**PRACTICAL-1**

**Aim:** Implement and analyze algorithms given below.

* 1. Factorial (Iterative and Recursive)
  2. Fibonacci Series(Iterative and Recursive)
  3. Matrix Addition and Matrix Multiplication(Iterative)
  4. Recursive Linear Search and Binary Search (Comparative Study)

**Software Required:** CodeBlocks

**Hardware Required:** NA

**Knowledge Required:** Basic knowledge of c, c++

* 1. **Factorial (Iterative and Recursive)**
* Iterative Factorial :

Theory:

* The factorial of a positive number n is given by:

Factorial of n = 1\*2\*…\*n

* The factorial of a negative number doesn't exist. And, the factorial of 0 is 1, 0! = 1.

Algorithm:

1. Start

2. Read the number n

3. [Initialize]

i=1, fact=1

4. Repeat step 4 through 6 until i=n

5. Fact=fact\*i

6. I=i+1

7. Print fact

8. Stop

Code:

#include<stdio.h>

int main(){

int i=1,f=1,num,c=0;

printf("Enter a number: ");

scanf("%d",&num);

while(i<=num){

f=f\*i;

i++;

c=c+2;

}

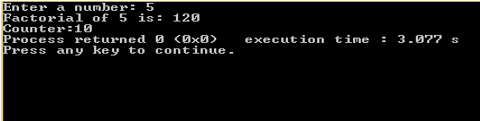
printf("Factorial of %d is: %d",num,f);

printf(“\nCounter:%d”,c);

return 0;

}

Output:



Graph:

|  |  |  |
| --- | --- | --- |
| Sr. No. | Input No. | No. of Count |
| 1 | 1 | 6 |
| 2 | 2 | 7 |
| 3 | 3 | 8 |
| 4 | 4 | 9 |
| 5 | 5 | 10 |

* Recursive Factorial :

Code:

#include<stdio.h>

int fact(int num);

int c=0;

int main(){

int num;

c++;

printf("Enter a number: ");

c++;

scanf("%d",&num);

c++;

printf("Factorial of %d is: %d",num,fact(num));

c++;

printf("\nCounter : %d",c);

return 0;

}

int fact(int n){

if(n>=1){

c++;

return n\*fact(n-1);

c++;

}

else{

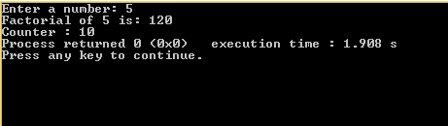
c++;

return 1;

}

}

Output:



Graph:

|  |  |  |
| --- | --- | --- |
|  | Size of Input | No. of count |
| 1 | 5 | 10 |
| 2 | 10 | 15 |
| 3 | 15 | 20 |
| 4 | 20 | 25 |
| 5 | 25 | 30 |

* 1. **Fibonacci Series(Iterative and Recursive)**
* Iterative Fibonacci series :

Theory:

* The Fibonacci sequence is a series where the next term is the sum of pervious two terms. The first two terms of the Fibonacci sequence is 0 followed by 1.

Fibonacci series :0,1,1,2,3,5,8…

Algorithm:

1. declare f1,f2,fib,loop

2. set f2 to 0

3. set f2 to 0

4. display f1,f2

5. for loop 1 to n

6. fib=f1+f2

7. f1=f2

8. f2=fib

9. end for loop

10. display fib

Code:

#include<stdio.h>

int main(){

int n,first=0,second=1,next,i,c=0;

printf("Enter the number of terms\n");

scanf("%d",&n);

printf("First %d terms of Fibonacci series are :-\n",n);

c=c+5;

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

c=c+2;

if ( i<= 1 ){

next = i;

c=c+2;

}

else{

next = first + second;

first = second;

second = next;

c=c+3;

}

printf(" %d ",next);

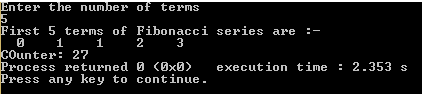
}

printf("\nCOunter: %d",c);

return 0;

