1. Big O notation [Asymptotic Upper Bound]

$$O(g(n)) = \{ \exists c, n_0 \text{ s.t } 0 \le f(n) \le cg(n) \ \forall\ n \ge n_0 \}$$
$$c > 0 \quad n_0 > 0$$

2. Omega Ω notation [Asymptotic lower Bound]

$$\Omega(g(n)) = \{ \exists c, n_0 \text{ s.t } 0 \le cg(n) \le f(n) \ \forall\ n \ge n_0$$
$$c > 0 \quad n_0 > 0$$

3. Θ Theta notation [Asymptotic Tight Bound]

~~$\Theta(g(n)) = \{ \exists c, n_0 \text{ s.t}$~~

$$\Theta(g(n)) = \{ \exists c_1, c_2, n_0 \text{ s.t } 0 \le c_1 g(n) \le f(n) \le c_2 g(n)$$
$$\{ \exists c_1, c_2, n_0 > 0 \} \qquad \forall n \ge n_0$$

4. small o notation

$$o(g(n)) = \{ \exists c, n_0 \quad c, n_0 > 0 \quad \text{s.t } 0 \le f(n) < cg(n)$$
$$\forall n \ge n_0$$

5. small omega ω notation.

$$\omega(g(n)) = \{ \exists c, n_0 \quad c, n_0 > 0 \quad 0 \le cg(n) < f(n)$$
$$\forall n \ge n_0 \}$$

Analogy →

$a \le b$	$O(g(n))$	b monotonically increasing
$a \ge b$	$\Omega(g(n))$	b — " — decreasing
$a = b$	$\Theta(g(n))$	
$a < b$	$o(g(n))$	b strictly increasing
$a > b$	$\omega(g(n))$	— " — decreasing

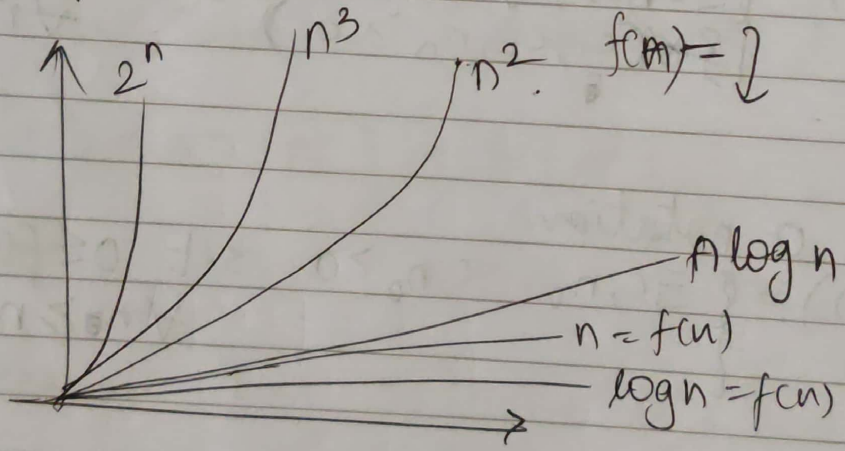
Properties →

- Transitivity $f(n) = \Theta(g(n))$ $g(n) = \Theta(h(n))$ {for all} $\Rightarrow f(n) = \Theta(h(n))$
- for O, Ω, Θ Reflexivity (Big Notation)

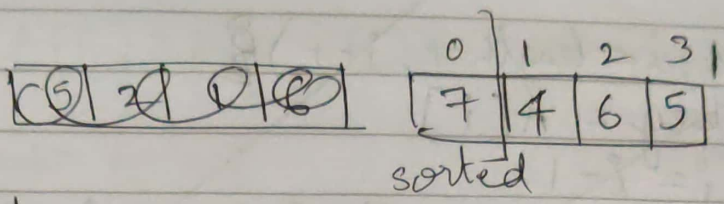
$f(n) = \Theta(f(n))$; $f(n) = O(f(n))$; $f(n) = \Omega(f(n))$

- Symetry $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
  $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$
- Transpose $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

