DAA – Unit II

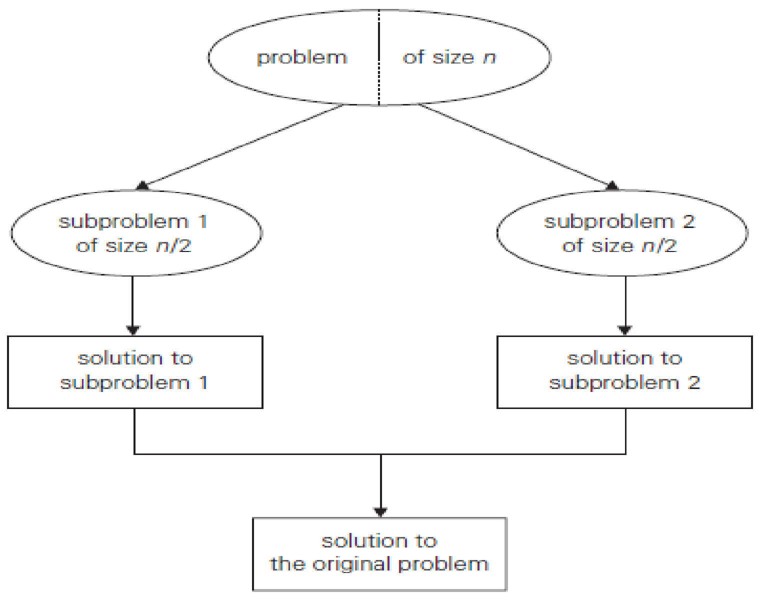
Dr.S.Prasanna Devi
Professor & Head, CSE Department

SYLLABUS

Duration (hour)		15	15	15	15	15
S-1	SLO-1	*Introduction-Algorithm Design*	*Introduction-Divide and Conquer*	*Introduction-Greedy and Dynamic Programming*	*Introduction to backtracking - branch and bound*	*Introduction to randomization and approximation algorithm*
	SLO-2	*Fundamentals of Algorithms*	*Maximum Subarray Problem*	*Examples of problems that can be solved by using greedy and dynamic approach*	*N queen's problem - backtracking*	*Randomized hiring problem*
S-2	SLO-1	*Correctness of algorithm*	*Binary Search*	*Huffman coding using greedy approach*	*Sum of subsets using backtracking*	*Randomized quick sort*
	SLO-2	*Time complexity analysis*	*Complexity of binary search*	*Comparison of brute force and Huffman method of encoding*	*Complexity calculation of sum of subsets*	*Complexity analysis*
S-3	SLO-1	*Insertion sort-Line count, Operation count*	*Merge sort*	*Knapsack problem using greedy approach*	*Graph introduction*	*String matching algorithm*
	SLO-2	*Algorithm Design paradigms*	*Time complexity analysis*	*Complexity derivation of knapsack using greedy*	*Hamiltonian circuit - backtracking*	*Examples*
S 4-5	SLO-1 SLO-2	*Lab 1: Simple Algorithm-Insertion sort*	*Lab 4: Quicksort, Binary search*	*Lab 7: Huffman coding, knapsack and using greedy*	*Lab 10: N queen's problem*	*Lab 13: Randomized quick sort*
S-6	SLO-1	*Designing an algorithm*	*Quick sort and its Time complexity analysis*	*Tree traversals*	*Branch and bound - Knapsack problem*	*Rabin Karp algorithm for string matching*
	SLO-2	*And its analysis-Best, Worst and Average case*	*Best case, Worst case, Average case analysis*	*Minimum spanning tree - greedy Kruskal's algorithm - greedy*	*Example and complexity calculation. Differentiate with dynamic and greedy*	*Example discussion*
S-7	SLO-1	*Asymptotic notations Based on growth functions.*	*Strassen's Matrix multiplication and its recurrence relation*	*Minimum spanning tree - Prims algorithm*	*Travelling salesman problem using branch and bound*	*Approximation algorithm*
	SLO-2	*O,O,Θ, ω, Ω*	*Time complexity analysis of Merge sort*	*Introduction to dynamic programming*	*Travelling salesman problem using branch and bound example*	*Vertex covering*
S-8	SLO-1	*Mathematical analysis*	*Largest sub-array sum*	*0/1 knapsack problem*	*Travelling salesman problem using branch and bound example*	*Introduction Complexity classes*
	SLO-2	*Induction, Recurrence relations*	*Time complexity analysis of Largest sub-array sum*	*Complexity calculation of knapsack problem*	*Time complexity calculation with an example*	*P type problems*
S 9-10	SLO-1 SLO-2	*Lab 2: Bubble Sort*	*Lab 5: Strassen Matrix multiplication*	*Lab 8: Various tree traversals, Krukshall's MST*	*Lab 11: Travelling salesman problem*	*Lab 14: String matching algorithms*
S-11	SLO-1	*Solution of recurrence relations*	*Master Theorem Proof*	*Matrix chain multiplication using dynamic programming*	*Graph algorithms*	*Introduction to NP type problems*
	SLO-2	*Substitution method*	*Master theorem examples*	*Complexity of matrix chain multiplication*	*Depth first search and Breadth first search*	*Hamiltonian cycle problem*
S-12	SLO-1	*Solution of recurrence relations*	*Finding Maximum and Minimum in an array*	*Longest common subsequence using dynamic programming*	*Shortest path introduction*	*NP complete problem introduction*
	SLO-2	*Recursion tree*	*Time complexity analysis-Examples*	*Explanation of LCS with an example*	*Floyd-Warshall Introduction*	*Satisfiability problem*
S-13	SLO-1	*Solution of recurrence relations*	*Algorithm for finding closest pair problem*	*Optimal binary search tree (OBST)using dynamic programming*	*Floyd-Warshall with sample graph*	*NP hard problems*
	SLO-2	*Examples*	*Convex Hull problem*	*Explanation of OBST with an example.*	*Floyd-Warshall complexity*	*Examples*
S 14-15	SLO-1 SLO-2	*Lab 3: Recurrence Type-Merge sort, Linear search*	*Lab 6: Finding Maximum and Minimum in an array, Convex Hull problem*	*Lab 9: Longest common subsequence*	*Lab 12: BFS and DFS implementation with array*	*Lab 15: Discussion over analyzing a real time problem*

