Unit 2: Brute Force Approaches

Searching Techniques:

Searching techniques are fundamental algorithms in computer science used to find an item from a list or a collection. The efficiency of a searching algorithm is critical in many applications, such as databases, file systems, and networking. Two of the most basic and widely used searching techniques are Sequential Search and Binary Search.

Sequential Search (or Linear Search)

Sequential search is the simplest searching algorithm. It involves scanning each element in the list one by one until the desired element is found or the end of the list is reached.

How Sequential Search Works:

1. Start from the first element of the list.
2. Compare the current element with the target element.
3. If the current element matches the target, return the index of that element.
4. If the current element does not match, move to the next element.
5. If the end of the list is reached and no match is found, return an indication that the search was unsuccessful (e.g., -1).

Complexity:

Best Case: (O(1)) - The element is found at the first position.
Worst Case: (O(n)) - The element is not in the list or is at the last position.
Average Case: (O(n)) - On average, the algorithm checks half of the elements in the list.

Use Cases:

- Suitable for small or unsorted data.
- Practical when data is received in a stream and needs to be checked in real time.

Binary Search

Binary search is a much faster algorithm than sequential search but requires that the data be sorted beforehand. It repeatedly divides the search interval in half and compares the midpoint against the target until it finds the target or concludes that the target isn't in the list.

How Binary Search Works:

1. Start with the entire array.

2. Find the middle element of the array.
3. If the middle element equals the target, return the index of that element.
4. If the target is less than the middle element, repeat the search on the left subarray (elements before the middle).
5. If the target is greater than the middle element, repeat the search on the right subarray (elements after the middle).
6. If the subarray reduces to zero size and no match is found, return an indication of failure (e.g., `-1`).

Complexity:

Best Case: (O(1)) - The central element of the list is the target.
Worst Case: (O(log n)) - The element is not in the list or is near the end or beginning.
Average Case: (O(\log n)) - Each step cuts the list size by half.

Use Cases:

- Efficient for large datasets that are sorted.
- Commonly used in file searching algorithms and databases.

Comparing Sequential and Binary Search

Linear Search	**Binary Search**
In linear search input data need not to be sorted.	In binary search input data needs to be in sorted order.
It is also called linear search.	It is also called half-interval search.
The time complexity of linear search **O(n)**.	The time complexity of binary search **O(log n)**.
Multidimensional arrays can be used.	Only a single dimensional array is used.
Linear search performs equality comparisons	Binary search performs ordering comparisons
It is less complex.	It is more complex.
It is a very slow process.	It is a very fast process.

Conclusion

Choosing between sequential and binary search depends on the nature of the data (sorted vs. unsorted) and the size of the dataset. For small or unsorted data, sequential search is simple and effective. For large and sorted datasets, binary search offers a significant performance advantage.

Sorting Algorithms

A **Sorting Algorithm** is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Sorting Terminology:

- **In-place Sorting:** An in-place sorting algorithm uses **constant space** for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list. Examples: Selection Sort, Bubble Sort Insertion Sort and Heap Sort
- **Stable sorting:** When two same data appear in the **same order** in sorted data without changing their position is called stable sort. Examples: Merge Sort, Insertion Sort, Bubble Sort.
- **Unstable sorting:** When two same data appear in the **different order** in sorted data it is called unstable sort. Examples: Quick Sort, Heap Sort, Shell Sort.

Bubble Sort Algorithm

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

In Bubble Sort algorithm,

- traverse from left and compare adjacent elements and the higher one is placed at the right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

C Code for Bubble Sort:

```
void bubbleSort(int a[], int n){
        for(int i=0; i<n-1; i++){
                for(int j=0; j<n-i-1; j++){
                        if(a[j]>a[j+1])
                                swap(&a[j], &a[j+1]);
                }
        }
}
```

How does Bubble Sort Work?

Input Array: [5, 1, 4, 2, 8]

First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

And So on for 4th (n-1)th Pass

Total no. of passes: n-1
Total no. of comparisons: n*(n-1)/2

Selection Sort

> **Selection sort** is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

How does Selection Sort Algorithm work?

Lets consider the following array as an example: **arr[] = {64, 25, 12, 22, 11}**

Algorithm of the Selection Sort Algorithm

The selection sort algorithm is as follows:

Step 1: Set Min to location 0 in Step 1.

Step 2: Look for the smallest element on the list.

Step 3: Replace the value at location Min with a different value.

Step 4: Increase Min to point to the next element

Step 5: Continue until the list is sorted.

