UNIT – III

Transform and Conquer:

Presorting: Best technique for transform & conquer.

```
Algorithm PresortEleUniqueness (A[0...n-1])
// solves the ele uniqueness by sorting the array first.
// input: an array A[0...n-1] of orderable elements.
// output: returns 'true' if A has no equal ele, 'false' otherwise.
for i ← 0 to n-1 do
if A[i] = A[i+1]
return false
else
return true.
```

Efficiency:

Here, the alg has 2 main components: 1. sorting the elements.
2. checking consecutive elements. } Time consuming

1. What time complexity for merge sort (good sorting alg) is $O(n \log n)$.

$$C(n) = 2\,C(n/2) + C_{worstcase}$$

$$= 2\,C(n/2) + (n-1)$$

$\Rightarrow$ By M.T:

$$a = 2 \quad b = 2 \quad d = 1$$

$$b^d = 2$$

$$\therefore a = b^d \quad \therefore T(n) = \Theta(n \log n)$$

2. For checking consec ele / scanning, the time complexity is $O(n)$ as it takes only one linear scan of the sorted list.

$\therefore T(n) = \underbrace{O(n \log n)}_{T_{sort}} + \underbrace{O(n)}_{T_{scan}} = O(n \log n)$ since $O(n \log n)$ is the dominant term.

$O(n \log n)$ is better than quadratic $O(n^2)$ efficiency of brute-force method i.e bubble sort, selec sort.

Q) Construct a 2-3 tree for the foll: (data ↑, children ↑)

(i) 5 12 11 6 10 18 16 4

⇒ nodes with two children are called 2-Nodes. Nodes with three children are called 3-nodes.

⇒ 2-nodes : one data val & two children.

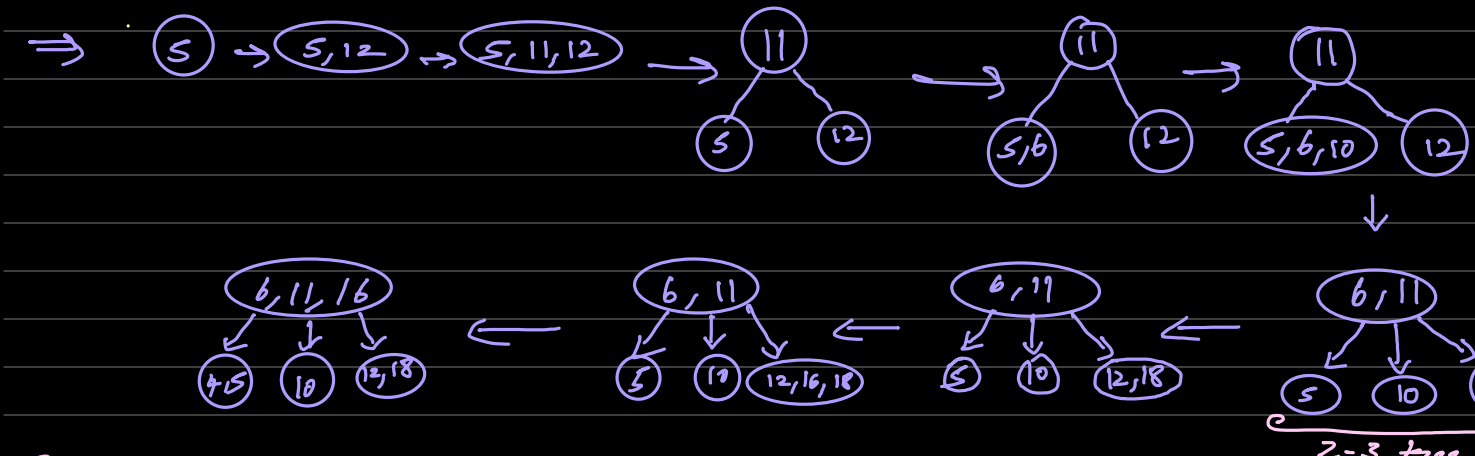
⇒ 3-nodes : two data vals & three children.

⇒ all leaf nodes are at same level.

⇒ each node can either be a leaf node, 2-node, 3-node.

