

INDEX

Subject: Analysis and Design of Algorithms (3150703)

Enrollment No: 180170107030

Name of the Student: Gabu Siddharth Merambhai

Sr. No	Aim	CO	Date	Page No	Marks (out of 10)	Faculty Signature
1	Implementation and Time analysis of sorting algorithms. Bubble sort, Selection sort, Insertion sort. Also Derive the time complexity of above algorithms using Asymptotic analysis.	CO1	7/7/2020	1		
2	Implementation and Time analysis of factorial and fibonacci program using iterative and recursive method. Also Derive the time complexity of above algorithms using Asymptotic analysis.	CO1	14/7/2020	5		
3	Implementation and Time analysis of linear and binary search algorithm. Also Derive the time complexity of above algorithms using Asymptotic analysis.	CO1	21/7/2020	18		
4	Implementation of two way Merge sort and Quick sort using divide and conquer strategy. Also Derive the time complexity of above algorithms using Asymptotic analysis.	CO2	28/7/2020	20		
5	Implementation of a fractional knapsack problem using greedy approach.	CO3	4/8/2020	25		
6	Implementation of chain matrix multiplication using dynamic programming.	CO3	1/9/2020	27		
7	Implementation of making a change problem using dynamic programming	CO3	8/9/2020	29		
8	Implementation of Graph Searching strategy (DFS and BFS).	CO6	15/9/2020	32		
9	Implementation of any string matching algorithm.	CO4	6/10/2020	35		
10	Prepare a detailed write up for various P and NP Problems.	CO5	13/10/2020	37		

Practical - 1

AIM: Implementation and Time analysis of sorting algorithms. Bubble sort, Selection sort, Insertion sort. Also derive the time complexity of above algorithms using asymptotic analysis.

Inputs	1000	10,000	25,000	50,000	1,00,000
Bubble sort					
Best Case	0 μ s	0 μ s	0 μ s	0 μ s	992 μ s
Average Case	10ms	827ms	4933ms	20s	81s
Worse Case	6ms	905ms	4920ms	21s	82s
Selection Sort					
Best Case	4ms	281ms	1s	6s	25s
Average Case	4ms	252ms	1580ms	6s	25s
Worse Case	3ms	253ms	1597ms	6s	25s
Insertion sort					
Best Case	0 μ s	0 μ s	0 μ s	1ms	1ms
Average Case	1ms	142ms	821ms	3s	13s
Worse Case	1ms	146ms	824ms	3s	13s

```
void BubbleSort(int arr[],int n){
    int temp;
    bool swapped = true;
    while(swapped){
        swapped = false;
        for(int i=0;i<n-1;++i){
            if(arr[i]>arr[i+1]){
                temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
        }
    }
}
```

```
        arr[i+1] = temp;
        swapped = true;
    }
}
}
}
void insertionsort(int arr[], int n){
    int i, temp, j;
    for (i = 1; i < n; ++i){
        temp = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > temp){
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = temp;
    }
}
void selectionsort(int arr[], int n){
    int i, j, yaad,temp;
    for (i = 0; i < n-1; ++i){
        yaad = i;
        for (j = i+1; j < n; ++j)
            if (arr[j] < arr[yaad])
                yaad = j;

        temp = arr[yaad];
        arr[yaad] = arr[j];
        arr[j] = temp;
    }
}
void selectsort(int arr[], int n){
    int i, j, yaad,temp;
    for (i = 0; i < n-1; ++i){
        yaad = i;
        for (j = i+1; j < n; ++j)
            if (arr[j] > arr[yaad])
                yaad = j;

        temp = arr[yaad];
```

```
        arr[yaad] = arr[j];
        arr[j] = temp;
    }
}
int n = 100000;

int main() {
    int i,a[n],b[n],c[n];
    for(i = 0; i < n ; ++i)
        a[i] = b[i] = c[i] = rand() % n + 1; // 1 to n range

    selectsort(a,n);
    selectsort(b,n);
    selectsort(c,n);

    auto start2 = high_resolution_clock::now();
    cout<<"selection sort started"<<endl;
    selectionsort(c,n);
    auto stop2 = high_resolution_clock::now();
    auto duration2 = duration_cast<milliseconds>(stop2 - start2);
    cout << "Time taken by selection sort: "
        << duration2.count() << " milliseconds" << endl;

    auto start = high_resolution_clock::now();
    cout<<"Bubble sort started"<<endl;
    Bubbleesort(a,n);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);
    cout << "Time taken by BubbleSort: "
        << duration.count() << " milliseconds" << endl;

    auto start1 = high_resolution_clock::now();
    cout<<"insertion sort started"<<endl;
    insertionsort(b,n);
    auto stop1 = high_resolution_clock::now();
    auto duration1 = duration_cast<milliseconds>(stop1 - start1);
    cout << "Time taken by insertion sort: "
        << duration1.count() << " milliseconds" << endl;
```

```
return 0;}
```

Serial no.	Algorithm	Best case	Average case	Worst case
1.	Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
2.	Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
3.	Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
4.	Merge sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
5.	Quick sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$

Practical - 2

AIM: Implementation and Time analysis of factorial and fibonacci program using iterative and recursive method. Also Derive the time complexity of above algorithms using Asymptotic analysis.

