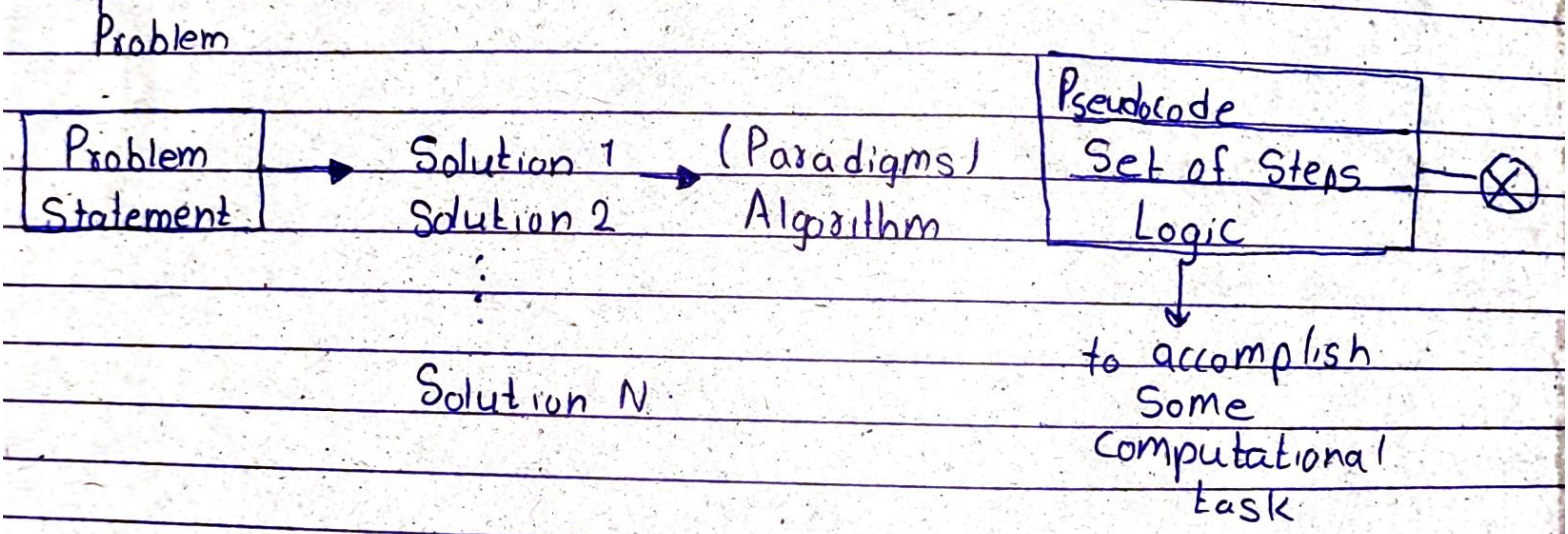
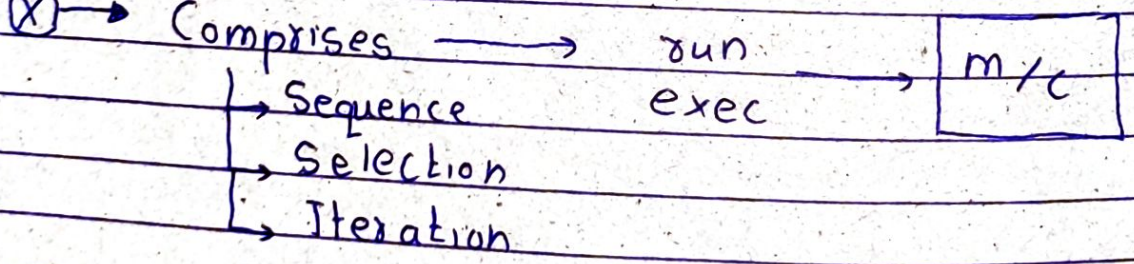
25-01-2025
Saturday
2025

# Advanced Algorithms

* Algorithm:
An algorithm is any well defined computational procedure that takes some value, or set of values, as input and produces some values, or set of values, as output.
An algorithm is thus a sequence of computational steps that transform the input into the output.




* Algorithm Analysis Tools.

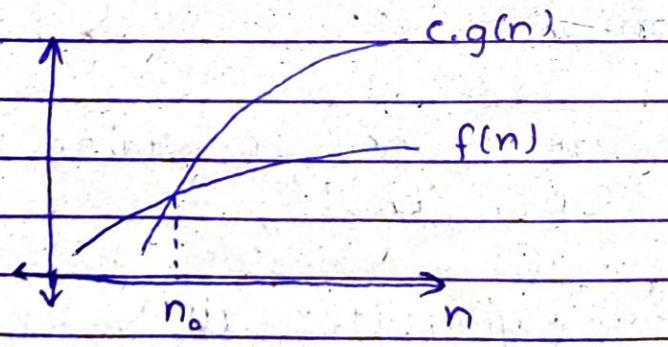
* Asymptotic Analysis (Notations)
In asymptotic analysis, we evaluate the performance of an algorithm in terms of input size.
We calculate, order of growth of time taken (or space) by an algorithm in terms of input size

- Algorithm comparison should be independent of machine, So, for this purpose we count the number of basic steps in an algorithm

- Asymptotic Notations are mathematical tools used to analyze the performance of algorithm by understanding how their efficiency changes as the input size grows.

* Big-O Notation:
- It represents the upper bound of the running time of an algorithm.
- ∴ it gives the worst-case complexity of an algorithm

$F(n) = O(g(n))$ iff there exists positive constants C and $n_0$ such that $F(n) \leq C \cdot g(n)$ for all $n \geq n_0$



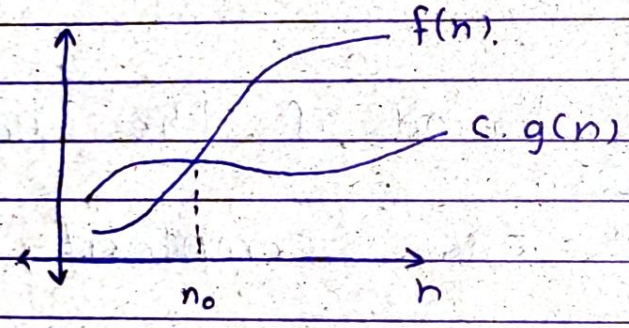

Q1 $f(n) = 10n^2 + 5n + 6$
$f(n) \leq 21 n^2$
$f(n) = O(n^2)$

Q2. $f(n) = 6 \cdot 2^n + 2n^2$
$f(n) \leq 8 \, 2^n$
$\Rightarrow f(n) = O(2^n)$

* Omega Notation ( Ω - Notation):

- Omega Notation represents the lower bound of the running time of an algorithm.
- Thus, it provides the best case complexity of an algorithm

- $F(n) = \Omega(g(n))$ iff there exist positive constants c and $n_0$ such that $F(n) \geq c.g(n)$ for all $n \geq n_0$

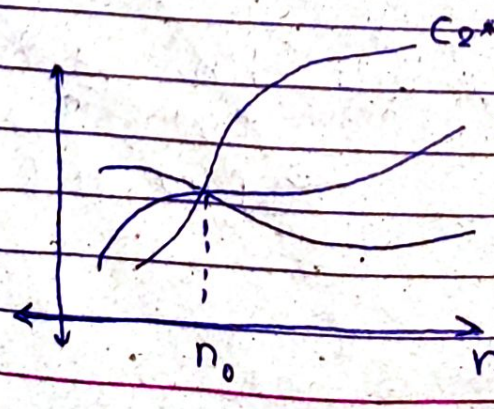



* Theta Notation (Θ-Notation):

- It encloses the function from above and below.
- It is used for analyzing the average-case complexity of an algorithm.

$F(n) = \Theta(g(n))$ iff there exist constants $c1, c2 > 0$ and $n_0$ such that.

$c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all $n > n_0$.



$f(n) = 10n^2 + 5n + 6$

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

Theta notation of an expression is to drop low-order terms and ignoring leading constants.

* Important Series

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

2) $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$

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

4) $\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$

5) $\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$

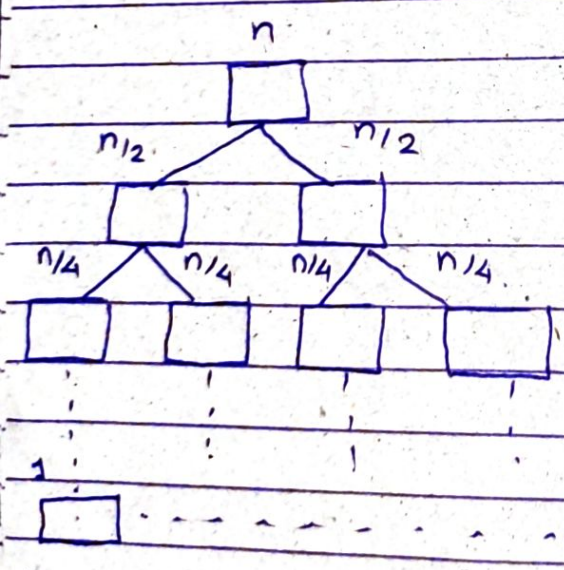
* If $0 < A < 1$ then

$$\sum_{i=0}^{N} A^i = \frac{1}{1 - A}$$

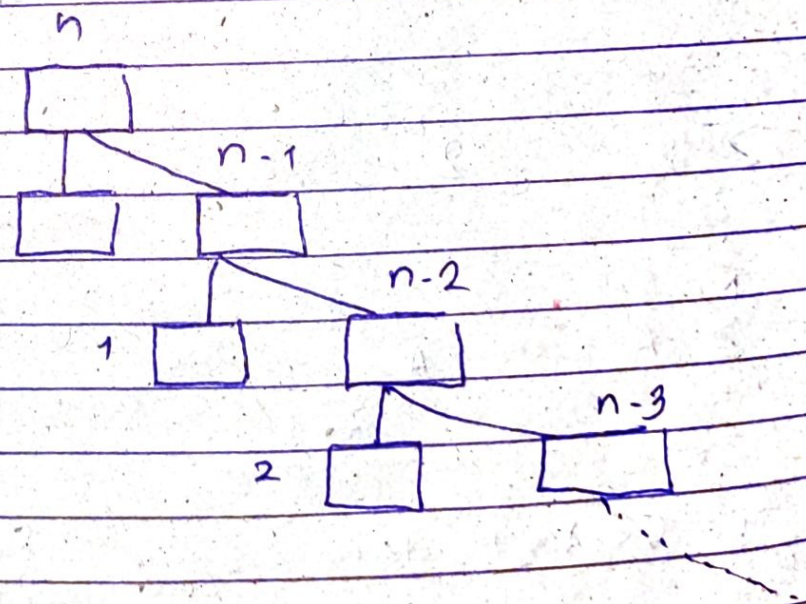
* Divide and Conquer Approach:

- The problems divide themselves into several subproblems recursively, and then combine these solutions to create a solution to the original problem.
- The divide and conquer paradigm involves three steps at each level of the recursion.
  - Divide : Problem → Subproblems.
  - Conquer : Solving Subproblems recursively.
  - Combine : Combines Solutions of subproblems.

