

## Week-4:

# Sorting Algorithms

Q1. Implement the Merge Sort algorithm as a function that takes an array of integers and sorts it.

Code:

```
#include<iostream>
using namespace std;
void merge(int *arr,int s,int e)
{
    int mid=s+(e-s)/2;
    int l1=mid-s+1;
    int l2=e-mid;
    int *first=new int[l1];
    int *second=new int[l2];
    int mainarrayindex=s;
    for(int i=0;i<l1;i++)
    {
        first[i]=arr[mainarrayindex++];
    }
    mainarrayindex=mid+1;
    for(int i=0;i<l2;i++)
    {
        second[i]=arr[mainarrayindex++];
    }
    int index1=0;
    int index2=0;
    mainarrayindex=s;
    while(index1<l1 &&index2<l2)
    {
        if(first[index1]<second[index2])
        {
            arr[mainarrayindex++]=first[index1++];
        }
        else{
            arr[mainarrayindex++]=second[index2++];
        }
    }
    while(index1<l1)
    {
        arr[mainarrayindex++]=first[index1++];
    }
    while(index2<l2)
    {
        arr[mainarrayindex++]=second[index2++];
    }
    delete []first;
    delete []second;
}
void mergesort(int *arr,int s,int e)
```

```

{
    if(s >= e) return ;
    int mid = s + (e - s) / 2;

    mergesort(arr, s, mid);
    mergesort(arr, mid + 1, e);
    merge(arr, s, e);
}
int main()
{
    int n;
    cout << "enter the size of array:";
    cin >> n;
    int arr[n];
    cout << "\n enter the " << n << " elements:";
    for(int i = 0 ; i < n ; i++) {
        cin >> arr[i];
    }
    mergesort(arr, 0, n - 1);
    for(int i = 0 ; i < n ; i++)
    {
        cout << arr[i] << " ";
    }
}

```

Output:

```

enter the size of array:7

enter the 7 elements:4 5 6 4 3 2 1
1 2 3 4 4 5 6

```

Q2. Modify the Merge Sort implementation to sort an array of strings in lexicographical order. Write a function that takes an array of strings as input and sorts it using Merge Sort.

Code:

```

#include <bits/stdc++.h>
using namespace std;
void merge(vector<string> & arr, int left, int right, int mid){
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<string> l1(n1), l2(n2);
    for(int i = 0 ; i < n1 ; i++) {
        l1[i] = arr[left + i];
    }
    for(int i = 0 ; i < n2 ; i++) {
        l2[i] = arr[mid + i + 1];
    }
}

```

```

int i = 0, j = 0, k = left;
while(i < n1 && j < n2) {
    if(l1[i] < l2[j]) {
        arr[k++] = l1[i++];
    }
    else {
        arr[k++] = l2[j++];
    }
}
while(i < n1) {
    arr[k++] = l1[i++];
}
while(j < n2) {
    arr[k++] = l2[j++];
}
}
void mergesort(vector<string> & arr, int s, int e) {
    if(s >= e)
        return;
    int mid = s + (e - s) / 2;
    mergesort(arr, s, mid);
    mergesort(arr, mid + 1, e);
    merge(arr, s, e, mid);
}
int main() {
    vector<string> arr = {"banana", "apple", "orange",
"grape", "pineapple"};
    int n = arr.size();
    mergesort(arr, 0, n - 1);
    cout << "Sorted array:" << endl;
    for (const string& s : arr) {
        cout << s << endl;
    }
}

```

Output:

```

Sorted array:
apple
banana
grape
orange
pineapple

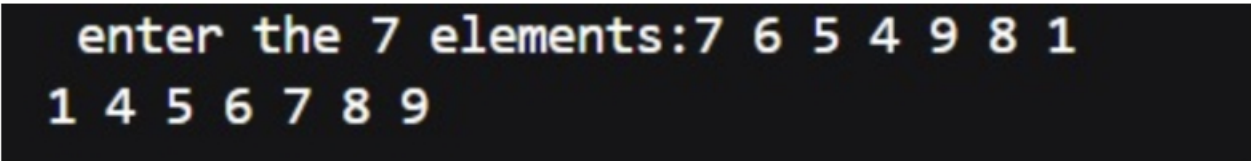
```