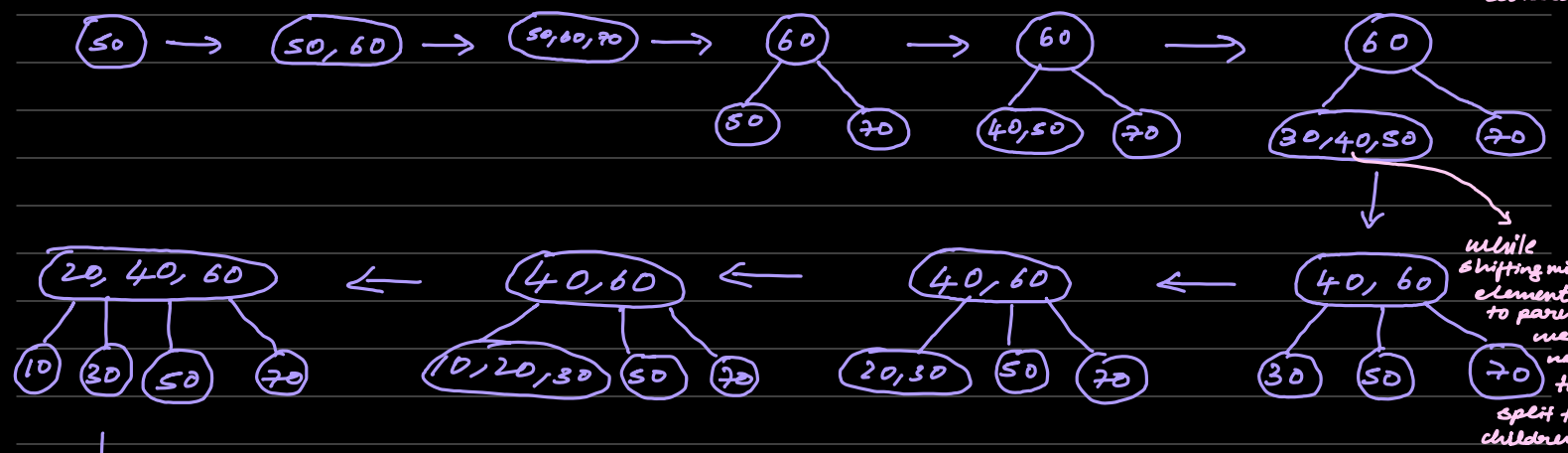
⇒ insertion is always done at leaf.

(do not write commas, write ':')

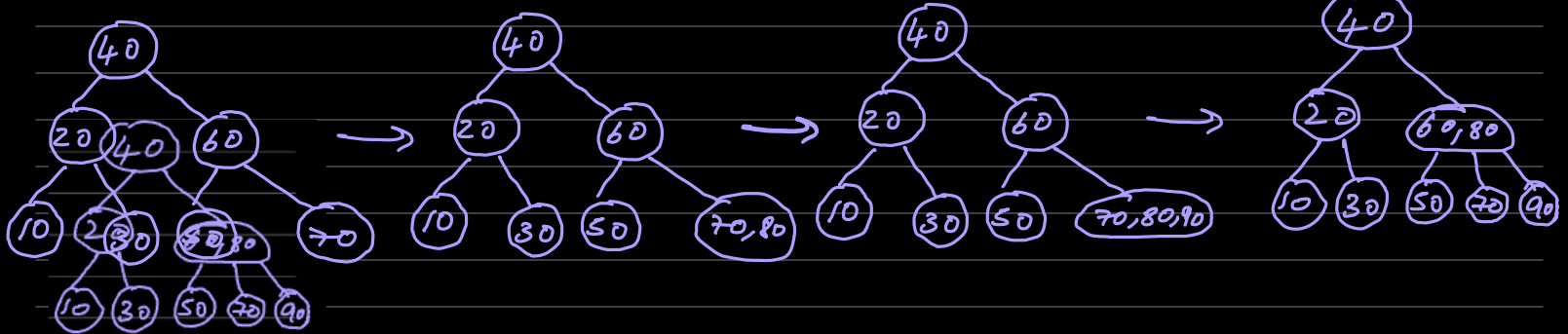


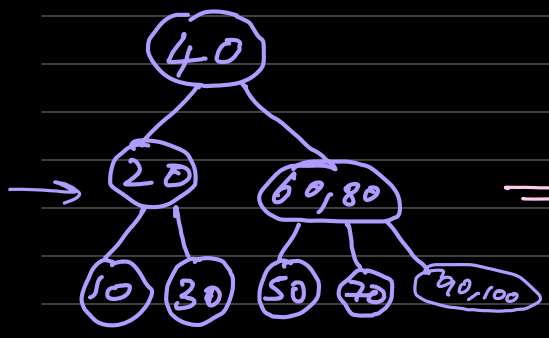
2-3 tree, wherever we have 2 data ele, we should have 3 children

(ii) 50, 60, 70, 40, 30, 20, 10, 80, 90, 100.



while shifting mid element to parent we need to split the children.





⟹ Balanced
2-3 tree //.

Mode by presort: (not imp)

→ mode: val that appears most frequently.
→ alg is designed to find the mode by sorting the array first.

