Lab 4

Title: WAP to Perform the Comparative analysis of selection sort and merge so

Source Code:

a. Selection Sort:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
void selectionSort(int *arr,int *size);
int main()
{
    clock_t start, end;
    double time;
    printf("-----SELECTION SORT--------\n");
    int size, arr[500000], num;
    printf("\nEnter size of array\n");
    scanf("%d",&size);
    printf("\nEnter %d numbers\n",size);
    for (int i = 0; i < size; i++)
    {
        num = (rand()%10000);
        arr[i] = num;
    }
    start = clock();
    selectionSort(arr,&size);
    end = clock();
    printf("\nSORTED ARRAY\n");
    for (int i = 0; i < size; i++)
    {
        printf("%d  ",arr[i]);
    }
    printf("\n\n");
    time=((double)(end-start)*1000) / CLOCKS_PER_SEC;
    printf("Time=%lf mili", time);
}
```

```
void swap(int *first,int *second)
{
    int temp;
    temp=*first;
    *first=*second;
    *second=temp;
}

void selectionSort(int *arr,int *size)
{
    int temp,counter=0;
    while (counter<*size)
    {
        for (int i = counter; i < *size; i++)
        {
            if(arr[counter]>arr[i])
            {
                temp=arr[counter];
                arr[counter]=arr[i];
                arr[i]=temp;
            }
        }
        counter++;
    }
}
```

b. Merge Sort:

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
#define MAX 500000
int a[MAX];
void merge_sort(int a[],int low,int high);
void merge(int a[],int low,int high,int mid);
```

```
int main()
{
     int n,i,randNum;
     double time,start,end;
     printf("Enter the size of array:\n");
     scanf("%d",&n);
     for(i=0;i<n;i++)
     {
          randNum=(rand()%1000);
          a[i]=randNum;
          printf("%d \t",a[i]);
     }
     start=clock();
     merge_sort(a,0,n-1);
     end=clock();
     printf("\nThe sorted array is:");
     for(i=0;i<n;i++)
     {
          printf("%d \t",a[i]);
     }
     time=((double)(end-start)*1000)/CLOCKS_PER_SEC;
     printf("\nTime=%lf milli seconds",time);
}
void merge(int a[],int low,int high ,int mid)
{
     int i=low,j=mid+1,k,temp[500000];
     for (k=low;k<=high;k++)
     {
          if(i>mid)
          {
               temp[k]=a[j];
               j++;
          }
          else if(j>high)
```

```
            {
                temp[k]=a[i];
                i++;
            }
            else if(a[i]<a[j])
            {
                temp[k]=a[i];
                i++;
            }
            else
            {
                temp[k]=a[j];
                j++;
            }
        }
        for(k=low;k<=high;k++)
        {
            a[k]=temp[k];
        }
}
void merge_sort(int a[],int low,int high)
{
        int mid;
        if(low<high)
        {
            mid=(low+high)/2;
            merge_sort(a,low,mid);
            merge_sort(a,mid+1,high);
            merge(a,low,high,mid);
        }
}
```

Result Analysis and Discussion:

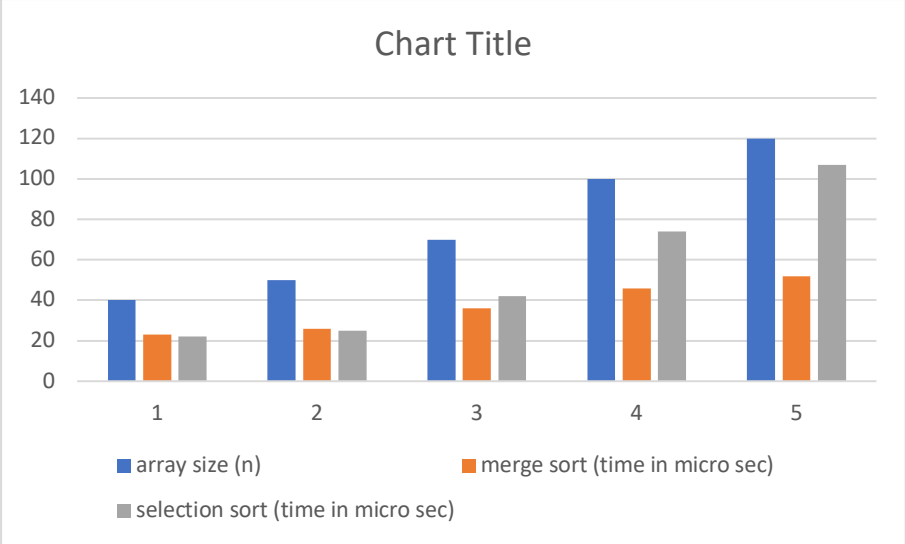
This experiment is conducted using following specifications. The algorithm is implemented using C language (clang-1400.0.29.202). During this test all the apps were closed to improve the results of the experiment.

