LAB MANUAL

Department of Information Technology

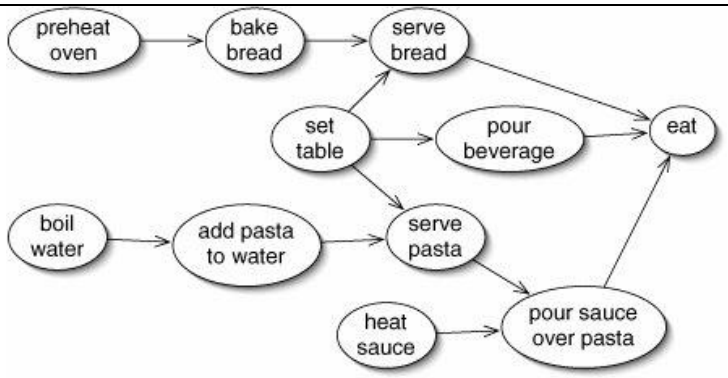
Chandubhai S Patel Institute of Technology

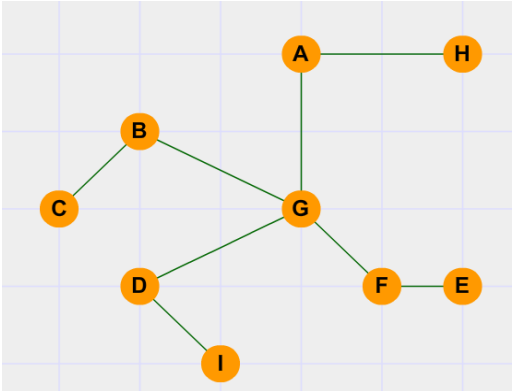
Charotar University of Science and Technology

List of Experiments

<table>
<tr><th>Sr No.</th><th colspan="2">Practical Aim</th><th>Hrs</th></tr>
<tr><td>1</td><td colspan="2">Introduction to profiling in C and Implement and analyze algorithms given below</td><td>02</td></tr>
<tr><td></td><td>1.1</td><td>Fibonacci Series(Iterative and Recursive)</td><td></td></tr>
<tr><td></td><td>1.2</td><td>Factorial of a given number (Iterative and Recursive)</td><td></td></tr>
<tr><td>2</td><td colspan="2">Implement and analyze algorithms given below.(1 Lab for 2.1, 1 Lab for 2.2 & 2.3)</td><td>04</td></tr>
<tr><td></td><td>2.1</td><td>Bubble Sort</td><td></td></tr>
<tr><td></td><td>2.2</td><td>Selection Sort</td><td></td></tr>
<tr><td></td><td>2.3</td><td>Insertion Sort</td><td></td></tr>
<tr><td>3</td><td colspan="2">Implement and analyze algorithms given below.(Divide and Conquer Strategy) (1 Lab for 3.1 & 3.2, 1 Lab for 3.3)</td><td>04</td></tr>
<tr><td></td><td>3.1</td><td>Design and implement searching algorithm to find given word from English dictionary using minimum number of comparisons. Also find out time complexity of algorithm.</td><td></td></tr>
<tr><td></td><td>3.2</td><td>Merge Sort</td><td></td></tr>
<tr><td></td><td>3.3</td><td>Quick Sort</td><td></td></tr>
<tr><td>4</td><td colspan="2">Implement and analyze any one (Greedy Approach)</td><td>02</td></tr>
<tr><td></td><td>4.1</td><td>Suppose there are two persons A & B. For Given amount N, If person A wants change for N Rupees, and suppose the person B having infinite number of coin for each value of C, where C={c1,c2,c3,c4,c5}. Person A wants minimum number of coins from Person B for the amount N.
Design and implement an algorithm to minimize the number of coins to make up the given amount.</td><td></td></tr>
<tr><td></td><td>4.2</td><td>A Burglar has just broken into the Fort! He sees himself in a room with n piles of gold dust. Because the each pile has a different purity, each pile also has a different value (v[i]) and a different weight (w[i]). A Burglar has a bag that can only hold W kilograms.
Given n number of piles, v={v1,v2,v3,…vn}, w={w1,w2,w3,….wn} and capacity of bag W.
Design and implement an algorithm to get maximum piles of gold using given bag with W capacity, Burglar is also allowed to take fractional of pile.</td><td></td></tr>
<tr><td>5</td><td colspan="2">Design & Implement given problems (Greedy Approach)</td><td>04</td></tr>
<tr><td></td><td>5.1</td><td>There are eleven Professors in a Department. Each professor wants to deliver lecture in same day. Each professor has some time limits for lecture. Professor earns credit if and only if lecture is arranged on or before its time limit.
<table>
<tr><td>Professor</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td></tr>
<tr><td>credit</td><td>78</td><td>90</td><td>50</td><td>60</td><td>75</td><td>10</td><td>80</td><td>55</td><td>88</td><td>74</td><td>59</td></tr>
<tr><td>Lecture Limit</td><td>5</td><td>4</td><td>5</td><td>3</td><td>2</td><td>1</td><td>4</td><td>6</td><td>4</td><td>5</td><td>6</td></tr>
</table>
Design and implement greedy approach to schedule maximum number of lectures in the department without and with credit.</td><td></td></tr>
<tr><td></td><td>5.2</td><td>Design LAN topology using distributed computer and communication networks, wiring</td><td></td></tr>
</table>

		connections, transportation networks among cities, and designing pipe capacities in flow networks. It is intended to network five computers at a large theme park. There is one computer in the office and one at each of the four different entrances. Cables need to be laid to link the computers. Cable laying is expensive, so a minimum length of cable is required.	
6	**Implement and analyze given problems (Dynamic Programming)**		**04**
	6.1	Given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).	
	6.2	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.	
7	**Implement and analyze given problem Any one(Dynamic Programming)**		**02**
	7.1	Find the minimum of characters to be inserted to convert it into palindrome.	
	7.2	Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown.	
8	**String Matching**		**02**
	8.1	Suppose you are given a source string S[0 ..n − 1] of length n, consisting of symbols a and b. Suppose further that you are given a pattern string P[0 ..m − 1] of length m < n, consisting of symbols a, b, and *, representing a pattern to be found in string S. The symbol * is a "wild card" symbol, which matches a single symbol, either a or b. The other symbols must match exactly. The problem is to output a sorted list M of valid "match positions", which are positions j in S such that pattern P matches the substring S[j..j + |P|− 1]. For example, if S = ababbab and P = ab*, then the output M should be [0, 2]. Implement Naive and Rabin karp algorithm to solve the problem.	
9	**Implement and analyze the problem**		**02**
	9.1	Eight Queen Problem	
10	**Design and analyze any two (Graph)**		**04**
	10.1	Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices. Solve this using Backtracking.	
	10.2	Dinner involves a number of different tasks, shown in this directed graph. An edge indicates that one task must be performed before another. For example, the oven must be preheated before the bread can be baked. It should be noted that only a programmer would consider this a complete meal.	

<table>
<tr>
<td></td>
<td></td>
<td> 

Implement this problem to generate all the possibilites.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.3</td>
<td>Let us consider the situation of a war. In your country, there is a network of telephone lines between 9 cities (A, B, C...H, I) i.e. the 9 cities are connected by telephone line, which means that a message from one city to any other city can be transmitted through the line. Like we can transfer message from city A to city B even though they are not "directly" connected by a line. So what's the catch, everything seems fine, right?
You are the "army-general" of your country and you've to take a decision, you have to find the city which, if damaged would incur the greatest network blockage (Considering that damaging the city damages all the connected telephone lines in it).
Implement this situation and find out Which city would you try to protect the most and why?
 