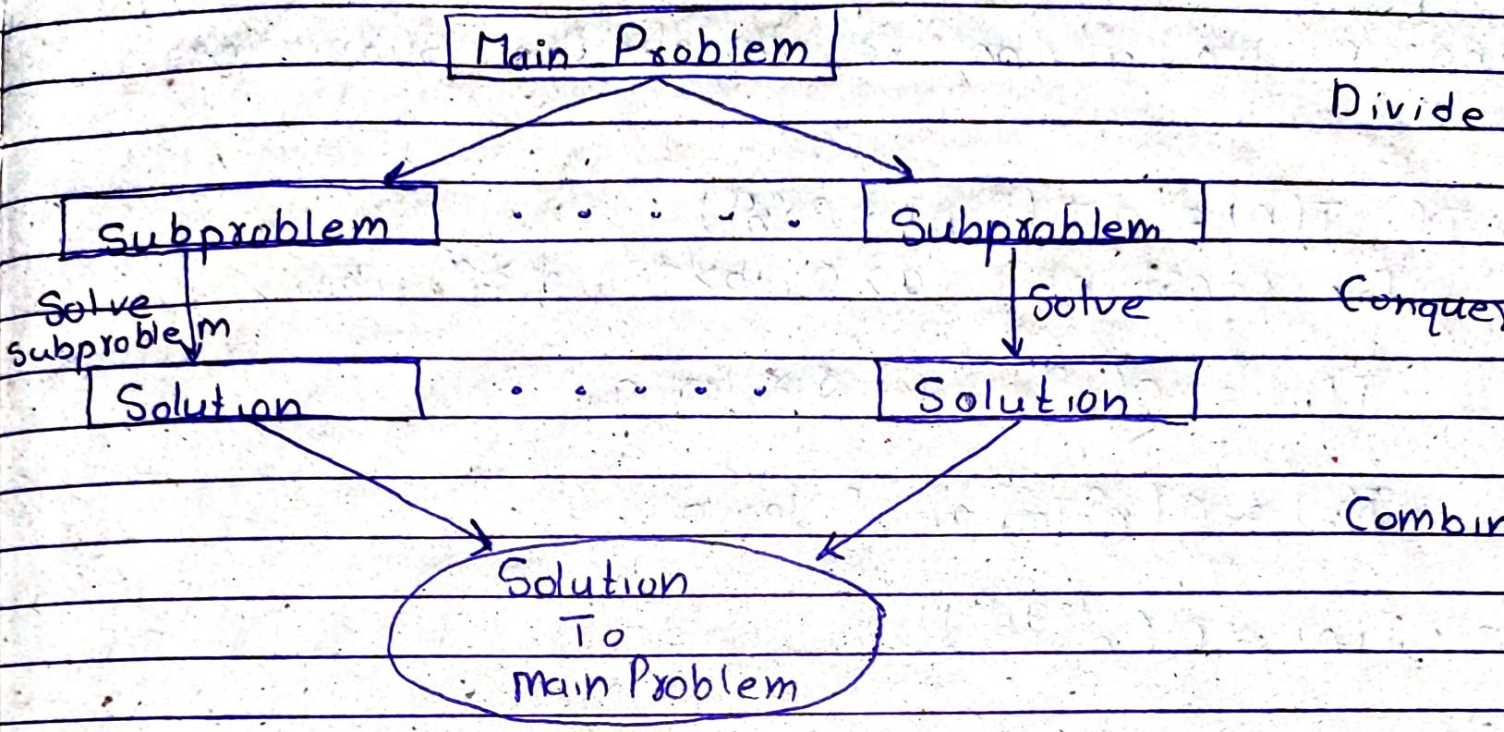
- When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation, which describes the overall running time on a problem of size n in terms of running time on smaller inputs.




Balanced.




Skew.




* The Master's theorem :
The master method provides a "cookbook" method for solving recurrences of the form

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are three cases to solve recurrence relations.

The expression describes the running time of an algorithm that divides a problem of size $n$ into 'a' subproblems of size $n/b$. The 'a' subproblems are solved recursively, each in time $T(n/b)$. The function $f(n)$ encompasses the cost of dividing the problem

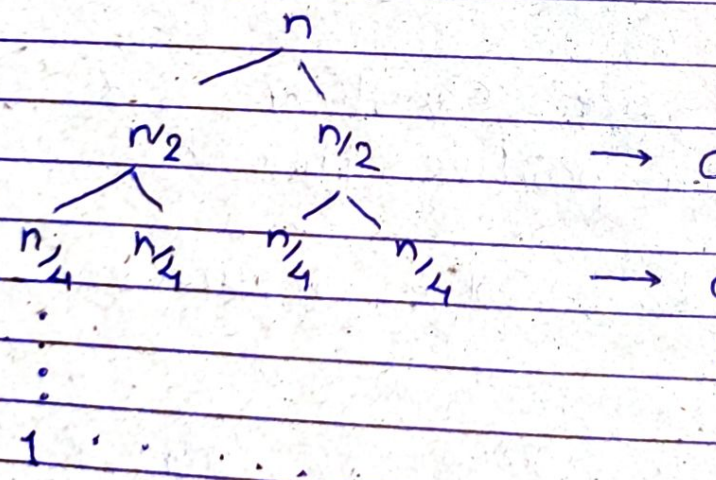
and combining the results of the subproblems.

If $T(n) = aT(n/b) + O(n^d)$
for constants $a > 0$, $b > 1$, $d \geq 0$ then

- $T(n) = O(n^d)$ if $d > \log_b a$.
- $T(n) = O(n^d \log n)$ if $\log_b a$.
- $T(n) = O(n^{\log_b a})$.

* Recursive-tree method for Solving recurrences.

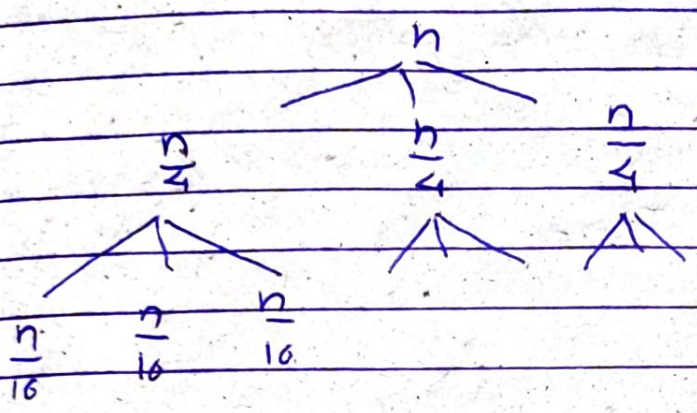
Q1) $T(n) = 2T(n/2) + Cn$




It will go till trivial level equal to height of tree.
Thus $T(n)$ depends on height of tree.

$$T(n) = Cn + Cn + \ldots\ldots$$
$$= Cn(\log_2 n)$$
$$= O(n \log n)$$

Q2) $T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$.




$$T(n) = cn^2 + \frac{3}{16} cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots \left(\frac{3}{16}\right)^k cn^2$$

$$= cn^2 \left[ 1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots \right]$$

It is in GP $\rightarrow \frac{1}{1-r}$

$$T(n) = cn^2 \left[ \frac{1}{1 - \frac{3}{16}} \right]$$

$$= cn^2 \frac{16}{13}$$

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

Q State whether true or false

1) $2^{n+1} = O(2^n)$

$2^{n+1} = 2^n \times 2^1$

$= 2 \times 2^n$

$= O(2^n) \Rightarrow$ True.

2) $2^{2n} = O(2^n)$

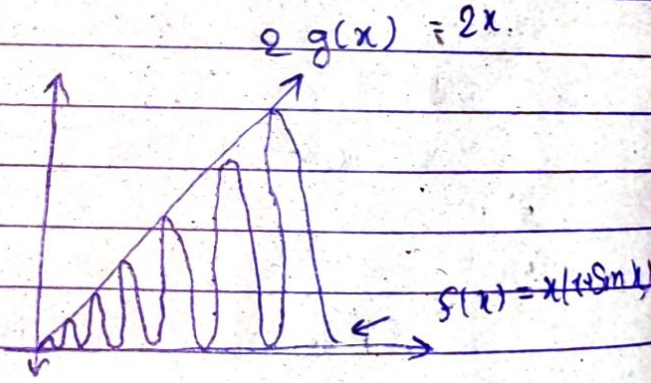
$2^{2n} = (2^n)^2 \neq O(2^n)$

$\Rightarrow$ False.

Q $f(x) = x(1+\sin x)$

$g(x) = x$.

Check if $f(x) = O(g(x))$




$\sin x$ oscillates between $(-1, 1)$.