Introduction to Divide and Conquer

$$a_0 + . . . + a_{n-1} = (a_0 + . . . + a_{\ \ -1}) + (a \ \ + . . . + a_{n-1}).$$



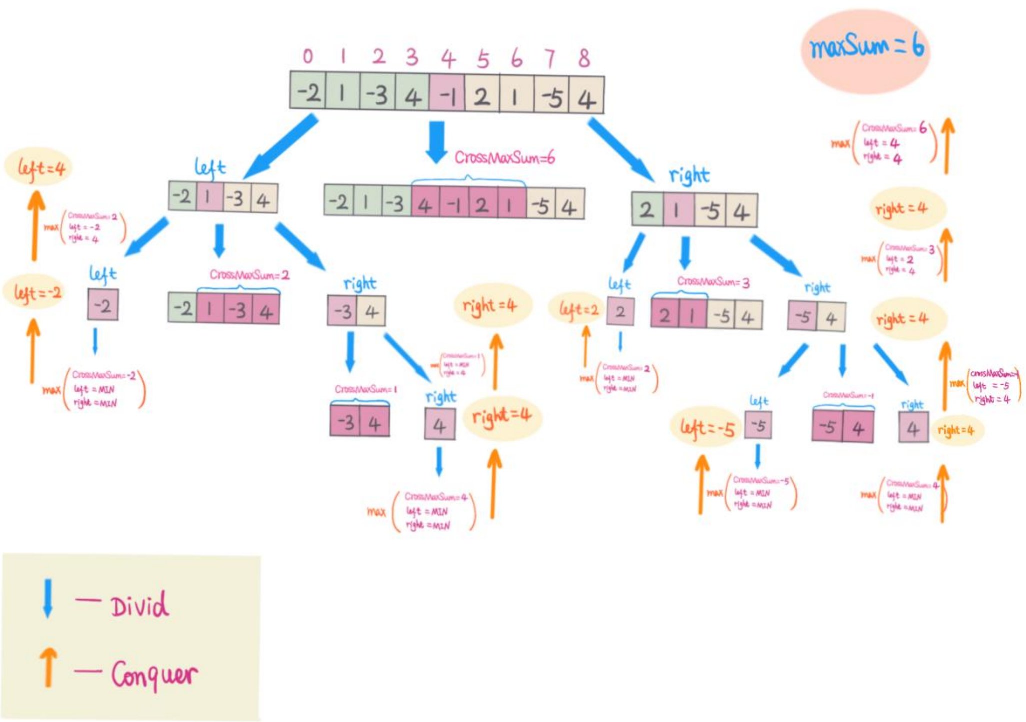
Maximum Sub-array problem

Maximum-subarray problem – divide-and-conquer algorithm

- What is the time complexity?
- **FindMaxSubarray**:

1. if(j<=i) return (A[i], i, j);
2. mid = floor(i+j);
3. (sumCross, startCross, endCross) = **FindMaxCrossingSubarray**(A, i, j, mid);
4. (sumLeft, startLeft, endLeft) = **FindMaxSubarray**(A, i, mid);
5. (sumRight, startRight, endRight) = **FindMaxSubarray**(A, mid+1, j);
6. Return the largest one from those 3

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



Binary Search & Complexity

```
Algorithm R_Binary_Search (l, h, key)
{
    if(l==h)
    {
        if (a[l]=key)
        return l;
        else
        return 0;
    }
    else
    {
        mid = (l+h)/2;
        if (a[mid] = key)
        return mid;
        if (key < a[mid])
        return R_Binary_Search(l, mid-1, key)
        else
        return R_Binary_Search(mid+1, h, key)
    }
}
```

First **m** is determined and the element at index **m** is compared to **x**.