Graph cheat Sheet

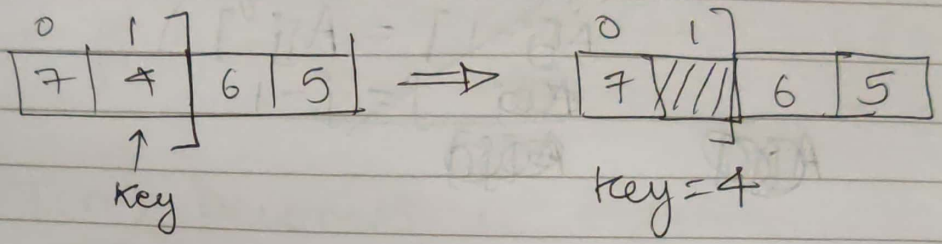


Insertion Sort

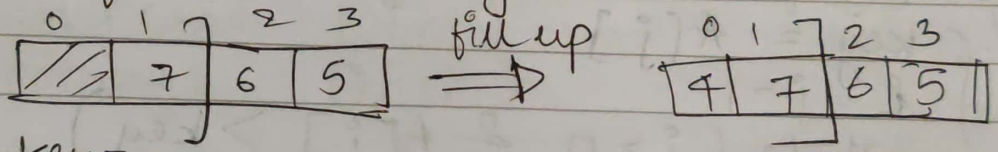


Includes 4.

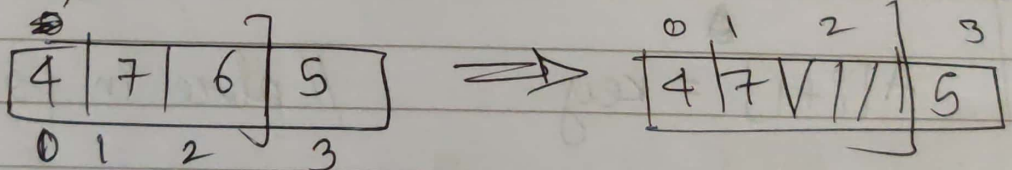
Pass 1



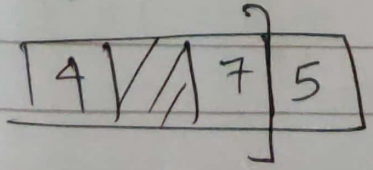
(1) if elements under consideration > key shift those many elements



Pass 2 include next.



If elements do (1) shift if > key



∵ key < 4 place

Repeat

Algo →

```
for (i=1; i< length-1; i++)
    key = A[i]
    j = i-1
    while (j>=0 AND key < A[j])
        A[j+1] = A[j]
        j = j-1
```

Algo

Times Run

```
for (j from 1 to (length-1) ; j++)        n = length-1
{   key = A[j]                            n-1
    i = j-1                               n-1
    while (i>0 & A[i] > key)              Σ_{j=2}^{n} t_j
    {   A[i+1] = A[i]                     Σ_{j=2}^{n} (t_j - 1)
        i = i-1                           same
    }                                     same
    A[i+1] = key                          n-1
```

// do shifts

// place in position

let costs be $C_1, C_2 \dots C_8$

Total Runtime $= C_1 n + C_2 (n-1) + C_3 (n-1) + C_4 \left(\sum_{j=2}^{n} t_j\right) + C_5 \left(\sum_{j=2}^{n} t_j - 1\right) + C_6 \left(\sum_{j=2}^{n} (t_j - 1)\right) + C_7 (n-1)$

Best case: if $t_j = 1$, $\sum_{j=2}^{n} t_j = n-1$, $\sum_{j=2}^{n} (t_j - 1) = 0$