$\Rightarrow (1+\sin x)$ oscillates between $(0, 2)$

Thus $f(x) = x(1+\sin(x))$ oscillates between 0 and $2x$

$\Rightarrow f(x) \leq 2x$

$f(x) \leq 2g(x)$

As $f(x) \leq C\,g(x)$ for $x \geq 0$

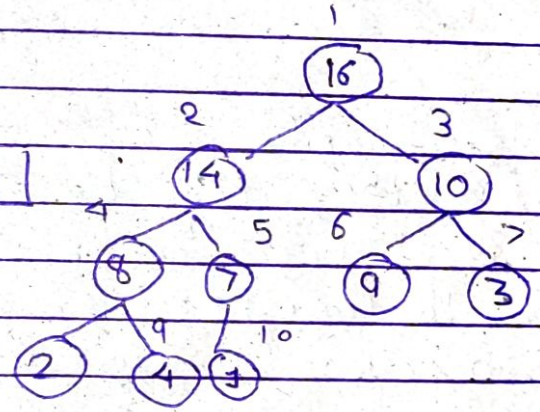
$\Rightarrow f(x) = O(g(x))$.

* Heap Structure (using Array) and Heap Sort.

- The (binary) heap data structure is an array object that we can view as a nearly complete binary tree.
- Each node of the tree corresponds to an element of the array.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

An array A that represents a heap has the root of tree as A[1] (the first element in 1-based indexing

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |




Parent (i) = $\lfloor i/2 \rfloor$

Left (i) = 2i

Right (i) = 2i+1

* There are two kinds of binary heaps: max-heaps and min-heaps.
- In a max-heap, the max-heap property is that for every node i other than the root

A[Parent(i)] ≥ A[i]

that is, the value of a node is at most the value of its parent

Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains value no larger than that contained at the node itself.

* A min-heap follows min-heap property such that for every node, other than the root

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

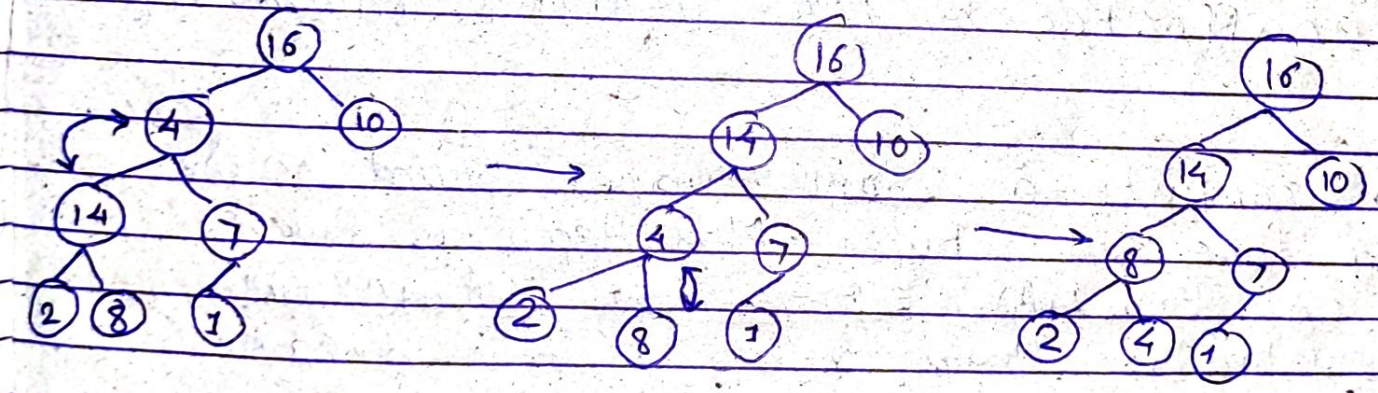
- The smallest element in a min-heap is at the root.

- The Max-heapify procedure, which runs in O(logn) time, is the key to maintaining the max-heap property.
- The build-Max-heap procedure, which run in linear time, produces a max heap from an unordered input array
- The heapSort procedure, which runs in O(nlogn) time, Sorts an array in place.

* Maintaining the heap property:
In order to maintain the max-heap property, we call the procedure Max-Heapify.
Its inputs are an array A and an index i into the array.

```
MAX_HEAPIFY (A,i)
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-Size and A[l] > A[i]
        largest = l.
    else largest = i
    if r ≤ A.heap-Size and A[r] > A[largest]
        largest = r.
    if largest ≠ i
        A[i] ⟷ A[largest]
        MAX-Heapfy(A, largest)
```

At Each Step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined. If A[i] is not largest, then it is swapped And Max_Heapfy is called at the Child Subtree




The time complexity is for a Subtree of Size n rooted at a given node i is O(1) to find largest and Swap and plus time complexity for recursive calls on child nodes.
The children subtree each can have at most $2n/3$ nodes

$\Rightarrow T(n) \leq T(2n/3) + O(1)$

which when Solved gives $T(n) = O(\log h)$ or $T(n) = h$

h = heigh.

* Building a heap:
We can use procedure MAX-HEAPIFY in a bottom up approach (manner) to convert an array A[1...n], into a max-heap.

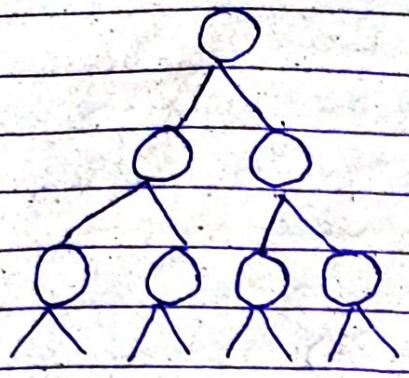
Input: Array of unsorted numbers A[]
Total no of elements n.

```
procedure Buildheap(n);
    for i = n down to 1 do
        MAX-HEAPIFY (A,i)
    end for
```

we can also start from $(n/2 - 1)$ down to 1 to skip all leaf nodes, as the will be unaffected by MAX HEAPIFY (downheap).

We can compute a simple upper bound on the running time of build-Max-Heap as follows. Each call to MAX HEAPIFY costs O(logn) time and build-Max-Heap makes O(n) such calls. Thus, the running complexity is O(nlogn). This is upper bound but we can find more tighter bound.

- Complexity of build-heap seems to be nlogn but in reality it is O(n)




$h = \log_2 n$

The total time complexity of building a heap can be derived by analyzing the work done at each level of the heap.

1. A heap is a complete binary tree, so its height is $h = \log_2 n$, where n is number of nodes.
2. At height i, there are $2^i$ nodes
3. Each node at height i may need to "heapify" through a maximum of h-i levels to restore heap property

⇒ Total Complexity.

$$= \sum_{i=0}^{h} 2^i * O(h-i)$$

$$= O\left( h \sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} i 2^i \right)$$

$$= O\left( h\left(2^{h+1} - 1\right) - \left[ (h+1)2^{h+1} - 2^{h+2} + 2 \right] \right)$$

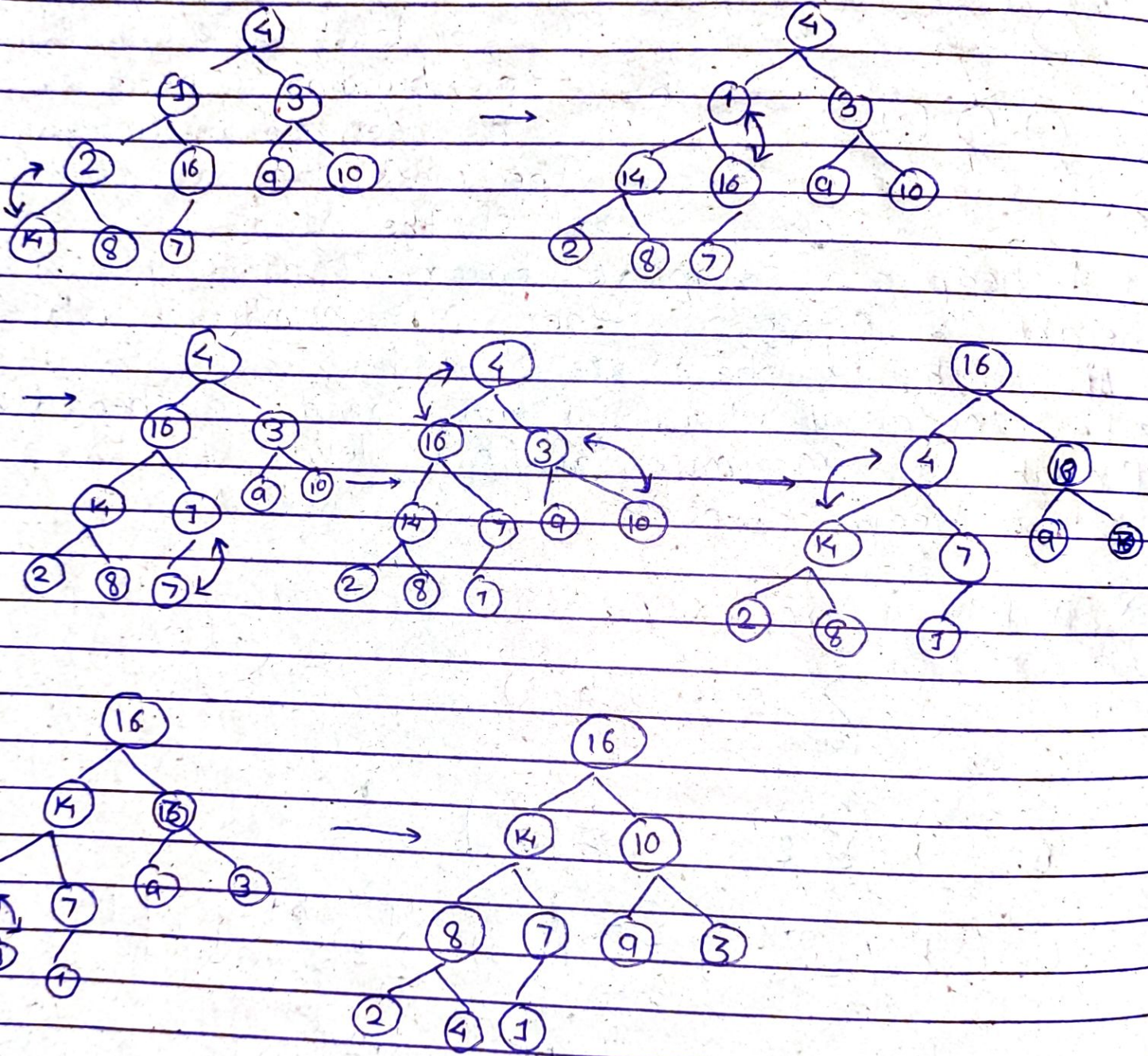
$$= O\left( h 2^{h+1} - h - (h+1)2^{h+1} + 2^{h+2} - 2 \right)$$