5	13	27	30	50	57	63	76
l=0			m=3				r=7

As x > numbers[3], the element may reside in numbers[4...7]. Hence, the first half is discarded and the values of l, m and r are updated as shown below.

~~5~~	~~13~~	~~27~~	~~30~~	50	57	63	76
				L=4	m=5		r = 7

Now the element **x** needs to be searched in numbers[4...7]. As x > numbers[5], new values of l, m and r are updated in a similar way.

~~5~~	~~13~~	~~27~~	~~30~~	~~50~~	~~57~~	63	76
						l=m=6	r = 7

Now, comparing **x** with numbers[6], we get the match. Hence, the position of x = 63 have been determined.

Analysis

Linear search runs in ***O(n)*** time. Whereas binary search produces the result in ***O(log n)*** time

Let **T(n)** be the number of comparisons in worst-case in an array of **n** elements.

Hence,

$$T(n) = \begin{cases} 0 & if\, n = 1 \\ T(\frac{n}{2}) + 1 & otherwise \end{cases}$$

Using this recurrence relation $T(n) = log\, n$.

Therefore, binary search uses $O(log\, n)$ time.

Merge Sort and Time Complexity

ALGORITHM *Mergesort(A[0..n − 1])*

//Problem Description:*Sorts array A[0..n − 1] by recursive mergesort*

//Input: *An array A[0..n − 1] of orderable elements*

//Output: *Array A[0..n − 1] sorted in nondecreasing order*

if *n > 1*

copy A[0.. Floor(*n/2) to B[0.. n− 1]*

copy A[ceil(n/2)… n-1] to C[0.. n − 1]

Mergesort(B[0.. − 1])

Mergesort(C[0.. − 1])

Merge(B, C, A)

ALGORITHM *Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])*

//Problem Description: *Merges two sorted arrays into one sorted array*

//Input: *Arrays B[0..p − 1] and C[0..q − 1] both sorted*

//Output: *Sorted array A[0..p + q − 1] of the elements of B and C*

i ←0; j ←0; k←0

while *i <p* ***and*** *j <q* ***do***

if *B[i]≤ C[j]*

A[k]←B[i]; i ←i + 1

else *A[k]←C[j]; j ←j + 1*

k←k + 1

if *i = p*

copy C[j..q − 1] to A[k..p + q − 1]

else *copy B[i..p − 1] to A[k..p + q − 1]*

Analysis:

Assuming for simplicity that *n* is a power of 2, the recurrence relation for the number of key comparisons *C(n)* is

$C(n) = 2C(n/2) + Cmerge(n)$ for $n > 1$,

$C(1) = 0$.

As per Master theorem

$T(n) = \Theta(n^d \log n)$ if

Given data

a = 2, b = 2 f(n) = cn therefore $n^d = n^1$

=> d = 1

$a = b^d$ => $2 = 2^1$

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

Time complexity of merge sort for all cases is Θ(n log n))

