Advance Algorithms

LAB 01 Roll No.: CE092

Aim: To implement Randomized quicksort and analyse its working with different types of inputs.

Code:

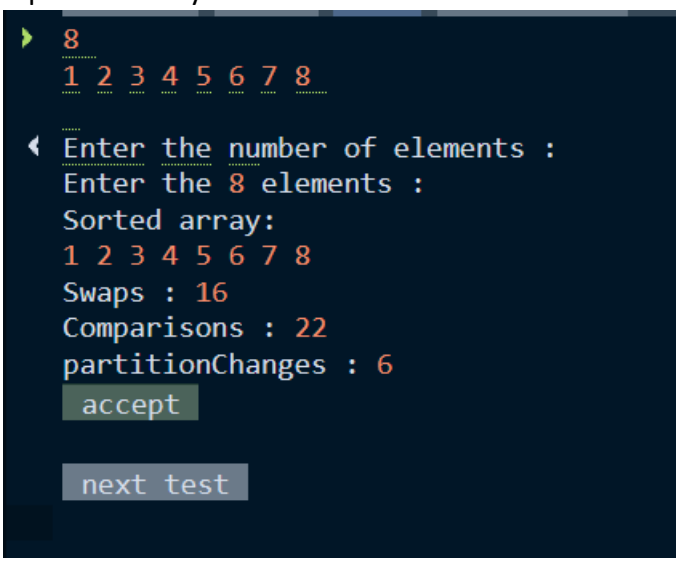
```
/*
* @Author: nevil
* @Date:   2020-07-06 16:52:29
* @Last Modified by:   nevil
* @Last Modified time: 2020-07-18 04:24:26
*/
#include<bits/stdc++.h>
using namespace std;
int partitionChanges = 0;
int swaps = 0;
int comparisons = 0;

int partition(int arr[], int low, int high)
{
	int pivot = arr[high];
	int i = (low - 1);

	for (int j = low; j <= high - 1; j++) {

		comparisons++;
		if (arr[j] <= pivot) {

			i++;
```

```
                swap(arr[i], arr[j]);
                swaps++;
            }
        }
        swap(arr[i + 1], arr[high]);
        swaps++;
        return (i + 1);
}
int partition_r(int arr[], int low, int high)
{
        srand(time(NULL));
        int random = low + rand() % (high - low);

        swap(arr[random], arr[high]);
        swaps++;
        partitionChanges++;

        return partition(arr, low, high);
}
void quickSort(int arr[], int low, int high)
{
        if (low < high) {
            comparisons++;
            int pi = partition_r(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
}
void printArray(int arr[], int size)
```

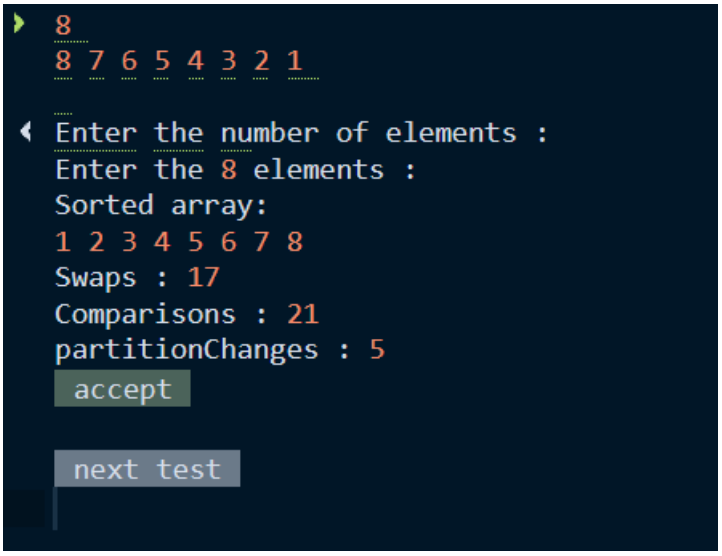
```
{
	int i;
	for (i = 0; i < size; i++)
		printf("%d ", arr[i]);
	printf("\n");
}
int main()
{
	int n ;
	cout << "Enter the number of elements : " << endl;
	cin >> n;
	int arr[n];

	cout << "Enter the " << n << " elements : " << endl;

	for (auto i = 0 ; i < n ; i++)
	{
		cin >> arr[i];
	}

	quickSort(arr, 0, n - 1);
	printf("Sorted array: \n");
	printArray(arr, n);

	cout << "Swaps : " << swaps << endl;
	cout << "Comparisons : " << comparisons << endl;
	cout << "partitionChanges : " << partitionChanges << endl;
	return 0;
}
```

Output:

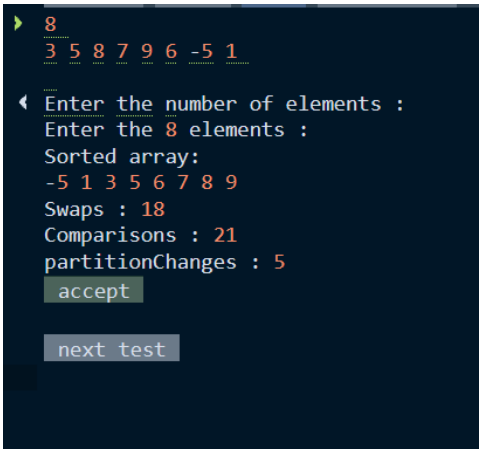
1. Input is already sorted



2. Input is reverse sorted



3. Input is in random order



Performance of this code on larger size of input array.

This performance is measured for larger n with rand () function which generates elements in random order and stores it into array on which this randomized quicksort is performed.

Comparison table:

Performance of Randomized Quicksort

Input Size	Sorted Input # of comparisons	Reversed Sorted Input # of comparisons	Random Input #of comparisons
100	598	580	584
1000	9810	10816	10626
10000	151883	152974	163644

Performance of Normal Quicksort

Input Size	Sorted Input # of comparisons	Reversed Sorted Input # of comparisons	Random Input # of comparisons
100	4950	4950	647
1000	499500	499500	10421
10000	4999500	4999500	174072

Conclusion:

We can see from the above table that for various input size, # of comparisons are very less in randomized quicksort as compared to normal quicksort which takes last element as pivot value. Which indirectly proves the probabilistic analysis of randomized quicksort true.

Thus, we can conclude that using random pivoting we can improve the expected or average time complexity to O (N*log N). The worst-case complexity is still O (N^2) but the probability of worst-case occurrence is very less.