

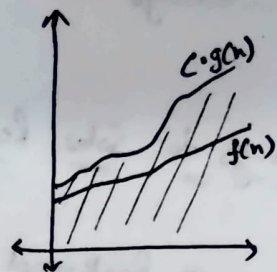
Asymptotic Notation

1) Big-oh (O) $\Rightarrow$ worst case upper bound

$$f(n) = O(g(n))$$

$$f(n) \leq C \cdot g(n)$$

- * $C > 0$
- * $n \geq n_0$
- * $n_0 \geq 0$

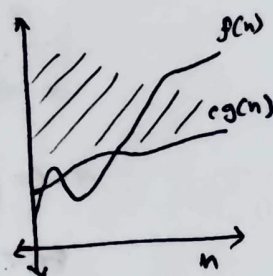


2) Big-Omega (Ω) $\Rightarrow$ best case lower bound

$$f(n) = \Omega(g(n))$$

$$f(n) \geq c \cdot g(n)$$

- * $c > 0$
- * $n > n_0$
- * $n_0 \geq 0$



3) Theta (Θ) $\Rightarrow$ Average Case Exact time

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

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Recurrence Relations

A recurrence relation represents a function in terms of its smaller inputs or the previous values.

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \rightarrow \text{additional work done}$$

$\swarrow$ factor number of subproblems $\searrow$ factor by reducing

* Important recursive

bubble Sort $\Rightarrow$

bubble sort (arr, n) {

if $n == 1 \rightarrow$ base case
return

for (int i = 0; i < n; i++) {
if (arr[i] > arr[i+1]) {

} Swapping

swap

}
bubble sort (arr, n-1); $\rightarrow$ recursive call

$$* T(n) = T(n-1)$$

$$+ O(n);$$

$$T(1) = 1$$

Amey

* Binary Search $\Rightarrow T(n) = T(n/2) + c ; T(1) = 1$

* Merge Sort $\Rightarrow T(n) = 2T(n/2) + O(n) ; T(1) = 1$

Ways to Solve Recurrence Relation

* Backward Substitution $\Rightarrow$

- 1 $\rightarrow$ Expand the recurrence by substituting $T(n)$
- 2 $\rightarrow$ Summarize pattern in a closed form
- 3 $\rightarrow$ Equate the term inside with base case.
- 4 $\rightarrow$ Substitute k back in closed form.

Ex:- Merge Sort $\Rightarrow T(n) = 2T(n/2) + n ; T(1) = 1$

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

$$T(n/4) = 2T(n/8) + n/4$$

$$\Rightarrow T(n) = 2(2(2T(n/8) + n/4) + n/2) + n$$

$$= 2(4T(n/8) + n) + n$$

$$= 8T(n/8) + 3n$$

Generalised $\Rightarrow 2^3 T(n/2^3) + 3n$

$$\boxed{2^k T(n/2^k) + kn}$$

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\boxed{k = \log_2 n}$$

$$\Rightarrow 2^k T(1) + kn$$

$$\Rightarrow 2^k + kn$$

$$\Rightarrow 2^{\log_2 n} + \log_2 n \times n$$

$$\Rightarrow n + \log_2 n$$

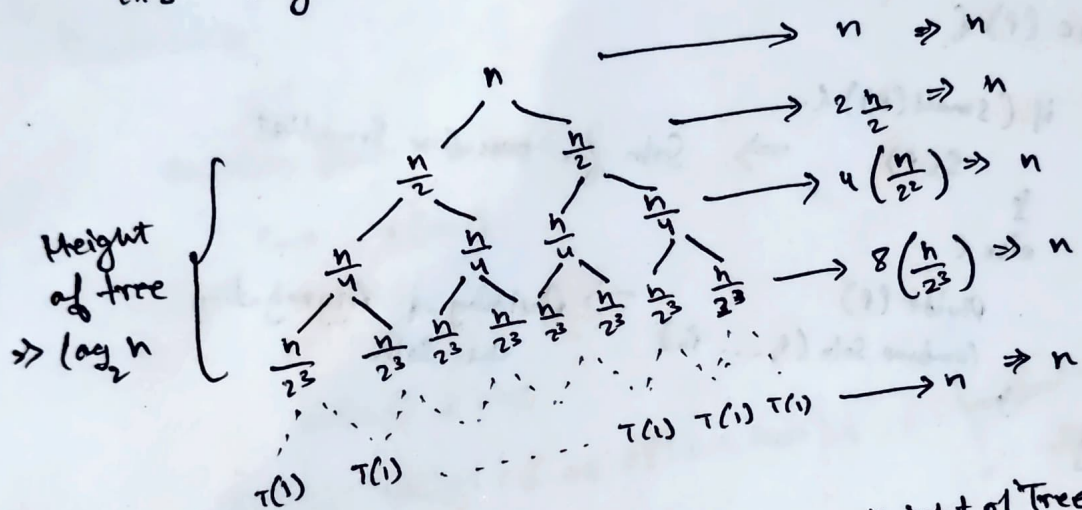
$$\Rightarrow \boxed{n \log n} \text{ Ans}$$