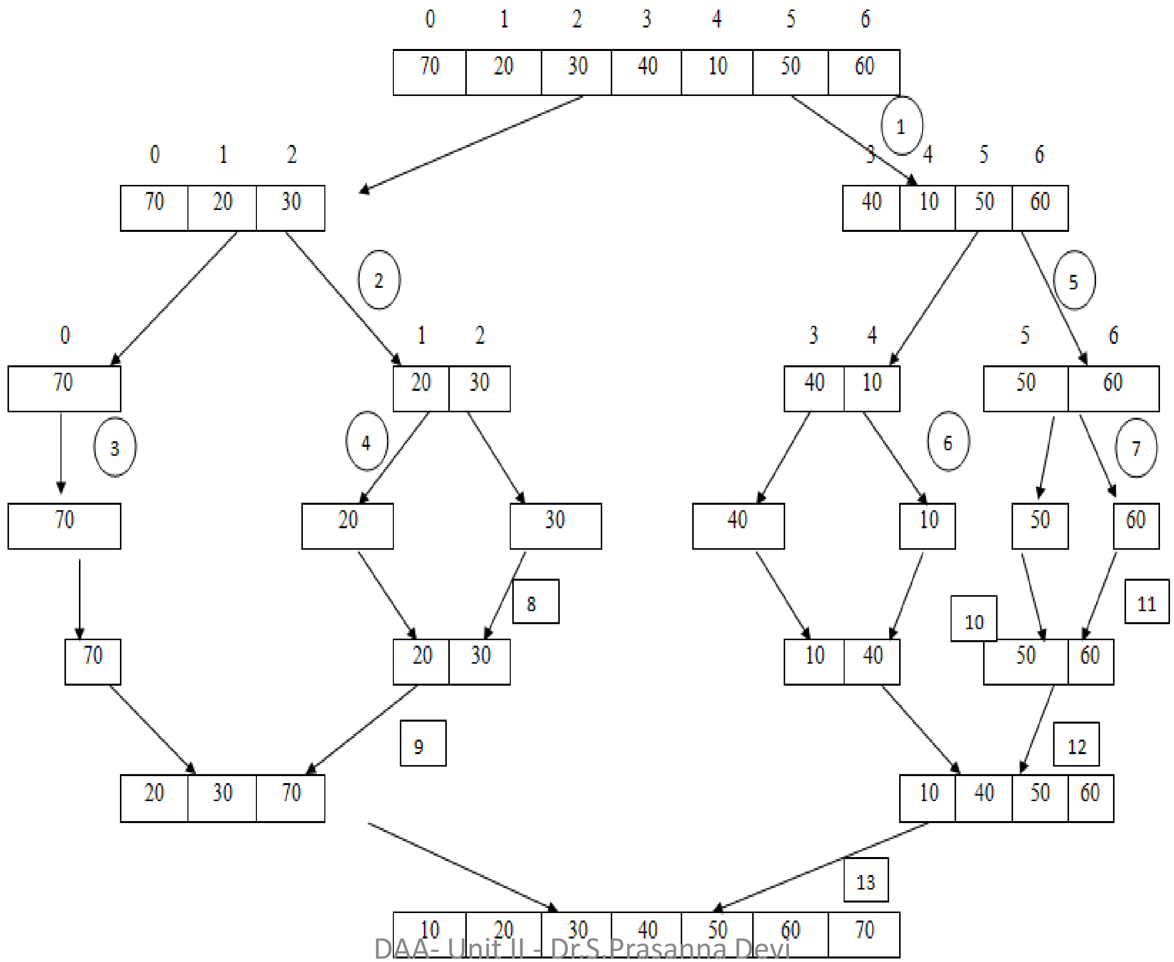
Merge Sort and Time Complexity

Sort the following set of elements using merge sort: 70, 20, 30, 40, 10, 50, 60.