}

Output:



Graph:

|  |  |  |
| --- | --- | --- |
| Sr. No | No Of Inputs | Counts |
| 1 | 5 | 27 |
| 2 | 10 | 62 |
| 3 | 15 | 97 |
| 4 | 20 | 132 |
| 5 | 25 | 167 |

* Recursive Fibonacci series :

Code:

#include<stdio.h>

int c=0;

int main(){

int n,a=0,b=1;

printf("Enter The Limit:");

scanf("%d",&n);

printf("%d ",a);

printf("%d ",b);

fibo(n,a,b);

printf("\nCounter:%d",c);

return 0;

}

int fibo(int n,int a,int b){

a=a+b;

c++;

printf("%d ",a);

c++;

if(n-2>0){

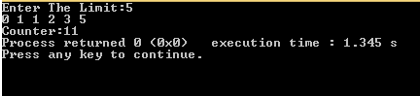
c++;

return fibo(n-1,b,a);

c++;

}

Output:



Graph:

|  |  |  |
| --- | --- | --- |
| Sr. No | No Of Inputs | Counts |
| 1 | 5 | 11 |
| 2 | 10 | 26 |
| 3 | 15 | 41 |
| 4 | 20 | 56 |
| 5 | 25 | 71 |

* 1. **Matrix Addition and Matrix Multiplication (Iterative)**
* Matrix Addition :

Theory:

* This program asks user to enter the size of the matrix (rows and column) then, it asks the user to enter the elements of two matrices and finally it adds two matrix and displays the result.

Algorithm:

1. Start

2. Read: m and n

3. Read: Take inputs for Matrix A[1:m, 1:n] and Matrix B[1:m, 1:n]

4. Repeat for i := 1 to m by 1:

              Repeat for j := 1 to n by 1:

                              C[i, j] := A[i, j] + B[i, j]

                  [End of inner for loop]

      [End of outer for loop]

5. Print: Matrix C

6. Exit.

Code:

#include<stdio.h>

#include<conio.h>

int main(){

int a[10][10],b[10][10],c[10][10],d[10][10],i,j,k,c1=0,c2=0,c3=0,n;

c1++;

printf("Enter Size of Matrix:");

scanf("%d",&n);

printf("Enter Matrix A of size %d:\n",n);

c1++;

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

c2=c2+3;

for(j=0;j<n;j++){

c3=c3+3;

scanf("%d",&a[i][j]);

c3++;

}

}

printf("Enter Matrix B of size %d:\n",n);

c1++;

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

c2=c2+3;

for(j=0;j<n;j++){

c3=c3+3;

scanf("%d",&b[i][j]);

c3++;

}

}

printf("Addition Matrix:\n");

c1++;

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

c2=c2+3;

for(j=0;j<n;j++){

c3=c3+3;

c[i][j]=a[i][j]+b[i][j];

c3++;

}

}

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

c2=c2+3;

for(j=0;j<n;j++){

c3=c3+3;

printf(" %d ",c[i][j]);

c3++;

}

printf("\n");

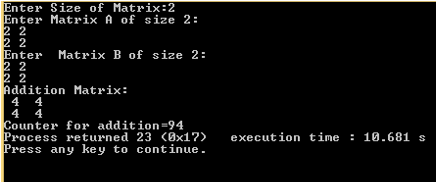
c1++;

}

printf("Counter for addition=%d",c1+c2+c3);

}

Output:



Graph:

|  |  |  |
| --- | --- | --- |
| Sr. No. | Size of Matrix | No. of count |
| 1 | 2\*2 | 84 |
| 2 | 3\*3 | 157 |
| 3 | 4\*4 | 254 |
| 4 | 5\*5 | 375 |
| 5 | 6\*6 | 520 |

* Matrix multiplication :

Theory:

* This program asks user to enter two matrices and this program multiplies these two matrix and displays it. If you don't know matrix multiplication, visit this page to learn, [how two matrix can be multiplied](http://en.wikipedia.org/wiki/Matrix_multiplication#Matrix_product_.28two_matrices.29).

