Experiment No: 5

Implement a program for randomized version of quick sort and compare its performance with the normal version of quick sort using steps count on various inputs (1000 to 5000) ofrandom nature, ascending order and descending order sorted data.Also, draw a comparative chart of number of input versus steps executed/time taken for each cases (random, ascending, and descending).

Date:

Competency and Practical Skills: Algorithmic thinking, Programming Skills, Performance analysis

Relevant CO: CO1, CO2

Objectives: (a) Improve the performance of quick sort in worst case.
(b) Compare the performance of both the version of quick sort on various inputs

Equipment/Instruments: Computer System, Any C language editor

Theory:

Steps to implement randomized version of quick sort are as below:

```
RANDOMIZED-QUICKSORT(A, low, high)
        if (low< high) {
        pivot= RANDOMIZED_PARTITION(A, low, high);
        RANDOMIZED-QUICKSORT(A, low, pivot);
        RANDOMIZED-QUICKSORT(A, pivot+1,high);
}
RANDOMIZED_PARTITION (A,low,high) {
    pos = Random(low, high)
    pivot = A[pos]
    swap(pivot, a[low])
    left = low
    right = high
    while ( left <right ) {
         /* Move left while item < pivot */
        while( A[left] <= pivot ) left++;
        /* Move right while item > pivot */
        while( A[right] > pivot ) right--;
        if ( left < right )
                swap(A[left],A[right]);}
 /* right is final position for the pivot */
swap(A[right], pivot);
return right; }
```

Implement a function of randomized version of quick sort as per above instructions and use basic version of quick sort (that selects first element as pivot element). Compare the steps count of both the functions on various inputs ranging from 1000 to 5000 for each case (random, ascending, and descending).

Observations:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void quick_sort(int a[], int si, int ei);
void randomized_quicksort(int a[], int low, int high);
int randomized_partition(int arr[], int low, int high);
void swap(int* x, int* y);
void generate_random_data(int arr[], int n);
void generate_ascending_data(int arr[], int n);
void generate_descending_data(int arr[], int n);

int quick_step = 0;
int randomized_step =0;

void swap(int* x, int* y) {
   int temp = *x;
   *x = *y;
   *y = temp;
}

void quick_sort(int a[], int si, int ei) {
   if (si < ei) {
      int p = partition(a, si, ei);
      quick_sort(a, si, p - 1);
      quick_sort(a, p + 1, ei);
   }
}
int partition(int arr[],int si,int ei){
   int pivot = arr[ei];
   int i = (si - 1);
   quick_step+=2;
   for (int j = si; j <= ei - 1; j++) {
      quick_step++;
      if (arr[j] < pivot) {
         i++;
         swap(&arr[i], &arr[j]);
         quick_step+=2;
      }
      quick_step++;
   }

   swap(&arr[i + 1], &arr[ei]);
   quick_step++;
   return (i + 1);
}

int randomized_partition(int arr[], int low, int high) {
   int pivot_index = low + rand() % (high - low + 1);
   swap(&arr[pivot_index], &arr[high]);
```

```
    int pivot = arr[high];
    int i = low - 1;
    randomized_step+=4;
    for (int j = low; j < high; j++) {
        randomized_step++;
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
            randomized_step+=2;
        }
        randomized_step++;
    }
    swap(&arr[i + 1], &arr[high]);
    randomized_step++;
    return (i + 1);
}

void randomized_quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pi = randomized_partition(arr, low, high);
        randomized_quicksort(arr, low, pi - 1);
        randomized_quicksort(arr, pi + 1, high);
    }
}

void generate_random_data(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000;
    }
}

void generate_ascending_data(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }
}

void generate_descending_data(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = n - i;
    }
}

int main() {
    printf("Input Size\tStandard Steps\tRandomized Steps\n");
    for(int i=1000;i<=5000;i=i+1000)
    {
        int *arr = (int *)malloc(i * sizeof(int));
        int *arr_copy = (int *)malloc(i * sizeof(int));

        // Random data
        generate_random_data(arr, i);
```

```
        quick_step = 0;
        quick_sort(arr, 0, i - 1);
        printf("%d\t\t%d\t\t", i, quick_step);

        randomized_step = 0;
        for (int j = 0; j < i; j++) {arr_copy[j] = arr[j];}
        randomized_quicksort(arr_copy, 0, i - 1);
        printf("%d\n", randomized_step);

        // Ascending data
        generate_ascending_data(arr, i);
        quick_step = 0;
        quick_sort(arr, 0, i - 1);
        printf("%d\t\t%d\t\t", i, quick_step);

        randomized_step= 0;
        for (int j = 0; j < i; j++) {arr_copy[j] = arr[j];}
        randomized_quicksort(arr_copy, 0, i - 1);
        printf("%d\n", randomized_step);

        // Descending data
        generate_descending_data(arr, i);
        quick_step = 0;
        quick_sort(arr, 0, i - 1);
        printf("%d\t\t%d\t\t", i, quick_step);

        randomized_step = 0;
        for (int j = 0; j < i; j++) {arr_copy[j] = arr[j]; }
        randomized_quicksort(arr_copy, 0, i - 1);
        printf("%d\n", randomized_step);

