ALGO-101

Week 2 - Sorting Algorithms and Array Problems

Ömer Faruk Erdem

ITU ACM

October 2022

Topics

Topics covered at week 2:

- Sorting Algorithms
  - Insertion Sort
  - Selection Sort
  - Bubble Sort
  - Merge Sort
  - Quick Sort
- Array Problems
  - Prefix Sum
  - Two Pointers

Insertion Sort

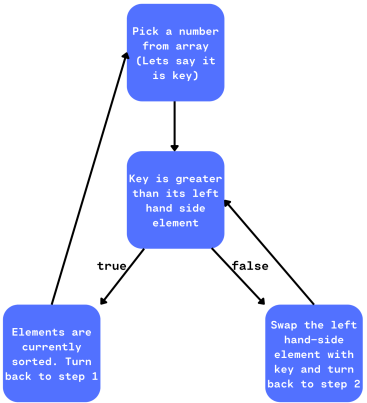
- **Time Complexity :** $O(n^2)$
- **Space Complexity :** $O(1)$

Insertion Sort is one of the simplest algorithm with simple implementation.
The algorithm iterates through the array and at every iteration it keeps a **key** value and then inserts that key value to the suitable position where is key value's left-hand side is smaller than or equal to itself and its right-hand side value is greater.
Algorithm works as :

- Pick a key element.
- If the key element's left side value is greater than the key value then swap the key element with left side element.
  - To make sure there are no greater element on the left of the key value, check the key value again and do the swap until there is no element on the left that is greater than key value.
- If key elements left value is smaller than key value pick another element.

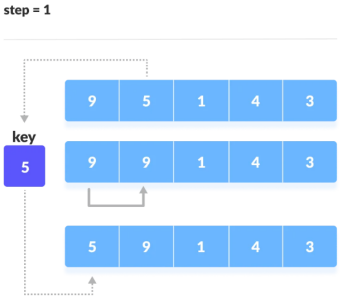
Insertion Sort

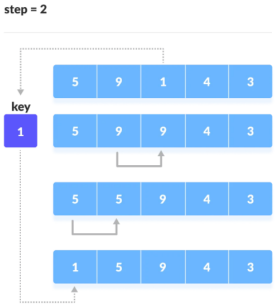
The process shown in the figure:



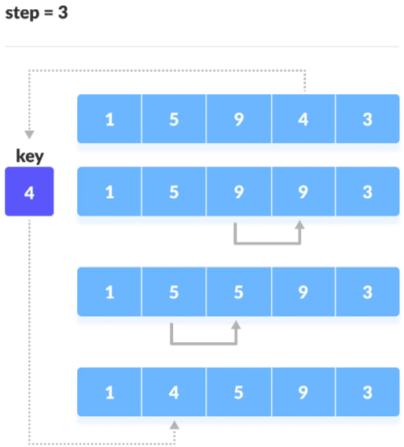
Steps of Insertion Sort

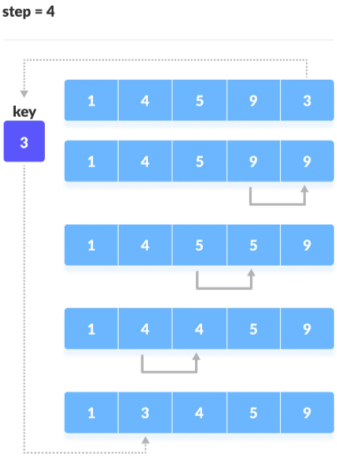
Start from second element!..





Steps of Insertion Sort





Insertion Sort Code

```
1  void insertionSort(vector<int>&v){
2      int size=v.size();
3
4      for(int i = 1 ; i < size ; i++ ){
5          int indexToCompare = i;
6          while( indexToCompare > 0 ){
7              if( v[indexToCompare] >= v[indexToCompare-1] ) break;
8              else {
9                  swap(v[indexToCompare] , v[indexToCompare-1] );
10                 indexToCompare--; // Decreasing position by one since we
       will check the key value's position to make sure it is on the right
       place
11             }
12         }
13     }
14 }
```

Selection Sort

- **Time Complexity :** $O(n^2)$
- **Space Complexity :** $O(1)$

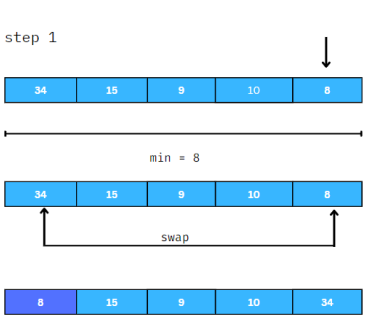
Selection Sort is a simple algorithm that divides array to two subarray .This subarrays are sorted and unsorted subarray.Initially sorted subarray is empty and unsorted subarray is array's itself.