Fig: PC used in experiment

In this experiment the comparative analysis of iterative sort algorithms has been implemented and executed for different value of n. During this experiment for different value of n the time taken by the algorithm has been measured and tabulated as shown in table below.

Array size (n)	Time in micro sec (Selection Sort)	Time in micro sec (Merge Sort)
40	22	23
50	25	26
70	42	36
100	74	46
120	107	52

The bar graph shown below is the plot of input n and the time in milliseconds taken by the algorithm while running on a system recorded in table above.



Comparative Analysis:

1. **Selection Sort:**

   Selection Sort performed less better than Merge Sort. Execution times increased with larger array sizes and it is not the most efficient choice for sorting larger datasets.

2. **Merge Sort:**

   Merge Sort performed better than Selection Sort. Execution times decreased with larger array sizes and it is most efficient choice for sorting larger datasets.

Conclusion:

In conclusion, based on the experimental data, we can recommend the following insights:

- Selection Sort are not suitable for sorting large datasets efficiently due to their poor performance with increasing array size.
- For significantly larger datasets, more advanced sorting algorithms such as Merge Sort should be considered to achieve better performance.

This experiment highlights the importance of choosing the appropriate sorting algorithm based on the dataset size and performance requirements.

Lab 5

Title: WAP to Perform the Comparative analysis of Quick sort and Randomized Quick.

Source Code:

a. Quick Sort:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main()
{
    int n, x, result;
    printf("Enter the size of array:");
    scanf("%d", &n);
    int arr[n], randNum;
    for (int i = 0; i < n; i++)
    {
        randNum = rand() % n;
        arr[i] = randNum;
    }
    clock_t start, end;
    start = clock();
    int num = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, num - 1);
    end = clock();
    printf("\nTime=%lf micro sec\n",
    ((double)(end - start)*100000) / CLOCKS_PER_SEC);
    return 0;
}
```

b. Randomized Quick Sort:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int partition(int arr[], int low, int high)
{
    int pivot = arr[low];
    int i = low - 1, j = high + 1;
    while (1)
    {
```

```
        do
        {
            i++;
        } while (arr[i] < pivot);
        do
        {
            j--;
        } while (arr[j] > pivot);
        if (i >= j)
            return j;
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
int partition_r(int arr[], int low, int high)
{
    srand(time(0));
    int random = low + rand() % (high - low);
    int temp = arr[random];
    arr[random] = arr[low];
    arr[low] = temp;
    return partition(arr, low, high);
}
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition_r(arr, low, high);
        quickSort(arr, low, pi);
        quickSort(arr, pi + 1, high);
    }
}
void printArray(int arr[], int n)
```

```
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
int main()
{
    int n, x, result;
    printf("Enter the size of array:");
    scanf("%d", &n);
    int arr[n], randNum;
    for (int i = 0; i < n; i++)
    {
        randNum = rand() % n;
        arr[i] = randNum;
    }
    clock_t start, end;
    start = clock();
    int num = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, num - 1);
    end = clock();
    printf("\nTime=%lf sec\n",
           ((double)(end - start)) / CLOCKS_PER_SEC);
    return 0;
}
```

Result Analysis and Discussion:

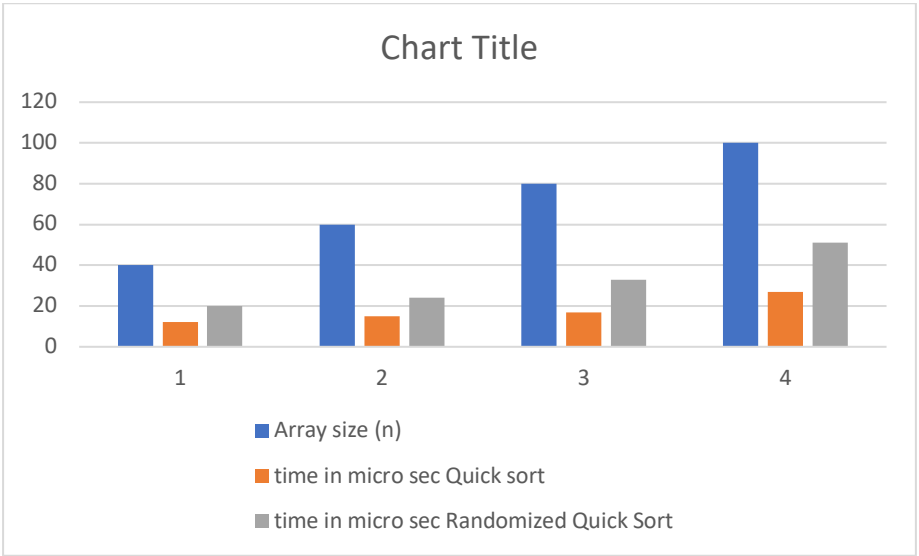
This experiment is conducted using following specifications. The algorithm is implemented using C language (clang-1400.0.29.202). During this test all the apps were closed to improve the results of the experiment.

Fig: PC used in experiment

In this experiment the comparative analysis of iterative sort algorithms has been implemented and executed for different value of n. During this experiment for different value of n the time taken by the algorithm has been measured and tabulated as shown in table below.

Array size (n)	Time in micro sec (Quick Sort)	Time in micro sec (Randomized Quick Sort)
40	12	20
60	15	24
80	17	33
100	27	51

The bar graph shown below is the plot of input n and the time in milliseconds taken by the algorithm while running on a system recorded in table above.



Comparative Analysis:

1. **Quick Sort:**

   Quick Sort exhibited the lowest execution times among the two algorithms for all array sizes. It demonstrated best performance, especially as the array size increased, with a substantial less increase in execution time. Quick Sort is recommended for sorting large datasets efficiently.

2. **Randomized Quick Sort:**

   Randomized Quick Sort exhibited the highest execution times among the two algorithms for all array sizes. It demonstrated poor performance, especially as the array size increased, with a substantial increase in execution time. Randomized Quick Sort is not recommended for sorting large datasets efficiently.

Conclusion:

In conclusion, based on the experimental data, we can recommend the following insights:

- Randomized Quick Sort are not suitable for sorting large datasets efficiently due to their poor performance with increasing array size.
- For significantly larger datasets, more advanced sorting algorithms such as Quick Sort should be considered to achieve better performance.

This experiment highlights the importance of choosing the appropriate sorting algorithm based on the dataset size and performance requirements.

Lab 6

Title: WAP to Perform the Comparative analysis of Quick sort and Heap sort.

Source Code:

a. **Quick Sort:**

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
```

```
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main()
{
    int n, x, result;
    printf("Enter the size of array:");
    scanf("%d", &n);
    int arr[n], randNum;
    for (int i = 0; i < n; i++)
    {
        randNum = rand() % n;
        arr[i] = randNum;
    }
    clock_t start, end;
    start = clock();
    int num = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, num - 1);
    end = clock();
    printf("\nTime=%lf micro sec\n",
    ((double)(end - start)*100000) / CLOCKS_PER_SEC);
    return 0;
}
```

b. Heap sort:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void heapify(int arr[], int N, int i)
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < N && arr[left] > arr[largest])
        largest = left;
    if (right < N && arr[right] > arr[largest])
        largest = right;
    if (largest != i)
    {
        swap(&arr[i], &arr[largest]);
        heapify(arr, N, largest);
    }
}
void heapSort(int arr[], int N)
{

    for (int i = N / 2 - 1; i >= 0; i--)

        heapify(arr, N, i);

    for (int i = N - 1; i >= 0; i--)
    {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
void printArray(int arr[], int N)
{
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

