Practical File

Index

Practical - 1

Aim: Write a program to perform Linear Search and also find its complexity in Best, Worst, and Average case.

Code

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

int main() {
    int num;
    int i, key, element_found = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);
    int arr[num];
    printf("\nEnter the elements: ");
    for (i = 0; i < num; i++) {
        scanf("%d", &arr[i]);
    }

    printf("\nEnter the element to be searched: ");
    scanf("%d", &key);

    clock_t start, end;
    double cpu_time_used;
    start = clock();

    for (i = 0; i < num; i++) {
        if (key == arr[i]) {
            element_found = 1;
            break;
        }
    }

    end = clock();
    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    if (element_found == 1)
        printf("Element found at index %d\n", i + 1);
    else
        printf("Element not found\n");

    printf("Runtime: %f seconds\n", cpu_time_used);
    return 0;
}
```

Output

- Input: Number of elements, array elements, and the key to search.
- Output: Whether the element is found (with index) or not, and the runtime.

Complexity Analysis

- **Best Case:** $O(1)$
- **Worst Case:** $O(n)$
- **Average Case:** $O(n)$

Practical - 2

Aim: Write a program to perform Binary Search and also find its complexity in Best, Worst, and Average case.

Code

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

int main() {
    int n, i, x;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter %d elements in sorted order: ", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Enter the element to search: ");
    scanf("%d", &x);

    clock_t start, end;
    start = clock();
    int low = 0, high = n - 1, result = -1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == x) {
            result = mid;
            break;
        } else if (arr[mid] < x) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    end = clock();
    double runtime = ((double)(end - start)) / CLOCKS_PER_SEC;

    if (result == -1)
        printf("Element is not present in array\n");
    else
        printf("Element is present at index %d\n", result + 1);

    printf("Runtime: %f seconds\n", runtime);
    return 0;
}
```

Output

- Input: Number of elements (in sorted order), array elements, and the key to search.
- Output: Whether the element is found (with index) or not, and the runtime.

Complexity Analysis

- **Best Case:** $O(1)$
- **Worst Case:** $O(\log n)$
- **Average Case:** $O(\log n)$

Practical - 3

Aim: Write a program to implement Selection Sort and also find its complexity in Best, Worst, and Average case.

Code

```c
#include<stdio.h>
void main(){
    int n,i,j,temp,minindex,flag=0;
    printf("enter the number of elements:");
    scanf("%d",&n);
    int arr[n];
    printf("enter the elements:");
    for(i=0;i<n;i++){
        scanf("%d",&arr[i]);
    }
    for(i=0;i<=n-1;i++){
        minindex=i;
        for(j=i+1;j<n;j++){
            if (arr[j]<arr[minindex]){
                minindex=j;
                flag=1;
            }
        }
        if(flag==1){
            temp=arr[i];
            arr[i]=arr[minindex];
            arr[minindex]=temp;
        }
    }
    for(i=0;i<n;i++){
        printf("%d \t",arr[i]);
    }
}
```

Output

- Input: Number of elements and array elements.
- Output: Array elements sorted in ascending order.

Complexity Analysis

- **Best Case:** $O(n^2)$
- **Worst Case:** $O(n^2)$
- **Average Case:** $O(n^2)$

Practical - 4

Aim: Write a program to implement Merge Sort and also find its complexity in Best, Worst, and Average case.

Code

```c
#include<stdio.h>
void merge(int a[],int low,int mid,int high){
    int temp[high-low+1],i=0;
    int left=low,right=mid+1;
    while(left<=mid && right<=high){
        if(a[left]<=a[right]){
            temp[i]=a[left];
            left++;
        }
        else{
            temp[i]=a[right];
            right++;
        }
        i++;
    }
    while(left<=mid){
        temp[i]=a[left];
        left++;
        i++;
    }
    while(right<=high){
        temp[i]=a[right];
        right++;
        i++;
    }
    for(i=low;i<=high;i++){
        a[i]=temp[i-low];
    }
}
int sort(int a[],int low,int high){
    if(low<high){
        int mid=(high+low)/2;
        sort(a,low,mid);
        sort(a,mid+1,high);
        merge(a,low,mid,high);
    }
}
int main(){
    int n;
    printf("enter number of elements:");
    scanf("%d",&n);
    int a[n];
    printf("enter the elements:");
```

```
    for(int i=0;i<n;i++){
        scanf("%d",&a[i]);
    }
    sort(a,0,n-1);
    for(int i=0;i<n;i++){
        printf("%d ",a[i]);
    }
}
```

Output

- Input: Number of elements and array elements.
- Output: Array elements sorted in ascending order.