1. sort the array to make it easier to count no. of occurences as they will be adjacent to each other.
2. no. of occurences is initialized to '0'.
3. while loop to find the mode.

```
ALGORITHM PresortMode(A[0....n-1])
// computes the mode of array by sorting it first.
// input: an array A[0...n-1] of orderable elements.
// output: the array's mode
sort(A)
i ← 0
modefreq ← 0
while i ≤ n-1 do
runlength ← 1
runvalue ← A[i]
while i+runlength ≤ n-1 and A[i+runlength] = runval
runlength ← runlength+1
if runlength > modefreq
modefreq ← runlength
modeval ← runval
i ← i+runlength
return modeval
```

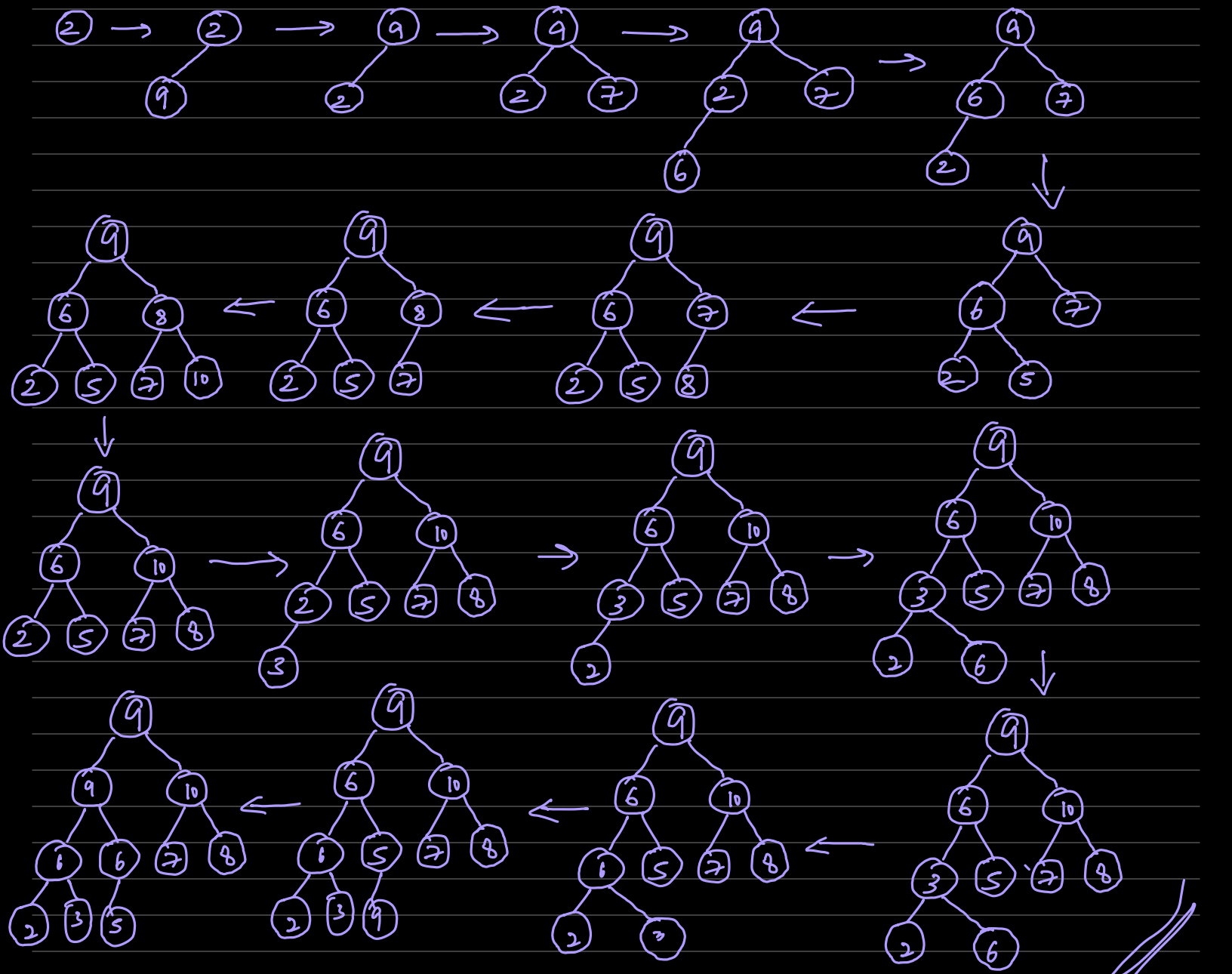
Heapsort:

→ A heap is a binary tree where 2 conditions has to be met:
1. Structural property: complete tree or simply complete.
2. Parental dominance property / heap property: greater than keys in child nodes.

→ 2 Types of heaps:
*1. max heap parent > child
2. min heap parent < child

→ 2 construc methods:
*1. Top down.
2. Bottom up.

