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

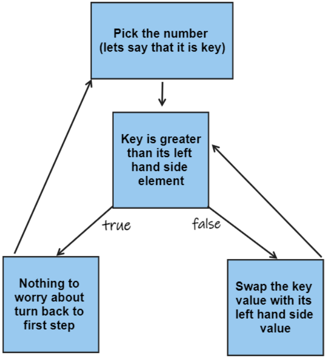
Insertion Sort

Insertion Sort is one of the simplest algorithm with simple implementation.
Algorithm iterates through the array and at every iteration it keeps a key value in order then inserts that key value to the suitable place where is left hand side is smaller or equal and right hand side is greater or equal .
Main idea behind the insertion sort is that algorithm takes elements in order from the array and puts this element in a suitable position.
In suitable position elements on the left hand side is less (or greater) than or equal to our key value and the elements on the right hand side is greater (or less) than or equal to our key value .

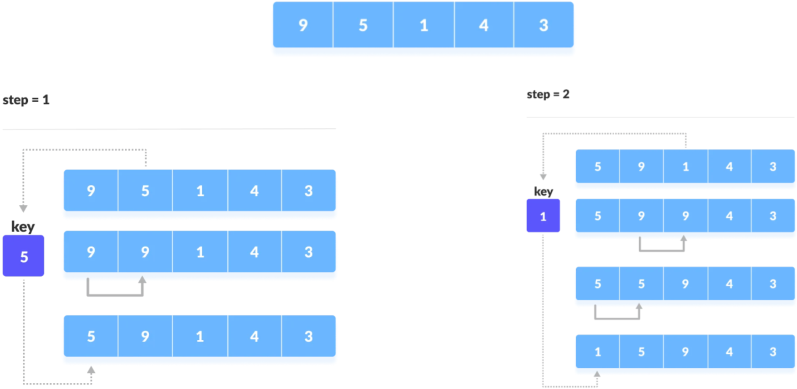
Insertion Sort

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

The insertion sort algorithm is one of the simplest algorithms to understand and easy to implement.Main idea of the algorithm is to insert every element into suitable position while iterating through the array . The process shown in the figure :

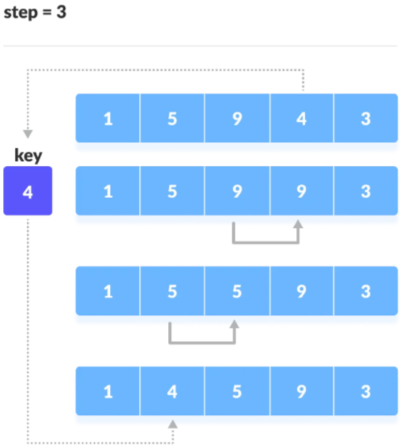

Pick the number (lets say that it is key)
Key is greater than its left hand side element
true
false
Nothing to worry about turn back to first step
Swap the key value with its left hand side value

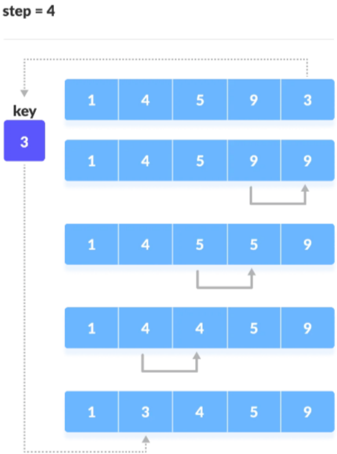
Steps of Insertion Sort



Start from second element!..

Steps of Insertion Sort





Start from second element!..

Insertion Sort Code

```cpp

void insertionSort(vector<int>&v){
    for(int i=1;i<v.size();i++){
        int key=v[i]; // Key is the element that we are going to compare
        int position=i-1;

        while( x>=0 && key < v[x] ){
            v[x+1]=v[x]; // Shifting to the right
            position--; // Decreasing position by one since we will check the previous position to insert
        }
        v[position+1]=key;
    }
}
```

Selection Sort

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

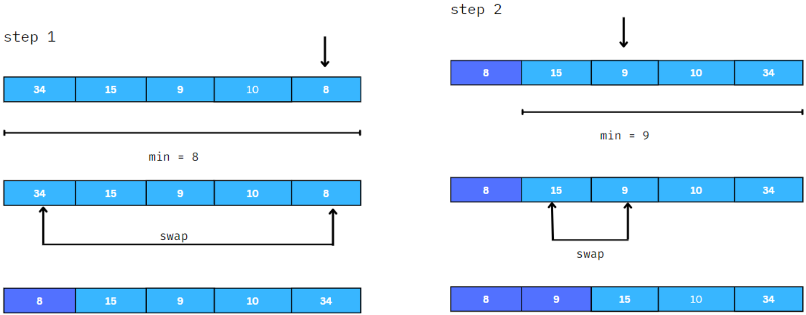
Selection Sort is a simple algorithm that divides array to two sub array .This sub arrays are sorted and unsorted sub array.Initially sorted sub array is empty and unsorted sub array is array's itself.

