Sumedh

- Topics (Unit -1,2 combined)
  - General
    1. Asymptotic notatn
    2. Recurrence rel & types
  - Brute force (unit - 1)
    1. Linear Search
    2. Insertn sort
    3. Bubble sort
  - Divide n Conquer (unit -2)
    1. Binary srch
    2. Merge sort
    3. Quick sort
    4. Strassen Matrix
    5. Min - max
    6. Closest pair
    7. Convex Hull
    8. Subarray
- Asymptotic Notations

- Big Notations
  - i) Big - oh (O)
  - ii) Big omega (Ω)
- Small Notations
  - i) Small oh (o)
  - ii) " omega (ω)

→ lose bound

lose + tight bound

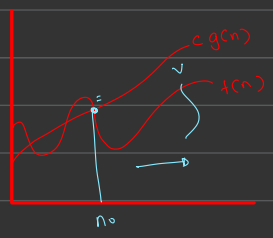
v) Theta (Θ)

↳ tight bound

oh - upper bound
omega - lower bound

1. Big-Oh (O)

- UB
- $f(n)$ is $O(g(n))$ exists if $c > 0$ & $n_0 > 0$
- $f(n) \leq c \cdot g(n)$ where $n \geq n_0$



("think larger vals of n)

eg $F(n) = 2n$
$g(n) = 2^n$

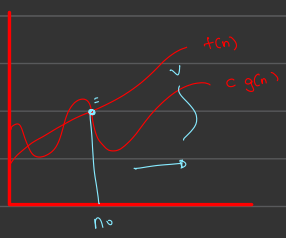
→

at $n =$	1	2	2
	2	4	4
	3	6	8
	⋮	⋮	⋮

$\therefore \; 2n = O(2^n)$

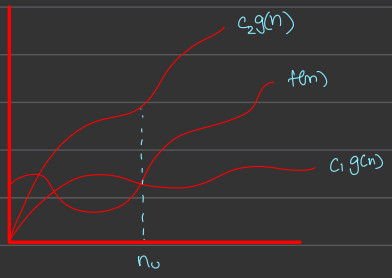
2. Big-omega (Ω)

- LB
- $f(n) = \Omega(g(n))$ exists
- If $f(n) \geq c \cdot g(n)$ where $n \geq n_0$



3. Theta (Θ)

- Tight bound
- $f(n) = \Theta(g(n))$ if $fn = O(g(n)) = \Omega(g(n))$
- $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$



eg $f(n) = n^2 + n + 10$

if $n^2 + n + 10 \leq 1000 \times n^2$

then

$n^2 + n + 10 = O(n^2)$

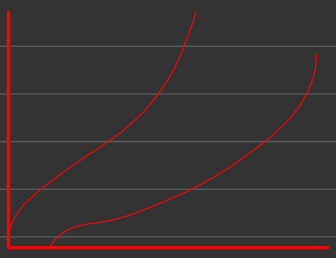
if $n^2 + n + 10 \geq 1 \times n^2$

$n^2 + n + 10 = \Omega(n^2)$

$\therefore \quad n^2 + n + 10 = \Theta(n^2)$

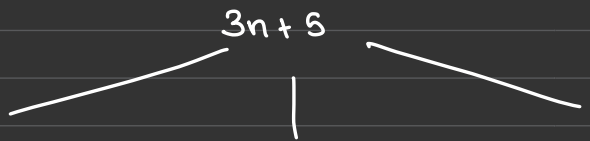
Note ($n^x + n^{x-1} + n^{x-2} \ldots$) $= \Theta(n^x)$

4. Small - oh
   - Proper UB
   - $f(n) = o(g(n))$ for all $c > 0$
   - $f(n) < c \cdot g(n)$ when $n > n_0 > 0$ ⟶ shld be true for all c



5. Small - omega (ω)
   - Proper LB
   - $f(n) = \omega(g(n))$ for all $c > 0$
   - $f(n) > c \cdot g(n)$ $n > n_0 > 0$ ⟶ shld be true for all c

eg $f(n) = 3n + 5$



tight UB ← $3n + 5 \leq 100 * n$ $\quad 3n+5 = \Theta(n)$ $\quad 3n + 5 \geq 1 \times n$ → Tight LB