Inputs (n)	Recursive fibonacci	Iterative fibonacci
20	0s	0 μ s
30	1s	0 μ s
40	137s	0 μ s

Inputs (n)	Recursive factorial	Iterative factorial
100	0 μ s	0 μ s
1000	5003 μ s	1999 μ s
10000	668ms	596ms

```
#include <iostream>
#include <chrono>
#include <boost/multiprecision/cpp_int.hpp>
#define ll long long
using namespace boost::multiprecision;
using namespace std;
using namespace std::chrono;
```

```
cpp_int recursionfactorial(cpp_int n){
    if (n == 0)
        return 1;
    return n * recursionfactorial(n - 1);
}
cpp_int iterativefactorial(cpp_int n){
    cpp_int i;
    cpp_int res = 1;
    for(i = 2; i <= n; ++i)
        res *= i;
    return res;
}
cpp_int recursionFibonacci(cpp_int n){
```

```
        if (n <= 1)
            return n;
        return recursionFibonacci(n-1) + recursionFibonacci(n-2);
    }
void iterativeFibonacci(cpp_int n){
    cpp_int i,n1=0,n2=1,n3;
    cout<<n1<<" "<<n2<<" ";
    for(i=2;i<n;++i) //loop starts from 2 because 0 and 1 are already printed
    {
        n3=n1+n2;
        n1=n2;
        n2=n3;
    }
    cout<<n3<<" ";
}
int main() {
    cpp_int m,n;
    cout << "enter fibonacci number: " << endl; // prints !!!Hello World!!!
    cin>>n;
    cout<<"recursion"<<endl;
    cout<<"0 "<<"1 ";
    high_resolution_clock::time_point start1 = high_resolution_clock::now();
    cout<<recursionFibonacci(n);
    high_resolution_clock::time_point start2 = high_resolution_clock::now();
    auto duration = duration_cast<seconds>(start2 - start1);
    cout << "\nTime taken by recursion: "
        << duration.count() << " seconds" << endl;

    cout<<"\n"<<"iterative"<<endl;
    start1 = high_resolution_clock::now();
    iterativeFibonacci(n);
    start2 = high_resolution_clock::now();
    auto duration3 = duration_cast<microseconds>(start2 - start1);
    cout << "Time taken by iterativefactorial: "
        << duration3.count() << " microseconds" << endl;

    cout<<"\nEnter recursion number: ";
    cin>>m;
    start1 = high_resolution_clock::now();
    cout<<"recursion factorial of number:"<<recursionfactorial(m)<<endl;
```

```
start2 = high_resolution_clock::now();
auto duration1 = duration_cast<milliseconds>(start2 - start1);
cout << "Time taken by recursionfactorial: "
      << duration1.count() << " milliseconds" << endl;

start1 = high_resolution_clock::now();
cout<<"iterative factorial of number:"<<iterativefactorial(m)<<endl;
start2 = high_resolution_clock::now();
auto duration2 = duration_cast<milliseconds>(start2 - start1);
cout << "Time taken by iterativefactorial: "
      << duration2.count() << " milliseconds" << endl;

return 0;
}
```

Time Complexity:

Recursive Fibonacci: $T(n) = T(n-1) + T(n-2)$

Iterative Fibonacci: $T(n) = O(n)$

Recursive Factorial: $O(n)$

Iterative Factorial: $O(n)$

Practical - 3

Gprof analysis of linked list program

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	ms/call	ms/call	name
37.56	0.03	0.03	20	1.50	1.50	delete_at_beging()
25.04	0.05	0.02	23	0.87	0.87	append()
25.04	0.07	0.02	8	2.50	2.50	display()
12.52	0.08	0.01	4	2.50	3.38	insert_node_ascd()
0.00	0.08	0.00	9	0.00	0.00	add_beging()
0.00	0.08	0.00	5	0.00	0.00	delete_pos()
0.00	0.08	0.00	4	0.00	0.00	Asort()

% the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
 listing.

calls the number of times this function was invoked, if
 this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
 else blank.

total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
 function is profiled, else blank.

name the name of the function. This is the minor sort
 for this listing. The index shows the location of

the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Copyright (C) 2012-2018 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification, are permitted in any medium without royalty provided the copyright notice and this notice are preserved.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 12.48% of 0.08 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	0.08		main [1]
		0.03	0.00	20/20	delete_at_beging() [2]
		0.02	0.00	8/8	display() [4]
		0.02	0.00	19/23	append() [3]
		0.01	0.00	4/4	insert_node_ascd() [5]
		0.00	0.00	9/9	add_beging() [12]
		0.00	0.00	5/5	delete_pos() [13]

		0.03	0.00	20/20	main [1]
[2]	37.5	0.03	0.00	20	delete_at_beging() [2]

		0.00	0.00	4/23	insert_node_ascd() [5]
		0.02	0.00	19/23	main [1]
[3]	25.0	0.02	0.00	23	append() [3]

		0.02	0.00	8/8	main [1]
[4]	25.0	0.02	0.00	8	display() [4]

		0.01	0.00	4/4	main [1]
[5]	16.8	0.01	0.00	4	insert_node_ascd() [5]
		0.00	0.00	4/23	append() [3]
		0.00	0.00	4/4	Asort() [14]

		0.00	0.00	9/9	main [1]
[12]	0.0	0.00	0.00	9	add_beging() [12]

	0.00	0.00	5/5	main [1]
[13]	0.0	0.00	0.00	5 delete_pos() [13]
	0.00	0.00	4/4	insert_node_ascd() [5]
[14]	0.0	0.00	0.00	4 Asort() [14]

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function.