$$= O\left( -2^{h+1} + 2^{h+2} - 2 - h \right)$$

$$= O(2^h) = O(2^{\log_2 n})$$

$$= O(n)$$

A[] = | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |



* Heap Sort Algorithm

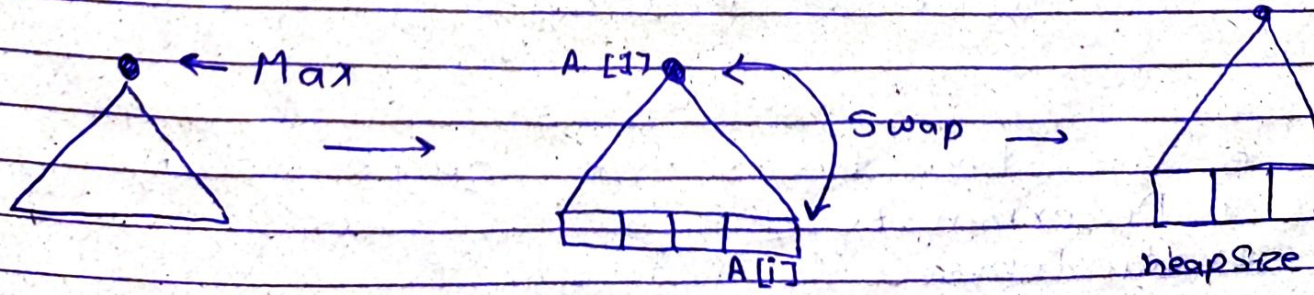
The heap Sort algorithm starts by using Build-Max-heap to build max heap on the input array A[n], Since the maximum element is present at root A[1] (1 indexed array).

So we put it at correct position A[n].
Now we discard (ignore) element $n^{th}$ from heap and by decrementing heap Size
Now we restore max heap property with MAX-HEAPIFY(A, 1) which gives max heap for A[1...n-1].
Next we follow Same process of Swapping from size n-1 down to 2.

```
HeapSort(A, n)
    BuildHeap(A, n)
    for i := n down to 2 do
        A[1] ↔ A[i];
        heap-Size[A] = i-1
        MAX-HAEPIFY(A, 1)
    end for
end
```

The heapSort procedure takes time O(n log n)
Since call to build-Max heap takes time O(n) and each of the n-1 calls to MAX-HEAPIFY takes O(lg n).

$$\Rightarrow T(n) = O(n + n\log n)$$
$$= O(n\log n)$$



*Quick Sort:

Quick Sort is a sorting algorithm based on the divide and conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the Sorted array.

Divide: Partition (rearrange) array A[P..r] into two Subarrays, A[P..q-1] and A[q+1..r] such that each element of A[P..q-1] is less than or equal to A[q] and each element in A[q+1, r] is greater than or equal to A[q]. where q is pivot.

Conquer: Sort the two Subarrays A[P..q-1] and A[q+1..r] by recursive calls to quick Sort.

Combine: No work is needed in combine as Subarrays are already Sorted.

```
QuickSort (A, P, r)
    if P < r
        q := partition (A, P, r)
        QuickSort (A, P, q)
        QuickSort (A, q+1, r)
```

* There are two types of quick Sort
  1) Deterministic
  2) Randomised.

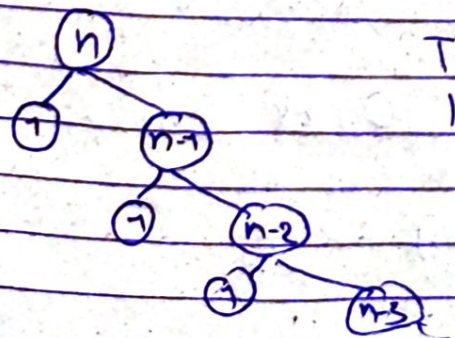
In deterministic quick Sort, pivot is Selected as left most or rightmost element.

```
Partition (A, p, r)
    x = A[p]
    i = p - 1
    j = r + 1
    loop
        repeat j = j - 1 until A[j] ≤ x
        repeat i = i + 1 until A[i] ≥ x
        if i < j then A[i] ↔ A[j]
        else partition = j;
             exit
    end loop.
```

* The running time of quicksort depends on whether the partition is balanced or unbalanced.
* If partition is balanced, the algorithm runs fast. Howevever if unbalanced then it reaches worst complexity.

Worst Case : $T(n) = T(n-1) + T(0) + O(n)$

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




Thus cost incurred at each level of recursion add up to give $O(n^2)$

- If partition is balanced the height of tree becoms $\log n$ hence complexity is $O(n \log n)$

$T(n) = 2T(n/2) + O(n)$ → best Case

$\Rightarrow T(n) = (n \log n)$

* Randomized Quick Sort:
In this the pivot element is selected randomly each time instead of selecting leftmost or right-most element.

- As pivot is randomly selected, we ~~ensure~~ expect split of array to be reasonably balanced.

Random (a, b)
return random number between 'a' and 'b'.

```
Random_Partition (A, P, r)
    i = random (P, r).
    P[P] ↔ A[i];
    Partition (A, P, r)
end.
```

* Complexity Analysis of randomized Quick Sort:

$$T(n) = \frac{1}{n}\left( T(1) + T(n-1) + \sum_{q=1}^{n-1} T(q) + T(n-q) \right) + O(n)$$

Case 1. Partition

$$T(n) \leq \frac{1}{n}\left(\sum_{q=1}^{n-1} T(q) + T(n-q)\right) + \Theta(n)$$

$$\leq \frac{1}{n}\sum_{k=1}^{n-1} 2T(k) + \Theta(n)$$

$$\leq \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n)$$

There are 2 worst cases when the pivot is first Smallest and Pivot is Second Smallest

Tool $\sum_{k=1}^{n-1} k \log k \leq \log n\left(\frac{n^2}{2}\right) + \left(\frac{n^2}{8}\right)$

Assume by induction $T(n) \leq an\log n + b$ where $a, b$ are constants.

Hypothesis: If $k < n$ if then $T(k) \leq ak\log k + b$.

$$T(n) \leq \frac{2}{n}\sum_{k=1}^{n-1} (ak\log k + b) + \Theta(n)$$

$$\leq a \cdot n\log n - \frac{an}{4} + 2b + \Theta n$$

$$\leq an\log n + b + \Theta(n) + b - \frac{an}{4}$$

(as $a >> b$ we can ignore few term

$$T(n) \leq \Theta(nb\,gn)$$

Tool: $\sum_{k=1}^{n-1} k \log k \leq \log n \left(\frac{n^2}{2}\right) - \left(\frac{n^2}{8}\right)$

Proof: $\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$.

max = $\lceil n/2 \rceil$ max = n.

$\leq (\log n - 1) \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log n \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k$.

$\leq \log n \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$.

$\leq \log n \sum_{k=1}^{n-1} k - \sum_{n=1}^{\lceil n/2 \rceil - 1} k$

$\leq \log n \left(\frac{n}{2}\right) - \left(\frac{n+2}{2}\right)$ $\left(\because \sum_{i=1}^{n} i = \left(\frac{n+1}{2}\right)\right.$

$\leq \log n \frac{n!}{(n-2)!\,2!} - \frac{x!}{(x-2)!\,2!}$ $(x = n/2.)$

$\leq \log n \cdot \frac{n(n-1)}{2!} - \frac{x(x-1)}{2!}$ $\frac{\frac{n}{2}\left(\frac{n}{2}-1\right)}{2} = \frac{n^2-2n}{8}$

$\leq \log n \frac{n^2-n}{2} - \frac{n^2-2n}{8}$

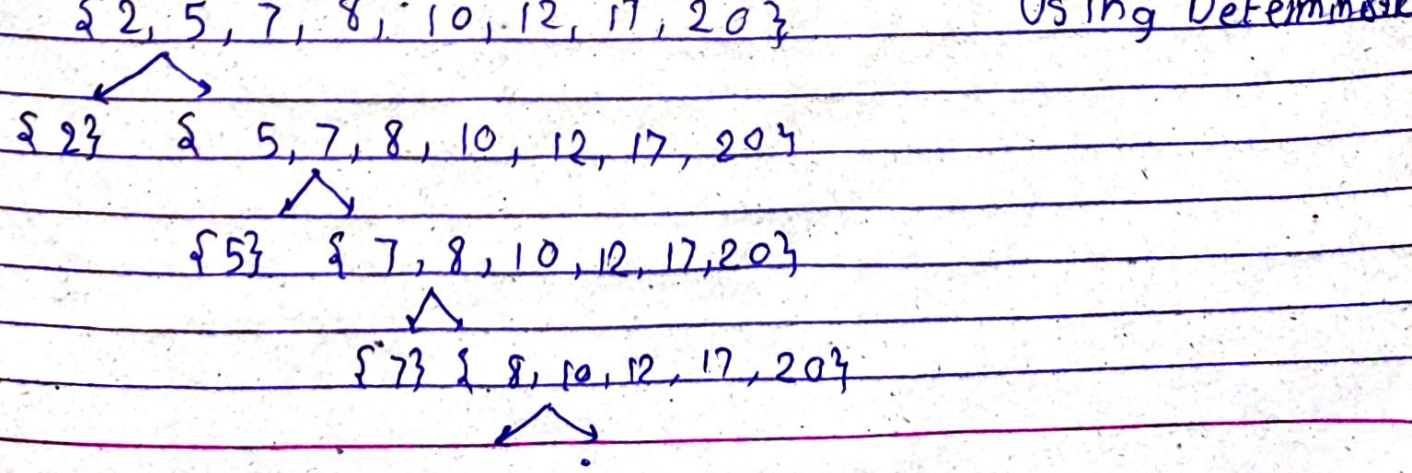
$\leq \log n \left(\frac{n^2}{2}\right) - \log n \left(\frac{n}{2}\right) - \frac{n^2}{8} + \frac{n}{4}$

$\leq \log n \left(\frac{n^2}{2}\right) - \frac{n^2}{8}$.