$\therefore \quad 3n + 5 = O(n)$ $\quad 3n + 5 = \Omega(n)$

also $= O(n^2)$ $\quad$ also $= \Omega(\sqrt{n})$

$= O(n^3)$ $\quad = \Omega(1)$

$\vdots$

Loose UB $\quad$ Loose LB

* Growth rate

$O(1) < O(\log n) < O(n) <$

$O(n \log n) < O(n^2) <$

$O(n^n) < O(2^n)$

- Recurrence relatn & types

recurrence relatn is to find Time complexity of recursive algo

eg $T(n) = T(n/2) + c$ — [for Binary Srch]

There r 3 types to solve recurrence relatn

1. Back Substitution method
2. Recursion tree method
3. Master's theorem

- Back Substitution

eg : $T(n) = T(n-1) + n$

→ Base Case : at $n = 1$

$T(1) = T(1-1) + 1$ or $O(\text{constant})$

$T(1) = 1$ ~ $O(c)$

$T(n) = T(n-1) + n$ — (i)

$T(n-1) = T(n-2) + (n-1)$ — (ii) [reduce n by 1]

$T(n-2) = T(n-3) + (n-2)$ — (iii) same

Substitute (iii) in (ii)

$T(n-1) = [T(n-3) + (n-2)] + n - 1$ — (iv)

Now substitute (iv) in (i)

$$T(n) = [T(n-3) + (n-2) + (n-1)] + n$$

$$T(n) = T(n-3) + 3n - 3$$

$$\vdots$$

similarly till k steps

$$T(n) = T(n-k) + kn - k$$

Base cond^n : $T(1) = 1$

i.e $T(n-k) = T(1)$

$$n - k = 1$$

$$k = n-1$$

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

$$\therefore T(n) = 1 + n^2 - n - n - 1$$

i.e $\boxed{T(n) = O(n^2)}$

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

→ $T(n) = 2T(n/2) + n \quad -(i)$

reduces by $n/2$

$$T(n/2) = 2T(n/4) + n/2 \quad -(ii)$$

$$T(n/4) = 2T(n/8) + n/4 \quad -(iii)$$

Sub. (iii) in (ii)

$$T(n/2) = 2\left[2T(n/8) + n/4\right] + n/2 \quad -(iv)$$

$$= 2^2 T(n/8) + \frac{2n}{4} + n/2$$

Sub (iv) in (i)

$$T(n) = 2\left[2^2 T(n/8) + \frac{2n}{4} + \frac{n}{2}\right] + n$$

$$= 2^3 T(n/8) + \cancel{\frac{2^2 n}{4}} + \cancel{\frac{2n}{2}} + n$$

$$T(n) = 2^3 T(n/2^3) + 3n$$

$$\vdots$$

Similarly till k

$$T(n) = 2^k T(n/2^k) + kn$$

Base condition $T(n/2^k) = T(1) = O(c)$

$$n/2^k = 1$$

$$n = 2^k$$

take log

$$\log n = \log 2^k$$

$$\therefore k = \log_2 n$$

$$T(n) = 2^k T(n/2^k) + kn$$

$2^k = n$

$k = \log_2 n$

$$T(n) = n \times O(c) + (\log_2 n) \times n$$

$$\approx \boxed{T(n) = n \log n}$$

• Master Method

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$ and $f(n)$ is an asymptotically positive function.

- There are following three cases:

1. If $f(n) < O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) > \Omega(n^{\log_b a})$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.

• Recursive Tree Method

Example 1: Solving `T(n) = 3T(n/2) + n`

Step 1: Expand the Recurrence

Each call branches into 3 subproblems with half the size:

```
pgsql                                                    Copy  Edit

Level 0 (Root):     T(n)          → Work = n
Level 1:          3T(n/2)         → Work = 3(n/2)
Level 2:        9T(n/4)           → Work = 9(n/4)
Level k:    3^k T(n/2^k)          → Work = 3^k (n/2^k)
```

Step 2: Find the Height of the Tree

Stop when `n/2^k = 1`:

$$n = 2^k$$

Taking log:

$$k = \log_2 n$$

Step 3: Sum Work at Each Level

$$W = n + \frac{3}{2}n + \frac{9}{4}n + \cdots + 3^k O(1)$$

Using sum formula:

$$W = O(n^{\log_2 3})$$

Final Answer: `T(n) = O(n^{1.58})`

- Linear Search

Approach : Check each element in array until u get target

Algorithm :-

1. Make a for loop
2. Start from 1st elem
3. Compare with targe
4. If not, move nesct
5. repeat until array end
6. If not found, -1

