$\therefore$ runtime complexity for worst case $\Big\}$ $= O(n^2)$ //

1.6)

a) * This can be implementedusing an array.



let's consider the i^{th} element. The ~~chi two~~

~~parent~~ element = i

~~two~~ left child position = 2i

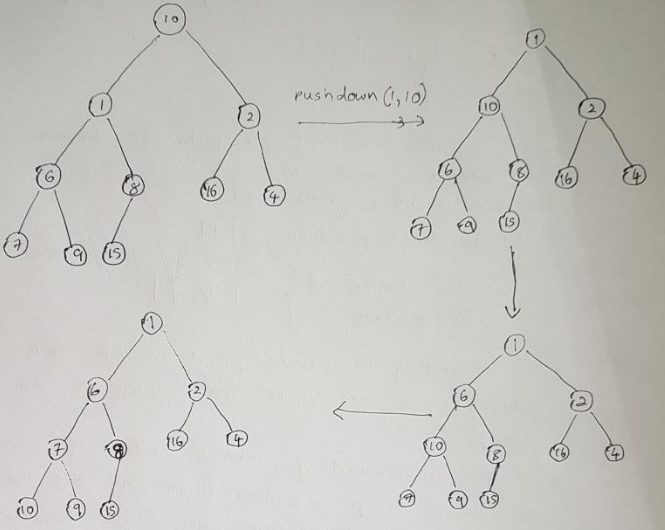
right child position = 2i + 1

parent element of i^{th} element $\}$ = $\lfloor \frac{i}{2} \rfloor$

* So according to above positions of the array, we can store the elements as a heap.

* when implementing, above code is using as mini heap. So when arranging the heap parent element always should be smaller than child elements.

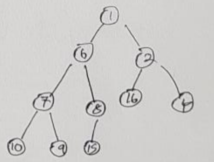
b) * In this a[1] is not in correct position, because child nodes are less than a[1].



* By using pushdown (1,10), we can place 10 in the correct position without vialating the properties.

* Now we have correct min heap. Do do the sorting deteting the minimum should be done.

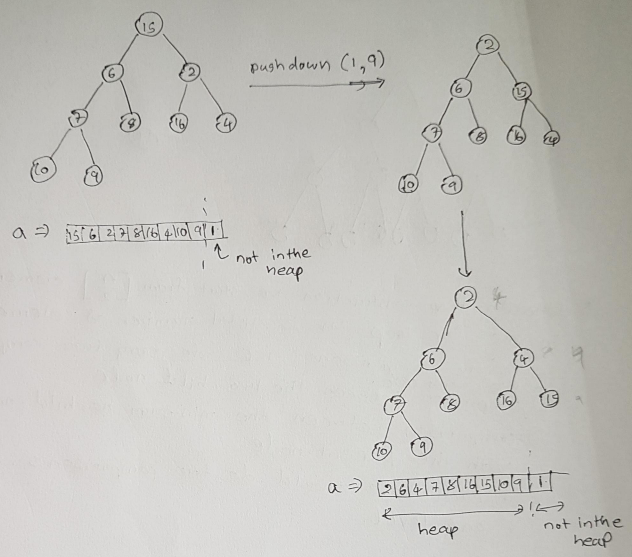
Initial stage →



a ⇒

1	6	2	7	8	16	4	10	9	15

* swapping a[i] with a[1] i= 10



a =>

15	6	2	7	8	16	4	10	9	1

a =>

2	6	4	7	8	16	15	10	9	1

* ~~If we~~ Then we swap a[9] with a[1]. In this also pushdown (1,8) happens as above. We get,

Sorted part.

a =>

4	6	9	7	8	16	15	10	2	1

in the heap

not in the heap

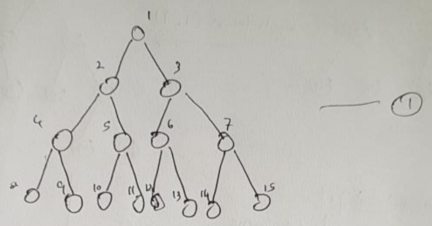
* Accordingly above procedure happens until i= 2. In every i, sorting part happend in the right side. So finaly we get,

a =>

16	15	10	9	8	7	6	4	2	1	[illegible]

* finally We got sorted array as a decending order from left to right.

c) * In initial heap, Let's consider below binary tree.



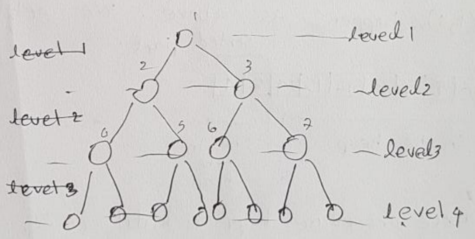
* ~~when~~ In heap construction we start from $\left\lfloor \frac{n}{2} \right\rfloor$ element in the for loop. n is total number of elements. So in the comparisons we do two comparisons, in the worst case.

i) comparison between the two child node.

ii) comparison between the minimum child node with the parent node.

* So each node should do two comparisons. for a one level.

from ①



* In this we start from element 7, and it can go for only 1 level below. So level 2 node can go for two level's of comparisons in worst case.