</td>
<td></td>
</tr>
</table>

PROCEDURE TO FOLLOW FOR ALL EXPERIMENTS

1. Analyze the problem using Running Time
    1. Implement a problem in any language.
    2. Measure the running time of program for at least different 5 input and make a table of Input Size Vs Running Time.
    3. Draw the graph of Input size Vs Running Time.
    4. Repeat step 2 and 3 for Best Case, Average Case and Worst Case.
    5. Compare the practical complexity with theoretical complexity.
    6. Conclude from above graph or data table.

2. Analyze the problem using No of Instructions
    1. Implement a problem in any language.
    2. Count the no of instruction of the program for at least different 5 input and make a table of Input Size Vs No of Instructions.
    3. Draw the graph of Input size Vs No of Instructions.
    4. Repeat step 2 and 3 for Best Case, Average Case and Worst Case.
    5. Compare the practical complexity with theoretical complexity.
    6. Conclude from above graph or data table.

Common Guidelines for Analysis:

1. Do not assign any counter for scanf and printf statement unless instructed explicitly

2 Do not assign any counter for variable declaration

3 Assign a counter for variable initialization

4 Assign Counter for arithmetic operations like ADD,SUB,ASSIGN etc

5 Do not assign counter for clrscr and getch functions

6 Assign counter for if conditions

7 Assign a counter before starting of for loop for initilization of variable eg i=0

8 Assign a counter immediately in for loop for checking condition i<n

9 Assign a counter just before closing of for loop for increment of a variable i++

10 Assign 1 counter immediately after exit of for loop for false condition

11 Assign one counter for return statement just before return

12 Assign one counter for function call

13 For composite conditions assign counters based on no of conditions

EXPERIMENT 1

Aim: Introduction to profiling in C and Implement and analyze algorithms given below:

1. Fibonacci Series(Iterative and Recursive)
2. GCD (Iterative and Recursive)
3. Factorial of a given number (Iterative and Recursive)
4. Matrix Addition
5. Matrix Multiplication

1. Objectives:

- To learn how to measure the time of program
- To learn how to count the no of instructions of the program
- To learn how to do analysis of program

2. Background Information:

2.1. Profiling in C:

There are different functions used to measure the time of running program respective to language.

In Turbo C for windows, there is a code block given here which is used to measure the time of running program.

```
clock_t start = clock();
abs(); //function for which we want to measure the time
clock_t end = clock();
double elapsed_time = (end - start)/(double)CLOCKS_PER_SEC;
printf("Elapsed time: %.2f.\n", elapsed_time);
```

Here clock estimates the CPU time used by your program; that's the time the CPU has been busy executing instructions belonging to your program. So we just have to take the difference between start and finish time of our program. Dividing by CLOCKs_PER_SEC is used to convert the time into seconds.

In GCC for linux, Compile the code using the -pg option to include the profiler code in the executable, as follows:

```
gcc -pg gprof_test_code_svp.c
```

This will generate an a.out executable file. You can specify another output filename with -o. Now run the binary with ./a.out. After executing the code, the following output will be generated:

Flat-profile generation

After successful execution of the program, it will generate the file gmon.out. Use Gprof on this file to generate the graph:

```
gprof -p a.out gmon.out

Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative self self total
 time seconds seconds calls us/call us/call name
 97.98 0.97 0.97 9601 101.03 101.03 func3
  2.02 0.99 0.02 399 50.13 50.13 func2
  0.00 0.99 0.00 1197 0.00 0.00 filegen
  0.00 0.99 0.00 399 0.00 50.13 func1
<output snipped>
```

This generates flat-profile analytics, where time calls are given for comparison. Indications from this profile are:

The program took 1.46 minutes of clock time to complete. From the flat-profile, func3 consumed 0.97 seconds per call and was called 9601 times; around 101 milliseconds are spent in this function for every call to func3. The same values in the self us/call and total us/call field indicate that the complete time was spent on the function's own operations, and not on the children functions.

In func1, which was the last entry presented in the profile, there is practically no time spent in itself (self us/call) but as it calls func2, most of its time is spent there. Hence, the total us/call field shows 50.13 milliseconds — the same as that of func2.

In func2, the self us/call and total us/call field are the same — the time is spent in doing self operations.

There is practically no time spent in the filegen function according to Gprof, as the md5sum is a system call, which is not presented in Gprof.

So, using this, we can clearly look for bottlenecks present in the code consuming CPU time.

Call-graph profile generation

To generate a call-graph profile of the functions used in the program, use gprof with the qswitch:

```
gprof -q a.out gmon.out
```

Call graph (explanation follows)…

```
granularity: each sample hit covers 4 byte(s) for 1.01% of 0.99 seconds

index % time self children called name
                                                 <spontaneous>
[1]    100.0 0.00 0.99 main [1]
                0.97 0.00 9601/9601 func3 [2]
                0.00 0.02 399/399 func1 [3]
-----------------------------------------------
                0.97 0.00 9601/9601 main [1]
```

```
[2]     98.0 0.97 0.00 9601 func3 [2]
-----------------------------------------------
                0.00 0.02 399/399 main [1]
[3]      2.0 0.00 0.02 399 func1 [3]
                0.02 0.00 399/399 func2 [4]
-----------------------------------------------
                0.02 0.00 399/399 func1 [3]
[4]      2.0 0.02 0.00 399 func2 [4]
                0.00 0.00 1197/1197 filegen [5]
-----------------------------------------------
                0.00 0.00 1197/1197 func2 [4]
[5]      0.0 0.00 0.00 1197 filegen [5]
-----------------------------------------------

<Output Snipped>
```

Indications from call-graph are:

- The number after the name indicates the index numbers. So we can see that main spends 0.99 seconds in the children waiting for func3 to complete, which is in the children field. This is substantiated by the 0.97 seconds spent per call in the self field of func3. This is again followed up by func1.
- Every entry has the function name in the middle of it with the same index number; i.e., main with value 1 in index 1 field. So you can easily see the cycle for it, in Index 1 field kernel operations calling main, then main calling func3 and so on. The same follows for the rest of the index.
- In Index 3, func1 is called by main with 399 calls; spent 0.02 time in children process (this is substantiated by 0.02 in func2 call). func3 is directly executed from main.
- The middle line in each section is referenced by a unique index number. Above the line indicates originating function for the function given in index number and below the line indicates the process call forwarded to the next function. For example, in index 3 section, the call originated from main() function to func1() and then to func2(). Here func1() has index number 3.
- <spontaneous> indicates that the parent function cannot be determined, generally due to system-related APIs.
- A cumulative time recording implies that 98 per cent of the total time was spent in performing func3 operations, as per index 2.

Function calls substantiate that the filegen() function is called thrice for a single call offunc2(). func1() and func2() are called for the same time; the counter values present in the code prove this. Counters are displayed after successful execution of the code. func1() andfunc2() functions comprise the same timings for calculations. func3 is called for 9601 times. The call-graph gives the call-tree data.

