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

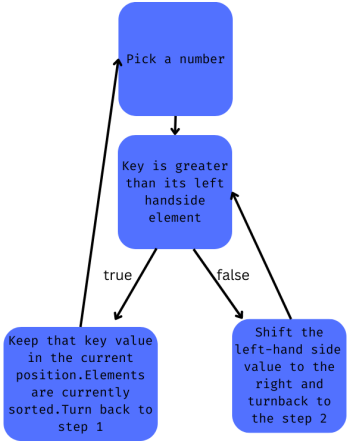
- **Time Complexity :** $O(n^2)$
- **Space Complexity :** $O(1)$

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 shift the left element to the position of the key value.
  - Then continue to this process until left hand value is smaller than the key value.
- If key elements left value is smaller than key value pick another element.

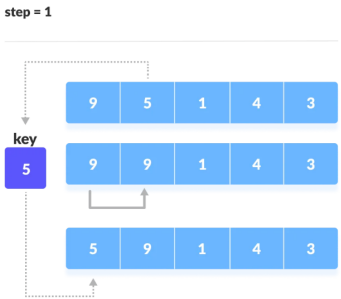
Insertion Sort

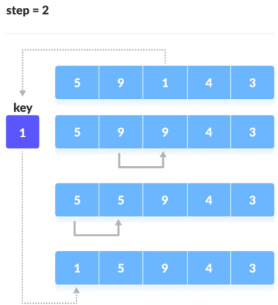
The process shown in the figure:



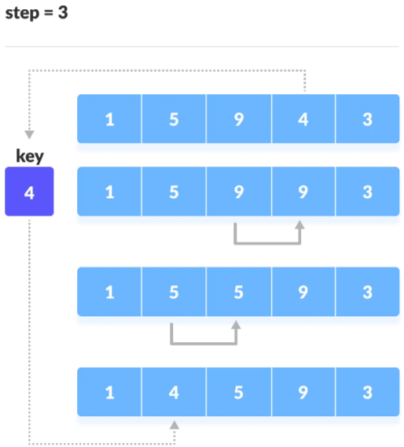
Steps of Insertion Sort

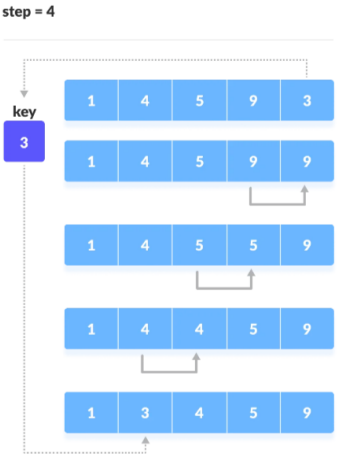
Start from second element!..





Steps of Insertion Sort





Insertion Sort Code

```
1  void insertionSort(vector<int>&v){
2      for(int i = 1 ; i < v.size() ; i++ ){
3          int key = v[i]; // Key is the element that we are going to compare
4          int position = i-1;
5
6          while( position >= 0 && key < v[position] ){
7              v[position+1] = v[position]; // Shifting to the right
8              position--; // Decreasing position by one since we will check
       the previous position to insert
9          }
10
11         v[position+1] = key;
12     }
13 }
```

Selection Sort

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

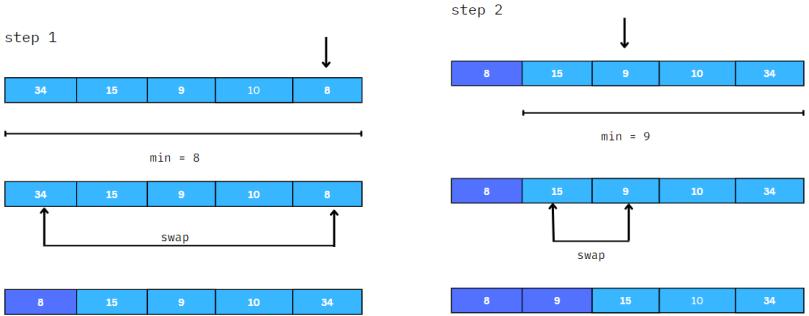
Selection Sort is a simple algorithm that divides array to two subarray .This subarrays are sorted and unsorted subarray.Initially sorted subarray is empty and unsorted subarray is array's itself.