The lines above it list the functions that called this function, and the lines below it list the functions this one called.

This line lists:

- index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so it is easier to look up where the function is in the table.
- % time This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.
- self This is the total amount of time spent in this function.
- children This is the total amount of time propagated into this function by its children.
- called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a '+' and the number of recursive calls.
- name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

- self This is the amount of time that was propagated directly from the function into this parent.
- children This is the amount of time that was propagated from the function's children into this parent.
- called This is the number of times this parent called the function '/' the total number of times the function was called. Recursive calls to the function are not included in the number after the '/'.
- name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word '<spontaneous>' is printed in the 'name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

- self This is the amount of time that was propagated directly from the child into the function.
- children This is the amount of time that was propagated from the child's children to the function.
- called This is the number of times the function called this child '/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the '/'.
- name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed

between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.)

The '+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Copyright (C) 2012-2018 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification, are permitted in any medium without royalty provided the copyright notice and this notice are preserved.

Index by function name

[12] add_beging()	[5] insert_node_ascd()	[4] display()
[13] delete_pos()	[14] Asort()	
[2] delete_at_beging()	[3] append()	

C - PROGRAM

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

```
void Asort();
void append();
void add_beging();
void display();
void delete_at_beging();
void delete_pos();
void insert_node_ascd();
```

```
struct node{
    int data;
    struct node *next;};
struct node *root=NULL;
int asc,count;
int main(){
    int ch;
    count=0;
```

```

printf("\n\t***||--- SOME GUIDELINES FOR YOU ---||***");
printf("\n1:for insert node at the end ");
printf("\n2:for insert node at the beging ");
printf("\n3:for display linked list ");
printf("\n4:for delete node at the beging ");
printf("\n5:for inserting node asc order ");
printf("\n6:for delete position number ");
printf("\n7:for EXIT ");
do{
printf("\n enter your choice here:");
scanf("%d",&ch);
switch(ch){
case 1:append(); break;
case 2: add_beging();break;
case 3:display();break;
case 4:delete_at_beging();break;
case 5:insert_node_ascd();break;
case 6:delete_pos();break;
case 7:break;
default:printf("\nenter valid choice");}
}while(ch!=7);
return 0;}
void insert_node_ascd()
{int i;
asc=1;
for(i=0;i<100000;i++){ }
Asort();
append();
}
void append(){
int i;
struct node *temp;
temp=(struct node*)malloc(sizeof(struct node));
count++;
printf("\n enter your value here:");
scanf("%d",&temp->data);
temp->next=NULL;
if(root==NULL)root=temp;
else if(asc==1){
struct node *ASC,*pre;

```

```
asc=0;
ASC=root;
pre=ASC;
while(ASC->next!=NULL){
    if(ASC->data>=temp->data){
        temp->next=ASC;
        pre->next=temp;
        break;}
    pre=ASC;
    ASC=ASC->next;}
if(ASC->next==NULL){
    if(ASC->data<temp->data)
        ASC->next=temp;
    else{
        temp->next=ASC;
        root=temp;
    } }
else{
    struct node *p;
    p=root;
    while(p->next!=NULL){
        p=p->next;}
    p->next=temp;}
for(i=0;i<100000;i++){ }
}
void add_beging()
{int i;
 struct node *temp;
 temp=(struct node*)malloc(sizeof(struct node));
 count++;
 printf("\nenter your value here:");
 scanf("%d",&temp->data);
 for(i=0;i<100000;i++){
 }
 if(root==NULL)
 {
    temp->next=NULL;
    root=temp;
 }
 else
```

```
{
    temp->next=root;
    root=temp;
}
}
void display()
{
    int i;
    for(i=0;i<100000;i++){
    }
    struct node *p;
    p=root;
    if(root==NULL)
        printf("\nlinked list is empty");
    else
    {
        printf("\nyour values are:");
        while(p->next!=NULL)
        {
            printf("\n%d",p->data);
            p=p->next;
        }
        printf("\n%d",p->data);
    }
}
void delete_at_beging()
{
    int i;
    for(i=0;i<100000;i++){
    }
    struct node *p;
    p=root;
    if(root==NULL)
        printf("\nlinked list is empty");
    else
    {
        root=p->next;
        count--;
        printf("\nyour deleted value is=%d",p->data);
        free(p);
    }
}
```



```
}  
}  
void Asort()  
{ int t,loop;  
  struct node *a;  
  do  
  {  
    a=root;  
    loop=0;  
    while(a->next!=NULL)  
    {  
      if(a->data > a->next->data)  
      {  
        t=a->data;  
        a->data=a->next->data;  
        a->next->data=t;  
        loop=1;  
      }  
      a=a->next;  
    }  
  }while(loop);  
}  
void delete_pos()  
{  
  int po,d,i;  
  struct node *t,*pre;  
  t=root;  
  if(root==NULL)  
    printf("\nlinked list is empty");  
  else  
  {  
    printf("\n enter your position here:");  
    scanf("%d",&po);  
    printf("\nPress 1 for before position or 2 for after:");  
    scanf("%d",&d);  
    if(po<=count&&(d==1||d==2))  
    {  
      if(po==1)  
        printf("\n Nothing here to delete after and before position");  
      else if(d==1)
```

```
{
if(po==2)
{ root=t->next;
  free(t);
}
else
{
  for(i=2;i<po;i++)
  {pre=t;
   t=t->next; }
  pre=t->next;
  free(t);
  count--;
}
}
else if(d==2)
{
  for(i=1;i<po;i++)
  { t=t->next; }
  if(t->next==NULL)
    printf("\nnothing here to delete:");
  else
  {
    count--;
    t->next=t->next->next;
  }}
else
  printf("\nyour input is wrong");
}
```

