UNIT - II

MASTER'S THEOREM:

* It is also called as table look-up method. This method directly produce solutions to non-linear recurrence equations.
* The master theorem is used for solving divide & conquer recurrence relations.

Simplified Master Theorem:

Theorem Statement:

Let the time complexity function $T(n)$ be a positive and eventually a non-decreasing function of the following form:

$$T(n) = a \cdot T(n/b) + cn^k$$

$$T(1) = d$$

where a, b, d and k are all constants. $b \geq 2$, $k \geq 0$, $a > 0$ and $c > 0$ and $d \geq 0$.

The solution for the recurrence eqn. is given as follows.

case 1: $T(n) \in \Theta(n^k)$ if $a < b^k$

case 2: $T(n) \in \Theta(n^k \log n)$ if $a = b^k$

case 3: $T(n) \in \Theta(n^{\log_b a})$ if $a > b^k$

Example 1: Solve the following recurrence using simplified master theorem. $T(n) = 8 \cdot T(n/2) + n^2$.

The recurrence is of form $T(n) = a \cdot T(n/b) + cn^k$.

$a = 8$, $b = 2$, $c = 1$, $k = 2$.

Check the conditions, $a < b^k$, $a = b^k$ and $a > b^k$

In this eqn. $a > b^k$ (ie) $8 > 2^2 \Rightarrow 8 > 4$.

$\therefore T(n) = \Theta(n^{\log_2 8})$ ~~$= \Theta(n \log^2$~~

$= \Theta(n^{\log_2 2^3}) = \Theta(n^{3 \cdot \log_2 2}) = \Theta(n^3)$.

Example 2: Solve the recurrence equation using the simplified master theorem $T(n) = 2T(n/2) + n$

Here $a = 2, b = 2, c = 1, k = 1$.

Since $a = b^k$ (i.e) $2 = 2^1$

$T(n) = \Theta(n^1 \log n)$

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

Generalized Master Theorem:

Theorem Statement:

If the recurrence equation is of the form

$$T(n) = \begin{cases} d & n \leq n_0 \\ a \cdot T(n/b) + f(n) & n > n_0. \end{cases}$$

and $a \geq 1, b > 1, n_0 \geq 1, d > 0$ and $f(n)$ is positive for $n > n_0$, then the solution is subjected to the following conditions:

1. If $f(n) = O(n^{\log_b a} / n^{\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$
3. If $f(n) = \Omega(n^{\log_b a} \cdot n^{\epsilon})$ for $\epsilon > 0$ and if $af(n/b) \leq \delta f(n)$ for $\delta < 1$, then for large n, $T(n) = \Theta(f(n))$.

* The steps involved in the application of the generalized master theorem are as follows:

1. Compare $f(n)$ with the factor $n^{\log_b a}$
2. Based on step 1, the corresponding cases can be identified. $n^{\log_b a} < f(n) \Rightarrow$ case 3

$n^{\log_b a} > f(n) \Rightarrow$ case 1

$n^{\log_b a} = f(n) \Rightarrow$ case 2

Example 1: Solve the following recurrence eqn using generalized master theorem $T(n) = 8 \cdot T(n/2) + n^2$.

Here, $a = 8$, $b = 2$, $f(n) = n^2$.

Compare $f(n)$ with factor $n^{\log_b a}$

$$n^{\log_2 8} = n^{\log_2 2^3} = n^{3 \cdot \log_2 2} = n^3.$$

$$n^2 = O\left(\frac{n^3}{n^\epsilon}\right)$$

$$= O(n^{3-\epsilon}), \text{ when } \epsilon = 1$$

$$= O(n^2).$$

Therefore, the given time complexity function belongs to case 1.

Hence, $T(n) = \Theta(n^{\log_b a})$

$$= \Theta(n^{\log_2 8})$$

$$= \Theta(n^3).$$

Example 2: Solve using generalized master theorem.

$$T(n) = 2 \cdot T(n/2) + n.$$

Here $a = 2$, $b = 2$, $f(n) = n$.

Compare $f(n)$ with factor $n^{\log_b a}$

$$n^{\log_2 2} = \Theta(n).$$

This implies, it belongs to case 2.

$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$

$$= \Theta(n^{\log_2 2} \cdot \log n)$$

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

Example 3: $T(n) = 3 \cdot T(n/4) + n \cdot \log n$

$a = 3$, $b = 4$, $f(n) = n \cdot \log n$

$f(n)$ compare $n^{\log_b a}$

$n \cdot \log n = n^{\log_4 3} < n \log n$ [$\because \log_4 3 < 1$]

$\Downarrow$

0.79248

So case 3.

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

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

BINARY SEARCH:

* Binary search is a popular search algorithm.

Adv: It is faster and easier to implement.

Disadv: It is applicable to sorted lists only.

Steps:

Step 1: Read the ordered array of elements and search for a target key.

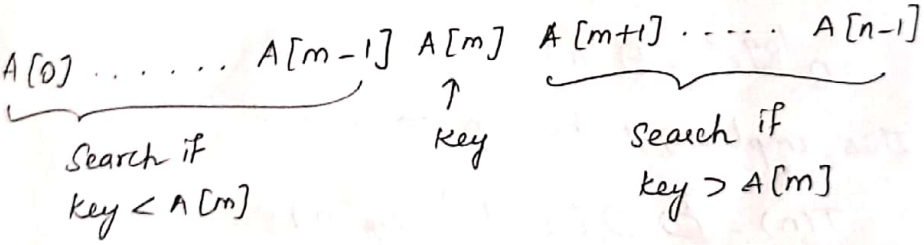
Step 2: Comp Find the middle element of ordered array.

Step 3: Compare the key and middle element

3a: If key is equal to middle element, element is found. Stop

3b: If key is lesser than middle element, search the lower half of ordered array A[1... mid-1] & go to step 2.

3c: If key is larger than middle element, search the upper half of ordered array A[mid+1...n] & go to step 2



Algorithm:

Algorithm BinSearch (A [0.... n-1], key)

// I/P: Array and key to be searched

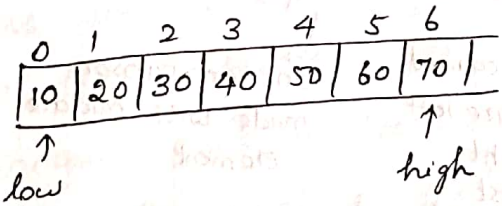
// O/P: Returns the index of array element if it is equal to key otherwise it returns -1.

low ← 0

high ← n-1

```
while (low < high) do
{
    m ← (low + high) / 2
    if (key == A[m]) then
    return m
    else if (key < A[m]) then
      high ← m-1        // search lower array (or) left sublist
    else
      low ← m+1         // search upper array (or) right sublist
}
return -1
```

Eg:



Key = 60.

m = (low + high)/2 = (0+6)/2 = 3.

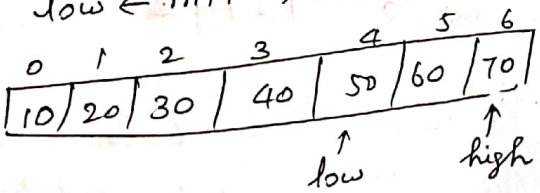
A[m] ⇒ A[3] = 40.

(60 == 40) false

(60 < 40) false

So search right sublist

low ← m+1 ⇒ low = 3+1 = 4.



m ← (4+6)/2 = 10/2 = 5.

A[m] ⇒ A[5] = 60.