Total $= T(n) = (C_1 + C_2 + C_4 + C_5 + C_8) n + (C_2 + C_4 + C_5 + C_8)$

$= k \cdot n$

$T(n) = \theta(n)$

Worst case – Reverse Sorted Array as inputs

If $A[i] > key$ always $\Rightarrow$ compare with all on left of j^{th} position

$\Rightarrow t_j = j$

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

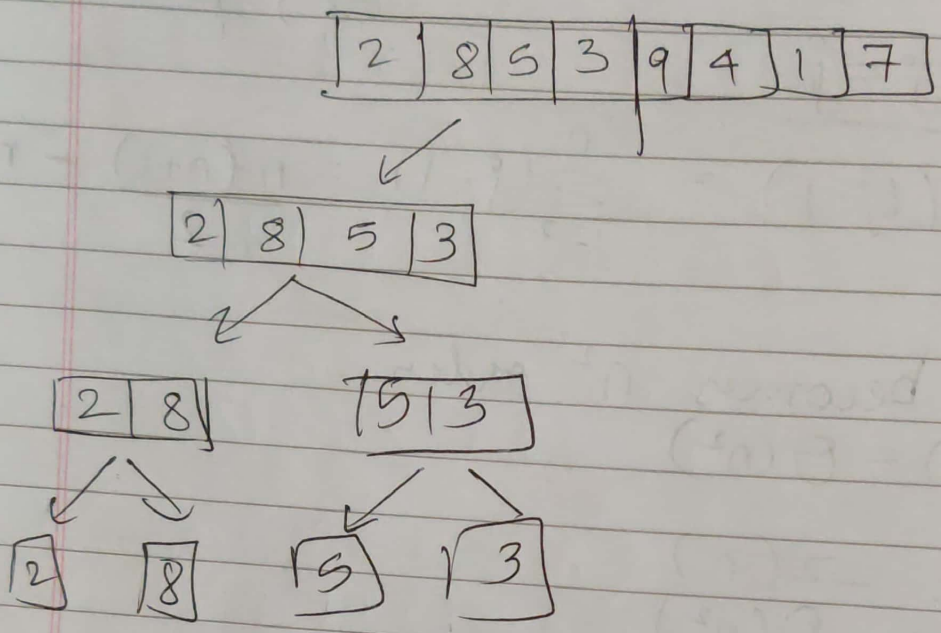
$\therefore T(n)$ becomes n^2 order

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

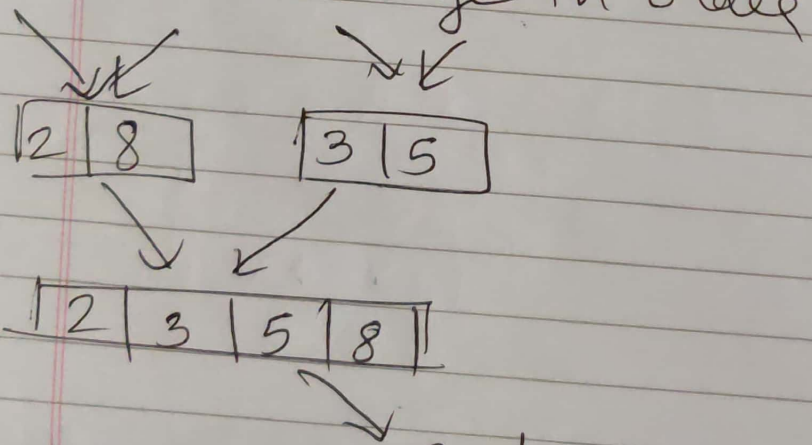
Best case	$\Omega(n)$
Worst case	$O(n^2)$
Average	$\Theta(n^2)$

Merge Sort (A, p, q, r)

Take an Array, Divide into 2



Then we merge in order



and so on.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \end{cases}$$

~~Subproblems of size n/b~~

divide a in subproblems of 1/b
each of size n/b
each takes time T(n/b) ⇒ Total = aT(n/b)

Time to Divide = D(n) = $\Theta(1)$
— I — Combine = C(n) = $\Theta(n)$

Assuming $n = 2^m$
$\therefore T(n) = 2T\left(\frac{n}{2}\right) + D(2^m) + C(2^m)$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + C_1(n) & n > 1 \end{cases} \rightarrow T(n) = \Theta(1)$$

~~$T(n) = \Theta(n \log n)$~~ ~~$T(n) = \Theta(n(\log n))$~~

$T_n = \Theta(n \log n)$

Algo.

```
Divide (A, p, r)
    if (p<r)
    {   q = (p+r)/2   // mid point
        Divide (A, p, q)
        Divide (A, q+1, r)
        conquer (A, p, q, r)
```

~~Conquer (A, p, q, r)~~

Conquer(A, p, q, r) →
(mid+1 - low) (high - mid)

$n_L = q - p + 1$, $n_R = r - q$

Let $L[n_L + 1]$ $R[n_R + 1]$ arrays

```
for (i from 1 to nL)
    L[i] = A[p+i-1]

for (j from 1 to nR)
    R[j] = A[q+j]

for (p ≤ k ≤ r)
    if L[i] ≤ R[j]
        { A[k] = L[i]
          i = i+1
    else
        { A[k] = R[j]
          j = j+1
    k++ = k+1
```

Masters Theorem $a, b, f(n) > 0 \mid T(n) = aT\left(\frac{n}{b}\right) + f(n)$

- Case 1
  if $f(n) = O\left(n^{\log_b^{(a-\epsilon)}}\right)$ for some $\epsilon > 0$
  Then $T(n) = \Theta\, n^{\log_b^a}$