Ans

* Height Recursive Tree Method

- 1 $\Rightarrow$ Expand the Recurrence tree
- 2 $\Rightarrow$ Calculate Cost at each level
- 3 $\Rightarrow$ Total Cost for the tree using either height $\times$ cost or sum of cost of individual.
- 4 $\Rightarrow$ $O(\text{cost})$ will be the answer

Ex: Merge Sort $\Rightarrow T(n) = 2T(n/2) + O(n)$



$\therefore$ Total Cost $\Rightarrow n \times \text{Height of Tree}$
 $\Rightarrow n \times \log_2 n$

Ans $\Rightarrow O(n \log n)$

* Master Theorem

$T(n) = aT(\frac{n}{b}) + f(n)$, $a \geq 1$; $b > 1$ & $f(n) = O(n^c)$

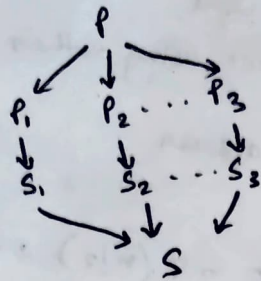
Case 1 $\Rightarrow n^{\log_b a} > f(n) \rightarrow T(n) = \Theta(n^{\log_b a})$

Case 2 $\Rightarrow n^{\log_b a} < f(n) \rightarrow T(n) = \Theta(f(n))$

Case 3 $\Rightarrow n^{\log_b a} = f(n) \rightarrow T(n) = \Theta(n^{\log_b a} \log n)$

Ans

Divide and Conquer Approach



→ Division of a large Problem into smaller parts and then solving the smaller chunk.

DAC (P) {

if (small(P)) {

S(P)

⇒ Soln for base case smallest

}

else {

Divide (P)

Combine Soln (P1, ..., Pn)

⇒ Dividing & Aggregating the Soln

}

}

* Quick Sort :-

1 ⇒ Select Pivot Index

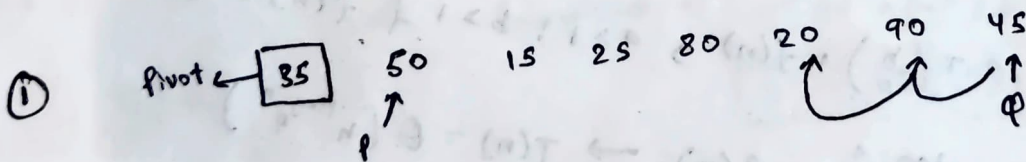
2 ⇒ Partition Algorithm

3 ⇒ Recursively Apply Quick Sort

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

$$\begin{aligned} &\rightarrow O(n^2) \\ &\rightarrow O(n \log n) \\ &\rightarrow O(n \log n) \end{aligned}$$

↓
for optimised Partition



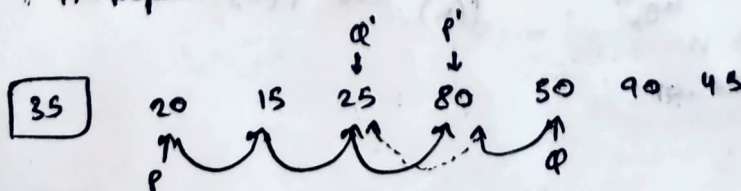
Move P to right till P value is smaller than Pivot.

Move Q to left till Q value is greater than Pivot.

Swap P & Q. (if i < j or i < Q)

Repeat this till P ≤ Q.

