Sorting: worst to best

Final Report

PREPARED BY

Dev Kalra (170001018)

Nitesh Ahirwar (170004019)

Acknowledgement

We are highly indebted to Dr Kapil Ahuja, Associate Professor, IIT Indore for his guidance and constant supervision as well as for providing an opportunity to work on Algorithm analysis in real life. We have gained a lot of knowledge through this project, and have had our first experience on a research project.

Overview

Sorting basically means putting things into order, depending upon some parameters. Sorting has lot of applications in almost every field from sorting a shopping list to Commercial computing to searching for some data, which requires sorting, numerical computations and so many other algorithms requires sorting of data

Here we implemented ,analyze and tried to improve the quick sort. So that its worst case complexity is less than O(n^2). Also as quick sort only sorts numbers we will try to add to its functionality. So that it can sort an array of characters and strings.

Quicksort Algorithm

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.

4. Pick median as pivot.

Divide and Conquer

In quick sort sorting is based on a divide and conquer approach. Always pick first element as pivot.

1. First a pivot point is selected i.e, array is divided into two sub array.
2. The first sub array will contain the elements smaller than the pivot point, the second one contains the larger one.
3. Then the subarray are passed in a recursive manner.

C++ Implementation for Quicksort Algorithm

```
#include <iostream>

using namespace std;

void quick_sort(int[],int,int);
int partition(int[],int,int);
int main()
{
    int a[50],n,i;
    cout<<"How many elements?";
    cin>>n;
    cout<<"\nEnter array elements:";

    for(i=0;i<n;i++)
        cin>>a[i];

    quick_sort(a,0,n-1);
    cout<<"\nArray after sorting:";

    for(i=0;i<n;i++)
        cout<<a[i]<<" ";
```

```
    return 0;
}
void quick_sort(int a[],int l,int u)
{
    int j;
    if(l<u)
    {
        j=partition(a,l,u);
        quick_sort(a,l,j-1);
        quick_sort(a,j+1,u);
    }
}
int partition(int a[],int l,int u)
{
    int v,i,j,temp;
    v=a[l];
    i=l;
    j=u+1;

    do
    {
        do
            i++;

        while(a[i]<v&&i<=u);

        do
            j--;
        while(v<a[j]);

        if(i<j)
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);

    a[l]=a[j];
    a[j]=v;

    return(j);
}
```

Complexity Analysis Quicksort Algorithm

T(n) = T(k) + T(n-k-1) + O(n)

where k, n-k-1 are the sizes of the partitions and O(n) is the the time taken by the partition function

Best Case: k = n/2

T(n) = 2T(n/2) + O(n)

by Master's Theorem:

T(n) = O(nlogn)

Worst Case: k = 0

T(n) = T(n-1) + O(n)

T(n) = O(n^2)

Average Case: For some k

Showing that the average-case running time is also $O(n\log n)$ takes some pretty involved mathematics, and so we won't go there.

The only drawback in using this algorithm is that it takes O(n^2) in the worst case

Improvise Quicksort

Approach

This algorithm is implemented using Divide and Conquer algorithm design approach.We have merged insertion and quick sort in our algorithm.

Implementation

- As we have merged two sorting algorithm,so when the total number of inputs are less than or equal to twenty we use insertion algorithm as it works better than quick sort for small number of inputs and for other we use modified quick sort.
- We have used variables(i and j) and according to them we check if there is a Ascending or Descending pattern in our input.
- There are two different sorting functions each for one pattern(either ascending or descending).