- Case 2
  if $f(n) = \Theta\left(n^{\log_b^a} \cdot \lg^k n\right)$ Then $\left\{T(n) = \Theta\, n^{\log_b^a} \lg^{k+1} n \;\; \forall\, k \geq 0\right\}$

- Case 3
  $f(n) = \Omega\left(n^{\log_b^{a+\epsilon}}\right)$ for $\epsilon > 0$ if $af\left(\frac{n}{b}\right) \leq cf(n)$
  for some $c < 1$ for large n
  $T(n) = \Theta(f(n))$

Shortcut →
$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$

* $a \geq 1$ $b \geq 1$ $k \geq 0$ p is a real number

① if $a > b^k$ ⟹ $T(n) = \Theta\left(n^{\log_b^a}\right)$

② if $a = b^k$
- if $p > -1$ ⟹ $T(n) = \Theta\left(n^{\log_b^a} \lg^{p+1} n\right)$
- if $p = -1$ ⟹ $T(n) = \Theta\left(n^{\log_b^a} \lg^2 n\right)$
- ~~if $p \leq -1$ ⟹ $T(n) = \Theta(n^k \log^p n)$~~
- if $p < -1$ ⟹ $T(n) = \Theta\left(n^{\log_b^a}\right)$

③ if $a < b^k$
- if $p \geq 0$ ⟹ $T(n) = \Theta\left(n^k \log^p n\right)$
- if $p < 0$ ⟹ $T(n) = \Theta(n^k)$

Max Sub Array.

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

Algo →

Main Array = 100, 113, 110, 85, 105, 102 ....

Index = 0 1 2 3 4 .. .. n-1

delta = [—] 113-100, 110-113, 85-110, ... A[i]-A[i-1]

Index = x, 0, 1, 2 . . . n-2

Algo

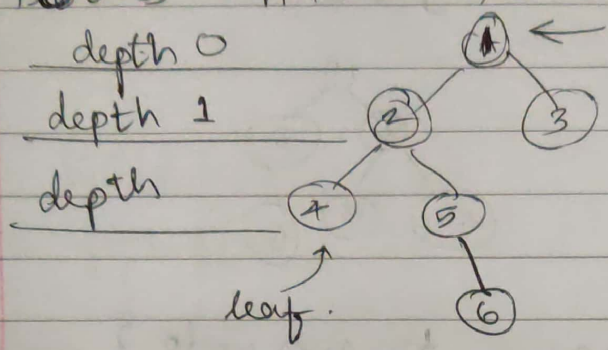
```
Divide (A, low, high)
  if(low == high)
  { return low, high, A[low]
  else
    mid = (low+high)/2  // int not float
    (leftlow, lefthigh, leftsum) = Divide(A, low, mid)
    (right ——— ) = Divide (A, mid+1, high)
    (crossing ——— ) = Find cross (A, low, high, mid)

  if leftsum > rightsum & leftsum > cross sum
     return (leftlow, left high, left sum)
  else if right ——

  else return (cross , , )
```

```
Find Cross (A, low, high, mid)
  leftsum = -∞
  sum = 0
    for (i from mid to low)  //including "low"
    {
      ~~if (sum > leftsum)~~
        ~~leftsum~~
      sum = sum + A[i]
      if (sum > leftsum)
        { leftsum = sum
        { maxleft = i
  // similarly for right part
    return (maxleft, maxright, leftsum + rightsum)
```

→ no. of nodes at level h $\leq d_k^h$ tree base $bi = 2$ $tri = 3$

Tree

Terms → Ancestor, Descendent, parent, child, siblings



1 is ancestor of 4 & 5
2 is parent of 4 & 5
4,5 are child of 2
2,3 are child of 1
2,3 are siblings
4,5 are siblings.

Node Terms

degree, Depth, Level, Height

- degree of 3 is 0 (no. of children of a node x)
  degree of 2 is 2
- depth is length of simple path from root
- Level = nodes at same depth
- Height = no. of edges on longest simple downward path to a leaf (Height is 3 in above case)

Binary Tree →

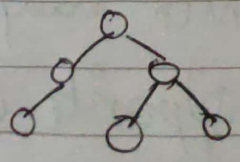
each node degree ≤ 2

Full Binary tree degree(x) is 1 or 2

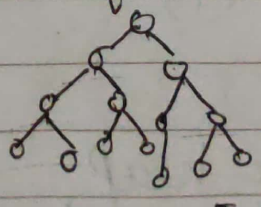
{ Perfect binary Tree degree(x) = 2 or zero and $2^h - 1$ nodes where h is height

{ Complete binary Tree → perfect till h-1 then has some leaf to the left.