60 == 60 ⇒ True.

returns 5 (i.e.) index of array element 60.

Algorithm Analysis:

+ The dominant operation done in binary search is comparison. The comparison done is 3-way comparison to determine whether key is smaller, equal to or greater than A[m].

The Best case analysis is O(1) done in one comparison. The search element equal to middle element of sorted array.

The worst case analysis

$$C_{worst}(n) = C_{worst}(n/2) + 1 \quad \text{for } n > 1$$

- $C_{worst}(n/2)$: Time required to compare left and right sublist
- $+1$: one comparison made with middle element

$$C_{worst}(1) = 1.$$

Assume $n = 2^k$, it becomes.

$$C_{worst}(2^k) = C_{worst}\left(\frac{2^k}{2}\right) + 1$$

$$= C_{worst}(2^{k-1}) + 1.$$

Apply backward substitution method.

$$C_{worst}(2^k) = [C_{worst}(2^{k-2}) + 1] + 1$$

$$\because C_{worst}(2^{k-1}) = C_{worst}(2^{(k-1)-1}) + 1$$

$$= C_{worst}(2^{k-2}) + 1$$

$$= C_{worst}(2^{k-2}) + 2$$

.

.

.

$$= C_{worst}(2^{k-k}) + k$$

$$= C_{worst}(2^0) + k$$

$$= C_{worst}(1) + k$$

$$= 1 + k$$

$C_{worst} = 1 + k$ (where $n = 2^k$, $k = \log_2 n$)

$= 1 + \log_2 n$ for $n > 1$.

∴ Worst case Complexity is $O(\log_2 n)$

Also avg case complexity is $O(\log_2 n)$

Best case complexity is $O(1)$

MERGE SORT:

* Merge Sort uses divide and conquer strategy for sorting unordered arrays.

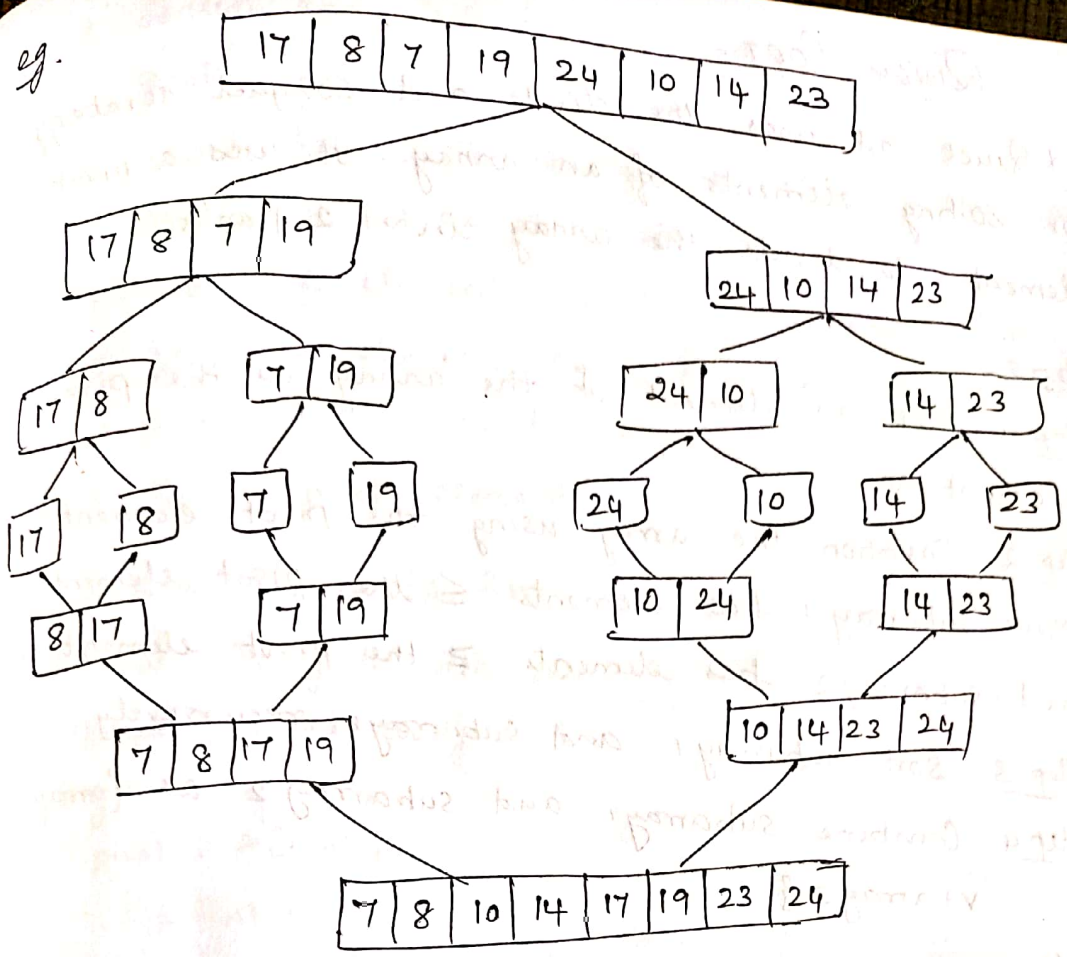
* The first phase of merge sort is to divide the array of numbers into 2 equal parts. If necessary, these sub-arrays are divided further.

* The conquer part involves sorting of sub-arrays recursively and the combine step involves combining the sorted subarrays to give a final sorted list.

```
Algorithm MergeSort (A[first ... last])
// I/P : Unsorted array A with first = 1 & last = n
// O/P : Sorted array
Begin
  if (first == last) then
    return A[first]
  else
    mid = (first + last)/2.
    for i ∈ {1, 2, .... mid}
      B[i] = A[i]
    end for
    for j ∈ {mid+1, ..... n}
      C[i] = A[i]
    end for
    MergeSort (B[1 ... mid])
    MergeSort (C[mid+1 ... n])
    merge (B, C, A)
End.
```

```
Algorith merge (B, C, A)
// I/P : 2 sorted arrays B & C
// O/P : Sorted array A.
Begin
    i = 1          // ptr i tracks B
    j = 1          // ptr j tracks C
    k = 1          // ptr k tracks A.
    m = length (B)
    n = length (C)
    while ((i<=m) && (j<=n)) do.
     if (B[i] < C[j]) then
        A[k] = B[i]
        i = i+1
     else
        A[k] = C[j]
        j = j+1
     end if
     k = k+1
    end while.
    if (i > m) then
     while (k <= m+n) do
       A[k] = C[j]
       k++; j++
     end while
    else if (j > n) then
      while (k <= m+n) do
        A[k] = B[i]
        k++; i++;
      end while
    end if
    return (A)
end.
```

eg.



Complexity Analysis :-

The recurrence equation for merge sort is given as,

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + n & \text{for } n \geq 2 \\ 1 & \text{for } n = 2 \\ 0 & \text{when } n < 2 \end{cases}$$

Using master's theorem.

$a = 2,\ b = 2,\ f(n) = n.$

Calculate $n^{\log_b a}$ & compare with $f(n)$

$n^{\log_2 2} = n.$

$f(n) = n$

$\therefore n^{\log_2 2} = f(n)$

$\therefore$ Case 2.

$$\theta(n^{\log_b a} \cdot \log n) = \theta(n^{\log_2 2} \cdot \log n)$$

Best case, average case, worst case $= \theta(n \log n)$

* Merge sort divides the array & merges it in all situations (ie) even if array is already sorted or not sorted, it performs division & merge
$\therefore$ Best case, average case & worst case are all same in merge sort.

