SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

UNIT II

Divide and Conquer

CO2: Solve problems using divide and conquer approach and analyze them.

SLO 1

- Understand the key concepts of divide and conquer strategy.
- Design a solution for Maximum Sub array Problem using divide and conquer strategy.

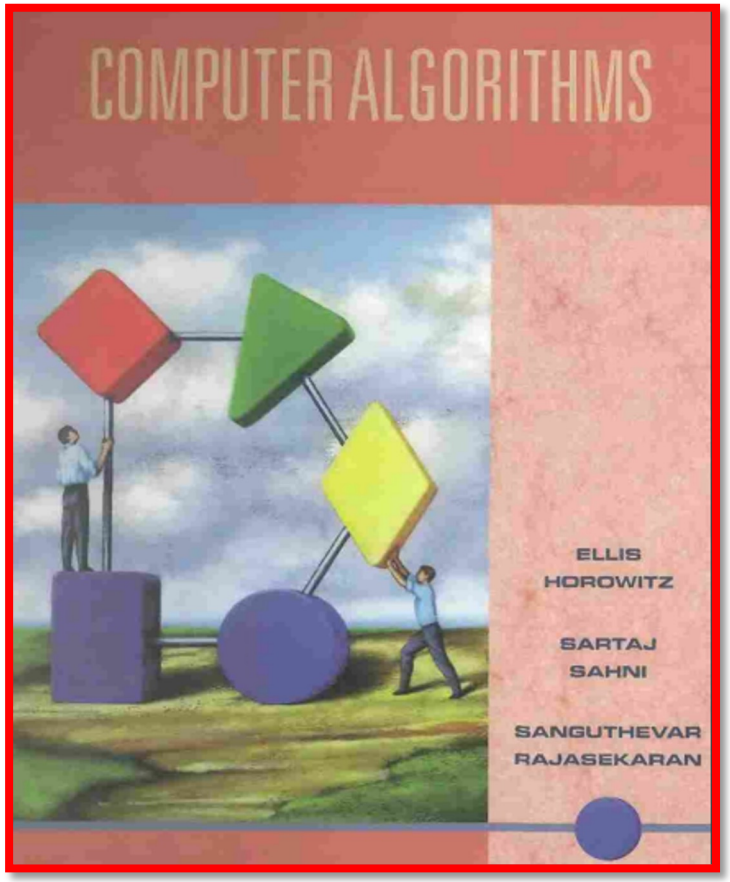
Course Code	18CSC204J	Course Name	DESIGN AND ANALYSIS OF ALGORITHMS

UNIT - II

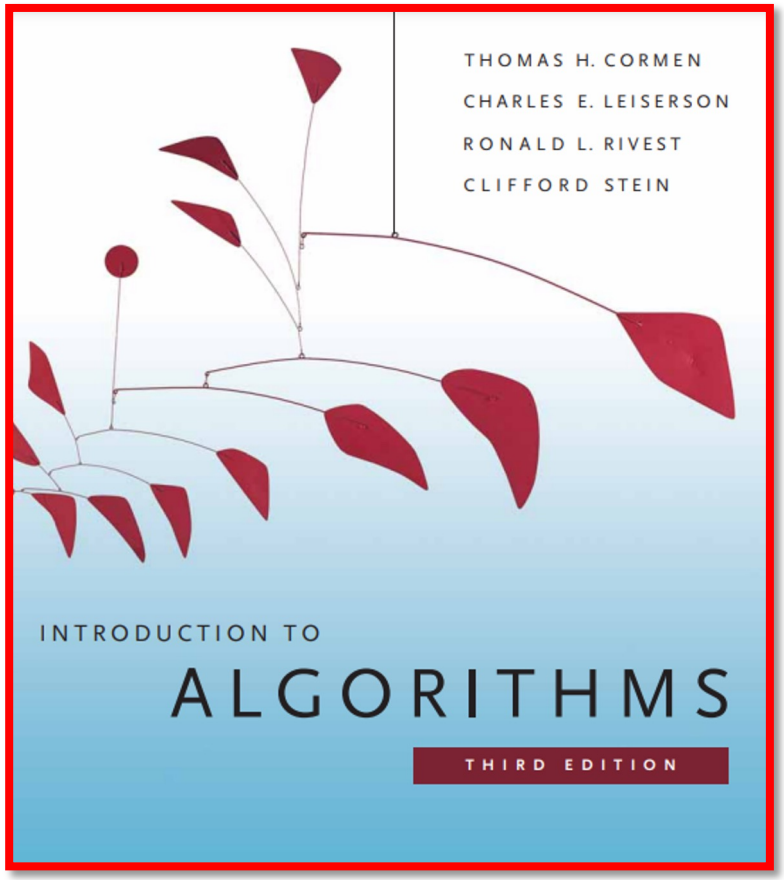
UNIT - II: DIVIDE AND CONQUER

Introduction-Divide and Conquer
Maximum Subarray Problem
Binary Search
Complexity of binary search
Merge sort
Time complexity analysis
Lab 4: Quicksort, Binary search
Quick sort and its Time complexity analysis
Best case, Worst case, Average case analysis

Strassen's Matrix multiplication and its recurrence relation
Time complexity analysis of Merge sort
Largest sub-array sum
Time complexity analysis of Largest sub-array sum
Lab 5: Strassen Matrix multiplication
Master Theorem Proof
Master theorem examples
Finding Maximum and Minimum in an array
Time complexity analysis-Examples
Algorithm for finding closest pair problem
Convex Hull problem
Lab 6: Finding Maximum and Minimum in an array, Convex Hull problem



Ellis Horowitz, Sartajsahni, Sanguthevar, Rajesekaran, Fundamentals of Computer Algorithms, Galgotia Publication, 2010



Thomas H Cormen, Charles E Leiserson, Ronald L Revest, Clifford Stein, Introduction to Algorithms, 3rd ed., The MIT Press Cambridge, 2014

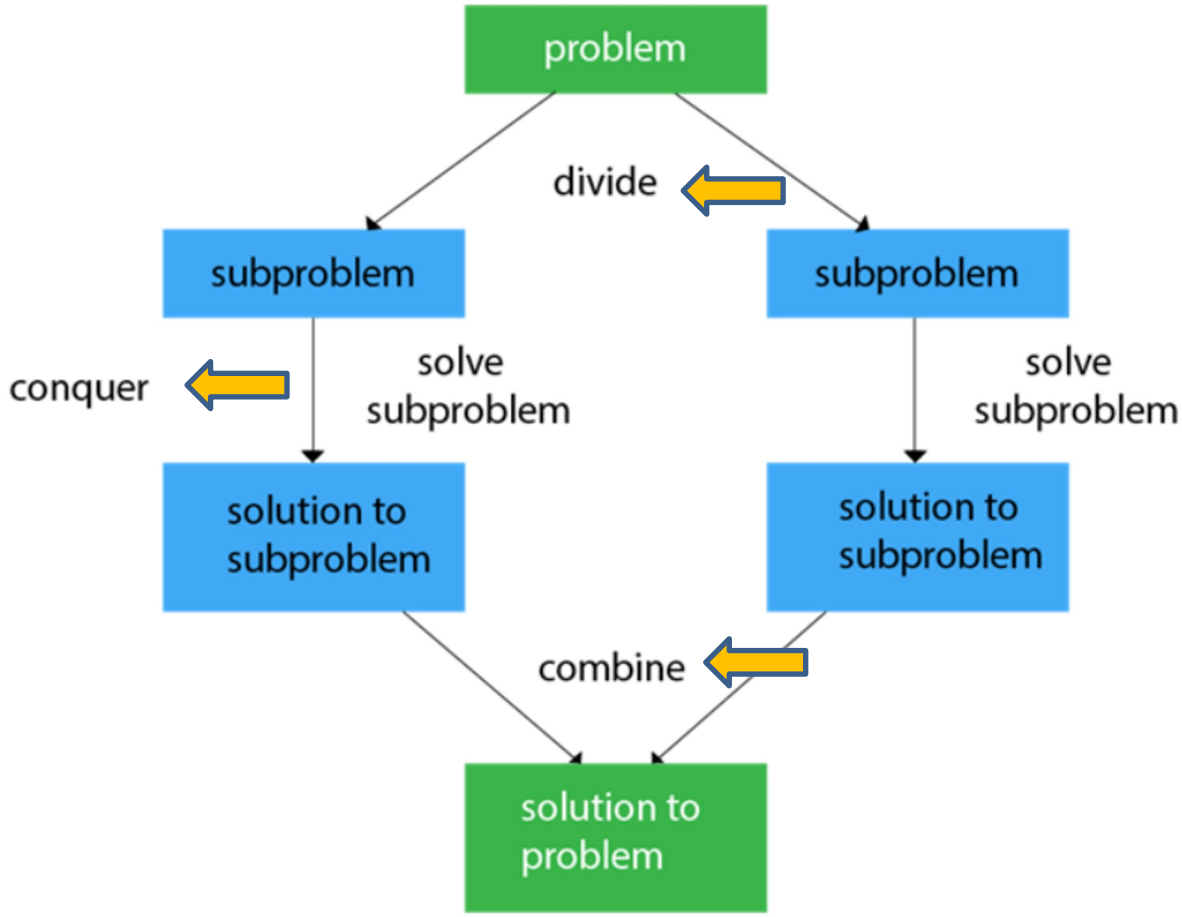
Divide and Conquer

General Method:

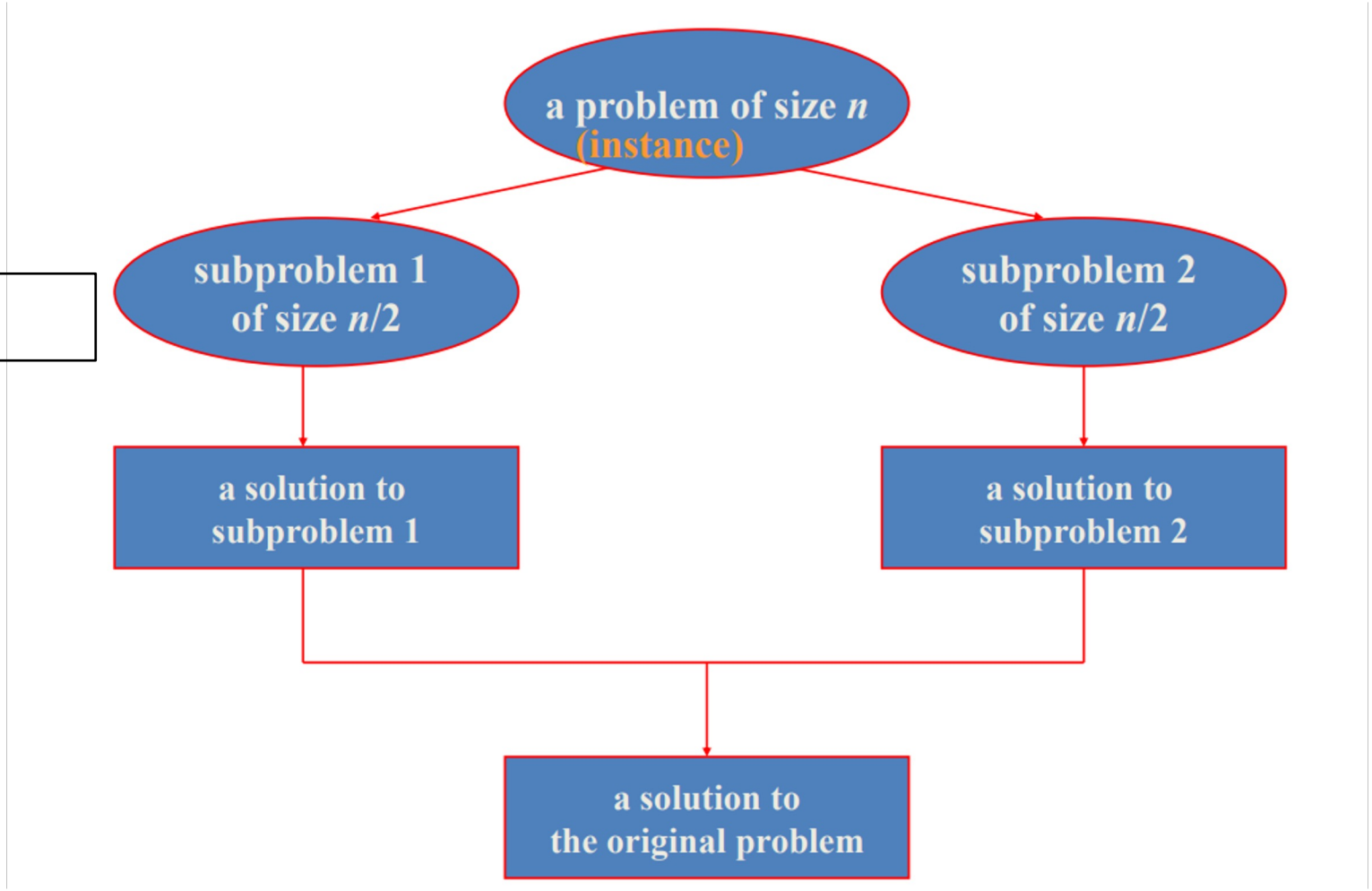
- Given a function to compute on 'n' inputs, divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, $1<k\leq n$, yielding 'k' **subproblems**.
- These subproblems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- If the subproblems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- For those cases the re-application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

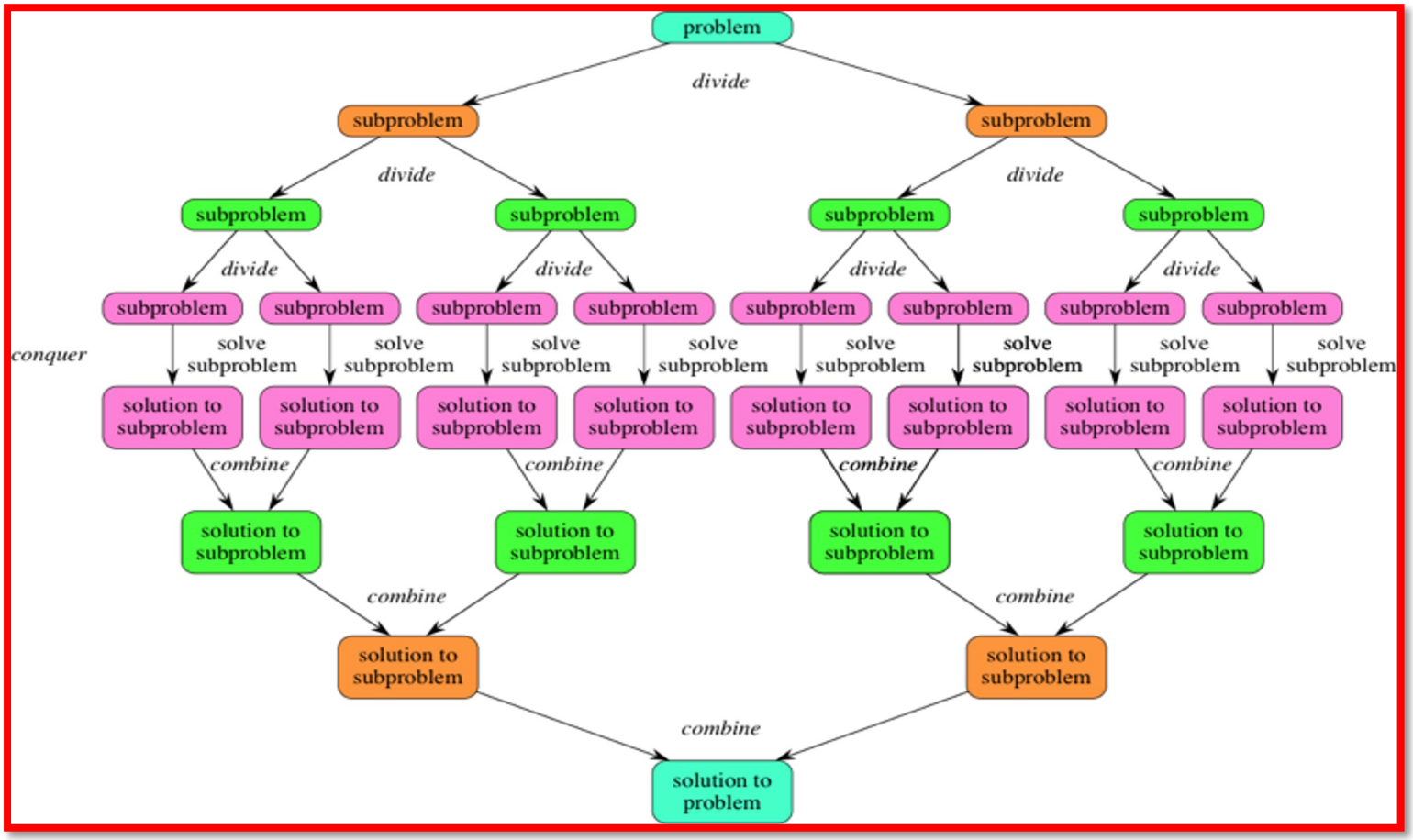
Divide and Conquer

- In divide and conquer method, the problem is divided into smaller sub-problems and then each problem is solved independently.
- When the subproblems is divided into even smaller sub-problems, it should reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved.
- The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.


problem
divide
subproblem
subproblem
conquer
solve
subproblem
solve
subproblem
solution to
subproblem
solution to
subproblem
combine
solution to
problem

SAME AS PROBLEM




problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
combine
combine
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
solution to subproblem
solution to subproblem
combine
solution to problem

Divide and Conquer

Control Abstraction of Divide and Conquer

```
Algorithm DAndC(P)
{
if small(P) then
    return S(P);
else
{
   divide P into smaller instance P1, P2……., Pk, k≥1;
   apply DAndC to each of these subproblems;
   return combine(DAndC(P1), DAndC(P2), ……., DAndC(Pk));
}
}
```

Divide and Conquer

Computing time of DAndC is:

$$T(n) = \begin{cases} g(n) & \text{n small} \\ T(n1) + T(n2) + \ldots + T(nk) + f(n) & \text{otherwise} \end{cases}$$

where

T(n) is the time for DAndC on any input of size n

g(n) is the time to compute the answer directly for small inputs

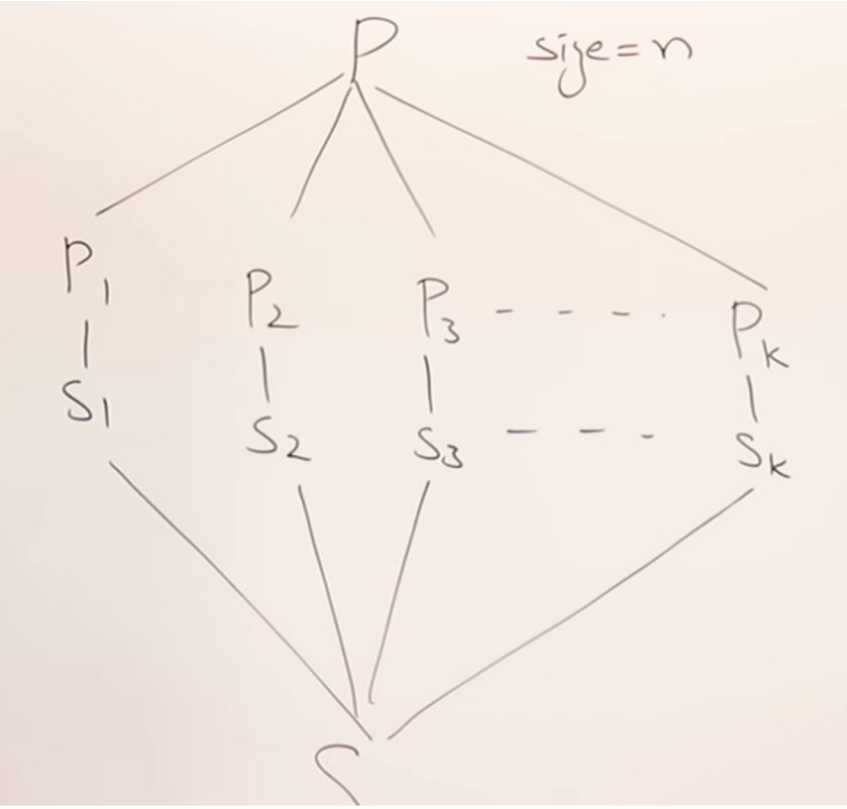
f(n) is the time for dividing P and combining the solutions to subproblems

Introduction

- In divide and conquer method, the problem is divided into smaller sub-problems and then each problem is solved independently.
- When the subproblems is divided into even smaller sub-problems, it should reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved.
- The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Divide and Conquer

- If a problem cannot be solved if it is big then solve it by breaking into subproblems.
- If the subproblem is also not able solve again break into sub problems.
- Repeat step 2 until the subproblem can be solved.