* worst case comparisons.

level 4 → 0

level 3 → $4 \times 2 \times 1 = 8$

level 2 → $2 \times 2 \times 2 = 8$

level 1 → $1 \times 2 \times 3 = 6$

we can get total comparisons as below for n nodes. for fully complete binary tree. (n is odd)

$$\left(0 \times \frac{(n+1)}{2}\right) + \left(2 \times \frac{(n+1)}{4}\right) + \left(4 \times \frac{(n+1)}{8}\right) + \left(6 \times \frac{(n+1)}{16}\right) \cdots$$

we can write above sum as, kn

~~$\sum_{i=\log_2 n}^{1}$~~ $k =$ some constant.

Total work $= kn$

$\therefore$ complexity $= O(n)$ //

d) * In this i goes from n to 2

when $i=n$, we have the biggest heap.

Then i reduces, which means ~~hea~~ size of the heap is reducing.

Then total levels of ~~of~~ heap for each i $\}= \log_2(i)$

comparisons for each level $= 2$.

$\therefore$ Total work $= \sum_{i=n}^{2} 2 \times \log_2(i)$

$$\sum_{i=n}^{2} 2 \times \log_2(i)$$

* This can be written as some form ass below.

$$= 2n\log_2(n) - 2n\log(e) + \log(2\pi) + \log(n)$$

*dominating part of above is $n\log_2(n)$.

∴ wort-case time complexity for sorting is $O(n\log n)$ //

1.7)

a) V - this is the pivot element ~~wh~~ considered. By using the pivot element, we can separate the array for two sides. One side is containing the elements which are smaller than the pivot element. (left side) Other side contains the elements which are greater than or equal to the pivot elements

b)

1 ↓   10 ↓

3	1	4	1	5	9	2	6	5	3

pivot = 3 ← larger of first two listincts key $l = 1$ $r = 10$

↓ swap a[1] and a[10]

3	1	4	1	5	9	2	6	5	3

$l = 1$ ← a[1] = pivot

$r = 7$ ← a[7] < pivot

↓ swap a[1], a[7]

2	1	2	1	5	9	3	6	5	3

$l = 5$ pivot $\not<$ a[5]

$r = 4$ pivot > a[4]

↓ * $l > r$ → returning $l = 5$

2	1	2	1

* not sorted

5	9	3	6	5	3

* not sorted.

considering →

1			4
2	1	2	1

pivot=2 l=1 r=4

↓ swap a[1], a[4]

1	1	2	2

↓ l=3 ← a[3] = pivot
r=2 ← a[1] < pivot

returning → l=3

1	1

① sorted

2	2

② sorted.

* After doing same steps as above, ~~for~~ ① and ② are sorted now.
So a[1] to a[4] part is sorted now.

1			4
1	1	2	2

* considering

5					10
5	9	3	6	5	3

l=5 r=10 pivot = 9

↓ swap a[5], a[10]

3	9	3	6	5	5

↓ l=6 ← a[6] = pivot
r=10 ← a[10] < pivot
swap a[6] and a[10]

3	5	3	6	5	9

↓ l=10 ← a[10] = pivot
r=9 ← a[9] < pivot
l>r ∴ returning l=10.

3	5	3	6	5

9

* sorted.

• now a[10] is sorted.

considering

5 ↓ … ↓ 9

3	5	3	6	5

pivot = 5   l = 5   r = 9

↓ swap a[5], a[9]

5	5	3	6	3

l = 5 ← a[5] = pivot

r = 9 ← a[9] < pivot

↓ swap a[5], a[9]

3	5	3	6	5

l = 6 ← a[6] = pivot

r = 7 ← a[7] < pivot

↓ swap a[6] and a[7]

3	3	5	6	5

↓ l = 7 ← a[7] = pivot

r = 6 ← a[6] < pivot

return l = 7

↙ ↘

3	3

* sorted

5	6	5

* by using above steps. we can get below

↙ ↘

5	5

* sorted.

6

* sorted

* Finally we get

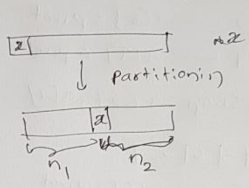
1.	1	2	2	3	3	5	5	6	9

← a is sorted now .

c) * when consider the worst-case, we get this from, when given array of elements are already sorted.

complexity of partitioning is O(n). Berause we check & element by element.

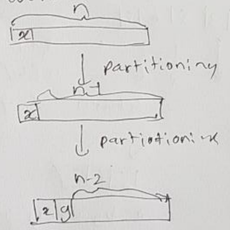
* In the normal case,



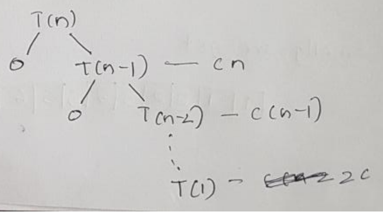
$\therefore T(n) = T(n_1) + T(n_2) +$ time for pantioning $(O(n))$

$T(n) = T(n_1) + T(n_2) + c \times n$

* But in worst case.



* We ~~does~~ don't get the values less than the pivot. Because the array is already sorter. According to this. we get the time as below



- the sizes of elements to sort is changing.

Therefore

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

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

$$\vdots$$

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

∴ So work is changing $cn, cn-1, cn-2, cn-3 \cdots 1$.

∴ Total work $= c((n) + (n-1) + (n-2) + \cdots 1)$

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

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

~~∴ Time complexity = $O(n^2)$~~ //

∴ Time complexity for worst case = $O(n^2)$ //

d) * In place means, in this sorting is done in the given same array: We don't make a new array to get the sorted array. Sorting is done in the same array in place. ~~That is why~~