C++ Implementation for the improvised Quicksort Algorithm

```
#include<iostream>
#include<chrono>

using namespace std;
using namespace std::chrono;

int k;
void printarray(int a[]){
	for(int i=0;i<k;i++)
		cout<<a[i]<<" ";
	cout<<endl;
}
void sortDesc(int low,int high,int a[]){
	int i=low;
	while(i<(low+high)/2+1){
		int temp = a[i];
```

```
                a[i] = a[high-i];
                a[high-i] = temp;
                i++;
        }
}
int checkDesc(int low,int high,int a[]){
        int i = 0;
        int k = low+1;
        while(k<=high){
                if(a[k-1]>=a[k])
                        i++;
                k++;
        }
        if(i==(high-low)){
                sortDesc(low,high,a);
                return 1;
        }
        return 0;
}
int checkAsc(int low,int high,int a[]){
        int i = 0;
        int k = low+1;
        while(k<=high){
                if(a[k-1]<=a[k])
                        i++;
                k++;
        }
        if(i==(high-low))
                return 1;
        return 0;
}

int partition_wmb(int low,int high,int a[]){
        int key,i,j,temp;
        key=a[low];
        i=low;
        j=high+1;
        while(i<=j){
                do{
                        i++;
                }
                while(key>=a[i]&&i<high+1);

                do{
                        j--;
                }
                while(key<a[j]);

                if(i<j){
```

```
                        temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
                }
        }
        if((j==high)&&(i==high+1)){
                if(checkDesc(low,high,a))
                        return -1;
        }
        else if((i==1)&&(j==0)){
                if(checkAsc(low,high,a))
                        return -1;
        }
        temp=a[low];
        a[low]=a[j];
        a[j]=temp;
        return j;
}
void insertionSort(int low,int high,int arr[]){
        int i, key, j;
        for (i = low; i <=high; i++) {
                key = arr[i];
                j = i - 1;
                while (j >= low && arr[j] > key) {
                        arr[j + 1] = arr[j];
                        j = j - 1;
                }
        arr[j + 1] = key;
        }
}
void quicksort_wmb(int low,int high,int a[])
{
        if(checkAsc(low,high,a))
                return;
        //if(checkDesc(low,high,a))
        //      return;
        if((high-low)<=20){
                insertionSort(low,high,a);
                return;
        }
        int j;
        if(low<high){
                j=partition_wmb(low,high,a);
                if(j == -1)
                        return;
                //cout<<"ho rhi he"<<endl;
                quicksort_wmb(low,j-1,a);
                quicksort_wmb(j+1,high,a);
        }
```

```
}
int main(){
	int n;
	cin>>n;
	int a[n];
	for(int i=0;i<n;i++)
		cin>>a[i];
	k=n;
	auto start = high_resolution_clock::now();
	quicksort_wmb(0,n-1,a);
	auto stop =  high_resolution_clock::now();
	auto duration = duration_cast<microseconds>(stop - start);
	cout<<duration.count()<<endl;
	return 0;
}
```

Procedure

1. We select the leftmost element as a pivot point.
2. Then we find the correct position of this pivot using variables i and j in loop.
3. Depending upon these variables we check if the given array is already in ascending or descending order.
4. If in either of them we return -1 and stop the quick sort there and return the sorted array.
5. for array to be in descending order we have implemented sortDesc function.
6. Other implementation is same as the quicksort.

Disadvantages

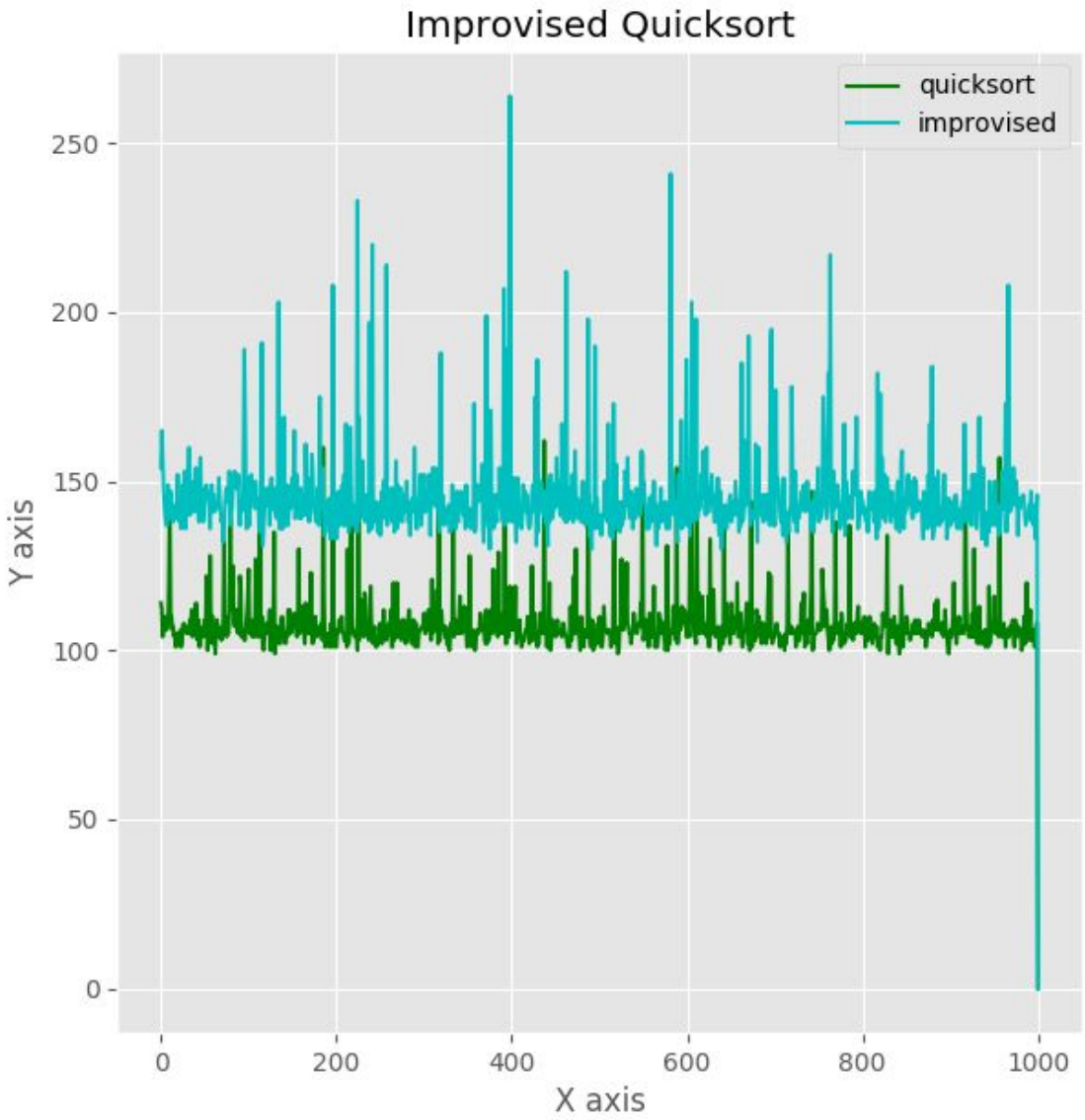
- To improve efficiency for the worst case we have to compromise a little for the time efficiency of a random or an average case.