Code :-

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 30;

    int result = linearSearch(arr, n, key);
    if (result != -1)
        printf("Element found at index %d\n", result);
    else
        printf("Element not found\n");

    return 0;
}
```

pseudo code

Time Complexity :

Best-Case : O(1) → 1st elem = target
Worst-Case : O(n)
Avg. Case : O(n)

Space Complexity : O(1)

Analysis : Eg A[] = {1,2,3,4,5}, key = 5

i = 1
i ≠ key
i++
i = 2
i ≠ key
i++
i = 3
i == key ✓

- Insertion Sort

Approach : Sort the arr one elem at a time, comparing with prev elem n shift

Algo :-

1. Start from elem 2
2. Compare with prev elem n shift if greater
3. Insert the curr elem in correct positn
4. Repeat for all

Code :

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
                        // compare with prev
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]; // shifting
            j--;  // go till first elem
        }
        arr[j + 1] = key;
    }
}
```

Time Complexity :

BC : O(n) → already sorted
WC : $O(n^2)$
AC : $O(n^2)$

SC : O(1)

Analysis :

A[] = 60 40 45 87 36

pass-1 : 60 | 40 45 87 36

p-2 : 40 60 | 45 87 36

p-3 : 40 45 60 | 87 36

p-4 : 40 45 60 87 | 36

p-6 : 36 40 45 60 87

- Bubble Sort

Approach : Compare adjacent elem and swap if wrong order

Algo :-

1. Start from 1st elem, compare with adj one
2. if adj > 1st swap
3. if no swaps occur in a pass, break
4. Repeat for entire array

Code :

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int flag = 0;  // 0 means no swaps
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                flag = 1;  // Swap occurred
            }
        }
        if (flag == 0)  // No swaps in this pass, array is sorted
            break;
    }
}
```

TC : BC : $O(n)$
WC : $O(n^2)$
AC : $O(n^2)$

SC : $O(1)$

Analysis : A[] = 70 5 90 15 20 25 80 40

p-1 : 5 70 15 20 25 80 40 90

p-2 : 5 15 20 25 70 40 80 90

p-3 : 5 15 20 25 40 70 80 90

p-4 : No swaps, break

- **Binary Search**

Approach : divide the arr in 2 halves n compare middle with take n accordingly search in left / rgt arr

(* Only for sorted array)

Algo :-

1. Take two var, low & high
2. Set low = 0 & high = n-1
3. repeat until low ≤ high
   - i) mid = (low + high)/2
   - ii) key = mid, return key
   - iii) mid > key, search left
   - iv) < right
4. repeat

Code :

```
int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key)
            return mid;
        else if (arr[mid] > key)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}
```

TC :

BC : O(1)
WC : O(log n)
AC : O(log n)

SC : O(1)

TC relatn : $T(n) = T(n/2) + c$

Analysis :

A: | 10 | 20 | 30 | 40 | 50 | , key = 20

| 10 | 20 | ∵ 20 < 30 ; left

| 10 | 20 | — 20 = 20 ✓

- Merge Sort

Approach : recursively divide arr in two halves then sort n merge sort

Algo :
1. If arr has 1/0 elem, return base case
2. div arr in 2 halves
3. recursively sort em
4. Merge

Code :