2.2. Fibonacci series:

1. Iterative Approach:

```
int iterative_fib(int n) {
```

```
  if (n <= 2) {
    return 1;
  }
  int a = 1, int b = 1, c;
  for (int i = 0; i < n - 2; ++i) {
    c = a + b;
    b = a;
    a = c;
  }
  return a;
}
```

2. Recursive Approach

```
int recursive_fib(int n) {
  if (n <= 2) {
    return 1;
  }
  return recursive_fib(n - 1) + recursive_fib(n-2);
}
```

Data Table:

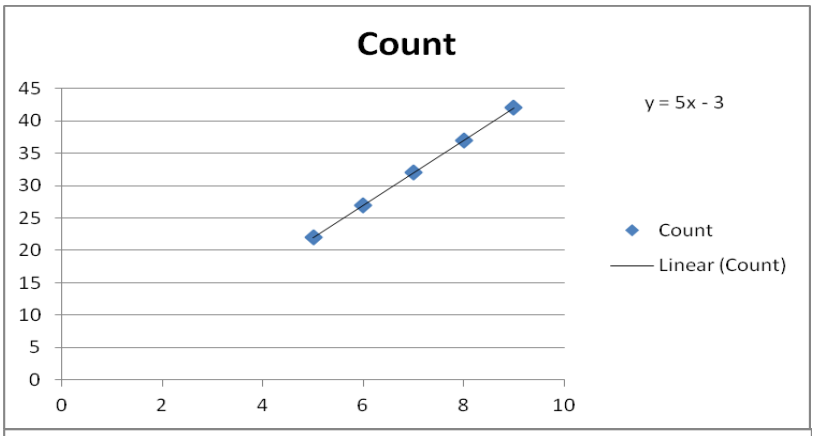
Iterative Approach

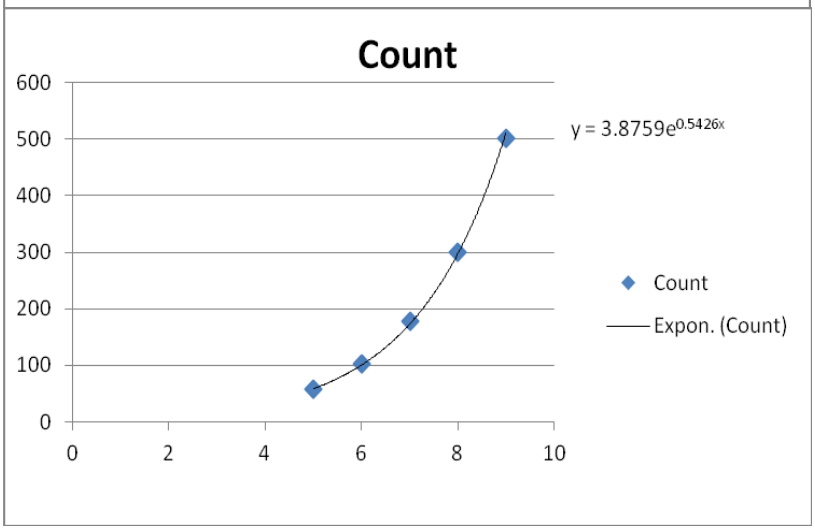
Input	Count
5	22
6	27
7	32
8	37
9	42

Recursive Approach

Input	Count
5	57
6	102
7	177
8	300
9	501

Analysis:





Conclusion:

Iterative approach is better than Recursive Approach for Fibonacci Series

2.3. GCD:

1. Iterative Approach:

```
int gcd ( int a, int b )
{
  int c;
  while ( a != 0 ) {
     c = a; a = b%a;  b = c;
  }
  return b;
}
```

2. Recursive Approach:

```
int gcdr ( int a, int b )
{
  if ( a==0 ) return b;
  return gcdr ( b%a, a );
}
```

Data Table:

Analysis:

Conclusion:

2.4. Factorial of a given number:

1. Iterative Approach

```
unsigned int iter_factorial(unsigned int n)
{
    unsigned int f = 1;
    for(unsigned int i = 1; i <= n; i++)
    {
        f *= i;
    }
    return f;
}
```

2. Recursive Approach

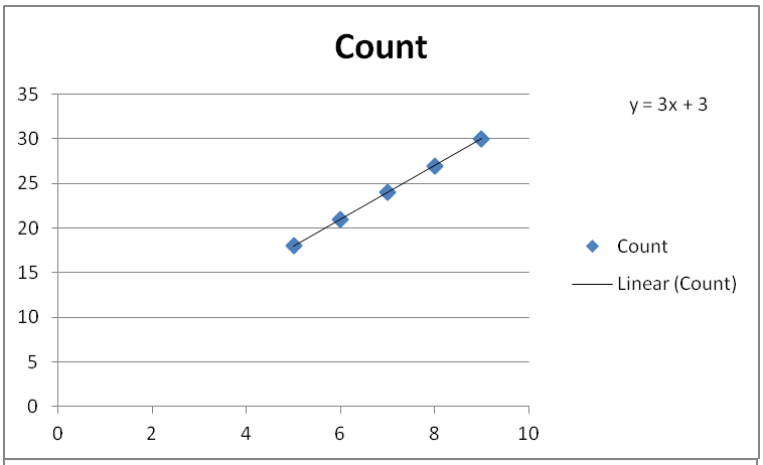
```
unsigned int recursive_factorial(unsigned int n)
{
    return n >= 1 ? n * recr_factorial(n-1) : 1;
}
```

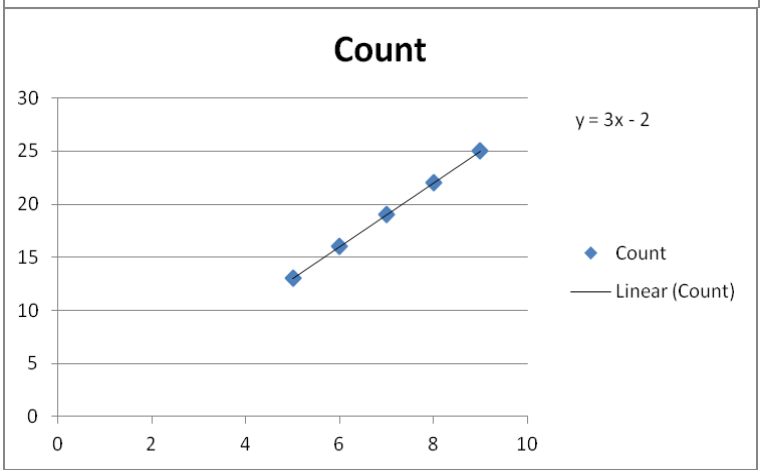
Data Table:

Iterative Approach	
Input	Count
5	18
6	21
7	24
8	27
9	30

Iterative Approach	
Input	Count
5	13
6	16
7	19
8	22
9	25

Analysis:





Conclusion:

2.5. Matrix Addition:

Algorithm

Input Two matrices a and b

Output Output matrix ccontaining elements after addition of a and b

complexity O(n^2)

Matrix-Addition(a,b)

1 for i =1 to rows [a]

2 for j =1 to columns[a]

3 Input a[i,j];

4 Input b[i,j];

5 C[i, j] = A[i, j] + B[i, j];

6 Display C[i,j];

Program

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

```
void main()
{
int i,j,a[10][10],b[10][10],c[10][10],m1,n1,m2,n2;

/* m - Number of rows
   n - Number of Columns */

clrscr();

printf("nEnter the number of Rows of Mat1 : ");
scanf ("%d",&m1);

printf("nEnter the number of Columns of Mat1 : ");
scanf ("%d",&n1);

/* Accept the Elements in m x n Matrix */

for(i=0;i<m1;i++)
    for(j=0;j<n1;j++)
    {
    printf("Enter the Element a[%d][%d] : ",i,j);
    scanf("%d",&a[i][j]);
    }

// ----------------------------------------

printf("nEnter the number of Rows of Mat2 : ");
scanf ("%d",&m2);

printf("nEnter the number of Columns of Mat2 : ");
scanf ("%d",&n2);

/* Before accepting the Elements Check if no of
   rows and columns of both matrices is equal */

if ( m1 != m2 || n1 != n2 )
 {
 printf("nOrder of two matrices is not same ");
 exit(0);
 }

 // ------ Terminate Program if Orders are unequal
 // ------ exit(0) : 0 for normal Termination

/* Accept the Elements in m x n Matrix */
```

```
for(i=0;i<m2;i++)
    for(j=0;j<n2;j++)
    {
    printf("Enter the Element b[%d][%d] : ",i,j);
    scanf("%d",&b[i][j]);
    }

// ----------------------------------------

/* Addition of two matrices */

for(i=0;i<m1;i++)
    for(j=0;j<n1;j++)
    {
    c[i][j] = a[i][j] + b[i][j] ;
    }

/* Print out the Resultant Matrix */

printf("nThe Addition of two Matrices is : n");

for(i=0;i<m1;i++)
  {
    for(j=0;j<n1;j++ )
    {
    printf("%dt",c[i][j]);
    }
  printf("n");
  }

getch();
}
```