```
int main()
{
    int n, x, result;
    printf("Enter the size of array:");
    scanf("%d", &n);
    int arr[n], randNum;
    for (int i = 0; i < n; i++)
    {
        randNum = rand() % n;
        arr[i] = randNum;
    }
    clock_t start, end;
    start = clock();
    int num = sizeof(arr) / sizeof(arr[0]);
    heapSort(arr, num);
    end = clock();
    printf("\nTime=%lf micro sec\n",
    ((double)(end - start)*1000000) / CLOCKS_PER_SEC);
    return 0;
}
```

Result Analysis and Discussion:

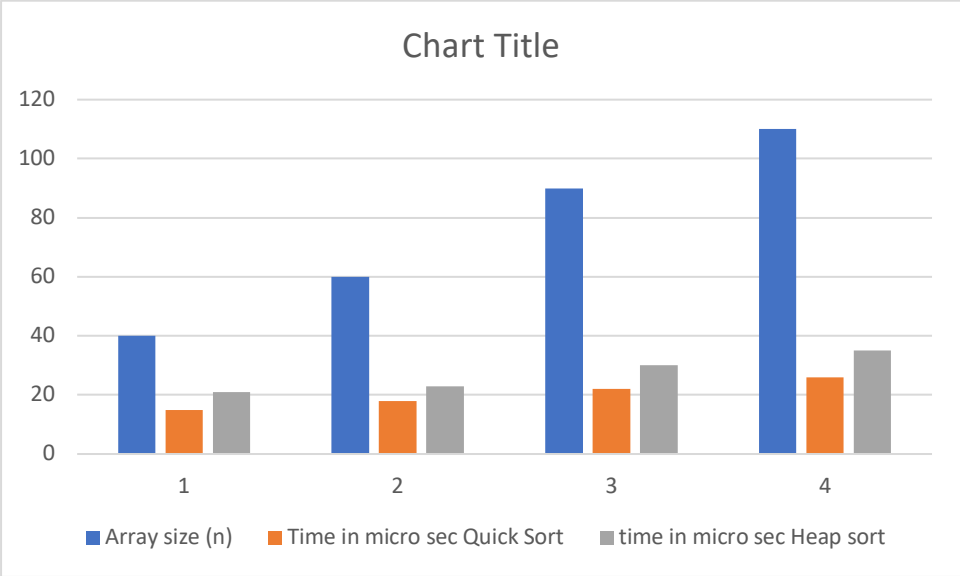
This experiment is conducted using following specifications. The algorithm is implemented using C language (clang-1400.0.29.202). During this test all the apps were closed to improve the results of the experiment.

Fig: PC used in experiment

In this experiment the comparative analysis of iterative sort algorithms has been implemented and executed for different value of n. During this experiment for different value of n the time taken by the algorithm has been measured and tabulated as shown in table below.

Array size (n)	Time in micro sec (Quick Sort)	Time in micro sec (Heap Sort)
40	15	21
60	18	23
90	22	30
110	26	35

The bar graph shown below is the plot of input n and the time in milliseconds taken by the algorithm while running on a system recorded in table above.



Comparative Analysis:

1. **Quick Sort:**

   Quick Sort exhibited the lowest execution times among the two algorithms for all array sizes. It demonstrated best performance, especially as the array size increased, with a substantial less increase in execution time. Quick Sort is recommended for sorting large datasets efficiently.

2. **Heap Sort:**

   Heap Sort exhibited the highest execution times among the two algorithms for all array sizes. It demonstrated poor performance, especially as the array size increased, with a

substantial increase in execution time. Randomized Quick Sort is not recommended for sorting large datasets efficiently.

Conclusion:

In conclusion, this comparative analysis indicates that for sorting large datasets in the tested conditions, Quicksort is the preferred algorithm when speed is a primary concern, as it consistently delivered faster execution times than Heapsort across varying array sizes. However, selecting the most suitable sorting algorithm should always consider the unique requirements and constraints of the given problem or application.