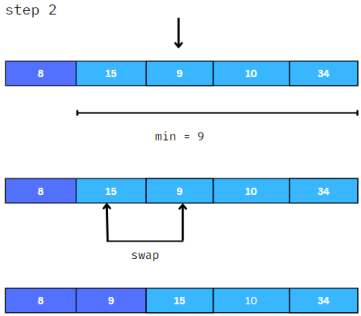
Algorithm works like this: Find the smallest value in the array and swap that value with array's first element and find the second smallest value in the array and swap that value with array's second element and that process goes until the end of the array.

As you have noticed left subarray(which is sorted) expanding and right subarray is narrowing as the process continues.At the end right subarray which is unsorted is empty.

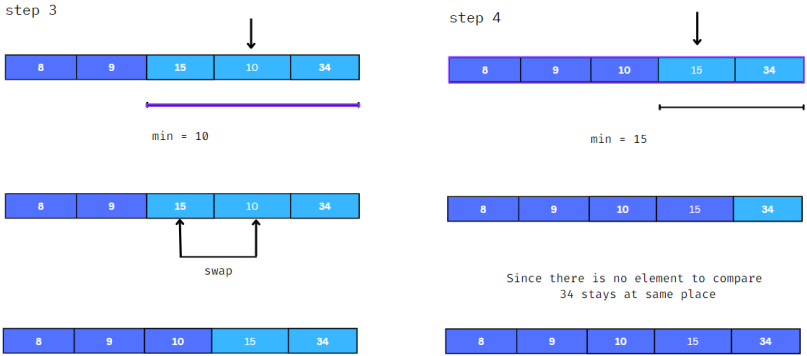
- Since the complexity of selection sort is $O(n^2)$ it is not suitable for large data sets.

Selection Sort





Selection Sort



Selection Sort Code

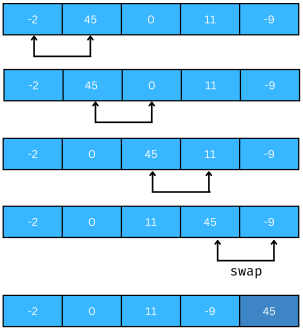
```
1  void selectionSort(vector<int> &v){
2      int size=v.size();
3
4      for(int i=0; i<size ; i++){
5
6          int minIndex=i;
7          for(int s=i;s<size;s++){
8
9              // To sort in descending order you can change "<"  with ">" in
       this line
10             if(v[s]<v[minIndex]){
11                 minIndex=s;
12             }
13         }
14         swap(v[i],v[minIndex]);
15     }
16 }
```

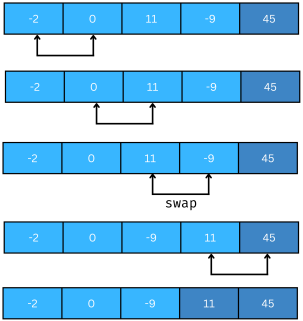
Bubble Sort

- **Time Complexity :** $O(n^2)$
- **Space Complexity :** $O(1)$

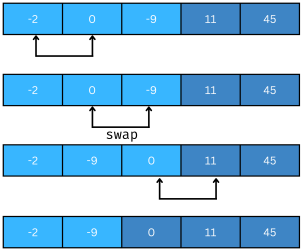
Bubble Sort is a simple sorting algorithm that is comparison based.
Adjacent elements compared and if they are not in the intended order (ascending or descending), they are swapped.
It is called **bubble** sort since the movement of array elements is just like the movement of air bubbles in the water.