Data Table:

Analysis:

Conclusion:

2.6. Matrix Multiplication:

3. Input Two matrices a and b
4. Output Output matrix c containing elements after addition of a and b
5. complexity O(n^3)
6.
7. Matrix-Addition(a,b)

8. Input a[i,j];
9. Input b[i,j];
10. 1 for i =1 to rows [a]
11. 2 for j =1 to columns[a]
12. 3. for k = 1 to columns[b]
13. 4 C[i, j]+ = A[i, j] * B[i, j];
14. 5 Display C[i,j];

```
# include "stdio.h"
main()
{
   int m1[10][10],i,j,k,m2[10][10],mult[10][10],r1,c1,r2,c2;
   printf("Enter number of rows and columns of first matrix (less than 10)\n");
   scanf("%d%d",&r1,&c1);
   printf("Enter number of rows and columns of second matrix (less than 10)\n");
   scanf("%d%d",&r2,&c2);
   if(r2==c1)
   {
      printf("Enter rows and columns of First matrix \n");
      printf("Row wise\n");
      for(i=0;i<r1;i++)
         for(j=0;j<c1;j++)
            scanf("%d",&m1[i][j]);
      printf("First Matrix is :\n");
      for(i=0;i<r1;i++)
      {
         for(j=0;j<c1;j++)
            printf("%d\t",m1[i][j]);
         printf("\n");
      }
      printf("Enter rows and columns of Second matrix \n");
      printf("Row wise\n");
      for(i=0;i<r2;i++)
         for(j=0;j<c2;j++)
            scanf("%d",&m2[i][j]);
      printf("Second Matrix is:\n");
      for(i=0;i<r2;i++)
      {
         for(j=0;j<c2;j++)
            printf("%d\t",m2[i][j]);
         printf("\n");
      }
      printf("Multiplication of the Matrices:\n");
      for(i=0;i<r1;i++)
      {
         for(j=0;j<c2;j++)
         {
            mult[i][j]=0;
            for(k=0;k<r1;k++)
               mult[i][j]+=m1[i][k]*m2[k][j];
```

```
                printf("%d\t",mult[i][j]);
            }
            printf("\n");
        }
    }
    else
    {
        printf("Matrix multiplication cannot be done");
    }
    return 0;
}
```

Data Table:

Analysis:

Conclusion:

EXPERIMENT 2

Aim: Implement and analyze algorithms given below:

1. Insertion Sort
2. Bubble Sort
3. Selection Sort

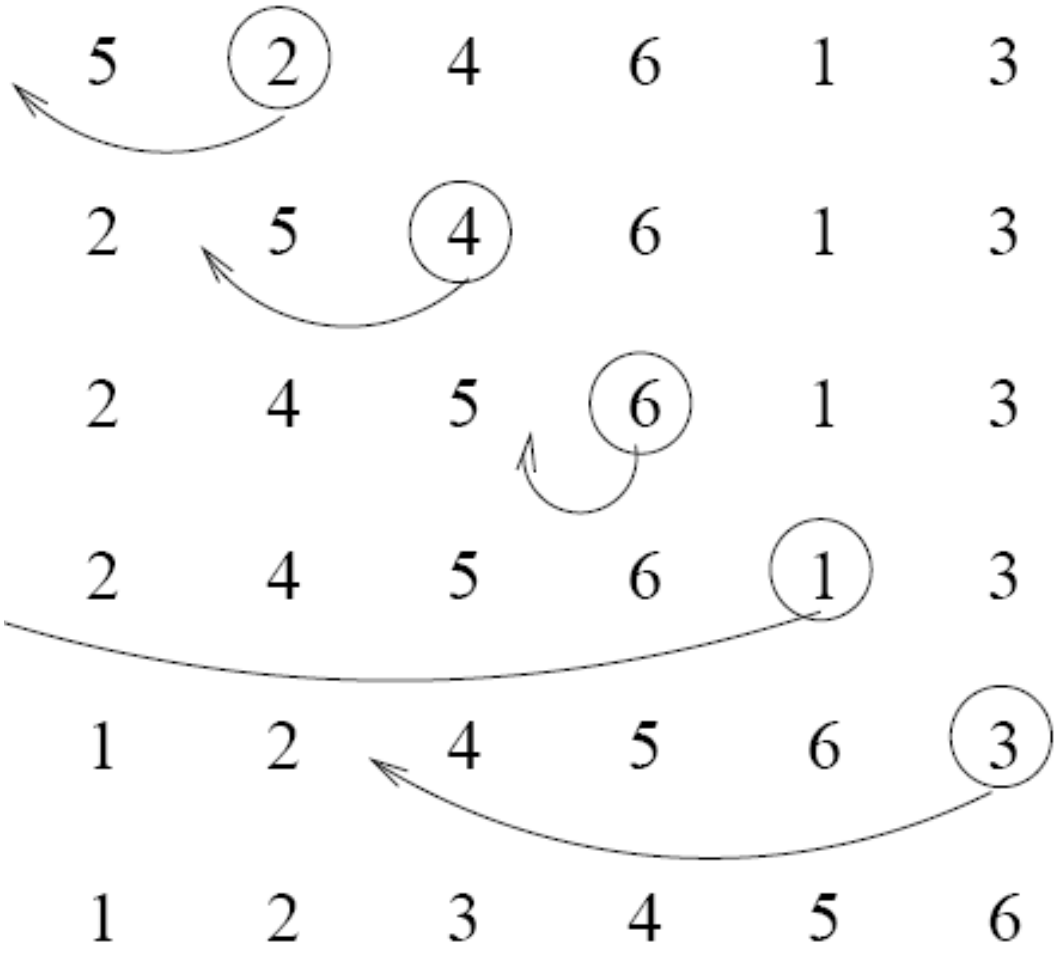
2.1 Insertion Sort

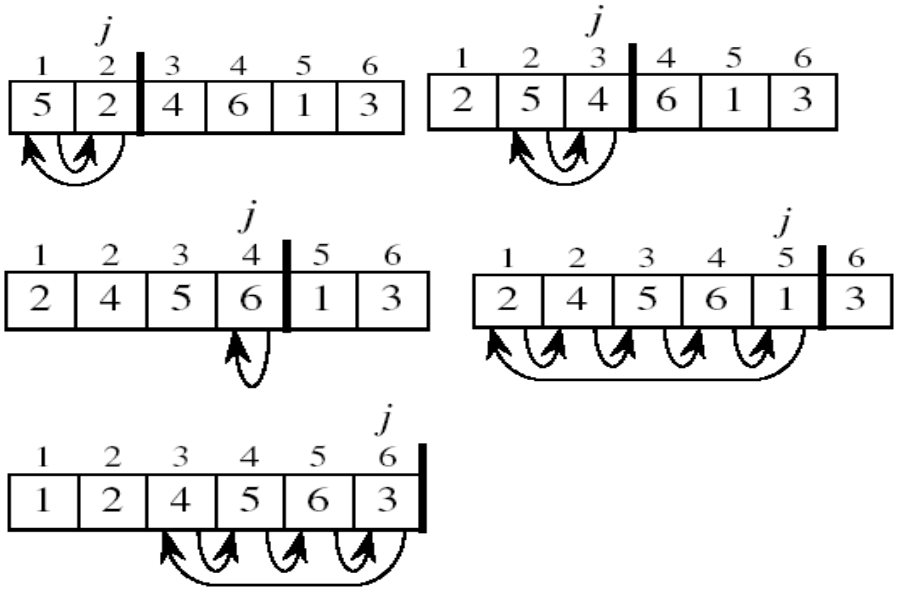
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Alg.: INSERTION-SORT*(A)*

```
for j ← 2 to n
  do key ← A[ j ]
      Insert A[ j ] into the sorted sequence A[1 . . j -1]
     i ← j - 1
     while i > 0 and A[i] > key
   do A[i + 1] ← A[i]
       i ← i – 1
     A[i + 1] ← key
```