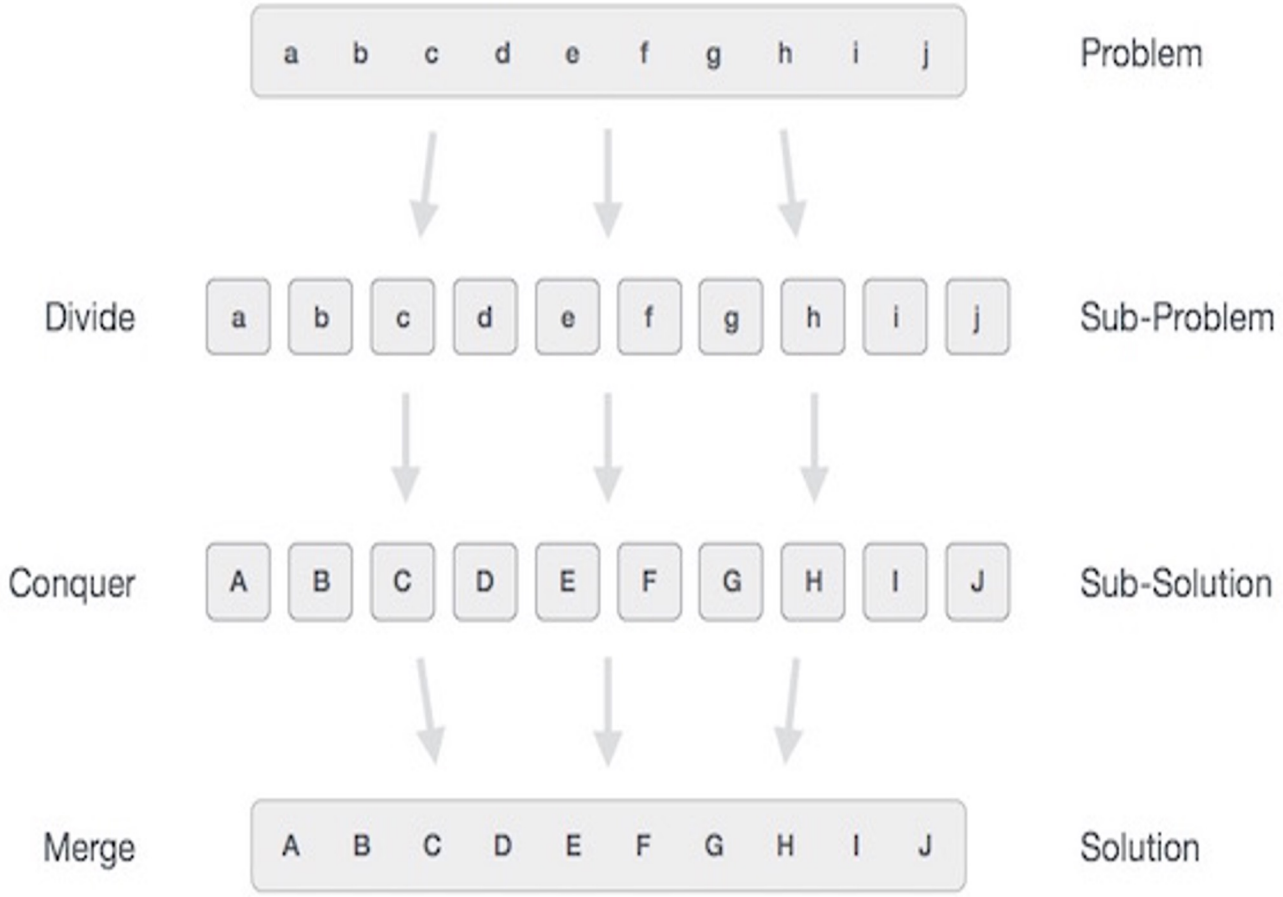


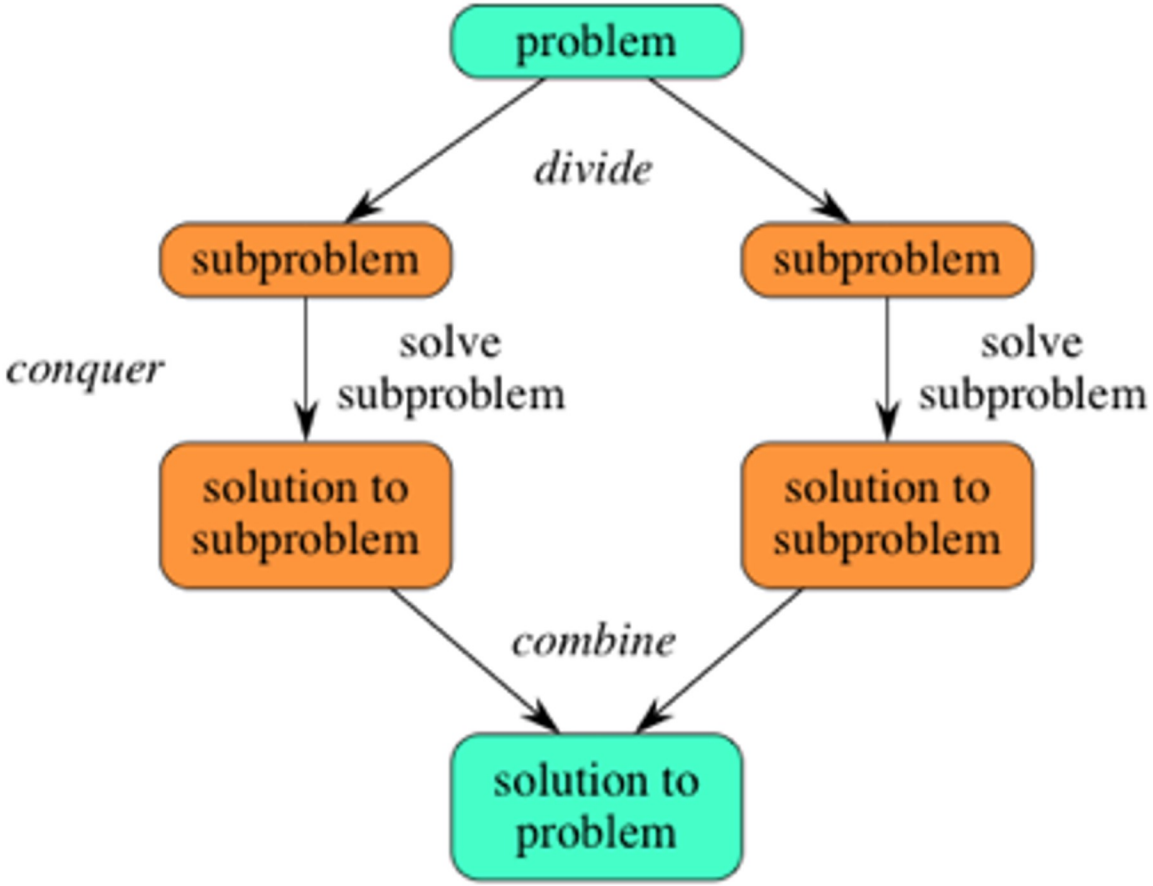
Divide and Conquer

- In divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

Divide-and-conquer method in a three-step process:




problem
divide
subproblem
subproblem
conquer
solve
subproblem
solve
subproblem
solution to
subproblem
solution to
subproblem
combine
solution to
problem


problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
combine
combine
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
solution to subproblem
solution to subproblem
combine
solution to problem

Control Abstraction of Divide and Conquer

```
DAC(Problem P)
{
    if(small(P)
    {
        Solve(P);
    }
    else
    {
        Divide P into P1, P2, P3,..Pk;
        apply  DAC(P1), DAC(P2)…DAC(Pk);
        combine(DAC(P1),DAC(P2)….DAC(Pk));
    }
}
```

General relation expressing Time Complexity of DaC Algorithms

computing time of DAndC is described by the recurrence relation

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

- T(n) is the time for DAndC on any input of size n and
- g(n) is the time to compute the answer directly for small inputs.
- The function f(n) is the time for dividing P and combining the solutions to sub problems.
- For divide and-conquer-based algorithms that produce sub problems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

General relation expressing Time Complexity of DaC Algorithms

The complexity of many divide-and-conquer algorithms is given by recurrences of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

Applications

- Maximum subarray Problem
- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix multiplication
- Largest sub-array sum
- Finding closest pair problem
- Convex Hull problem

The following are some standard algorithms that follows Divide and Conquer algorithm.

1.Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of the middle element, else recurs for the right side of the middle element.

2.Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3.Merge Sort is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4.Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.

5.Strassen's Algorithm is an algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3).

Maximum sub array Problem

- **Problem:** given an array of n numbers, find the (a) contiguous sub array whose sum has the largest value.

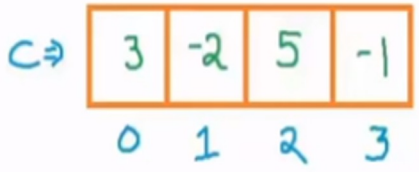
2	**4**	**1**	**5**	**8**

- Maximum sub array is 2+4+1+5+8= 20

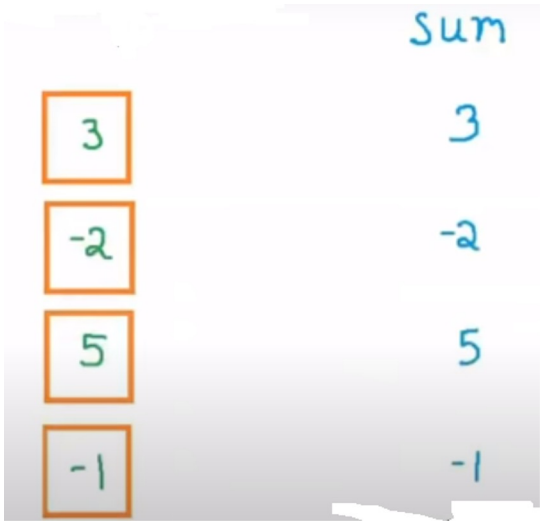
-2	**4**	**-1**	**-5**	**8**

- Sub array1,(-2+4-1=1)
- Sub array2, (4-1-5=-2)
- Sub array 3, (-1-5+8=2)
- Max(1,-2,2) is 2 sub array is (-1,-5,8)

A brute-force solution




```
for each subarray
{
   sum = total of elements in subarray
    if(sum > ans)
            ans = sum
}
```



Brute Force algorithm

```
int Maximum_Sum_Subarray(int arr[],int n)
{
    int ans = INT_MIN;
    for(int sub_array_size = 1;sub_array_size <= n; ++sub_array_size)
    {
        for(int start_index = 0;start_index < n; ++start_index)
        {
            if(start_index+sub_array_size > n)  //Subarray exceeds array bounds
                break;
            int sum = 0;
            for(int i = start_index; i < (start_index+sub_array_size); i++)
                sum+= arr[i];
            ans = max(ans,sum);
        }
    }
    return ans;
}
```

- $O(n^3)$

Algorithm 1: Brute force: $O(n^2)$.

input : a vector A of n numbers.
output: maximum subarray sum.

$S_{\max} \leftarrow -\infty$
for $i = 1$ **to** n **do**
    $sum \leftarrow 0$
    for $j = i$ **to** n **do**
        $sum \leftarrow sum + A[j]$
        $S_{\max} \leftarrow \max\{S_{\max}, sum\}$
    end
end
return $S_{\max}$

- $O(n^2)$

Max Subarray

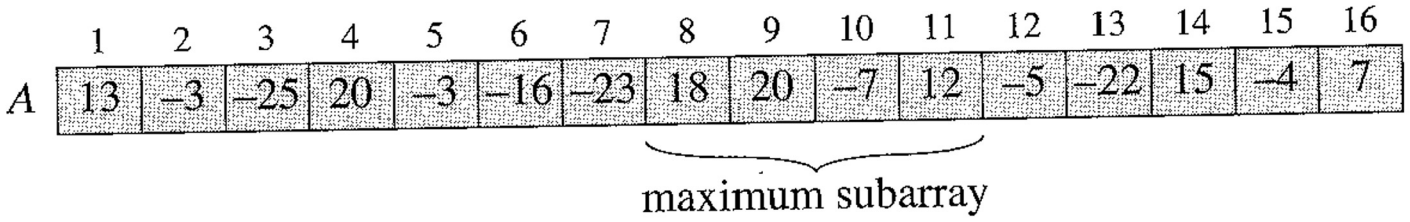


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

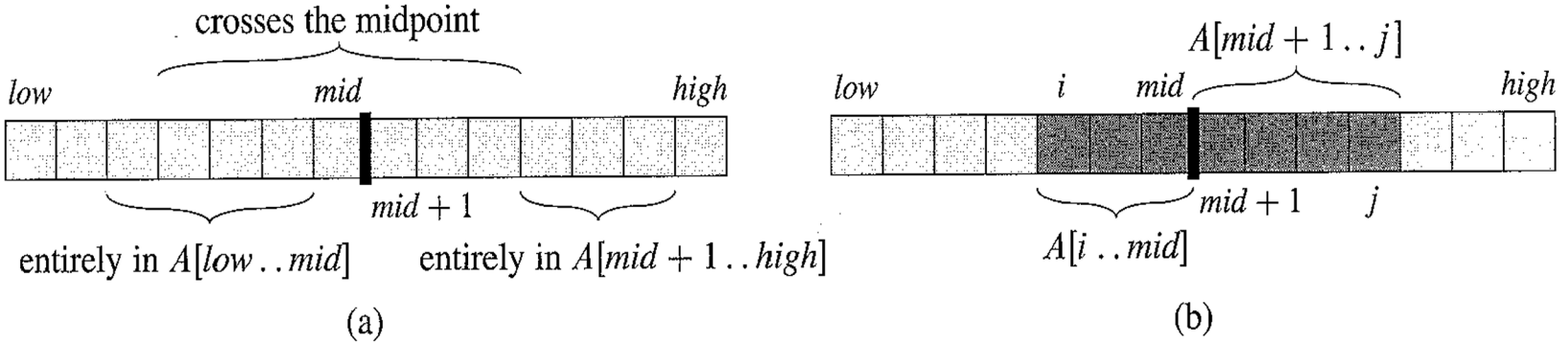


Figure 4.4 **(a)** Possible locations of subarrays of $A[low..high]$: entirely in $A[low..mid]$, entirely in $A[mid + 1..high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low..high]$ crossing the midpoint comprises two subarrays $A[i..mid]$ and $A[mid + 1..j]$, where $low \le i \le mid$ and $mid < j \le high$.

Max Subarray

How do we divide?

We observe that a maximum contiguous subarray A[i…j] must be located as follows:

1.It lies entirely in the left half of the original array: [low…mid];

2.It lies entirely in the right half of the original array: [mid+1…high];

3.It straddles the midpoint of the original array: i ≤ mid < j.

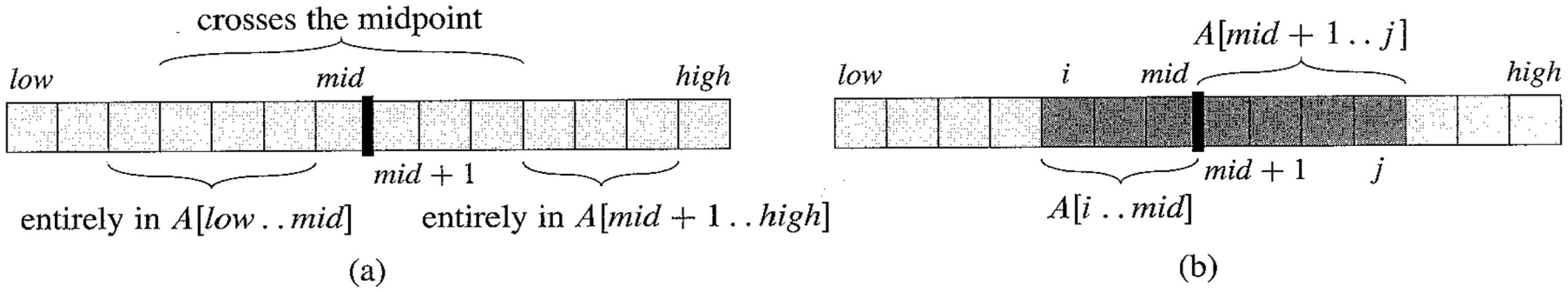


Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.

Max Subarray: Divide & Conquer

- The “left” and “right” subproblems are smaller versions of the original problem, so they are part of the standard Divide & Conquer recursion.
- The “middle” subproblem is not, so we will need to count its cost as part of the “combine” (or “divide”) cost.
  - The crucial observation (and it may not be entirely obvious) is that **we can find the maximum crossing subarray in time linear in the length of the A[low…high] subarray**.

How? A[i,…,j] must be made up of A[i…mid] and A[m+1…j] – so we find the largest A[i…mid] and the largest A[m+1…j] and combine them.

Maximum Sub Array Sum → using Divide & Conquer Approach

A: | 9 | 3 | -1 | 8 | 7 | -2 | -3 | 11 | 1 |
(0 1 2 3 4 5 6 7 8)

→ 33

L=26
R=12
C=26+7=33

* Conditions
  * Sub array - continuous
  * Array may / may not have Negative values.
    - ALL positive → add all for Maximum Value
    - combination of +ve & -ve Maximum Sub Array Method
      - Brute force
      - ✓ Divide & Conquer
      - Greedy Alg.
      - Kadane's Alg. (Dynamic Prog)
    - All Negative → find Maximum of elements given

mid = (0+8)/2 = 4 ⇒ l+r/2

Divide until no more division is possible

L=12 R=15 C=11+15=26

L=-2 R=12 C=7

L=> left subarray sum
R=> Right "
C=> cross "

mid = 0+4/2 = 2

mid = 5+8/2 = 13/2 = 6.5

L=12 R=-1 C=11

L=8 R=7 C=15

L=-2 R=-3 C=-5

L=11 R=1 C=12

m = 0+2/2 = 1

m = 3+4/2 = 3

m = 5+6/2 = 11/2 = 5

m = 7+8/2 = 15/2 = 7

L=9 R=3 C=12

L=-1 R=-1 C=-1

L=8 R=8 C=8

L=7 R=7 C=7

L=-2 R=-2 C=-2

L=-3 R=-3 C=-3

L=11 R=11 C=11

L=1 R=1 C=1

mid = 0+1/2 = 0

L=9 R=9 C=9

L=3 R=3 C=2

Return MAX(L, R, C) to previous level.