Algorithm:

1. Start.

2. Read: m, n, p and q

3. Read: Inputs for Matrices A[1:m, 1:n] and B[1:p, 1:q].

4. If n ≠ p then:

Print: Multiplication is not possible.

Else:

Repeat for i := 1 to m by 1:

Repeat for j := 1 to q by 1:

C[i, j] := 0 [Initializing]

Repeat k := 1 to n by 1

C[i, j] := C[i, j] + A[i, k] x B[k, j]

[End of for loop]

[End of for loop]

[End of for loop]

[End of If structure]

5. Print: C[1:m, 1:q]

6. Exit.

Code:

#include<stdio.h>

#include<conio.h>

void main(){

int a[3][3],b[3][3],c[3][3],d[3][3],i,j,k,c1=0,c2=0,c3=0,n;

c1++;

printf("Enter Size of Matrix:");

c1++;

scanf("%d",&n);

c1++;

printf("Enter Matrix A of size %d:\n",n);

c1++;

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

c2=c2+3;

for(j=0;j<n;j++){

c3=c3+3;

scanf("%d",&a[i][j]);

c3++;

}

}

printf("Enter Matrix B of size %d:\n",n);

c1++;

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

c2=c2+3;

for(j=0;j<n;j++){

c3=c3+3;

scanf("%d",&b[i][j]);

c3++;

}

}

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

c2=c2+3;

for(j=0;j<n;j++){

c3=c3+3;

d[i][j]=0;

c3++;

}

}

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

c1=c1+3;

for(j=0;j<n; ++j){

c2=c2+3;

for(k=0; k<n; ++k){

c3=c3+3;

d[i][j]+=a[i][k]\*b[k][j];

c3++;

}

}

}

printf("\nMultipication Matrix:\n");

c1++;

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

c2=c2+3;

for(j=0; j<n; ++j){

c3=c3+3;

printf(" %d ",d[i][j]);

c3++;

}

printf("\n");

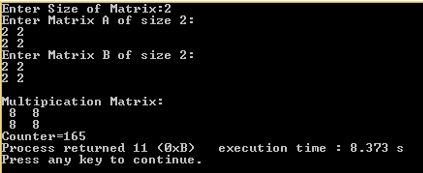
c1++;

}

printf("Counter=%d",c1+c2+c3);

}

Output:



Graph:

|  |  |  |
| --- | --- | --- |
| Sr. No. | Size of Matrix | No. of count |
| 1 | 2\*2 | 165 |
| 2 | 3\*3 | 362 |
| 3 | 4\*4 | 671 |
| 4 | 5\*5 | 1116 |
| 5 | 6\*6 | 1721 |

* 1. **Recursive Linear Search and Binary Search (Comparative Study)**
* Recursive Linear search :

Theory:

* LinearSearch(value, list)
* if the list is empty, return Λ;
* else
* if the first item of the list has the desired value, return its location;
* else return LinearSearch(value, remainder of the list)

Code:

#include<stdio.h>

#include<conio.h>

int c=0;

int main(){

int \*data;

int i,n,find,start,end;

printf("Enter The Range:");

scanf("%d",&n);

data = (int \*) malloc(sizeof(int)\*n);

printf("\nEnter The Data:");

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

scanf("%d",(data+i));

}

printf("Enter The No To Be Found:");

scanf("%d",&find);

start=0;

end=n-1;

linear(data,start,end,find);

c=c+6;

printf("Counter:%d",c);

return 0;

}

int linear(int \*data,int start,int end,int find){

c++;

if(start<=end){

c++;

if(\*(data+start)==find){

c++;

printf("Element found at index %d.",start+1);

c++;

}

else{

c++;

return linear(data,start+1,end,find);