- Insertion sort – sorts the elements in place





Best case: the inner loop is never executed
Worst case: the inner loop is executed exactly j − 1 times for every iteration of the outer loop

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 $\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and is executed n times will contribute $c_i n$ to the total running time.[5] To compute $T(n)$, the running time of INSERTION-SORT, we sum the products of the *cost* and *times* columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1) .$$

Even for inputs of a given size, an algorithm's running time may depend on *which* input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each $j = 2, 3, \ldots, n$, we then find that $A[i] \leq key$ in line 5 when i has its initial value of $j - 1$. Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

This running time can be expressed as $an + b$ for *constants* a and b that depend on the statement costs c_i; it is thus a ***linear function*** of n.

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \ldots j-1]$, and so $t_j = j$ for $j = 2, 3, \ldots, n$. Noting that

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

(see Appendix A for a review of how to solve these summations), we find that in the worst case, the running time of INSERTION-SORT is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

This worst-case running time can be expressed as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i; it is thus a ***quadratic function*** of n.

```
#include<stdio.h>
#include<conio.h>
void insertion(int [], int );
int main()
```

```
{
 int arr[30];
 int i,size;
 printf("\n\t------- Insertion sorting
using function -------\n\n");
 printf("Enter total no. of elements :
");
 scanf("%d",&size);
 for(i=0; i<size; i++)
 {
    printf("Enter %d element : ",i+1);
    scanf("%d",&arr[i]);
 }
 insertion(arr,size);
 printf("\n\t------- Insertion sorted
elements using function -------\n\n");
 for(i=0; i<size; i++)
    printf(" %d",arr[i]);
 getch();
 return 0;
}
void insertion(int arr[], int size)
{
 int i,j,tmp;
 for(i=0; i<size; i++)
 {
   for(j=i-1; j>=0; j--)
   {
    if(arr[j]>arr[j+1])
    {
      tmp=arr[j];
```

```
            arr[j]=arr[j+1];
            arr[j+1]=tmp;
          }
          else
            break;
        }
    }
}
```

Data Table:

Analysis:

Conclusion:

2.2 Bubble Sort

Alg.: BUBBLESORT(A)
for i ← 1 **to** length[A]
do for j ← length[A] **downto** i + 1
do if A[j] < A[j -1]
then exchange A[j] ↔ A[j-1]

Bubble-Sort Running Time

Alg.: BUBBLESORT(A)

for i ← 1 to length[A] c_1

do for j ← length[A] downto i + 1 c_2

Comparisons: ≈ $n^2/2$ do if A[j] < A[j -1] c_3

Exchanges: ≈ $n^2/2$ then exchange A[j] ↔ A[j-1] c_4

$$T(n) = c_1(n+1) + c_2\sum_{i=1}^{n}(n-i+1) + c_3\sum_{i=1}^{n}(n-i) + c_4\sum_{i=1}^{n}(n-i)$$

$$= \Theta(n) + (c_2 + c_2 + c_4)\sum_{i=1}^{n}(n-i)$$

$$\text{where } \sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n}n - \sum_{i=1}^{n}i = n^2 - \frac{n(n+1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

Thus, $T(n) = \Theta(n^2)$

27

```c
#include <stdio.h>

void bubble_sort(long [], long);

int main()
{
  long array[100], n, c, d, swap;

  printf("Enter number of elements\n");
  scanf("%ld", &n);

  printf("Enter %ld integers\n", n);

  for (c = 0; c < n; c++)
    scanf("%ld", &array[c]);

  bubble_sort(array, n);

  printf("Sorted list in ascending order:\n");

  for ( c = 0 ; c < n ; c++ )
     printf("%ld\n", array[c]);

  return 0;
```

```
}

void bubble_sort(long list[], long n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        /* Swapping */

        t          = list[d];
        list[d]    = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```

Data Table:

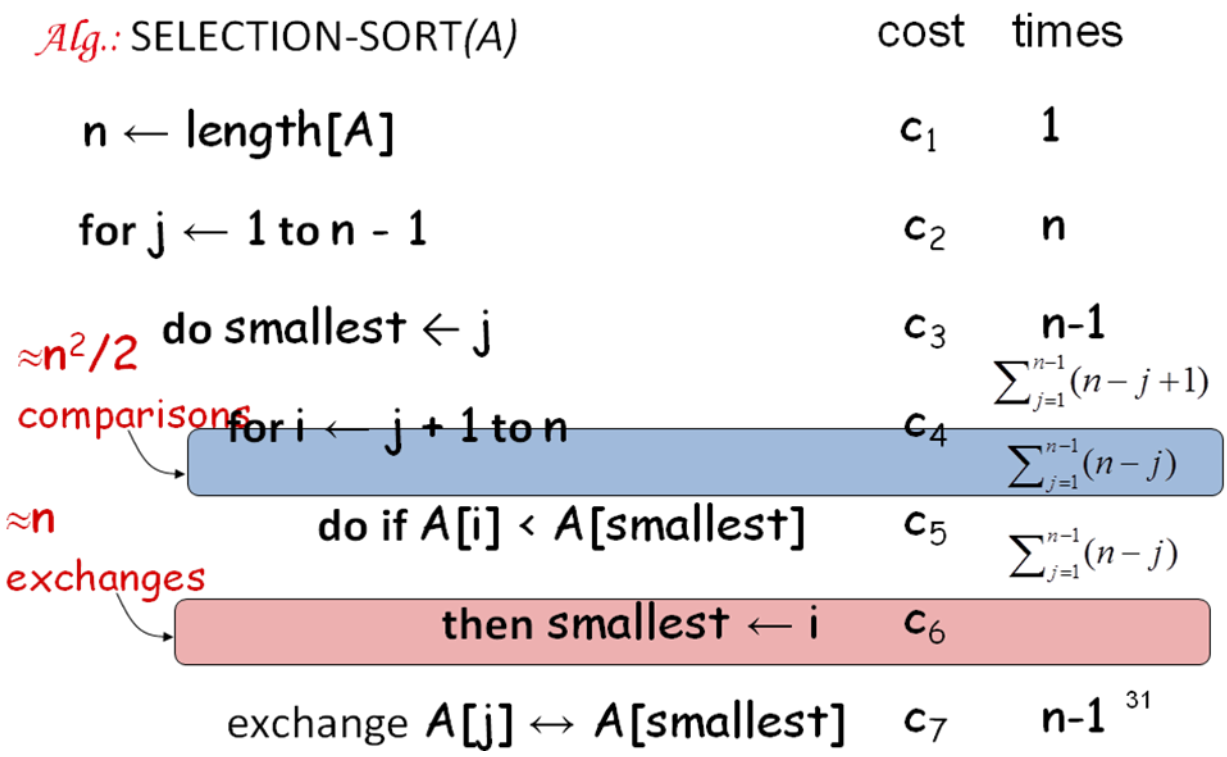
Analysis:

Conclusion:

2.3 Selection Sort

Alg.: SELECTION-SORT*(A)*
n ← length[A]
for j ← 1 **to** n - 1
do smallest ← j
for i ← j + 1 **to** n
do if A[i] < A[smallest]
then smallest ← i
exchange A[j] ↔ A[smallest]

Analysis of Selection Sort



31

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4 \sum_{j=1}^{n-1}(n - j + 1) + c_5 \sum_{j=1}^{n-1}(n - j) + c_6 \sum_{j=2}^{n-1}(n - j) + c_7(n-1) = \Theta(n^2)$$