C = Left mid mid+1 R.

C = S(mid to left) + S(mid+1 → R)

Stop only +ve nos. -ve STOP

Max Subarray

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Example:

Input: [-2,1,-3,4,-1,2,1,-5,4],

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6.

Algorithm

1) Divide the given array in two halves

2) Return the maximum of following three

   a) Recursively calculate the Maximum subarray sum in left half

   b) Recursively calculate the Maximum subarray sum in right half

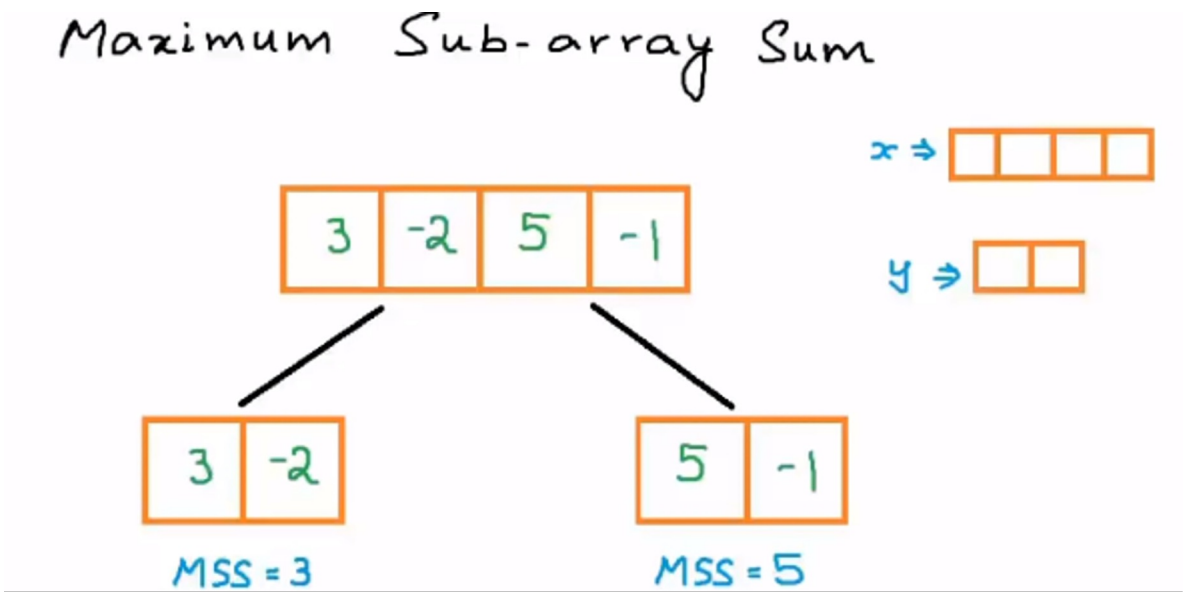
   c) Recursively calculate the Maximum subarray sum such that the subarray crosses the midpoint.

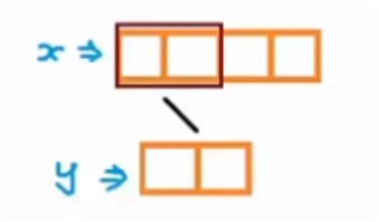
      i. Find the maximum sum starting from mid point and ending at some point on left of mid.

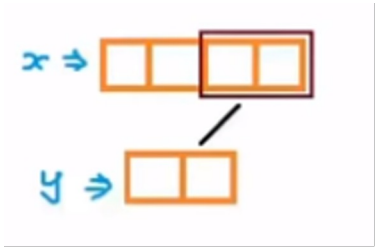
      ii. Find the maximum sum starting from mid + 1 and ending with some point on right of mid + 1.

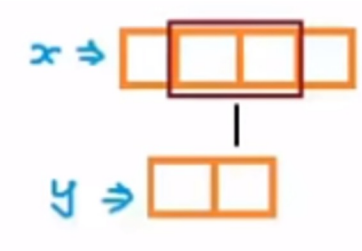
      iii. Finally, combine the two and return.

Example

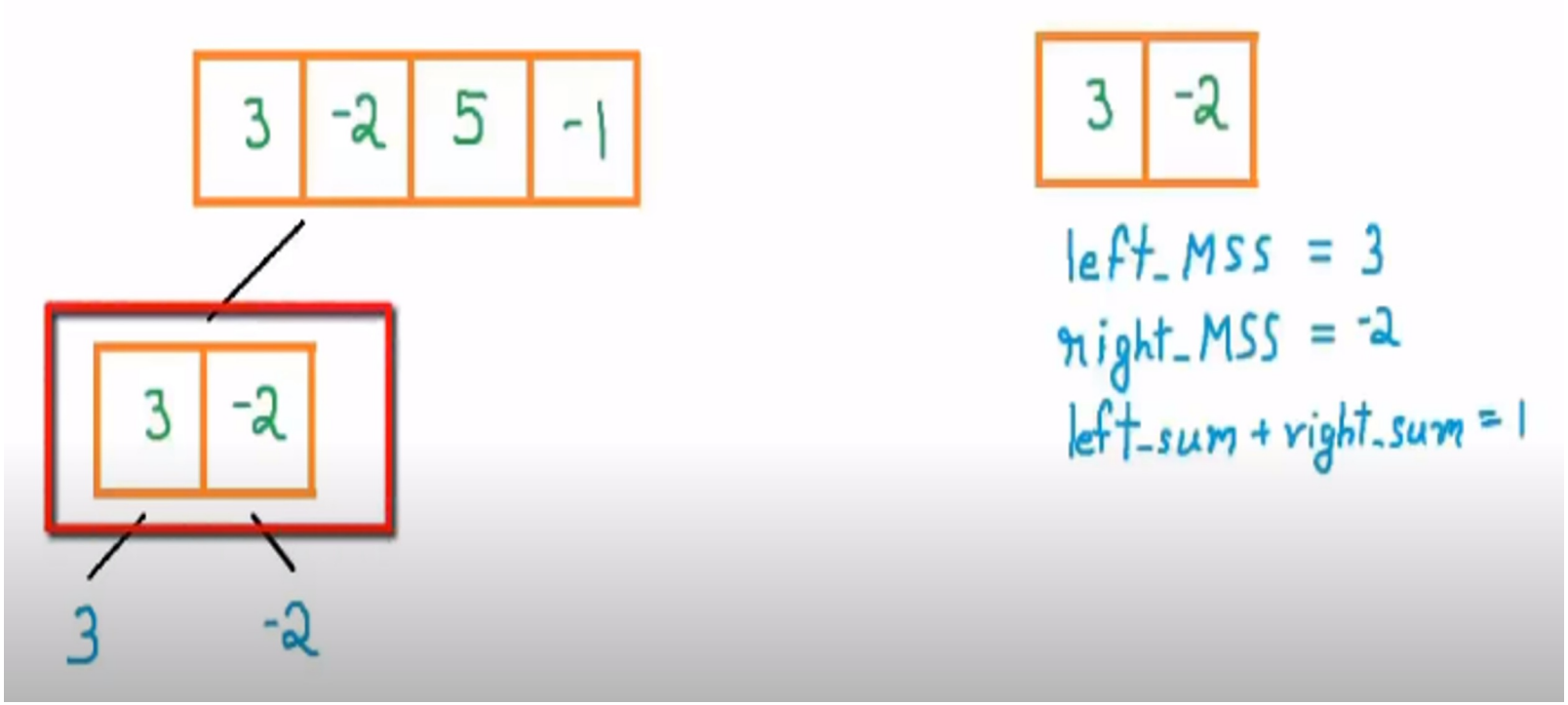




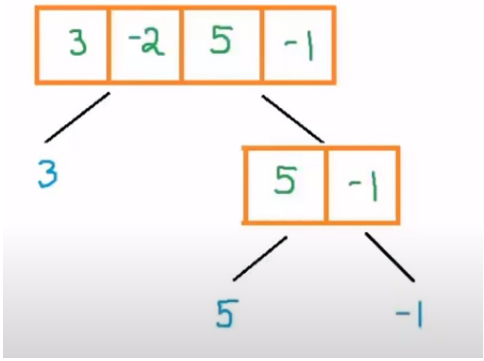


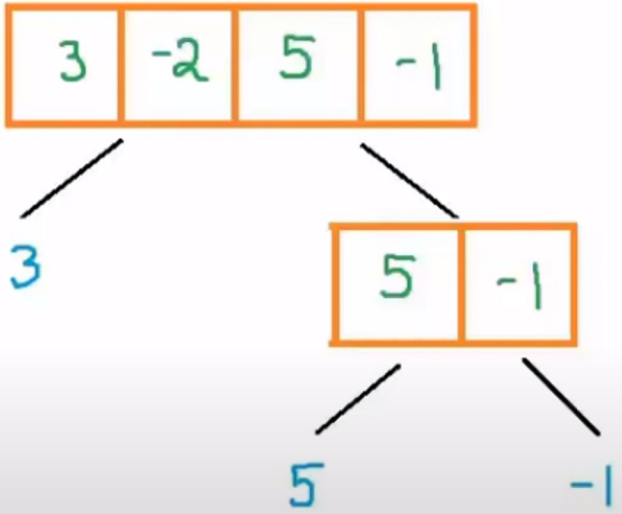


Example



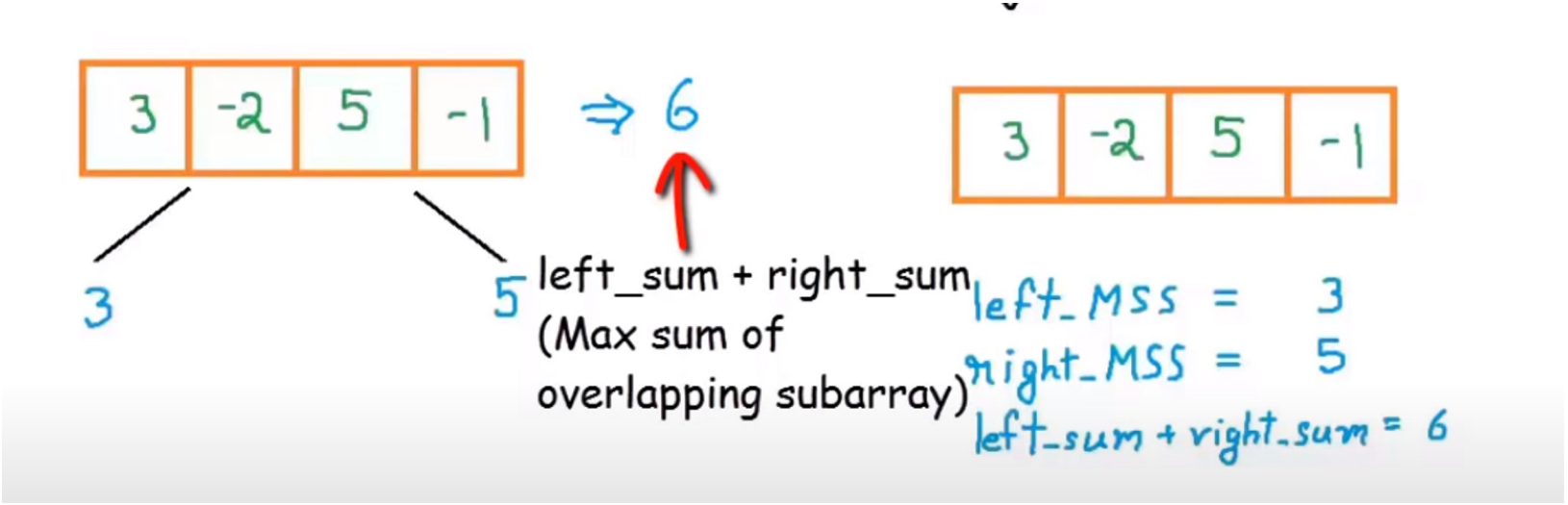
Example



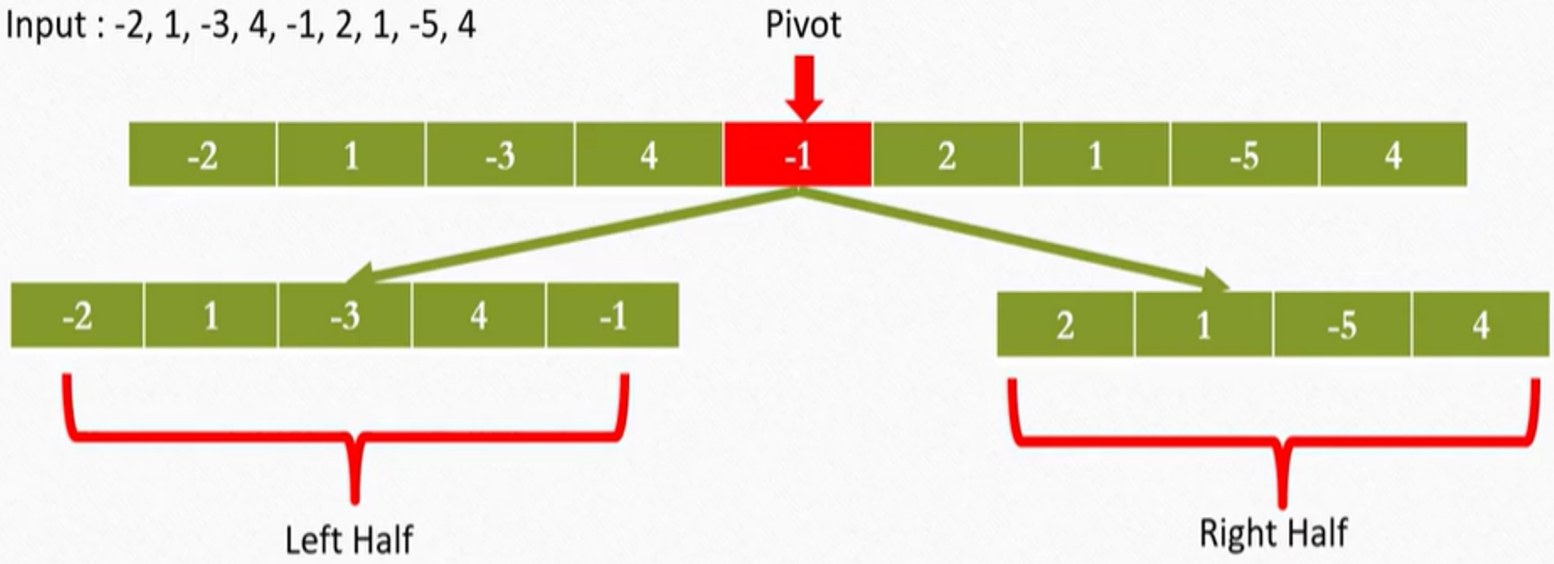


left_MSS = 5
right_MSS = -1
left_sum + right_sum = 4

Example


3 -2 5 -1
⇒ 6
3
5
left_sum + right_sum
(Max sum of
overlapping subarray)
3 -2 5 -1
left_MSS = 3
right_MSS = 5
left_sum + right_sum = 6

Example

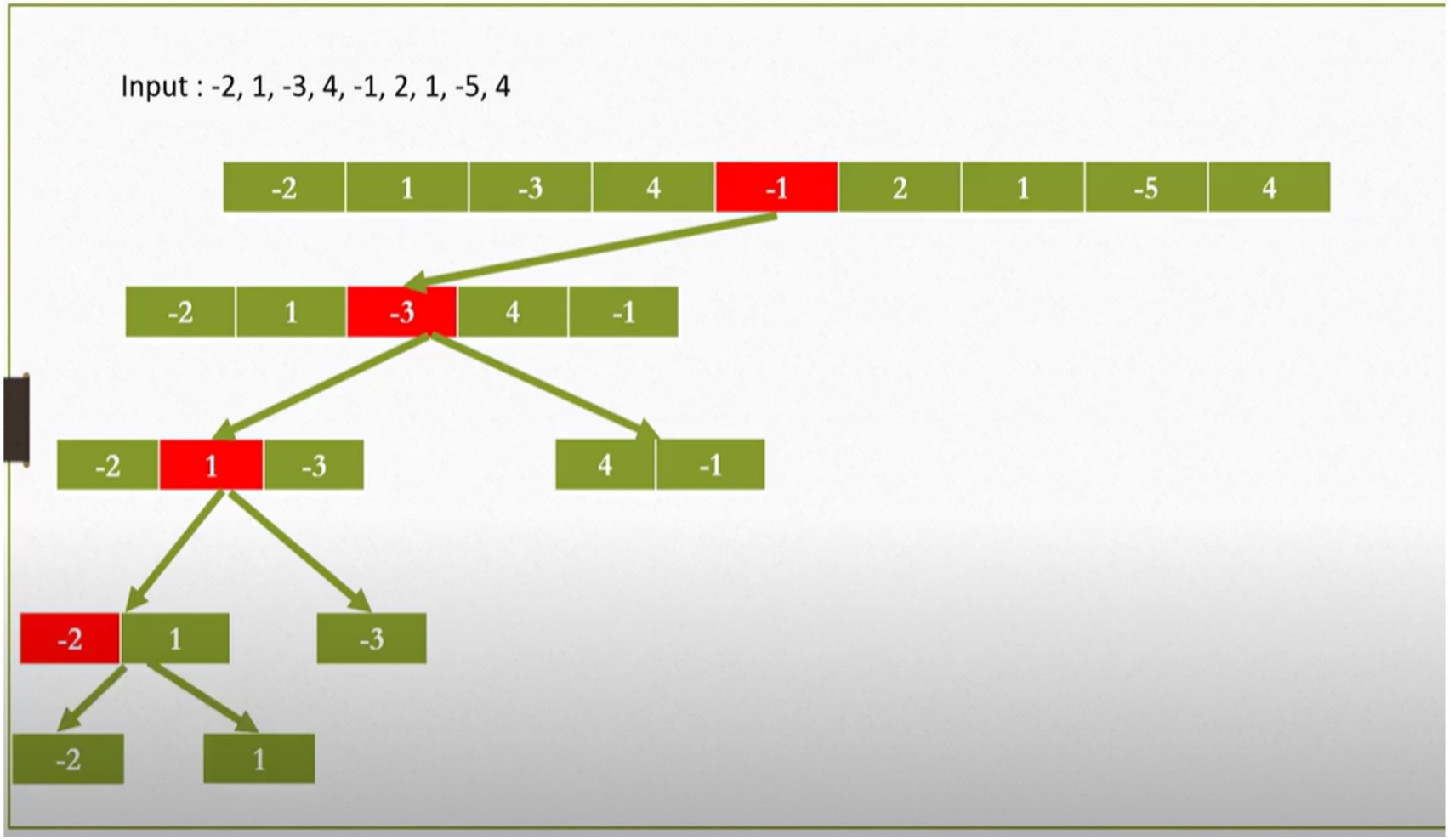


$$\text{Index of Pivot} = \frac{(\text{begin} + \text{end})}{2} = \frac{(0+8)}{2} = 4$$