* Advantages of Randomized Quick Sort:

- Avoids Worst case performance on Sorted inputs
  - If deterministic Quick Sort can suffer $O(n^2)$ performance when given already Sorted or nearly Sorted input.
  - Randomized Quick Sort selects pivots randomly, making it highly unlikely to consistently choos the worst case partitioning.
- Ensures Expected $O(n \log n)$ Performance.
  - The randomization ensures that, on average, Quick sort runs in $O(n \log n)$ time, regardless of the input distribution.
- In-place Sorting:
  - Requires not extra space.
- Fast in Practice
  - Compared to mergeSort, Quick Sort has better cache efficiency.

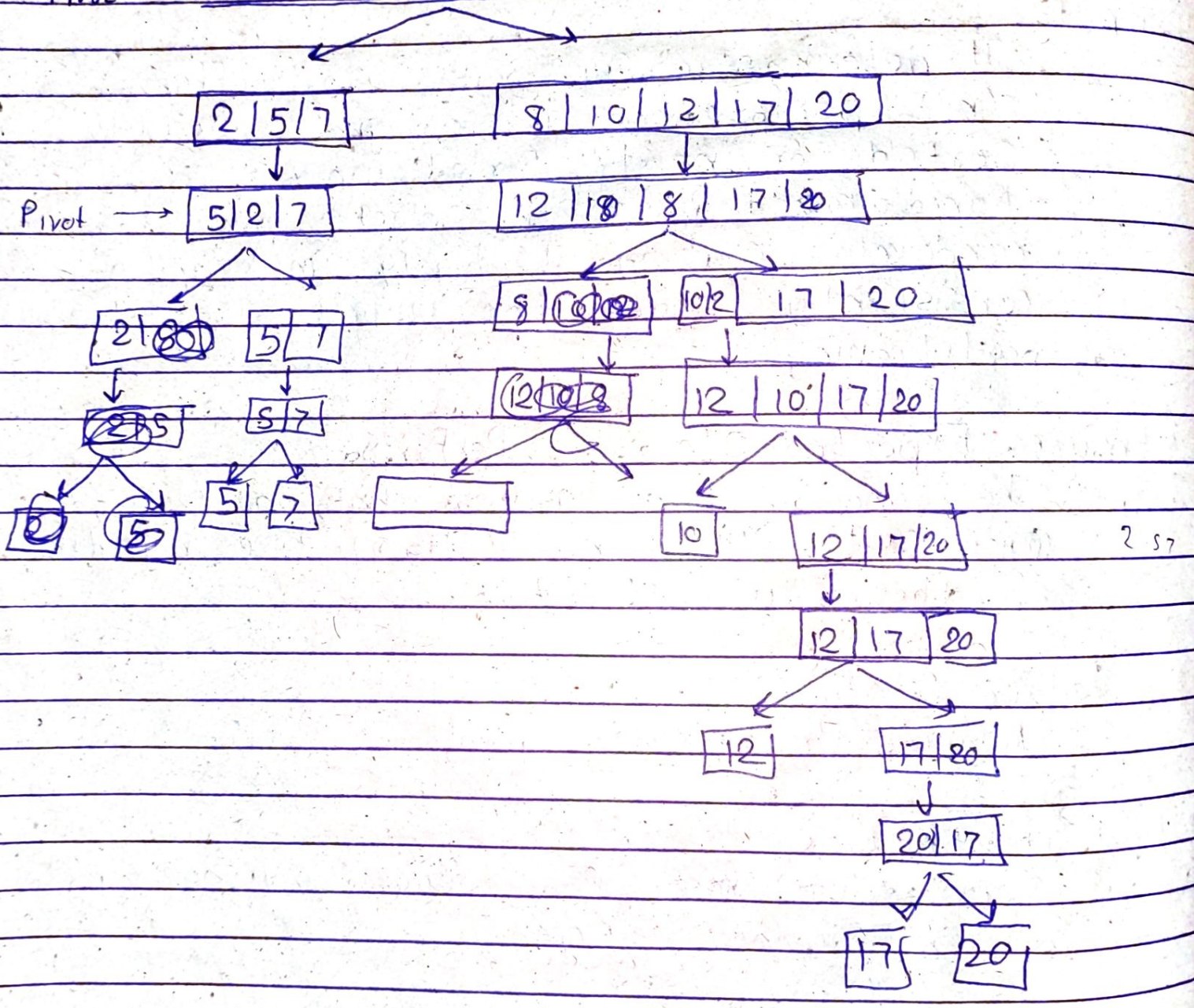
Ex. $\{2, 5, 7, 8, 10, 12, 17, 20\}$ Using Determinste



Ex. | 2 | 5 | 7 | 8 | 10 | 12 | 17 | 20 |

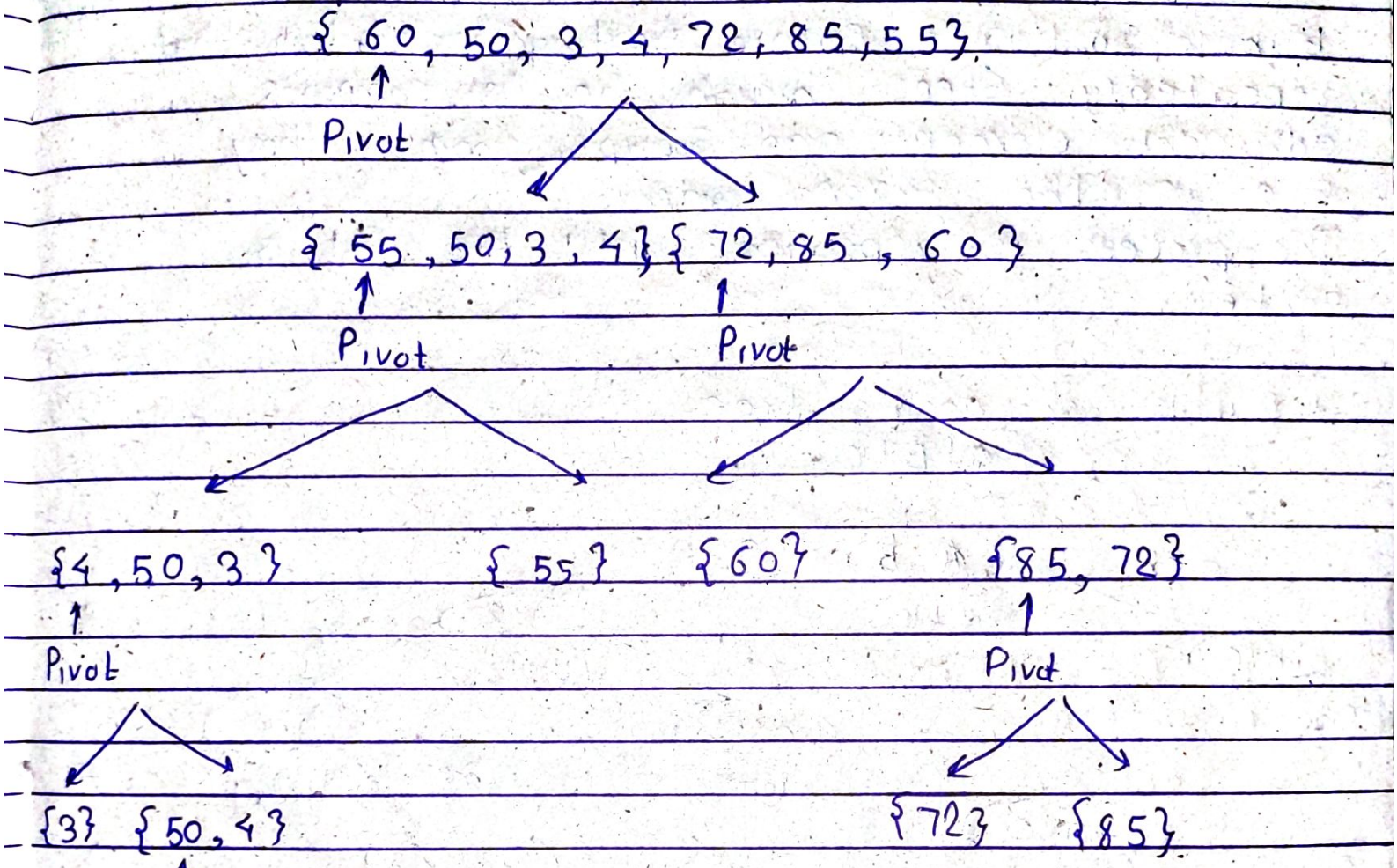
↓ (randomly Select & Swap with 0 index.

Pivot → | 8 | 5 | 7 | 2 | 10 | 12 | 17 | 20 |



Q. | 60 | 50 | 3 | 4 | 72 | 85 | 55 |

Sorting Using deterministic Quick Sort.