Code For Selection Sort:

```
void selectionSort(int a[], int n){
  if(n<2)
     return;
  for(int i=0; i<n-1; i++){
     int min_i=i;
     for(int j=i+1; j<n;j++){
        if(a[min_i]> a[j])
           min_i=j;
     }
     swap(&a[min_i], &a[i]);
  }
}
```

Insertion Sort

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a **stable sorting** algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

Insertion sort is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

Insertion Sort Algorithm:

Insertion sort is a simple sorting algorithm that works by building a sorted array one element at a time. It is considered an "**in-place**" sorting algorithm, meaning it doesn't require any additional memory space beyond the original array.

Algorithm:

To achieve insertion sort, follow these steps:

- We have to start with the second element of the array as the first element in the array is assumed to be sorted.
- Compare the second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
- Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
- Repeat until the entire array is sorted.

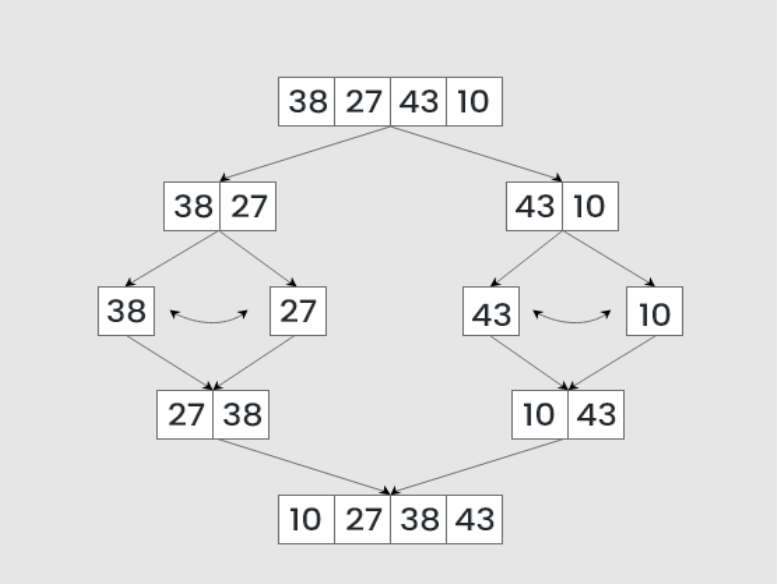
Code for Insertion Sort:

```
void insertionSort(int a[], int n){
  for(int i=1; i<n; i++){
      int j=i-1;
      int temp= a[i];
      while (j>=0 && a[j]>temp)
      {
         a[j+1]= a[j];
         j--;
      }
      a[j+1]= temp;
  }
}
```

Merge Sort

Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



How does Merge Sort work?

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach to sort a given array of elements.

Here's a step-by-step explanation of how merge sort works:

1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Illustration of Merge Sort:

Let's sort the array or list **[38, 27, 43, 10]** using Merge Sort

Divide:

- **[38, 27, 43, 10]** is divided into **[38, 27**] and **[43, 10]**.
- **[38, 27]** is divided into **[38]** and **[27]**.
- **[43, 10]** is divided into **[43]** and **[10]**.

Conquer:

- **[38]** is already sorted.
- **[27]** is already sorted.
- **[43]** is already sorted.

- **[10]** is already sorted.

Merge:

- Merge **[38]** and **[27]** to get **[27, 38]**.
- Merge **[43]** and **[10]** to get **[10,43]**.
- Merge **[27, 38]** and **[10,43]** to get the final sorted list **[10, 27, 38, 43]**

Therefore, the sorted list is **[10, 27, 38, 43]**.

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

1. MERGE_SORT(arr, beg, end)
2. if beg < end
    a. set mid = (beg + end)/2
    b. MERGE_SORT(arr, beg, mid)
    c. MERGE_SORT(arr, mid + 1, end)
    d. MERGE (arr, beg, mid, end)
3. end of if
4. END MERGE_SORT

The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg…mid]** and **A[mid+1…end]**, to build one sorted array **A[beg…end]**. So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.

The MERGE function algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.
Have we reached the end of any of the arrays?
No:
Compare current elements of both arrays
Copy smaller element into sorted array
Move pointer of element containing smaller element
Yes:
Copy all remaining elements of non-empty array

C Code:

```
void MERGE(int a[], int l, int mid, int h){

  int i=l;
  int j=mid+1;
  int temp[h-l+1], k=0;
  while(i<=mid && j<=h){
     if(a[i]<=a[j])
        temp[k++]=a[i++];
     else
        temp[k++]= a[j++];
```

```
  }
  while (i<=mid)
     temp[k++]= a[i++];

  while (j<=h)
     temp[k++]= a[j++];

  for(i=0; i<k; i++)
     a[l+i]= temp[i];

}

void MERGE_SORT(int a[],int l, int h){
  if(l<h){
     int mid= (l+h)/2;

     MERGE_SORT(a, l, mid);
     MERGE_SORT(a, mid+1, h);

     MERGE(a, l, mid, h);
  }
}
```

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.
- **Simple to implement:** The divide-and-conquer approach is straightforward

Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Inversion counting (counting the number of inversions in an array)
- Finding the median of an array

QuickSort

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort work?