Q3. Implement the Quick Sort algorithm as a function that takes an array of integers and sorts it.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int partition(vector<int> & arr,int s, int e)
{
    int pivot=arr[e];
    int i = s - 1;
    for(int j = s ; j <= e - 1 ; j++) {
        if(arr[j]<=pivot) {
            i++;
            swap(arr[i],arr[j]);
        }
    }
    swap(arr[i+1],arr[e]);
    return (i+1);
}
void quicksort(vector<int> & arr,int s,int e)
{
    if( s < e ) {
        int p=partition(arr,s,e);
        quicksort(arr,s,p-1);
        quicksort(arr,p+1,e);
    }
}
int main()
{
    int n;
    cout<<"enter the size of array:";
    cin>>n;
    vector<int>arr(n);
    cout<<" \n enter the "<<n<<" elements:";
    for(int i = 0 ; i < n ; i++) {
        cin>>arr[i];
    }
    quicksort(arr,0,n-1);
    for(int i=0;i<n;i++)
    {
        cout<<arr[i]<<" ";
    }cout<<endl;
}
```

Output:



```
enter the 7 elements:7 6 5 4 9 8 1
1 4 5 6 7 8 9
```

Q4. Modify the Quick Sort implementation to randomly select the pivot element instead of always choosing the last element as the pivot. Implement a function that chooses a random pivot for each partitioning step. Evaluate the performance of the randomized Quick Sort algorithm and compare it with the standard Quick Sort.

- Print the partition details
- Execution time in terms of microseconds on a relatively large input array generated using random number generator.

Code:

```
#include <bits/stdc++.h>
using namespace std;
int partition(vector<int> & arr,int s, int e)
{
    int pivot=arr[e];
    int i = s - 1;
    for(int j = s ; j <= e - 1 ; j++) {
        if(arr[j]<=pivot) {
            i++;
            swap(arr[i],arr[j]);
        }
    }
    swap(arr[i+1],arr[e]);
    return (i+1);
}
int partition_r(vector<int> & arr, int low, int high)
{
    srand(time(NULL));

    int random = low + rand() % (high - low);

    swap(arr[random], arr[high]);
    return partition(arr, low, high);
}
void quicksort(vector<int> & arr,int s,int e)
{
    if( s < e) {
        int p=partition_r(arr,s,e);
        quicksort(arr,s,p-1);
        quicksort(arr,p+1,e);
    }
}
int main()
{
    int n;
    cout<<"enter the size of array:";
    cin>>n;
    vector<int>arr(n);
    cout<<" \n enter the "<<n<<" elements:";
    for(int i = 0 ; i<n ; i++) {
        cin>>arr[i];
    }
}
```

```

quicksort(arr,0,n-1);
for(int i=0;i<n;i++)
{
    cout<<arr[i]<<" ";
}cout<<endl;
}

```

Output:

```

enter the 7 elements:6 5 4 7 8 9 1
1 4 5 6 7 8 9

```

Q5. Modify the Quick Sort implementation to handle arrays with duplicate elements efficiently using three-way partitioning. Implement a function that partitions the array into three parts: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot.

Code:

```

#include <bits/stdc++.h>
using namespace std;
int partitionRight(int arr[], int p, int r) {
    int x = arr[r];
    int i = p - 1;
    for(int j = p; j < r; j++) {
        if(arr[j] < x) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[r]);
    return i + 1;
}
int partitionLeft(int arr[], int p, int r) {
    int x = arr[p];
    int i = r + 1;
    for(int j = r; j > p; j--) {
        if(arr[j] > x) {
            i--;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i - 1], arr[p]);
    return i - 1;
}
void quickSort(int arr[], int p, int r) {
    if(p < r) {
        int q = partitionRight(arr, p, r);
        int s = partitionLeft(arr, p, r);
        quickSort(arr, p, q - 1);
        quickSort(arr, s + 1, r);
    }
}

```