Consider the list of elements as

70	20	30	40	10	50	60
0	1	2	3	4	5	6

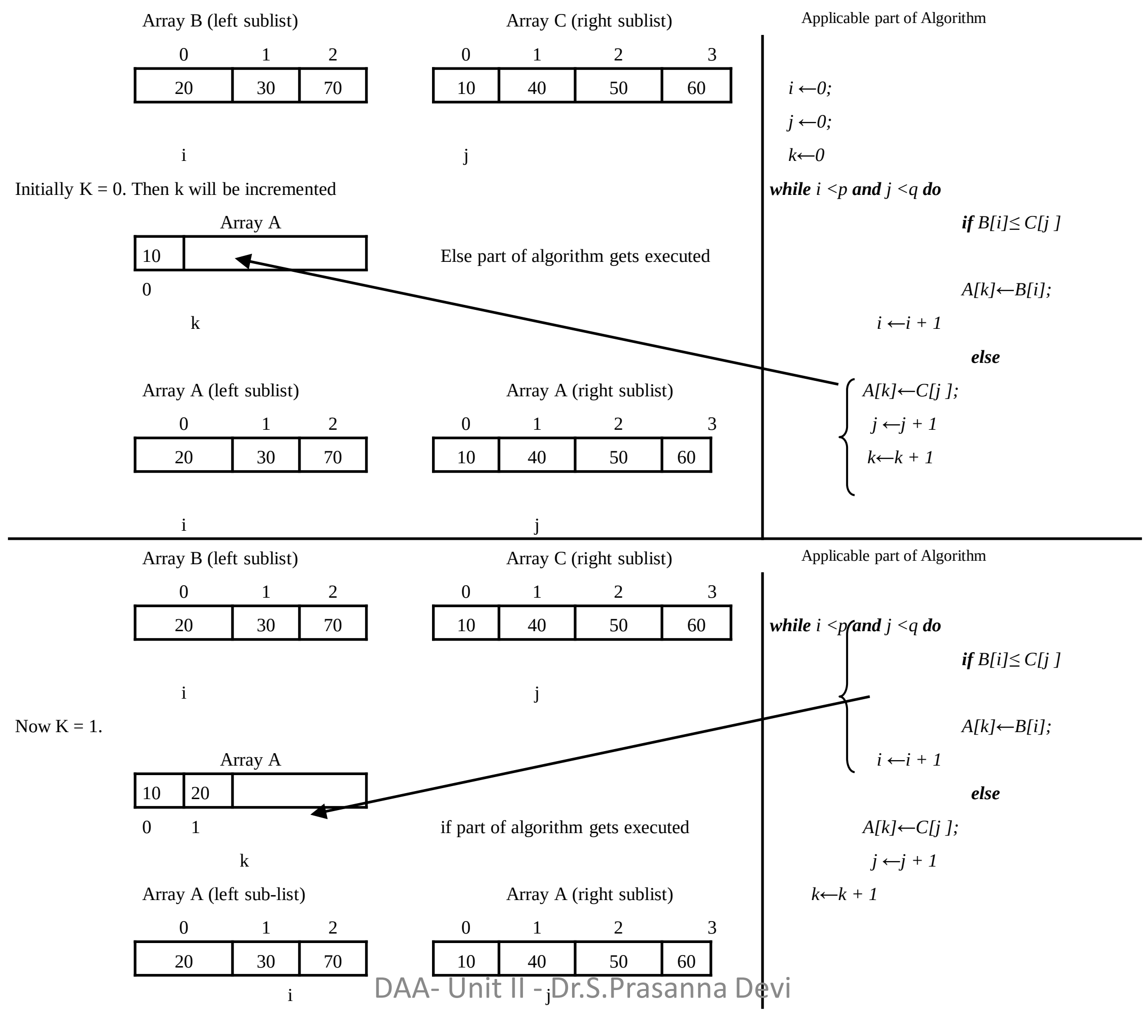
Low Mid high



Merge Sort and Time Complexity

Let us see the **combine** operation more closely with the help of some example.

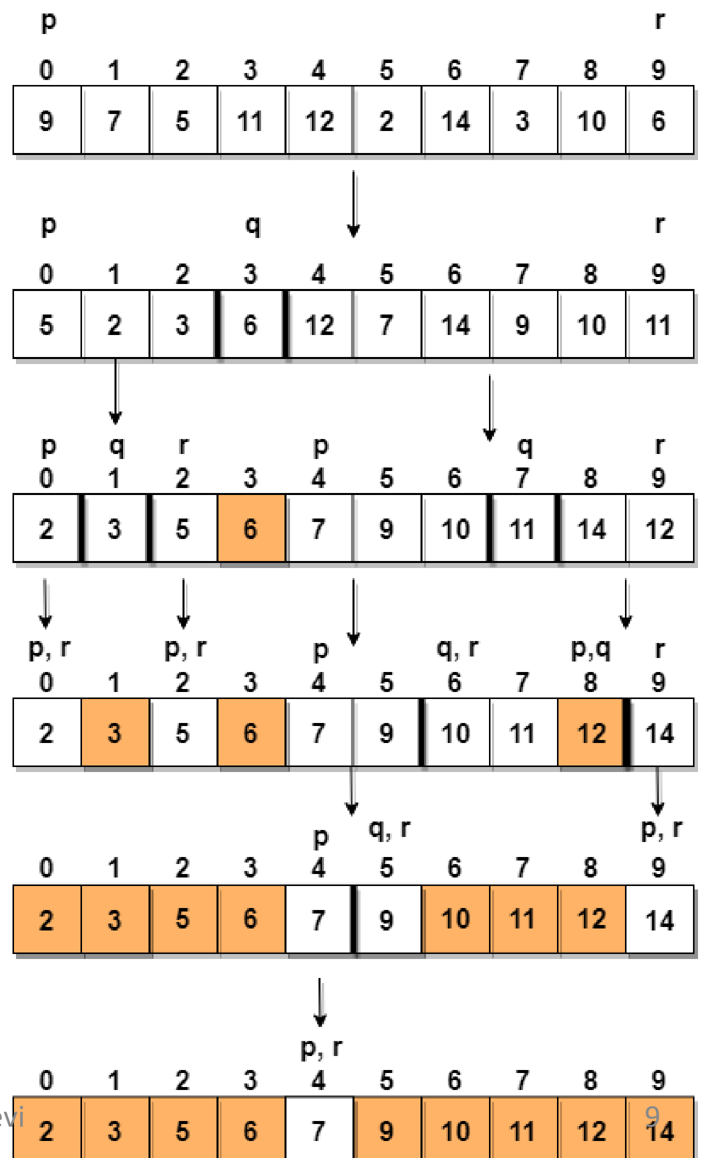
Consider that at some instance we have got two sub-lists 20, 30, 40, 70 and 10, 50, 60. then



Quick Sort & Time Complexity

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



- Quickest recognized sorting algorithm in practice:
- T(n) = 2T(n/2) + O(n)
- **Average case**: O(N log N)
- **Worst case**: O(N^2)

Strassen's Matrix Multiplication & Time Complexity

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{X}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \text{X} \underset{Y}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

X , Y and **C** are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

```
procedure MatrixMultiplication(A, B)
  input A, B n*n matrix
  output C, n*n matrix

begin
  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      C[i,j] = 0;
    end for
  end for

  for ( i = 0; i < n; i++)
    for ( j = 0; j < n; j++)
      for( k = 0; k < n; k++)
        C[i,j] = C[i,j] + A[i,k] * B[k,j]
      end for
    end for
  end for
end MatrixMultiplication
```

Analysis

$$T(n) = \begin{cases} c & if\, n = 1 \\ 7\,x\,T(\frac{n}{2}) + d\,x\,n^2 & otherwise \end{cases}$$ where *c* and *d* are constants

