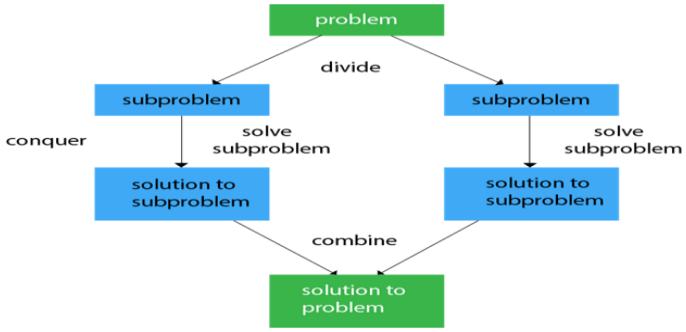
UNIT - IV	**Divide and conquer, Greedy method**

Divide and conquer: General method, applications-Binary search, Finding Maximum and minimum, Quick sort, Merge sort, Strassen's matrix multiplication.
Greedy method: General method, applications-Job sequencing with deadlines, knapsack problem, Minimum cost spanning trees, Single source shortest path problem.

Divide And Conquer

- Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide.**(2 Marks)**
- This technique can be divided into the following three parts:
  - Divide: This involves dividing the problem into smaller sub-problems.
  - Conquer: Solve sub-problems by calling recursively until solved.
  - Combine: Combine the sub-problems to get the final solution of the whole problem.



Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{
        if SMALL (P) then return S (p);
        else
        {
                divide p into smaller instances p1, p2, .... Pk, k ≥ 1;
                apply DANDC to each of these sub problems;
                return (COMBINE (DANDC (p1) , DANDC (p2),...., DANDC (pk));
        }
}
```

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

END

Binary Search

- Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.
- Binary search is an algorithm that uses divide and conquer method to search a element in a sorted list.
- It works by comparing the target to the middle element in a collection. If the target is greater than the middle element, the left elements are discarded. If the target is less than the middle element the right elements are discarded. This process is repeated until the middle element is equal to the target and if no further splitting can be done. The target does not exist in the list.

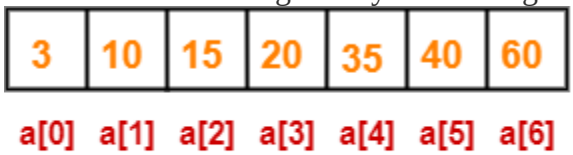
Algorithm:

```
Def BinarySearch (a, low, high, key)
        if  high>= low:
                mid = low+high/2
                if key == a[mid]:
                        return mid
                elif key< a[mid]:
                        return BinarySearch (a, low, mid-1, key)
                else
                        return BinarySearch (a, mid+1, high, key)
        else:
                return -1
        a = [3, 10, 15, 20,35, 40, 60]
        key = 15
        high = len(a)-1
        result = BinarySearch (a, 0, high, key)
        if result != -1:
                print(" element is found at index", mid)
        else:
                print(" element is not found in list")
```

Binary Search Example-

Consider-

- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.



Binary Search Example

Binary Search Algorithm works in the following steps-

Step-01:

- To begin with, we take beg=0 and end=6.
- We compute location of the middle element as-

mid

= (beg + end) / 2
= (0 + 6) / 2
= 3

- Here, a[mid] = a[3] = 20 ≠ 15 and beg < end.
- So, we start next iteration.

Step-02:

- Since a[mid] = 20 > 15, so we take end = mid – 1 = 3 – 1 = 2 whereas beg remains unchanged.
- We compute location of the middle element as-

mid
= (beg + end) / 2
= (0 + 2) / 2
= 1

- Here, a[mid] = a[1] = 10 ≠ 15 and beg < end.
- So, we start next iteration.

Step-03:

- Since a[mid] = 10 < 15, so we take beg = mid + 1 = 1 + 1 = 2 whereas end remains unchanged.
- We compute location of the middle element as-

mid
= (beg + end) / 2
= (2 + 2) / 2
= 2

- Here, a[mid] = a[2] = 15 which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

Binary Search complexity(2 Marks or 5 marks-Analysis slove recurrence relation)

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

- **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1).**
- **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
- **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

2. Space Complexity

- The space complexity of binary search is O(1).

Binary Search Applications

- In libraries of Java, .Net, C++ STL
- While debugging, the binary search is used to pinpoint the place where the error happens.

Finding MIN and MAX algorithm

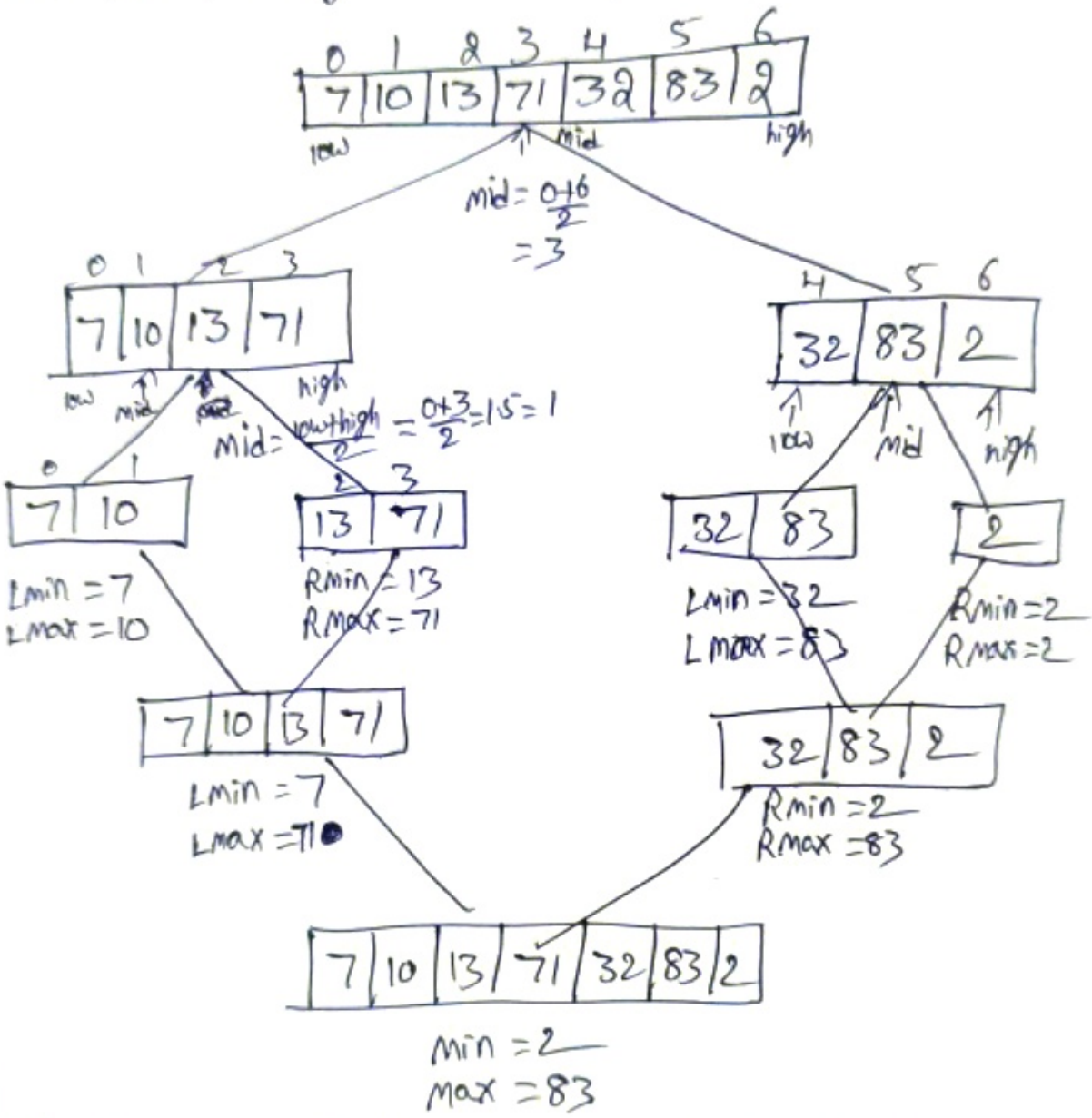
- Max-Min problem is to find a maximum and minimum element from the given array. We can effectively solve it using divide and conquer approach.
- In the traditional approach, the maximum and minimum element can be found by comparing each element and updating Max and Min values as and when required. This approach is simple but it does (n – 1) comparisons for finding max and the same number of comparisons for finding the min. It results in a total of 2(n – 1) comparisons. **Using a divide and conquer approach, we can reduce the number of comparisons.**
- Divide and conquer approach for Max. Min problem works in three stages.
    1. If a1 is the only element in the array, a1 is the maximum and minimum.
    2. If the array contains only two elements a1 and a2, then the single comparison between two elements can decide the minimum and maximum of them.
    3. If there are more than two elements, the algorithm divides the array from the middle and creates two subproblems. Both subproblems are treated as an independent problem and the same recursive process is applied to them. This division continues until subproblem size becomes one or two.
- After solving two subproblems, their minimum and maximum numbers are compared to build the solution of the large problem. This process continues in a bottom-up fashion to build the solution of a parent problem.

Algorithm:

```
Algorithm MinMax(A, low, high) {
  // if list a has one element
   If (low == high)
         min = max = a[low]
         return (min, max)
   // if list a has two elements
   if ((high - low) == 1)
         if (A[low] < A[high])
                max = a[high]
                min = a[low]
          else
                max = a[low]
                min = a[high]
          return (min, max)
   else
        mid = (low + high) / 2;
        LMin, LMax = MinMax(A, low, mid);
        RMin, RMax = MinMax(A, mid + 1, high);
        if(LMax > RMax)
              max = LMax;
        else
              max = RMax;
        if(LMin < RMin)
              min = LMin;
        else
              min = RMin;
        return (min, max);
```

* Example:-

Find min & max by using Divide & conquer for the following list 7, 10, 13, 71, 32, 83, 2



* Recurrence relation for min & max Problem

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

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

sub n/2 in place of n in (1)

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

sub n/4 in place of n in (1)

$T(n/4) = 2T(n/8)+2$ - (3)

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

In place of $T(n/2)$ sub (2)

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

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

In place of $T(n/4)$ sub (3)

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

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

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

At k-1 step

$T(n) = 2^{k-1} T(n/2^{k-1}) + 2^{k-1} + 2^{k-2} + 2^{k-3} + \dots$

$= 2^{k-1} T(n/2^{k-1}) + \sum_{i=1}^{k-1} 2^i$

where $\sum_{i=1}^{k-1} 2^i = 2^k - 2$

$= 2^{k-1} T(n/2^{k-1}) + 2^k - 2 \quad -(4)$

$2^k = n \Rightarrow 2^{k-1} = n/2$

sub 2^{k-1} in (4)

$= n/2\, T(n/n/2) + n - 2$

$= n/2\, T(2^n/n) + n - 2$

$= n/2\, T(2) + n - 2$ where $T(2) = 1$

$= n/2\,(1) + n - 2$

$= n/2 + n - 2$

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

* Space complexity $= O(1)$

Time complexity $= O(n)$

MERGE SORT

- Merge sort is one of the most popular sorting algorithm that is based on the principle of divide and conquer
- Here a problem is divided in to multiple sub problems. Each sub problems is solved individually and finally sub problems are combined to form the final solution
- Merge sort works in 3 stages
    1. **Divide:** Recursively divide the single list in to two sublists until each sublist contains one element
    2. **Conquer:** sort both sublists of parent list
    3. **Combine:** Recursively combine the sorted sublist until the list of size 'n' is produced

Merge sort algorithm time complexity analysis

- Let's assume that T(n) is the worst-case time complexity of merge sort for n integers. We can break down the time complexities as follows:
    1. **Divide part:** The time complexity of the divide part is O(1), because calculating the middle index takes constant time.
    2. **Conquer part:** We recursively solve two sub-problems, each of size n/2. So the time complexity of each subproblem is T(n/2), and the overall time complexity of the conquer part is 2T(n/2).
    3. **Combine part:** As calculated above, the worst-case time complexity of the merging process is O(n).
- To calculate T(n), we need to add up the time complexities of the divide, conquer, and combine parts:
  $T(n) = O(1) + 2T(n/2) + O(n) = 2T(n/2) + O(n) = 2T(n/2) + cn$
- We can use the following formulas to calculate T(n):
  $T(n)=c$, if $n=1$
  $T(n) = 2T(n/2) + cn$, if $n > 1$

Time Complexity

- Best Case Complexity: O(n*log n)
- Worst Case Complexity: O(n*log n)
- Average Case Complexity: O(n*log n)

Space Complexity

- The space complexity of merge sort is O(n).

Applications of Merge Sort:

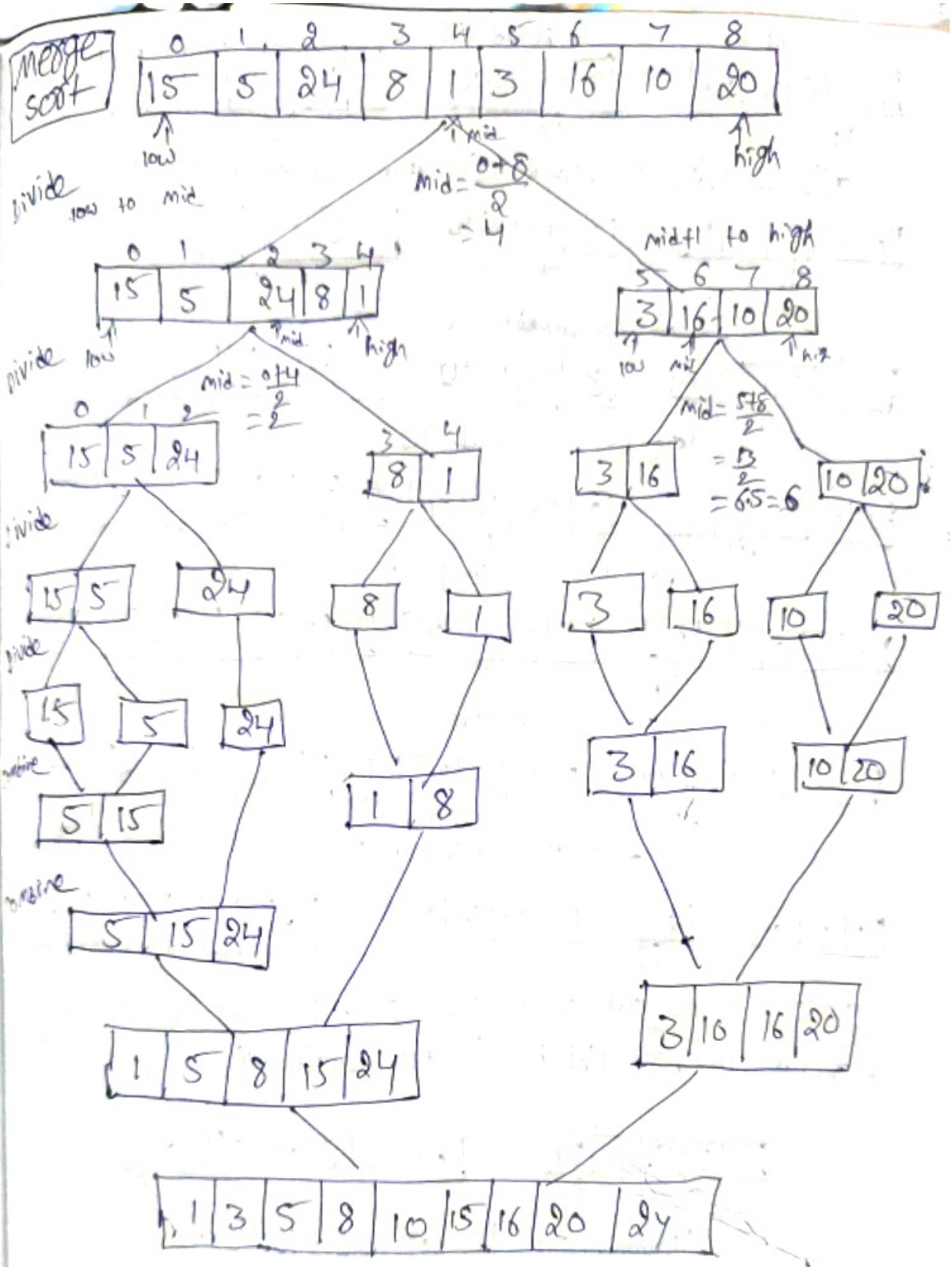
- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
- Inversion Count Problem

Advantages of Merge Sort:

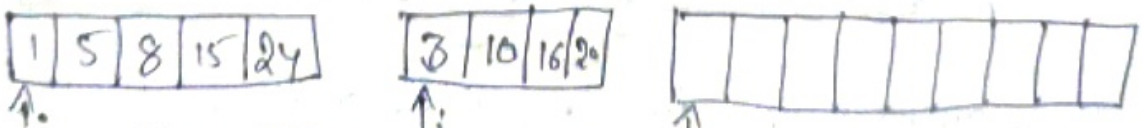
- **Stability**: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
- **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

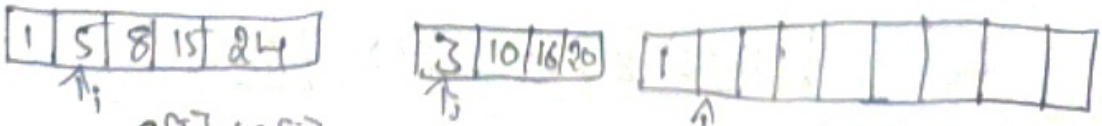
- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.



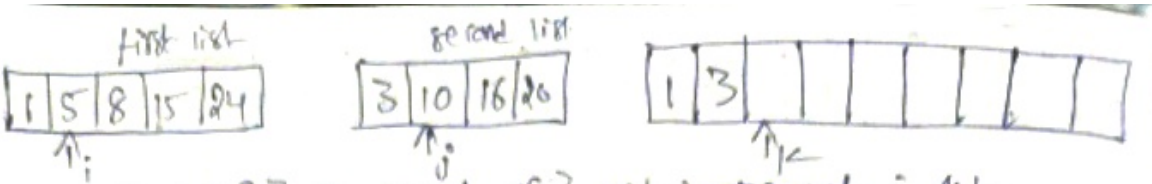
combine process:-



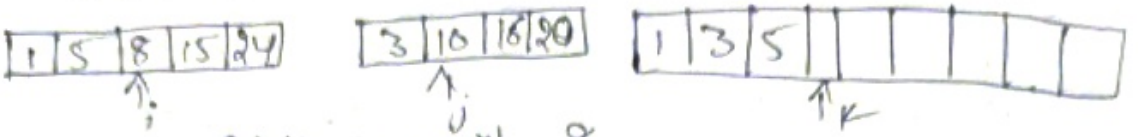
a[i] < a[j] so insert a[i] and increment i & k



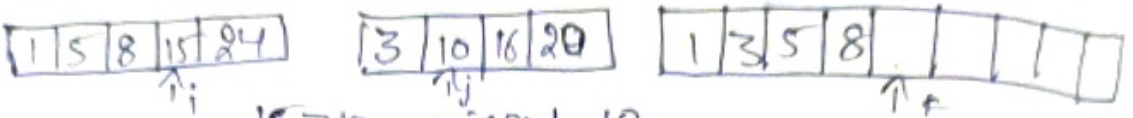
a[j] < a[i] so insert a[j] and increment j & k



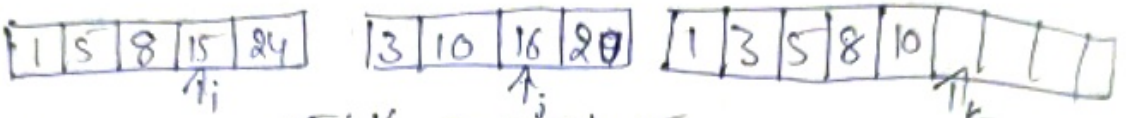
a[i] < a[j] so insert a[i] and increment i & k



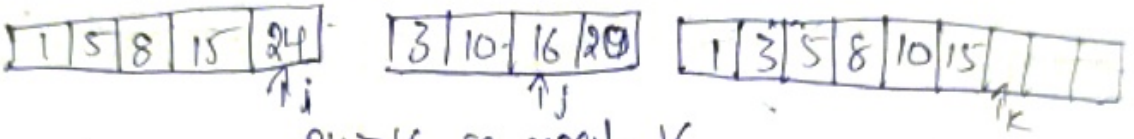
8 < 10 so insert 8



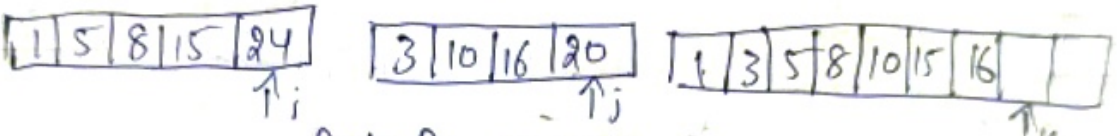
15 > 10 so insert 10



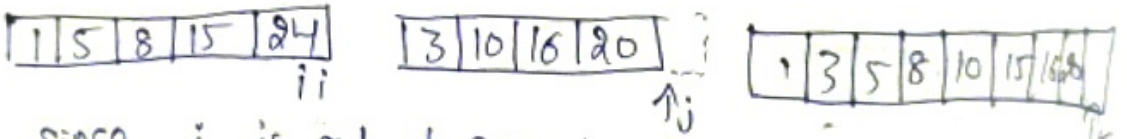
15 < 16 so insert 15



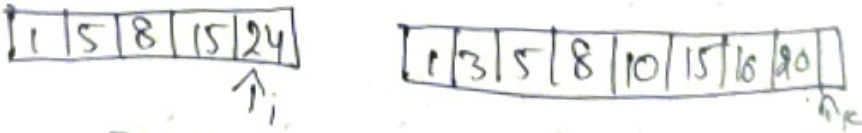
24 > 16 so insert 16



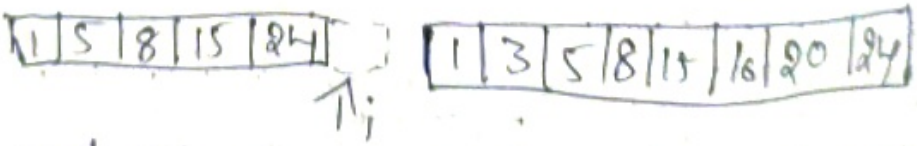
24 > 20 so insert 20



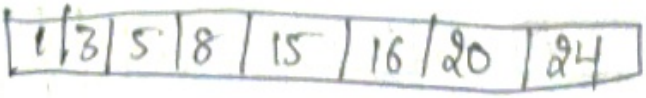
Since j is out of second list. so copy the remaining elements in first list i.e i to main list ie k



Insert 24 to combined list & increment i



Now first list also completed. so the final combined list



Algorithm: mergesort