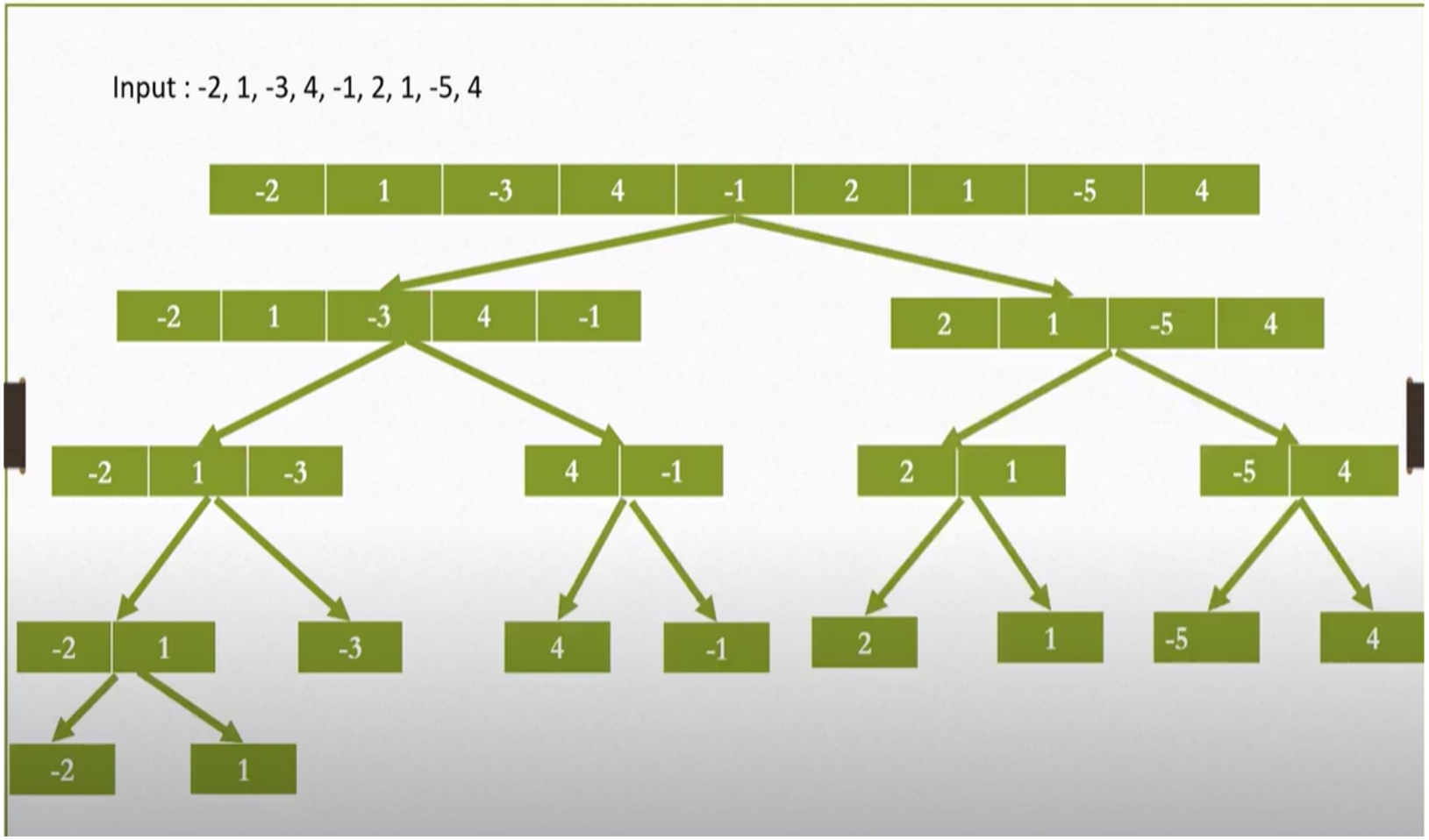
Left Half = begin to pivot (inclusive)

Right Half = pivot +1 to end

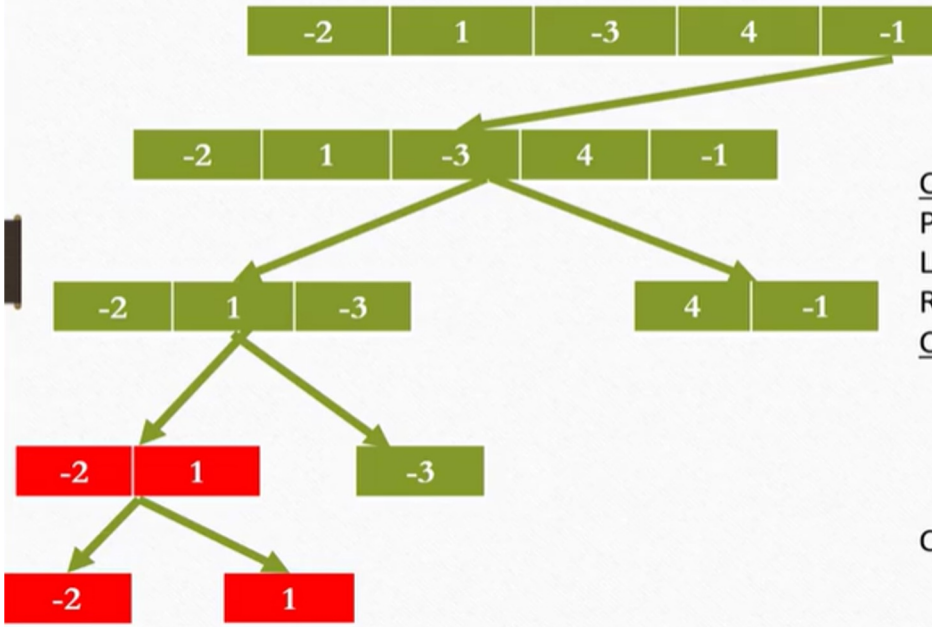
Example



Example


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4
-2 1 -3 4 -1 2 1 -5 4
-2 1 -3 4 -1
2 1 -5 4
-2 1 -3
4 -1
2 1
-5 4
-2 1
-3
4
-1
2
1
-5
4
-2
1

Example



Calculations :

Pivot Element: = -2

Left Sum = -2

Right Sum = 1

Cross Sum Calculations :

Left Sub Sum = -2

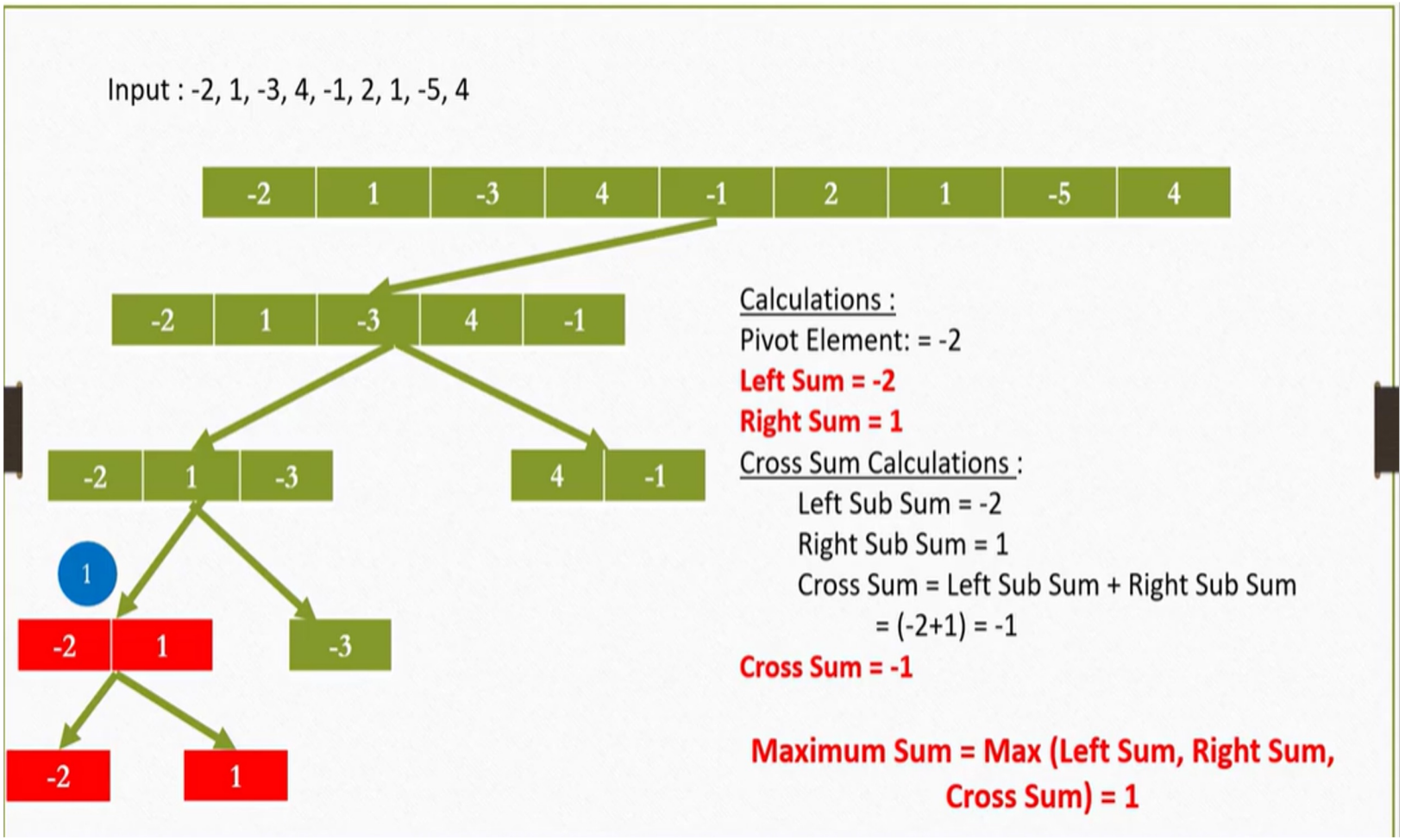
Right Sub Sum = 1

Cross Sum = Left Sub Sum +Right Sub Sum

= (-2+1) = -1

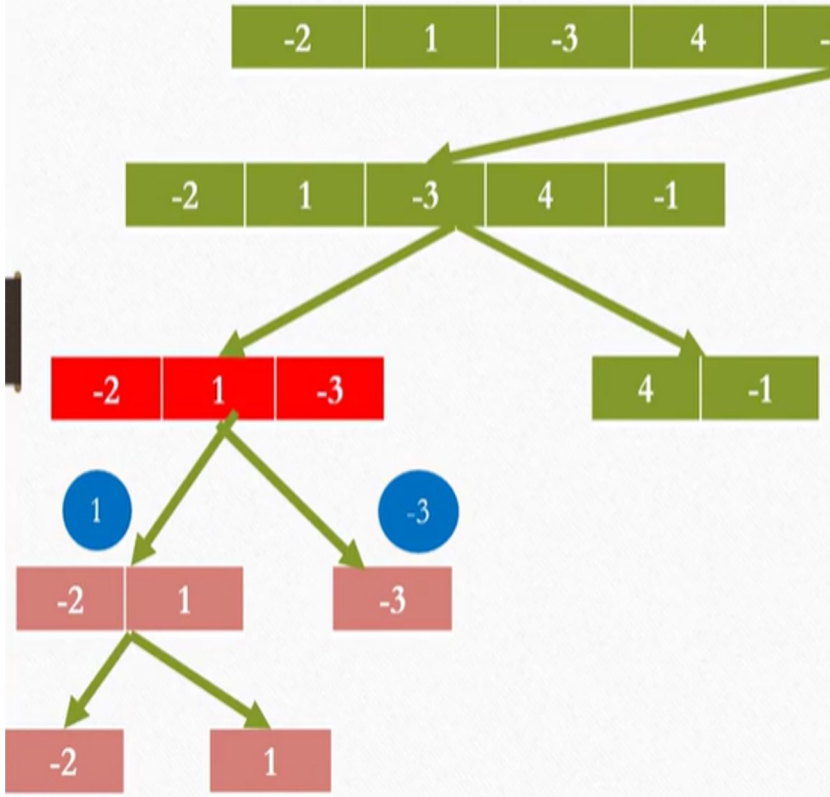
Cross Sum = -1

Example


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4
-2 1 -3 4 -1 2 1 -5 4
-2 1 -3 4 -1
-2 1 -3
4 -1
1
-2 1
-3
-2
1
Calculations :
Pivot Element: = -2
Left Sum = -2
Right Sum = 1
Cross Sum Calculations :
Left Sub Sum = -2
Right Sub Sum = 1
Cross Sum = Left Sub Sum + Right Sub Sum
= (-2+1) = -1
Cross Sum = -1
Maximum Sum = Max (Left Sum, Right Sum, Cross Sum) = 1

Example

Input : -2, 1, -3, 4, -1, 2, 1, -5, 4



Calculations :

Pivot Element: = 1

Left Sum = 1

Right Sum = -3

Cross Sum Calculations :

Left Sub Sum = Max(1, 1-2) = 1

Right Sub Sum = -3

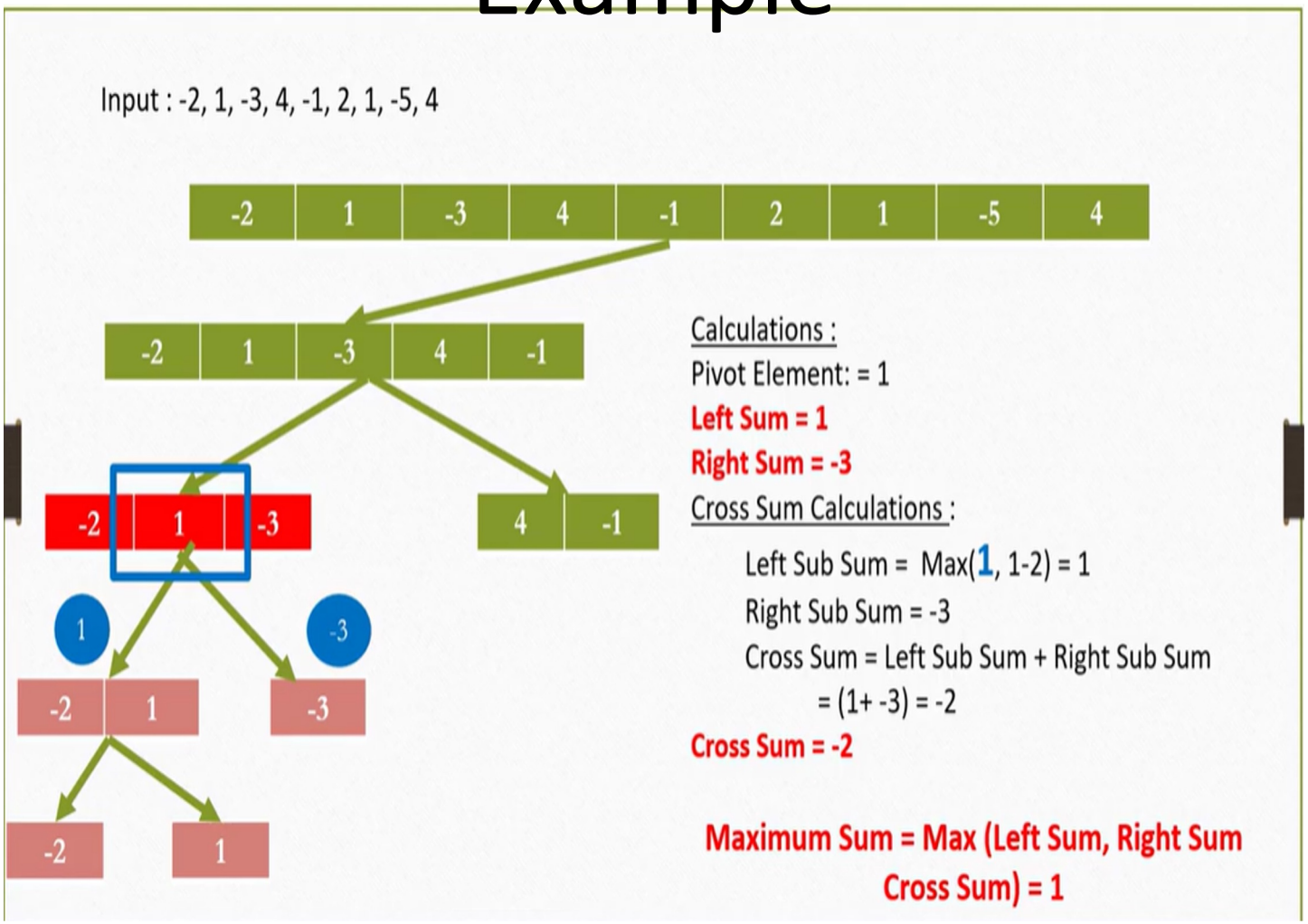
Cross Sum = Left Sub Sum + Right Sub Sum