Advantages

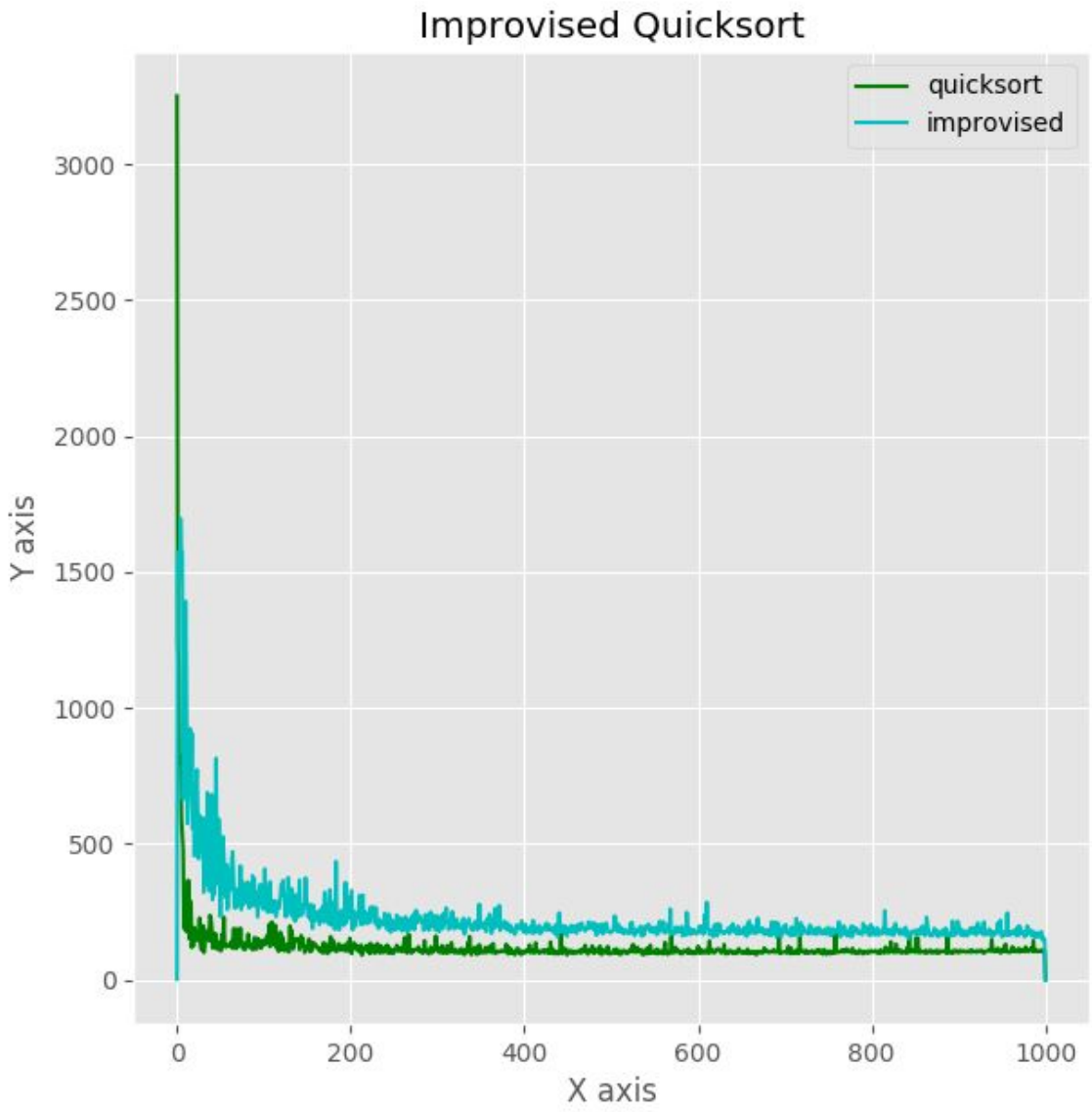
- This algorithm improves the time complexity greatly in the worst case and can be seen in the following plottings.