```
#include<stdio.h>
#include<conio.h>
void select(int n,int a[])
{
 int i=0,j=0,t=0,k=0;
 for (i=1;i<n;i++)
 {
  t=a[i];
  for(j=i-1;((j>=0)&&(t<a[j]));j--)
```

```
 a[j+1]=a[j];
 a[j+1]=t;
}
printf("\n\nThe sorted list is : ");
for(j=0;j<n;j++)
printf("%d ",a[j]);
return 0;
getch();

}
void main()
{
int n,i=0,a[30];
clrscr();
printf("\nEnter how many numbers you want to sort\n");
scanf("%d",&n);
printf("\nEnter the numbers \n");
for (i=0;i<n;i++)
{
 scanf("%d",&a[i]);
}
select(n,a);
getch();

}
```

Data Table:

Analysis:

Conclusion:

EXPERIMENT 3

Aim: Implement and analyze algorithms given below.(Divide and Conquer Strategy)

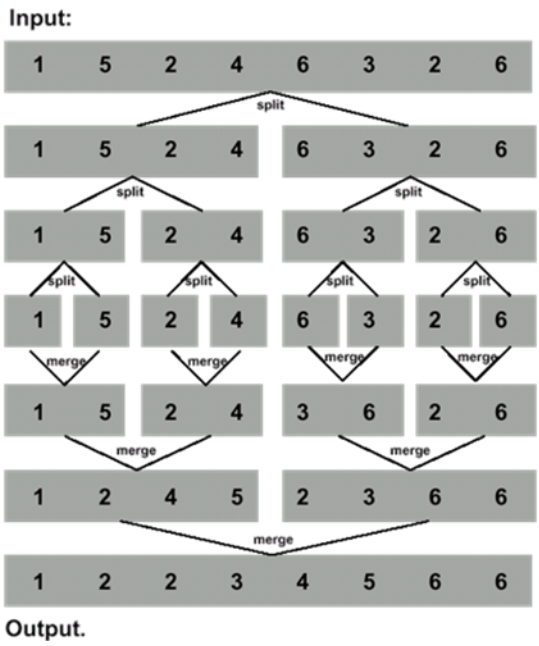
1. Merge Sort
2. Quick Sort
3. Binary Search

3.1 Merge Sort

Merge-Sort(A, p, r)
  if p < r **then**
    q¬(p+r)/2
    Merge-Sort(A, p, q)
    Merge-Sort(A, q+1, r)
    Merge(A, p, q, r)
Merge(A, p, q, r)

Take the smallest of the two topmost elements of sequences A[p..q] and A[q+1..r] and put into the resulting sequence. Repeat this, until both sequences are empty. Copy the resulting sequence into A[p..r].

- To sort *n* numbers
  - if n=1 done!
  - recursively sort 2 lists of numbers *n*/2 and *n*/2 elements
  - merge 2 sorted lists in Θ(*n*) time
- Strategy
  - break problem into similar (smaller) subproblems
  - recursively solve subproblems
  - combine solutions to answer



Use a ***recurrence equation*** (more commonly, a ***recurrence***) to describe the running time of a divide-and-conquer algorithm.

Let $T(n)$ = running time on a problem of size n.

- If the problem size is small enough (say, $n \leq c$ for some constant c), we have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
- Let the time to divide a size-n problem be $D(n)$.
- There are a subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve $\Rightarrow$ we spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \text{ ,} \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

For simplicity, assume that n is a power of $2 \Rightarrow$ each divide step yields two sub-problems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

Divide: Just compute q as the average of p and $r \Rightarrow D(n) = \Theta(1)$.

Conquer: Recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$.

Combine: MERGE on an n-element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

```
#include<stdio.h>
#define MAX 50

void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);

int main(){

    int merge[MAX],i,n;

    printf("Enter the total number of elements: ");
    scanf("%d",&n);

    printf("Enter the elements which to be sort: ");
    for(i=0;i<n;i++){
         scanf("%d",&merge[i]);
    }

    partition(merge,0,n-1);

    printf("After merge sorting elements are: ");
    for(i=0;i<n;i++){
         printf("%d ",merge[i]);
    }
```

```
    return 0;
}

void partition(int arr[],int low,int high){

     int mid;

     if(low<high){
          mid=(low+high)/2;
          partition(arr,low,mid);
          partition(arr,mid+1,high);
          mergeSort(arr,low,mid,high);
     }
}

void mergeSort(int arr[],int low,int mid,int high){

     int i,m,k,l,temp[MAX];

     l=low;
     i=low;
     m=mid+1;

     while((l<=mid)&&(m<=high)){

          if(arr[l]<=arr[m]){
               temp[i]=arr[l];
               l++;
          }
          else{
               temp[i]=arr[m];
               m++;
          }
          i++;
     }

     if(l>mid){
          for(k=m;k<=high;k++){
               temp[i]=arr[k];
               i++;
```

```
            }
        }
        else{
            for(k=l;k<=mid;k++){
                temp[i]=arr[k];
                i++;
            }
        }

        for(k=low;k<=high;k++){
            arr[k]=temp[k];
        }
}
```

3.2 Quick Sort

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

Quicksort is based on the three-step process of divide-and-conquer.

- To sort the subarray $A[p..r]$:

  Divide: Partition $A[p..r]$, into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$, such that each element in the first subarray $A[p..q-1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray $A[q+1..r]$.

  Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

  Combine: No work is needed to combine the subarrays, because they are sorted in place.

- Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

□

```
QUICKSORT(A, p, r)
if p < r
  then q ← PARTITION(A, p, r)
       QUICKSORT(A, p, q − 1)
       QUICKSORT(A, q + 1, r)

Initial call is QUICKSORT(A, 1, n).
```

Partitioning

Partition subarray $A[p\,..\,r]$ by the following procedure:

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
    do if A[j] ≤ x
          then i ← i + 1
               exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then quicksort can run as fast as mergesort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

Best case

- Occurs when the subarrays are completely balanced every time.
- Each subarray has $\leq n/2$ elements.
- Get the recurrence

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

Worst case

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and $n - 1$ elements in the other subarray.
- Get the recurrence

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

- Same running time as insertion sort.
- In fact, the worst-case running time occurs when quicksort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Balanced partitioning

- Quicksort's average running time is much closer to the best case than to the worst case.
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence

$$T(n) \leq T(9n/10) + T(n/10) + \Theta(n)$$
$$= O(n \lg n).$$

```
#include<stdio.h>

void quicksort(int [10],int,int);

int main(){
  int x[20],size,i;

  printf("Enter size of the array: ");
  scanf("%d",&size);

  printf("Enter %d elements: ",size);
  for(i=0;i<size;i++)
    scanf("%d",&x[i]);

  quicksort(x,0,size-1);

  printf("Sorted elements: ");
  for(i=0;i<size;i++)
    printf(" %d",x[i]);

  return 0;
}
```

```
void quicksort(int x[10],int first,int last){
    int pivot,j,temp,i;

     if(first<last){
         pivot=first;
         i=first;
         j=last;

         while(i<j){
             while(x[i]<=x[pivot]&&i<last)
                 i++;
             while(x[j]>x[pivot])
                 j--;
             if(i<j){
                 temp=x[i];
                  x[i]=x[j];
                  x[j]=temp;
             }
         }

         temp=x[pivot];
         x[pivot]=x[j];
         x[j]=temp;
         quicksort(x,first,j-1);
         quicksort(x,j+1,last);