Using this recurrence relation, we get $T(n) = O(n^{log7})$

Hence, the complexity of Strassen's matrix multiplication algorithm is $O(n^{log7})$

Analysis using Naive Method:
T(n)= O(n^3)

Largest Sub-array sum & its complexity

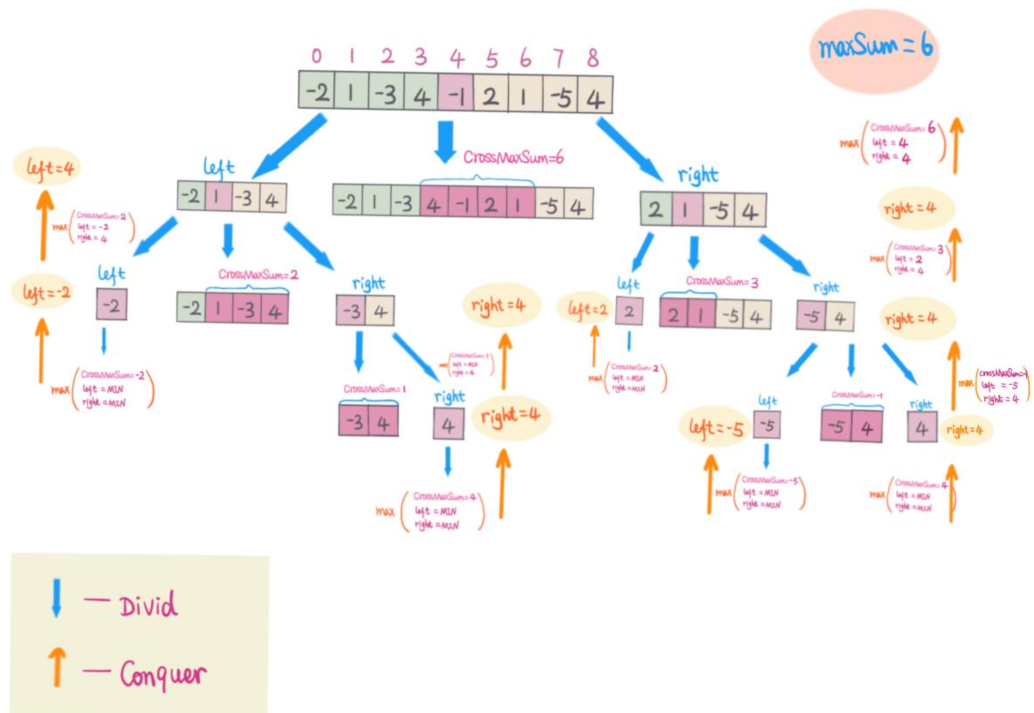
Same as Max sub array problem

Maximum-subarray problem – divide-and-conquer algorithm

- What is the time complexity?
- **FindMaxSubarray**:

1. if(j<=i) return (A[i], i, j);
2. mid = floor(i+j);
3. (sumCross, startCross, endCross) = **FindMaxCrossingSubarray**(A, i, j, mid);
4. (sumLeft, startLeft, endLeft) = **FindMaxSubarray**(A, i, mid);
5. (sumRight, startRight, endRight) = **FindMaxSubarray**(A, mid+1, j);
6. Return the largest one from those 3

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



Masters Theorem and Examples

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < Log_b a$ then $T(n) = \Theta(n^{Log_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = Log_b a$ then $T(n) = \Theta(n^c Log\ n)$

3.If $f(n) = \Theta(n^c)$ where $c > Log_b a$ then $T(n) = \Theta(f(n))$

Case 2 can be extended for $f(n) = \Theta(n^c Log^p n)$

If $f(n) = \Theta(n^c Log^p n)$ for some constant $k >= 0$ and $c = Log_b a$, then $T(n) = \Theta(n^c Log^{p+1} n)$

- **Examples of some standard algorithms whose time complexity can be evaluated using Master Method**
  Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $Log_b a$ is also 1. So the solution is $\Theta(n\ Logn)$
- Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $Log_b a$ is also 0. So the solution is $\Theta(Logn)$