        free(arr);
        free(arr_copy);
    }

    return 0;
}
```

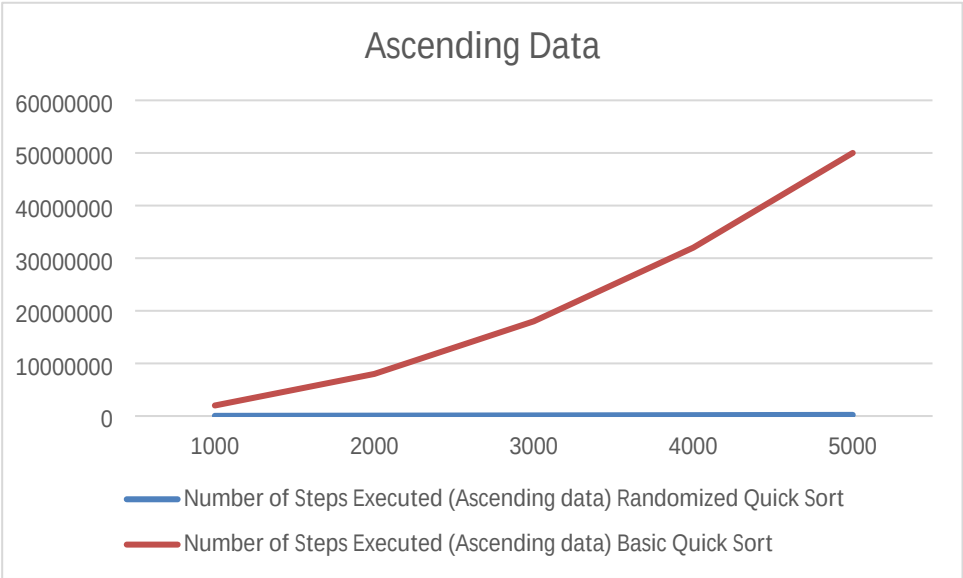
Result:

Inputs	Number of Steps Executed (Random data)	
	Randomized Quick Sort	Basic Quick Sort
1000	35623	34001
2000	82082	80929
3000	127614	119506
4000	175843	181391
5000	231386	222708
Time Complexity	nlogn	nlogn

Chart:

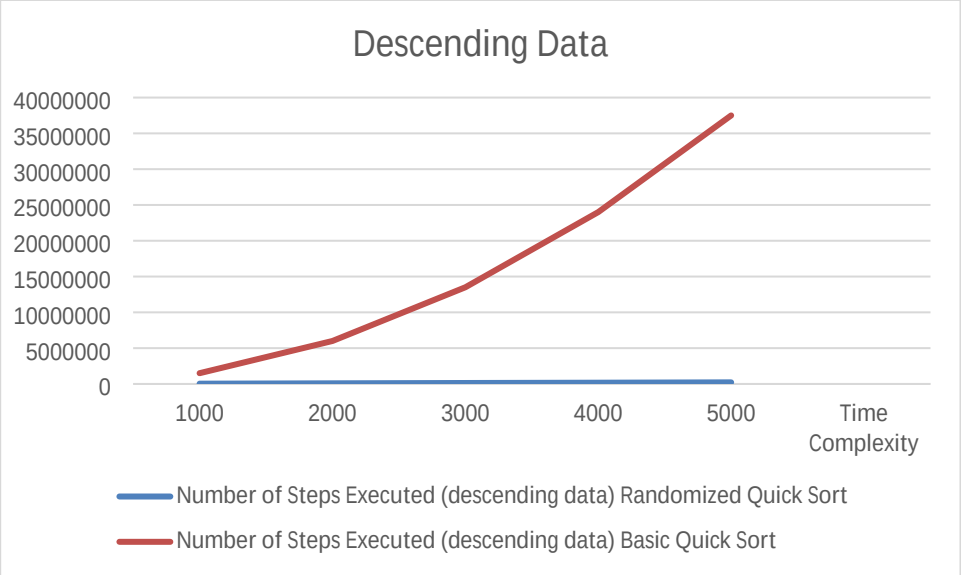
Inputs	Number of Steps Executed (Ascending data)	
	Randomized Quick Sort	Basic Quick Sort
1000	36022	2000997
2000	74598	8001997
3000	117975	18002997
4000	166250	32002997
5000	219966	50004997
Time Complexity	nlogn	n^2

Chart:



Inputs	Number of Steps Executed (descending data)	
	Randomized Quick Sort	Basic Quick Sort
1000	37594	1500997
2000	86896	6001997
3000	131833	13502997
4000	173313	24003997
5000	221996	37504997
Time Complexity	nlogn	n^2

Chart:



Quiz:

1. What is the time complexity of Randomized Quick Sort in worst case?

Answer:

The worst-case time complexity of Randomized Quick Sort is n^2.

2. What is the time complexity of basic version of Quick Sort on sorted data? Give reason of your answer.

Answer:

The time complexity of the basic version of Quick Sort on sorted data is n^2 because the pivot selection leads to highly unbalanced partitions, causing the algorithm to degenerate into a recursive sorting of nearly the entire array each time.

3. Can we always ensure O(n*lg n) time complexity for Randomized Quick Sort?

Answer:

Yes, we can generally ensure an average-case time complexity of nlogn for Randomized Quick Sort, but it cannot be guaranteed in the worst case because randomization still allows for a small probability of encountering a degenerate partitioning that could lead to n^2 complexity.

4. Which algorithm executes faster on ascending order sorted data?

Answer:

Merge Sort executes faster on ascending order sorted data compared to Quick Sort because its time complexity remains consistently nlogn, regardless of the input order, while Quick Sort can degrade to n^2 in this scenario.

Suggested Reference:

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
2. “Fundamentals of Algorithms”by E. Horowitz et al.

Rubric wise marks obtained:

Rubrics	**Understanding of problem (3)**			**Program Implementation (5)**			**Documentation &Timely Submission (2)**			**Total (10)**
	Good (3)	**Avg. (2-1)**	**Poor (1-0)**	**Good (5-4)**	**Avg. (3-2)**	**Poor (1-0)**	**Good (2)**	**Avg. (1)**	**Poor (0)**	
Marks										