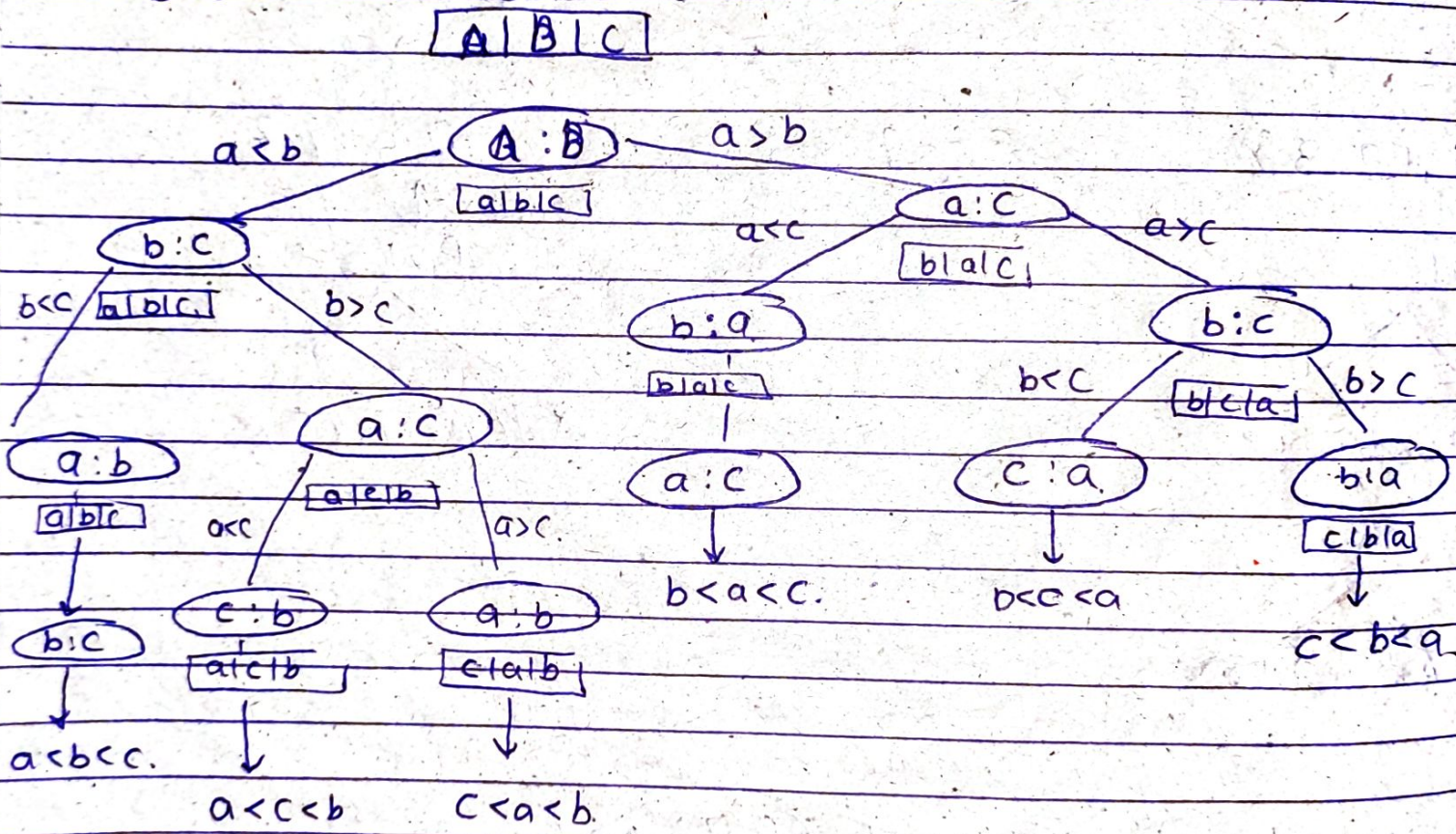

Sorted: {3, 4, 50, 55, 60, 72, 85}

* Bubble Sort:

```
for i = 1 to n-1 do
    for j = 1 to n-1-1 do
        if A[j] > A[j+1] then
            A[j] ↔ A[j+1]
        end if
    end for
end for
```

Bubble Sort is a Simple Sorting algorithm that repeatedly steps through the list, compares adjacent element, and swaps them if they are in the wrong order.
The process is repeated until the list is sorted.

* Bubble Sort decision tree.




The total number of nodes are n!
If there are n elements then there are n! leaves in the decision tree.

Worst Case behavior = longest Path = most number of comparables

A binary tree of height h can have at most $\leq 2^h$ leaves.

$$2^h \geq n!$$

$$h \log_2(2) \geq \log_2 n!$$

$$\Rightarrow h \geq \log n!$$

By Stering formula $n! \geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$.

$$\Rightarrow h \geq \frac{1}{2} \log_2(2\pi n) + n \log\left(\frac{n}{e}\right)$$

$h = \Omega(n \log n)$ lower Bound.

* Order Statistics / Selection prom

- The selection problem refers to finding the ith smallest element in a unordered array. This element is also known as the ith order statistic.
- There are two methods
  - Deterministic
  - Randomised.

* Brute Force : (Finding min)

```
Function min
    min = 1;
    For i = 2 to n do
        if A[min] > A[i] then
            min = i;
        end if
    end for
    min = A[min]
```

Brute For (Finding $k^{th}$ Smallest)

1) Sort array and the return $k^{th}$ element
   ↳ Complexity = O(n log n)

* Randomized Selection:

- The randomized Selection algorithm is a probabilistic approach to finding the ith Smallest element in an unsorted array.
  It is based on Quick Sort
- But instead of using a fixed pivot, it randomly selects a pivot.
- It can return $i^{th}$ Smallest number without Sorting entire array.
- We recursively partition only one half where the i-th element might be located.

```
Random_Select (P, r, i)
    if p = r then return p;
    else
        q = Randomized_partition (p, r)
        k = q - p + 1.    // Find no of element in left Subproblem
        if i ≤ k then
            return Random_Select (p, q, i)
        else
            return Random_Select (q+1, r, i-k)
        end if
    end if
```

Initial p=1, r, n.

ex. Find 5th Smallest ⇒ Random Select (1, n, 5)

* Overall Complexity = O(n)

Q. {30, 25, 3, 40, 43, 33, 6, 12, 31, 18, 1}

Find 5th Smallest (use 1st element as pivot)