Masters Theorem and Examples

1. $T(n) = 3T(n/2) + n^2 \Longrightarrow T(n) = \Theta(n^2)$ (Case 3)
2. $T(n) = 4T(n/2) + n^2 \Longrightarrow T(n) = \Theta(n^2 \log n)$ (Case 2)
3. $T(n) = T(n/2) + 2^n \Longrightarrow \Theta(2^n)$ (Case 3)
4. $T(n) = 2^nT(n/2) + n^n \Longrightarrow$ Does not apply (a is not constant)
5. $T(n) = 16T(n/4) + n \Longrightarrow T(n) = \Theta(n^2)$ (Case 1)
6. $T(n) = 2T(n/2) + n \log n \Longrightarrow T(n) = n \log^2 n$ (Case 2)
7. $T(n) = 2T(n/2) + n/\log n \Longrightarrow$ Does not apply (non-polynomial difference between $f(n)$ and $n^{\log_b a}$)
8. $T(n) = 2T(n/4) + n^{0.51} \Longrightarrow T(n) = \Theta(n^{0.51})$ (Case 3)
9. $T(n) = 0.5T(n/2) + 1/n \Longrightarrow$ Does not apply ($a < 1$)
10. $T(n) = 16T(n/4) + n! \Longrightarrow T(n) = \Theta(n!)$ (Case 3)
11. $T(n) = \sqrt{2}T(n/2) + \log n \Longrightarrow T(n) = \Theta(\sqrt{n})$ (Case 1)
12. $T(n) = 3T(n/2) + n \Longrightarrow T(n) = \Theta(n^{\lg 3})$ (Case 1)
13. $T(n) = 3T(n/3) + \sqrt{n} \Longrightarrow T(n) = \Theta(n)$ (Case 1)
14. $T(n) = 4T(n/2) + cn \Longrightarrow T(n) = \Theta(n^2)$ (Case 1)
15. $T(n) = 3T(n/4) + n \log n \Longrightarrow T(n) = \Theta(n \log n)$ (Case 3)
16. $T(n) = 3T(n/3) + n/2 \Longrightarrow T(n) = \Theta(n \log n)$ (Case 2)
17. $T(n) = 6T(n/3) + n^2 \log n \Longrightarrow T(n) = \Theta(n^2 \log n)$ (Case 3)
18. $T(n) = 4T(n/2) + n/\log n \Longrightarrow T(n) = \Theta(n^2)$ (Case 1)
19. $T(n) = 64T(n/8) - n^2 \log n \Longrightarrow$ Does not apply ($f(n)$ is not positive)
20. $T(n) = 7T(n/3) + n^2 \Longrightarrow T(n) = \Theta(n^2)$ (Case 3)
21. $T(n) = 4T(n/2) + \log n \Longrightarrow T(n) = \Theta(n^2)$ (Case 1)
22. $T(n) = T(n/2) + n(2 - \cos n) \Longrightarrow$ Does not apply

Find Max and Min of an array & its complexity

Naive Method Algorithm:

Max-Min-Element (numbers[])

```
max := numbers[1]
min := numbers[1]
 for i = 2 to n do
if numbers[i] > max
then max := numbers[i]
if numbers[i] < min
 then min := numbers[i]
return (max, min)
```

T(n) = O(n)

```
Algorithm: Max - Min(x, y)
if y - x ≤ 1 then
   return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))
else
   (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)
   (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)
return (max(max1, max2), min(min1, min2))
```

Analysis

Let ***T(n)*** be the number of comparisons made by $Max - Min(x, y)$, where the number of eleme

$n = y - x + 1$.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\, n > 2 \\ 1 & for\, n = 2 \\ 0 & for\, n = 1 \end{cases}$$

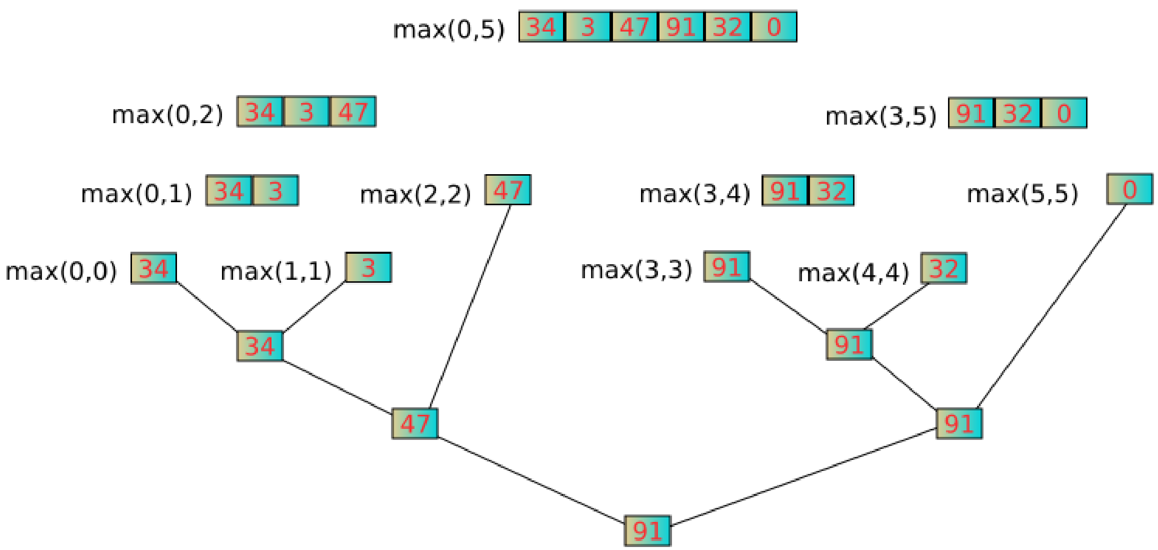
Let us assume that ***n*** is in the form of power of **2**. Hence, **n = 2^k** where **k** is height of the recursion tree.