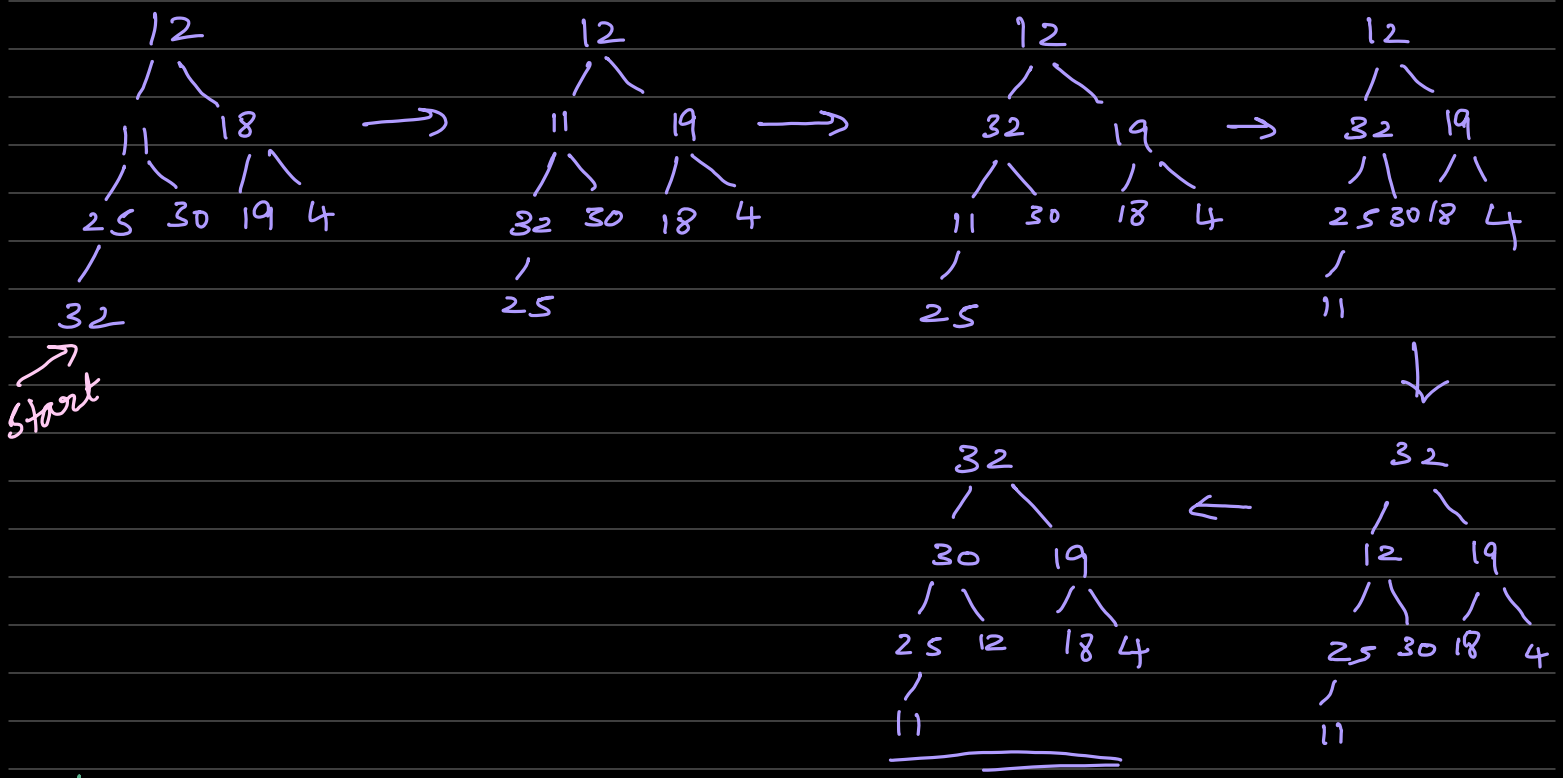
a) 2 9 7 6 5 8 10 3 6 9, max heap by top down method.



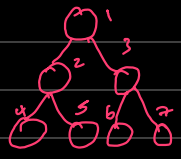
(a) 12 11 18 25 30 19 4 32, maxheap

(i) Top-down:

(ii) Bottom-up: (first construct a Binary tree)



(i) Alg to construct a heap:



$n = 7$

$\lfloor \frac{n}{2} \rfloor \Rightarrow$ no. of non-leaf nodes

$\lfloor \frac{7}{2} \rfloor = 3.5 = 3$

position of node = i
left child = 2 * i
right child = 2i + 1.