```
Algorithm mergesort (A, low, high)
    if low < high
        mid = low+high//2
        mergesort (A, low, mid)
        mergesort (A, mid+1, high)
        combine (A, low, mid, high)
    stop mergesort
```

It is used to divide the list until single element

```
Algorithm combine (A, low, mid, high)
    i = low
    j = mid+1
    k = low
    while i<=mid and j<=high
    {
        if a[i]<a[j]
        {
            b[k]=a[i]
            i++
        }
        else
        {
            b[k]=a[j]
            j++
        }
        k++
    }
    if (i>mid)
    {
        while (j<=high)
        {
            b[k]=a[j]
            j++
            k++
        }
    }
```

```
else
{
    while (i<=mid)
    {
        b[k]=a[i]
        i++
        k++
    }
}
```

Stop combine.

QUICK SORT:

Quicksort is a sorting algorithm based on the **divide and conquer approach** where

1. An array is divided into subarrays by selecting a pivot element (element selected from the array) i.e first element of list or last element of list or any other element. While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

Less than the Pivot	PIVOT	Greater than the Pivot

2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Choice of Pivot:

- There are many different choices for picking pivots.
    - Always pick the first element as a pivot.
    - Always pick the last element as a pivot (implemented below)
    - Pick a random element as a pivot.
    - Pick the middle as the pivot.

Time Complexities(2 Marks)

- **Worst Case Complexity [Big-O]**: $O(n^2)$
  It occurs when the pivot element picked is either the greatest or the smallest element. This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains $n-1$ elements.
- **Best Case Complexity [Big-omega]**: $O(n*\log n)$
  It occurs when the pivot element is always the middle element or near to the middle element.
- **Average Case Complexity [Big-theta]**: $O(n*\log n)$
  It occurs when the above conditions do not occur.

Space Complexity(2 Marks)

- The space complexity for quicksort is $O(\log n)$.

Quicksort Applications(2 Marks)

- Quicksort algorithm is used when
    - the programming language is good for recursion
    - time complexity matters
    - space complexity matters

Advantages of Quick Sort(2 Marks):

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort(2 Marks):

- It has a worst-case time complexity of O(N2), which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Quick sort

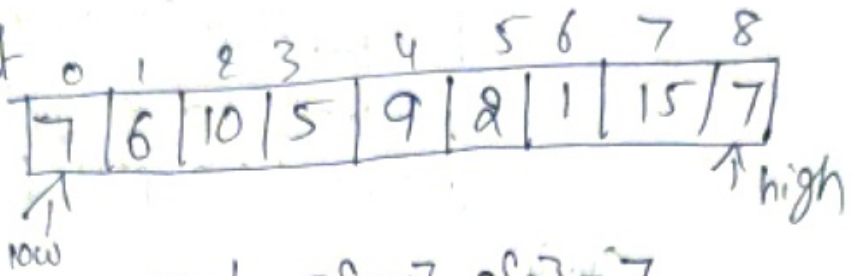
```
Algorithm quicksort(a, low, high)
    if low < high then
        Pi = partition (a, low, high)
        quicksort(a, low, Pi-1)
        quicksort(a, Pi+1, high)
    end if
end quicksort