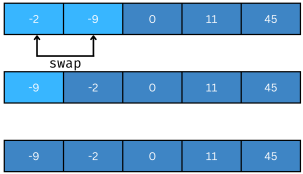
Bubble Sort


-2 45 0 11 -9
-2 45 0 11 -9
-2 0 45 11 -9
-2 0 11 45 -9
swap
-2 0 11 -9 45


-2 0 11 -9 45
-2 0 11 -9 45
-2 0 11 -9 45
swap
-2 0 -9 11 45
-2 0 -9 11 45

Bubble Sort


-2
0
-9
11
45
-2
0
-9
11
45
swap
-2
-9
0
11
45
-2
-9
0
11
45


-2
-9
0
11
45
swap
-9
-2
0
11
45
-9
-2
0
11
45

Bubble Sort Code

```
1  void bubbleSort(vector<int> & v) {
2    int n = v.size();// Size of the vector
3    for (int step = 0; step < n; step++ ) {// There will be total of n step.
       Since we want to sort n elements.
4      // For each step we place one element to the its right place.
5      // There is "n-step" iteration because when we done our x step we also
       sorted the last x elements.
6      // So we will not deal with that part again.
7      for (int i = 0; i < n - step; i++ ) {
8          // If an element is greater than its right adjacent then swap them
9          // Also if you want to make descending ordered array instead of
       ascending make ">" sign "<" .
10         if (v[i] > v[i + 1]) {
11             swap( v[i] , v[i+1] );
12         }
13     }
14   }
15 }
```

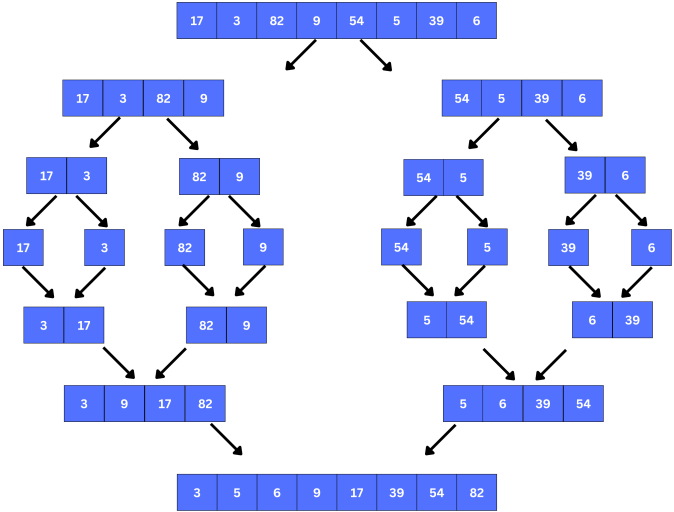
Merge Sort

- **Time Complexity :** $O(n.log(n))$
- **Space Complexity :** $O(n)$

Merge sort is a divide and conquer algorithm. Divide and conquer is a technique for solving large problems by breaking the problems into smaller subproblems and solving these subproblems then combining them to reach the final solution. To use divide and conquer technique ,recursion is used.

Merge sort algorithms divide the array into halves repeatedly and get subarrays until the size of the subarray is 1. Then merge function pick these subarrays and merge them to sort the entire array.

Merge Sort


17 3 82 9 54 5 39 6
17 3 82 9
54 5 39 6
17 3
82 9
54 5
39 6
17
3
82
9
54
5
39
6
3 17
82 9
5 54
6 39
3 9 17 82
5 6 39 54
3 5 6 9 17 39 54 82

Merge Sort Code

```
1  void mergeSort(vector<int> &v , int left , int right) {
2      // Base case is the case when the left and right index is equal
3      // That means we have only one element that passed the function so
       there is no need to any sorting
4      // We will do merging after that
5      if (left < right) {
6          // mid is the point where is the array is divided in to two
       subarray
7          int mid = (right + left)/ 2 ;
8
9          // recursivelly calling mergeSort function
10         mergeSort(v , left , mid);
11         mergeSort(v , mid + 1, right);
12
13         // Merge these two sorted subarrays
14         merge(v , left , mid , right);
15     }
16 }
```

Merge Sort Code

```
1  void merge(vector<int> &v, int left, int mid, int right) {
2      // Since we will merge the two sorted parts of the array we will need
       extra space
3      // We will need a Left vector to store left part's elements
4      // and RIGHT vector to store right part's elements
5      // Getting the sizes of the left and right subarrays's size to create
       same sized vector
6      int sizeLeft = mid - left + 1;
7      int sizeRight = right - mid;
8      vector<int> LEFT(sizeLeft),RIGHT(sizeRight);
9      // Storing elements in to LEFT and RIGHTvectors
10     for (int i = 0; i < sizeLeft; i++){
11         LEFT[i] = v[left + i];
12     }
13     for (int j = 0; j < sizeRight; j++){
14         RIGHT[j] = v[mid + 1 + j];
15     }
```