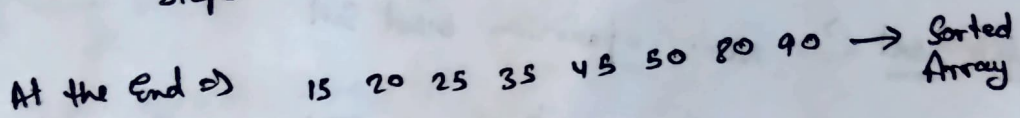
②



Swap Pivot with Q at last of partition

Day

③



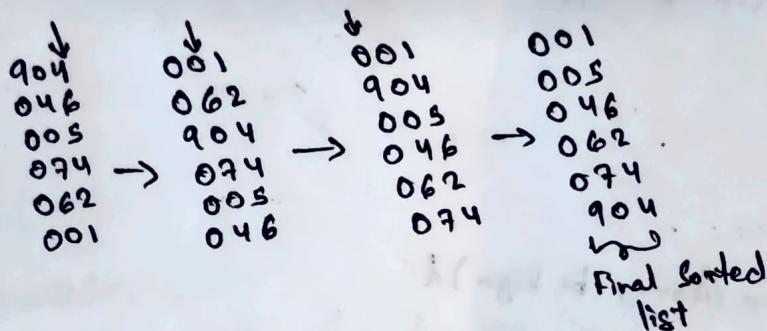
partition (Aarr, be, high) d

Recursive Calls Algo

002

Note: The Average number of comparisons performed for merging two sorted lists of length 2 in Merge Sort — ??

Radix Sort $\Rightarrow$ Non-Comparison based Sort



- # Sort the Number based on digit.
- # Don't change relative order for equals.
- # Repeat steps for all digits from right to left.

Bucket Sort $\Rightarrow$

- 1 $\Rightarrow$ find range of input values.
- 2 $\Rightarrow$ Create k empty buckets ($k = n$ usually, $n = \text{number of elements}$).
- 3 $\Rightarrow$ Distribute elements in bucket for insertion in filled bucket use insertion sort Algo.
- 4 $\Rightarrow$ concatenate all buckets.

Example $\Rightarrow$ 0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68

- 0 $\rightarrow$ []
- 1 $\rightarrow$ [0.12, 0.17]
- 2 $\rightarrow$ [0.21, 0.23, 0.26]
- 3 $\rightarrow$ [0.39]
- 4 $\rightarrow$ []
- 5 $\rightarrow$ []
- 6 $\rightarrow$ [0.68]
- 7 $\rightarrow$ [0.72, 0.78]
- 8 $\rightarrow$ []
- 9 $\rightarrow$ [0.94]

Can be Related as Hash Map with Chaining.

Concatenate all the get sorted Array

Ans

MaxMin Problem

The Max-min Problem involves finding maximum & minimum elements in an array using the divide & conquer approach.

1 $\Rightarrow$ Split the array into two halves until each half has one or two elements.

2 $\Rightarrow$ • Single element is max & min both.
• for two element compare them and return max min.

3 $\Rightarrow$ merge results from the two halves.

$O(n) \rightarrow$ Time Complexity

Pseudo Code $\Rightarrow$

```
MaxMin (arr, low, high) {
```

```
    if (low == high) {
```

```
        return (arr[low], arr[high])
```

} Base Case of One Element

```
    } else if (high == low + 1) {
```

```
        if (arr[low] > arr[high]) {
```

```
            return (arr[high], arr[low]);
```

} Base Case of two Elements

```
        } else {
```

```
            return (arr[low], arr[high]);
```

```
        } else {
```

```
            mid = (low + high) / 2;
```

```
            (min1, max1) = MaxMin (arr, low, mid);
```

```
            (min2, max2) = MaxMin (arr, mid+1, high);
```

```
            overall-min = min (min1, min2)
```

```
            overall-max = max (max1, max2)
```

```
            return (overall-min, overall-max);
```