Algorithm partition (a, low, high)
    pivot = low
    start = low+1
    end = high
    while (start < end)
    {
      while (a[start] <= pivot)
      {
        start++;
      }
      while (a[end] > pivot)
      {
        end--;
      }
      if (start < end)
      {
        swap(a[start], a[end]);
      }
    }
    swap(a[low], a[end])
    return end
stop partition
```

Quick sort Example:

Sort the following unsorted list using Quick sort

7, 6, 10, 5, 9, 2, 1, 15, 7

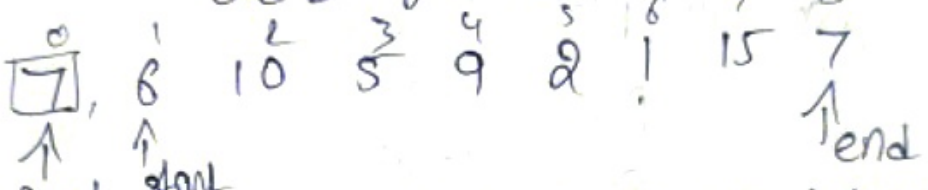
Given list



pivot = a[low] = a[0] = 7

start = low+1 = 0+1 = 1

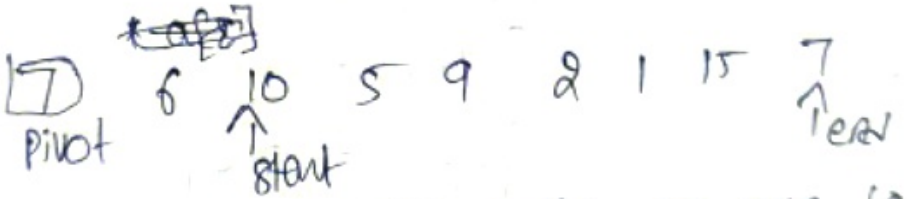
end = high = 8



a[start] <= pivot is true then increment start it false then stop incrementing start and go to end

a[start] <= pivot

6 <= 7 true increment start 1 to 2



10 <= 7 false so move to end and check a[end] > pivot if it is true then decrement end, else stop decrement end

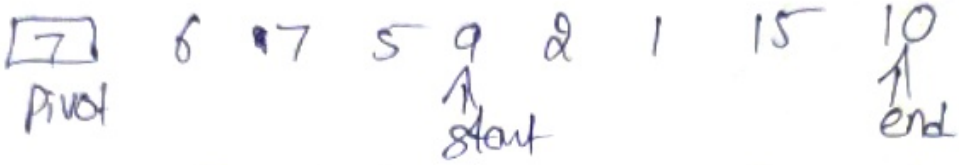
a[end] > pivot

7 > 7 false

* Now check start < end if true swap a[start] and a[end]
* so, Now swap a[end] i.e 7 with a[start] i.e 10

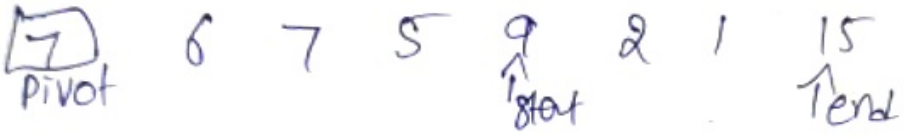
7<=7 true increment start

5<=7 true increment start



9<=7 false so, move to end

10>7 true decrement end



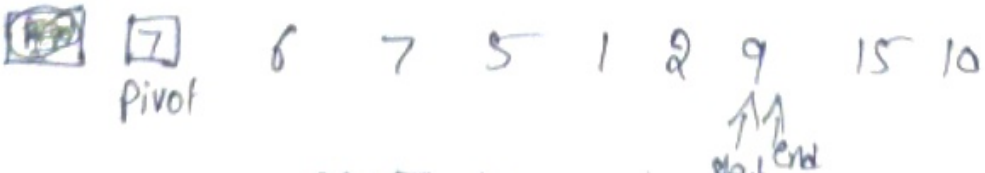
15>7 true decrement end

1>7 false now check start<end true so swap a[end] i.e 1 with a[start] i.e 9

1<=7 true start increment

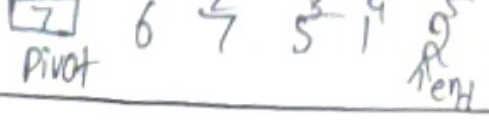
2<=7 true increment start

9<=7 false stop increment start



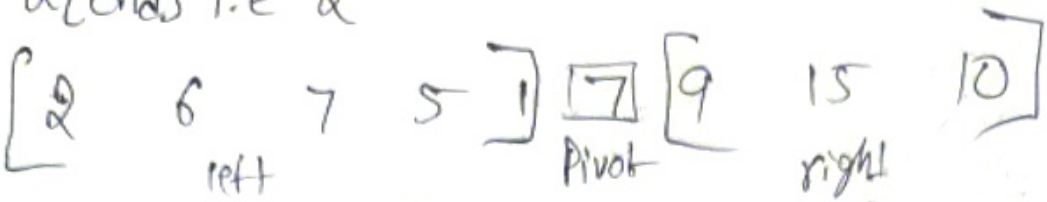
9>7 true decrement end

2>7 false stop decrement end



Now start < end false. So, swap Pivot ie 7 with a[end] i.e 2

6 < 5



x Here Pivot left are less than pivot and right are greater than pivot.

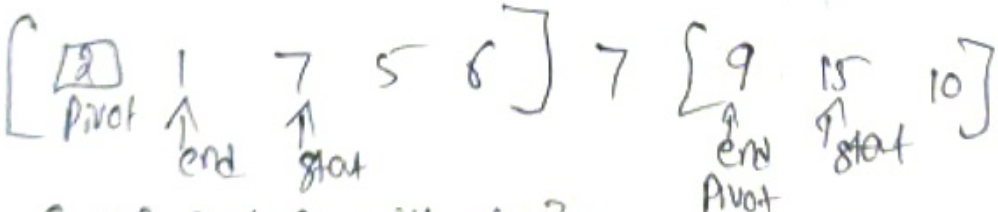
* Again same process for left list and right list

6 <= 2 false, 1 > 2 false
so swap 6 & 1

15 <= 9 false, 10 > 7 true
decrement end

1 <= 2 true
7 <= 2 false, 6 > 2 true
5 > 2 true
7 > 2 true
1 > 2 false

15 > 7 true decrement end



Swap Pivot 2 with a[end] 1

swap Pivot 9 with a[end] 9

[1] 2 [7 5 6] 7 , 9 [15, 10]

Pivot

Pivot

Finally After final step

1 2 5 6 7 7 9 10 15

5. Strassen's Matrix Multiplication:

- Strassen in 1969 gave an overview to find the multiplication of 2×2 Matrix by Brute-force algorithm.
- previous divide and conquere. for 4×4 matrix multiplication it uses divides 8 sub-Matrix and then recursively compute the sub matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

A1, A2, A3, A4 and B1, B2, B3, B4 are the corresponding sub-matrices of A and B.

- For above Multiplication, it requires 8 recursive calls. so final Time complexity is $T(n) = 8T(n/2) + n^2$ → Master method
- But Strassen comes with a solution where we don't need 8 recursive calls but we can done in only 7 calls and some extra addition & subtraction operations.
- Time complexity of strassen's $T(n) = 7T(n/2) + n^2$

19/12 • Strassen has used some formulas for multiplying the 2x2 Matrix

• For example, Let us consider 2 matrices A & B of nxn dimension where n is the power of 2.

Procedure for strassens Matrix Multiplication:

1. Divide a matrix of order 2x2 recursively until we get the matrix of order 2x2.
2. To carry out the multiplication of 2x2 matrix, we use the set of formulas.
3. To find the final matrix, combine the result of 2x2 matrices.

Formulas for strassen's Matrix Multiplication

The following are the seven formulas used in strassen's Matrix Multiplication.

1. $P = (A_{11} + A_{22}) * (B_{11} + B_{22})$
2. $Q = (A_{21} + A_{22}) * B_{11}$
3. $R = (B_{12} - B_{22}) * A_{11}$
4. $S = (B_{21} - B_{11}) * A_{22}$
5. $T = (A_{11} + A_{12}) * B_{22}$
6. $U = (A_{21} - A_{11}) * (B_{11} + B_{12})$
7. $V = (A_{12} - A_{22}) * (B_{21} - B_{22})$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$C_{11} = P+S-T+V$

$C_{12} = R+T$

$C_{21} = Q+S$

$C_{22} = P+R-Q+U.$

Example:- $a = \begin{bmatrix} a_{11} & a_{12} \\ 2 & 2 \\ 2 & 2 \\ a_{21} & a_{12} \end{bmatrix}$ $b = \begin{bmatrix} b_{11} & b_{12} \\ 2 & 2 \\ 2 & 2 \\ b_{21} & b_{22} \end{bmatrix}$

$P = (A_{11}+A_{22})*(B_{11}+B_{22}) = 4\times4 = 16$

$Q = (A_{21}+A_{22})*B_{11} = 4*2 = 8$

$R = (B_{12}-B_{22})*A_{11} = 0\times2 = 0$

$S = (B_{21}-B_{11})*A_{22} = 0\times2 = 0$

$T = (A_{11}+A_{12})*B_{22} = 4\times2 = 8$

$U = (A_{21}-A_{11})*(B_{11}+B_{12}) = 0*4 = 0$

$V = (A_{12}-A_{22})*(B_{21}+B_{22}) = 0\times0 = 0$

$C_{11} = P+S-T+V$
$= 16+0-8+0$
$= 8$

$C_{12} = R+T$
$= 0+8 = 8$

$C_{21} = Q+S$
$= 8+0 = 8$

$C_{22} = P+R-Q+U$
$= 16+0-8+0$
$= 8$

$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix}$

Analysis of TIME complexity (or) Recurrence Relation of Strassen's

$T(n) = 7T(n/2) + n^2 \rightarrow (1)$

$T(n/2) = 7T(n/4) + \frac{n^2}{4} \rightarrow (2)$

$T(n/4) = 7T(n/8) + \frac{n^2}{16} \rightarrow (3)$

Take

$T(n) = 7T(n/2) + n^2$

$T(n) = 7\left(7T(n/4) + \frac{n^2}{4}\right) + n^2$

$= 49T(n/4) + \frac{7n^2}{4} + n^2$

$T(n) = 49\left(7T(n/8) + \frac{n^2}{16}\right) + \frac{7n^2}{4} + n^2$

$= 343T(n/8) + \frac{49n^2}{16} + \frac{7n^2}{4} + n^2$

$= 7^3T(n/2^3) + n^2\left[\frac{49}{16} + \frac{7}{4} + 1\right]$

$T(n) = 7^3 T(n/2^3) + n^2\left(1 + \frac{7}{4} + \left(\frac{7}{4}\right)^2 + \cdots\right)$

At k th step,

$T(n) = 7^k T(n/2^k) + n^2 (7/4)^k$

$2^k = n$

$\log 2^k = \log n$

$\log n = k \log 2$

$\log n = k$

$\boxed{k = \log n}$

$T(n) = 7^{\log n} T(n/n) + n^2 (7/4)^{\log n}$

$T(n) = 7^{\log n} T\left(\frac{2^k}{2^k}\right) + n^2 (7/4)^{\log n}$

$= 7^{\log n} T(1) + n^2 \cdot \left(\frac{7^{\log n}}{4^{\log n}}\right)$

$= 7^{\log n} + n^2 \left(\frac{7^{\log n}}{4^{\log n}}\right)$

$= 7^{\log n} + n^2 \left(\frac{7^{\log n}}{n^2}\right)$

$4^{\log n} = n^{\log 4} = n^{\log 2^2}$

$= n^{\log 7} + 7^{\log n}$

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

$= O(n^{2.81})$

Remaining First part Refer from Notes

Some Uncovered Topics in Class Notes

1. Difference between Quick sort and Merge sort

QUICK SORT	MERGE SORT
Splitting of array depends on the value of pivot and other array elements	Splitting of array generally done on half
Worst-case time complexity is O(n2)	Worst-case time complexity is O(nlogn)
It takes less n space than merge sort	It takes more n space than quick sort
It works faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
It is in-place	It is out-place
Not stable	Stable

Basis for comparison	Quick Sort	Merge Sort
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	The splitting of a array of elements is in any ratio, not necessarily divided into half.
Worst case complexity	O(n2)	O(nlogn)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor

2. Explain complexity analysis of Quick Sort?

Complexity Analysis

Time Complexity of Quick sort

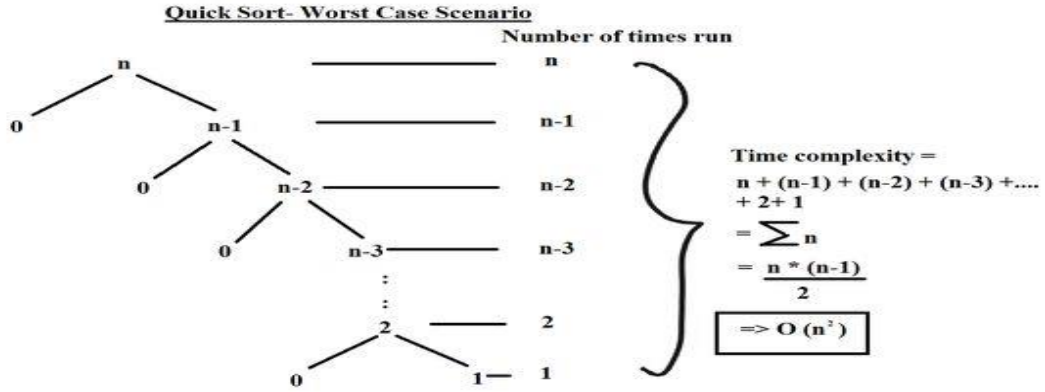
Best case scenario:

- The best case scenario occurs when the partitions are as evenly balanced as possible, i.e their sizes on either side of the pivot element are either are equal or are have size difference of 1 of each other.
- The following is recurrence for the best case.

- $T(n) = 2T(n/2) + (n)$
- **The solution for the above recurrence is (n.logn).**

Worst case scenario:

- The worst case occurs when the partition process always picks the greatest or smallest element as the pivot.



- Following is recurrence for the worst case.

  $T(n) = T(0) + T(n-1) + (n)$ *which is equivalent to* $T(n) = T(n-1) + (n)$

  The solution to the above recurrence is (n^2).

Average case scenario:

Same as best case

UNIT 4 Part 2
Greedy Method

- The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results.
- A greedy method is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result**.(2 Marks)**
- For a given problem it selects the subset that satisfies the constraints given is called a *feasible* solution. We are required to find a feasible solution that either maximizes or minimizes a given *objective function.* A feasible solution that does this is called an *optimal solution.*

Greedy General method Algorithm: (2 Marks)