The key process in **quickSort** is a **partition()**. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

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.

Partition Algorithm:

The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as **i**. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

QUICKSORT (array A, start, end)

{

1. if (start < end)
2. {
3. p = partition(A, start, end)
4. QUICKSORT (A, start, p - 1)
5. QUICKSORT (A, p + 1, end)
6. }

}

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

PARTITION (array A, start, end)

{

1. pivot ? A[end]
2. i ? start-1
3. for j ? start to end -1 {
4. do if (A[j] < pivot) {
5. then i ? i + 1
6. swap A[i] with A[j]

7. }
}
8. swap A[i+1] with A[end]
9. return i+1

}

C-Code:

```
int partition(int arr[], int l, int h){

  int pivot= arr[h];
  int i=l-1;
  for(int j=l; j<h; j++){
     if(arr[j]< pivot){
        i++;
        swap(&arr[i], &arr[j]);
     }
  }
  swap(&arr[i+1], &arr[h]);

  return i+1;
}

void quickSort(int arr[], int l, int h){
  if(l<h){
     int pivot= partition(arr, l, h);

     quickSort(arr, l, pivot-1);
     quickSort(arr, pivot+1, h);
  }
}
```

Advantages of Quick Sort:

- 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:

- It has a worst-case time complexity of $O(N^2)$, 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).

Counting Sort

What is Counting Sort?

Counting Sort is a **non-comparison-based** sorting algorithm that works well when there is a limited range of input values. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind **Counting Sort** is to count the **frequency** of each distinct element in the input array and use that information to place the elements in their correctly sorted position.

Counting Sort Algorithm:

- Declare an auxiliary array **countArray[]** of size **max(inputArray[])+1** and initialize it with **0**.
- Traverse array **inputArray[]** and map each element of **inputArray[]** as an index of **countArray[]** array, i.e., execute **countArray[inputArray[i]]++** for **0 <= i < N**.
- Calculate the prefix sum at every index of array **inputArray**[].
- Create an array **outputArray[]** of size **N**.
- Traverse array **inputArray[]** from end and update **outputArray[countArray[inputArray[i]] – 1] = inputArray[i]**. Also, update **countArray[inputArray[i]] = countArray[inputArray[i]]- –** .

C- Code:

```
// return maximum of two element
int maximumTwo(int a, int b){
  return a > b ? a : b;
}
//Count sort implementation
void CountSort(int arr[], int n){
  int maxSize= -1;
  for (int i = 0; i < n; i++)
     maxSize = maximumTwo(maxSize, arr[i]);
```

```
  maxSize++;
  int countArray[maxSize];

  for (int  i = 0; i < maxSize; i++){
     countArray[i]=0;}

  for (int  i = 0; i < n; i++){
     ++countArray[arr[i]];}

  for (int  i = 1; i < maxSize; i++){
     countArray[i] += countArray[i-1];}

  displayArray(countArray, maxSize);
  int result[n];
  for(int i=0; i<n; i++){
     int pos= --countArray[arr[i]];
     result[pos]= arr[i];
  }
  for (int i = 0; i < n; i++)
     arr[i]= result[i];
}
```

Advantage of Counting Sort:

- Counting sort generally performs faster than all comparison-based sorting algorithms, such as merge sort and quicksort, if the range of input is of the order of the number of input.
- Counting sort is easy to code
- Counting sort is a **stable algorithm**.

Disadvantage of Counting Sort:

- Counting sort doesn't work on decimal values.
- Counting sort is inefficient if the range of values to be sorted is very large.
- Counting sort is not an **In-place sorting** algorithm, It uses extra space for sorting the array elements.

Radix Sort

> **Radix Sort** is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

Rather than comparing elements directly, Radix Sort distributes the elements into buckets based on each digit's value. By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, Radix Sort achieves the final sorted order.

Radix Sort Algorithm

The key idea behind Radix Sort is to exploit the concept of place value. It assumes that sorting numbers digit by digit will eventually result in a fully sorted list. Radix Sort can be performed using different variations, such as Least Significant Digit (LSD) Radix Sort or Most Significant Digit (MSD) Radix Sort.