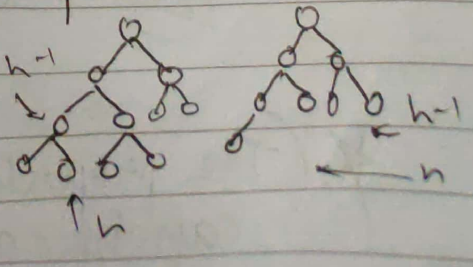
Full



Perfect



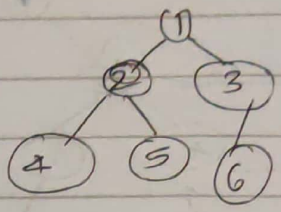
Complete



Heap

- It is a complete d-nary tree (we studied b
  In array form

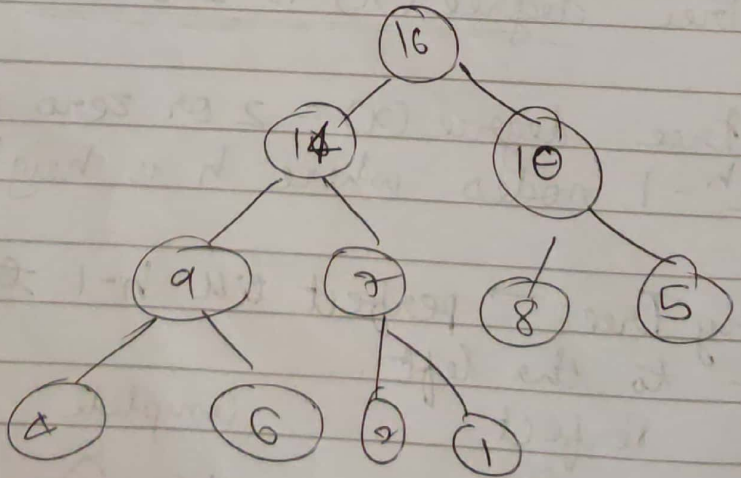
A = | 1 | 2 | 3 | 4 | 5 | 6 |



* ⇒ for $d = 2$ Bi-nary for any node i
  Parent $A[\lfloor i/2 \rfloor]$ Child 1 $= A[2i]$ | Child 2 $= A[2i+1]$

~~for any d~~
for $d > 2$ use linked list

Properties of Heap
Root is A[1] or A[0]
Every ~~sub~~ subtree is also a heap →
$0 \leq A.heapsize \leq A.length$
This means value of parent node is always the max or min of the children



each parent is greater than children (MAX Heap)

{ Min heap is opposite of this

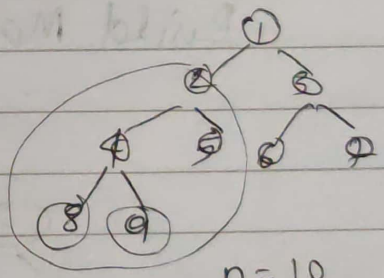
* (The property of the heap can be anything, not always max or min)

Algo for MAX HEAP

Step ① Build the heap → $T(n) = O(n)$

```
Build MaxHeap (A, n)
  for (⌊n/2⌋ ≥ i ≥ 1)   // look for all parents.
  { MAX-HEAPIFY(A, i, n)
```

```
MAX-HEAPIFY(A, i, n)
l = left(i)    r = right(i)   // get left & right child index
if (l ≤ n & A[l] > A[i]) then
    largest = l
else
    largest = i
if (r ≤ n & A[r] > A[i])
    largest = r
if (largest ≠ i) then
  { swap A[i] & A[largest]
    MAX-HEAPIFY(A, largest, n)
```



n = 10

at most subtree size = $\frac{2n}{3}$

$= \frac{2 \times 10}{3} \approx 6$

(half full case)

→ each children subtree size atmost $(2n/3)$

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

$p = 0$

$a = 1 \quad b = 3/2 \quad k = 0$

$a = b^k \Rightarrow$ MT 2

$T(n) = \Theta(n^k \log^{p+1} n) = \Theta(\log n)$

(using MT. 2)

(for Heapify)