```
procedure GREEDY(A,n)
   //A(1:n) contains the n inputs//
   solution ← φ   //initialize the solution to empty//
   for i ← 1 to n do
      x ← SELECT(A)
      if FEASIBLE(solution,x)
         then solution ← UNION(solution,x)
      endif
   repeat
   return (solution)
end GREEDY
```

The function SELECT selects an input from *A*, removes it and assigns its value to *x*. FEASIBLE is a Boolean-valued function which determines if *x* can be included into the solution vector. UNION actually combines *x* with solution and updates the objective function

Characteristics of Greedy method

The following are the characteristics of a greedy method:

- To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
- A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Components of Greedy Algorithm

The components that can be used in the greedy algorithm are:

- **Candidate set:** A solution that is created from the set is known as a candidate set.
- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
- **Feasibility function:** A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- **Objective function:** A function is used to assign the value to the solution or the partial solution.
- **Solution function:** This function is used to intimate whether the complete function has been reached or not.

Applications of Greedy Algorithm(2 Marks)

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in a job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

Advantages of Greedy Approach

- The algorithm is **easier to describe**.
- This algorithm can **perform better** than other algorithms (but, not in all cases).

Drawback of Greedy Approach

- The greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm
- The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct.

List of Algorithms based on Greedy Algorithm(2 Marks)

- The greedy algorithm is quite powerful and works well for a wide range of problems. Many algorithms can be viewed as applications of the Greedy algorithms, such as :
  - Knapsack Problem
  - Minimum Spanning Tree
  - Single-Source Shortest Path Problem
  - Job Scheduling Problem
  - Prim's Minimal Spanning Tree Algorithm
  - Kruskal's Minimal Spanning Tree Algorithm
  - Dijkstra's Minimal Spanning Tree Algorithm
  - Huffman Coding
  - Ford-Fulkerson Algorithm

END

Job sequencing with deadlines(5 Marks)

- In the Job Sequencing with Deadlines problem, the objective is to find a sequence or the order of the jobs, which is completed within their deadlines and gives maximum profit Profit earned only if the job is completed on or before its deadline, given the condition that we are using a single processor OS (operating system) which is capable of performing one job at a time.
- **Job Sequencing with Deadlines** problem uses the greedy approach. So we have to find the best method/option in the greedy method out of many present ways.

Approach to Solution-

- A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.
- Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.
- An optimal solution of the problem would be a feasible solution which gives the maximum profit.

Algorithm for Job Scheduling

Algorithm for job scheduling is described below:

Algorithm JOB_SCHEDULING(J, D, P)
// Input :
J: Array of N jobs
D: Array of the deadline for each job
P: Array of profit associated with each job
Sort all jobs in J in decreasing order of profit
S ← Φ // S is set of scheduled jobs, initially it is empty
SP ← 0 // Sum is the profit earned
for i ← 1 to N **do**
  if Job J[i] is feasible **then**
    Schedule the job in the latest possible free slot meeting its deadline.
    S ← S ∪ J[i]
    SP ← SP + P[i]
  e**nd if**
end for
end JOB_SCHEDULING

Example:

Problem: Solve the following instance of "job scheduling with deadlines" problem : n = 7, profits (p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (3, 5, 20, 18, 1, 6, 30) and deadlines (d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2). Schedule the jobs in such a way to get maximum profit.

Solution:

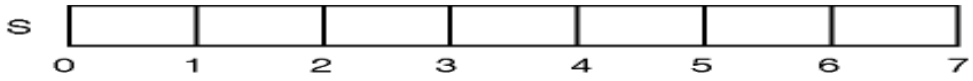
Given that,

Jobs	j_1	j_2	j_3	j_4	j_5	j_6	j_7
Profit	3	5	20	18	1	6	30
Deadline	1	3	4	3	2	1	2

- Sort all jobs in descending order of profit.

Jobs	j_7	J_3	J_4	J_6	J_2	J_1	J_5
Profit	30	20	18	6	5	3	1
Deadline	2	4	3	1	3	1	2

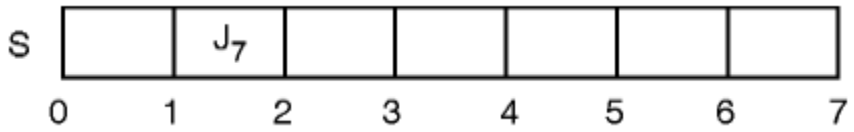
- We shall select one by one job from the list of sorted jobs J, and check if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one. Initially,



Profit of scheduled jobs, SP = 0

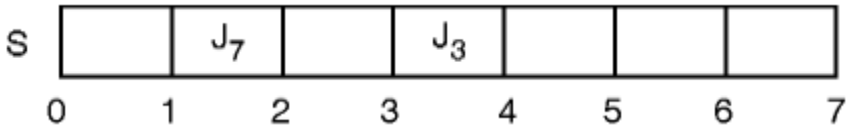
Iteration 1:

Deadline for job J_7 is 2. Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2. Solution set S = {J_7}, P=30, and Profit SP = SP+P SP= 0+30 =30



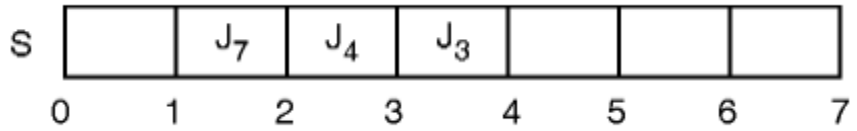
Iteration 2:

Deadline for job J_3 is 4. Slot 4 (t = 3 to t = 4) is free, so schedule it in slot 4. Solution set S = {J_7, J_3}, P=20 and Profit SP = SP+P SP=30+20 =50



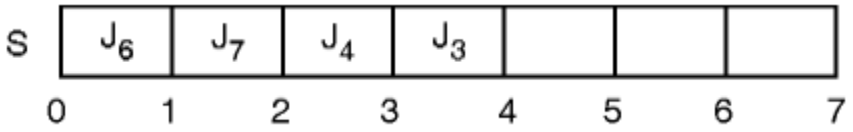
Iteration 3:

Deadline for job J_4 is 3. Slot 3 (t = 2 to t = 3) is free, so schedule it in slot 3. Solution set S = {J_7, J_3, J_4}, P= 18 and Profit SP = 50+18 = 68



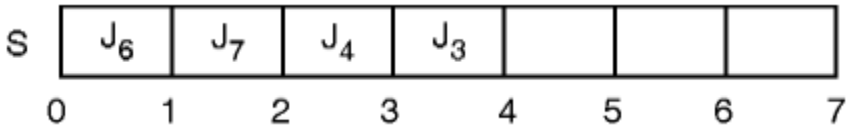
Iteration 4:

Deadline for job J_6 is 1. Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1. Solution set S = {J_7, J_3, J_4, J_6}, P=6 and Profit SP = 68+6 = 74



Iteration 5:

Deadline for job J_2 is 3. Slot 3 (t = 2 to t = 3) is not free, so Skip J2



Iteration 6:

Deadline for job J_1 is 1. Slot 1 (t = 0 to t = 1) is not free, so Skip J1

Iteration 7:

Deadline for job J_5 is 2. Slot 2 (t = 1 to t = 2) is not free, so Skip J5

- First, all four slots are occupied and none of the remaining jobs has deadline lesser than 4. So none of the remaining jobs can be scheduled.
- Thus, with the greedy approach, we will be able to schedule four jobs $\{J_7, J_3, J_4, J_6\}$, which give a profit of (30 + 20 + 18 + 6) = 74 units.

