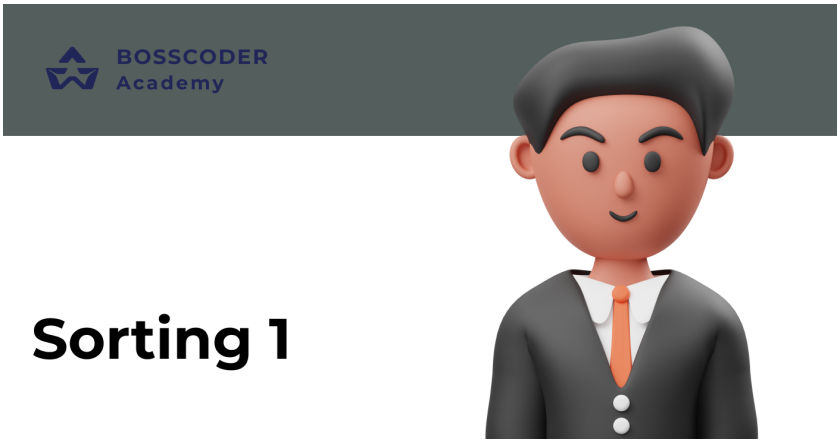
Sorting 1

Gyanu Mayank

17th February, 2023



Agenda

1. Definition of Sorting and different types of sorting
2. Problems using different sorting algorithms

Sorting

Sorting is any process of arranging items systematically. In computer science, sorting algorithms put elements of a list in a certain order.

list or array is sorted in ascending order if $\forall i$, such that $i \leq n-1$, $A[i] \leq A[i+1]$. Similarly, a list or array is sorted in descending order if $\forall i$, such that $i \leq n-1$, $A[i] \geq A[i+1]$.

Let's discuss a few sorting algorithms

1. **Selection sort algorithm**

the beginning. This way, the algorithm maintains two lists.

1. The sublist of already-sorted elements, which is filled from left to right
2. The sublist of the remaining unsorted elements that need to be sorted

In other words, this algorithm works by iterating over the list and swapping each element with the minimum (or maximum) element found in the unsorted list with that in the sorted list.

Implementation

```
#include <bits/stdc++.h>
using namespace std;

void swap(int *a, int *b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
}

void printArray(int array[], int size)
{
  for (int i = 0; i < size; i++)
  {
    cout << array[i] << " ";
  }
  cout << endl;
}
```

```
  for (int step = 0; step < size - 1; step+
  {
    int min_idx = step;
    for (int i = step + 1; i < size; i++)
    {

      if (array[i] < array[min_idx])
        min_idx = i;
    }

    swap(&array[min_idx], &array[step]);
  }
}

int main()
{
  int n;
  cin >> n;
  int arr[n];
  for (int i = 0; i < n; i++)
    cin >> arr[i];
  selectionSort(arr, n);
  cout << "Sorted array in Acsending Order:
  printArray(arr, n);
}
```

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

2. **Bubble sort algorithm**

This is another famous sorting algorithm also known as ‘sinking sort’. It works by comparing adjacent pairs of elements and swapping them if they are in the wrong order. This is

maximum/minimum element, and brings it over to the right side.

Implementation

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

void bubbleSort(int array[], int size)
{

  for (int step = 0; step < size; ++step)
  {

    for (int i = 0; i < size - step - 1; ++i
    {

      if (array[i] > array[i + 1])
      {

        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

void printArray(int array[], int size)
{
  for (int i = 0; i < size; ++i)
  {
    cout << "  " << array[i];
```

```
}

int main()
{
  int size;
  cin >> size;
  int data[size];
  for (int i = 0; i < size; i++)
    cin >> data[i];
  bubbleSort(data, size);

  cout << "Sorted Array in Ascending Order:
  printArray(data, size);
}
```

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

3. **Insertion Sort**

Insertion sort is another famous sorting algorithm and works the way you would naturally sort in real life. It iterates over the given list, figures out what the correct position of every element is, and inserts it there.

Implementation

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

void printArray(int array[], int size)
{
  for (int i = 0; i < size; i++)
  {
```

```
  cout << endl;
}

void insertionSort(int array[], int size)
{
  for (int step = 1; step < size; step++)
  {
    int key = array[step];
    int j = step - 1;
    while (key < array[j] && j >= 0)
    {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}

int main()
{
  int size;
  cin >> size;
  int data[size];
  for (int i = 0; i < size; i++)
    cin >> data[i];
  insertionSort(data, size);

  cout << "Sorted Array in Ascending Order:
  printArray(data, size);
}
```