Practical - 3

AIM: Implementation and Time analysis of linear and binary search algorithm Also Derive the time complexity of above algorithms using Asymptotic analysis.

Inputs	Time complexity	1000	10000	50000	100000
Linear Search	$O(n)$	0 μ s	0 μ s	0 μ s	1001 μ s
Binary Search	$O(\log n)$	0 μ s	0 μ s	0 μ s	0 μ s

```
#include <iostream>
#include <chrono>
#define ll long long
using namespace std;
using namespace std::chrono;

int search(int arr[],int n,int x){
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

int binarySearch(int arr[],int l,int r,int x){
    while (l <= r){
        int m =l + (r - l) / 2;
        if (arr[m] == x) // Check if x is present at mid
            return m;
        if (arr[m] < x) // If x greater, ignore left half
            l = m + 1;
        else // If x is smaller, ignore right half
            r = m - 1;
    }
    return -1; // if we reach here, then element was not present
}

void insertionsort(int arr[], int n){
    int i, temp, j;
    for (i = 1; i < n; ++i){
```

```
        temp = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > temp){
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = temp;
    }
}
int n = 100000;
int main(void) {
    int i, arr[n];
    for(i = 0; i < n ; ++i)
        arr[i] = rand() % n + 1; // 1 to n range

    int x = 98937;

    auto start = high_resolution_clock::now();
    int result = search(arr, n, x);
    auto end = high_resolution_clock::now();
    (result == -1) ? cout << "Linear search Element is not present in array"
                  : cout << "Linear search Element is present at index " << result;

    auto duration1 = duration_cast<microseconds>(end - start);

    cout << "\nTime taken by Linear search: "
          << duration1.count() << " microseconds" << endl;

    insertionsort(arr, n);
    auto start1 = high_resolution_clock::now();
    result = binarySearch(arr, 0, n - 1, x);
    auto start2 = high_resolution_clock::now();
    (result == -1) ? cout << "\nBinary search Element is not present in array"
                  : cout << "\nBinary search Element is present at index " << result;

    auto duration = duration_cast<microseconds>(start2 - start1);

    cout << "\nTime taken by Binary search: "
          << duration.count() << " microseconds" << endl;
    return 0;
}
```

Practical – 4

AIM: Implementation of two way Merge sort and Quick sort using divide and conquer strategy. Also derive the time complexity of above algorithms using asymptotic analysis.

Number of inputs	cases	Merge sort	Quick sort
1000	best	0.00s	0.00s
	average	0.00s	0.00s
	worst	0.00s	0.00s
10000	best	0.00s	0.00s
	average	0.00s	0.016s
	worst	0.00s	0.204s
50000	best	0.00s	0.00s
	average	0.56s	0.35s
	worst	1.15s	0.204s
100000	best	1.31s	1.31s
	average	1.32s	1.31s
	worst	4.67s	2.57s

Time complexity of quick sort:

Worst case: $O(n^2)$

Best case: $O(n \log n)$

Time complexity of merge sort:

Best & Worst case: $O(n \log n)$

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

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

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

```
void swap(long int* a,long int* b)
```

```
{
```

```
    long int t = *a;
```

```
    *a = *b;
```

```
    *b = t;
```

```
}
```

```
void quickSortMiddle(long int a[],long int left,long int right){
```

```
    if(left >= right) return;
```

```
    long int pivot = a[left + (right - left)/2];
```

```
    long int leftI = left-1;
    long int rightI = right+1;
    while(1)
    {
        while(a[++leftI] < pivot);
        while(a[--rightI] > pivot);
        if(leftI >= rightI)break;
        swap(&a[leftI], &a[rightI]);
    }
    quickSortMiddle(a,left,rightI);
    quickSortMiddle(a,rightI+1,right);
}

int partition (long int arr[],long int low,long int high)
{
    long int pivot;
    pivot = arr[high];
    //long int pivot = arr[high];  // pivot
    long int i = (low - 1); // Index of smaller element

    for (long int j = low; j < high; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(long int arr[],long int low,long int high)
{
    if (low < high)
    {
        long int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
}  
void merge(long int arr[],long int l,long int m,long int r)  
{  
    long int i, j, k;  
    long int n1 = m - l + 1;  
    long int n2 = r - m;  
  
    /* create temp arrays */  
    long int L[n1], R[n2];  
  
    for (i = 0; i < n1; i++)  
        L[i] = arr[l + i];  
    for (j = 0; j < n2; j++)  
        R[j] = arr[m + 1 + j];  
  
    i = 0; j = 0; k = l;  
    while (i < n1 && j < n2) {  
        if (L[i] <= R[j]) {  
            arr[k] = L[i];  
            i++;  
        }  
        else {  
            arr[k] = R[j];  
            j++;  
        }  
        k++;  
    }  
    while (i < n1) {  
        arr[k] = L[i];  
        i++;  
        k++;  
    }  
    while (j < n2) {  
        arr[k] = R[j];  
        j++;  
        k++;  
    }  
}  
  
void mergeSort(long int arr[],long int l,long int r)
```

```
{
    if (l < r) {
        long int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

void printArray(long int A[], long int size)
{
    long int i;
    for (i = 0; i < size; i++)
        printf("%ld ", A[i]);
    printf("\n");
}

/* Driver program to test above functions */
int main() {
    clock_t begin, end;
    long int n, i, j;
    int ch;
    srand(time(0));
    printf("Merge Sort & Quick Sort\n");
    printf("Enter number of terms:");
    scanf("%ld", &n);
    long int arr[n], arr1[n];
    printf("\nEnter 1 for best case:\n");
    printf("Enter 2 for average case:\n");
    printf("Enter 3 for worst case:\n");
    scanf("%d", &ch);

    switch(ch)
    {
        case 1:
            for(i=0; i<n; i++)
                { arr1[i]=arr[i]=i; }
```



```
        break;
    case 2:
        for(i=0;i<n;i++)
            { arr1[i]=arr[i]=rand();}
        break;
    case 3:
        for(j=n,i=0;i<n;i++,j--)
            { arr1[i]=arr[i]=j;}
        break;
    default:
        printf("\nEnter valid choice");
    }

// printf("Given array in merge & quick: \n");
// printArray(arr, n);

begin=clock();
mergeSort(arr, 0, n - 1);
end=clock();
double timediff=(double)(end-begin)/CLOCKS_PER_SEC;
printf("The time difference in MergeSort:%lf seconds",timediff);
//printf("\nsorted array in merge sort: \n");
//printArray(arr, n);

if(ch==1){
    begin=clock();
    quickSortMiddle(arr1,0,n-1);
    end=clock();
} else {
    begin=clock();
    quickSort(arr1,0,n-1);
    end=clock();
}
timediff=(double)(end-begin)/CLOCKS_PER_SEC;
printf("\nThe time difference in QuickSort:%lf seconds",timediff);
//printf("\nsorted array in quick sort: \n");
//printArray(arr1, n);

return 0;
}
```

Practical - 5

Aim: Implementation of a fractional knapsack problem using greedy approach.

```
#include <iostream>
using namespace std;

double fractionalKnapsack(int W, int val[],int wt[], int n){
    int curWeight = 0; // Current weight in knapsack
    double finalvalue = 0.0; // Result (value in Knapsack)
    // Looping through all Items
    for(int i = 0; i < n; i++){
        if(curWeight + wt[i] <= W){
            curWeight += wt[i];
            finalvalue += val[i];
        }
        else{
            int remain = W - curWeight;
            finalvalue += val[i] * ((double) remain / wt[i]);
            break;
        }
    }
    return finalvalue;
}
```

```
// driver program to test above function
int main()
{
    int W = 50; // Weight of knapsack
    int val[] = {60,80,10};
    int wt[] = {10,20,40};
    int temp=0,mid;
    bool swap;
    int n = sizeof(val) / sizeof(val[0]);
    for(int i=0;i<n-1;i++){
        double d1 = (double)val[i]/wt[i];
        swap=false;
        for(int j=i+1;j<n;j++){
            double d2 = (double)val[j]/wt[j];
            if(d1<d2){
                temp=j;
            }
        }
        if(temp != i){
            swap = true;
        }
    }
}
```

```
        swap=true;
    }
}
if(swap){
    mid = val[i];
    val[i] = val[temp];
    val[temp] = mid;
    mid = wt[i];
    wt[i] = wt[temp];
    wt[temp] = mid;
}
}
cout << "Maximum value we can obtain = "
    << fractionalKnapsack(W, val,wt, n);
return 0;
}
```