1. radixSort(arr)
2. max = largest element in the given array
3. d = number of digits in the largest element (or, max)
4. Now, create d buckets of size 0 - 9
5. for i -> 0 to d
6. sort the array elements using counting sort (or any stable sort) according to the digits at the ith place

Working of Radix Sort Algorithm

- The Radix sort algorithm works by ordering each digit from least significant to most significant.
- In base 10, radix sort would sort by the digits in the one's place, then the ten's place, and so on.
- To sort the values in each digit place, Radix sort employs counting sort as a subroutine.
- This means that for a three-digit number in base 10, counting sort will be used to sort the 1st, 10th, and 100th places, resulting in a completely sorted list. Here's a rundown of the counting sort algorithm.

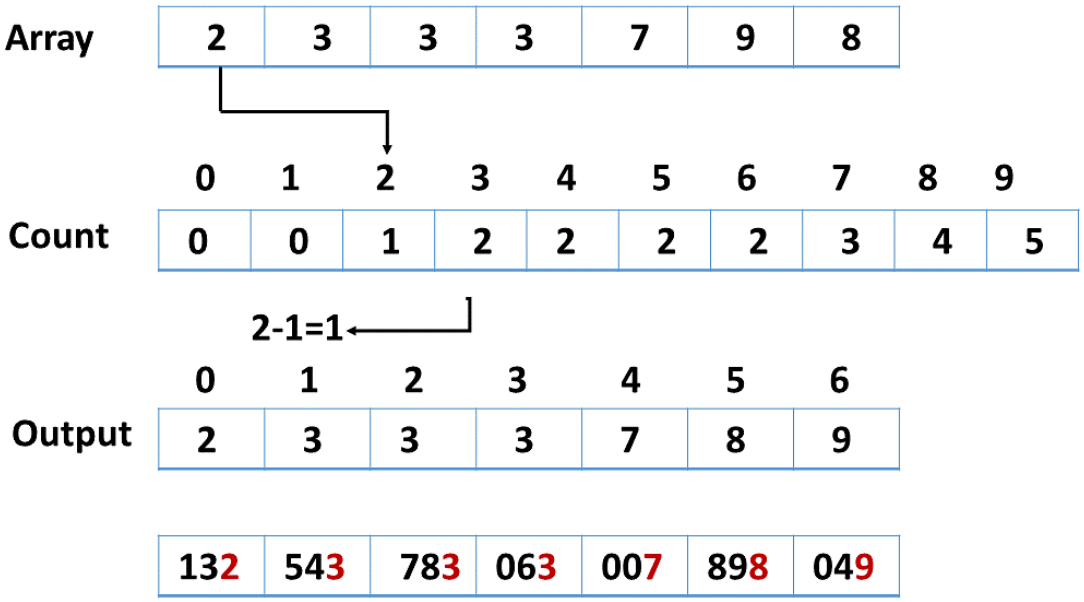
Assume you have an 8-element array. First, you will sort the elements by the value of the unit place. It will then sort the elements based on the value of the tenth position. This process is repeated until it reaches the last significant location.

Let's start with [132, 543, 783, 63, 7, 49, 898]. It is sorted using radix sort, as illustrated in the figure below.

- Find the array's largest element, i.e., maximum. Consider A to be the number of digits in maximum. A is calculated because we must traverse all of the significant locations of all elements.

The largest number in this array [132, 543, 783, 63, 7, 49, 898] is 898. It has three digits. As a result, the loop should be extended to hundreds of places (3 times).

- Now, go through each significant location one by one. Sort the digits at each significant place with any stable sorting technique. You must use counting sort for this. Sort the elements using the unit place digits (A = 0).



- Sort the elements now by digits in the tens place.

- Finally, sort the elements by digits in the hundreds place.

C Code implementation:

```
void rcountSort(int arr[], int n, int place){
  int countArray[10]={0};

  for(int i=0; i<n; i++)
     countArray[(arr[i]/place)%10]++;
```

```
  for(int i=1; i<10; i++)
     countArray[i] += countArray[i-1];

  int output[n];
  for(int i=n-1; i>=0; i--){
     int pos= --countArray[(arr[i]/place)%10];
     output[pos]= arr[i];
  }
  for(int i=0; i<n; i++)
     arr[i]= output[i];
  printf("%d place->", place);
  displayArray(arr,n);
}
void radixSort(int arr[],int n){
  int maxElement=arr[0];
  for(int i=0; i<n; i++)
     maxElement= maximumTwo(maxElement, arr[i]);
  for(int place=1; maxElement/place>0; place *= 10){
     rcountSort(arr, n, place);
  }
}
```

Heap Sort

> **Heap sort** is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Heap Sort Algorithm

To solve the problem follow the below idea:

> First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of the heap is greater than 1.