So,

$$T(n) = 2.T(\frac{n}{2}) + 2 = 2.\left(2.T(\frac{n}{4}) + 2\right) + 2 \ldots\ldots = \frac{3n}{2} - 2$$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. Howe using the asymptotic notation both of the approaches are represented by **O(n)**.

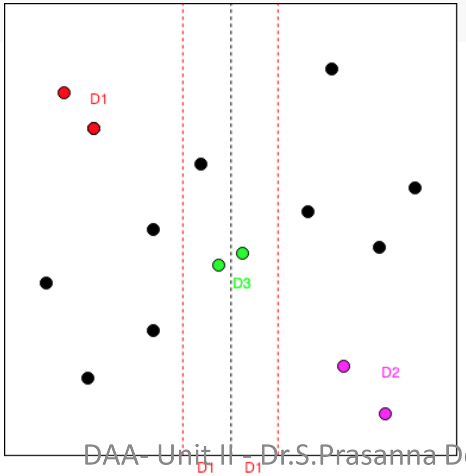
Find Max and Min of an array & its complexity




```
Algorithm: Max - Min(x, y)
if y - x ≤ 1 then
   return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))
else
   (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)
   (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)
return (max(max1, max2), min(min1, min2))
```

Finding Closest Pair problem & its complexity

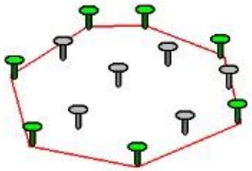
```
• Closest-Pair (S).
• If |S| = 1, output δ = ∞. If |S| = 2, output δ = |p2 - p1|.

Otherwise, do the following steps:

1. Let m = median(S).
2. Divide S into S1, S2 at m.
3. δ1 = Closest-Pair(S1).
4. δ2 = Closest-Pair(S2).
5. δ12 is minimum distance across the cut.
6. Return δ = min(δ1, δ2, δ12).

Recurrence is T(n) = 2T(n/2) + O(n), which solves to T(n) = O(n log n).
```



Problem : Refer CW

Convex Hull Problem & complexity

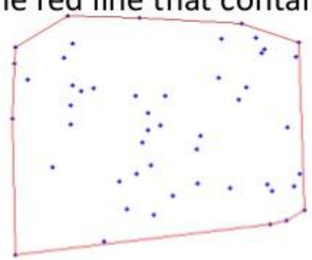
Convex Hull



- What is the convex hull ?

  It is the smallest convex set containing the points. Or we can also say it is a rubber band wrapped around the "outside" points.

  In the example below, the convex hull of the blue points is the red line that contains them.

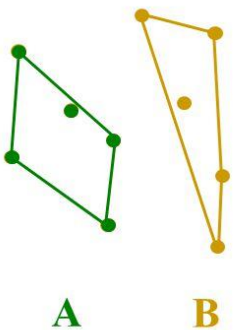


- In divide and conquer, method we divide the set of n points in 0(n) time into two subsets, one containing the leftmost [n/2] points, and one containing the right most [n/2] points, recursively compute the convex hull of the subsets, and then combine the hulls in 0(n) time. The running time is described by the familiar recurrence
- T(n) =2T(n/2) +o(n),

  so the divide and conquer method runs in o(n log n) time.

Convex Hull: Divide & Conquer

- Preprocessing: sort the points by x-coordinate
- Divide the set of points into two sets **A** and **B**:
  - **A** contains the left $\lfloor n/2 \rfloor$ points,
  - **B** contains the right $\lceil n/2 \rceil$ points
- Recursively compute the convex hull of **A**
- Recursively compute the convex hull of **B**
- Merge the two convex hulls



Problem: Refer CW

Question Bank

1.(i) Write a pseudo code for divide and conquer algorithm for merging two sorted arrays into a single sorted one. Explain with an example.

(ii) Set up and solve a recurrence relation for the number of key comparisons made the above pseudo code.

2. Design a recursive decrease by-one algorithm for sorting the n real numbers in an array with an examples and also determine the number of key comparisons and time efficiency of an algorithm.

3. Write a simple example to explain quick sort algorithm.

4.(i) Write an algorithm to sort a set of N numbers using insertion sort.

(ii) Trace the algorithm for the following set of numbers:20,35,18,8,14,41,3,39. 8 5.

5. (i) Write an algorithm to sort a set of N numbers using quick sort.

(ii) Trace the algorithm for the following set of numbers:20,35,18,8,14,41,3,39. 8 5.

6. Explain Strassen's Matrix multiplication algorithm with an example.
7. Explain Convex Hull problem and derive its complexity.
8. Write a recursive algorithm to find the max and min of an array. Derive its complexity.
9. Explain algorithm to find the closest pair problem and derive its complexity.
10. Write an algorithm to find the sum of a sub-array and to get the least element of a given sub array. What is the time complexity?
11. Problems on application of Master's Theorem to find the time complexity.