{30, 25, 3, 40, 43, 33, 6, 12, 3, 18, 1}

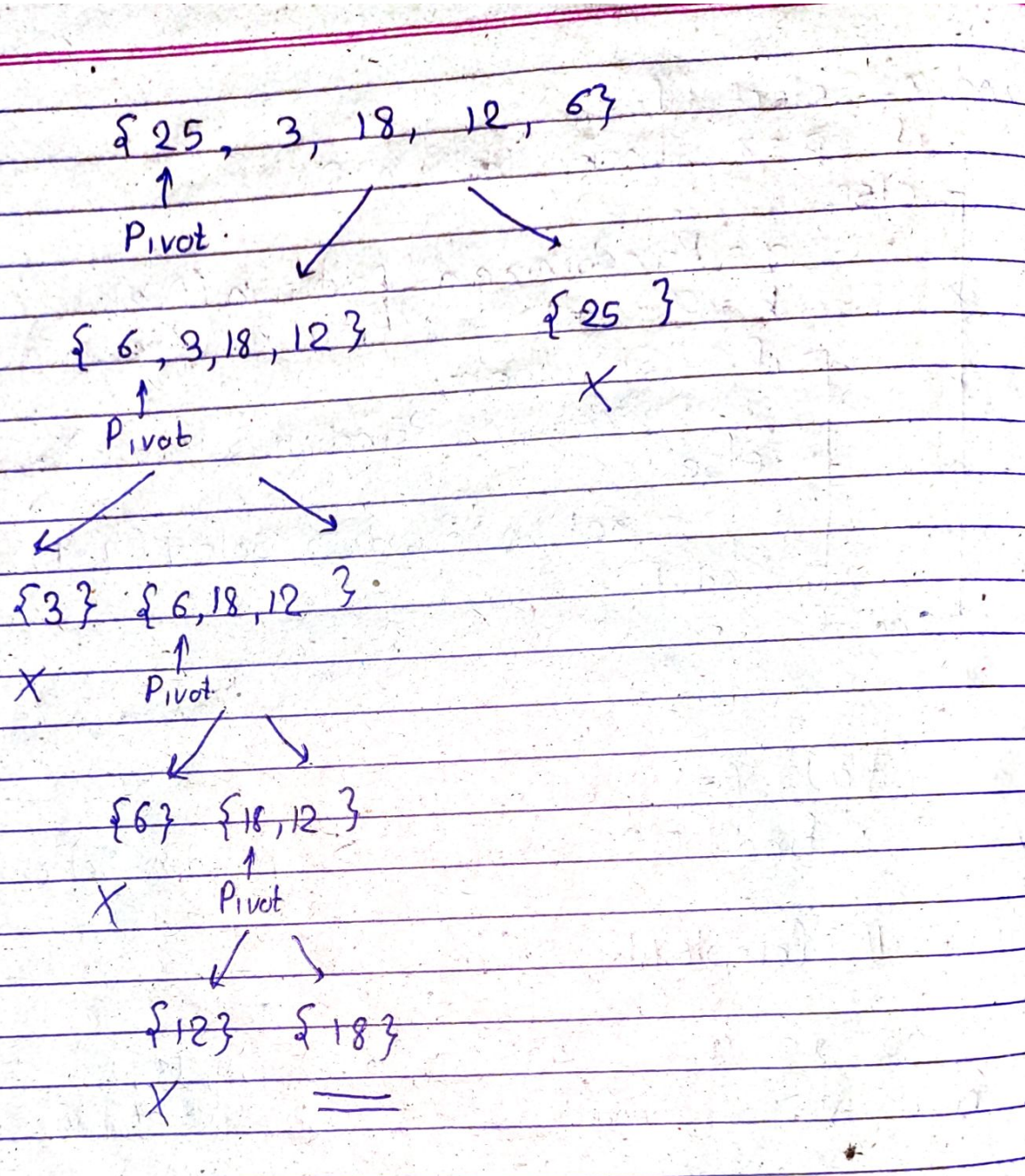
↑ Pivot

{1, 25, 3, 18, 3, 12, 6}  {33, 43, 40, 30}

↑ Pivot  X

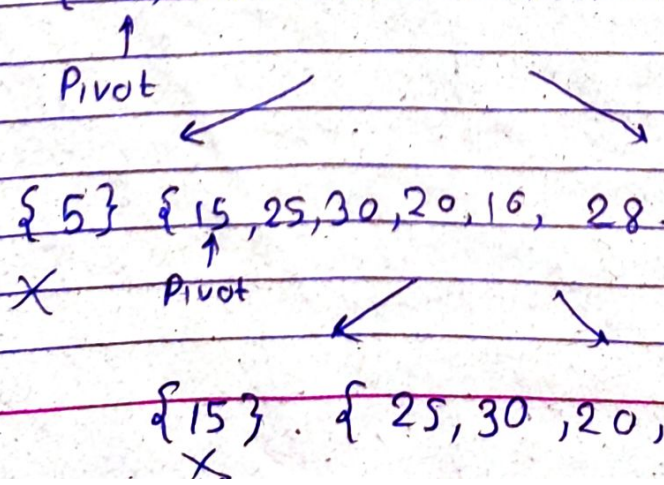
{1}  {25, 3, 18, 3, 12, 6}

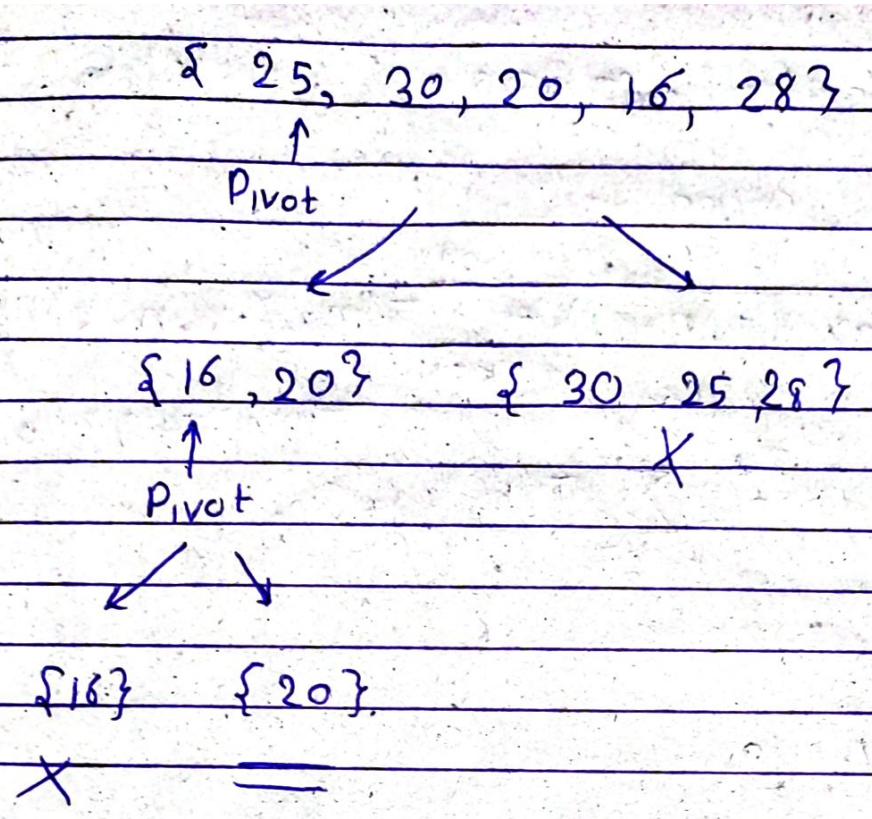
X  ↑ Pivot




5th Smallest = 18.

Q. {5, 15, 25, 30, 20, 16, 28} Find 4th Smallest






4th Smallest = 20

- The Randomised Selection follows a recursive approach to efficiently find th ith smallest element
- If P == r, then there is only one element in left array, So return A[P].
- Else a random pivot is selected and the array is partitioned.
- If number of elements in left Subproblem is greater than or equal to i we recursively call random Selection on left Subarray
- Else we recursively call selection on right Subarray.
- In case of right Subarray call we reduce i value to correctly match the req elements as per left Size.

## Average Complexity Analysis.

There are two cases

$$T(n) \leq \frac{1}{n}\left( T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k)) \right) + O(n)$$

As each term is twice. $n-2$. So we reduce Summation & multiply

$$T(n) \leq \frac{1}{n}\left( T(n-1) + 2\sum_{k=\lceil \frac{n}{2} \rceil} T(k) \right) + On.$$

ex. For $n = 10$

| k | n-k | k | n-k | |
|---|---|---|---|---|
| 1 | 9 | 9 | 1 | Same. |
| 2 | 8 | 8 | 2 | |
| 3 | 7 | 7 | 3 | |
| 4 | 6 | 6 | 4 | |
| 5 | 5 | 5 | 5 | |

$$\Rightarrow T(n) \leq \frac{O(n^2)}{n} + \frac{1}{n}\left( 2\sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} T(k) \right) + O(n)$$

Now $\frac{O(n^2)}{n} \geq O(n)$ So we ignore $\frac{O(n^2)}{n}$

$$T(n) \leq \frac{2}{n}\sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} T(k) + O(n)$$

Assume $T(n) \leq c \cdot n$ for some large constant $c$.

if $k < n$ is true.

$T(1) = \text{const.}$

$$T(n) \leq 2 \cdot \frac{c}{n} \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k + O(n)$$

$$T(n) \leq 2 \frac{c}{n} \left( \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \right) + O(n)$$

$$T(n) \leq 2 \frac{c}{n} \left( \frac{(n-1)\, n}{2} - \frac{\left(\frac{n}{2} - 1\right) \frac{n}{2}}{2} \right) \times O(n)$$

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

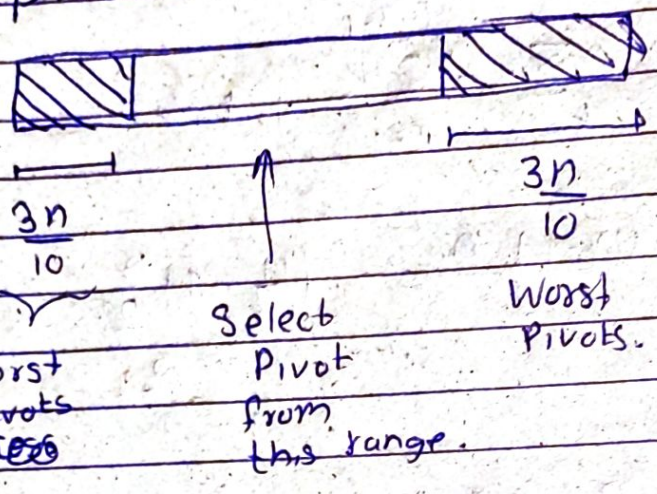
$$\leq \frac{c}{2} \left( \frac{n}{2} - 1 \right) + O(n)$$

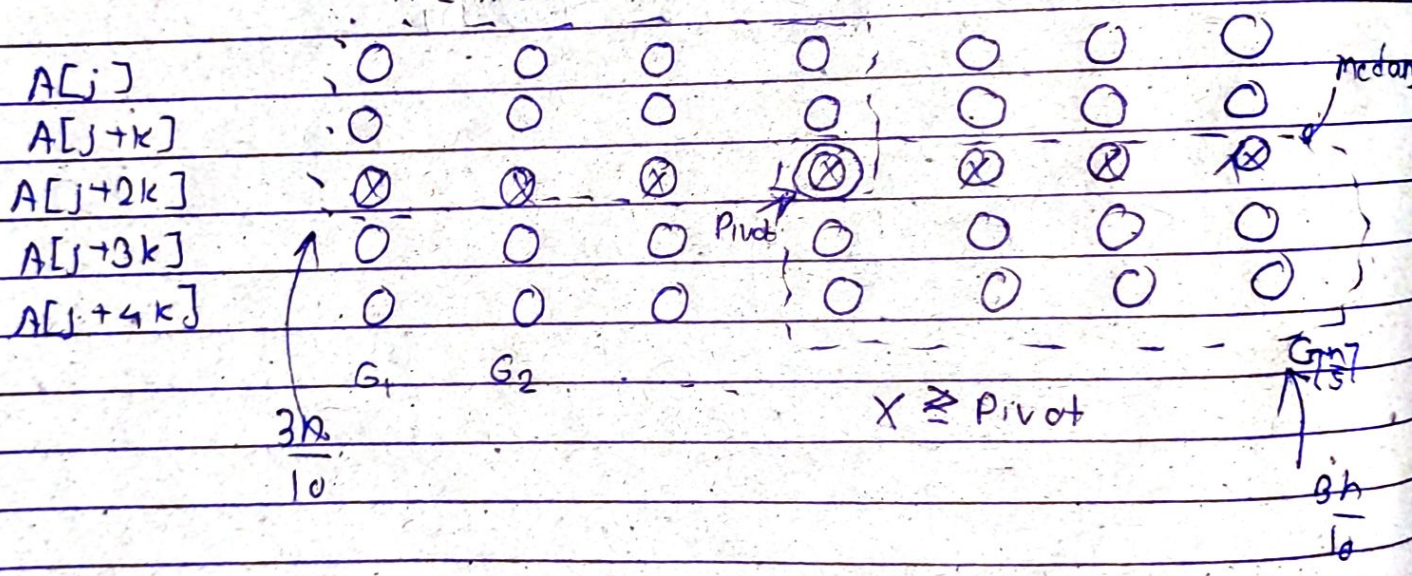
$$\leq \frac{3}{4} cn + O(n)$$

$$\leq O(n)$$

* Deterministic Approach of Selecting problem.
- The median of medians algorithm ensures a worst case $O(n)$ time complexity by selecting a good pivot deterministically.
- In this we distribute elements into five groups and find median for each groups.
- Then we find median of these medians and assign it as pivot

- This ensures good split by terminating worst pivots



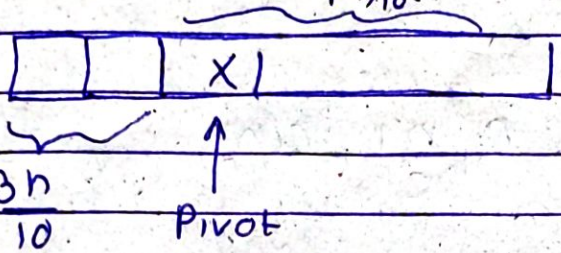




Step 1: Divide the nos in group of 5 & $k = \lceil \frac{n}{5} \rceil$ elements.
Step 2: Find the median of each group (using insertion)
Step 3: Recurse on $k = \lceil \frac{n}{5} \rceil$ median to find median pivot.
Step 4: Partition the array with median of medians (pivot) (call partition fun$^{n}$)
Step 5: if $i \leq q$ find the $i^{th}$ key on left side
else find $(i-q)^{th}$ key on right.