= (1+ -3) = -2

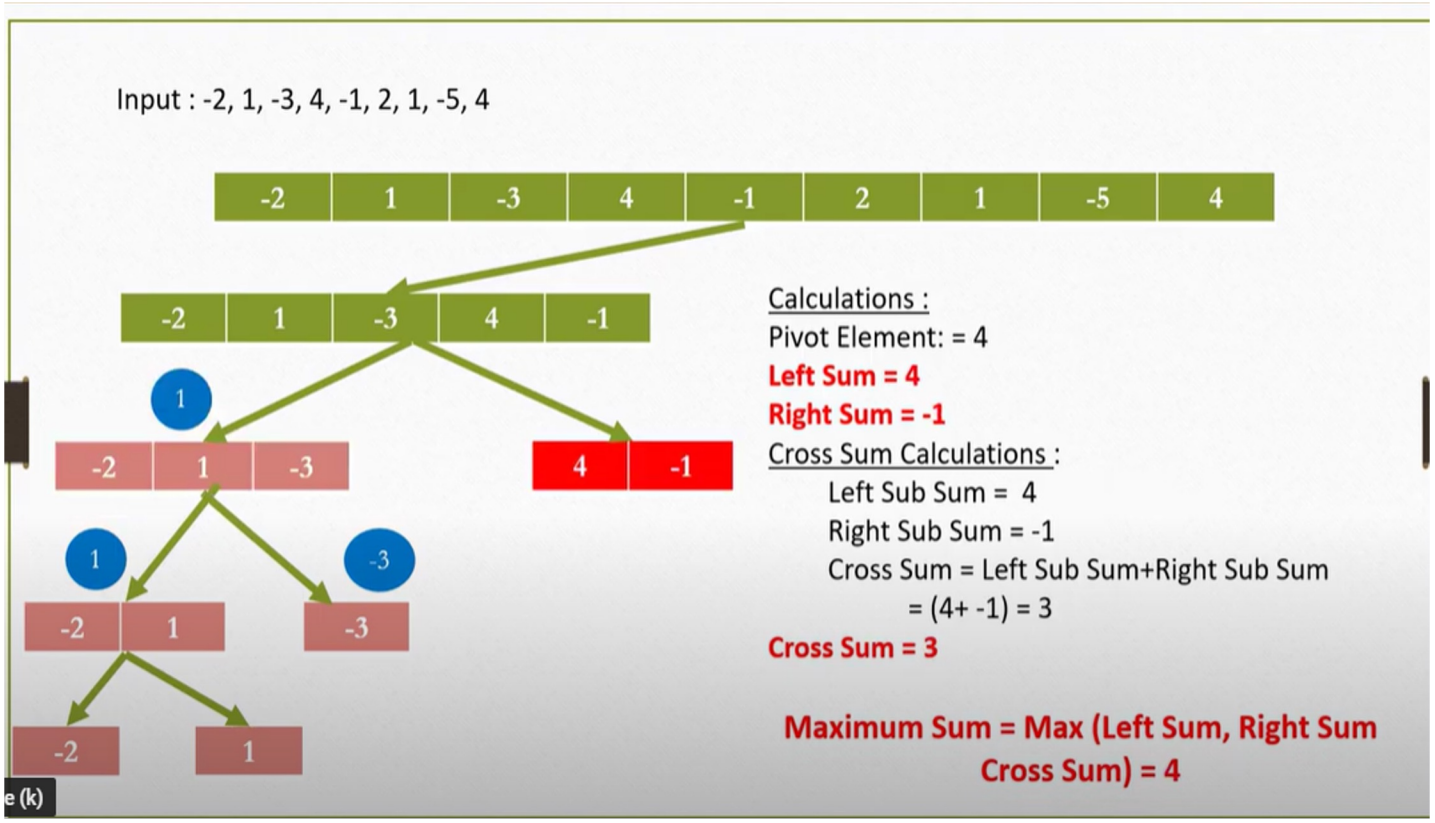
Cross Sum = -2

Maximum Sum = Max (Left Sum, Right Sum Cross Sum) = 1

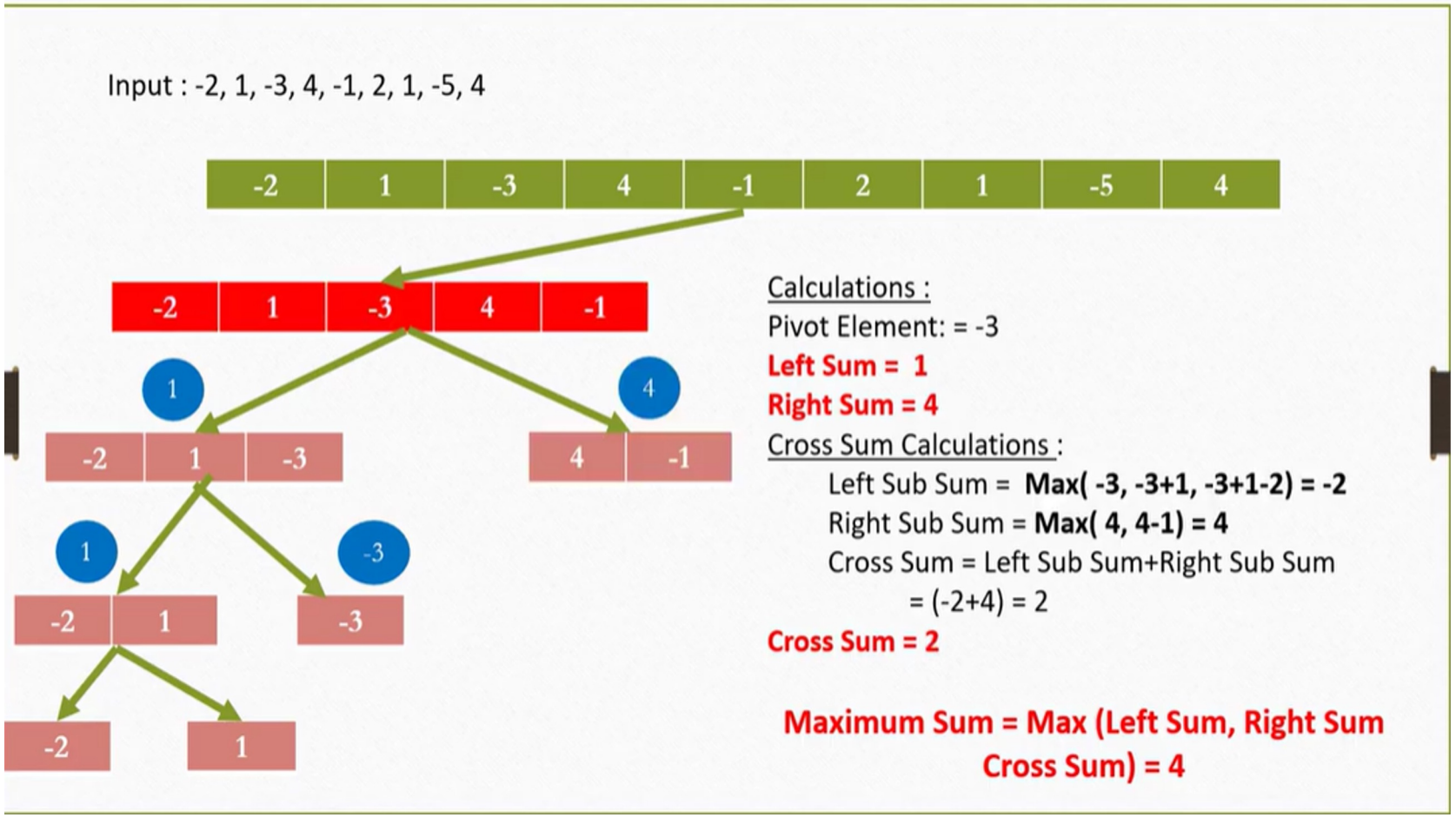
Example


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4
-2 1 -3 4 -1 2 1 -5 4
-2 1 -3 4 -1
-2 1 -3
4 -1
1
-3
-2 1
-3
-2
1
Calculations :
Pivot Element: = 1
Left Sum = 1
Right Sum = -3
Cross Sum Calculations :
Left Sub Sum = Max(1, 1-2) = 1
Right Sub Sum = -3
Cross Sum = Left Sub Sum + Right Sub Sum
= (1+ -3) = -2
Cross Sum = -2
Maximum Sum = Max (Left Sum, Right Sum
Cross Sum) = 1

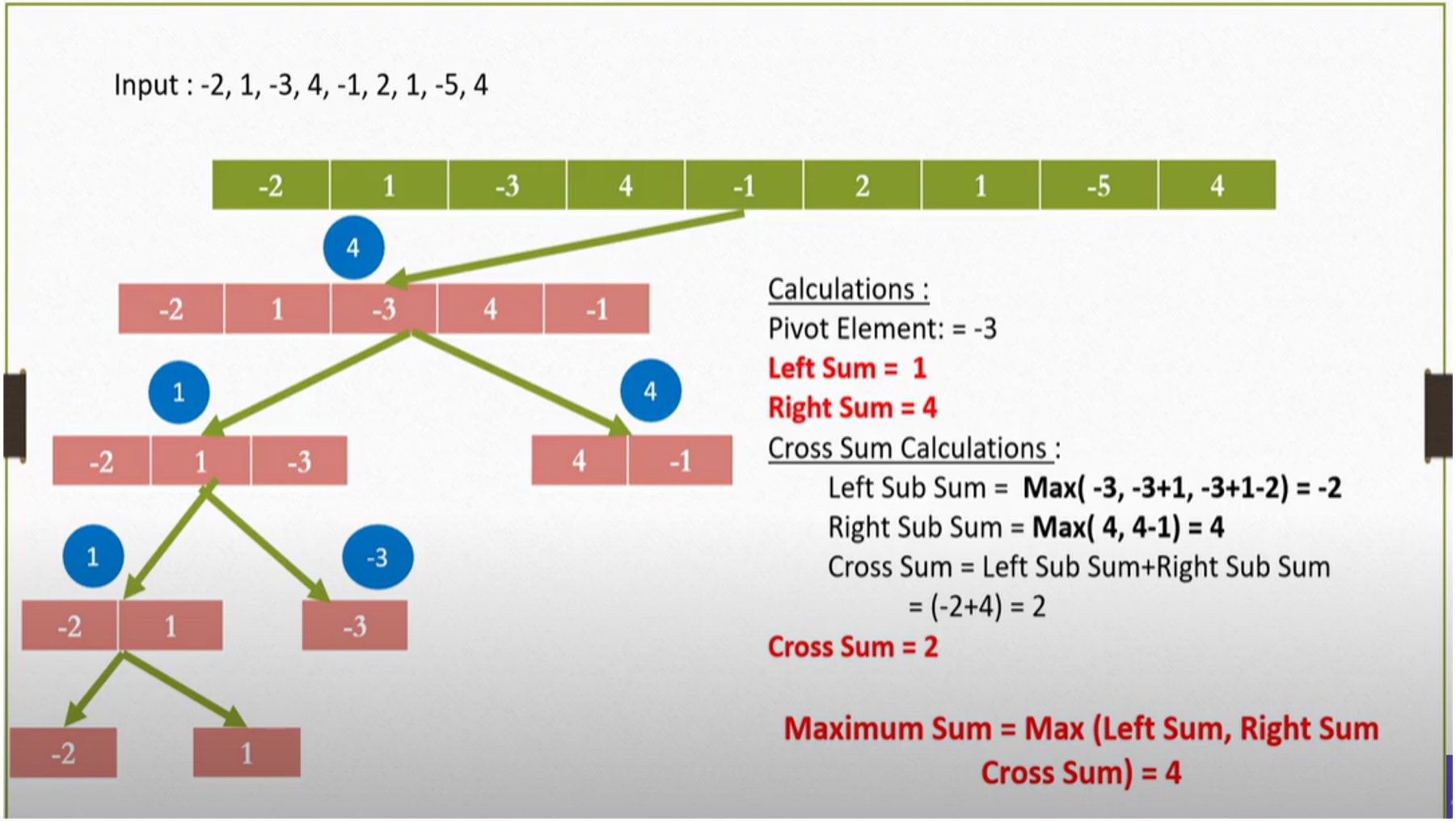
Example


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4
-2 1 -3 4 -1 2 1 -5 4
-2 1 -3 4 -1
1
-2 1 -3
4 -1
1
-3
-2 1
-3
-2
1
Calculations :
Pivot Element: = 4
Left Sum = 4
Right Sum = -1
Cross Sum Calculations :
Left Sub Sum = 4
Right Sub Sum = -1
Cross Sum = Left Sub Sum+Right Sub Sum
= (4+ -1) = 3
Cross Sum = 3
Maximum Sum = Max (Left Sum, Right Sum
Cross Sum) = 4

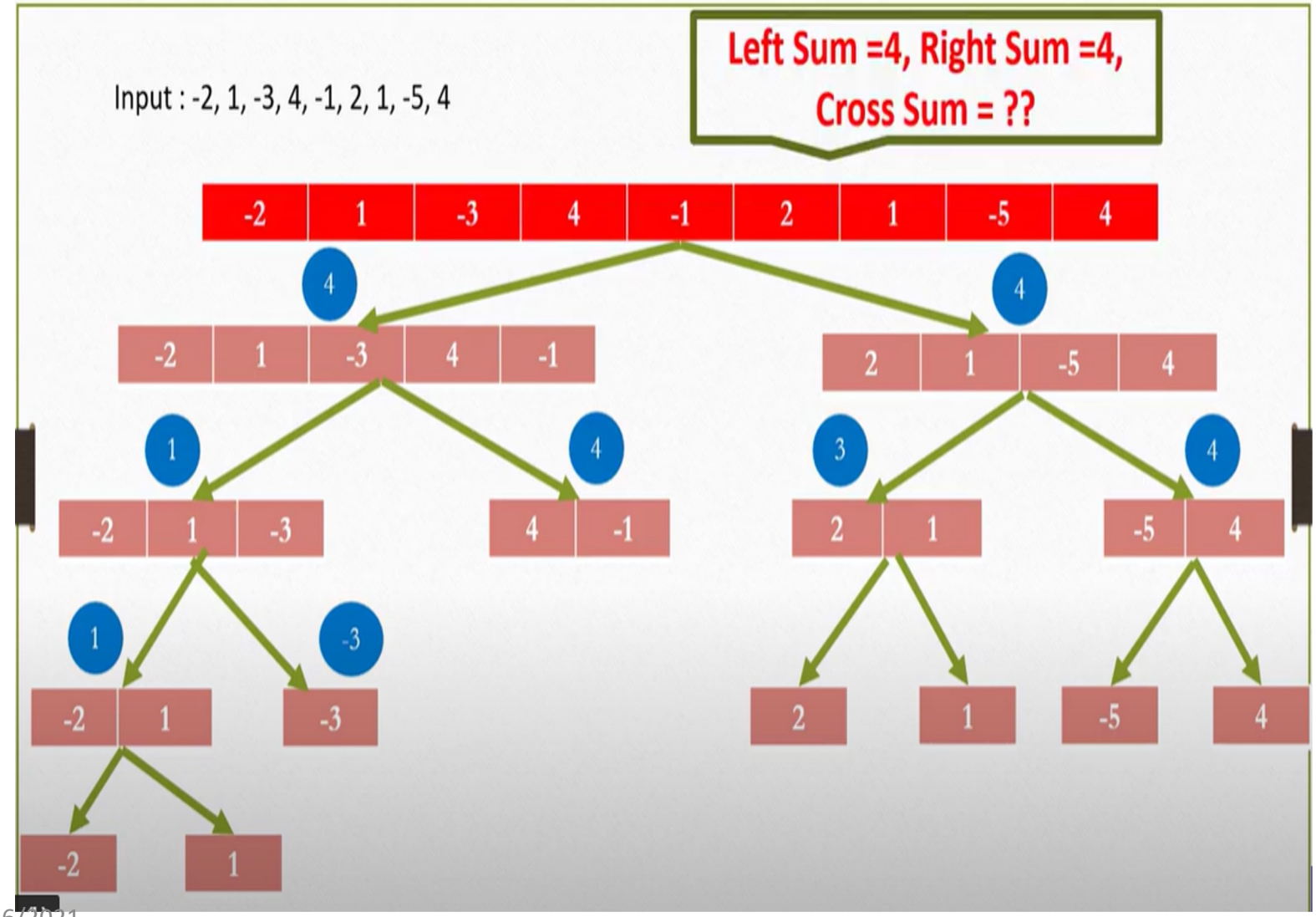
Example


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4
-2 1 -3 4 -1 2 1 -5 4
-2 1 -3 4 -1
1
4
-2 1 -3
4 -1
1
-3
-2 1
-3
-2
1
Calculations :
Pivot Element: = -3
Left Sum = 1
Right Sum = 4
Cross Sum Calculations :
Left Sub Sum = Max(-3, -3+1, -3+1-2) = -2
Right Sub Sum = Max(4, 4-1) = 4
Cross Sum = Left Sub Sum+Right Sub Sum
= (-2+4) = 2
Cross Sum = 2
Maximum Sum = Max (Left Sum, Right Sum Cross Sum) = 4

Example


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4
-2 1 -3 4 -1 2 1 -5 4
4
-2 1 -3 4 -1
1
-2 1 -3
4
4 -1
1
-2 1
-3
-2 1
-3
-2
1
Calculations :
Pivot Element: = -3
Left Sum = 1
Right Sum = 4
Cross Sum Calculations :
Left Sub Sum = Max(-3, -3+1, -3+1-2) = -2
Right Sub Sum = Max(4, 4-1) = 4
Cross Sum = Left Sub Sum+Right Sub Sum
= (-2+4) = 2
Cross Sum = 2
Maximum Sum = Max (Left Sum, Right Sum Cross Sum) = 4

Example


Input : -2, 1, -3, 4, -1, 2, 1, -5, 4
Left Sum =4, Right Sum =4,
Cross Sum = ??
-2 1 -3 4 -1 2 1 -5 4
4
4
-2 1 -3 4 -1
2 1 -5 4
1
4
3
4
-2 1 -3
4 -1
2 1
-5 4
1
-3
-2 1
-3
2
1
-5
4
-2
1

Example

Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

Left Sum =4, Right Sum =4,
Cross Sum = ??

Cross Sum Calculation :

Pivot Element: -1

Left Sub Sum = Max (-1, -1+4, -1+4-3, -1+4-3+1, -1+4-3+1-2)
= Max (-1, 3, 0, 1, -2)
= 3

Right Sub Sum = Max (2, 2+1, 2+1-5, 2+1-5+4)
= Max (2, 3, -2, 2)
= 3

Cross Sum = Left Sub Sum + Right Sub Sum = (3+3) = 6

Cross Sum = 6

Maximum Sum = Max (Left Sum, Right Sum, Cross Sum) = 6

Algorithm: Maximum Sub array

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

Algorithm: Maximum Crossing Sub Array

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

Time Complexity Analysis

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + c'n & \text{if } n > 1 \end{cases}$$

$$T(n) = cn + c'n \log n$$

$$O(n \log n)$$

Greedy Algorithm.

Algorithm

Considering input array as nums.

Initialize currentSum = maxSum as nums[0]

for each value in nums

currentSum = max(currentSum + num, num)

maxSum = max(maxSum, currentSum)

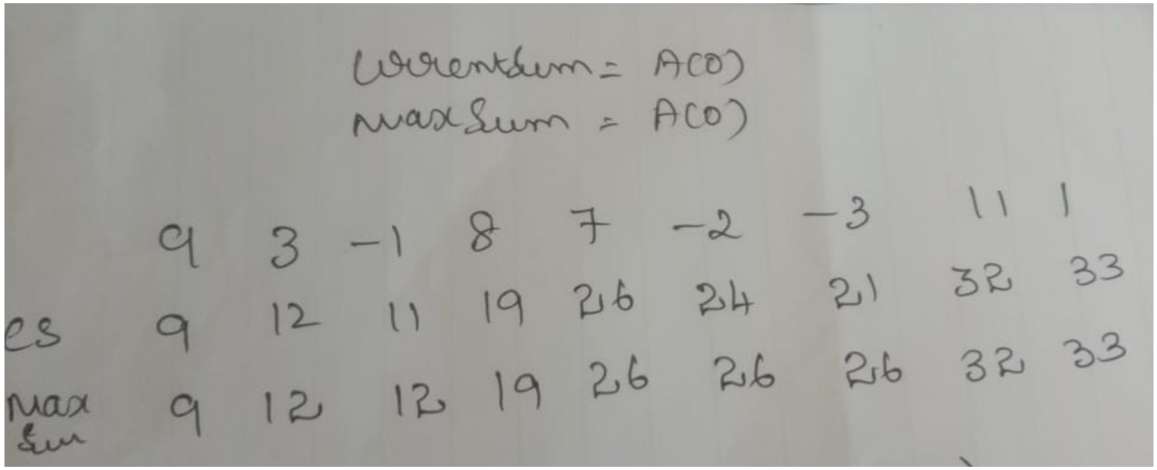
End for

return maxSum

-2	1	-3	4	-1	2	1	-5	4
0	1	2	3	4	5	6	7	8

i	0	1	2	3	4	5	6	7	8
currentsum	-2	1	-2	4	3	5	6	1	5
maxsum	-2	1	1	4	4	5	6	6	6

Maximum Sub-array Sum - Greedy Algorithm -example



Greedy Algorithm

Algorithm

Considering input array as nums.