Some Test Cases

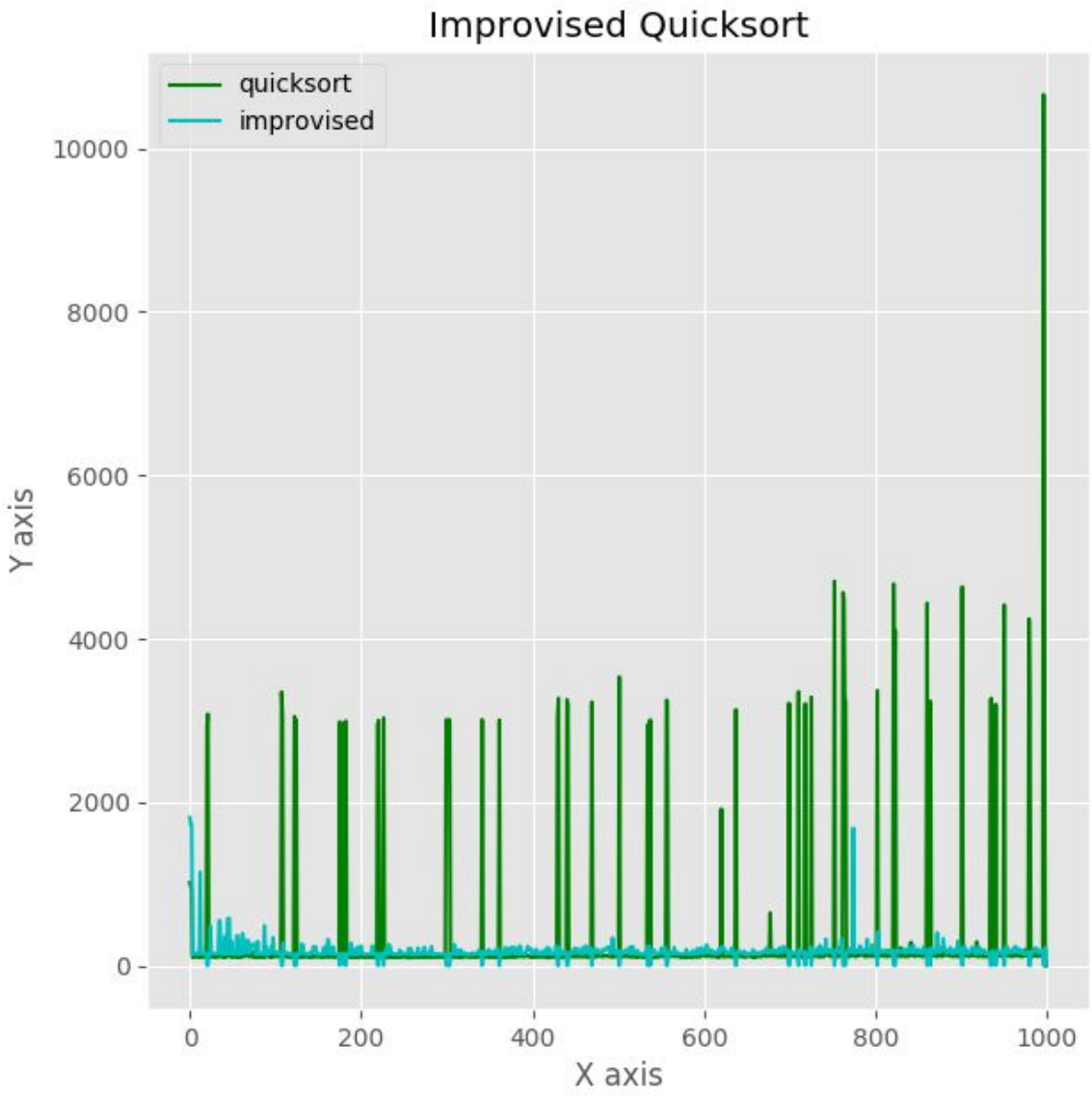
Here the x axis = test cases , y axis = time in microseconds



the average case as we said we have to compromise a little for the average case



improvised one is performing better in the worst case and is almost performing similar to quick sort in average case



the random case have some worst case and some average case

Conclusion

- For the worst cases improvise quicksort works far better than quick sort.
- For the average cases they approximately give same efficiency.

- If there is pattern present in our input then improvised quicksort perform exceptionally well.