}

}

else{

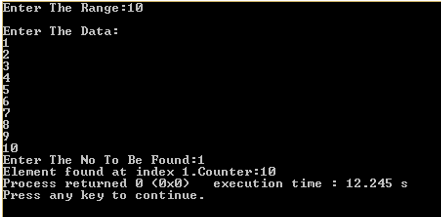
c++;

printf("Element Not Found.");

}

}

Output:



Graph:

|  |  |  |  |
| --- | --- | --- | --- |
| Linear Search Recursive | | | |
| Size of list | Best Case | Average List | Worst Case |
| 5 | 10 | 24 | 30 |
| 10 | 10 | 38 | 50 |
| 15 | 10 | 49 | 70 |
| 20 | 10 | 63 | 90 |

* Recursive Binary Search :

Theory:

* Find the midpoint of the array; this will be the element at arr[size/2]*.* The midpoint divides the array into two smaller arrays: the lower half of the array consisting of elements 0 to midpoint - 1, and the upper half of the array consisting of elements midpoint to size - 1.
* Compare key to arr[midpoint] by calling the user function cmp\_proc.
* If the key is a match, return arr[midpoint]; otherwise
* If the array consists of only one element return NULL, indicating that there is no match; otherwise
* If the key is less than the value extracted from arr[midpoint] search the lower half of the array by recursively calling *search;* otherwise
* Search the upper half of the array by recursively calling search.

Code:

#include<stdio.h>

int binarys(int m,int n,int \*pt,int k);

int count=0;

int main(){

int \*data;

int x,l,n,i,ans;

printf("\nEnter The Limit:");

scanf("%d",&n);

data = (int\*) malloc(n\*sizeof(int));

printf("\nEnter The Data:");

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

scanf("%d",(data+i));

}

printf("\nEnter The Element To Be Searched:");

scanf("%d",&x);

ans=binarys(0,n-1,data,x);

if(ans==-1){

printf("Element Not Found.");

}

else{

printf("\nElement Is Found At The Index %d. ",ans+1);

}

printf("\nCount : %d",count+9);

}

int binarys(int m,int n,int \*pt,int k){

int mid;

if(m>n){

count++;

return -1;

}

count+=2;

mid=(m+n)/2;

if(k<\*(pt+mid)){

count+=2;

n=mid-1;

return binarys(m,n,pt,k);

}

else if(k>\*(pt+mid)){

count+=4;

m=mid+1;

return binarys(m,n,pt,k);

}

else if(k==\*(pt+mid)){

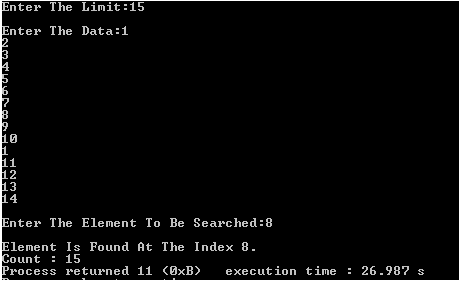
count+=4;

return mid;

}

}

Output:



Graph:

|  |  |  |  |
| --- | --- | --- | --- |
| Binary Search | | | |
| Size of list | Best Case | Average List | Worst Case |
| 5 | 15 | 25 | 28 |
| 10 | 15 | 30 | 33 |
| 15 | 15 | 35 | 41 |
| 20 | 15 | 40 | 49 |

* Best case :
* Average case & Worst case:

**Time complexity:**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No. | Name | Theorical | Practical |
| 1 | Iterative Factorial | O(n) | O(n) |
| 2 | Recursive Factorial | O(n) | O(n) |
| 3 | Iterative Fibonacci series | O(n) | O(n) |
| 4 | Recursive Fibonacci series | O(n) | O(n) |
| 5 | Matrix Addition | O(n^2) | O(n^2) |
| 6 | Matrix Multiplication | O(n^3) | O(n^3) |
| 7 | Iterative Linear Search – Best case | O(1) | O(1) |
| 8 | Iterative Linear Search – Average case | O(n) | O(n) |
| 9 | Iterative Linear Search – Worst case | O(n) | O(n) |
| 10 | Recursive Linear search – Best case | O(1) | O(1) |
| 11 | Recursive Linear Search – Average case | O(n) | O(n) |
| 12 | Recursive Linear Search – Worst case | O(n) | O(n) |
| 13 | Recursive Binary Search – Best case | O(1) | O(1) |
| 14 | Recursive Binary Search – Average case | O(n.log(n)) | O(n.log(n)) |
| 15 | Recursive Binary Search – Worst case | O(n.log(n)) | O(n.log(n)) |