QUICK SORT.

* Quick sort uses the divide and conquer strategy for sorting elements of an array. It uses a pivot element to divide the array into 2 parts.

Steps:

Step 1: Pick an element of the array as the pivot element V.

Step 2: Partition the array using the pivot element V. Now subarray 1 has elements $\leq$ the pivot element V and subarray 2 has elements $\geq$ the pivot element V.

Step 3: Sort subarray 1 and subarray 2 recursively.

Step 4: Combine subarray 1 and subarray 2 as {array 1 + V + array 2}

Algorithm

```
Algorithm quicksort (A, first, last)
// Input : Unsorted array A [first...last]
// Output : Sorted Array A
Begin
  if (first < last) then
     V = partition (A, first, last)
     quicksort (A, first, V-1)
     quicksort (A, V+1, last)
  end if
end.
```

Partitioning Algorithm: Hoare.

* Partitioning alg. uses 2 scans: one scan is from left to right and the other from right to left.

Steps:

Step 1: Choose the pivot element from array A, generally the first element.

Step 2 : Search from left to right for elements that are greater than pivot element.

Step 3 : Search from right to left for elements that are smaller than the pivot element.

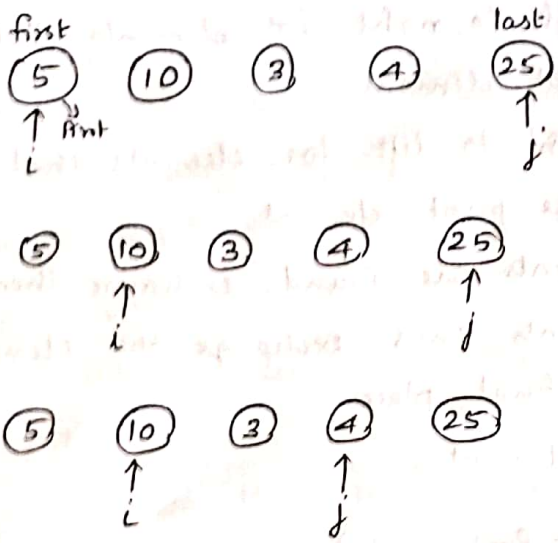
Step 4 : When 2 elements are found, exchange them.

Step 5 : When 2 elements cross, exchange the element such that it is in final place.

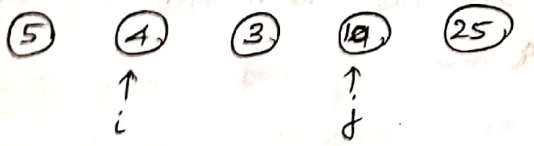
Step 6 : Return pivot element.

```
Algorithm Partition (A, first, last)
// Input : Array A with elements 1 to n. First = 1 & last = n
// Output : Sorted array A
 Begin
   pivot = A[first]
    i = first + 1
    j = last
    flag = ~~flag~~ false
    predicate = true
    while (predicate) do
     while (i ≤ j) and (A[i] ≤ pivot) do
       i = i+1
     end while
     while (j ≥ i) and (A[j] ≥ pivot) do
       j = j-1
     end while
     if (j < i)
       break
     else
       A[i] ↔ A[j]
     end if
   end while
   A[first] ↔ A[j]
   return j
 End.
```

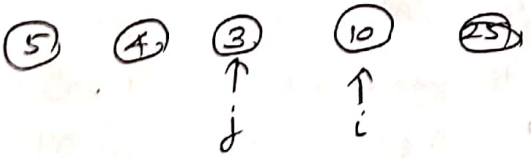
Eg:



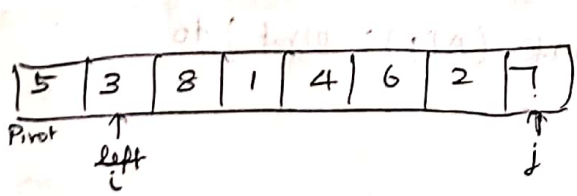
Swap A[i] & A[j]

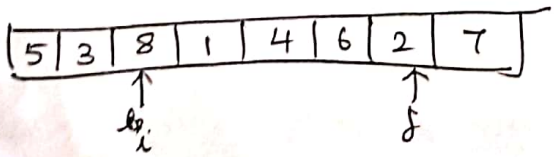


Swap pivot & A[j]

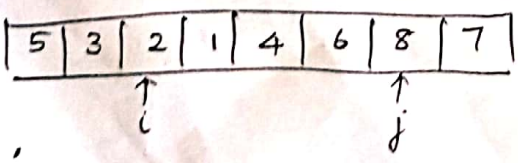


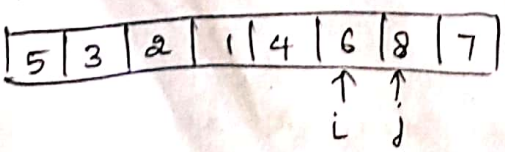
eg 2

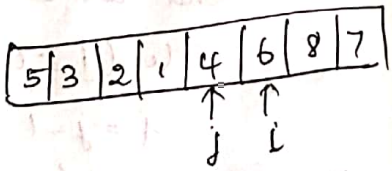




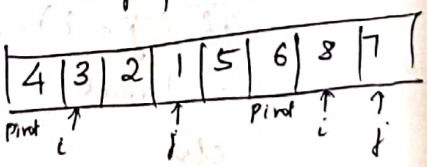
Swap

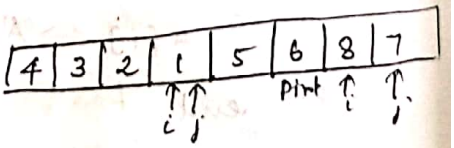




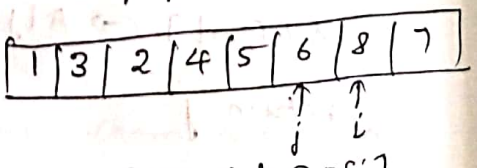


Swap pivot & A[j]

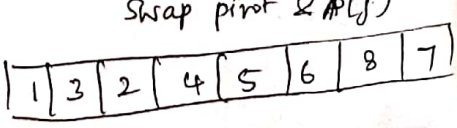




Swap pivot & A[j]



Swap pivot & A[j]



Complexity Analysis

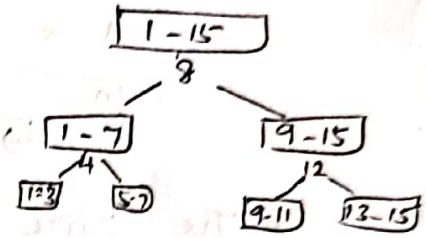
Best case complexity analysis:

Average case

* Best-case quicksort is a scenario where partition element is placed exactly in the middle of the array. It occurs when the pivot element partitions the list evenly.

* The recurrence eqn. is

$$T(n) = T\left(\frac{n}{2}\right) + n.$$



Using master's theorem, the complexity of best case is $T(n) \in \Theta(n \log n)$.