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 sub array(which is sorted) expanding and right sub array is narrowing as the process continues.At the end right sub array which is unsorted is empty.

-Since the complexity of selection sort is O(n^2)$itisnotsuitableforlargedatasets$.

Selection Sort



step 1

34 15 9 10 8

min = 8

34 15 9 10 8

swap

8 15 9 10 34

step 2

8 15 9 10 34

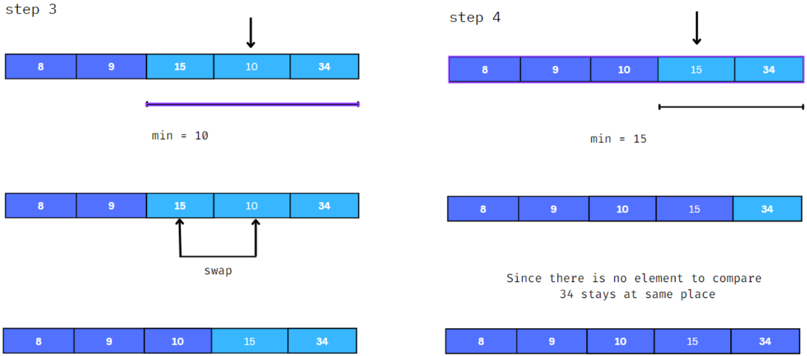
min = 9

8 15 9 10 34

swap

8 9 15 10 34

Selection Sort



step 3

8 9 15 10 34

min = 10

8 9 15 10 34

swap

8 9 10 15 34

step 4

8 9 10 15 34

min = 15

8 9 10 15 34

Since there is no element to compare
34 stays at same place

8 9 10 15 34

Selection Sort Code

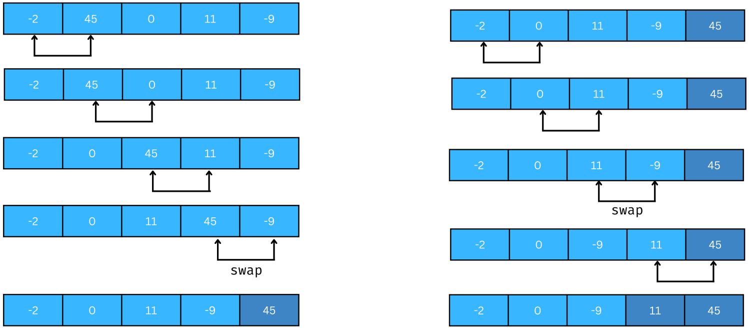
```cpp
void swap(vector<int>& v,int i1,int i2){
    int temp=v[i1];
    v[i1]=v[i2];
    v[i2]=temp;
}
void selectionSort(vector<int> &v){
    int size=v.size();
    for(int i=0; i<size ; i++){
        int minIndex=i;
        for(int s=i;s<size;s++){
            // To sort in descending order you can change "<"  with ">" in this line
            if(v[s]<v[minIndex]){
                minIndex=s;
            }
        }
        swap(v,i,minIndex);
    }
}
```