```

}
int main(){
    int arr[] = {5, 2, 4, 7, 1, 3, 5, 6, 3, 2, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    for(int i = 0; i < n; i++){
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}

```

Output:

1 2 2 3 3 4 5 5 6 6 7

Q6. Use Quick sort to develop an algorithm to find median of an array of integers. The median of a set of numbers is the middle value when the numbers are arranged in ascending or descending order.

Code:

```

#include <iostream>
using namespace std;
int partition(int arr[], int p, int r)
{
    int x = arr[r];
    int i = p - 1;
    for (int j = p; j < r; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[r]);
    return i + 1;
}
void quickSortMedian(int arr[], int p, int r, int index, bool found = false)
{
    if (p < r) {
        int q = partition(arr, p, r);
        int n = r - p + 1;
        if (!found && q == index) {
            cout << "Median: " << arr[q] << endl;
            found = true;
        }
        quickSortMedian(arr, q + 1, r, index, found);
        quickSortMedian(arr, p, q - 1, index, found);
    }
}

```

```

    }
}
int main()
{
    int arr[] = { 1, 2, 4, 8, 9 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSortMedian(arr, 0, n - 1, n/2);
    return 0;
}

```

Output:

**Median: 4**

Q7. Use the developed algorithm for Problem statement 6 to implement quick sort to show that the partition always result in best case.

Code:

```

#include <iostream>
using namespace std;
int partition(int arr[], int p, int r)
{
    int x = arr[r];
    int i = p - 1;
    for (int j = p; j < r; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[r]);
    return i + 1;
}
void quickSortMedian(int arr[], int p, int r, int index, bool found = false)
{
    if (p < r) {
        int q = partition(arr, p, r);
        int n = r - p + 1;
        if (!found && q == index) {
            cout << "Median: " << arr[q] << endl;
            found = true;
        }
        quickSortMedian(arr, q + 1, r, index, found);
        quickSortMedian(arr, p, q - 1, index, found);
    }
}

```



```

}
int main()
{
    int arr[] = {1, 2, 4, 8, 9};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSortMedian(arr, 0, n - 1, n/2);
    return 0;
}

```

Output:

**Median: 4**

Q8. Write a function to find the median of an array of integers using the Median of Medians algorithm. Employ this median of medians finding algorithm to revise the quick sort implementation. Compare the performance of the following quick sort

- Choose first element as pivot
- Choose last element as pivot
- Choose random element as pivot
- Choose median as pivot

Code:

```

#include<iostream>
#include<climits>
#include<chrono>
#include<algorithm>
using namespace std;
int partition(int arr[], int p, int r) {
    int i = p - 1;
    int pivot = arr[r];
    swap(arr[pivot], arr[r]);
    pivot = arr[r];
    for(int j = p; j < r; j++) {
        if(arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[r]);
    return i + 1;
}
int partitionFirst(int arr[], int p, int r) {
    int i = p - 1;
    int pivot = arr[p];
    for(int j = p; j < r; j++) {
        if(arr[j] <= pivot) {

```