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.

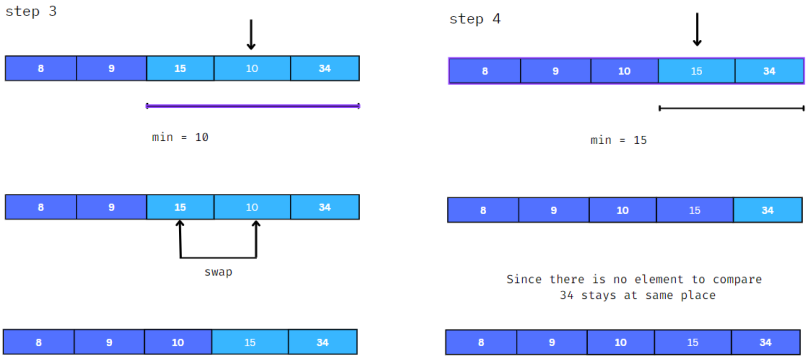
As you have noticed left subarray(which is sorted) expanding and right subarray is narrowing as the process continues.At the end right subarray which is unsorted is empty.

- 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

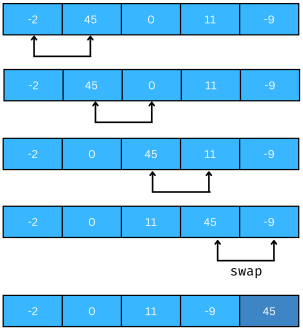
```
1  void selectionSort(vector<int> &v){
2      int size=v.size();
3
4      for(int i=0; i<size ; i++){
5
6          int minIndex=i;
7          for(int s=i;s<size;s++){
8
9              // To sort in descending order you can change "<"  with ">" in
       this line
10             if(v[s]<v[minIndex]){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

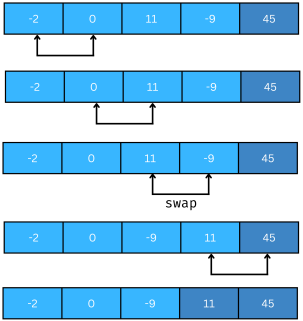
Bubble Sort

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

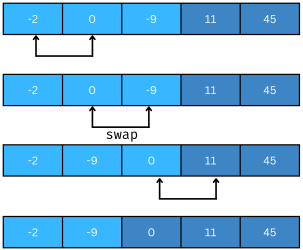
Bubble Sort is a simple sorting algorithm that is comparison based.
Adjacent elements compared and if they are not in the intended order (ascending or descending), they are swapped.
It is called **bubble** sort since the movement of array elements is just like the movement of air bubbles in the water.

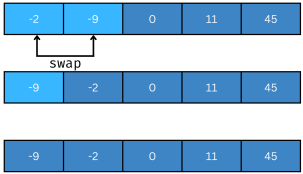
Bubble Sort


-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 }
```