Maximum value can obtain = 145

Practical - 6

Aim: Implementation of chain matrix multiplication using dynamic programming.

```
#include <iostream>
using namespace std;

int MatrixChainOrder(int p[], int n){
    int m[n][n];
    int i, j, k, l, q;
    // main work is below
    for (i = 0; i < n; i++)
        m[i][i] = 0;
    for(l=1;l<n;l++){
        for(j=1;j<n;j++){
            i=j-l;
            m[i][j]=INT_MAX;
            for(k=i;k<j;k++){
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }
    // for printing purpose only
    for(i=1;i<n;i++){
        for(j=1;j<n;j++){
            if(i>j)cout<<"X ";
            else
                cout<<m[i][j]<<" ";
        }cout<<"\n";
    }
    return m[1][n - 1];
}

int main(){
    int arr[] = { 4,10,3,12,20,7};
    int length = sizeof(arr)/sizeof(arr[0]);
    cout<<"Minimum number of multiplications is "<<MatrixChainOrder(arr, length);
    return 0;
}
```

```
0 120 264 1080 1344
X 0 360 1320 1350
X X 0 720 1140
X X X 0 1680
X X X X 0
Minimum number of multiplications is 1344
Process returned 0 (0x0)   execution time : 0.072 s
Press any key to continue.
```

Practical - 7

Aim: Implementation of making a change problem using dynamic programming

```
#include <iostream>
#include <iomanip>
using namespace std;

int min(int a, int b){
    return (a<b)?a:b;
}

int Minimum_number_Of_Coins(int coin[],int ncoin,int nchange){
    int Table[ncoin][nchange + 1];
    int i=ncoin,j=nchange;
    for(i=0;i<ncoin;++i)
        Table[i][0]=0;

    for(j=1;j<=nchange;++j)
        Table[0][j]=(j%coin[0]==0)?j/coin[0]:nchange;

    for(i=1;i<ncoin;++i){
        for(j=1;j<=nchange;++j){
            if(coin[i]>j)
                Table[i][j]=Table[i - 1][j];
            else{
                Table[i][j]= min(Table[i - 1][j],1 + Table[i][j - coin[i]]);
            }
        }
    }

    cout<<" ";
    for(j=0;j<=nchange;++j){
        cout<<setw(4)<<j;
    }
    cout<<"\n";
    for(i=0;i<ncoin;++i){
        cout<<setw(4)<<coin[i];
        for(j=0;j<=nchange;++j){
            cout<<setw(4)<<Table[i][j];
```

```

        }
        cout<<"\n";
    }
    return Table[ncoin - 1][nchange];
}

int Number_Of_Ways_to_Make_Change(int coin[],int ncoin,int nchange){
    int Table[ncoin + 1][nchange + 1];
    int i=ncoin,j=nchange;

    for(i=0;i<=ncoin;++i)
        Table[i][0]=1;

    for(j=1;j<=nchange;++j)
        Table[0][j]=0;

    for(i=1;i<=ncoin;++i){
        for(j=1;j<=nchange;++j){
            if(coin[i - 1]>j)
                Table[i][j]=Table[i - 1][j];
            else{
                Table[i][j]= Table[i - 1][j] + Table[i][j - coin[i - 1]];
            }
        }
    }
    cout<<" ";
    for(j=0;j<=nchange;++j){
        cout<<setw(4)<<j;
    }
    cout<<"\n";
    for(i=0;i<=ncoin;++i){
        if(i>0)cout<<setw(4)<<coin[i-1];
        else cout<<setw(4)<<i;
        for(j=0;j<=nchange;++j){
            cout<<setw(4)<<Table[i][j];
        }
        cout<<"\n";
    }

    return Table[ncoin][nchange];
}

```

```

}

int main(){
//  int arr[] = {1,5,6,9};
  int arr[]={1,2,4};
  int ncoin = sizeof(arr)/sizeof(arr[0]);
//  int nchange = 10;
  int nchange =7;

  cout<<"\n";
  nchange = Minimum_number_Of_Coins(arr,ncoin,nchange);
  cout<<"Minimum number Of Coins required for change is "<<nchange;
  cout<<"\n";
  nchange = Number_Of_Ways_to_Make_Change(arr,ncoin,nchange);
  cout<<"Number Of Ways to Make Change is "<<nchange;
  return 0;
}

```

```

      0  1  2  3  4  5  6  7
1  0  1  2  3  4  5  6  7
2  0  1  1  2  2  3  3  4
4  0  1  1  2  1  2  2  3
Minimum number Of Coins required for change is 3
      0  1  2  3
0  1  0  0  0
1  1  1  1  1
2  1  1  2  2
4  1  1  2  2
Number Of Ways to Make Change is 2

```


Practical - 8

Aim: Implementation of Graph Searching strategy (DFS and BFS).

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
using namespace std;

void addEdge(vector<int> arr[],int a,int b){
    arr[a].push_back(b);
}

void PrintGraph(vector<int> arr[],int n){
    for(int i=0;i<n;++i){
        cout<<"\n Adjacency list of vertex "<<i<<"\n head";
        for(int x:arr[i]){
            cout<<"-> "<<x;
        }
        cout<<"\n";
    }
}

void DFS(vector<int> arr[],int v,int n){
    bool visited[n];

    for(int i=0;i<n;++i)
        visited[i]=false;
    stack<int> stack;
    stack.push(v);

    while (!stack.empty()){
        v = stack.top();
        stack.pop();
        if(!visited[v]){
            cout << v << " ";
            visited[v] = true;
        }
        for (auto i = arr[v].begin(); i != arr[v].end(); ++i)
```