    }
}
```

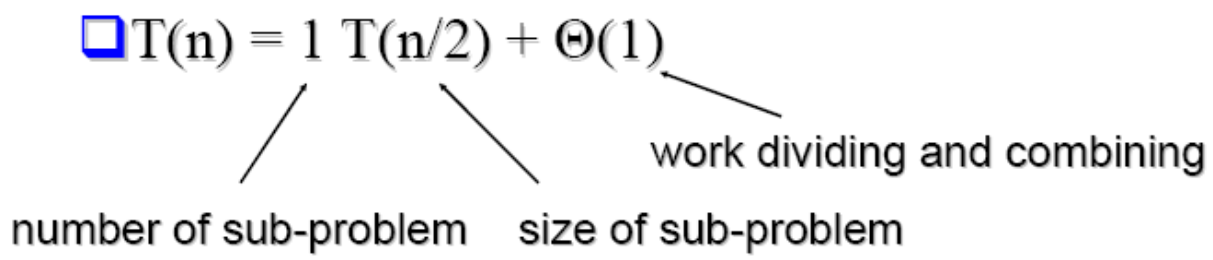
3.3 Binary Search

Problem: Determine whether x is in the sorted array S of size n.

Inputs: positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.

Outputs: *location*, the location of x in S (0 if x is not in S).

```
index location (index low, index high)
   {
  index mid;
  if (low > high)
       return 0;
  else {
       mid = [(low + high)/2];
       if (x == S[mid])
            return mid
       else if (x == S[mid])
            return location(low, mid - 1);
       else
            return location(mid + 1, high);
     }
   }
```



$$T(n) = \Theta(\lg n)$$

```
#include<stdio.h>
int main(){

    int a[10],i,n,m,c,l,u;

    printf("Enter the size of an array: ");
    scanf("%d",&n);
```

```
    printf("Enter the elements of the array: " );
    for(i=0;i<n;i++){
         scanf("%d",&a[i]);
    }

    printf("Enter the number to be search: ");
    scanf("%d",&m);

    l=0,u=n-1;
    c=binary(a,n,m,l,u);
    if(c==0)
         printf("Number is not found.");
    else
         printf("Number is found.");

    return 0;
 }

int binary(int a[],int n,int m,int l,int u){

     int mid,c=0;

     if(l<=u){
          mid=(l+u)/2;
          if(m==a[mid]){
              c=1;
          }
          else if(m<a[mid]){
              return binary(a,n,m,l,mid-1);
          }
          else
              return binary(a,n,m,mid+1,u);
     }
     else
       return c;
}
```

EXPERIMENT4

Aim: Implement 4.1 & 4.2 and Design any one from 4.3 to 4.4.(Greedy Approach)

1. Making Change
2. Knapsack
3. You are given n events where each takes one unit of time. Event i will provide a profit of gi dollars (gi > 0) if started at or before time ti where ti is an arbitrary real number. (Note: If an event is not started by t_i then there is no benefit in scheduling it at all. All events can start as early as time 0.)Give the most efficient algorithm to find a schedule that maximizes the profit.
4. Suppose you were to drive from station Louis to Denver along I-70. Your gas tank, when full, holds enough gas to travel m miles, and you have a map that gives distances between gas stations along the route. Let d1<d2 <.. < dn be the locations of all the gas stations along the route where di is the distance from station Louis to the gas station. You can assume that the distance between neighboring gas stations is at most m miles.
   Your goal is to make as few gas stops as possible along the way. Give the most efficient algorithm to determine at which gas stations you should stop and prove that your strategy yields an optimal solution. Be sure to give the time complexity of your algorithm as a function of n.

4.1 Making Change

MAKE-CHANGE (n)

```
C ← {100, 25, 10, 5, 1}     // constant.
Sol ← {};                   // set that will hold the solution set.
Sum ← 0 sum of item in solution set
WHILE sum not = n
   x = largest item in set C such that sum + x ≤ n
   IF no such item THEN
      RETURN    "No Solution"
   S ← S {value of x}
   sum ← sum + x
RETURN S
```

Example Make a change for 2.89 (289 cents) here n = 2.89 and the solution contains 2 dollars, 3 quarters, 1 dime and 4 pennies. The algorithm is greedy because at every stage it chooses the largest coin without worrying about the consequences. Moreover, it never changes its mind in the

sense that once a coin has been included in the solution set, it remains there.

4.2 Knapsack

- Algorithm:
  - Assume knapsack holds weight W and items have value v_i and weight w_i
  - Rank items by value/weight ratio: v_i / w_i
    - Thus: $v_i / w_i \geq v_j / w_j$, for all $i \leq j$
  - Consider items in order of decreasing ratio
  - Take as much of each item as possible

```
# include<stdio.h>

# include<conio.h>


void knapsack(int n, float weight[], float profit[], float

capacity)

{

 float x[20], tp= 0;

 int i, j, u;

 u=capacity;


 for (i=0;i<n;i++)

     x[i]=0.0;


 for (i=0;i<n;i++)
```

```
{
if(weight[i]>u)
     break;
else
    {
    x[i]=1.0;
    tp= tp+profit[i];
    u=u-weight[i];
    }
}

if(i<n)
      x[i]=u/weight[i];

tp= tp + (x[i]*profit[i]);

printf("n The result vector is:- ");
for(i=0;i<n;i++)
       printf("%ft",x[i]);

printf("m Maximum profit is:- %f", tp);
```

```
}

void main()
{
 float weight[20], profit[20], capacity;
 int n, i ,j;
 float ratio[20], temp;
 clrscr();

 printf ("n Enter the no. of objects:- ");
 scanf ("%d", &num);

 printf ("n Enter the wts and profits of each object:- ");
 for (i=0; i<n; i++)
 {
 scanf("%f %f", &weight[i], &profit[i]);
 }

 printf ("n enter the capacityacity of knapsack:- ");
 scanf ("%f", &capacity);
```

```
for (i=0; i<n; i++)
{
ratio[i]=profit[i]/weight[i];
}
for(i=0; i<n; i++)
{
   for(j=i+1;j< n; j++)
   {
     if(ratio[i]<ratio[j])
     {
     temp= ratio[j];
     ratio[j]= ratio[i];
     ratio[i]= temp;

    temp= weight[j];
    weight[j]= weight[i];
    weight[i]= temp;

    temp= profit[j];
    profit[j]= profit[i];
```

```
      profit[i]= temp;
      }
    }
 }
 knapsack(n, weight, profit, capacity);
 getch();
}
```

4.3 You are given n events where each takes one unit of time. Event i will provide a profit of gi dollars (gi > 0) if started at or before time ti where ti is an arbitrary real number. (Note: If an event is not started by t_i then there is no benefit in scheduling it at all. All events can start as early as time 0.)Give the most efficient algorithm to find a schedule that maximizes the profit.

4.4 Suppose you were to drive from station Louis to Denver along I-70. Your gas tank, when full, holds enough gas to travel m miles, and you have a map that gives distances between gas stations along the route. Let d1<d2 <.. < dn be the locations of all the gas stations along the route where di is the distance from station Louis to the gas station. You can assume that the distance between neighboring gas stations is at most m miles.

Your goal is to make as few gas stops as possible along the way. Give the most efficient algorithm to determine at which gas stations you should stop and prove that your strategy yields an optimal solution. Be sure to give the time complexity of your algorithm as a function of n.

EXPERIMENT 5

Aim: Implement 5.1 & 5.2 and Design any one from 5.3 to 5.4.(Dynamic Programming Approach)

1. Matrix Chain Multiplication
2. Knapsack
3. Find the minimum of characters to be inserted to convert it into palindrome.
4. Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown.

5.1 Matrix Chain Multiplication