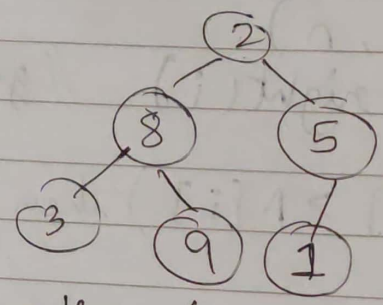
slower than merge sort but takes less sp

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

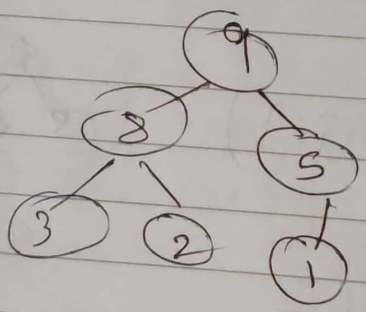
```
BuildMaxHeap (A, n)
for( n ≥ i ≥ 2) do
{ swap A[1] and A[i]
{ MAX-HEPIFY ( A, 1, i-1)
od
```

n times → $O(\log n)$

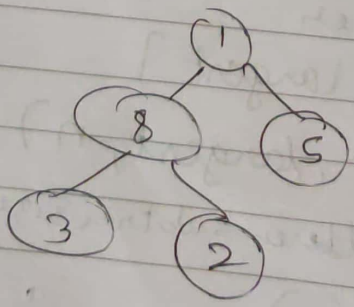
2	8	5	3	9	1



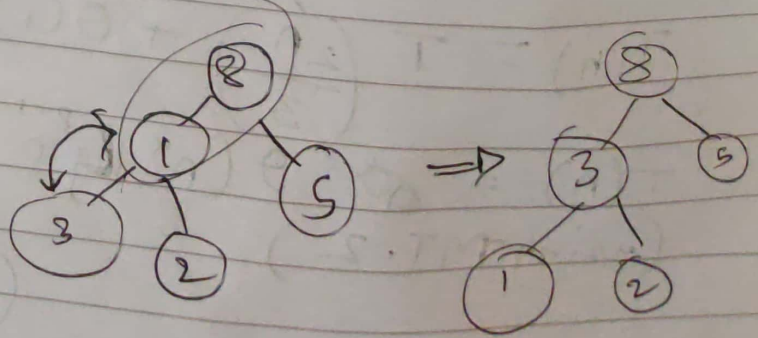
Build MaxHeap()



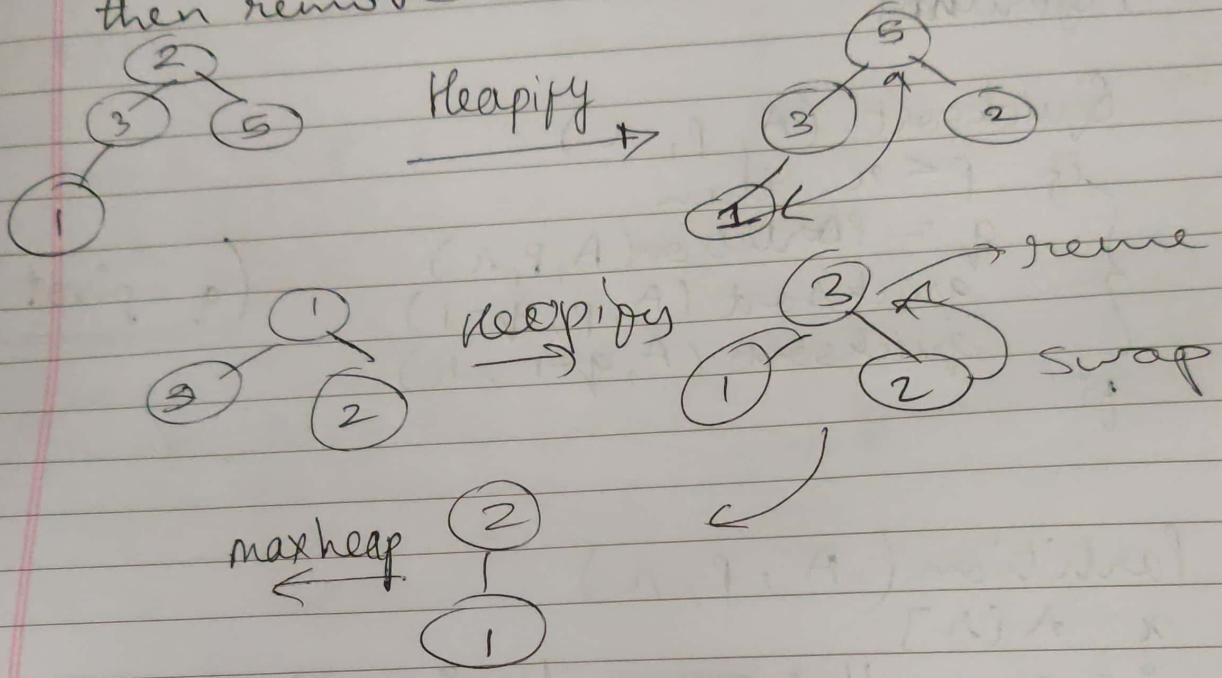
Swap 9 & 1



Heapify()