Initialize currentSum = maxSum as nums[0]

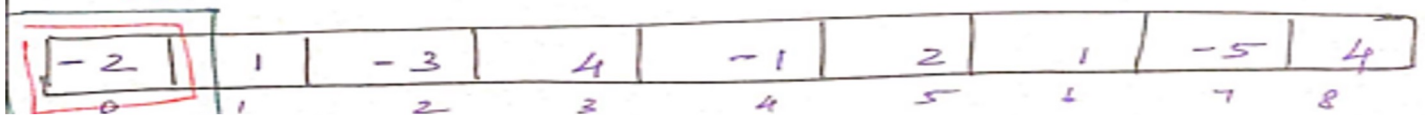
for each value in nums

currentSum = max(currentSum + num, num)

maxSum = max(maxSum, currentSum)

End for

return maxSum



KADANES'S ALGORITHMN –
DYNAMIC PROGRAMMING –
MAXIMUM SUB ARRAY SUM

Date: __/__/____

Dynamic Programming

Kadanis alg

```
{
  Current_Sum = 0
  max_Sum = Int_Min :
  for i = 0 to n  2.

   Current_Sum += a[i];

   if (Current_Sum > max_Sum)
     {
       max_Sum = Current_Sum
     }
   if (Current_Sum < 0)
       {
         Current_Sum = 0;
       }
  }
    Return max_Sum;
}.
```

CurrentSum = A[0]
maxSum = Int_Min

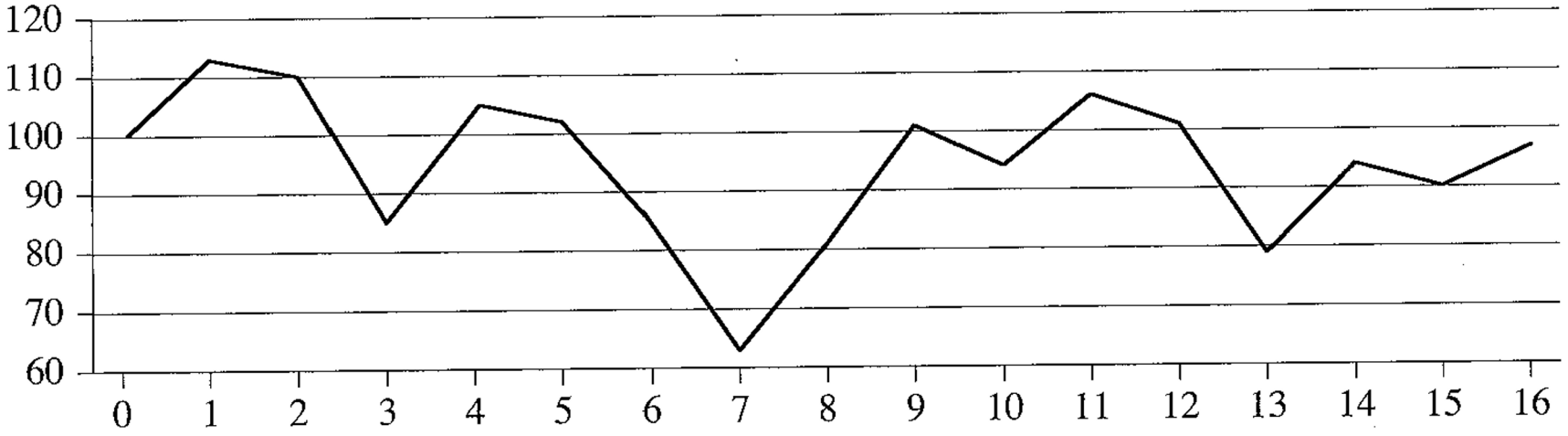
	9	3	−1	8	7	−2	−3	11	1
cs	9	12	11	19	26	24	21	32	33
max Sum	9	12	12	19	26	26	26	32	33

Application

Application: an unrealistic stock market game, in which you decide when to buy and see a stock, with full knowledge of the past and future. ***The restriction*** is that you can perform just one ***buy*** followed by a ***sell***. The buy and sell both occur right after the close of the market.

The interpretation of the numbers: each number represents the stock value at closing on any particular day.

Maximum Subarray Problem

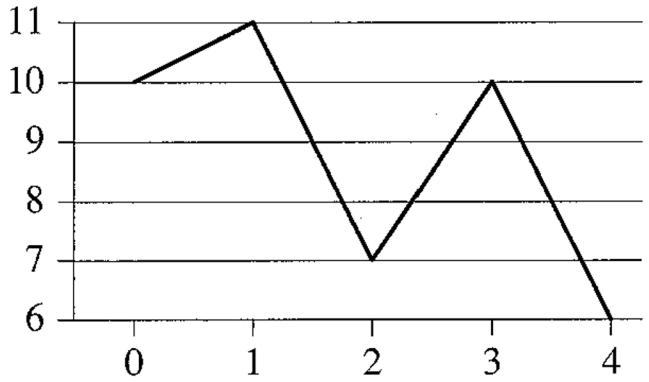


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

Maximum Subarray Problem

Another Example: buying low and selling high, even with perfect knowledge, is not trivial:



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of $3 per share would be earned by buying after day 2 and selling after day 3. The price of $7 after day 2 is not the lowest price overall, and the price of $10 after day 3 is not the highest price overall.

A Better Solution: Max Subarray

Transformation: Instead of the daily price, let us consider the daily change: A[i] is the difference between the closing value on day i and that on day i-1.

The problem becomes that of finding a contiguous subarray the sum of whose values is maximum.

Home Assignment

- Find the sub array sum of the following array,
  1. -2, -5, 6, -2, -3, 1, 5, -6
  2. 1,-3,2, -5, 7, 6, -1, -4, 11, -23

BINARY SEARCH

Binary Search

Session Learning Outcome-SLO-Solve problems using divide and conquer approaches

- Motivation of the topic

  Binary Search is one of the fastest searching algorithms.
  - It is used for finding the location of an element in a linear array.
  - It works on the principle of divide and conquer technique.

*Binary Search Algorithm can be applied only on **Sorted arrays**.*

- So, the elements must be arranged in-
  - Either ascending order if the elements are numbers.
  - Or dictionary order if the elements are strings.
- To apply binary search on an unsorted array,
  - First, sort the array using some sorting technique.
  - Then, use binary search algorithm.

- Binary search is an efficient searching method. While searching the elements using this method the most essential thing is that the elements in the array should be a sorted one.
- An element which is to be searched from the list of elements stored in the array A[0---n- 1] is called as Key element.
- Let A[m] be the mid element of array A. Then there are three conditions that need to be tested while searching the array using this method.
- They are given as follows
  - If key==A[m] then desired element is present in the list.
  - Otherwise if key <=A[m] then search the left sub list
  - Otherwise if Key>=A[m] then search the right sub list The following algorithm explains about binary search.

Algorithm

Input: A array A[0...n-1] sorted in ascending orderand search key K.

Output: An index of array element which is equal to k

L□0;h □n-1

```
Algorithm RBinSearch(l, h,key)
{
    If(l==h)     // for small problem(ie when n=1)
    {
        if(A[i]==key)
              return l;
        else
              return 0;
     }
    else
     {
           mid=(l+h)/2;
           If(key==A{mid})
               return mid;
           If(key<a[mid])r
               return RBinSearch(l,mid-1,key);
          else
               return RBinSearch ( mid+1,h,key);
}
```

Time Complexity Analysis

Binary Search

```
int bin_search (int arr [], int x, int start,
                int end)
{
 int mid = (start + end) / 2;
 if (arr [mid] == x)
  { return mid; }
 if (start == end)
  { return -1; }
 if (arr [mid] > x)
 {return bin_search (arr, x, start, mid-1);
 }
 if (arr [mid] < x)
{return bin_search (arr, x, mid+1, end);
}
}
```

$T(N) = c + T(N/2)$ — (1)

$T(N/2) = c + T(N/4)$ — (2)

Substitute (2) in (1)

$T(N) = T(N/4) + 2c$ — (3)

$T(N/4) = c + T(N/8)$ — (4)

Substitute (4) in (3)

$T(N) = T(N/8) + 3c$ — (5)

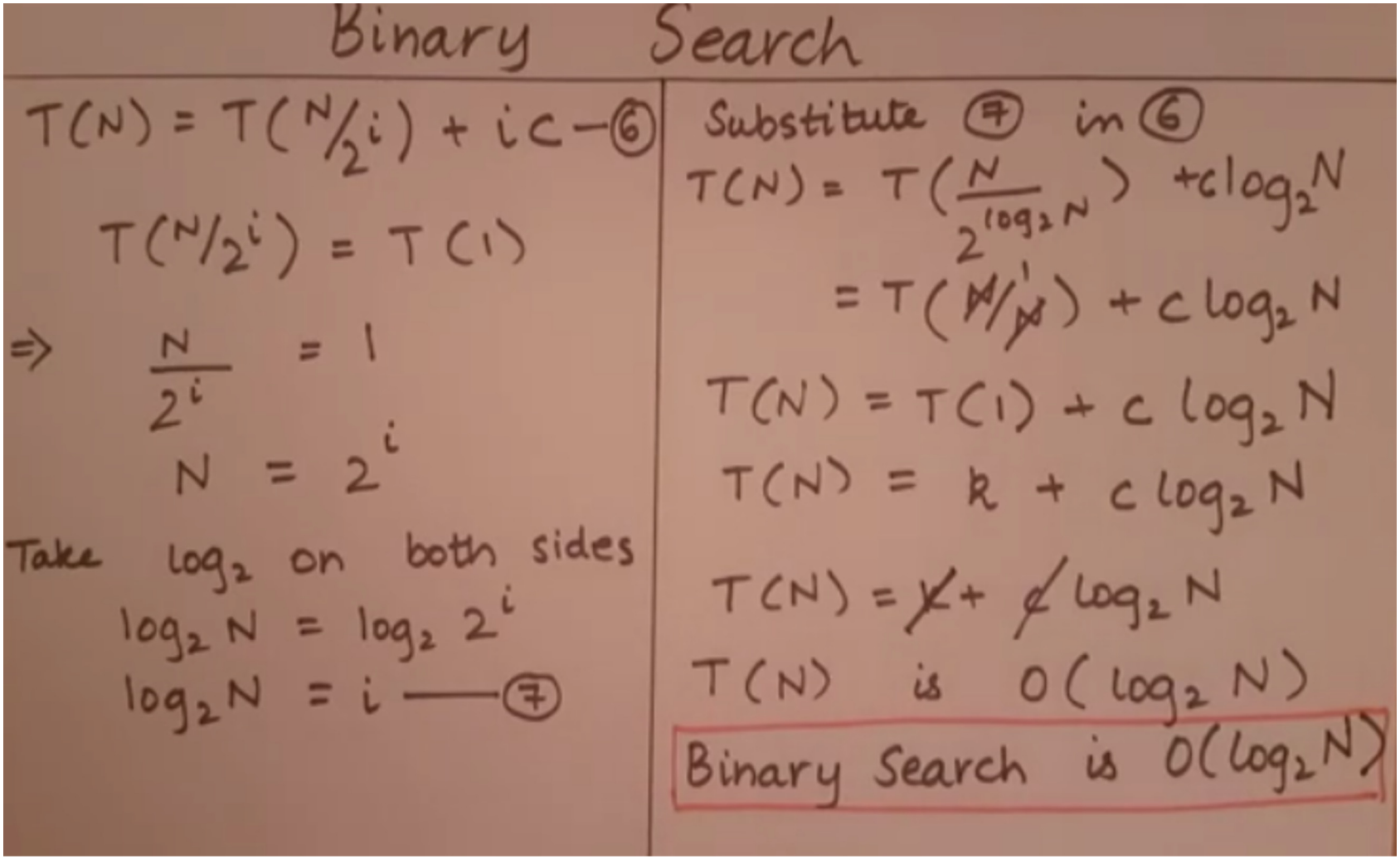
Pattern identified :

$T(N) = T(N/2^i) + ic$

At some point, as $N/2^i$ diminishes, we reach only 1 element

$T(N/2^i) = T(1)$

Time Complexity Analysis



Time Complexity Analysis

- $T(n) = \begin{cases} 1 & n=1 \\ T(n/2) & n>1 \end{cases}$

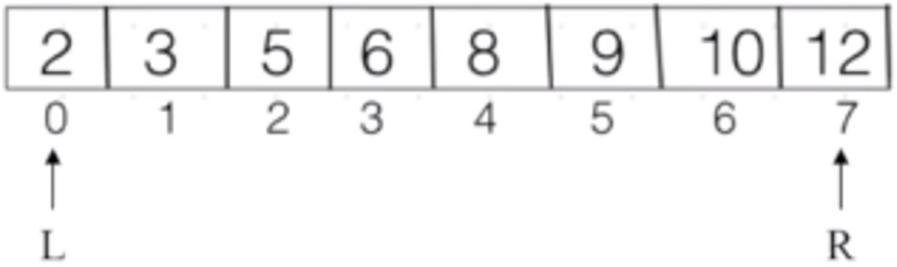
- Time complexity is $O(\log_2 n)$

ANALYSIS

- The basic operation in binary search is comparison of search key with array elements.
- To analyze efficiency of binary search we must count the number of times the search gets compared with the array elements.
- The comparison is also called three way comparisons because the algorithm makes the comparison to determine whether key is smaller, equal to or greater than A[m].
- In this algorithm after one comparison the list of n elements is divided into n/2 sub list.
- The worst case efficiency is that the algorithm compares all the array elements for searching the desired element.
- In this method one comparison is made and based on the comparison array is divided each time in n/2 sub list.

Best Case Analysis of Binary Search

Array size: N = 8

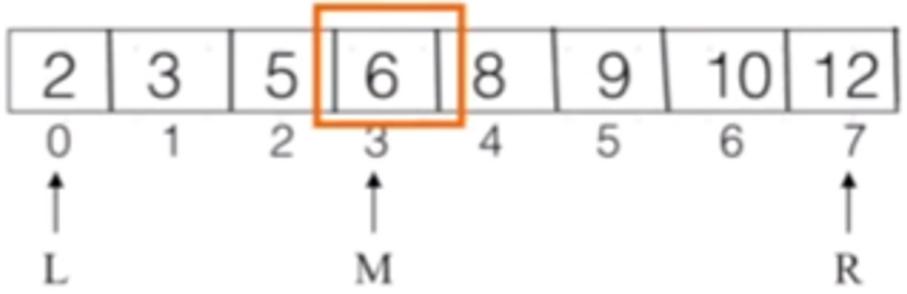


Search element = 6

Search element is equal
to the very first middle element.

Best Case Analysis of Binary Search

Array size: N = 8



Search element = 6

O(1)

$$m = \frac{L + R}{2}$$

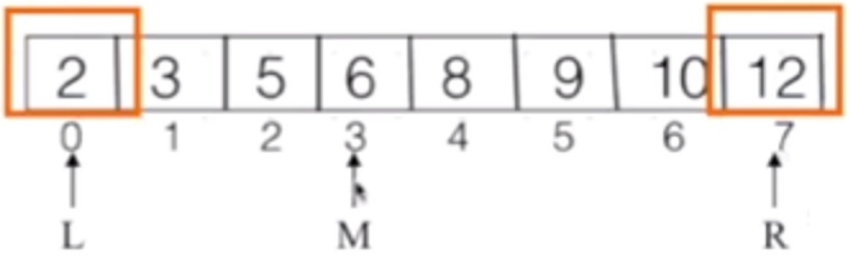
$$m = \frac{0 + 7}{2}$$

$$m = 3$$

Search element is equal
to the very first middle element.

Worst Case Analysis of Binary Search

Array size: N = 8



1st comparision:

Search element = 2

$m = \frac{L + R}{2}$

Middles element ≠ Search element

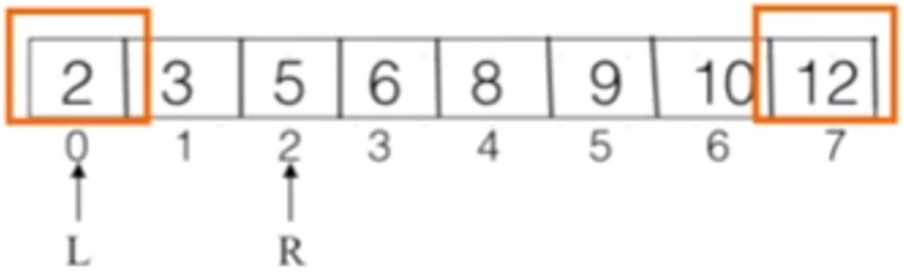
$m = \frac{0 + 7}{2}$

$m = 3$

Search element is present at either beginning of the array or end of the array.

Worst Case Analysis of Binary Search

Array size: N = 8



2nd comparision:

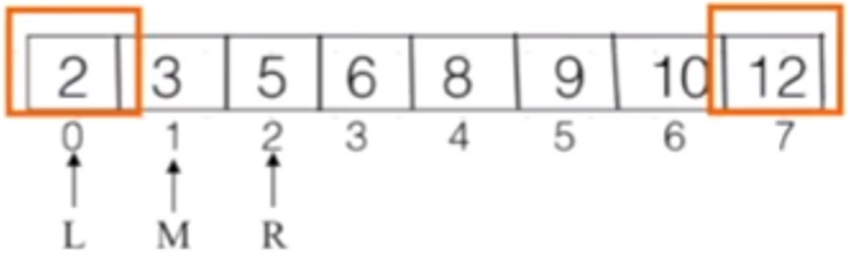
Search element = 2

Middles element ≠ Search element

Search element is present at either beginning of the array or end of the array.

Worst Case Analysis of Binary Search

Array size: N = 8



2nd comparision:

$m = \frac{L + R}{2}$

$m = \frac{0 + 2}{2}$

$m = 1$

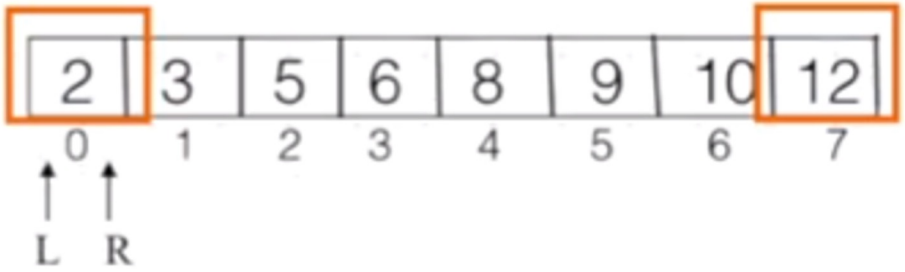
Search element = 2

Middles element ≠ Search element

Search element is present at either beginning of the array or end of the array.

Worst Case Analysis of Binary Search

Array size: N = 8



3rd comparision:

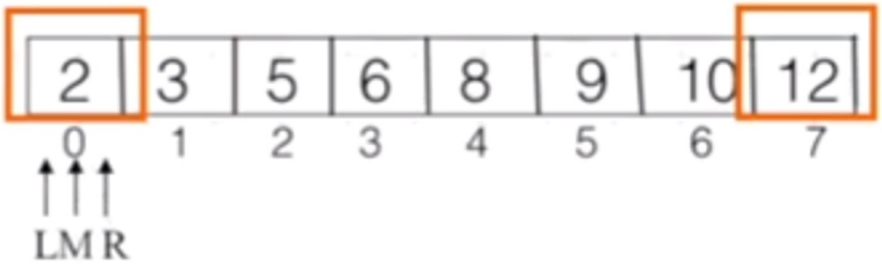
Search element = 2

Search element = Middle element

Search element is present at either beginning of the array or end of the array.

Worst Case Analysis of Binary Search

Array size: N = 8



3rd comparision:

Search element = 2

$m = \frac{L + R}{2}$

Search element = Middle element

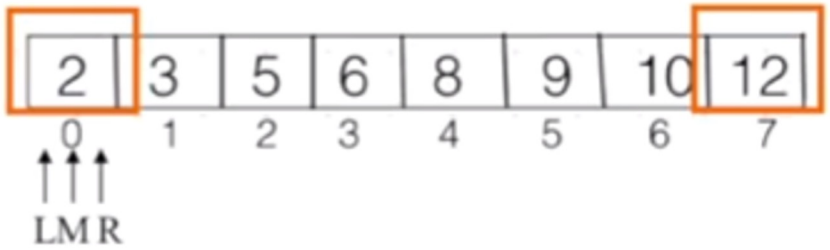
$m = \frac{0 + 0}{2}$

$m = 0$

Search element is present at either beginning of the array or end of the array.