Average case complexity is $T(n) \in \Theta(n \log n)$

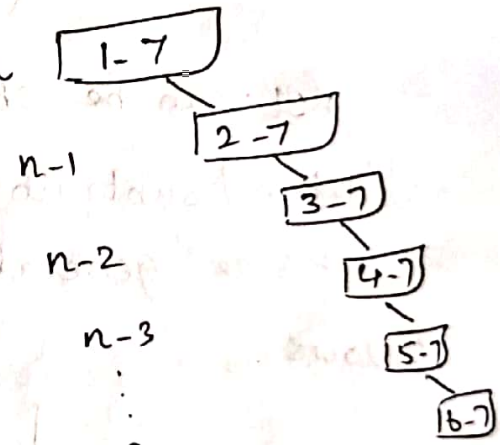
Worst case Complexity:

* In worst case, the partition is always done at first or last of the list.

eg. (5) 8, 10, 12, 14, 18, 15

Pivot = j, i

(5)	8	10	12	14	18	15



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

$$= \frac{n^2 + n}{2}$$

Worst-case analysis = $O(n^2)$

STRASSEN MATRIX MULTIPLICATION:

* Let 2 matrices A and B are of dimensions $n \times n$. The product of 2 matrices A and B, $C = A \times B$.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} * B_{kj}$$

A × B

n×n n×n

↳ equal, able to multiply A and B.

* The time complexity of the traditional matrix multiplication algorithm is $O(n^3)$.

* Consider 2 matrices A and B of order 2×2.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} ; B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$C = A \times B = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

* It can be observed that 8 multiplication and four addition / subtraction operations are required.

* The general recurrence equation is given as follows:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 8T\left(\frac{n}{2}\right) + an^2 & \text{if } n \geq 2 \end{cases}$$

* Therefore, the complexity of algorithm is $\theta(n^3)$.

* Using algebra techniques, Strassen reduced the no. of multiplication from 8 to 7. The reduction of one multiplication provided a faster multiplication algorithm.

* The Strassen's matrix multiplication carried out using 7 multiplications.

$d_1 = (a_{11} + a_{22})(b_{11} + b_{22})$

$d_2 = (a_{21} + a_{22})\, b_{11}$

$d_3 = a_{11}(b_{12} - b_{22})$

$d_4 = a_{22}(b_{21} - b_{11})$

$d_5 = (a_{11} + a_{12})\, b_{22}$

$d_6 = (a_{21} - a_{11})(b_{11} + b_{12})$

$d_7 = (a_{12} - a_{22})(b_{21} + b_{22})$

* The elements of resulting product matrix C is,

$$c_{11} = d_1 + d_4 - d_5 + d_7$$

$$c_{12} = d_2 + d_4$$

$$c_{21} = d_3 + d_5$$

$$c_{22} = d_1 + d_3 - d_2 + d_6$$

$$C = A \times B = \begin{bmatrix} d_1 + d_4 - d_5 + d_7 & d_2 + d_4 \\ d_3 + d_5 & d_1 + d_3 - d_2 + d_6 \end{bmatrix}$$

* The Strassen technique can also be combined effectively with divide-and-conquer strategy.

Steps:

1: Divide a matrix of order $n \times n$ recursively till matrices of 2×2 order obtained.

2: Use Strassen's formula to carry out 2×2 multiplication.

3: Combine results to get final product matrix.

Algorithm Strassen (n, a, b, d)

Begin

if n = threshold, then compute

C = a x b in conventional manner

else

Partition a into 4 submatrices $a_{11}, a_{12}, a_{21}, a_{22}$

Partition b into 4 submatrices $b_{11}, b_{12}, b_{21}, b_{22}$

Strassen $(n/2, a_{11} + a_{22}, b_{11} + b_{22}, d_1)$

Strassen $(n/2, a_{21} + a_{22}, b_{11}, d_2)$

Strassen $(n/2, a_{11}, b_{12} - b_{22}, d_3)$

Strassen $(n/2, a_{22}, b_{21} - b_{11}, d_4)$

Strassen (n/2 , $a_{11}+a_{12}$, b_{22} , d_5)

Strassen (n/2 , $a_{21}-a_{11}$, $b_{11}+b_{12}$, d_6)

Strassen (n/2 , $a_{12}-a_{22}$, $b_{21}+b_{22}$, d_7)

end if

$$C = \begin{bmatrix} d_1+d_4-d_5+d_7 & d_3+d_5 \\ d_2+d_4 & d_1+d_3-d_2+d_6 \end{bmatrix}$$

return C

end.

Complexity Analysis:

* The recurrence equation for Strassen matrix is

$$T = \begin{cases} 1 & \text{if } n=1 \\ 7T\left(\frac{n}{2}\right)+n^2 & \text{if } n>1 \end{cases}$$

Solve using master theorem.

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$$

$$n^{\log_b a} > f(n)$$

$$n^{2.81} > n^2$$

Case 1: $T(n) = \Theta(n^{\log_b a})$

$$= \Theta(n^{\log_2 7})$$

$$= \Theta(n^{2.81})$$

Ex 1: Multiply following 2 matrices using Strassen method.

$$A = \begin{pmatrix} 2 & 5 \\ 5 & 2 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$a_{11}=2$, $a_{21}=5$, $a_{12}=5$, $a_{22}=2$

$b_{11}=1$, $b_{12}=0$, $b_{21}=0$, $b_{22}=1$

$d_1 = (a_{11}+a_{22})(b_{11}+b_{22}) = (2+2)(1+1) = (4)(2) = 8$

$d_2 = (a_{21}+a_{22})b_{11} = (5+2)1 = 7$

$d_3 = a_{11}(b_{12}-b_{22}) = 2(0-1) = -2$

$d_4 = a_{22}(b_{21}-b_{11}) = 2(0-1) = -2$

$d_5 = (a_{11} + a_{12})\, b_{22} = (2+5)\,1 = 7$

$d_6 = (a_{21} - a_{11})(b_{11} + b_{12}) = (5-2)(1+0) = 3$

$d_7 = (a_{12} - a_{22})(b_{21} + b_{22}) = (5-2)(0+1) = 3$

$c_{11} = d_1 + d_4 - d_5 + d_7 = 8 - 2 - 7 + 3 = 2$

$c_{12} = d_3 + d_5 = -2 + 7 = 5$

$c_{21} = d_2 + d_4 = 7 - 2 = 5$

$c_{22} = d_1 + d_3 - d_2 + d_6 = 8 - 2 - 7 + 3 = 2$

Therefore $C = \begin{pmatrix} 2 & 5 \\ 5 & 2 \end{pmatrix}$

Ex 2: Perform Conventional and Strassen multiplication for the following 2 matrices

$$A = \begin{bmatrix} 2 & 4 & 6 & 3 \\ 1 & 2 & 2 & 1 \\ 3 & 1 & 1 & 3 \\ 1 & 1 & 1 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ 2 & 1 & 1 & 2 \\ 3 & 1 & 1 & 3 \end{bmatrix}$$

Solution:
Conventional Method.

$$C = \begin{bmatrix} 2+4+12+9 & 2+4+6+3 & 2+4+6+3 & 2+8+12+9 \\ 1+2+4+3 & 1+2+2+1 & 1+2+2+1 & 1+4+4+3 \\ 3+1+2+9 & 3+1+1+3 & 3+1+1+3 & 3+2+2+9 \\ 1+1+2+3 & 1+1+1+1 & 1+1+1+1 & 1+2+2+3 \end{bmatrix}$$

$$= \begin{bmatrix} 27 & 15 & 15 & 31 \\ 10 & 6 & 6 & 12 \\ 15 & 8 & 8 & 16 \\ 7 & 4 & 4 & 8 \end{bmatrix}$$