Merge Sort Code

```
1       // Here I have two pointers for two subarrays and
2       // one pointer for our main array to keep track of the index
3       // of the numbers that merged.
4       int ptLeft = 0;
5       int ptRight = 0;
6       int ptMain = left;
7       // Until we reach either end of L or R , we will pick smaller elements
8       // among the elements of L and R and put them in to
9       // our main vector v with correct positioning
10      while(ptLeft < sizeLeft && ptRight < sizeRight) {
11          if (LEFT[ptLeft] <= RIGHT[ptRight]) {
12              v[ptMain] = LEFT[ptLeft];
13              ptLeft++;
14          }
15          else {
16              v[ptMain] = RIGHT[ptRight];
17              ptRight++;
18          }
19          ptMain++;
```

Merge Sort Code

```
1       // If there will be no other elements in either L or R we will
2       // put rest of the elements in to our main vector v
3       while (ptLeft < sizeLeft) {
4           v[ptMain] = LEFT[ptLeft];
5           ptLeft++;
6           ptMain++;
7       }
8
9       while (ptRight< sizeRight) {
10          v[ptMain] = RIGHT[ptRight];
11          ptRight++;
12          ptMain++;
13      }
14  }
```

Quick Sort

- **Time Complexity :** $O(n.log(n))$
- **Space Complexity :** $O(1)$
- **Preliminary Information:**
  The Dictionary definition of pivot is "The most important part of something".But in this algorithm, the pivot is the element that we will choose for partitioning the array by comparing it with other elements.

Quick sort is a highly efficient sorting algorithm and is based on **partitioning** the array into smaller subarrays. A large array is partitioned into two subarrays, one of which holds values smaller than the specified value that chosen from the array which we will call a **pivot**. Other array holds values greater than the pivot value.
After the partition process Quick Sort calls itself recursively twice to sort these two resulting subarrays.

- There are different ways of choosing pivot :first element,last element,random element,median value. But we will pick the last element as pivot.

Quick Sort

How Quick Sort works?

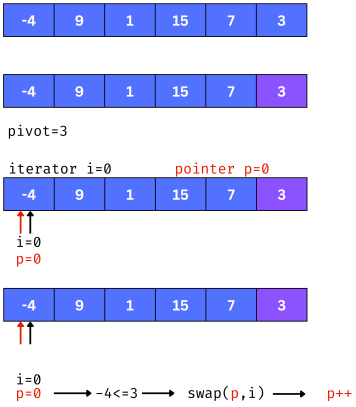
- Make the right-most index value pivot.
- Partition the array using pivot value.
- Quick Sort left subarray recursively.
- Quick Sort right subarray recursively.

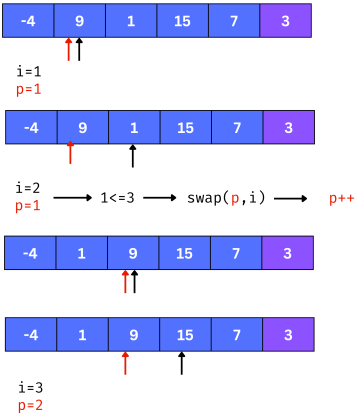
How partitioning process works?

- Keep a pointer **p** for greater elements.
- Traverse each element of the array.
- If an element is less than or equal to the pivot, swap that element with p and increment p by one.

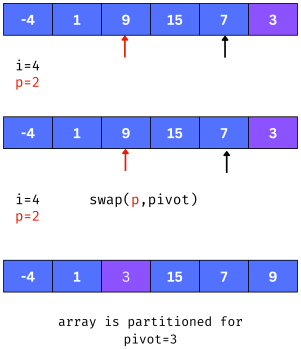
This process continues until we looked at every element until the pivot. Then pivot is swapped with p because every element before the **p** is less than or equal to the pivot and every element after p is greater than the pivot.