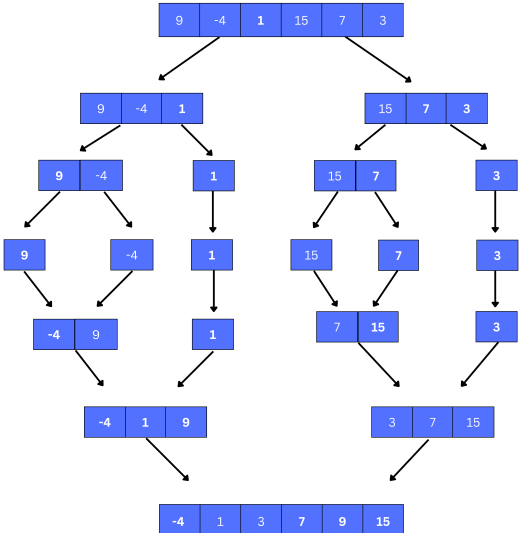
Merge Sort

- **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



Merge Sort Code

```
1  void merge(vector<int> &v, int left, int mid, int right) {
2      // Getting the sizes of the left and right subarrays's size to create
       same sized vector
3      int size1 = mid - left + 1;
4      int size2 = right - mid;
5
6      vector<int> L(size1),R(size2);
7      // Storing elements in to L(left) and R(right) vectors
8      for (int i = 0; i < size1; i++){
9          L[i] = v[left + i];
10     }
11     for (int j = 0; j < size2; j++){
12         R[j] = v[mid + 1 + j];
13     }
14     int i, j, k;
15     i = 0;
16     j = 0;
17     k = left;
18     // Until we reach either end of L or R , we will pick greater elements
```

Merge Sort Code

```
1     // Until we reach either end of L or R , we will pick greater elements
2     // among the elements of L and R and put them in to our main vector v
      with correct positioning
3     while (i < size1 && j < size2) {
4         if (L[i] <= R[j]) {
5             v[k] = L[i];
6             i++;
7         }
8         else {
9             v[k] = R[j];
10            j++;
11        }
12        k++;
13    }
14    while (i < size1) {
15        v[k] = L[i];
16        i++;
17        k++;
18    }
```

Merge Sort Code

```
1       while (j < size2) {
2           v[k] = R[j];
3           j++;
4           k++;
5       }
6   }
7   void mergeSort(vector<int> &v , int left , int right) {
8       if (left < right) {
9           // mid is the point where is the array is divided in to two
        subarray
10          int mid = (right + left)/ 2 ;
11          // recursivelly calling mergeSort function
12          mergeSort(v , left , mid);
13          mergeSort(v , mid + 1, right);
14
15          // Merge these two sorted subarrays
16          merge(v , left , mid , right);
17      }
18  }
```

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 choosen 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?

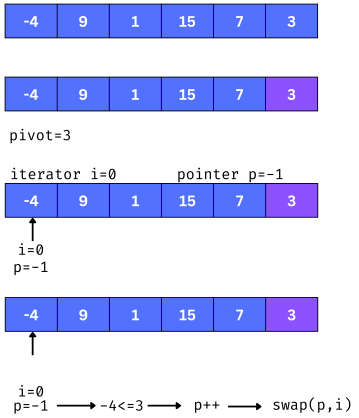
- Make the right-most index value pivot.
- Partition the array using pivot value.
- Quick Sort left subarray recursively.
- Quick Sort right subarray recursively.

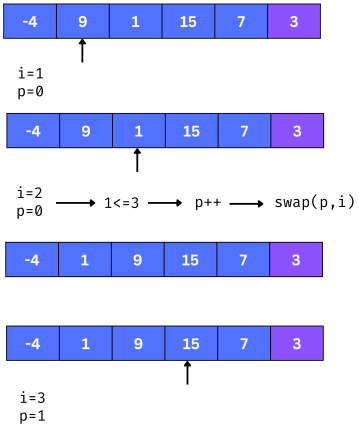
How partitioning process works?

- Keep a pointer **p** for greater elements and initialize it as (low-1).
- Traverse each element of the array.
- If an element is less than or equal to the pivot, increment p by one and swap that element with p.

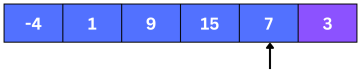
This process continues until we looked at every element until the pivot. Then pivot is swapped with p+1 because every element before the **p+1** is less than or equal to the pivot and every element after p+1 is greater than the pivot.

Quick Sort


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


-4 9 1 15 7 3
i=1
p=0
-4 9 1 15 7 3
i=2
p=0
1<=3
p++
swap(p,i)
-4 1 9 15 7 3
-4 1 9 15 7 3
i=3
p=1

Quick Sort



i=3
p=1

i=3
p=1

swap(p+1,pivot)

partition for left subarray

pivot=1

iterator i=0 pointer p=-1

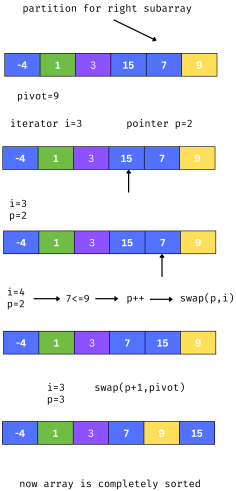
i=0
p=-1

-4<=1

p++

swap(p,i)