Complexity Analysis of Job Scheduling

Simple greedy algorithm spends most of the time looking for the latest slot a job can use. On average, N jobs search N/2 slots. This would take $O(N^2)$ time.

******END******

Fractional Knapsack Problem(5 Marks)

- The fractional knapsack problem is also one of the techniques which are used to solve the knapsack problem. In fractional knapsack, the items are broken in order to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

This problem can be solved with the help of using two techniques:

- **Brute-force approach:** The brute-force approach tries all the possible solutions with all the different fractions but it is a time-consuming approach.
- **Greedy approach:** In Greedy approach, we calculate the ratio of profit/weight, and accordingly, we will select the item. The item with the highest ratio would be selected first.Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

There are basically three approaches to solve the problem:

- The first approach is to select the item based on the maximum profit.
- The second approach is to select the item based on the minimum weight.
- The third approach is to calculate the ratio of profit/weight (Greedy Method)

NOTE: But third approach i.e., Greedy method will give maximum profit when compared to First and Second Approach

Knapsack using Greedy Method

Now, let us try to apply the greedy method to solve a more complex problem. This problem is the knapsack problem. We are given n objects and a knapsack. Object i has a weight w_i and the knapsack has a capacity M. If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is M, we require the total weight of all chosen objects to be at most M. Formally, the problem may be stated as:

$$\text{maximize} \sum_{1 \leq i \leq n} p_i x_i$$

$$\text{subject to} \sum_{1 \leq i \leq n} w_i x_i \leq M$$

$$\text{and} \quad 0 \leq x_i \leq 1, \quad 1 \leq i \leq n$$

The profits and weights are positive numbers.

Algorithm:

Follow the given steps to solve the problem using the above approach:

- Start
- Calculate the ratio(value/weight) for each item.
- Sort all the items in decreasing order of the ratio.
- Initialize **res =0**, curr_cap = given_cap.
- Do the following for every item "i" in the sorted order:
    - If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
    - Else add the current item as much as we can and break out of the loop.
- Return **res**.
- Stop

Example:

Consider the knapsack instances, the capacity of the knapsack is M= 15, and 7 object has profit and weight as follows:

Object	**X_1**	**X_2**	**X_3**	**X_4**	**X_5**	**X_6**	**X_7**
Weight	2	3	5	7	1	4	1
Profit	10	5	15	7	6	18	3

Solution:

Given, capacity of the knapsack (M)=15, weights and profit

Let us first find each object's profit by weight ratio.

object	X_1	X_2	X_3	X_4	X_5	X_6	X_7
Profit/weight	10/2	5/3	15/5	7/7	6/1	18/4	3/1
Profit/weight	5	1.66	3	1	6	4.5	3

Now we will arrange the profit by weight ratio in descending order and pick the objects accordingly.

X_5	X_1	X_6	X_7	X_3	X_2	X4
6	5	4.5	3	3	1.6	1

In this approach, we will select the objects based on the profit/weight ratio in descending order

Object	**Profit**	**Weight**	**Remaining weight= M-weight**
X5	6	1	15 - 1 = 14
X1	10	2	14 - 2 = 12
X6	18	4	12 - 4 = 8
X7	3	1	8 -1 = 7
X3	15	5	7 -5 = 2
X_2	5	(2/3)*3	2-2 =0
X_4			

As we can observe in the above table that the remaining weight is zero which means that the knapsack is full. We cannot add more objects in the knapsack. Therefore, the total profit would be equal to (6 +10 + 18 + 3 + 15 + 5), i.e., **57.**

Complexity Analysis(2 Marks)

- In general, for n items, knapsack has 2^n choices. So brute force approach runs in $O(2^n)$ time.
- We can improve performance by Greedy Method where it sorts items in advance. Using merge sort or heap sort, n items can be sorted in $O(n\log_2 n)$ time. Merge sort and heap sort are non-adaptive and their running time is the same in best, average and worst case.
- To select the items, we need one scan to this sorted list, which will take O(n) time.
- So the total time required is
  $T(n) = O(n\log_2 n) + O(n) = O(n\log_2 n)$.

Time Complexity: O(N * log N)

Auxiliary Space: O(N)

******END******

Minimum Spanning trees

What is a Spanning Tree? (2 Marks)

- A Spanning tree is a subset to a connected graph G, where all the edges are connected, i.e, one can traverse to any edge from a particular edge with or without intermediates.
- Also, a spanning tree must not have any cycle in it. Thus we can say that if there are **N** vertices in a connected graph then the no. of edges that a spanning tree may have is **N-1**.

What is a Minimum Spanning Tree? (2 Marks)

- Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has? (2 Marks)

- A minimum spanning tree has (V – 1) edges where **V** is the number of vertices in the given graph.

Applications of Minimum-Cost Spanning Trees(2 Marks)

Minimum-cost spanning trees have many applications. Some are:

- Building cable networks that join n locations with minimum cost.
- Building a road network that joins n cities with minimum cost.
- Obtaining an independent set of circuit equations for an electrical network.
- In pattern recognition minimal spanning trees can be used to find noisy pixels.

This problem can be solved by many different algorithms. Two of them include :

- ***"Prims"*** Algorithm
- ***"Kruskal's"*** Algorithm.

Kruskal's Algorithm: (5 Marks)

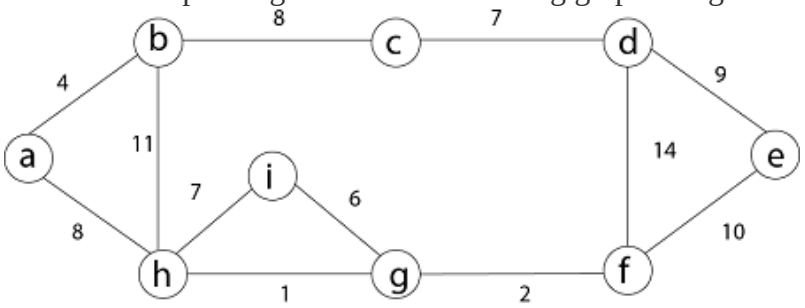
- An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the Minimum Spanning Tree(MST) constructed so far.
- Below are the Algorithm steps for finding MST using Kruskal's algorithm:
  1. Sort all the edges in non-decreasing order of their weight.
  2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