Complexity Analysis

- **Best Case:** $O(n \log n)$
- **Worst Case:** $O(n \log n)$
- **Average Case:** $O(n \log n)$

Practical - 5

Aim: Write a program to implement Quick Sort and also find its complexity in Best, Worst, and Average case.

Code

```c
#include <stdio.h>
#include <stdbool.h>

int partition(int a[], int low, int high) {
    int pivot = a[low];
    int i = low - 1;
    int j = high + 1;
    while (true) {
        do {
            i++;
        } while (a[i] < pivot);

        do {
            j--;
        } while (a[j] > pivot);

        if (i >= j)
            return j;

        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}

void quicksort(int a[], int low, int high) {
    if (low < high) {
        int p = partition(a, low, high);
        quicksort(a, low, p);
        quicksort(a, p + 1, high);
    }
}

int main() {
    int arr[100], n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter the array elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    quicksort(arr, 0, n - 1);
    printf("Sorted array: ");
```

```
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Output

- Input: Number of elements and array elements.
- Output: Array elements sorted in ascending order.

Complexity Analysis

- **Best Case:** $O(n \log n)$
- **Worst Case:** $O(n^2)$
- **Average Case:** $O(n \log n)$

Practical - 6

Aim: Write a program to multiply two matrices using Strassen's multiplication algorithm.

Code

```c
#include<stdio.h>
int main(){
  int a[2][2], b[2][2], c[2][2], i, j;
  int m1, m2, m3, m4 , m5, m6, m7;

  printf("Enter the 4 elements of first matrix: ");
  for(i = 0;i < 2; i++)
      for(j = 0;j < 2; j++)
           scanf("%d", &a[i][j]);

  printf("Enter the 4 elements of second matrix: ");
  for(i = 0; i < 2; i++)
      for(j = 0;j < 2; j++)
           scanf("%d", &b[i][j]);

  printf("\nThe first matrix is\n");
  for(i = 0; i < 2; i++){
      printf("\n");
      for(j = 0; j < 2; j++)
           printf("%d\t", a[i][j]);
  }

  printf("\nThe second matrix is\n");
  for(i = 0;i < 2; i++){
      printf("\n");
      for(j = 0;j < 2; j++)
           printf("%d\t", b[i][j]);
  }

  m1= (a[0][0] + a[1][1]) * (b[0][0] + b[1][1]);
  m2= (a[1][0] + a[1][1]) * b[0][0];
  m3= a[0][0] * (b[0][1] - b[1][1]);
  m4= a[1][1] * (b[1][0] - b[0][0]);
  m5= (a[0][0] + a[0][1]) * b[1][1];
  m6= (a[1][0] - a[0][0]) * (b[0][0]+b[0][1]);
  m7= (a[0][1] - a[1][1]) * (b[1][0]+b[1][1]);

  c[0][0] = m1 + m4 - m5 + m7;
  c[0][1] = m3 + m5;
  c[1][0] = m2 + m4;
  c[1][1] = m1 - m2 + m3 + m6;

  printf("\nAfter multiplication using Strassen's algorithm \n");
```

```
  for(i = 0; i < 2 ; i++){
       printf("\n");
       for(j = 0;j < 2; j++)
            printf("%d\t", c[i][j]);
  }

  return 0;
}
```

Output

- Input: Elements of two 2x2 matrices.
- Output: Resultant matrix after multiplication using Strassen's algorithm.

Complexity Analysis

- **Time Complexity:** Approximately $O(n^{2.81})$ (better than the standard $O(n^3)$ for matrix multiplication).