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

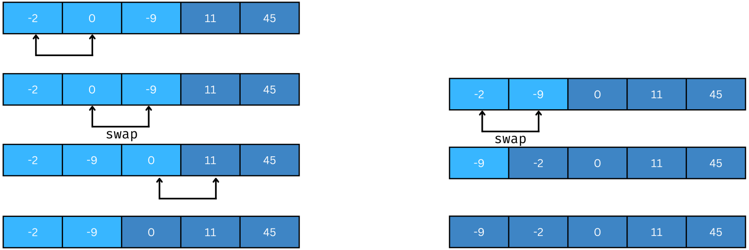
-2 0 11 -9 45

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

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

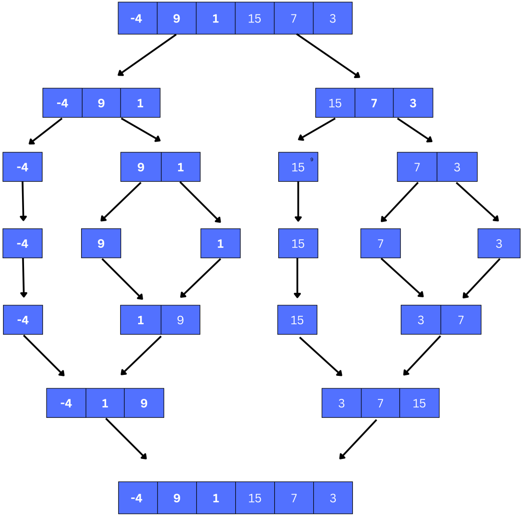
Merge Sort

- 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


-4 9 1 15 7 3
-4 9 1
15 7 3
-4
9 1
15
9
7 3
-4
9
1
15
7
3
-4
1 9
15
3 7
-4 1 9
3 7 15
-4 9 1 15 7 3

Merge Sort Code

```cpp
void merge(vector<int> &v, int left, int mid, int right) {
    // Getting the sizes of the left and right subarrays’s size to create same sized vector
    int size1 = mid - left + 1;
    int size2 = right - mid;

    vector<int> L(size1),R(size2);
    // Storing elements in to L(left) and R(right) vectors
    for (int i = 0; i < size1; i++){
        L[i] = v[left + i];
    }
    for (int j = 0; j < size2; j++){
        R[j] = v[mid + 1 + j];
    }
    int i, j, k;
    i = 0;
    j = 0;
    k = left;
    // Until we reach either end of L or R , we will pick greater elements
```

Merge Sort Code

```
    // Until we reach either end of L or R , we will pick greater elements
    // among the elements of L and R and put them in to our main vector v with correct positioning
    while (i < size1 && j < size2) {
        if (L[i] <= R[j]) {
            v[k] = L[i];
            i++;
        }
        else {
            v[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < size1) {
        v[k] = L[i];
        i++;
        k++;
    }
```

Merge Sort Code

```
    while (j < size2) {
        v[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(vector<int> &v , int left , int right) {
    if (left < right) {
        // mid is the point where is the array is divided in to two
    subarray
        int mid = (right + left)/ 2 ;
        // recursivelly calling mergeSort function
        mergeSort(v , left , mid);
        mergeSort(v , mid + 1, right);

        // Merge these two sorted subarrays
        merge(v , left , mid , right);
    }
}
```

Quick Sort

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

Quciksort is another sorting algorithm that uses divide and conquer aproach. As we have learned before idea behind the divide and conquer aproach is dividing problems to subproblems then solving these subproblems and finalizing the solution.

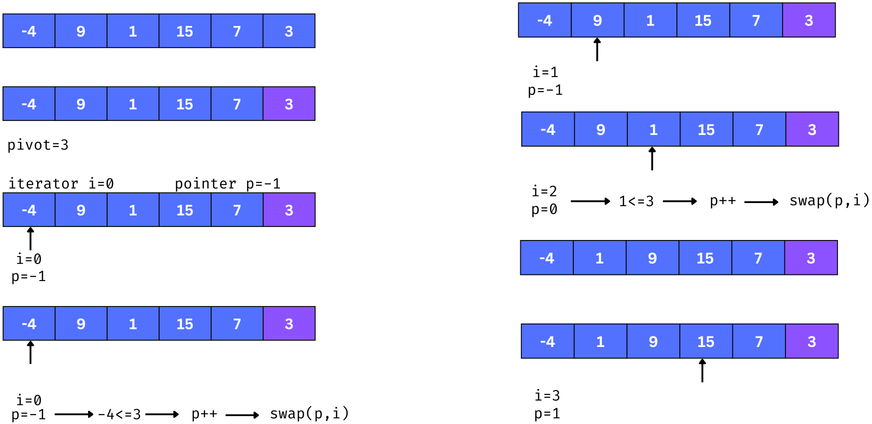
- Pivot : the most important part of something.
  The main idea of the algorithm is to choose an array element as a pivot and then partition this array by comparing them to the pivot element. Now we have left and right subarray. Our array will be sorted when the same process is applied for these subarrays recursively. There are different ways of choosing pivot (First element,Last element,Random element,Median value).But today we will pick the last element as pivot.

Quick Sort

How do we do this partitioning ?