ALGORITHM Heapify (H[1...n])
// constructs max heap using bottom-up approach.
// input: an array H of n ele
// output: Heap, represented using an array.
for $i \leftarrow \lfloor \frac{n}{2} \rfloor$ down to 1 do // to construct a heap, no. of iterations shd be tot no. of leaf nodes to first i.e 1.
  $k \leftarrow i$ // current index = i
  $v \leftarrow H[k]$ // stores val of cur node in 'v'
  heap $\leftarrow$ FALSE // flag
  while not heap and $2k \leq n$ do // until not heap & cur node has atleast one child.
    $j \leftarrow 2 * k$ // sets j as left child of 'k'.
    if $j < n$ // if cur node has a left child.
      if $H[j] < H[j+1]$ then // if right child is > than left, set 'j' to right child.
        $j \leftarrow j + 1$
      if $v \geq H[j]$ then // if val of cur node $\geq$ larger child, heap satisfied.
        heap $\leftarrow$ TRUE // flag -> true
      else
        $H[k] \leftarrow H[j]$ // or move large child up to cur node position and keep heapifying the subtree rooted at 'j'.
        $k \leftarrow j$ // update 'k' to child's index
  $H[k] \leftarrow v$ and continue.
  // after loop, place ini val 'v' in its correct pos in the heap

Efficiency:

⇒ let h ⇒ height of tree and $h = \lfloor \log_2 n \rfloor$

$$C(n)_{worst} = \sum_{i=0}^{h-1} \sum_{\substack{\text{level} \\ i \text{ keys}}} 2(h-i) \qquad H[i] \geq \max\left[H[2i], H[2i+1] \ldots \left\lfloor \frac{n}{2} \right\rfloor\right]$$

$$= \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$$

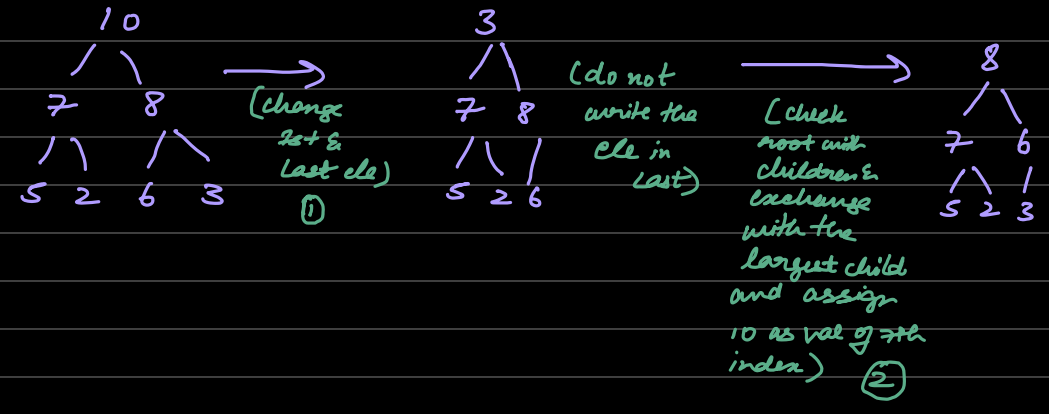
② Alg for heapsort: → heap construction → max deletions

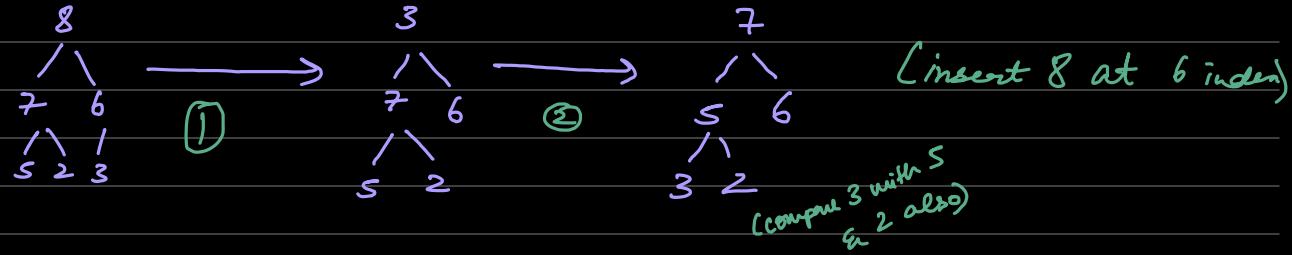
Procedure to apply heapsort:

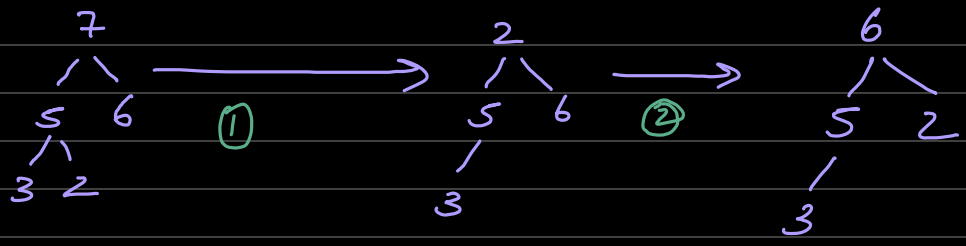
1. Construct a max heap by any method.
2. exchange the 1st and last element and reduce the size of heap by 1. Construct heap for remaining $(n-1)$ elements.
3. repeat step 2 until only one ele is left out.