**Conclusion**: From this practical, we have learnt about iterative and recursive approach to find factorial of any number and fibonacci series. We have also learnt various search techniques like linear search and binary search and an iterative approach for matrix addition and matrix multiplication.

**PRACTICAL 2**

**Aim:** Implement and analyze algorithms given below (compare them).

2.1 Bubble Sort

2.2 Selection Sort

2.3 Insertion Sort

**Software Required:** CodeBlocks

**Hardware Required:** NA

**Knowledge Required:** Basic knowledge of c, c++

**2.1 Bubble Sort**

Theory:

* Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.
* The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
* The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list.

Example:

* Array list : 5 1 4 2 8
* Pass 1 : (**5 1** 4 2 8) → (**1 5** 4 2 8)
  + - * (1 5 4 2 8) → (1 **4 5** 2 8)
      * (1 4 **5 2** 8) → (1 4 **2 5** 8)
      * (1 4 2 **5 8**) → (1 4 2 **5 8**)
* Pass 2 : (**1 4** 2 5 8) → (**1 4** 2 5 8)
  + - * (1 **4 2** 5 8) → (1 **2 4** 5 8)
      * (1 2 **4 5** 8) → (1 2 **4 5** 8)
      * (1 2 4 **5 8**) → (1 2 4 **5 8**)
* Pass 3 : (**1 2** 4 5 8) → (**1 2** 4 5 8)  
  (1 **2 4** 5 8) → (1 **2 4** 5 8)
  + - * (1 2 **4 5** 8) → (1 2 **4 5** 8)
      * (1 2 4 **5 8**) → (1 2 4 **5 8**)
* Sorted array is : 1 2 4 5 8

Algorithm:

1. Last ← n

2. Repeat thru step 5 for pass: 1,2,3..,n-1

3. E ← 0

4. Repeat for i=1,2,..,last-1

If k[i] > k[i+1]

then k[i] ↔ k[i+1]

E ← E+1

5. If E=0

then return

else

last ← last-1

6. Return number of passes

Code:

#include<stdio.h>

#include<conio.h>

#include<malloc.h>

int main(){

int i,j,k,temp,n,\*p,e,c=0,swap=0,comp=0;

printf("Enter n : ");

scanf("%d",&n);

printf(" \nEnter an array of %d integers :\n",n);

p=(int\*)malloc(sizeof(n));

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

scanf("%d",p+i);

}c++;

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

c=c+2;

e=0;

c++;

c++;

for(j=0;j<n-i-1;j++){

c=c+2;

comp=comp+1;

if(\*(p+j)>\*(p+j+1)){

c++;

temp=\*(p+j);

\*(p+j)=\*(p+j+1);

\*(p+j+1)=temp;

e=e+1;

swap=swap+1;

c=c+4;

}

}

printf("\nAfter Pass %d: ",i+1);

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

printf(" %d ",\*(p+k));

if(e==0){

c++;

break;

}

}

printf("\nSorted Array:");

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

printf(" %d ",\*(p+i));

}

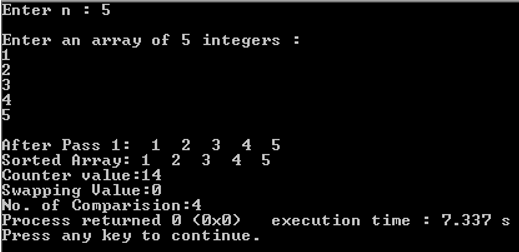
printf("\nCounter value:%d \nSwapping Value:%d \nNo. Of Comparision:%d",c,swap,comp);

return 0;