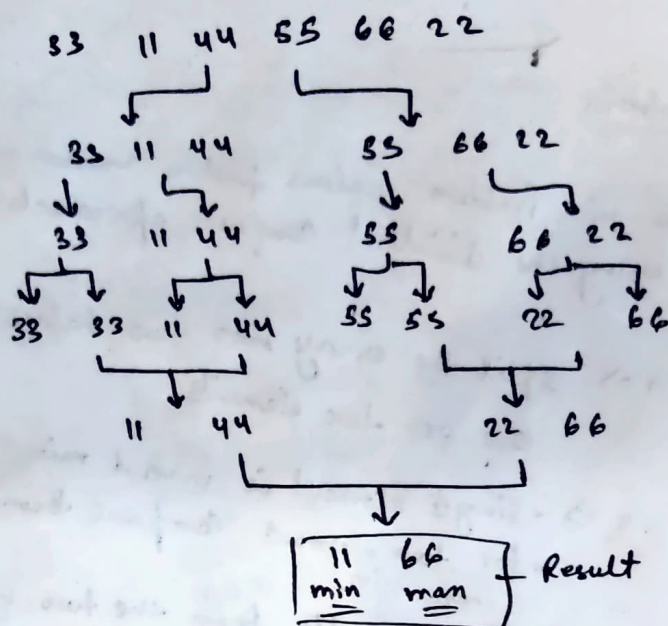
```
        }
```

```
}
```

} Dividing & Aggregating

Done

Ex 3-



Median of two Sorted Array

We can find Median of two Sorted Array by Merging them & calculating Middle which will take $O(m+n)$ but Divide & Conquer will take $O(\log(\min(m, n)))$. $\downarrow$ two sizes

- 1 $\Rightarrow$ Partition ^{smaller} both array $\&$ using binary Search
- 2 $\Rightarrow$ Ensure correct partitioning.
- 3 $\Rightarrow$ Find Median based on partition.

Pseudo code $\Rightarrow$

findMedian (arr1, arr2) {

if (len(arr1) > len(arr2)) {

Swap(arr1, arr2)

n1 = len(arr1)

n2 = len(arr2)

low = 0, high = n1

while (low < high) {

p1 = (low + high) / 2;

p2 = (n1 + n2) / 2 - p1;

maxL = - ∞ if p1 == 0 else arr1[p1-1];

minR = ∞ if p1 == n1 else arr1[p1];

maxR = - ∞ if p2 == 0 else arr2[p2-1];

minR = ∞ if p2 == n2 else arr2[p2];

Qay

```

if (maxL < minR && maxL <= minR) {
    if ((n+m)/2 == 0) {
        return (max(maxL, maxR)) + min(minL, minR) / 2;
    }
    else {
        return max(maxL, maxR);
    }
}
else if (maxL > minR) {
    high = Partition - 1;
}
else {
    low = Partition + 1;
}
return 0;

```

Optimization Problems

Finding out most efficient & feasible solution for a problem.

Solution space $\Rightarrow$ set all possible soln in n inputs.

Feasible space $\Rightarrow$ solutions from soln space satisfy constraints.

Optimal soln $\Rightarrow$ one of feasible soln which optimizes our goal.

maximization or minimization

Types $\Rightarrow$ Greedy Algo
 Dynamic Algo
 Branch & Bound

Greedy Algorithm

optimization problem where we find local best solution for getting general best soln.

Local best soln $\Rightarrow$ best soln at the place of process can be max or min.

General best soln $\Rightarrow$ chaining of these local soln to a best general soln.

DD

- 1 $\Rightarrow$ Finding All possible at the instance.
- 2 $\Rightarrow$ Evaluating cost of possible soln.
- 3 $\Rightarrow$ Selecting local max or min.
- 4 $\Rightarrow$ Repeating.

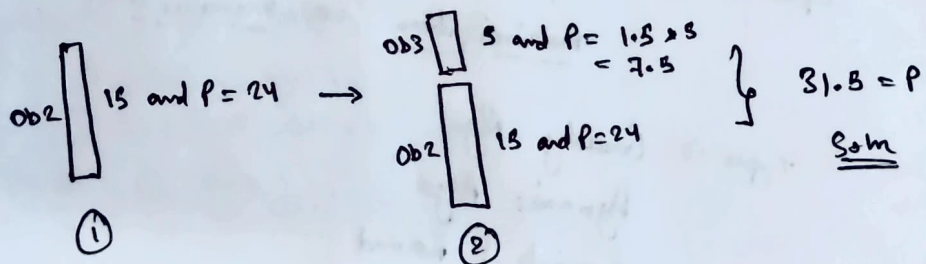
KnapSack (fractional)

Given n objects and a knapsack (bag). Object has profit & corresponding weight for each object. Constraint is we can have at max M weight in bag & want maximum profit. Can select object fractionally.

- Algo $\Rightarrow$
- 1 $\Rightarrow$ calculate profit / weight ration for all
 - 2 $\Rightarrow$ Select highest P/W .
 - 3 $\Rightarrow$ Add it accordingly in bag.
 - 4 $\Rightarrow$ Repeat till bag is full.