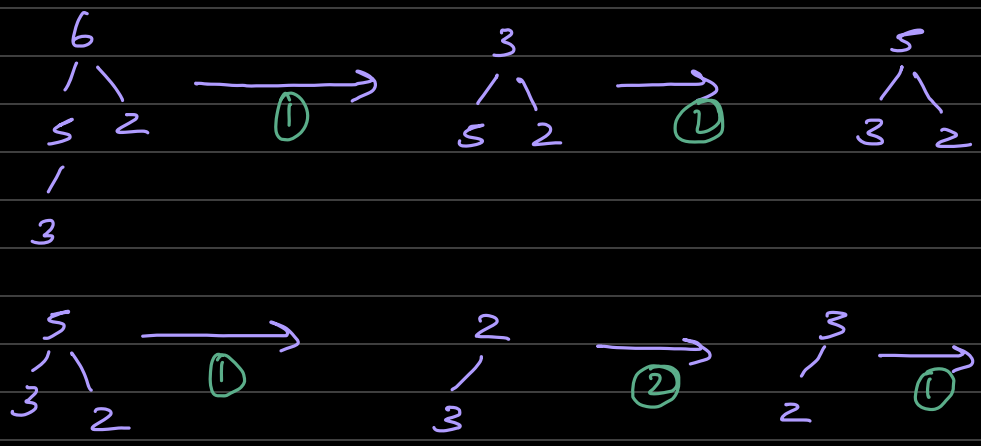
Ex) Apply heapsort for: 2 (1) 7 (2) 6 (3) 5 (4) 10 (5) 8 (6) 3 (7) → index.

max heap:









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

heapsort.

ALGORITHM Heapsort(H[1...n])

//

//

Heapify(H[1...n]) // max heap for all ele.

for i ← n down to 1 do

swap(H[1] and H[i]) // swap the root and lowest element

Heapify(H[1...(n-1)]) // heapify and add to array (do not consider last ele)

Efficiency:

construction of heap + exchanging 1st & last element (n-1) times = efficiency of heapsort

- construction of heap
  - Bottom up = O(n)
  - Top down = O(log n)
- exchanging 1st & last element (n-1) times
  - ↳ (n-1) log n as height of tree is log n

⇒ highest one is (n-1) log n . 1 = const ∴ O(n log n)

Input Enhancement in string-matching:

→ It is an idea to preprocess the problem's input. (in whole or in parts) and to store the additional info obtained to accelerate to solve the problem afterwards.

(i) Sorting: counting → comparision counting sort.
→ distribution counting sort.

(ii) String matching: Horspool's alg.
Boyer moore's alg.

---

(1) Comparision count sort:

input array A[0....5] → 62 (0) 31 (1) 84 (2) 96 (3) 19 (4) 47 (5)

Count: 0 0 0 0 0 0 (initially)
(array) 0 1 2 3 4 5 (index)

(62) Count 1: 5 0 1 1 0 0
(1st iteration: 62 31 84 96 19 47
31 is less than 62 so update the count of 62. Next compare 84 with 62 as 62 is smaller, count of 84 is updated. Similarly for 92 & 62, 92 count is updated, for 19 and 62 count of 62 is updated to 2. For 47 & 62 count of 62 is updated as 3)

(31) Count 2: 3 1 2 2 0 1 (62 31 84 96 19 47)
(discard 1st ele & do the same for 31 & other elements. * Update prev count itself *)

(84) count 3: 3 1 4 3 0 1
(96) count 4: 3 1 4 5 0 1
(19) count 5: 3 1 4 5 0 2 //

∴ Final count array:

	62	31	84	96	19	47
Count	3	1	4	5	0	2
	→ elements					→ index no.

∴ final sorted array:

	0	1	2	3	4	5
S	19	31	47	62	84	96
	→ index					→ elements.

ALGORITHM ComparisionCountSort(A[0...n-1])
// sorts an array by comparision counting
// input: array A of n ele.
// output: array in increasing order.
for i ← 0 to n-1 do // ini all as 0
Count[i] ← 0

```
for i ← 0 to n-2 do    // for iterations. // first ele loop
  for j ← i+1 to n-1 do    // 2nd ele loop
    if A[i] < A[j]    // compare.
      Count[j] ← Count[j] + 1    // increment count for whichever ele is larger.
    else
      Count[i] ← Count[i] + 1
for i ← 0 to n-1 do
    S[Count[i]] ← A[i]    // array is placed in final sorted array
return S.
```

Time Complexity:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1)-(i+1)+1 = \sum_{i=0}^{n-2} (n-1)-(i-1)+1$$

$$= \sum_{i=0}^{n-2} (n-1)-(i) = \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \quad \text{(split)}$$