```
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}
```

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

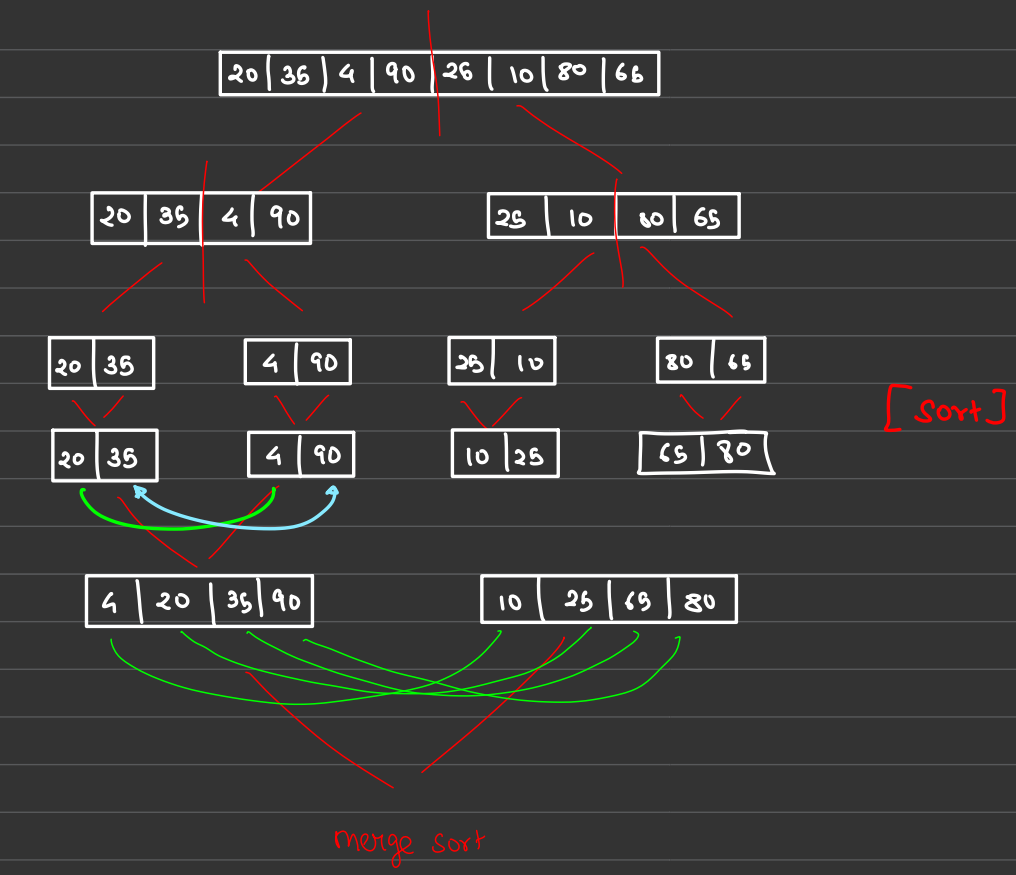
TC : BC : $O(\log n)$
WC : $O(\log n)$
AC : $O(\log n)$

SC : $O(n)$ → extra arr

TC relation : $T(n) = 2T(n/2) + O(n)$

Analysis :

Next pg



- Quick Sort

Approach : pick a pivot elem , partition the arr around it such that smaller on left & great on right then recursively sort both

Algo :
1. Select a pivol (last / first)
2. partition the arr
3. recursively sort them

Code :

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

TC : BC : $O(n \log n)$
WC : $O(n^2)$
AC : $O(n \log n)$

SC : $O(1)$

TC relatn : $T(n) = T(k) + T(n-k-1) + O(n)$

- Strassen Matrix

Approach : divides matrices into submat and recursively muliplies them using only 7 instead of 8

Algo :

1. Given two N × N matrices A and B, divide them into four (N/2 × N/2) submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

2. Compute 7 intermediate products:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

3. Compute the resultant matrix C:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

4. Recursively apply Strassen's algorithm until the matrices are reduced to size 1×1, where direct multiplication occurs.

Tc : BC : $O(n^{2.81})$
WC : $O(n^{2.81})$
AC : $O(n^{2.81})$

SC : $O(1)$

TC relatn : $T(n) = 7T(n/2) + O(n^2)$

$\therefore T(n) = O(n^{\log_2 7}) = O(n^{2.81})$

- Min - Max problem

Approach : divide the arr in two halves, recursively find min & max in each half then merge

Algo :
1. divide
2. recursively find min-max in both
3. Compare

Code :

```
MinMax findMinMax(int arr[], int low, int high) {
    MinMax result, left, right;
    int mid;

    // Base Case: Only one element
    if (low == high) {
        result.min = arr[low];
        result.max = arr[low];
        return result;
    }

    // Base Case: Two elements
    if (high == low + 1) {
        if (arr[low] < arr[high]) {
            result.min = arr[low];
            result.max = arr[high];
        } else {
            result.min = arr[high];
            result.max = arr[low];
        }
        return result;
    }