- Build a heap from the given input array.
- Repeat the following steps until the heap contains only one element:
    - Swap the root element of the heap (which is the largest element) with the last element of the heap.
    - Remove the last element of the heap (which is now in the correct position).
    - Heapify the remaining elements of the heap.
- The sorted array is obtained by reversing the order of the elements in the input array.

```
// To heapify a subtree rooted with node i which is an index in arr[].
void heapify(int arr[], int n, int i) {

    // Initialize largest as root
    int largest = i;

    // left child = 2*i + 1
    int l = 2 * i + 1;

    // right child = 2*i + 2
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest]) {
        largest = l;
    }

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest]) {
        largest = r;
    }

    // If largest is not root
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int n) {

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {

        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a classic optimization problem in computer science and operations research. The objective is to find the shortest possible route that visits a given set of cities exactly once and returns to the starting city. Here's a simple explanation and example of how the exhaustive search (also known as brute-force) approach works:

1. List all possible permutations: Generate all possible routes that visit each city once and return to the starting city. For `n` cities, there are `(n-1)!` possible routes to consider (since the starting city is fixed).

2. Calculate the total distance: For each route, calculate the total distance traveled by summing up the distances between consecutive cities.

3. Find the shortest route: Compare the total distances of all possible routes and choose the shortest one.

Let's consider a small example with four cities: A, B, C, and D. The distance between each pair of cities is given in the table below:

	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

Using the exhaustive search approach:

1. Generate all possible routes:
   - A -> B -> C -> D -> A
   - A -> B -> D -> C -> A
   - A -> C -> B -> D -> A
   - A -> C -> D -> B -> A
   - A -> D -> B -> C -> A
   - A -> D -> C -> B -> A

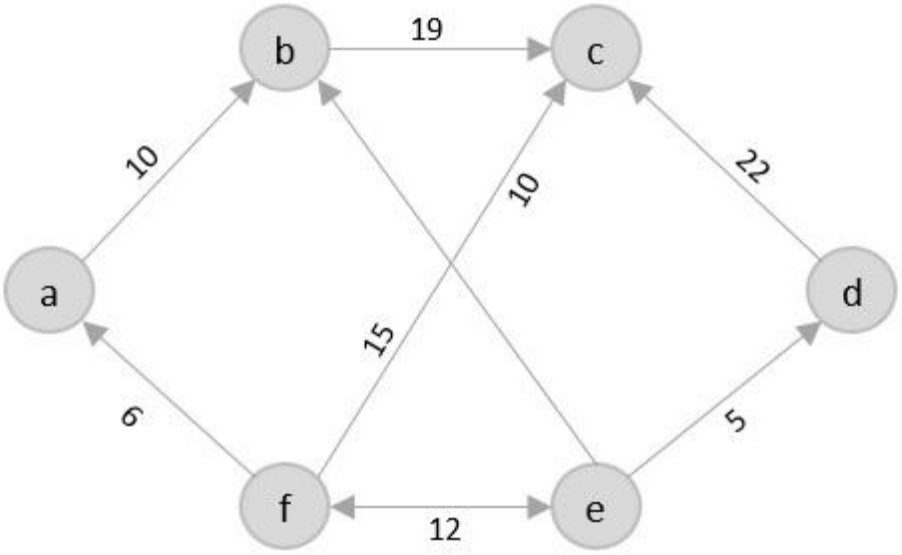
2. Calculate the total distance for each route:
   - A -> B -> C -> D -> A: 10 + 35 + 30 + 20 = 95
   - A -> B -> D -> C -> A: 10 + 25 + 30 + 15 = 80