Ex:

Object	<u>Ob1</u>	<u>Ob2</u>	<u>Ob3</u>	
P	25	24	15	M = 20
W	18	15	10	
P/W	1.39	1.6	1.5	



Pseudo code $\Rightarrow$ knapsack(w , items)

calculate & sort items on basis of P/W . $\Rightarrow O(n \log n)$

totalValue = 0;

remainingW = W;

for (item in items) {

if (remainingW $\geq$ item.weight)

take whole

else

take fraction

total $\Rightarrow O(n \log n)$

Proof

Huffman Coding

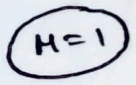
- (i) Count Frequency of Each letter
- (ii) Store them in Min-Heap (select lowest first)
- (iii) Create Huffman Tree - Intermediate Node

Huffman code is greedy Algo for lossless data compression. Compresses a word (data) into binary codes assigned to each unique letter based on their frequency.

Ex:- word = MISSISSIPPI, M=1, H=1, S=4, I=4, P=2

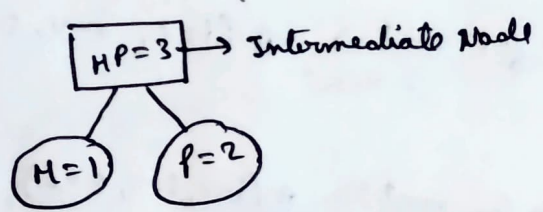
① Min Heap $\Rightarrow$ H=1, P=2, I=4, S=4

Huffman Tree $\Rightarrow$



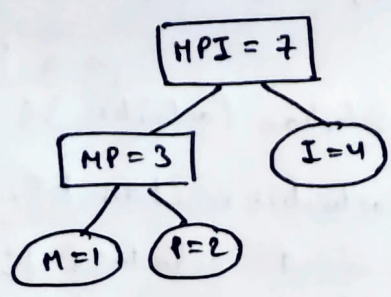
② Min Heap $\Rightarrow$ P=2, I=4, S=4

Huffman Tree $\Rightarrow$



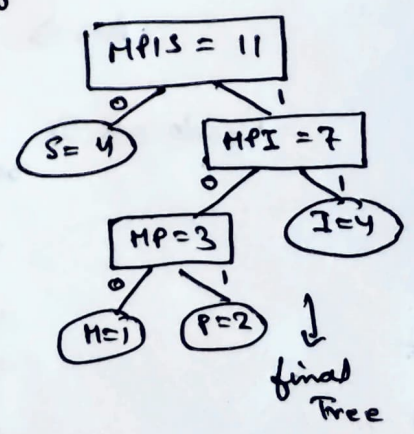
③ Min Heap $\Rightarrow$ I=4, S=4

Huffman Tree $\Rightarrow$



④ Min Heap $\Rightarrow$ S=4

Huffman tree $\Rightarrow$



④ Traverse and assign zero for each left & Right = 1. from bottom to top will traversal.

M=001, P=101, I=11, S=0 $\Rightarrow$ Huffman codes

Done

Activity Selection Problem

A Greedy Problem where we have n activities, each have start time, end time. Goal is to select maximum number of activities that don't overlap in execution.

Algo $\Rightarrow$ 1 $\Rightarrow$ Sort activity based on end time smallest first.

2 $\Rightarrow$ Select first activity.

3 $\Rightarrow$ For each remaining activity if start time $>$ end time of current select it then.

4 $\Rightarrow$ Repeat until all activities are considered.

Ex:-
i $\Rightarrow$ 1 2 3 4 5
(ST) S $\Rightarrow$ 1 4 12 15 7
(ET) E $\Rightarrow$ 13 6 14 19 13
 $\Rightarrow$ Sort i $\Rightarrow$ 2 1 5 3 4
S $\Rightarrow$ 4 1 7 12 15
E $\Rightarrow$ 6 13 13 14 19

Select $i=2 \Rightarrow A(i=2, S=4, E=6)$

check for next $\Rightarrow A(i=2, S=4, E=6)$ A1 is not taken because start time less than current

check for next $\Rightarrow A(i=2, S=4, E=6), A(i=5, S=7, E=13)$

$\vdots$

Ans $\Rightarrow A(i=2, S=4, E=6), A(i=5, S=7, E=13), A(i=4, S=15, E=19)$

Pseudo code $\Rightarrow$ $\nearrow$ list of activity
activity Selection (activities) {

Sort activities based on ET.

activity current = activities[0];

ans.add(current);

for (activity (excluding 0) in activities) {

if (activity.start > current.end) {

current = activity;

ans.add(activity);

} // n-1
for
sorting

[Signature]