    // Divide the array into two halves
```

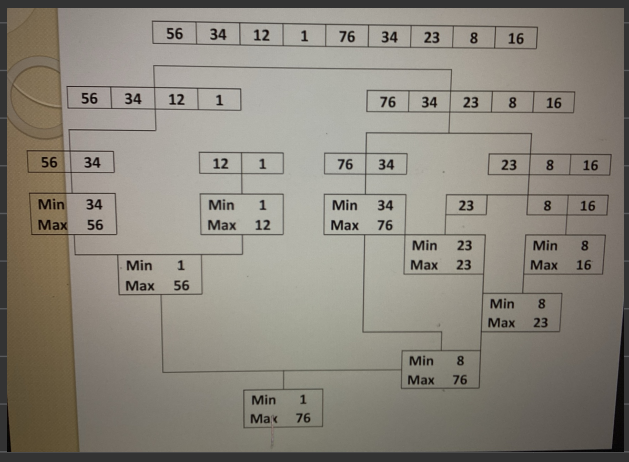
```
    // Divide the array into two halves
    mid = (low + high) / 2;
    left = findMinMax(arr, low, mid);
    right = findMinMax(arr, mid + 1, high);

    // Conquer: Compare results from left and right halves
    result.min = (left.min < right.min) ? left.min : right.min;
    result.max = (left.max > right.max) ? left.max : right.max;

    return result;
}
```

TC :
- BC : O(n)
- WC : O(n)
- AC : O(n)

SC : O(1)

TC rcl : $T(n) = 2T(n/2) + O(1)$

Analysis :



- Max^m Subarray

Approach : Splits arr into 2 halves, find max^m subarry sum in each half & considers subarrys crossing the middle

Algo :

1. **Base Case:** If the array has only one element, return it.
2. **Divide:** Split the array into two halves.
3. **Conquer:** Recursively find:
   - Maximum subarray sum in the left half.
   - Maximum subarray sum in the right half.
   - Maximum crossing subarray sum spanning both halves.
4. **Combine:** Return the maximum of these three values.

Code :

```
int maxCrossingSum(int arr[], int low, int mid, int high) {
    int left_sum = INT_MIN, right_sum = INT_MIN;
    int sum = 0;

    // Compute left half max sum
    for (int i = mid; i >= low; i--) {
        sum += arr[i];
        if (sum > left_sum)
            left_sum = sum;
    }

    sum = 0;
    // Compute right half max sum
    for (int i = mid + 1; i <= high; i++) {
        sum += arr[i];
        if (sum > right_sum)
            right_sum = sum;
    }

    return left_sum + right_sum;
}
```

```
int maxSubArraySum(int arr[], int low, int high) {
    if (low == high) // Base case: single element
        return arr[low];

    int mid = (low + high) / 2;

    int left_sum = maxSubArraySum(arr, low, mid);
    int right_sum = maxSubArraySum(arr, mid + 1, high);
    int cross_sum = maxCrossingSum(arr, low, mid, high);

    return (left_sum > right_sum) ? ((left_sum > cross_sum) ? left_sum : cross_sum)
                                  : ((right_sum > cross_sum) ? right_sum : cross_sum);
}
```

TC :

BC : O(nlogn)
WC : O(n log n)
AC : O(nlogn)

TC relat^n : $T(n) = 2T(n/2) + O(n)$

- **Closest Pair**

Approach : divides set of point in two halves, find closest pair in each half & then check for closer pair across dividing line

Algo :

1. **Sort** the points based on their x-coordinates.
2. **Divide** the points into two halves.
3. **Recursively compute** the closest pair in both halves.
4. **Merge step:**
   - Find the minimum distance d from both halves.
   - Create a **strip** of points within distance d from the dividing line.
   - Sort the strip based on y-coordinates.
   - Check only the nearest 7 neighbors for a closer pair (using geometry).
5. **Return** the minimum distance found.

TC : BC : $O(n \log n)$
WC : $O(n \log n)$
AV : $O(n \log n)$

TC rel : $T(n) = 2T(n/2) + O(n)$

- **Convex Hull**

Approach : Approach recursively divides set of pts into two halves, find the convex hulls of each half & merge

Algo :

1. **Sort** the points based on x-coordinates.
2. **Divide** the points into two halves.
3. **Recursively compute** the convex hull for both halves.
4. **Merge** the two convex hulls:
   - Find the **upper tangent** (leftmost and rightmost points forming the topmost bridge).
   - Find the **lower tangent** (bottommost bridge).
   - Merge both hulls to form the final convex hull.

TC : BC : $O(n \log n)$
WC : $O(n \log n)$
AV : $O(n \log n)$

TC rel : $T(n) = 2T(n/2) + O(n)$