swap 8 with item at end i.e. 2
then remove 8.



Steps

1. Assume unsorted
2. call build max heap → calls heapify.
3. take largest item swap with last leaf remove largest to some sorted section. call heapify.
   repeat (3) n-1 times.

Quicksort

Algorithm

```
Quicksort (A, p, r)
if p < r then
    q = Partition(A, p, r)
    quicksort (A, p, q-1)
    quicksort (A, q+1, r)
fi
```

(q = pivot)

```
Partition ( A , p, r)
x = A[r]
i = p-1  // or randomize it Random(p, r)
for (p ≤ j ≤ r-1) do
    if ( A[j] ≤ x) then
        i = i+1
        swap (A[i], A[j])
    fi
od
SWAP (A[i+1], A[r])
return i+1
```

0	1	2	3	4	5	6	7
8	1	6	4	0	3	9	5

$p = 0 \quad r = 7$

$p < r$ ✓

$q = Part(A, p, r)$

$x = A[r]$

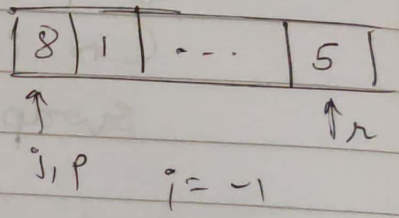
$i = p - 1 = -1$

for $j = p$; $j \leq r-1$; j++

loop 1 $j = 0$; $i = -1$

$A[j] \leq x$?

no ; j++



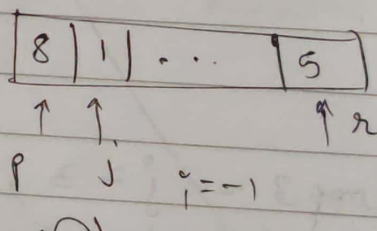
loop 2 $j = 1$, $i = -1$

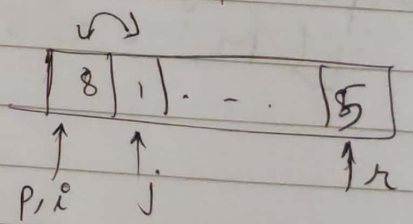
$A[j] \leq x$ ✓

↳ $i = i + 1 = -1 + 1 = 0$

∴ swap $A[i], A[j]$

loop back

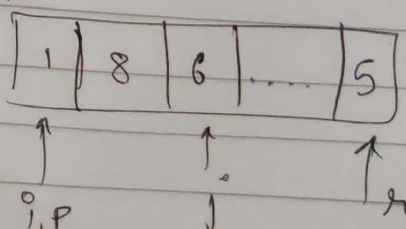




loop 3 $j = 2$; $i = 0$

$A[j] \leq x$

false

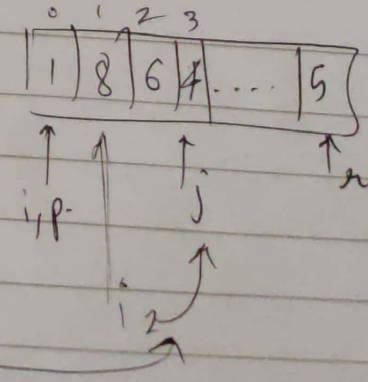


loop 4 $j = 3$ $i = 0$

$A[j] \leq x$? yes

$i = i + 1 = 0 + 1 = 1$

swap $A[i]$ $A[j]$



goes on till j reaches r-1

0	1	2	3	4	5	6	7
1	4	0	3	6	8	9	5

i, j, r

if fails.