   - A -> C -> B -> D -> A: 15 + 35 + 25 + 20 = 95
   - A -> C -> D -> B -> A: 15 + 30 + 25 + 10 = 80
   - A -> D -> B -> C -> A: 20 + 25 + 35 + 15 = 95
   - A -> D -> C -> B -> A: 20 + 30 + 35 + 10 = 95

3. Choose the shortest route:
   - The shortest routes are A -> B -> D -> C -> A and A -> C -> D -> B -> A, both with a total distance of 80.

Although the exhaustive search approach guarantees finding the optimal solution, it is computationally expensive for large numbers of cities due to the factorial growth of possible routes. For larger instances of TSP, heuristic and approximation algorithms like Genetic Algorithms, Simulated Annealing, and Ant Colony Optimization are often used.

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known. The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.

If you look at the graph below, considering that the salesman starts from the vertex a, they need to travel through all the remaining vertices b, c, d, e, f and get back to a while making sure that the cost taken is minimum.



There are various approaches to find the solution to the travelling salesman problem: naive approach, greedy approach, dynamic programming approach, etc. In this tutorial we will be learning about solving travelling salesman problem using greedy approach.

Travelling Salesperson Algorithm

As the definition for greedy approach states, we need to find the best optimal solution locally to figure out the global optimal solution. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges. The shortest path of graph G starting from one vertex returning to the same vertex is obtained as the output.

Algorithm

Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G) which will record the path the salesman is going to take from one node to another.
The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.

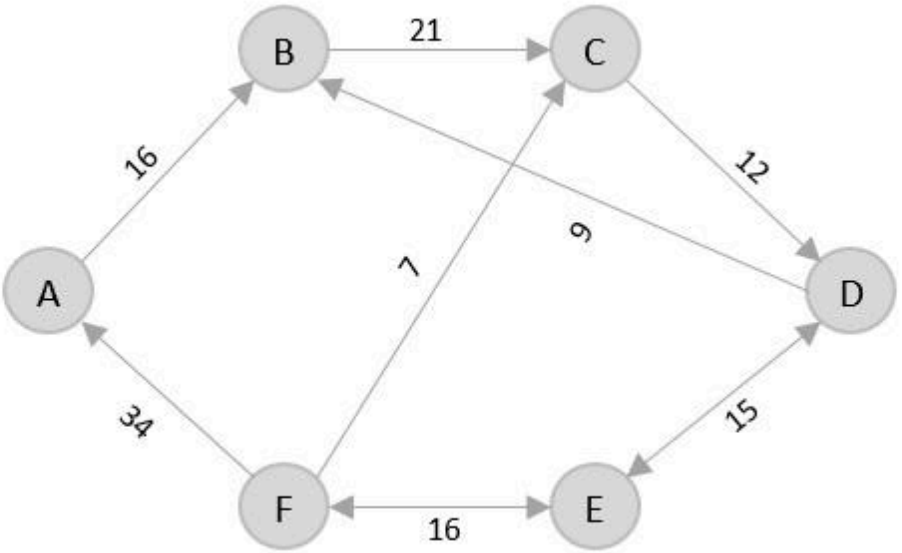
The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).

Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.

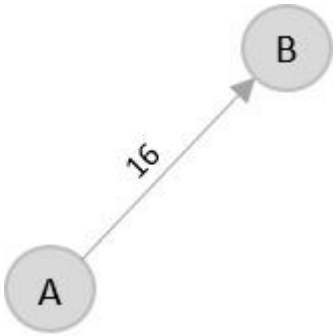
Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.

However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

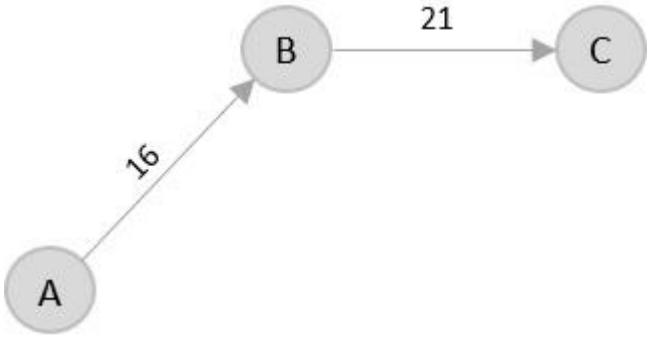
Consider the following graph with six cities and the distances between them –



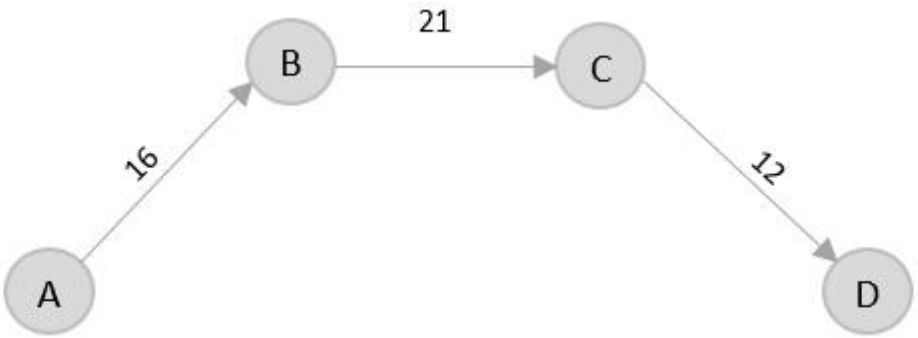
From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.



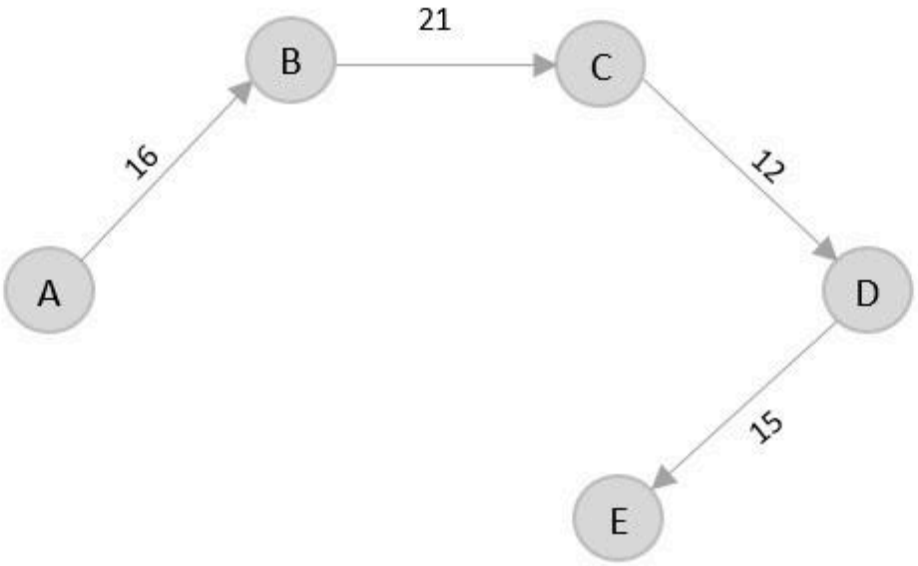
Then, B → C has the shortest and only edge between, therefore it is included in the output graph.