```

        i++;
        swap(arr[i], arr[j]);
    }
}
swap(arr[i + 1], arr[p]);
return i + 1;
}
int partitionRandom(int arr[], int p, int r) {
    int i = p - 1;
    int pivot = rand() % (r - p + 1) + p;
    swap(arr[pivot], arr[r]);
    pivot = arr[r];
    for(int j = p; j < r; j++) {
        if(arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[r]);
    return i + 1;
}

void quickSort(int arr[], int p, int r) {
    if(p < r) {
        int q = partition(arr, p, r);
        quickSort(arr, p, q - 1);
        quickSort(arr, q + 1, r);
    }
}

void quickSortFirst(int arr[], int p, int r) {
    if(p < r) {
        int q = partitionFirst(arr, p, r);
        quickSortFirst(arr, p, q - 1);
        quickSortFirst(arr, q + 1, r);
    }
}

void quickSortRandom(int arr[], int p, int r) {
    if(p < r) {
        int q = partitionRandom(arr, p, r);
        quickSortRandom(arr, p, q - 1);
        quickSortRandom(arr, q + 1, r);
    }
}

int findMedian(int arr[], int p, int r) {
    int n = r - p + 1;
    int numMedians = (n + 4) / 5;
    int *medians = new int[numMedians];
    for (int i = 0; i < numMedians; i++) {
        int start = p + i * 5;
        int end = min(p + (i + 1) * 5 - 1, r);
        sort(arr + start, arr + end + 1);
        medians[i] = arr[(start + end) / 2];
    }
}

```

```

    }
    if (numMedians == 1)
        { delete[] medians;
          return arr[p];
        } else {
            return findMedian(medians, 0, numMedians - 1);
        }
    }
int medianOfMedians(int arr[], int p, int r) {
    if (p == r) return arr[p];
    int median = findMedian(arr, p, r);
    int pivotIndex = partition(arr, p, r);
    if (arr[pivotIndex] == median) {
        return pivotIndex;
    } else if (arr[pivotIndex] < median) {
        return medianOfMedians(arr, pivotIndex + 1,
r); } else {
        return medianOfMedians(arr, p, pivotIndex - 1);
    }
}
int partitionMedian(int arr[], int p, int r)
{ int x = arr[medianOfMedians(arr, p, r)];
  int i = p - 1;
  for(int j = p; j < r; j++) {
      if(arr[j] <= x) {
          i++;
          swap(arr[i], arr[j]);
      }
  }
  swap(arr[i + 1], arr[r]);
  return i + 1;
}
void quickSortMedian(int arr[], int p, int r) {
    if(p < r) {
        int q = partitionMedian(arr, p, r);
        quickSortMedian(arr, p, q - 1);
        quickSortMedian(arr, q + 1, r);
    }
}
void populateArray(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        arr[i] = rand() % 1000;
    }
}
int main() {
    int n = 1000;
    int arr[n];
    populateArray(arr, n);
    auto start = chrono::high_resolution_clock::now();
    quickSort(arr, 0, n - 1);
    auto end = chrono::high_resolution_clock::now();

```

```

        cout << "Time taken for quickSort: " <<
chrono::duration_cast<chrono::milliseconds>(end -
start).count() << "ms" << endl;
        populateArray(arr, n);
        start = chrono::high_resolution_clock::now();
        quickSortFirst(arr, 0, n - 1);
        end = chrono::high_resolution_clock::now();
        cout << "Time taken for quickSortFirst: " <<
chrono::duration_cast<chrono::milliseconds>(end -
start).count() << "ms" << endl;
        populateArray(arr, n);
        start = chrono::high_resolution_clock::now();
        quickSortRandom(arr, 0, n - 1);
        end = chrono::high_resolution_clock::now();
        cout << "Time taken for quickSortRandom: " <<
chrono::duration_cast<chrono::milliseconds>(end -
start).count() << "ms" << endl;
        populateArray(arr, n);
        start = chrono::high_resolution_clock::now();
        quickSortMedian(arr, 0, n - 1);
        end = chrono::high_resolution_clock::now();
        cout << "Time taken for quickSortMedian: " <<
chrono::duration_cast<chrono::milliseconds>(end -
start).count() << "ms" << endl;
        return 0;
    }

```

Output:

```

Time taken for quickSort: 0ms
Time taken for quickSortFirst: 0ms
Time taken for quickSortRandom: 0ms
Time taken for quickSortMedian: 22ms

```