  3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Algorithm:

MST- KRUSKAL (G, w)

1. A ← ∅
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
7. then A ← A ∪ {(u, v)}
8. UNION (u, v)
9. return A

Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.

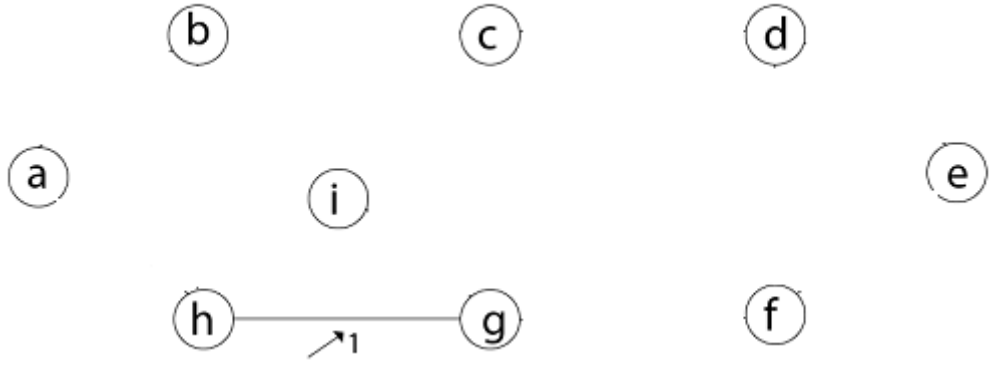


Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

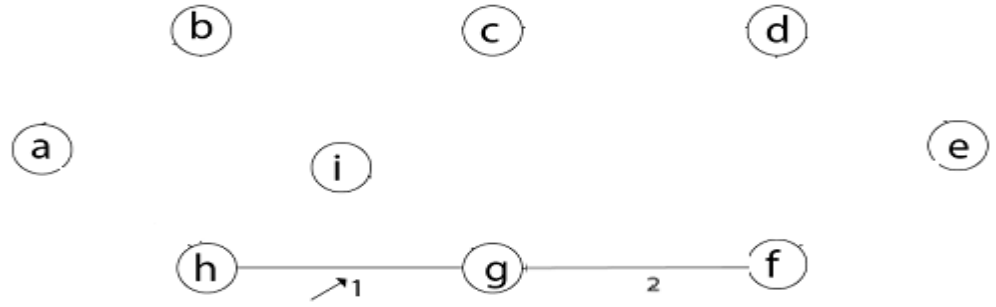
There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

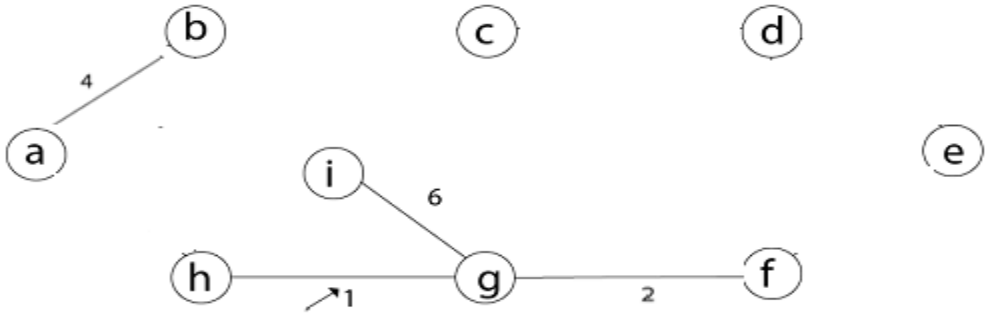
Step1: So, first take (h, g) edge



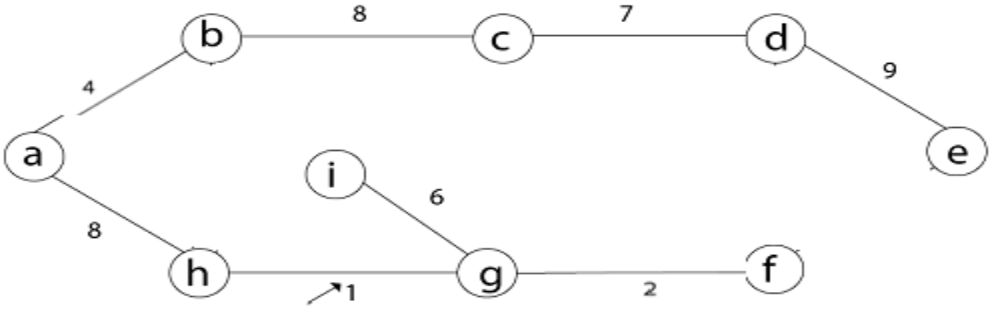
Step 2: then (g, f) edge is inserted where it doesn't form cycle



Step 3: then (a, b) and (i, g) edges are considered, and inserted into MST

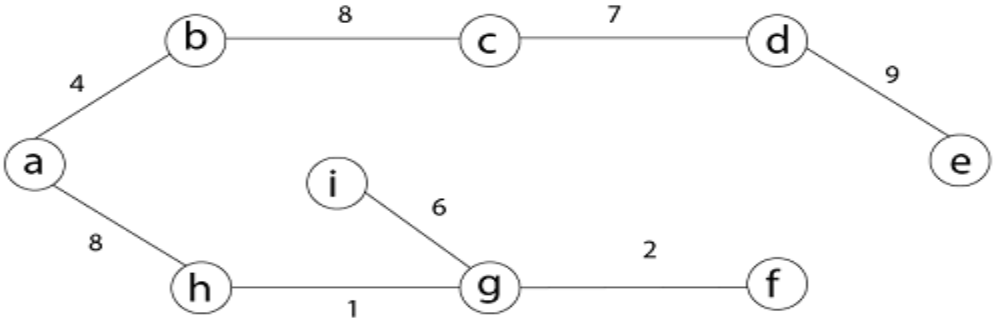


Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.
Then edge (c, d), (b, c), (a, h), (d, e) are considered where it doesn't form cycle



Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) and (d, f) edge, it also creates a cycle.

It is Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges



Time Complexity: (2Marks)

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and log V^2= 2 x log V is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is

O (E log E) = O (E log V).

Space Complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph

PRIMS Algorithm(5 Marks)

- **Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
- Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

The steps for implementing Prim's algorithm are as follows:

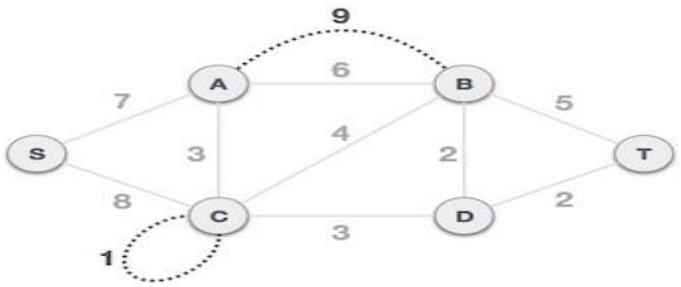
1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

Algorithm

1. Step 1: Select a starting vertex
2. Step 2: Repeat Steps 3 and 4 until there are fringe vertices
3. Step 3: Select an edge 'e' connecting the tree vertex and add vertex that has minimum wei ght
4. Step 4: Add the selected edge and the vertex to the minimum spanning tree T
5. [END OF LOOP]
6. Step 5: EXIT

Note:

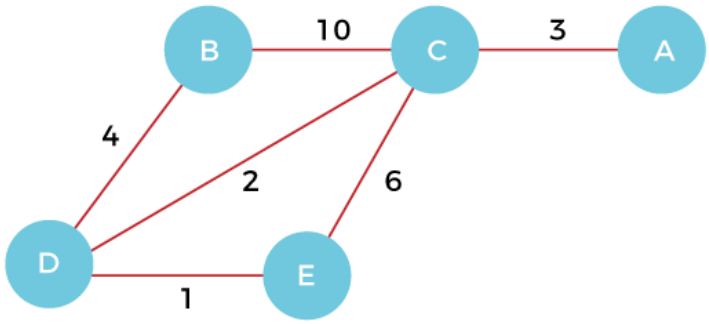
Remove all loops and parallel edges



Example of prim's algorithm

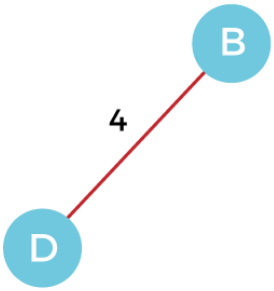
- Now, let's see the working of prim's algorithm using an example. It will be easier to understand the prim's algorithm using an example.

Suppose, a weighted graph is -

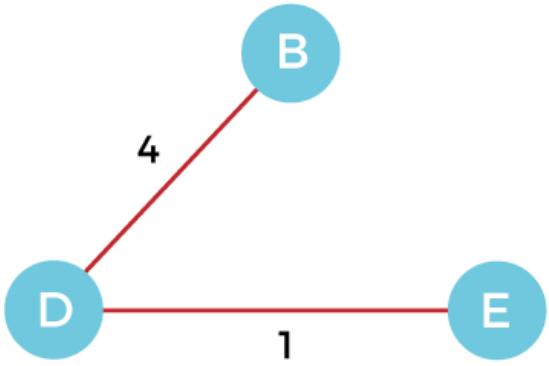


Step 1 - First, we have to choose a vertex from the above graph. Let's choose B.

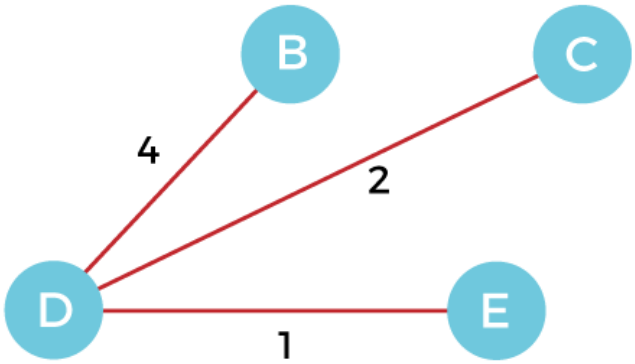
Step 2 - Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



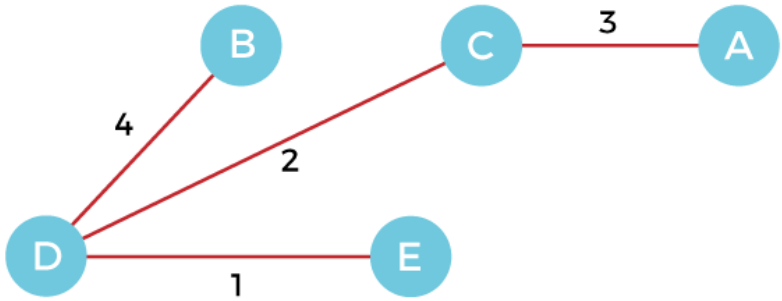
Step 3 - Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4 - Now, select the edge CD, and add it to the MST.



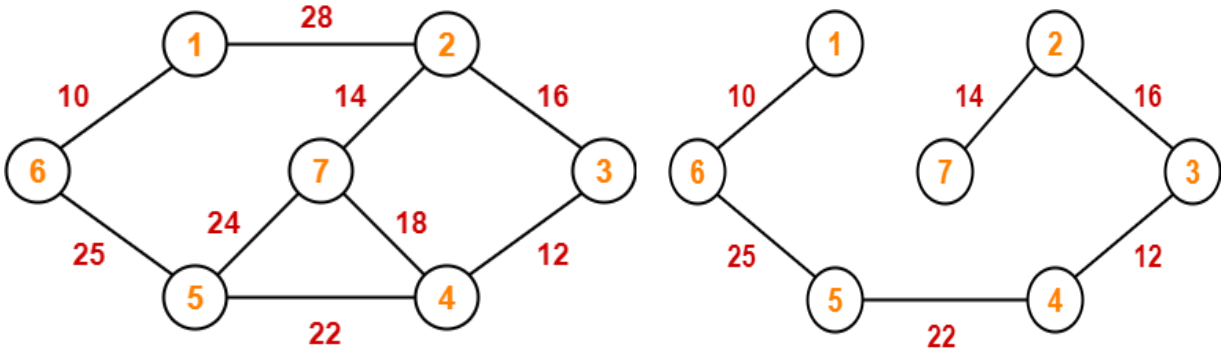
Step 5 - Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Example 2:



Complexity of Prim's algorithm(5 Marks)

Now, let's see the time complexity of Prim's algorithm. The running time of the prim's algorithm depends upon using the data structure for the graph and the ordering of edges. Below table shows some choices -

Time Complexity

Data structure used for the minimum edge weight	Time Complexity
Adjacency matrix, linear searching	$O(\|V\|^2)$
Adjacency list and binary heap	$O(\|E\| \log \|V\|)$
Adjacency list and Fibonacci heap	$O(\|E\|+ \|V\| \log \|V\|)$

Time Complexity Analysis

- If adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in O(V + E) time.

- We traverse all the vertices of graph using breadth first search and use a min heap for storing the vertices not yet included in the MST.
- To get the minimum weight edge, we use min heap as a priority queue.
- Min heap operations like extracting minimum element and decreasing key value takes O(logV) time.

  So, overall time complexity
  = O(E + V) * O(logV)
  = O((E + V)logV)
  = O(ElogV)

- This time complexity can be improved and reduced to O(E + VlogV) using Fibonacci heap.

The applications of prim's algorithm are -(2 Marks)

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

END

Single Source Shortest Path (SSSP) Problem or Dijkstra's Algorithm(5 Marks)

- SSSP is a problem of finding shortest paths from a source vertex s to all other vertices in the graph.
- Dijkstra's Algorithm is a solution to the Single Source Shortest Path (SSSP) problem and it was created by computer scientist Edsger W. Dijkstra in 1956.
- It Works on both directed and undirected graphs. However, all edges must have nonnegative weights.
  - Input: Weighted graph G={E,V} and source vertex s∈V, such that all edge weights are nonnegative
  - Output: Lengths of shortest paths

In this algorithm each vertex will have two properties defined for it-

Visited property:-

- This property represents whether the vertex has been visited or not.
- We are using this property so that we don't revisit a vertex.
- A vertex is marked visited only after the shortest path to it has been found.

Path property:-

- This property stores the value of the current minimum path to the vertex. Current minimum path means the shortest way in which we have reached this vertex till now.
- This property is updated whenever any neighbour of the vertex is visited.
- The path property is important as it will store the final answer for each vertex.

Algorithm

1. Start
2. Mark the source node with a current distance of 0 and the rest with infinity.
3. Set the non-visited node with the smallest current distance as the current node, lets say C.
4. For each neighbour N of the current node C: add the current distance of C with the weight of the edge connecting C-N. If it is smaller than the current distance of N, set it as the new current distance of N.

5. Mark the current node C as visited.
6. Go to step 2 if there are any nodes are unvisited.
7. Stop

Algorithm for Dijkstra's Algorithm

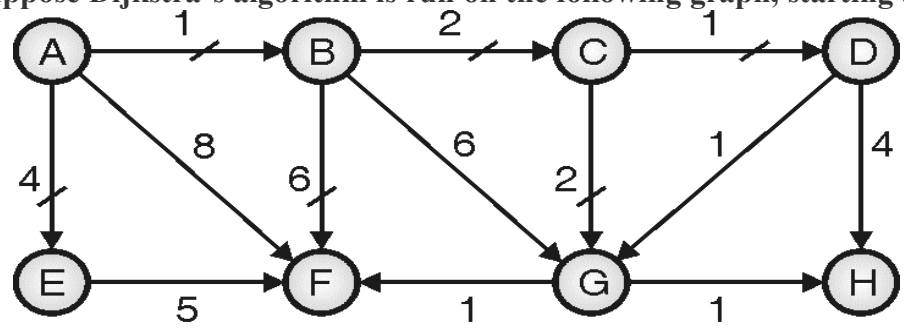
Dijkstra's shortest path algorithm is described below :