* This paradigm is called as prune and learn.

## Complexity Analysis

In worst case pivot will be




• So we need to partition right in 7n/10 size

$$\Rightarrow T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \Theta(n)$$

Lets assume $T(n) = O(n)$

(Using Induction Hypothesis)

$\Rightarrow T(n) \leq cn$ (for some constant c)

$T(k) \leq ck \quad \forall k < n.$

$$T(n) \leq \frac{cn}{5} + c \cdot \frac{7n}{10} + O(n)$$

$$\leq \frac{c9}{10} n + O(n)$$

$$\leq cn - \left(\frac{cn}{10} - \Theta(n)\right)$$

$$\leq cn$$

$\Rightarrow T(n) \leq cn$

$T(n) \leq O(n)$ By induction

2nd Way:

$T(n) = T\left(\frac{n}{c}\right) + O(n)$ where $c > 1$.

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

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

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

Each O has a constant.

$\leq O\left(n + \frac{n}{c} + \frac{n}{c^2} + \cdots \frac{n}{c^k}\right)$

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

$\leq O(n)$

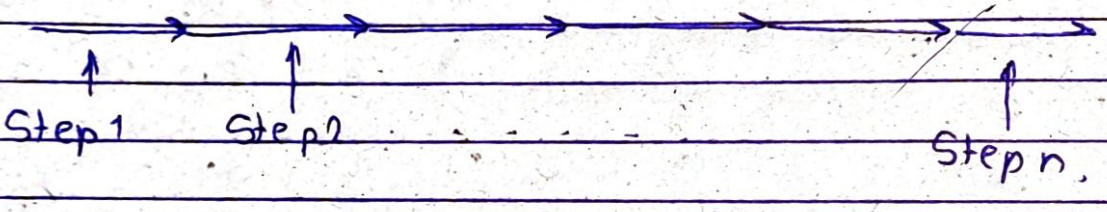
Start Step End

```
Procedure Insertion Sort (P, k, n)
    last = P+k;
    while last ≤ n do
        pos = last;
        while pos > P and A[Pos] < A[pos-k] do
            A[Pos] ↔ A[pos-k]
            Pos = Pos - k.
        end while
        last = last + k.
    end while
```

```
Function Select (P, r, i)
    if r-P+1 = 50 then
        insertion_Sort (P, i, r)
        return A[P+i-1];
    else
        k = ((r-P+1)+4) div 5;
        for j = 0 to k-1 do
            insertion_Sort (P+j, k, r);
        end for
        med_med = Select (P+2k, P+3k-1, k div 2);
        A[P] ↔ A[med_med];
        q = partition (P, r);
        if i ≤ q-P+1 then
            return Select (P, q, i);
        else
            return Select (q+1, r, i-(q-P+1))
        end if
    end if
```

* Greed Method.

- A greedy algorithm is an approach that makes the locally optimal choice at each step in the hope that this will lead to globally optimal solution.
- Greedy algorithms are efficient and widely used in optimization problem, but they do not guarantee the best solution.



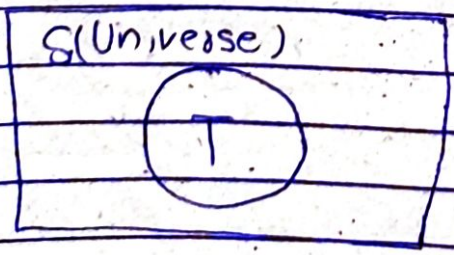

* An object activity Selection problem.

Choose a subset T of S of objects $A_i$

$S = \{ A_0, A_1, A_2 \ldots A_n \}$ such that

$$\sum_{A_i \in T} w_i \leq w$$

Summation of weights of all selected objects is less than w and |T| cardinality (no of object) is maximized.




$A_1, A_2, A_3 \ldots A_n$

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$

$w_1 \quad w_2 \quad w_2 \quad w_n$

Each object has weight $w_i$

Algo:

Sort such that
$w_i < w_j$ if $i < j$
$T = \phi$, $\omega(T) = 0$,
$i = 1$
while $\omega(T) + w_i \leq W$ do
$T = T \cup \{A_i\}$
$\omega(T) = \omega(T) + w_i$
$i = i + 1$;
end while

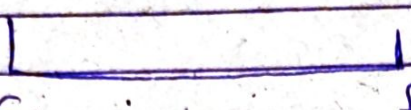
Ex $A = \{A_1, A_2, A_3, A_4\}$
$w_1 = 20$, $w_2 = 5$, $w_3 = 25$, $w_4 = 1$

Sorting as per weights.

| $A_4$ | $A_2$ | $A_1$ | $A_3$ |
|---|---|---|---|
| $w_4 <$ | $w_2 <$ | $w_1 <$ | $w_3$ |
| 1 | 5 | 20 | 25 |

* Scheduling Problem:
- Scheduling problem involve allocating resources over time to perform a set of task efficiently.
- We are having only one CPU which can run one process at a time
- We are given a set of processes each with a starting and ending time
- We have to schedule processes such that we can maximise the number of

processes without overlapping




| Process | 1 | 2 | 3 |
|---|---|---|---|
| Start | $x_1$ | $x_2$ | $x_3$ |
| End | $y_1$ | $y_2$ | $y_3$ |

Algo :

Sort so that $f_i \leq f_j$ if $i<j$

//Sort acc to end time

T = {1}, last = 1;

for i=2 to n do

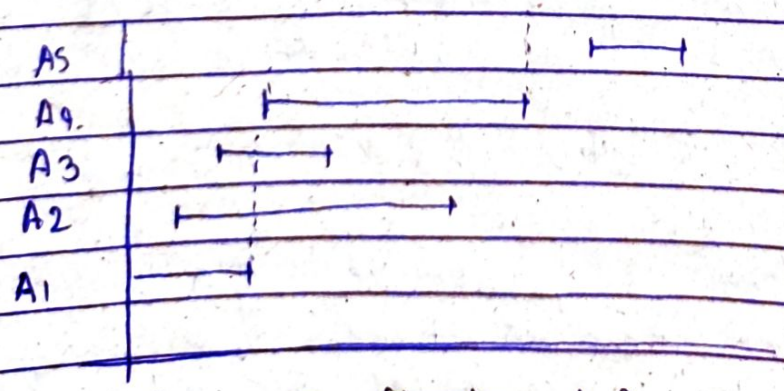
  if last ≤ $s_i$ then

    T = T ∪ {i};

    last = i

  end if

end for




T = {$A_1$, $A_4$, $A_5$}

Q Write greedy algorithm to pair numbers that minimises maximum sum

Algo:

```
Sort : So that A[i] < A[j] if i < j
left = 0
right = length (A)
max_sum = 0.
while left < right do.
    temp_Sum = A[left] + A[right]
    max_sum = max (max_sum, temp_Sum)
    left = left + 1
    right = right + 1.
end while.
return max_Sum
```

eg. [20, 1, 30, 15, 25, 6]

Sort : [ 1, 15, 20, 25, 30 ]

Pair first with last and so on.

{1, 30}, {15, 25}, {15, 20}

Minimum of max sum = 40

* Huffman Codes:

- Huffman coding is a greedy algorithm used for lossless data compression.
- It assigns variable - length codes to input characters, ensuring the most frequent character gets

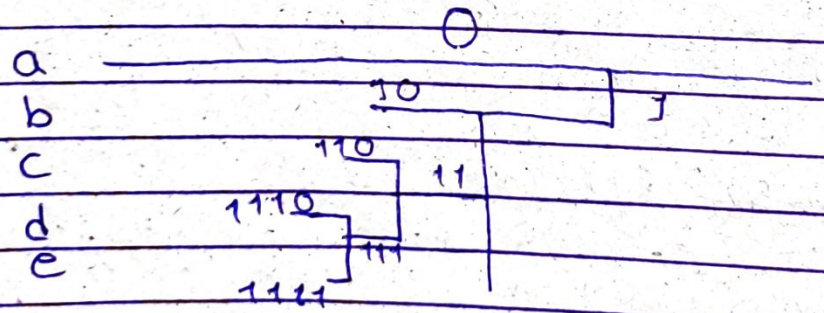
Shorter code and the less frequent character gets longer codes.

- It helps to reduce storage space and transmission cost.
- It saves 20% - 90% of typical transmission cost and size.

- It counts frequency of each characters and greedily assigns bits to each character.

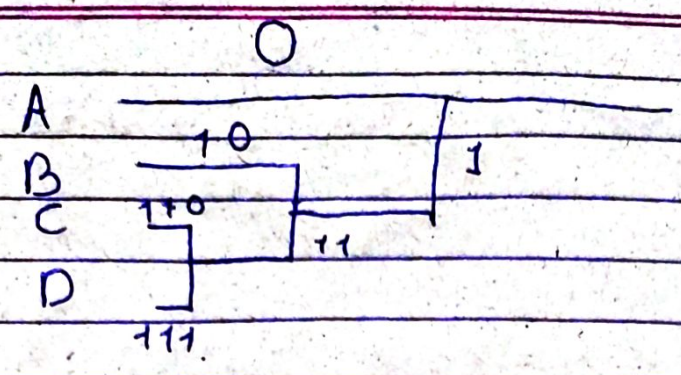
| | | fixed length | Variable leng |
|---|---|---|---|
| ex. | a | 000 | 0 |
| | b | 001 | 10 |
| | c | 010 | 110 |
| | d | 011 | 1110 |
| | e | 100 | 1111 |

Here we assum that freq a > b > c > d > e
Thus a has shortest length bits.




Ex. CABAAABBACDCB

| Symbol | Freq/Count |
|---|---|
| A | 5 |
| B | 4 |
| C | 3 |
| D | 1 |




- The variable-length code can do considerably better than a fixed-length code by giving frequent characters short code words, reducing overall length.