```
        if (!visited[*i])
            stack.push(*i);
    }

}

void BFS(vector<int> arr[],int v,int n){
    bool visited[n];

    for(int i=0;i<n;++i)
        visited[i]=false;

    queue<int> q;
    q.push(v);
    visited[v] = true;

    while(!q.empty()){
        int s = q.front();
        cout<<s<<" ";
        q.pop();
        //vector<int>::iterator i;
        for(int x:arr[s]){
            if (!visited[x]){
                visited[x] = true;
                q.push(x);
            }
        }
    }
    cout<<endl;
}

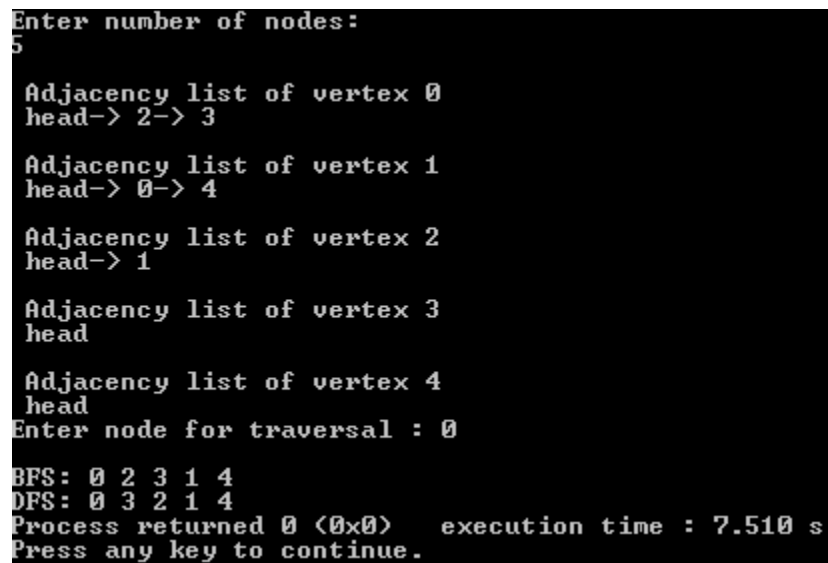
int main(){
    int n,t;
    cout<<"Enter number of nodes: "<<endl;
    cin>>n;

    vector<int> v[n];
    addEdge(v,1, 0);
    addEdge(v,0, 2);
    addEdge(v,2, 1);
```

```
addEdge(v,0, 3);
addEdge(v,1, 4);

PrintGraph(v,n);
cout<<"Enter node for traversal : ";
cin>>t;

cout<<endl;
cout<<"BFS: ";
BFS(v,t,n);
cout<<"DFS: ";
DFS(v,t,n);
return 0;
}
```



```
Enter number of nodes:
5

Adjacency list of vertex 0
head-> 2-> 3

Adjacency list of vertex 1
head-> 0-> 4

Adjacency list of vertex 2
head-> 1

Adjacency list of vertex 3
head

Adjacency list of vertex 4
head
Enter node for traversal : 0

BFS: 0 2 3 1 4
DFS: 0 3 2 1 4
Process returned 0 (0x0)   execution time : 7.510 s
Press any key to continue.
```

Practical – 9

Aim: Implementation of any string matching algorithm.

```
#include <iostream>
#include <cstring>
using namespace std;

int SearchPatt(string t,string p,int []);
void ComputePrefix(string s,int ips[]){
    int i,j,len=s.length();
    ips[0] = 0;
    for(i=1;i<len;++i){
        j = ips[i-1];
        while(j>0 && s[i]!=s[j]) j=ips[j-1];

        if(s[i]==s[j])++j;
        ips[i] = j;
    }
}

int main() {
    string p,t;
    cout<<"Enter Pattern:";
    cin>>p;
    cout<<"Enter String:";
    cin>>t;
    int ips[p.length()];
    ComputePrefix(p,ips);
    cout<<SearchPatt(t,p,ips);
    return 0;
}

int SearchPatt(string t,string p,int ips[]){
    int i=0,j=0,N=t.length(),M=p.length();
    int count=0;
    while(i<N){
        if(p[j]==t[i]){
            ++i;
            ++j;
        }
    }
```

```
        if(j==M){
            ++count;
            cout<<"at index: "<<(i - M)<<endl;
            j = ips[j-1];
        }
        else if(i<N&& p[j]!=t[i]){
            if(j!=0)j=ips[j-1];
            else i=i+1;
        }
    }
    return count;
}
```