4. **Counting Sort**

Counting sort is a sorting algorithm that sorts the elements of an array by counting the

auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Implementation

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

void printArray(int array[], int size)
{
  for (int i = 0; i < size; i++)
  {
    cout << array[i] << " ";
  }
  cout << endl;
}

void countSort(int array[], int size)
{
  int output[10];
  int count[10];
  int max = array[0];
  for (int i = 1; i < size; i++)
  {
    if (array[i] > max)
      max = array[i];
  }

  for (int i = 0; i <= max; ++i)
  {
    count[i] = 0;
  }
```

```
    count[array[i]]++;
  }

  for (int i = 1; i <= max; i++)
  {
    count[i] += count[i - 1];
  }

  for (int i = size - 1; i >= 0; i--)
  {
    output[count[array[i]] - 1] = array[i];
    count[array[i]]--;
  }

  for (int i = 0; i < size; i++)
  {
    array[i] = output[i];
  }
}

int main()
{
  int size;
  cin >> size;
  int data[size];
  for (int i = 0; i < size; i++)
    cin >> data[i];
  countSort(data, size);

  cout << "Sorted Array in Ascending Order:
  printArray(data, size);
}
```

5. **Bucket Sort**

Bucket Sort is a sorting algorithm that divides the unsorted array elements into several groups called buckets. Each bucket is then sorted by using any of the suitable sorting algorithms or recursively applying the same bucket algorithm. Finally, the sorted buckets are combined to form a final sorted array.

Bucket sort is used when:

1. Input is uniformly distributed over a range.
2. There are floating point values.

Implementation

```
#include <bits/stdc++.h>
using namespace std;
int findMax(int arr[], int n)
{
    int i,max=arr[0],cnt=0;
    for(i=1;i<n;i++)
    {
        if(arr[i]>max)
            max=arr[i];
    }
    while(max>0)
    {
        cnt++;
        max=max/10;
    }

    return cnt;
}

void bucketSort(int arr[],int *bucket[],int
```

```
    int c;
    c=findMax(arr,n);

    for(int m=0;m<c;m++)
    {
        for(i=0;i<10;i++)
            j[i]=0;
        for(i=0;i<n;i++)
        {
            k=(arr[i]/d)%10;
            bucket[k][j[k]]=arr[i];
            j[k]++;
        }

        l=0;
        for(i=0;i<10;i++)
        {
            for(k=0;k<j[i];k++)
            {
                arr[l]=bucket[i][k];
                l++;
            }
        }
        d*=10;
    }
}
int main()
{
    int n,*arr,i;
    int *bucket[10];
    cin>>n;
    arr=new int[n+1];
    for(i=0;i<10;i++)
        bucket[i]=new int[n];
    for(i=0;i<n;i++)
```

```
    cout<<"Sorted array : ";
    for(i=0;i<n;i++)
        cout<<arr[i]<<" ";
    return 0;
}
```

Time Complexity: O(nlogn)
Space Complexity: O(n * k) [The space complexity is quite high when compared to other sorting algorithms since the use of buckets.]

6. **Merge Sort**

Merge sort is one of the most prominent divide-and-conquer sorting algorithms in the modern era. It can be used to sort the values in any traversable data structure such as a list. Merge sort works by splitting the input list into two halves, repeating the process on those halves, and finally merging the two sorted halves together. The algorithm first moves from top to bottom, dividing the list into smaller and smaller parts until only the separate elements remain. From there, it moves back up, ensuring that the merging lists are sorted.

Implementation

```
#include <iostream>
using namespace std;
void merge(int arr[], int p, int q, int r)
```

```
int n1 = q - p + 1;
int n2 = r - q;

int L[n1], M[n2];

for (int i = 0; i < n1; i++)
  L[i] = arr[p + i];
for (int j = 0; j < n2; j++)
  M[j] = arr[q + 1 + j];

int i, j, k;
i = 0;
j = 0;
k = p;

while (i < n1 && j < n2)
{
  if (L[i] <= M[j])
  {
    arr[k] = L[i];
    i++;
  }
  else
  {
    arr[k] = M[j];
    j++;
  }
  k++;
}

while (i < n1)
{
  arr[k] = L[i];
  i++;
  k++;
```

```
  while (j < n2)
  {
    arr[k] = M[j];
    j++;
    k++;
  }
}

void mergeSort(int arr[], int l, int r)
{
  if (l < r)
  {
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
  }
}

void printArray(int arr[], int size)
{
  for (int i = 0; i < size; i++)
    cout << arr[i] << " ";
  cout << endl;
}

int main()
{
  int size;
  cin >> size;
  int arr[size];
  for (int i = 0; i < size; i++)
    cin >> arr[i];
  mergeSort(arr, 0, size - 1);
```