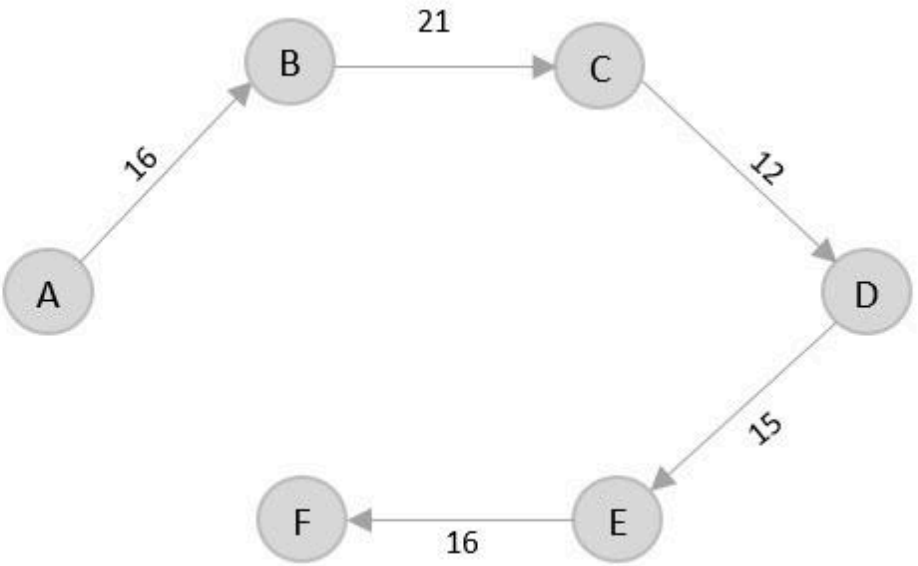
Theres only one edge between C → D, therefore it is added to the output graph.



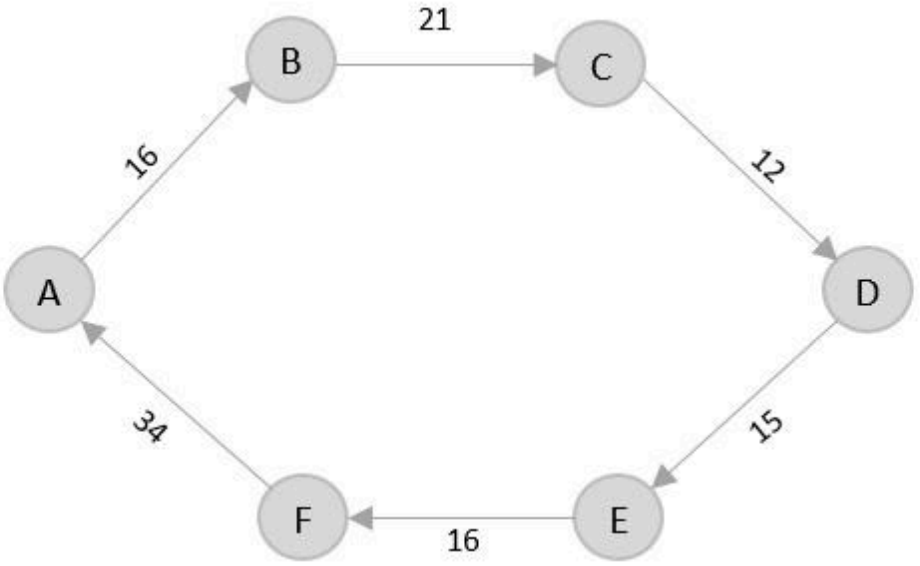
Theres two outward edges from D. Even though, D → B has lower distance than D → E, B is already visited once and it would form a cycle if added to the output graph. Therefore, D → E is added into the output graph.



Theres only one edge from e, that is E → F. Therefore, it is added into the output graph.



Again, even though F → C has lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: 16 + 21 + 12 + 15 + 16 + 34 = 114.

Even though, the cost of path could be decreased if it originates from other nodes but the question is not raised with respect to that.

What is Set Data Structure?

In computer science, a set data structure is defined as a data structure that stores a collection of distinct elements.
It is a fundamental Data Structure that is used to store and manipulate a group of objects, where each object is unique . The Signature property of the set is that it doesn't allow duplicate elements.
A set is a mathematical model for a collection of different things, a set contains elements or members, which can be mathematical objects of any kind numbers, symbols, points in space, lines, other geometrical shapes, variables, or even other sets.
A set can be implemented in various ways but the most common ways are:

1. Hash-Based Set: the set is represented as a hash table where each element in the set is stored in a bucket based on its hash code.
2. Tree-based set: In this implementation, the set is represented as a binary search tree where each node in the tree represents an element in the set.

Need for Set Data Structure:

Set data structures are commonly used in a variety of computer science applications, including algorithms, data analysis, and databases. The main advantage of using a set data structure is that it allows you to perform operations on a collection of elements in an efficient and organized way.

Types of Set Data Structure:

The set data structure can be classified into the following two categories:

1. Unordered Set

An unordered set is an unordered associative container implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the unordered set take constant time O(1) on an average which can go up to linear time O(n) in the worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.

2. Ordered Set

An Ordered set is the common set data structure we are familiar with. It is generally implemented using balanced BSTs and it supports O(log n) lookups, insertions and deletion operations.