Quick Sort


partition for right subarray
-4 1 3 15 7 9
pivot=9
iterator i=3
pointer p=2
-4 1 3 15 7 9
i=3
p=2
-4 1 3 15 7 9
i=4
p=2
7<=9
p++
swap(p,i)
-4 1 3 7 15 9
i=3
p=3
swap(p+1,pivot)
-4 1 3 7 9 15
now array is completely sorted

Quick Sort Code

```
// Function to partition the array and return pivots index
int partition(vector<int> &v , int low, int high) {
    // Select last element as pivot
    int pivot = v[high];

    // Pointer for greater elements
    int i = (low - 1);

    //Now iterate through the array and compare the elements with pivot
    //Then put them in to right position in the array
    for (int j = low; j < high; j++) {
        if (v[j] <= pivot) {

        // if element smaller than pivot is found
        // swap it with the greater element pointed by i
        i++;

        // swap element at i with element at j
        swap(v[i], v[j]);
```

Quick Sort Code

```
1           }
2       }
3       // Now partition is almost done but we need to put pivot to its
        position
4       // We will swap pivot with the greater element at i
5       // i+1 is always greater or equal to our pivot and belongs to right
        subarray
6       // so we can put our pivot freely there
7       swap(v[i + 1], v[high]);
8
9       // Return the partition point
10      // This is the point where our pivot is right now
11      return (i + 1);
12      // Actually what we have done right here is putting our pivot to its
        correct position in a sorted array
13      // and placing other elements at their right place compared to the
        pivot.
14  }
```

Quick Sort Code

```
1  void quickSort(vector<int> &v, int low, int high) {
2    if (low < high) {
3      // Find the pivot element such that
4      // elements less than or equal to pivot are on the left of pivot  and
5      // elements greater than pivot are on right of pivot
6      int pi = partition(v, low, high);
7      // Array is partitioned ,so we will recursively call qucikSort
       function to
8      // sort left and right side of the pivot. This process goes until
       there is only one
9      // element in the subarray. (In this case low == high)
10     quickSort(v, low, pi - 1);
11
12     quickSort(v, pi + 1, high);
13   }
14 }
```

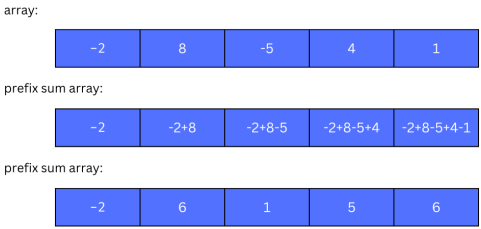
Array Problems

- 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

It is used for applications like:

- Find sum of all elements in a given range.
- Find product of all elements in a given range.
- Find maximum subarray sum.
- Maximum subarray such that sum is less than some number.

Prefix Sum Implementation

```
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

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

```
// This function returns if a pair exist with given sum S using two
    pointers method
bool pairExists(int arr[], int n, int S){
    // Here i and j are our two pointers
    int i = 0;
    int j = n-1
    while( i < j)
    {
        curr_sum = arr[i] + arr[j];
        if ( curr_sum == S){
            return true;
        }else if ( curr_sum < X ){
            i = i + 1;
        }else if ( curr_sum > X ){
            j = j - 1
        }
    }
    return false;
}
```

Questions

Range Sum Query **Solution**
Kth Largest Element in an Array **Solution**
Find the Highest Altitude **Solution**
Merge Sorted Array **Solution**
Remove Duplicates from Sorted Array **Solution**
Is Subsequence **Solution**
Two Sum-II **Solution**

References

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://afteracademy.com/blog/what-is-the-two-pointer-
techniquehttps://afteracademy.com/blog/what-is-the-two-pointer-technique
https://www.geeksforgeeks.org/prefix-sum-array-implementation-applications
https://algodaily.com/lessons/using-the-two-pointer-technique