$i = 3; j = 6$

after for

↳ swap (A[i+1] & A[r])

swap "6" & "5"

1	4	0	3	5	8	9	6

return i+1 = 3+1

$\therefore \boxed{q = 4}$

↳ Partition (A, p, q-1) // we placed 5

↳ Repeat all

* We ~~set~~ find pivots spots and place pivots in the correct spot in every recursion till every pivot in correct spot.

if array balanced good as merge sort = $\Theta(n \lg n)$

Worst if unbalanced as bad as insertion sort

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

Average = $\Theta(n)$

9:1 split always = balanced → ~~$T(n) + T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right)$~~

1:1 —u— = balanced

etc....

Best case

for a x:y split always.

$$T(n) = T\left(\frac{x\,n}{x+y}\right) + T\left(\frac{y\,n}{x+y}\right) + \Theta(n)$$

Counting Sort $T(n) = \Theta(n+k)$

if $k = O(n)$ $T(n) = \Theta(n)$

Algorithm →

~~create new array C[0 . . k]~~

CountingSort (A, B, n, k)

// A input

// B output

// n = length of A

create new array C[0 . . . k]

for ($0 \le i \le k$) do

  C[i] = 0

for ($1 \le j \le n$) do // A goes 1 to n

  C[A[j]] = C[A[j]] + 1 // count. reccurrence.

for ($1 \le i \le k$) do

  C[i] = C[i] + C[i - 1] // Sum till i in C

for ($n \ge j \ge 1$) do // ($j = 1$ $j \le n$ $j++$)

  B[C[A[j]]] = A[j]

  C[A[j]] = C[A[j]] - 1

	0	1	2	3	4	5	6	7
A =	2	5	3	0	2	3	0	3

for C[A[i]] += 1

	0	1	2	3	4	5
C =	0	0	0	0	0	0

i = 0 to 7 for A

i = 0 A[i] = 2

C[A[i]] = C[2] ∴ C[2]++

	0	1	2	3	4	5
C =	0	0	1	0	0	0

i++ i = 1 A[1] = 5 C[5]++

	0	1	2	3	4	5
C =	0	0	1	0	0	1

i++; i = 2; A[2] = 3 C[3]++

and so on.

we get

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

Now sum till i in C for all i

⇒ i = 0 to K = 5 {we cant do for i = 0 as i-1 = -1}

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

i = 1 ⇒ C[1] = C[1] + C[0]

C[1] = 0 + 2 = 2

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

and so on we get

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

Now we place the numbers back.

$j = 0$ to $7 < n-1$

$B[C[A[j]]] = A[j]$

$j = 0$

$B[C[A[0]]] \rightarrow B[C[2]] \rightarrow B[4]$

$\therefore B[4] = A[0] = 2$

B					2			
	0	1	2	3	4	5	6	7

and so on we get →

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

maintains order of the way we read it in.

This means the 2 at A[0] is placed first before the 2 at A[4]

Hence stable sort

★ Stable Sort →

★ keys with the same value appear in the same order in the output as they appeared in the input

~~Radix Sort (A, d)~~
~~for (1 ≤ i ≤ d) do~~
~~{sort (Array A on digit i)~~ with stable sorting algo)

~~example~~

Radix Sort

for (i from 1 to d)
{Sort Array A on digit i with a stable sorting algorithm. $\Theta(d \cdot \Theta(\ ))$}

Concept: sort least significant digits first

d = highest order digit

e.g

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

→ sort

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

sort ↙

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

sort ←

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

Radix Sort for strings.
we have n words broken into r pieces with
b bits per word.

$$d = \left\lceil \frac{b}{r} \right\rceil$$

we balance $\frac{b}{r}$ & $n + 2^r$

Bucket Sort

D nary Array

① root = A[1]
for any i

* $\boxed{\text{Parent} = A\left\lceil \frac{i-1}{d} \right\rceil}$

$\text{children} = \{di-d+2, di-d+3 \cdots di-d+d, di-d+d+1\}$

$\boxed{\text{Children} = \{di-d+j\} \quad (2 \le j \le d+1)}$ *

② In any tree no. of nodes at level h is $\le d^h$

The total no. of nodes (maximum) $= 1 + d + d^2 \cdots + d^h = \Theta(d^h)$

putting d^h as n we get
height $= \Theta(\log_d n)$

③ D-ary heapify

do max-heapify
l = max (children of A[i], A[i])
~~max (elements of A~~
if $l \neq i$ exchange A[i] A[l]
Heapify (A, i)

$\therefore T(n) = \Theta(d \log_d n)$

Heap increase key (A, i, k)

A[i] = k

while i > 1 A(parent(i)) < A[i]

do exchange A[i] ⟷ parent(i)

i = parent(i)