Worst Case Analysis of Binary Search

Array size: N = 8



$8/2 \rightarrow 4 = 1$

$4/2 \rightarrow 2 = 2$

$2/2 \rightarrow 1 = 3$

$8/2^1 \rightarrow 4 = 1$

$8/2^2 \rightarrow 2 = 2$

$8/2^3 \rightarrow 1 = 3$

Search element is present at either beginning of the array or end of the array.

$N/2^k = 1$

Worst Case Analysis of Binary Search

Array size: N

$N/2^k = 1$

$N = 2^k$

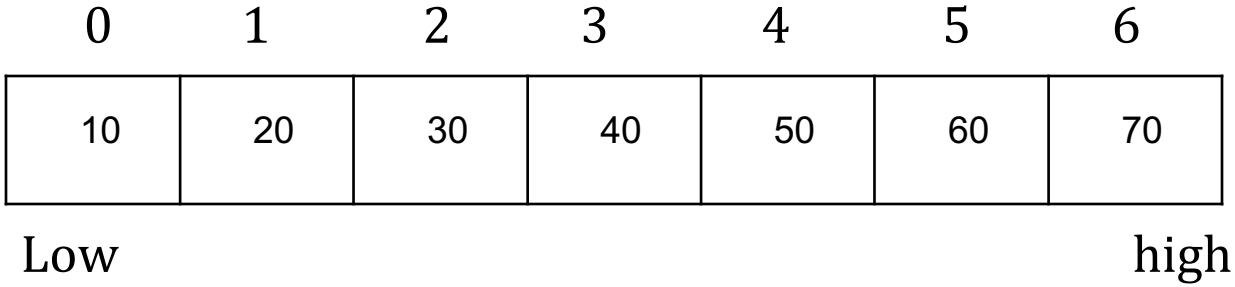
$\log_2 N = k \log_2 2$

$k = \log_2 N$

Search element is not present in the array.

EXAMPLE FOR BINARY SEARCH

Step:1 Consider a list of elements sorted in array A as



The Search key element is key=60
Now to obtain middle element we will apply formula
mid=(low+high)/2
mid=(0+6)/2
mid=3
Then check A[mid]==key, A[3]=40
A[3]!=60 Hence condition failed
Then check key>A[mid],A[3]=40
60>A[3] Hence condition satisfied so search the Right Sublist

Step 2:

- The Right Sublist is

50	60	70

- Now we will again divide this list to check the mid element

Left sublist mid right sublist

- mid=(low+high)/2
- mid=(4+6)/2
- mid=5
- Check if A[mid]==key
- (i.e)A[5]==60.Hence condition is satisfied. The key element is present in position 5.
- **The number is present in the Array A[ ] at index position 5.**

- We can analyze the best case , Worst case and A case . The time complexity of binary search is given as follows

Best case	Average case	Worst Case
$\theta(1)$	$\theta(\log n)$	$\theta(\log n)$

- In conclusion we are now able to completely describe the computing time of binary search by giving formulas that describe best, average and worst cases
- Successful searches unsuccessful searches
- $\theta(1)$ $\theta(\log n)$ $\theta(\log n)$ $\theta(\log n)$
- best average worst best, average, worst

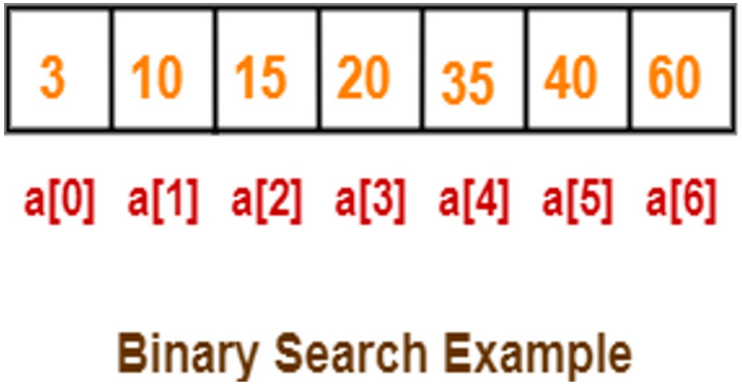
- **Advantages of Binary search:**
  - Binary search is an optimal searching algorithm using which we can search the desired element very efficiently
- **Disadvantages of binary Search :**
  - This Algorithm requires the list to be sorted . Then only this method is applicable
- **Applications of binary search:**
  - The binary search is an efficient searching method and is used to search desired record from database
  - For solving with one un known this method is used

Summary:

- Binary Search time complexity analysis is done below-
  - In each iteration or in each recursive call, the search gets reduced to half of the array.
  - So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.
- **Time Complexity of Binary Search Algorithm is $O(\log_2 n)$.**
  - Here, n is the number of elements in the sorted linear array.

Home assignment:

- Search the Element 15 from the given array using Binary Search Algorithm.



Binary Search Example

Merge Sort

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

- **Divide:** Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

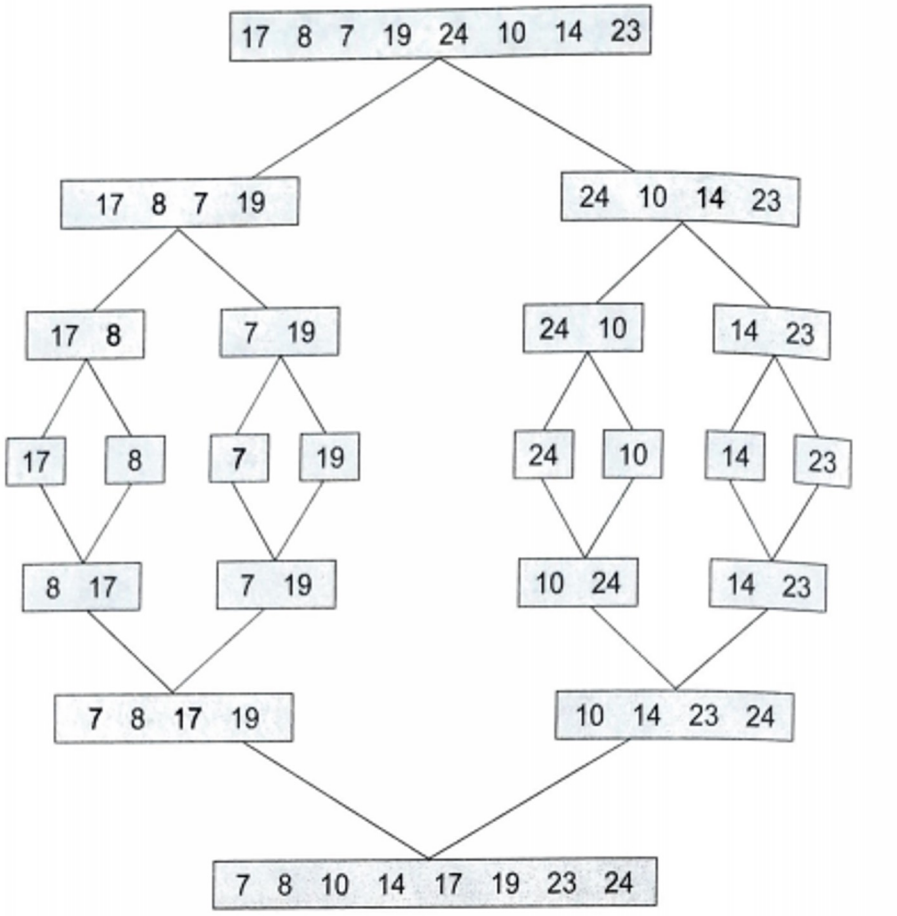
Merge Sort Procedure

Step1: Divide an array A into subarrays B and C

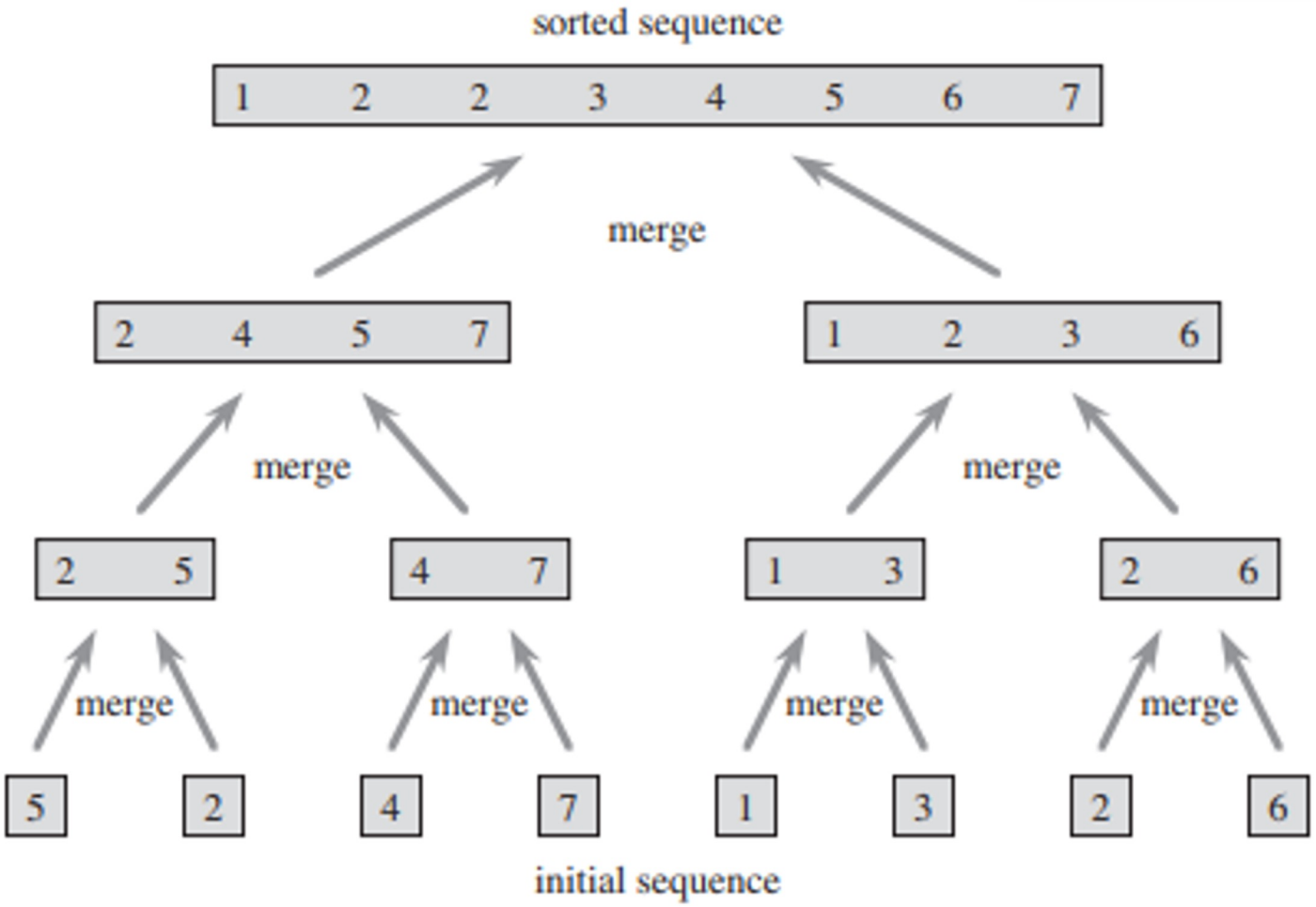
Step 2: Sort subarray B recursively; this yields B sorted subarray

Step 3: Sort subarray C recursively; this yields C sorted subarray

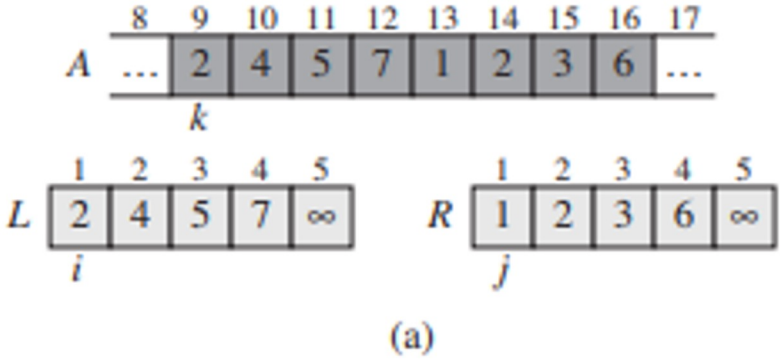
Step 4: Combine B and C sorted subarrays to obtain the final sorted array A



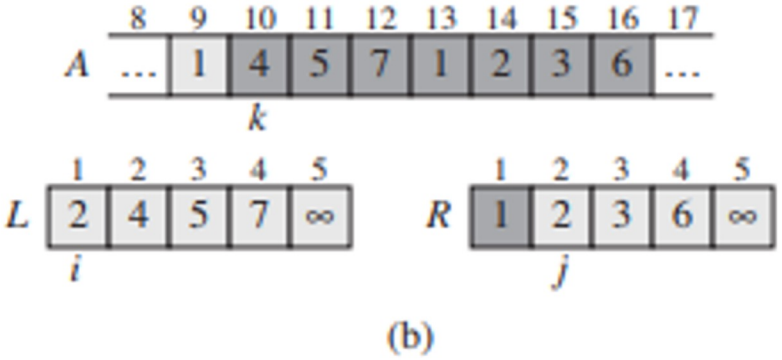
Merge Procedure



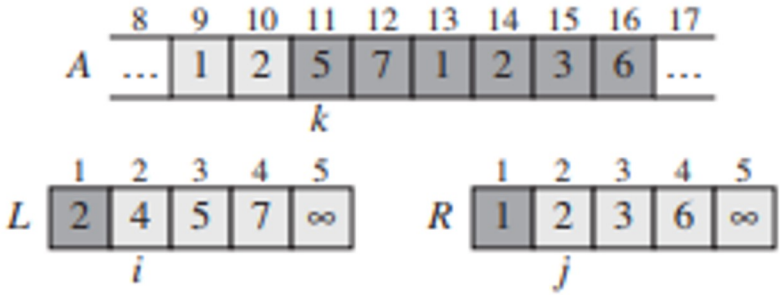
Merge Procedure



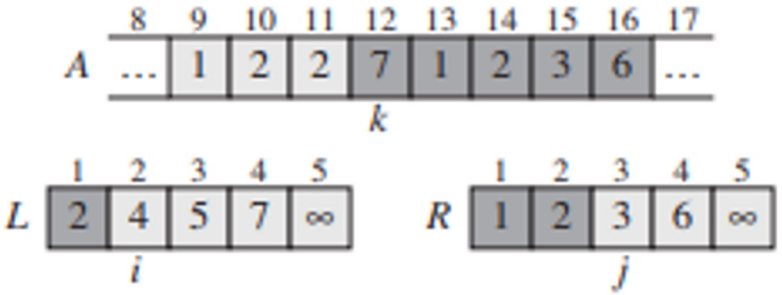
(a)



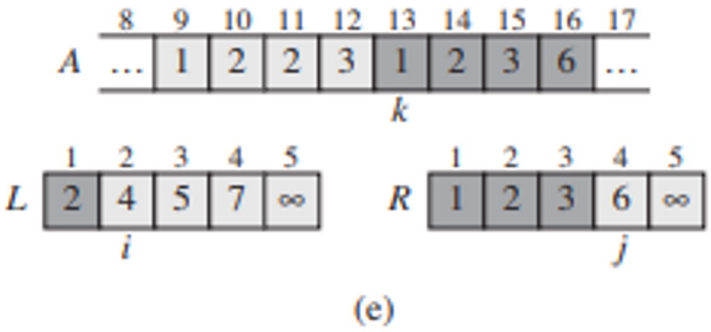
(b)



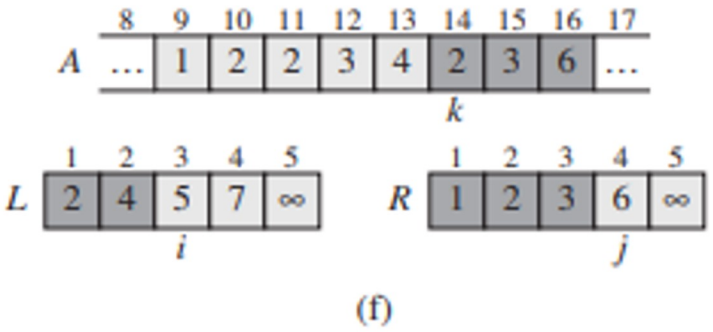
(c)



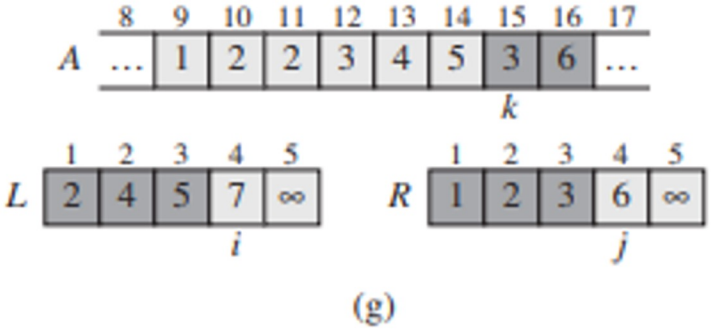
(d)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

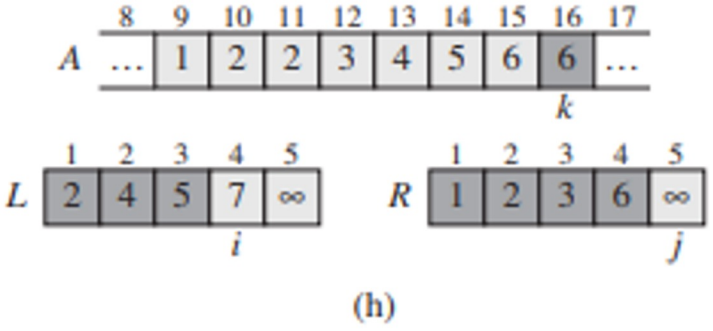
(e)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

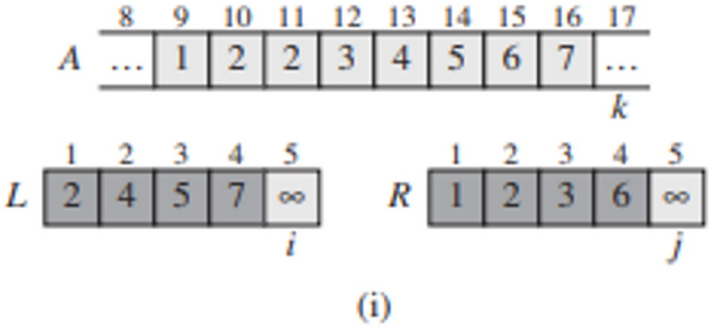
(f)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(g)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(h)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(i)

Algorithm: Merge-Sort

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Recurrence Relation of Merge Sort

$$T(n)=\begin{cases} 1 & n=1 \\ 2T(n/2)+D(n)+C(n) & n>1 \end{cases}$$

- It takes time T (n/b) to solve one subproblem of size n/b, and so it takes time aT (n/b) to solve a of them.
- D(n) time to divide the problem into subproblems and
- C(n) time to combine the solutions to the subproblems into the solution to the original problem.