$\sum_{i=0}^{n-2} i$ scales $i = 1, 2 \ldots n$

$$= (n-1) \sum_{i=0}^{n-2} 1 - [0+1+\ldots(n-2)]$$

$\sum_{i=0}^{n-2} 1$ solve

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

$$= (n-1)\left[\frac{n-1}{2}\right] = (n-1)\left[n \times \frac{1}{2}\right] = n^2 = O(n^2)$$

$(n-1)\left(\frac{n-1}{2}\right)$ → $n-1 \cdot n + 1$ → $(n-1)\left(n - \frac{n-2}{2}\right)$

$(n-1)$ const

$\left[\frac{1}{2}\right]$ const

Quadratic

$0+1+\ldots n = \frac{n(n+1)}{2}$

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

if $(n-2)$ in place of n ⇒ $\frac{(n-2)(n-2+1)}{2}$ ⇒ $\frac{(n-2)(n-1)}{2}$

---

② Distribution Count Sort:

Input array A[0...5] = 13 11 12 13 12 12

lower range : 11

upper range : 13

frequencies :	11	12	13
# elements	1	3	2

Distribution values (freq of ele + dist val of prev ele):

11	12	13
1	4	6
	1+3	4+2

with the help of dist vals we can fill the final sorted array.

	0	1	2	3	4	5
S	11	12	12	12	13	13
	1st ele	3-1 2nd ele	4-1 3rd ele	4th ele	6-1 5th ele	6th ele

array:

0	1	2	3	4	5
13	11	12	13	12	12

(start from the last & check its dist val. In the sorted array, it will be in that pos. Then, decrement the dist val to 3. Next 12 will be at 3rd index.. so on.)

present dist vals will be:

11	12	13
0	1	4

by these vals we can say that the 1st occurance of '11' is at 0 index. 1st occurance of '12' is at 1 index and '13' is at 4 index.

ALGORITHM DistCountSort(A[0...n-1], l, u)
// sorts an array of integers from a limited range.
// input: array A & range l and u.
// output: an array S.
for j ← 0 to u-l do D[j] ← 0 // freq = 0 to all ele.
  for i ← 0 to n-1 do D[A[i]-l] ← D[A[i]-l]+1 // find freq.
    for j ← 1 to u-l do D[j] ← D[j-1] + D[j] // dist val.
      for i ← n-1 down to 0 do
        j ← A[i]-l
        S[D[j]-1] ← A[i]
        D[j] ← D[j]-1
return S

Time Complexity:

⇒ C(n) = O(n) = linear

Horspool's Alg:

→ it is a string matching algorithm.
→ the goal is to find the pattern (P) in the text (T).
→ size of text = n characters T[0...n-1]
→ size of pattern = m characters T[0...n-1]
→ In Brute force (Naive) string matching, limitations is that the shift size is limited to 1. (left to right comp)
→ In Horspool algorithm, we do some sought of preprocessing to the pattern, before searching it in the text. (right to left comp)

Text size: n char
Pattern size: m char

scan right to left
if text is not present, shift entire word.

case (i):
T:
P: B A R B E R _ _ _ _ _ _
B A R B E R

case (ii):
T:
P: B A R B E R _ _ _ _ _
B A R B E R

if text not present, but is present front then shift that many times only.

case (iii):
T: X X K E R (1st case)
P: L E A D E R _ _ _ _
L E A D E R

case (iv):
T: K E R
P: B A R B E R _ _ _ _ _
B A R B E R

if text present, go to next one. If next one is not P, then, search for the rightmost text. If still not P, shift fully.

if text R, go to next, if not R, search for rightmost text. If R, shift that many times

① Text: JIM_SAW_ME_IN_A_BARBER_SHOP
Pattern: BARBER.

(shift by shift val of R & not A)

① JIM_SAW_ME_IN_A_BARBER_SHOP
BARBER
BARBER
BARBER
BARBER
BARBER
BARBER

Shift Table:

E - 1
B - 2
R - 3
A - 4

all other chars = 6 is shift value.

(shift vals one to 1st occ of text from rightmost el)

Q) I: FAIL_MEANS_FIRST_ATTEMPT_IN_LEARN (shift val of N, not I)

P: EARN

Shift table

R - 1	others = 4
A - 2	
E - 3	
N - 4	

pattern : m char
text : n char

ALGORITHM ShiftTable (P[0...n-1])
// fills the shift table used by horspool alg.
// input: pattern P[0...n-1]
// output: Table [0...size-1] indexed by alphabet's chars & filled with shift size.

→ size of pattern

Initialize all the elements of table with m.
for j ← 0 to m-2 do

Table [P[j]] ← m - 1 - j

return Table

ALGORITHM Horspool (P[0...m-1, T[0...n-1])
//implements horspool algorithm for string matching
// input: horspool alg for string matching
// output: index of left end of the first matching substring or -1 if no matches are available.