First of all, we will keep a pointer p for greater elements and initialize it as (low-1) then traverse each element of the array. If an element is less than or equal to the pivot, p is incremented one and it is swapped with p. This process continued until we looked at every element of the array. Then pivot is swapped with p+1 since before the p every element is less than or equal to the pivot and after p+1 every element 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=-1

-4	9	1	15	7	3

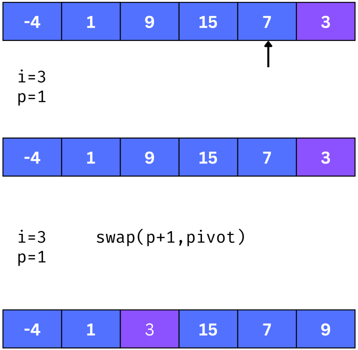
i=2
p=0 → 1<=3 → p++ → swap(p,i)

-4	1	9	15	7	3

-4	1	9	15	7	3

i=3
p=1

Quick Sort



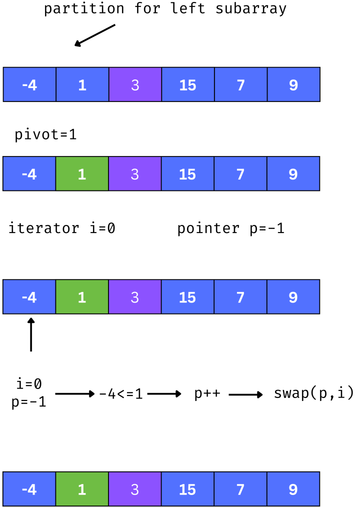
-4	1	9	15	7	3

i=3
p=1

-4	1	9	15	7	3

i=3 swap(p+1,pivot)
p=1

-4	1	3	15	7	9



partition for left subarray

-4	1	3	15	7	9

pivot=1

-4	1	3	15	7	9

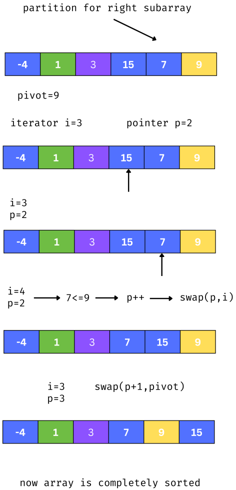
iterator i=0 pointer p=-1

-4	1	3	15	7	9

i=0
p=-1 → -4<=1 → p++ → swap(p,i)

-4	1	3	15	7	9

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

```cpp
// 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

```
        }
    }
    // Now partition is almost done but we need to put pivot to its position
    // We will swap pivot with the greater element at i
    // i+1 is always greater or equal to our pivot and belongs to right subarray
    // so we can put our pivot freely there
    swap(v[i + 1], v[high]);

    // Return the partition point
    // This is the point where our pivot is right now
    return (i + 1);
    // Actually what we have done right here is putting our pivot to its correct position in a sorted array
    // and placing other elements at their right place compared to the pivot.
}
```

Quick Sort Code

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

    quickSort(v, pi + 1, high);
  }
}
```

Complexities of Sorting Algorthms

Bubble Sort	Insertion Sort	Selection Sort	Merge Sort	Quick Sort
$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(nlog(n)$	$O(nlog(n)$

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 $arr[0] + arr[1] + arr[2] \ldots arr[i]$.
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

```c
void fillPrefixSum(int arr[], int n, int prefixSum[])
{
    prefixSum[0] = arr[0];

    for (int i = 1; i < n; i++){
        prefixSum[i] = prefixSum[i - 1] + arr[i];
    }

}
```

Two Pointers

Two pointer approach is an essential part of a programmer's toolkit.It involves using two pointers to save time and space. (Here, pointers are basically array indexes).
Just like Binary Search is an optimization on the number of trials needed to achieve the result, this approach is used for the same benefit. The idea here is to iterate two different parts of the array simultaneously to get the answer faster.
There are primarily two ways of implementing the two-pointer technique:

- One pointer at each end.
- Both pointers start from the beginning but one pointer moves at a faster pace than the other one.

Two Pointers

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;
}
```

References

Figures for insertion sort : https://www.programiz.com/dsa/insertion-sort
Quick Sort Algorithm's Code : https://www.programiz.com/dsa/quick-sort
(Changed partially)
Prefix Sum: https://iq.opengenus.org/prefix-sum-array/
https://www.geeksforgeeks.org/prefix-sum-array-implementation-applications-competitive-programming/
Two Pointers: https://afteracademy.com/blog/what-is-the-two-pointer-technique