```
Enter Pattern:load
Enter String:loldlosbhsybdjlbaokjloajcdlflloadhnjvulosncloseload
at index: 28
at index: 46|
2
```

Practical – 10

AIM: Prepare a detailed write up for various P and NP Problems.

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer 'yes' for a specific input. A language is the totality of inputs for which the answer is yes. Most of the algorithms discussed in the previous chapters are polynomial time algorithms.

For input size n , if worst-case time complexity of an algorithm is $O(n^k)$, where k is a constant, the algorithm is a polynomial time algorithm..

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where k is constant.

These problems are called tractable, while others are called intractable or superpolynomial.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial $p(n)$ such that the algorithm can solve any instance of size n in a time $O(p(n))$.

Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large n . Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of k .

The advantages in considering the class of polynomial-time algorithms is that all reasonable deterministic single processor model of computation can be simulated on each other with at most a polynomial slow-d

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time.

Minimum spanning tree (MST):

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.

There are quite a few use cases for minimum spanning trees. One example would be a telecommunications company trying to lay cable in a new neighborhood. If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths. Some of the paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight – there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, representing the least expensive path for laying the cable.

Matrix Chain Multiplication:

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications. Here are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have:

$$((AB) C)D = (A (BC)) D = (AB) (CD) = A ((BC) D) = A (B (CD)).$$

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product that is the computational complexity. For example, if A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix, then

Computing $(AB) C$ needs $(10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500$ operations, while
Computing $A (BC)$ needs $(30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000$ operations.

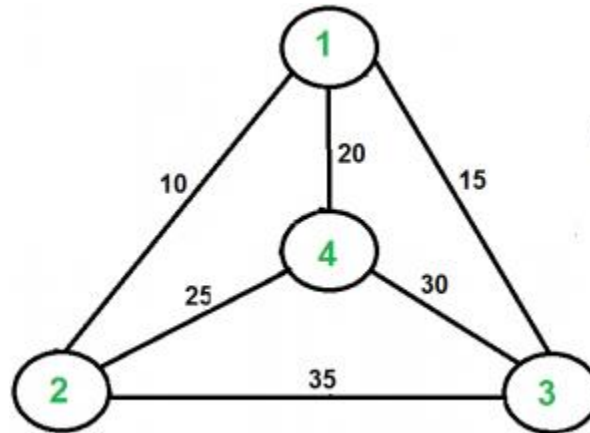
Many problem are fall in this category like sorting, searching etc.

NP-Class

NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).

However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the NP-Complete problems, whose status is unknown.

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Using the above recurrence relation, we can write dynamic programming based solution.



For example, consider the graph shown in figure. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is $10+25+30+15$ which is 80.

The problem is a famous NP hard problem. There is no polynomial time known solution for this problem.

There are at most $O(n \cdot 2^n)$ sub problems, and each one takes linear time to solve. The total running time is therefore $O(n^2 \cdot 2^n)$. The time complexity is much less than $O(n!)$, but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

Hamiltonian Cycle:

A Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not.

Every complete graph with more than two vertices is a Hamiltonian graph. This follows from the definition of a complete graph: an undirected, simple graph such that every pair of nodes is connected by a unique edge.

The graph of every platonic solid is a Hamiltonian graph. So the graph of a cube, a tetrahedron, an octahedron, or an icosahedron are all Hamiltonian graphs with Hamiltonian cycles.

A graph with n vertices (where $n > 3$) is Hamiltonian if the sum of the degrees of every pair of non-adjacent vertices is n or greater. This is known as Ore's theorem.

A search for Hamiltonian cycles isn't just a fun game for the afternoon off. It has real applications in such diverse fields as computer graphics, electronic circuit design, mapping genomes, and operations research.

For instance, when mapping genomes scientists must combine many tiny fragments of genetic code ("reads", they are called), into one single genomic sequence (a 'superstring'). This can be done by finding a Hamiltonian cycle or Hamiltonian path, where each of the reads are considered nodes in a graph and each overlap (place where the end of one read matches the beginning of another) is considered to be an edge.

In a much less complex application of exactly the same math, school districts use Hamiltonian cycles to plan the best route to pick up students from across the district. Here students may be considered nodes, the paths between them edges, and the bus wishes to travel a route that will pass each students house exactly once.

Longest path problem:

In graph theory and theoretical computer science, the longest path problem is the problem of finding a simple path of maximum length in a given graph. A path is called simple if it does not have any repeated vertices; the length of a path may either be measured by its number of edges, or (in weighted graphs) by the sum of the weights of its edges. In contrast to the shortest path problem, which can be solved in polynomial time in graphs without negative-weight cycles, the longest path problem is NP-hard and the decision version of the problem, which asks whether a path exists of at least some given length, is NP-complete. This means that the decision problem cannot be solved in polynomial time for arbitrary graphs unless $P = NP$. Stronger hardness results are also known showing that it is difficult to approximate. However, it has a linear time solution for directed acyclic graphs, which has important applications in finding the critical path in scheduling problems.

Boolean satisfiability problem:

In logic and computer science, the Boolean satisfiability problem (sometimes called propositional satisfiability problem and abbreviated SATISFIABILITY, SAT or B-SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE. If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments and the formula is unsatisfiable. For example, the formula "a AND NOT b" is satisfiable because one can find the values $a = \text{TRUE}$ and $b = \text{FALSE}$, which make $(a \text{ AND NOT } b) = \text{TRUE}$. In contrast, "a AND NOT a" is unsatisfiable.

SAT is the first problem that was proven to be NP-complete; see Cook–Levin theorem. This means that all problems in the complexity class NP, which includes a wide range of natural decision and optimization problems, are at most as difficult to solve as SAT. There is no known algorithm that efficiently solves each SAT problem, and it is generally believed that no such algorithm exists; yet this belief has not been proven mathematically, and resolving the question of whether SAT has a polynomial-time algorithm is equivalent to the P versus NP problem, which is a famous open problem in the theory of computing.