→ input

ShiftTable (P)
i ← m - 1
while i <= n-1 do
k ← 0 // no. of matched characters
while k <= n-1 & p[m-1-k] == T[i-k] do
k ← k+1
if k == m
return i - m + 1

T: ——— i
P: ——— ↑ m-1

```
else
i ← i + Table [T[i]]
return -1
```

Time complexity: C(worst) = O(nm)

C(avg) = O(n) //

Boyer Moore algorithm: → Bad symbol shift. → good suffix table.

Bad symbol shift: (d_1)

ex: T: S E R

P: B A R B E R

(in case of horspool, we would shift no. of times wrt shift val of R i.e 3)

(In Boyer-moore we shift wrt non-matching char i.e s i.e 6)

Here, the no. of chars matching = k = 2

$d_1 \Rightarrow 6 - 2 = 4$ times shifted.

t(i) → shift val of char

T: A E R

P: B A R B E R

shift val of A

$d_1 = t(A) - k$

$= 4 - 2 = 2$ times shift.

Good Suffix Shift: (d_2)

→ is created based on pattern.

ex: pattern is A B C B A B.

K	Pattern	d_2
1	A B C B A B (2 1)	2
2	A B C B A B (4 3 2 1)	4
3	A B C B A B (x ✓)	4
4	A B C B A B (x x ✓)	4

How?

assume: T: X B A B

P: D B C B A B

DBCBAB is it there? NO.

is AB there? NO.

∴ shift it by length of pattern.

T: X B A B

P: A B C B A B

BAB X is not there

AB ✓ is there

∴ Wkt length of AB = 4, shift 4 times.

eg: TORONTO

K	Pattern	d_2
1	TORONTO	3
2	TORONTO	5
3	TORONTO	5
4	TORONTO	5
5	TORONTO	5
6	TORONTO	5

eg: 01010

K	Pattern	d_2
1	01010	4
2	01010	4 ???How?
3	01010	2
4	01010	2

Q) Find the occurance of BAOBAB in BESS_KNEW_ABOUT_BAOBAB

⇒ Good suffix table:

K	Pattern	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB	5
5	BAOBAB	5

→ we cannot consider 1st occ of B as the leftside of it is 'O'. So, we consider 'B' 2nd occ.

⇒ Bad symbol shift:

T : BESS_KNEW_ABOUT_BAOBAB
P : BAOBAB

$d_1 = t(c) - k = 6 - 0 = 6$
$d_2 = k(0) = 0$. max = 6

$d_1 = t(c) - k = 6 - 2 = 4$
$d_2 = k(2) = 5$. max = 5

BAOBAB
BAOBAB
BAOBAB

$d_1 = t(-) - k$
$= 6 - 1 = 5$
$d_2 = k(1) = 2$
max = 5

Shift table:

A = 1
B = 2
O = 3
others = 6

always take max{d_1, d_2}

Q) TOMATO in VENDOR_SELLS_TOMATOES

⇒

K	pattern	d_2
1	TOMATO	6 ✓ 4 ✗ → how 6??
2	TOMATO	4
3	TOMATO	4
4		4
5		4
6		4

⇒ VENDOR_SELLS_TOMATOES

TOMATO

TOMATO

TOMATO

$d_1 = t(A) - 0$

$= 6 - 0 = 6$

$d_2 = 6$ max = 6

TOMATO

$d_1 = t(T) - 0$

$= 1 - 0 = 1$

Algorithm:

Step-1 : for a given pattern, construct bad symbol shift table.

Step-2 : using pattern, construct good-suffix shift table.

Step-3 : align the pattern against beginning of text.

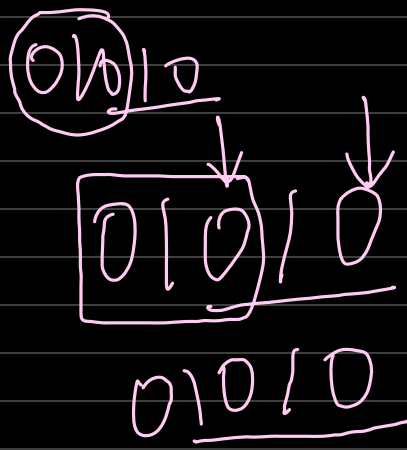
Step-4 : repeat the foll steps until matches / beyond the text:

* Start with last char, compare each char.
* if all 'm' chars are matched ⇒ STOP.
* if mismatch : $t(c) - k$ where k ⇒ no. of correct chars & $t(c)$ ⇒ shift val of char not matched.
* compare d_1, d_2 & shift the pattern $\max\{d_1, d_2\}$ times.

$$d = \begin{cases} d_1 & \text{if } k = 0 \\ \max\{d_1, d_2\} & \text{if } k > 0 \end{cases} \quad \text{where } d_1 = \max\{t_1(c) - k, 1\}.$$

01010

k=1 4
k=2 4
k=3 2
4 2



BAOBAB

k=1 2
=2 5