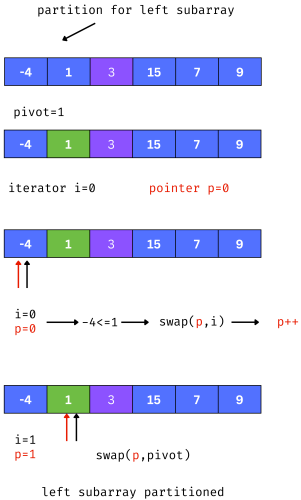
Quick Sort


-4 9 1 15 7 3
-4 9 1 15 7 3
pivot=3
iterator i=0
pointer p=0
-4 9 1 15 7 3
i=0
p=0
-4 9 1 15 7 3
i=0
p=0
-4<=3
swap(p,i)
p++

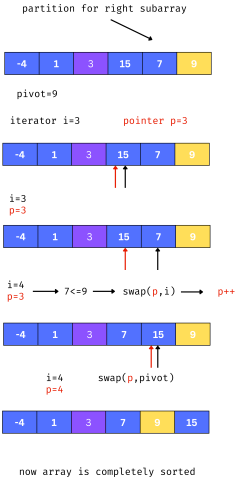

-4 9 1 15 7 3
i=1
p=1
-4 9 1 15 7 3
i=2
p=1
1<=3
swap(p,i)
p++
-4 1 9 15 7 3
-4 1 9 15 7 3
i=3
p=2

Quick Sort


-4 1 9 15 7 3
i=4
p=2
-4 1 9 15 7 3
i=4
p=2
swap(p,pivot)
-4 1 3 15 7 9
array is partitioned for
pivot=3


partition for left subarray
-4 1 3 15 7 9
pivot=1
-4 1 3 15 7 9
iterator i=0
pointer p=0
-4 1 3 15 7 9
i=0
p=0
-4<=1
swap(p,i)
p++
-4 1 3 15 7 9
i=1
p=1
swap(p,pivot)
left subarray partitioned

Quick Sort


partition for right subarray
-4 1 3 15 7 9
pivot=9
iterator i=3
pointer p=3
-4 1 3 15 7 9
i=3
p=3
-4 1 3 15 7 9
i=4
p=3
7<=9
swap(p,i)
p++
-4 1 3 7 15 9
i=4
p=4
swap(p,pivot)
-4 1 3 7 9 15
now array is completely sorted

Quick Sort Code

```
1  void quickSort(vector<int>& v,int start ,int end){
2      if(start>=end) return;
3      int pos=start; //Position for end point of smaller elements
4      int pivot=v[end];
5
6      for(int i = start ; i < end ; i++ ){
7        if( v[i] < pivot ){// If an element smaller than pivot
8          swap(v[pos],v[i]);// Swap them
9          pos++;  // Increment end point of smaller elements position
10       }
11     }
12
13     // Taking pivot to its correct position
14     swap(v[end],v[pos]);
15     // Do the same operations to left and right subarrays
16     quickSort( v , start , pos-1 );
17     quickSort(v,pos+1 , end);
18 }
```

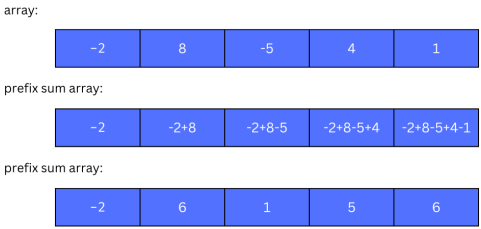
Array Problems

- Prefix Sum
- Two Pointers

Prefix Sum

- **Time Complexity :** $O(n)$
- **Space Complexity :** $O(n)$

Given an array **arr[ ]** of size n, its prefix sum array is another array **prefixSum[ ]** of the same size, such that the value of **prefixSum[i]** is equal to **arr[0] + arr[1] + arr[2] . . . arr[i]** .
To fill the prefix sum array, we run through index 1 to last and keep on adding the present element with the previous value in the prefix sum array.
It requires linear time preprocessing and is widely used due to its simplicity and effectiveness.



Prefix Sum

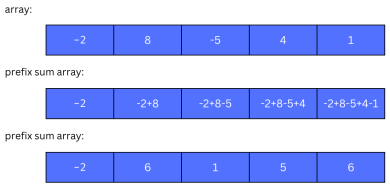
- Imagine that you have given an array of numbers such that the array's size is n. And you are asked to find the sum of the all elements in the given range.What would you do? .You would simply just iterate through the array in the given range and add all of them right ?
- But if you are asked to do this operation many times with different range , that kind of solution might not work. Even if it gives us the answers , that solution won't be the efficient one .In this case we have to find better solution.

sum in the range = ?

Prefix Sum

- Prefix Sum helps us here .So what is Prefix Sum? Prefix Sum array is a data structure design which helps us to answer several queries such as sum in a given range in **constant time** which would otherwise take linear time. It requires a linear time preprocessing and is widely used due to its simplicity and effectiveness.