Recurrence Relation of Merge Sort

$$T(n) = \begin{cases} c & , \text{if } n = 1 \\ 2T(n/2) + c'n & , \text{if } n > 1 \end{cases}$$

$$T(n) = 4T(n/4) + 2c'n$$

$$= 8T(n/8) + 3c'n$$

$$= 16T(n/16) + 4c'n$$

$$= 2^k T(n/2^k) + k \cdot c' \cdot n$$

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

$$\Rightarrow k = \log_2 n$$

$$= 2^{\log_2 n} T(1) + \log_2 n \cdot c' \cdot n$$

$$= nc + c' \cdot n \log n = \Theta(n \log n)$$

Analysis of Merge Sort

- Worst Case Time Complexity [Big-O]: **O(n*log n)**
- Best Case Time Complexity [Big-omega]: **O(n*log n)**
- Average Time Complexity [Big-theta]: **O(n*log n)**

Home Assignment

- A = {3, 41, 52, 26, 38, 57, 9, 49}
- B= { 38, 27, 43, 3, 9, 82,10}
- C= { 70, 20,30,40, 10, 50, 60}

Quick Sort

Introduction

- An efficient sorting algorithm
- Based on partitioning of array of **data** into smaller arrays
- Developed by British computer scientist Tony Hoare in 1959 and published in 1961
- Two or three times faster than other sorting algorithms
- Uses divide and conquer strategy
- Quicksort is a comparison sort
- Sometimes called **partition-exchange sort**

Three operations performed in quick sort

1. **Divide:** Select a 'pivot' element from the array and partition the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
2. **Conquer :** Recursively sort the two sub arrays
3. **Combine** : Combine all the sorted elements in a group to form a list of sorted elements.

How Quick sort works

- Select a PIVOT element from the array
- You can choose any element from the array as the pivot element.
- Here, we have taken the first element of the array as the pivot element.

34	42	16	71	9	22	53	13

- The elements smaller than the pivot element are put on the left and the elements greater than the pivot element are put on the right.

9	18	16	22	**34**	71	42	53

- Now sort the left and right arrays recursively.

Algorithm Quicksort

```
Quicksort(A, L,H)
{
// Array to be soerrted
// L – lower bound of array
// H- upper bound of array
If (L<H)
{
        P=Partition(A,L,H)
        Quicksort(A,L,P)
        Quicksort(A,P+1,H)
}
}
```

Algorithm Partition

```
Partition(A,L,H)
{
        pivot = A[L]
        i=L, j=H+1
         repeat
                repeat   i=i+1  until A[i]>=pivot
                repeat   j=j-1  until A[j]<=pivot
                swap(a[i],a[j])
         until (i>j)
         swap(A[L],A[j])
}
```

Method with Example (Partition)

- Numbers to be sorted

34	53	16	71	9	22	42	18

- Select 34(first element) as the pivot element

34	53	16	71	9	22	42	18

Find the element > 34 from the left, let its index be i

Find the element<34 from the right, let its index be j

34	**53**	16	71	9	22	42	**18**

34	**18**	16	71	9	22	42	**53**

- Do this process until the index I is greater than index j.

34	18	16	**71 i=4**	9	**22 j=6**	42	53

34	18	16	**22**	9	**71**	42	52

34	18	16	22	**9 j=5**	**71 i=6**	42	53

- I becomes > j , so exchange the pivot element and A(j)

9	18	16	22	**34**	71	42	53

- Note all the elements to the left of pivot is lesser and the elements to the right of pivot is larger than the pivot

Recurrence Relation of Quick Sort

$$T(n)=\begin{cases} 1 & n=1 \\ 2T(n/2)+cn & n>1 \end{cases}$$

Analysis of Quick Sort Algorithm

- If pivot element is either the largest or the smallest element
- Then one partition will have 0 elements and the other partition will have (n-1) elements.
- Eg : if pivot element is the largest element in the array. We will have all the elements except the pivot element in the left partition and no elements in the right partition
- The recursive calls will be on arrays of size 0 and (n-1)

Best Case Analysis

Best case:

$T(1) = c_1$

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

$= 2\left\{2T(n/4) + c \cdot \frac{n}{2}\right\} + c \cdot n$

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

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

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

$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

$= 2^{\log_2 n} \cdot T(1) + c \cdot n \cdot \log_2 n$

$= n \cdot c_1 + c \cdot n \cdot \log n$

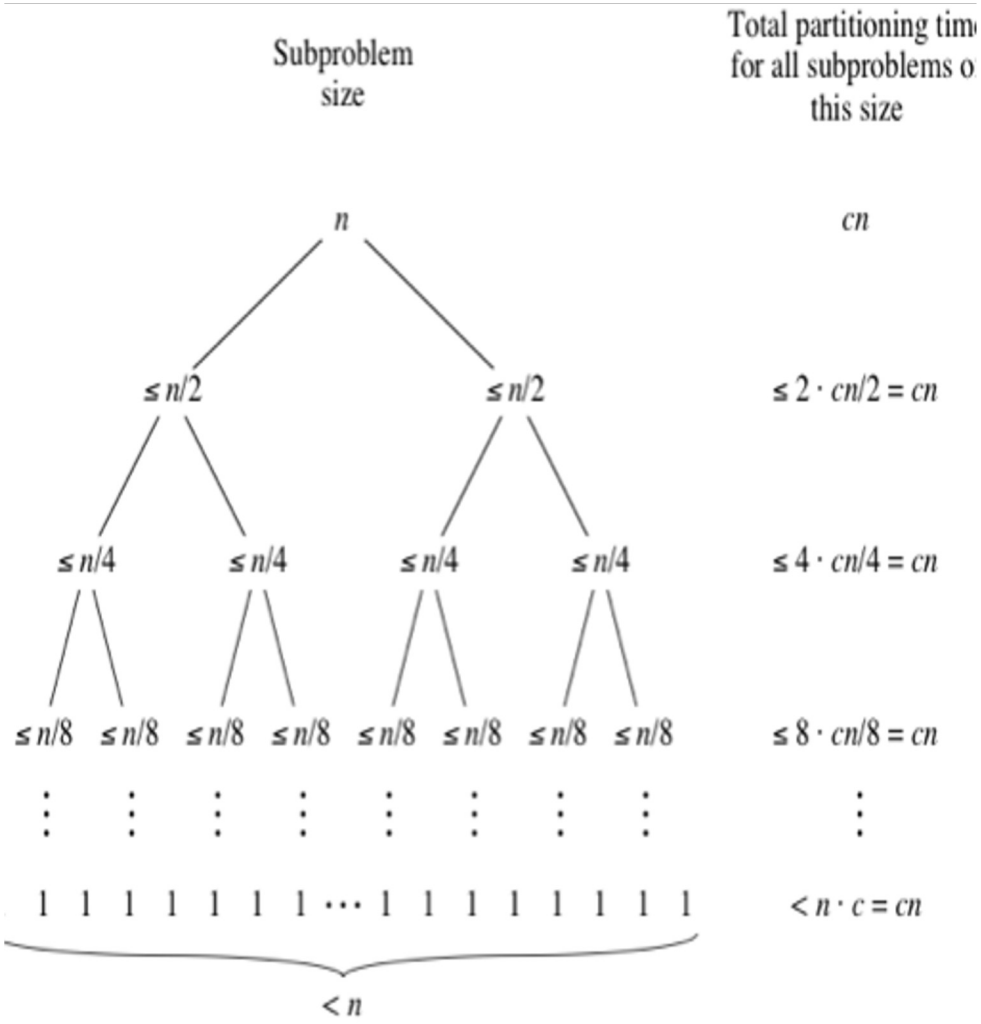
$O(n \log n)$

Quick Sort - Best-case running time

Quick Sort : Best-case running time

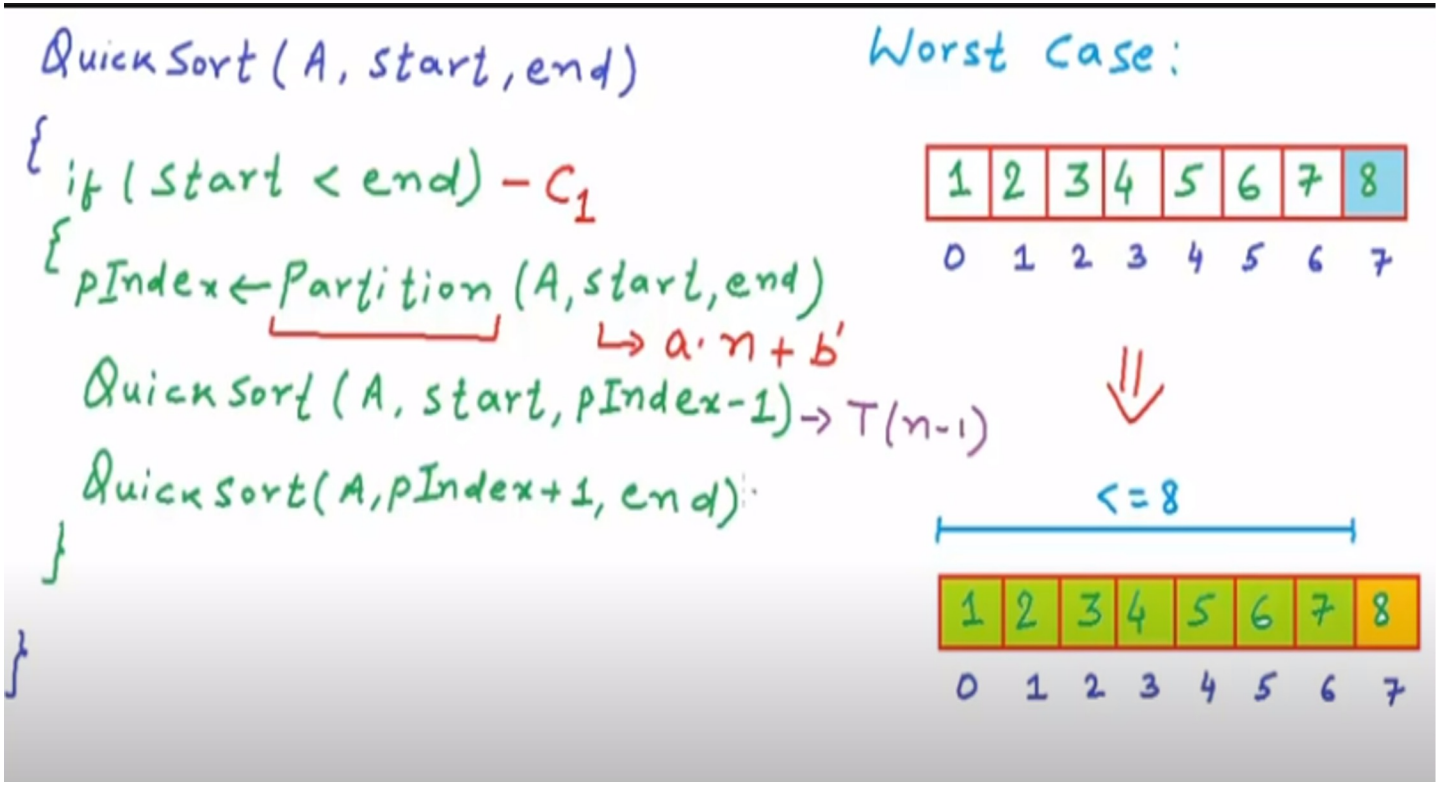
Occurs when the partitions are evenly partitioned.

quicksort's Best-case running time is $\Theta(n \log_2 n)$



Worst Case Analysis

- When the array is sorted and pivot is either first element or last element.



Quick Sort - Worst-case running time

$T(1) = c_1$

$T(n) = T(n-1) + c \cdot n$

$= T(n-2) + 2cn - c$

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

$= T(n-4) + 4cn - 6c$

$= T(n-k) + kcn - (1+2+3+\ldots+k-1)\,c$

$= T(n-k) + kcn - \frac{k(k-1)}{2} \cdot c$

$n - k = 1 \Rightarrow k = n$

$= T(1) + cn^2 - \frac{n(n-1)}{2} \cdot c$

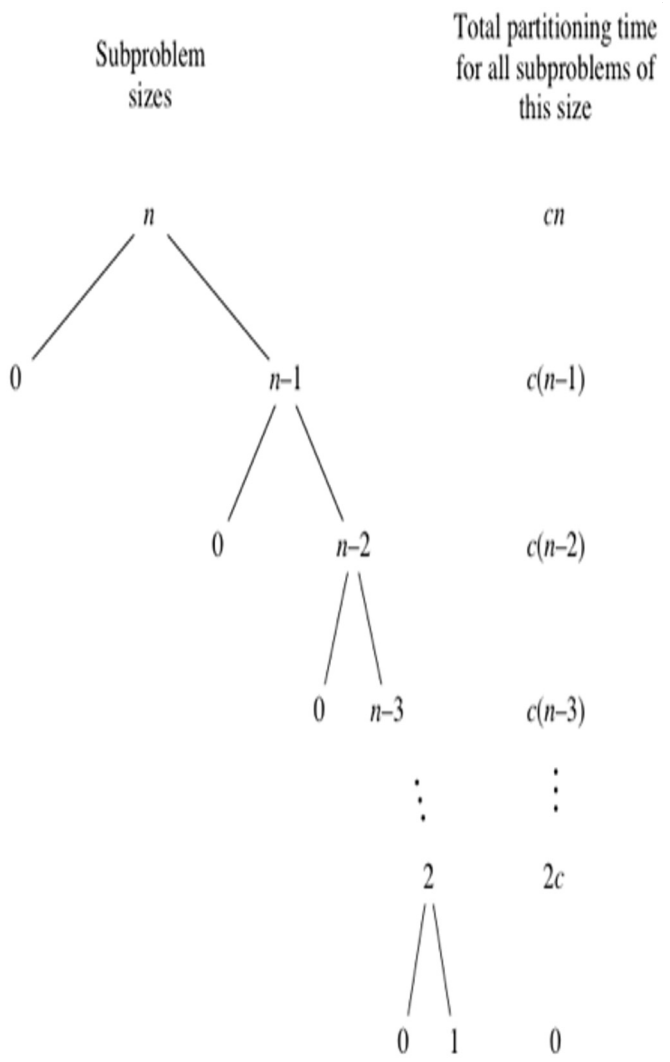
$T(n) = c_1 + \frac{cn \cdot (n+1)}{2}$

$= \frac{cn^2}{2} + \frac{cn}{2} + c_1$

$\Rightarrow O(n^2)$

Quick Sort - Worst-case running time

- Let the original call takes cn time for some constant c.
- Recursive call on n-1 elements takes c(n-1)times
- Recursive call on n-2 elements takes time c(n-2) times, and so on.
- Recursive call on 2 elements will take 2c times
- Finally recursive call on 1 element will take no time.



Worst-case running time

- Sum up the partitioning times, we get

$cn + c(n-1) + c(n-2) + \ldots + 2c$

$= c(n + (n-1) + (n-2) + \ldots + 2)$

$= c((n+1)(n/2) - 1)$

[Note : $1+2+\ldots+(n-1)+n=(n+1)n/2$

So $2+\ldots+(n-1)+n=((n+1)n/2)-1$]

quicksort's worst-case running time is $\Theta(n^2)$

Quick Sort – Average -case running time

Quick Sort : Average-case running time

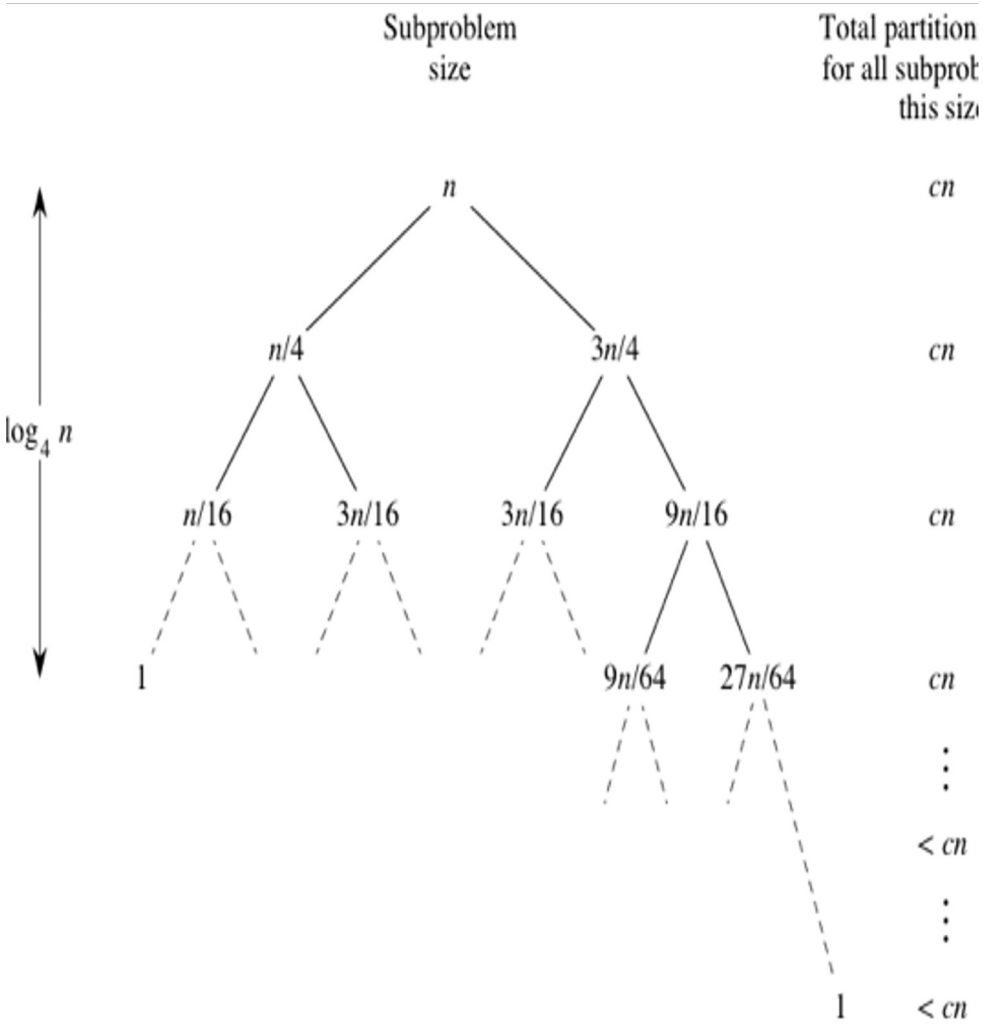
imagine that we don't always get evenly balanced partitions

but that we always get at worst a 3-to-1 split

That is, imagine that each time we partition, one side gets 3n/4 elements and the other side gets n/4 elements

quicksort's Average-case running time is

$$\Theta(n \log_2 n)$$



Quick Sort

Advantages of Quick Sort-

1. The advantages of quick sort algorithm are-
2. Quick Sort is an in-place sort, so it requires no temporary memory.
3. Quick Sort is typically faster than other algorithms.
4. (because its inner loop can be efficiently implemented on most architectures)
5. Quick Sort tends to make excellent usage of the memory hierarchy like virtual memory or caches.
6. Quick Sort can be easily parallelized due to its divide and conquer nature.

Disadvantages of Quick Sort-

1. The disadvantages of quick sort algorithm are-
2. The worst case complexity of quick sort is $O(n^2)$.
3. This complexity is worse than O(nlogn) worst case complexity of algorithms like merge sort, heap sort etc.
4. It is not a stable sort i.e. the order of equal elements may not be preserved.

Comparison

Algorithm	Time Complexity			
	Best	Average	Worst	
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	

Home assignment

- Sort the following numbers using Quick sort

- 21, 45, 32, 76, 12, 83, 47, 153, 52
- 75, 34, 64, 82,35,79,12,53,40,61