Two sets are called disjoint sets if they don't have any element in common. The disjoint set data structure is used to store such sets. It supports following operations:

- Merging two disjoint sets to a single set using Union operation.
- Finding representative of a disjoint set using Find operation.
- Check if two elements belong to same set or not. We mainly find representative of both and check if same.

Consider a situation with a number of persons and the following tasks to be performed on them:

- Add a new friendship relation, i.e. a person x becomes the friend of another person y i.e adding new element to a set.
- Find whether individual x is a friend of individual y (direct or indirect friend)

We are given 10 individuals say, a, b, c, d, e, f, g, h, i, j

Following are relationships to be added:
a <-> b
b <-> d
c <-> f
c <-> i
j <-> e
g <-> j

Given queries like whether a is a friend of d or not. We basically need to create following 4 groups and maintain a quickly accessible connection among group items:
G1 = {a, b, d}
G2 = {c, f, i}
G3 = {e, g, j}
G4 = {h}

```
#include <iostream>
#include <vector>
using namespace std;

class UnionFind {
    vector<int> parent;
public:
    UnionFind(int size) {

        parent.resize(size);

        // Initialize the parent array with each
        // element as its own representative
        for (int i = 0; i < size; i++) {
            parent[i] = i;
        }
    }

    // Find the representative (root) of the
    // set that includes element i
    int find(int i) {

        // If i itself is root or representative
        if (parent[i] == i) {
            return i;
        }

        // Else recursively find the representative
        // of the parent
        return find(parent[i]);
    }

    // Unite (merge) the set that includes element
    // i and the set that includes element j
    void unite(int i, int j) {
```

```
        // Representative of set containing i
        int irep = find(i);

        // Representative of set containing j
        int jrep = find(j);

        // Make the representative of i's set
        // be the representative of j's set
        parent[irep] = jrep;
    }
};

int main() {
    int size = 5;
    UnionFind uf(size);
    uf.unite(1, 2);
    uf.unite(3, 4);
    bool inSameSet = (uf.find(1) == uf.find(2));
    cout << "Are 1 and 2 in the same set? "
         << (inSameSet ? "Yes" : "No") << endl;
    return 0;
}
```