```
Algorithm DIJAKSTRA_SHORTEST_PATH(G, s, t)
// s is the source vertex
// t is the target vertex
// π[u] stores the parent / previous node of u
// V is the set of vertices in graph G

dist[s] ← 0
π[s] ← NIL

for each vertex v ∈ V do
    if v ≠ s then
        dist[v] ← ∞
        π[v] ← undefined
    end
    ENQUEUE(v, Q)          // insert v to queue Q
end

while Q is not empty do
    u ← vertex in Q having minimum dist[u]
    if u == t then
        break
    end
    DEQUEUE(u, Q)    // Remove u from queue Q

    for each adjacent node v of u do
        val ← dist[u] + weight(u, v)
        if val<dist[v] then
            dist[v] ← val
            π[v] ← u
```

```
            end
        end
end
```

Examples

Problem: Suppose Dijkstra's algorithm is run on the following graph, starting at node A



1. Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.
2. Show the final shortest path tree.

Solution:

Here, source vertex is A.

dist[u] indicates distance of vertex u from source

π[u] indicates parent / previous node of u

Initialization:

dist[source] = 0 ⇒ dist[A] = 0

π[source] = undefined ⇒ π[A] = NIL

dist[B] = dist[C] = dist[D] = dist[E] = dist [F] = dist[G]= dist[H] = ∞

π[B] = π[C] = π[D] = π[E] = π[F] = π[G] = π[H]= NIL

Vertex u	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
dist[u]	0	∞	∞	∞	∞	∞	∞	∞
π[u]	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Iteration 1:

u = unprocessed vertex in Q having minimum dist[u] = A

Adjacent[A] = {B, E, F}

val[B] = dist[A] + weight(A, B)
= 0 + 1
= 1
Here, val[B] <dist[B] i.e 1< ∞, so update dist[B] ∞ to 1
dist[B] = 1, and π[B] = A
val[E] = dist[A] + weight(A, E)
= 0 + 4
= 4
Here, val[E] < dist[E], so update dist[E] ∞ to 4
dist[E] = 4 and π[6] = A
val[F] = dist[A] + weight(A, F)
= 0 + 8
= 8
Here, val[F] < dist[F], so update dist[F] ∞ to 8
dist[F] = 8 and π[F] = A

Vertex u	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**

dist[u]	0	1	∞	∞	4	8	∞	∞
π[u]	NIL	A	NIL	NIL	A	A	NIL	NIL

Iteration 2:

u = unprocessed vertex in Q having minimum dist[u] = B

Adjacent[B] = {C, F, G}

val[C] = dist[B] + weight(B, C)
= 1 + 2
= 3
Here, val[C] < dist[C], so update dist[C]
dist[C] = 3 and π[C] = B
val[F] = dist[B] + weight(B, F)
= 1 + 6
= 7
Here, val[F] < dist[F], so update dist[F]
dist[F] = 7 and π[F] = B
val[G] = dist[B] + weight(B, G)
= 1 + 6
= 7
Here, val[G] < dist[G], so update dist[G]
dist[G] = 7 and π[G] = B

Vertex u	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
dist[u]	0	1	3	∞	4	7	7	∞
π[u]	NIL	A	B	NIL	A	B	B	NIL

Iteration 3:

u = unprocessed vertex in Q having minimum dist[u] = C

Adjacent [C] = {D, G}

val[D] = dist[C] + weight(C, D)
= 3 + 1
= 4
Here, val[D] < dist[D], so update dist[D]
dist[D] = 4 and π[D] = C
val[G] = dist[C] + weight(C, G)
= 3 + 2
= 5
Here, val[G] < dist[G], so update dist[G]
dist[G] = 5 and π[G] = C

Vertex u	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
dist[u]	0	1	3	4	4	7	5	∞
π[u]	NIL	A	B	C	A	B	C	NIL

Iteration 4:

u = unprocessed vertex in Q having minimum dist[u] = E

Adjacent[E] = {F}

val[F] = dist[E] + weight(E, F)
= 4 + 5
= 9
Here, val[F] > dist[F], so no change in table

Vertex u	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
dist[u]	0	1	3	4	4	7	5	∞
π[u]	NIL	A	B	C	A	B	C	NIL

Iteration 5:

u = unprocessed vertex in Q having minimum dist[u] = D

Adjacent[D] = {G, H}

val[G] = dist[D] + weight(D, G)
= 4 + 1
= 5
Here, val[G] = dist[G], so don't update dist[G]
val[H] = dist[D] + weight(D, H)
= 4 + 4
= 8
Here, val[H] < dist[H], so update dist[H]
dist[H] = 8 and π[H] = D

Vertex u	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
dist[u]	0	1	3	4	4	7	5	8
π [u]	NIL	A	B	C	A	B	D	D

Iteration 6:

u = unprocessed vertex in Q having minimum dist[u] = G

Adjacent[G] = { F, H }

val[F] = dist[G] + weight(G, F)
= 5 + 1
= 6
Here, val[F] < dist[F], so update dist[F]
dist[F] = 6 and π[F] = G
val[H] = dist[G] + weight(G, H)
= 5 + 1
= 6
Here, val[H] < dist[H], so update dist[H]
dist[H] = 6 and π[H] = G

Vertex u	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
dist[u]	0	1	3	4	4	6	5	6
π [u]	NIL	A	B	C	A	G	C	G

Iteration 7:

u = unprocessed vertex in Q having minimum dist[u] = F

Adjacent[F] = { }

So, no change in table

Iteration 8:

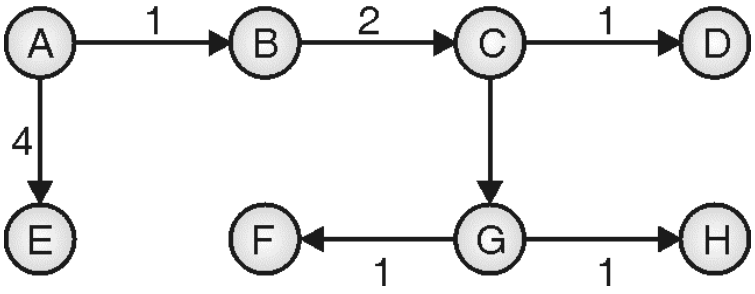
u = unprocessed vertex in Q having minimum dist[u] = H

Adjacent[H] = { }

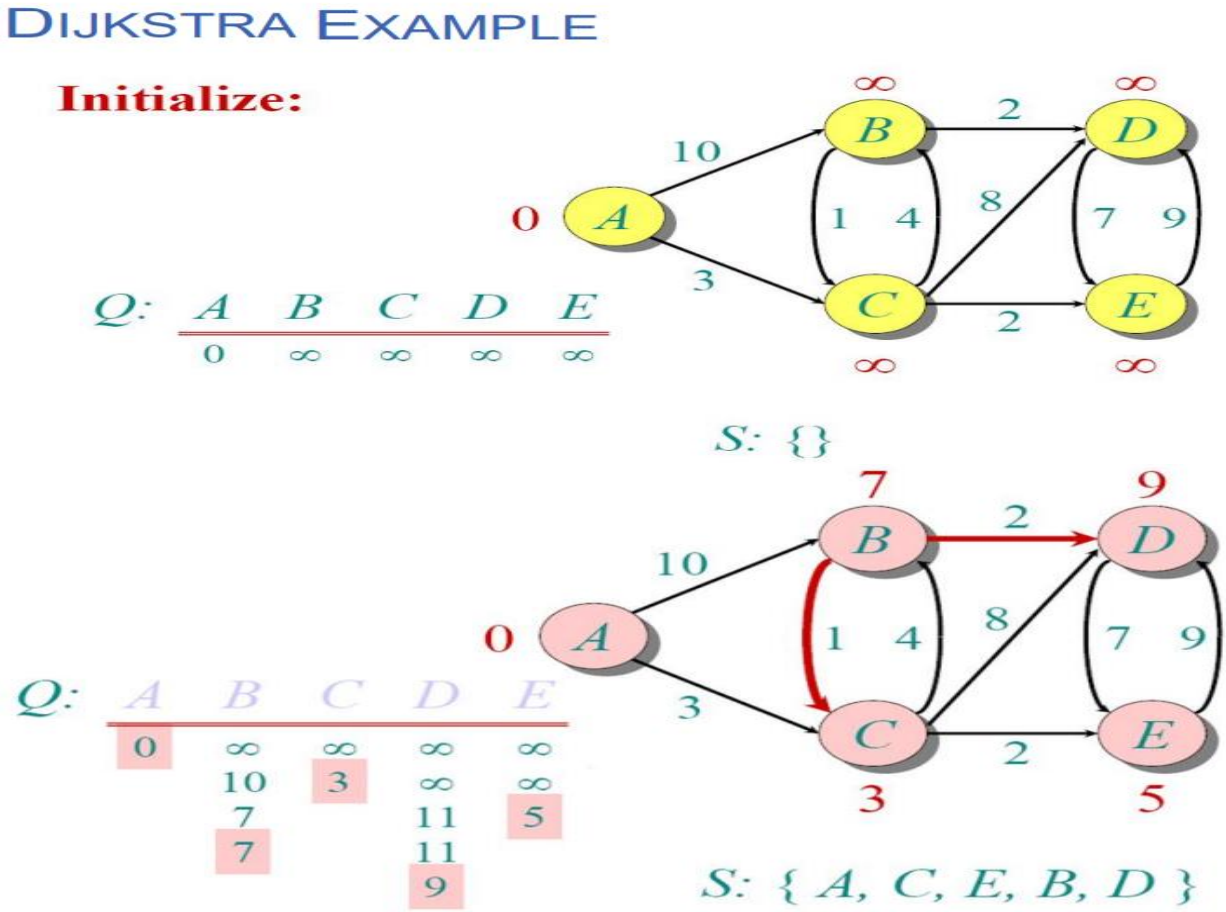
So, no change in table

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	3	4	4	6	5	6
p[u]	NIL	A	B	C	A	G	C	G

We can easily derive the shortest path tree for given graph from above table. In the table, p[u] indicates the parent node of vertex u. The shortest path tree is shown in following figure



Example 2:



APPLICATIONS OF DIJKSTRA'S ALGORITHM(2 Marks)

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- **Designate Servers:** In a network, Dijkstra's Algorithm can be used to find the shortest path to a server for transmitting files amongest the nodes on a network.
- **IP Routing:** Dijkstra's Algorithm can be used by link state routing protocols to find the best path to route data between routers.

Dijkstra Algorithm Time Complexity Analysis: (5 Marks)

Case 1: Naive Implementation:

- The given graph $G = (V, E)$ is represented as an adjacency matrix. Here $w[u, v]$ stores the weight of edge (u, v).
- The priority queue Q is represented as an unordered list.

Let $|E|$ and $|V|$ be the number of edges and vertices in the graph, respectively. Then the time complexity is calculated:

1. Adding all $|V|$ vertices to Q takes $O(|V|)$ time.
2. Removing the node with minimal $dist$ takes $O(|V|)$ time, and we only need $O(1)$ to recalculate $dist[u]$ and update Q. Since we use an adjacency matrix here, we'll need to loop for $|V|$ vertices to update the $dist$ array.
3. The time taken for each iteration of the loop is $O(|V|)$, as one vertex is deleted from Q per loop.
4. Thus, total time complexity becomes $O(|V|) + O(|V|) \times O(|V|) = O(|V|^2)$.

Worst case : $O(V^2)$
Best Case : $O(V^2)$
Average Case: $O(V^2)$

Case 2: Binary Heap + Priority Queue

Worst case : O(ElogV)
Best Case : O(ElogV)
Average Case: O(ElogV)

Case 3: Fibonacci Heap + Priority Queue:

Best Case : O(E+VlogV)
Average Case : O(E+VlogV)
Worst Case : O(E+VlogV)

*****END*****