* By Strassen method, a 4×4 matrix can be divided into 4 halves

$$A = \left[\begin{array}{cc|cc} 2 & 4 & 6 & 3 \\ 1 & 2 & 2 & 1 \\ \hline 3 & 1 & 1 & 3 \\ 1 & 1 & 1 & 1 \end{array}\right]$$

$A_{11} = \begin{bmatrix} 2 & 4 \\ 1 & 2 \end{bmatrix}$ $A_{12} = \begin{bmatrix} 6 & 3 \\ 2 & 1 \end{bmatrix}$

$A_{21} = \begin{bmatrix} 3 & 1 \\ 1 & 1 \end{bmatrix}$ $A_{22} = \begin{bmatrix} 1 & 3 \\ 1 & 1 \end{bmatrix}$

$$B = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 \\ \hline 2 & 1 & 1 & 2 \\ 3 & 1 & 1 & 3 \end{array}\right] \qquad B_{11} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad B_{12} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

$$B_{21} = \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix} \quad B_{22} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}$$

$d_1 = (A_{11} + A_{22})(B_{11} + B_{22})$

$= \left[\begin{pmatrix} 2 & 4 \\ 1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & 3 \\ 1 & 1 \end{pmatrix}\right]\left[\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 1 & 3 \end{pmatrix}\right]$

$= \begin{pmatrix} 3 & 7 \\ 2 & 3 \end{pmatrix}\begin{pmatrix} 2 & 3 \\ 2 & 4 \end{pmatrix}$

$= \begin{pmatrix} 6+14 & 9+28 \\ 4+6 & 6+12 \end{pmatrix}$

$= \begin{bmatrix} 20 & 37 \\ 10 & 18 \end{bmatrix}$

$d_2 = (A_{21} + A_{22}) \times B_{11}$

$= \begin{pmatrix} 4 & 4 \\ 2 & 2 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$

$= \begin{pmatrix} 8 & 8 \\ 4 & 4 \end{pmatrix}$

$d_3 = A_{11} \times [B_{12} - B_{22}]$

$= \begin{bmatrix} 0 & -6 \\ 0 & -3 \end{bmatrix}$

$d_4 = A_{22} \times (B_{21} - B_{11})$

$= \begin{pmatrix} 7 & 0 \\ 3 & 0 \end{pmatrix}$

$d_5 = (A_{11} + A_{12}) \times B_{22}$

$= \begin{pmatrix} 15 & 37 \\ 6 & 15 \end{pmatrix}$

$d_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$

$= \begin{pmatrix} -4 & -7 \\ -2 & -3 \end{pmatrix}$

$d_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

$= \begin{pmatrix} 15 & 15 \\ 3 & 3 \end{pmatrix}$

$$AB = \begin{bmatrix} d_1 + d_4 - d_5 + d_7 & d_3 + d_5 \\ d_2 + d_4 & d_1 + d_3 - d_2 + d_6 \end{bmatrix}$$

$d_1 + d_4 - d_5 + d_7 = \begin{pmatrix} 20 & 37 \\ 10 & 18 \end{pmatrix} + \begin{pmatrix} 7 & 0 \\ 3 & 0 \end{pmatrix} - \begin{pmatrix} 15 & 37 \\ 6 & 15 \end{pmatrix} + \begin{pmatrix} 15 & 15 \\ 3 & 3 \end{pmatrix}$

$= \begin{pmatrix} 27 & 15 \\ 10 & 6 \end{pmatrix}$

$d_3 + d_5 = \begin{pmatrix} 0 & -6 \\ 0 & -3 \end{pmatrix} + \begin{pmatrix} 15 & 37 \\ 6 & 15 \end{pmatrix} = \begin{pmatrix} 15 & 31 \\ 6 & 12 \end{pmatrix}$

$$d_2 + d_4 = \begin{pmatrix} 8 & 8 \\ 4 & 4 \end{pmatrix} + \begin{pmatrix} 7 & 0 \\ 3 & 0 \end{pmatrix} = \begin{pmatrix} 15 & 8 \\ 7 & 4 \end{pmatrix}$$

$$d_1 + d_3 - d_2 + d_6 = \begin{pmatrix} 20 & 37 \\ 10 & 18 \end{pmatrix} + \begin{pmatrix} 0 & -6 \\ 0 & -3 \end{pmatrix} - \begin{pmatrix} 8 & 8 \\ 4 & 4 \end{pmatrix} + \begin{pmatrix} -4 & -7 \\ -2 & -3 \end{pmatrix}$$

$$= \begin{pmatrix} 8 & 16 \\ 4 & 8 \end{pmatrix}$$

$$AB = \begin{bmatrix} 27 & 15 & 15 & 31 \\ 10 & 6 & 6 & 12 \\ 15 & 8 & 8 & 16 \\ 7 & 4 & 4 & 8 \end{bmatrix}$$

FINDING MAXIMUM AND MINIMUM ELEMENTS:

* In conventional method, the maximum & minimum elements can be obtained by tracking every element in an array.

```
Algorithm ConventionalMinMax (A, n)
Begin
    largest = A[i]
    smallest = A[i]
    i = 2
    while (i ≤ n) do
        if A[i] > largest then
            largest = A[i]
        end if
        if A[i] < smallest then
            smallest = A[i]
        end if
        i = i+1
    end while
return (largest, smallest)
end.
```

The time complexity is O(n) time as algorithm requires 2n comparisons.

Divide and Conquer Strategy for finding maximum and minimum

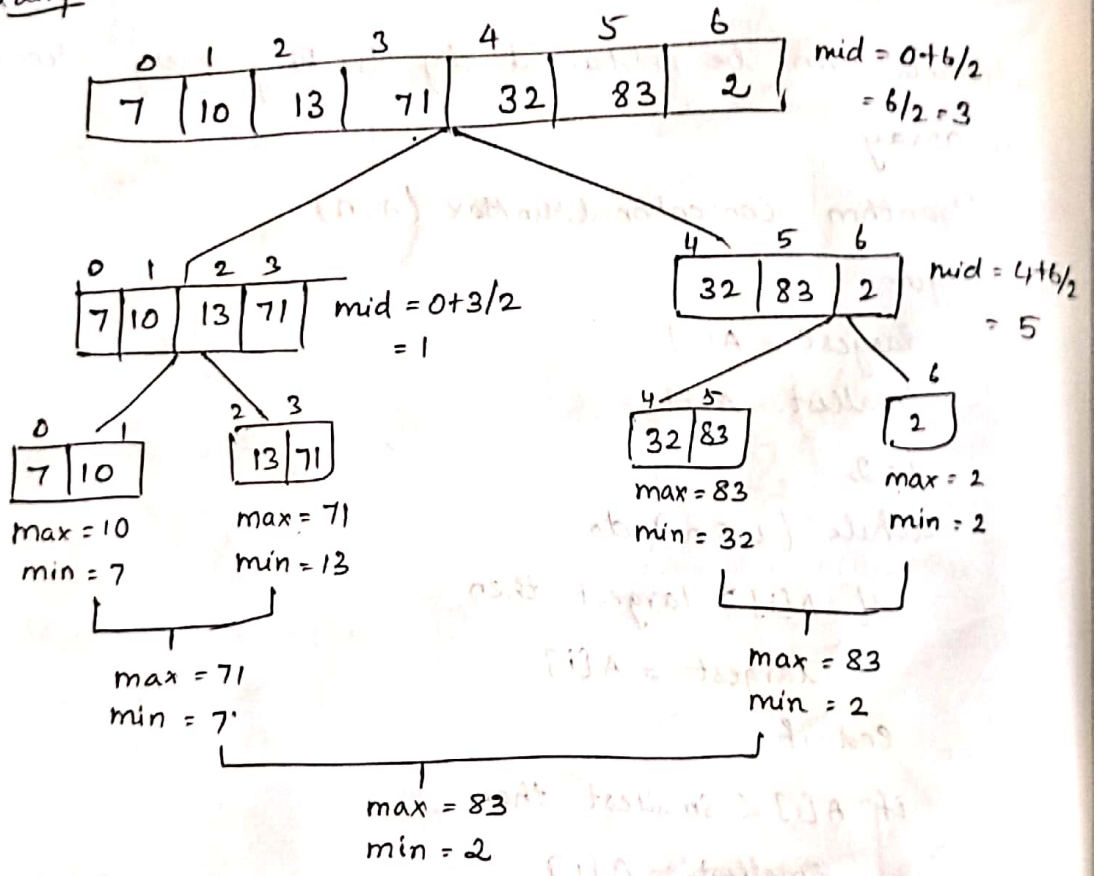
Procedure:

Step 1: Recursively divide an array A into subarrays

Step 2: Recursively find the maximum and minimum elements of the subarrays.

Step 3: Compare the minimum and maximum elements of these subarrays to get the global minimum & maximum of given array.

Example:



Algorithm MaxMin (low, high, max, min)

Begin

if (low == high)

max = min = a[low];

else

if (low == high - 1)

```
if (a[low] > a[high])
    max = a[low];
    min = a[high];
else
    max = a[high];
    min = a[low];
else
mid = (low + high) /2;
MaxMin (low, mid, max, min);
MaxMin (mid+1, high, max1, min1);
if (max1 > max)
    max = max1
if (min1 < min)
    min = min1
End.
```

Complexity Analysis:

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ 2 \cdot T(n/2) + 2 & \text{if } n>2. \end{cases}$$

The solution for the recurrence eqn can be obtained by assuming that $n = 2^k$. By repeated substitution,

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

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

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

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

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

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

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

$= 2^3 \cdot T(n/2^3) + 2^3 + 2^2 + 2^1$

⋮

Assuming $k-1$

$= 2^{k-1} \cdot T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + \ldots + 2^3 + 2^2 + 2^1$

$$T(n) = 2^{k-1} \cdot T\left(\frac{n}{2^{k-1}}\right) + \sum_{i=1}^{k-1} 2^i$$

$$= \frac{2^k}{2} \cdot T\left(\frac{n}{\left(\frac{2^k}{2}\right)}\right) + \sum_{i=1}^{k-1} 2^i$$

$$= \frac{2^k}{2} \cdot T\left(\frac{2n}{2^k}\right) + \sum_{i=1}^{k-1} 2^i$$

Let $2^k = n$

$$= \frac{n}{2} \cdot T\left(\frac{2n}{n}\right) + \sum_{i=1}^{k-1} 2^i$$

Geometrical Progression

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

2, 4, 8, ...

2nd element = r x 1st elem

$a = 2$: $4 = 2 \times 2$

$r = 2$ $\quad 8 = 2 \times 4$

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

$$2 \cdot 2^{k-1} - 2$$

$$2 \cdot \frac{2^k}{2} - 2 = 2^k - 2$$

$$T(n) = \frac{n}{2} \cdot T(2) + 2^k - 2$$

$$= \frac{n}{2} \cdot T(2) + n - 2$$

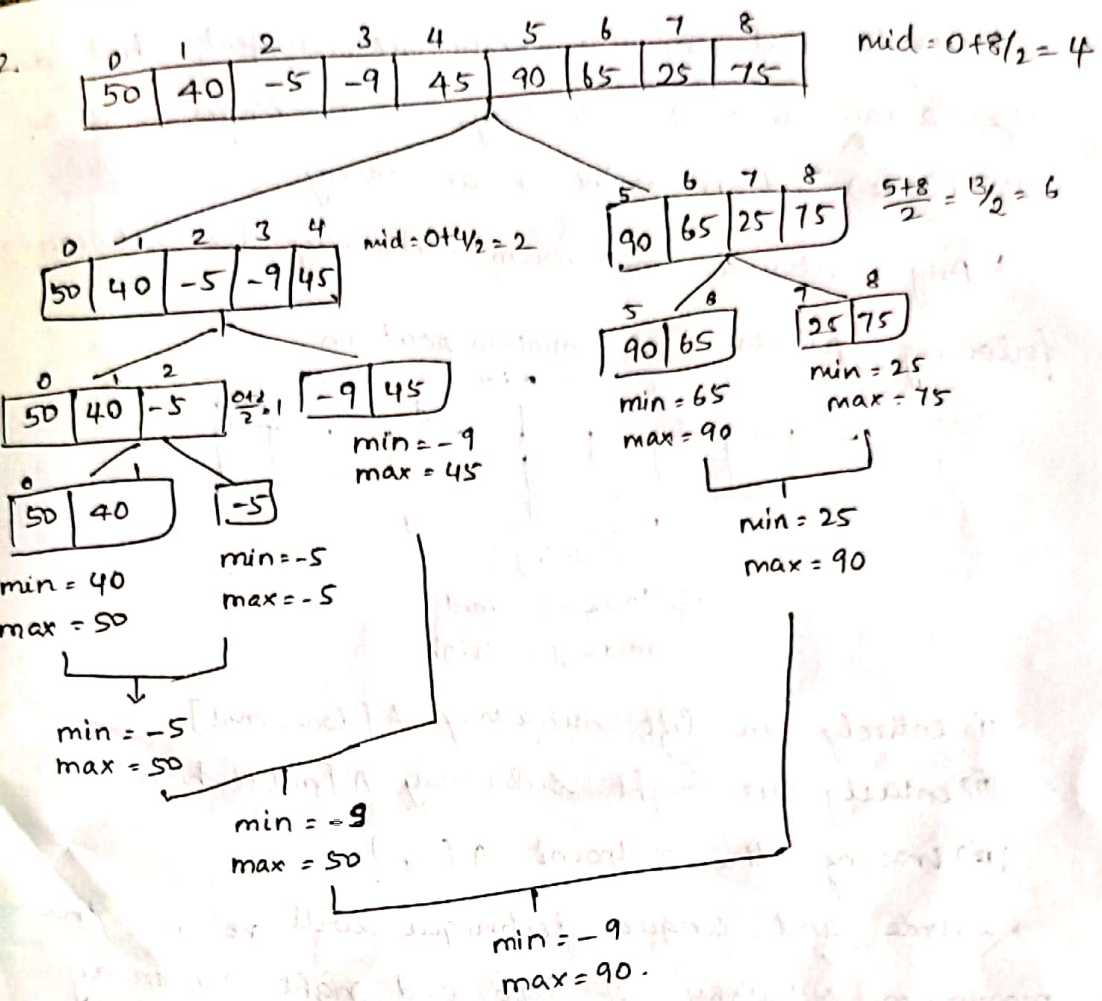
$$= \frac{n}{2}(1) + n - 2 \qquad [\because T(2) = 1 \text{ when } n = 2]$$

$$= \frac{n}{2} + n - 2$$

$$= \frac{n + 2n}{2} - 2$$

$$\boxed{T(n) = \frac{3n}{2} - 2.}$$

Eg 2.



MAXIMUM SUBARRAY PROBLEM:

*The problem is to find maximum subarray sum of given 1-D array. Array may contain positive and negative numbers.