- Given an array **arr[ ]** of size n, its prefix sum array is another array **prefixSum[ ]** of the same size, such that the value of **prefixSum[i]** is equal to **arr[0] + arr[1] + arr[2] . . . arr[i]** .To fill the prefix sum array, we run through index 1 to last and keep on adding the present element with the previous value in the prefix sum array.



Prefix Sum Implementation

- **Time Complexity :** $O(n)$
- **Space Complexity :** $O(n)$

```
1 void fillPrefixSum(int arr[], int n, int prefixSum[])
2 {
3     prefixSum[0] = arr[0];
4
5     for (int i = 1; i < n; i++){
6         prefixSum[i] = prefixSum[i - 1] + arr[i];
7     }
8
9 }
```

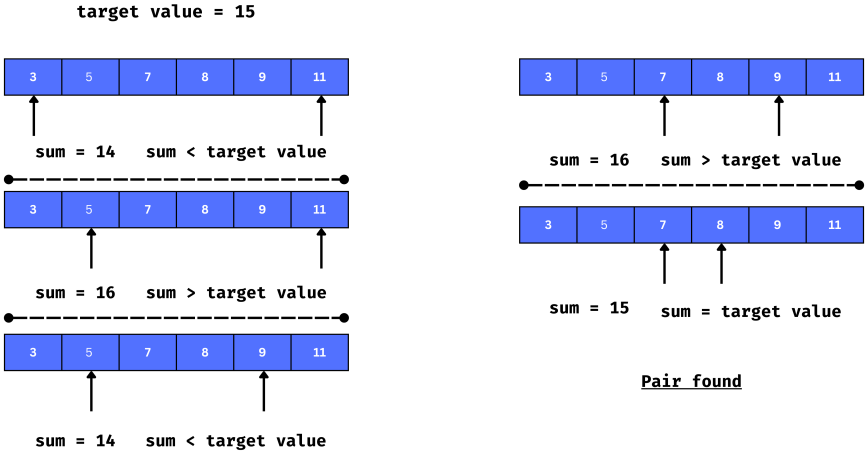
Two Pointers

Two pointers is a technique that involves using two pointers to save time and space. (Here, pointers are basically array indexes). The idea here is to iterate two different parts of the array simultaneously to get the answer faster.

- In many problems involving collections such as arrays or lists, we have to analyze each element of the collection compared to its other elements.
- There are many approaches to solving problems like these. For example we usually start from the first index and iterate through the data structure one or more times depending on how we implement our code.
- Sometimes we may even have to create an additional data structure depending on the problem's requirements. This approach might give us the correct result, but it likely won't give us the most space and time efficient result.

Two Pointers

Example:



Two Pointers

This is why the two-pointer technique is efficient. We are able to process two elements per loop instead of just one. Common patterns in the two-pointers approach :

- Two pointers, each starting from the beginning and the end until they both meet.
- One pointer moving at a slow pace, while the other pointer moves at twice the speed.

It is used for applications like:

- In a sorted array, find if a pair exists with a given sum S.
- Find the middle of a linked list .
- Reversing an array .
- Merging two sorted arrays.
- Partition function in quick sort.

Two Pointers

```
1  // This function returns if a pair exist with given sum S using two
       pointers method
2  bool pairExists(int arr[], int n, int S){
3      int pointer1 = 0;
4      int pointer2= n-1;
5      while( pointer1 < pointer2){
6          int current_sum = arr[pointer1] + arr[pointer2];
7          if ( current_sum == S){
8              return true;
9          }
10         else if ( current_sum < S ){
11             pointer1++;
12         }
13         else if ( current_sum > S ){
14             pointer2--;
15         }
16     }
17     return false;
18 }
```

Questions

- Range Sum Query
- Kth Largest Element in an Array
- Find the Highest Altitude
- Merge Sorted Array
- Remove Duplicates from Sorted Array
- Is Subsequence
- Two Sum-II
- Reverse Vowels of a String
- Reverse Words in a String

References

https://afteracademy.com/blog/what-is-the-two-pointer-technique
https://www.programiz.com/dsa/insertion-sort
https://www.programiz.com/dsa/quick-sort
https://www.tutorialspoint.com/data$_s$*tructures*$_a$*lgorithms*/*quick*$_s$*ort*$_a$*lgorithm.htm*
https://iq.opengenus.org/prefix-sum-array/
https://www.geeksforgeeks.org/prefix-sum-array-implementation-applications-competitive-programming/
https://algodaily.com/lessons/using-the-two-pointer-technique