```
Matrix-Chain(array p[1 .. n], int n) {
        Array s[1 .. n − 1, 2 .. n];
        FOR i = 1 TO n DO m[i, i] =
0;                                  // initialize
        FOR L =
2 TO n DO {                                          // L=length
of subchain
                FOR i = 1 TO n − L + 1 do {
                        j = i + L − 1;
                        m[i, j] = infinity;
                        FOR k = i TO j −
1 DO {                                  // check all splits
                                q = m[i, k] + m[k + 1, j] + p[i −
1] p[k] p[j];
                                IF (q < m[i, j]) {
                                        m[i, j] = q;
                                        s[i, j] = k;
                                }
                        }
                }
        }
```

return $m[1, n]$(final cost) and s (splitting markers);
}

Complexity Analysis

Clearly, the space complexity of this procedure $O(n^2)$. Since the tables m and s require $O(n^2)$ space. As far as the time complexity is concern, a simple inspection of the for-loop(s) structures gives us a running time of the procedure. Since, the three for-loops are nested three deep, and each one of them iterates at most n times (that is to say indices L, i, and j takes on at most $n - 1$ values). Therefore, The running time of this procedure is $O(n^3)$.

```
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one extra row and one extra column
are
       allocated in m[][].  0th row and 0th column of m[][] are not used
*/
    int m[n][n];

    int i, j, k, L, q;

    /* m[i,j] = Minimum number of scalar multiplications needed to
compute
       the matrix A[i]A[i+1]...A[j] = A[i..j] where dimention of A[i] is
       p[i-1] x p[i] */

    // cost is zero when multiplying one matrix.
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<=n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
```

```
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n-1];
}
int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
                      MatrixChainOrder(arr, size));

    getchar();
    return 0;
}
```

5.2 Knapsack

Let *i* be the highest-numbered item in an optimal solution S for W pounds. Then $S` = S - \{i\}$ is an optimal solution for $W - w_i$ pounds and the value to the solution S is V_i plus the value of the subproblem.

We can express this fact in the following formula: define $c[i, w]$ to be the solution for items 1,2, . . . , *i* and maximum weight w. Then

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } w_i \geq 0 \\ \max [v_i + c[i-1, w-w_i], c[i-1, w]\} & \text{if } i>0 \text{ and } w \geq w_i \end{cases}$$

This says that the value of the solution to i items either include i^{th} item, in which case it is v_i plus a subproblem solution for $(i - 1)$ items and the weight excluding w_i, or does not include i^{th} item, in which case it is a subproblem's solution for $(i - 1)$ items and the same weight. That is, if the thief picks item i, thief takes v_i value, and thief can choose from items $w - w_i$, and get $c[i - 1, w - w_i]$ additional value. On other hand, if thief decides not to take item *i*, thief can choose from item 1,2, . . . , *i*- 1 upto the weight limit *w*, and get $c[i - 1, w]$ value. The better of these two choices should be made.

Although the 0-1 knapsack problem, the above formula for *c* is similar to LCS formula: boundary values are 0, and other values are computed from the input and "earlier" values of c. So the 0-1 knapsack algorithm is like the LCS-length algorithm given in CLR for finding a longest common subsequence of two sequences.

The algorithm takes as input the maximum weight *W*, the number of items n, and the two sequences $v = <v_1, v_2, \ldots, v_n>$ and $w = <w_1, w_2, \ldots, w_n>$. It stores the $c[i, j]$ values in the table, that is, a two dimensional array, $c[0 \ldots n, 0 \ldots w]$ whose entries are computed in a row-major order. That is, the first row of c is filled in from left to right, then the second row, and so on. At the end of the computation, $c[n, w]$ contains the maximum value that can be picked into the knapsack.

Dynamic-0-1-knapsack (v, w, n, W)

```
FOR w = 0 TO W
   DO  c[0, w] = 0
FOR i=1 to n
   DO c[i, 0] = 0
      FOR w=1 TO W
```

DO IFf $w_i \leq w$
THEN IF $v_i + c[i\text{-}1, w\text{-}w_i]$
THEN $c[i, w] = v_i + c[i\text{-}1, w\text{-}w_i]$
ELSE $c[i, w] = c[i\text{-}1, w]$
ELSE
$c[\text{i}, w] = c[i\text{-}1, w]$

Analysis

This dynamic-0-1-kanpsack algorithm takes $\theta(nw)$ times, broken up as follows: $\theta(nw)$ times to fill the *c*-table, which has $(n+1).(w+1)$ entries, each requiring $\theta(1)$ time to compute. $O(n)$ time to trace the solution, because the tracing process starts in row n of the table and moves up 1 row at each step.

5.3 Find the minimum of characters to be inserted to convert it into palindrome.

5.4 Given n dice each with m faces, numbered from 1 to m, find the number of ways to get sum X. X is the summation of values on each face when all the dice are thrown.

EXPERIMENT6

Aim: Implement 6.1 Design any two from 6.2 to 6.4.(Graph)

1. Eight Queen Problem
2. Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices. Solve this using Backtracking.
3. Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented). Solve this using Backtracking
4. In the network of PSTN if you are found the problem at the leaf level i.e. at the telephone at your home. Apply the better approach to solve the problem from Breadth First search and Depth First Search. Design the algorithm for it.

6.1 Eight Queen Problem

6.2 Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices. Solve this using Backtracking.

6.3 Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented). Solve this using Backtracking

6.4 In the network of PSTN if you are found the problem at the leaf level i.e. at the telephone at your home. Apply the better approach to solve the problem from Breadth First search and Depth First Search. Design the algorithm for it.

EXPERIMENT 7

Aim: Solve the following problems

Problem: Suppose you are playing game of shooting balloon. You expect to shoot n balloons in the board, assuming you are sharpshooter, 100% hit. There are two scenarios, you need find the appropriate Big Oh notation for each scenario. In these problems, one unit of work is shooting one balloon.

1. For every 2 balloons you are able to shoot, one new balloon is inserted in the board. So, if there were 20 balloons, after you shoot the first 2, there are 19 on the board. After you shoot the next 2, there are 18 on the board. How many balloons do you shoot before the board is empty?
   - A: O(1)
   - B: O(n)
   - C: O(lgn)
   - D: $O(n^2)$

2. By the time you have shoot the first n balloons, n-1 new balloons have been inserted on the board. After shooting those n-1 balloons, there are n-2 new balloons are inserted on the board. After checking out those n-2 balloons , there are n-3 new balloons on the board. This same pattern continues until on new balloon are inserted on the board. How many total balloons do you shoot before the board is empty?
   - A: O(1)
   - B: O(n)
   - C: O(lgn)
   - D: $O(n^2)$

7.1 For every 2 balloons you are able to shoot, one new balloon is inserted in the board. So, if there were 20 balloons, after you shoot the first 2, there are 19 on the board. After you shoot the next 2, there are 18 on the board. How many balloons do you shoot before the board is empty?

7.2 By the time you have shoot the first n balloons, n-1 new balloons have been inserted on the board. After shooting those n-1 balloons, there are n-2 new balloons are inserted on the board. After checking out those n-2 balloons , there are n-3 new balloons on the board. This same pattern continues until on new balloon are inserted on the board. How many total balloons do you shoot before the board is empty?