Brute-force implementation:

*The idea is to generate all subarrays, compute sum of each subarray and return the subarray having the largest sum. It takes $O(N^2)$ time complexity

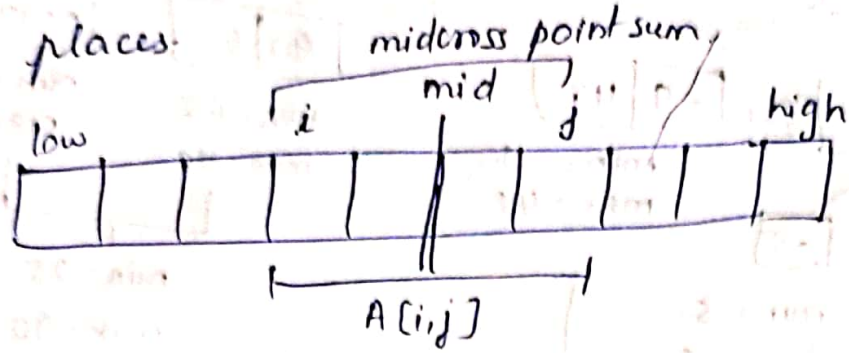
Divide and Conquer:

Alg:

1. Divide array in 2 halves.
2. Find maximum subarray sum in left half.
3. Find maximum subarray sum in right half.
4. Find maximum subarray sum which crosses midpoint.
5. Maximum of step 2, 3, 4 is the answer.

* Divide and conquer technique suggest that divide the array into 2 subarrays of as equal size as possible (ie) find midpoint of an array.

* Any contiguous maximum subarray lie between 3 following places.



low <= i < mid

mid < j <= high

(i) entirely in left subarray A[low, mid]

(ii) entirely in right subarray A[mid+1, high]

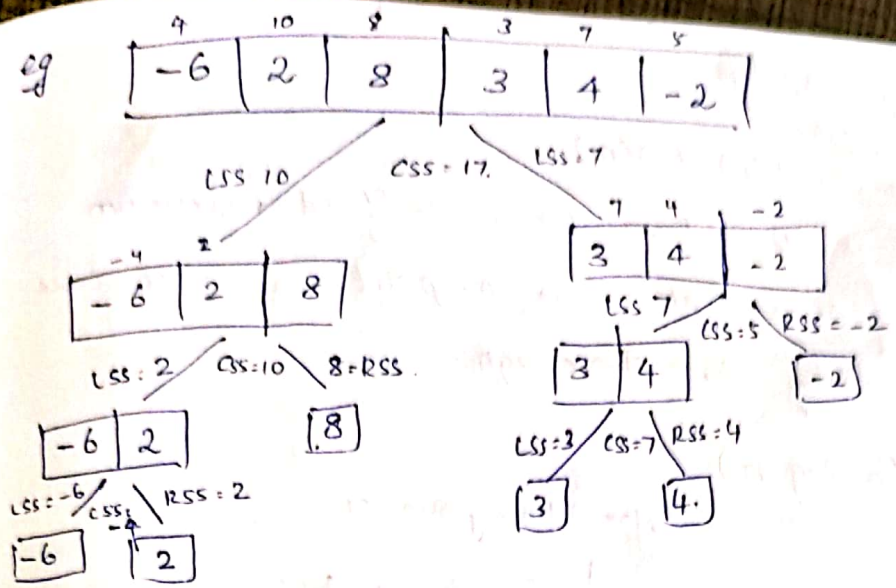
(iii) crossing the midpoint A[i,j].

* Divide and conquer technique will return the maximum subarray of left and right subarray.

* But we need maximum subarray the crosses the midpoint that contains elements from both left and right.

* To find this subarray, start from midpoint and traverse toward the left side till sum is increasing. Repeat the same on right side. The elements between the max. value in left and max. value in right form the subarray crossing the midpoint.

eg



Maximum Subarray Sum = 17.

Pseudocode :

```
LeftRight_Max_Subarray (A, low, high)
if high == low
    return A[low]
else
  mid = (low + high) / 2
  leftsum = LeftRight_Max_Subarray (A, low, mid);
  rightsum = LeftRight_Max_Subarray (A, mid+1, high);
  midsum = MidCross_Sum (A, low, mid, high);
  return maximum (leftsum, rightsum, midsum);
End.

Algorithm MidCross_Sum (A, low, mid, high)
Sum, leftsum, rightsum = 0.
for i = mid downto low do
    Sum = Sum + A[i]
    If Sum > leftsum then
       leftsum = Sum
Sum = 0
for i = mid+1 to high do
    Sum = Sum + A[i]
    if Sum > rightsum then
       rightsum = Sum;
return (leftsum + rightsum);
end.
```