```
  return 0;
}
```

Time Complexity: O(nlogn)
Space Complexity: O(n)

Problems

1. **Kth Largest Element in Array**
   Given an integer array `nums` and an integer `k`, return *the* k^{th} the *largest element in the array*. Note that it is the k^{th} the largest element in the sorted order, not the k^{th} distinct element.
   Example
   Input: nums = [3,2,1,5,6,4], k = 2
   Output: 5
   Approach Using Sorting
   Sort the given array and return the element at index K-1 in the sorted array.
   Implementation

```
#include <bits/stdc++.h>
using namespace std;
int kthLargest(int arr[], int N, int K)
{

  sort(arr, arr + N);
  return arr[N - K ];
}

int main()
{
  int size;
```

```
  cin >> target;
  int arr[size];
  for (int i = 0; i < size; i++)
    cin >> arr[i];
  cout << kthLargest(arr, size, target);
  return 0;
}
```

We can use various sorting algorithms to sort the array. In the following Implementation, we have used the inbuilt sort function.
Time Complexity: O(N log N)
Space Complexity: O(1)

Optimised Approach

To find the Kth maximum element in an array, insert the elements into the priority queue until the size of it is less than K, and then compare the remaining elements with the root of the priority queue if the element is greater than the root then remove the root and insert this element into the priority queue and finally return the root of the priority queue.

Implementation

```
#include <bits/stdc++.h>
using namespace std;
int kthLargest(int arr[], int N, int K)
{

  priority_queue<int, vector<int>, greater<
  for (int i = 0; i < N; ++i)
  {
    pq.push(arr[i]);
```

```
  }
  return pq.top();
}

int main()
{
  int size;
  cin >> size;
  int target;
  cin >> target;
  int arr[size];
  for (int i = 0; i < size; i++)
    cin >> arr[i];
  cout << kthLargest(arr, size, target);
  return 0;
}
```

Time complexity: O(K log K + (N – K) log K)
Space Complexity: O(K)

2. **Choose array elements such that difference between maximum and minimum<=k**

Select elements such that max-min<=k find the maximum size possible.

Example

Input: n=7 arr[]={1,3,8,5,3,2,1},k=2
Output: 5
Explanation: 1,1,2,3,3 total elements 5

Approach 1:Sorting

We can sort the array and use two pointers such that increment j till the difference is less than k if it exceeds then increment i and store

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int n;
  cin>>n;
  int k;
  cin>>k;
  int a[n];
  for(int i=0;i<n;i++) cin>>a[i];
    sort(a,a+n);
  int i=0,j=0;
  int cnt=0;
  int ma=0;
   while(j<=n){
     if(a[j]-a[i]<=k){
      cnt++;

     }
     else{
     ma=max(ma,cnt);
     cnt=0;
     i++;
     }
     j++;
   }
   cout<<ma<<endl;
}
```

Approach: Using Count Sort

We can use count sort to solve this problem as we will store the frequency of every element in

will store the max value. The only drawback of counting sort is we should know the range beforehand.

Implementation

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int n;
  cin>>n;
  int k;
  cin>>k;
  int a[n];
  for(int i=0;i<n;i++) cin>>a[i];
    sort(a,a+n);
    vector<int>freq(n);

    for(int i=0;i<n;++i){
  freq[a[i]]++;
    }
    vector<int>pre(n);
    pre[0]=freq[0];
    for(int i=1;i<n;i++){
      pre[i]=pre[i-1]+freq[i];
    }
int i=0,j=0,ma=0,cnt=0;
   while(j<=n){
     if(a[j]-a[i]<=k){
      cnt=pre[a[j]]-pre[a[i]-1];
     }
     else{
     ma=max(ma,cnt);
```

```
        }
        j++;
    }
     cout<<ma;
}
```

Time Complexity: O(range)
Space Complexity: O(n)

View All blogs