Complexity Analysis

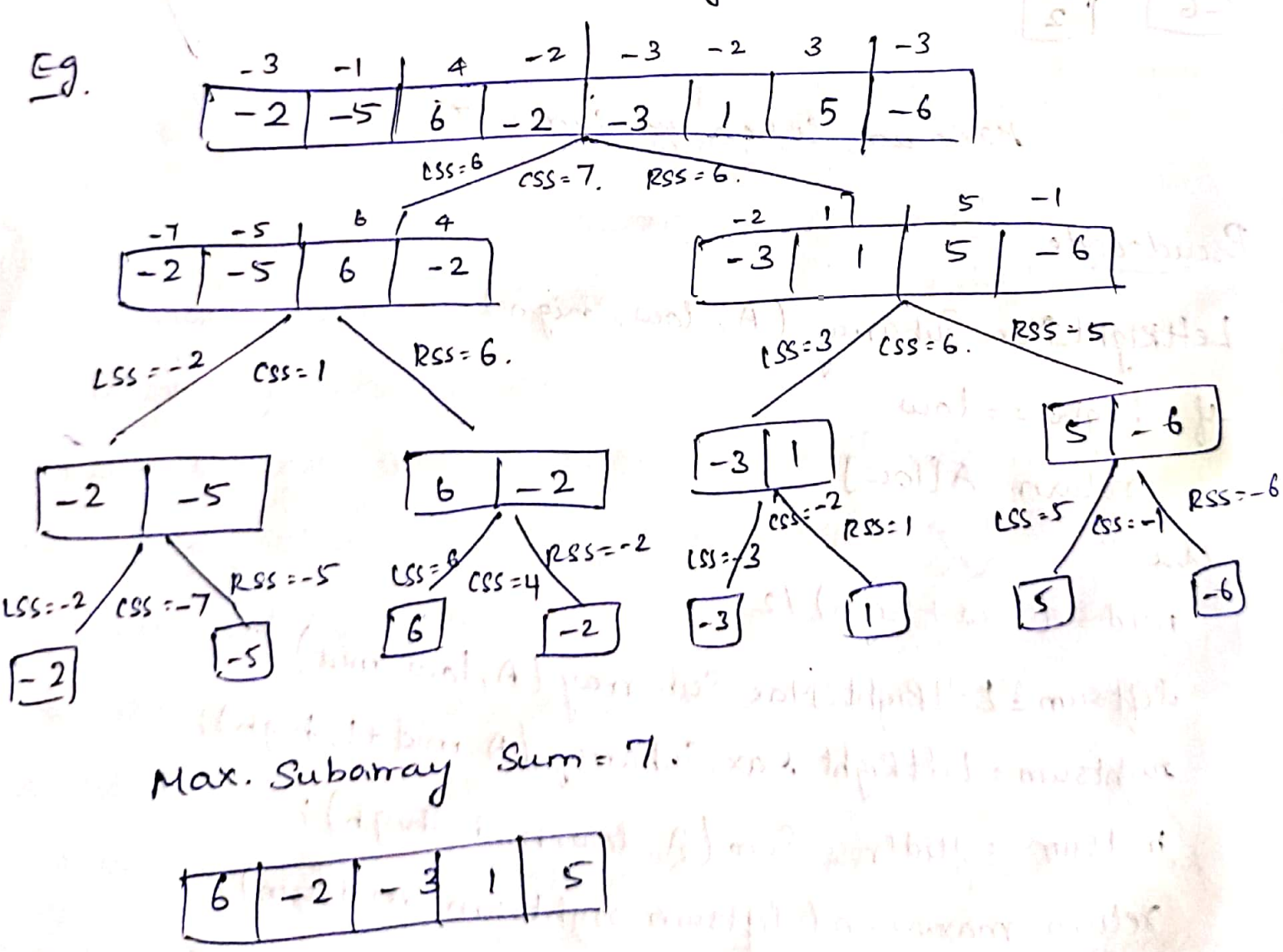
$T(n) = 2 \cdot T(n/2) + O(n)$

$O(n)$ is the time taken to find maximum subarray sum that crosses midpoint in worst case.

On solving the above eqn.

$O(n \log n)$.

Eg.



Max. Subarray Sum = 7.

6	-2	-3	1	5

Computational geometry problems — Deals with points, lines or polygons.

1) closest - pair problem

* simplest problem in computational geometry.
* objective - distance between two closest pair in space.
* Let us assume n points in 2D space. The distance b/w the points can be estimated by employing various distance measures.
  1) Euclidean distance
  2) city block distance
  3) chess board distance.

1) Euclidean distance.

Distance b/w 2 pts. with coordinate positions $p(x_1, y_1)$ and $q(x_2, y_2)$

$$d(p,q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

computation - expensive (2 squares and 1 square root)

2) city block distance.

Hermann Minkowski designed this distance measure.

Also known as taxicab distance, Manhattan distance and L_1 distance.

Distance is measured as

$$D(P,q) = |x_1 - x_2| + |y_1 - y_2|$$

3) chessboard distance - Chebyshev Distance.

Designed by Pafnutty Chebyshev so called as Chebyshev Distance

It is estimated as like the no. of moves of a king b/w 2 chessboard positions.

$$D(P,q) = \max(|x_1 - x_2|, |y_1 - y_2|)$$

The distance fn. is called a metric if the following properties are satisfied:

1. The distance b/w points p & q is well defined.
2. The distance b/w the point & itself is zero,
   ie, $D(P,P) = 0$
3. The relation $D(P,q) = D(q,P)$ is maintained.
4. The triangular inequality $D(P,q) + D(q,z) \leq D(P,z)$ is satisfied.

objective - find the closest pair among N given points. that have a least distance b/w them.

Brute force technique can be applied.

Steps:

1. for all given N points, find the Euclidean distance between a pair of points any other distance measure can also be used.
2. Find the points for which the distance is minimal.
3. Write the coordinates of the points for the distance is minimal.

Algorithm

```
Algorithm closestpair (A[1...N])
// Input - Array of N points.
// output - closest pairs and distance.
Begin
    d = -∞        // Initial Minimum.
    for all points Pi and Pj and (i ≠ j) do
        d1 = distance (Pi, Pj)    // find distance measure
                                  if the distance is small
                                  then updati.
    if d1 < d then
        close = min(d1, d2)
        close_x = Pi
```

close$_y$ = P_j // Mark the points.

end if

end for

end.

Complexity Analysis - $O(n^2)$.

~~Convex Hull~~

Problem Consider 2 pts. p(0,4) and q(2,3). Find the Euclidean, city block and chessboard distances.

solution.

$$D(P,q) = \sqrt{(0-2)^2 + (4-3)^2} = \sqrt{5}$$

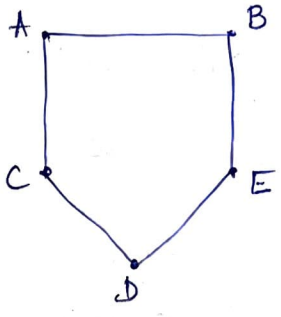
$$|(0-2)| + |(4-3)| = 3$$

$$\max(|0-2|, |4-3|) = \max(2,1) = 2$$

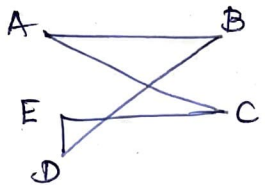
It can be observed that all distances measures yield different values.

Convex Hull problem.

* A set of points - called region.
* closed region ie, a set of points covered by a set of line segments called a polygon.



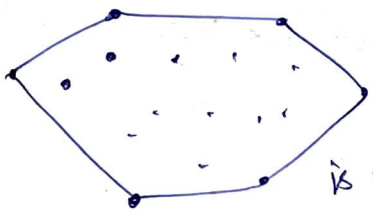
* Geometrical shapes - square, rectangle.. are polygons.
* A line segment connects two points in a plane.
* Meeting points of 2 line segments of a polygon is called a vertex.
* Vertex represents corner of a polygon. A, B, C, D, E are vertices



If the line crosses each other then the polygon called as reentrant polygon.

It is not considered for convex hull.

* A convex hull is the smallest polygon that encloses all the points.
* A polygon is called convex if it is not reentrant and any line that joins its two points lies within it.
* David Harrell, Algorithmics The Spirit of Computing quoted convex hull as constructing a fence around N sleeping tigers.



To construct the points extreme points should be constructed. Extreme point is not the middle point or any line segments.

Applications:

* Used for shape representation in image processing,
* shape analysis,
* statistics.
* Pattern recognition
* It is used to find extreme points which can be used to allot transmission towers in mobile applications.

Brute force algorithms for convex hulls:

The idea is to pick 2 points and connect them with a line. for all other points, say z, test whether z is inside the line. If so, add the point as a point of the polygon. Otherwise reconsider the vertex of the polygons.

Step 1: Take 2 points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ Define the equation:

$$ax + by = c$$

where $a = y_2 - y_1$

$b = x_1 - x_2$

$c = x_1 y_2 - y_1 x_2$

Step 2: connect the points with a line such that it divides the space in 2 halves:

$ax + by > c$ &

$ax + by < c$.

Step 3: check if all points lie on the same side of the line by $ax + by - c$. If all lie on same side then they are identified as part of the convex hull.

Step 4: Connect all the line segments to form a convex hull.

Algorithm ConvexHull(N)

// I/P - N points, O/P : ConvexHull.

Begin

for each pt. p & q do

if $p \neq q$ then compute the line segment

for every other point k ($k \neq p$ or q)

if each point is on one side of the line segment

label P_i, P_j as vertices of the convex hull.

end if

end for

end if

end for

end.

complexity Analysis.

$O(n^3)$

as there are $n(n-1)/2$ distance points and $(n-2)$ steps are required to find the sign.

Application of the closest - pair problem.

* Databases.
* clustering
* Pattern recognition. —— Nearest neighbourhood problem.

objective – given by n points.
choose the closest pair seperated by least distance.

In pattern recog., the feactures of objects are extracted and the objects are plotted as points.

K - nearest neighbour (KNN) algm.

k - integer.

if K=3, 3 closest neighbours are searched and based on the majority the object similarity is determined.