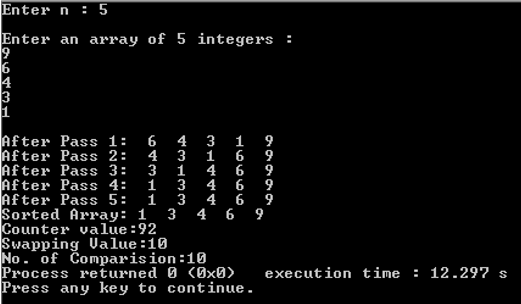
}

Output:

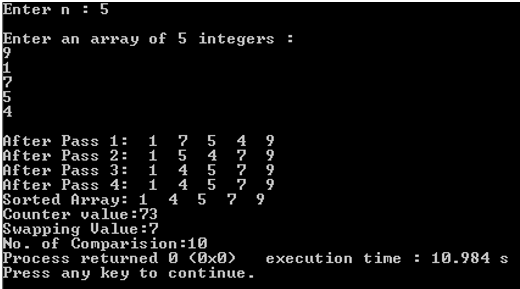
* Best case :



* Worst case :



* Average case :



Graph:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sr. No | No Of Inputs | Bubble Sort | | |
| Best Case | Average Case | Worst Case |
| Counts | | |
| 1 | 5 | 14 | |  | | --- | | 92 | | 73 |
| 2 | 10 | 24 | 357 | 103 |
| 3 | 15 | 34 | 797 | 133 |
| 4 | 20 | 44 | 1412 | 163 |
| 5 | 25 | 54 | 2202 | 193 |

* Best case :

Time Complexity: O(n)

* Worst case :

Time Complexity: O(n^2)

* Average case :

Time Complexity: O(n^2)

**2.2 Selection Sort**

Theory:

* Selection sort is a [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm), specifically an [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort).
* It has [O](https://en.wikipedia.org/wiki/Big_O_notation)(n2) time complexity, making it inefficient on large lists, and generally performs worse than the similar [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort).
* Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.
* The algorithm divides the input list into two parts: the sub list of items already sorted, which is built up from left to right at the front (left) of the list, and the sub list of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sub list is empty and the unsorted sub list is the entire input list.
* The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sub list, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sub list boundaries one element to the right.

Example:

* Array list : 64 25 12 22 11
* Pass 1 : 11 64 25 12 22
* Pass 2 : 11 12 64 25 22
* Pass 3 : 11 12 22 64 25
* Pass 4 : 11 12 22 25 64
* Sorted array : 11 12 22 25 64

Algorithm:

1. Repeat thru step 4 for Pass=1,2,..,n-1

2. Min\_index ← Pass

3. Repeat for i=Pass+1,Pass+2,..,n

if k[i] < k[Min\_index] then Min\_index ← i

4. if Min\_index!=Pass

then k[Pass] ↔ k[Min\_index]

5. Return

Code:

#include<stdio.h>

#include<conio.h>

#include<malloc.h>

int main(){

int n,\*p,i,j,k,min,temp,c,swap=0,comp=0,ct=0;

printf("Enter n :");

scanf("%d",&n);

printf("\nEnter an array of %d integers:\n",n);

p=(int\*)malloc(sizeof(n));

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

scanf("%d",p+i);

}

ct++;

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

ct=ct+2;

min=\*(p+i);

ct++;

c=i;

ct++;

for(j=i+1;j<n;j++){

ct=ct+2;

if(\*(p+j)<=min){

ct++;

comp++;

min=\*(p+j);

ct++;

c=j;

}

}

if(c!=i){

temp=\*(p+c);

\*(p+c)=\*(p+i);

\*(p+i)=temp;

ct=ct+3;

swap++;

}

printf("\nAfter Pass %d:",i+1);

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

printf(" %d ",\*(p+k));

}

printf("\nSelection : \n");

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

printf("%d ",\*(p+i));

}

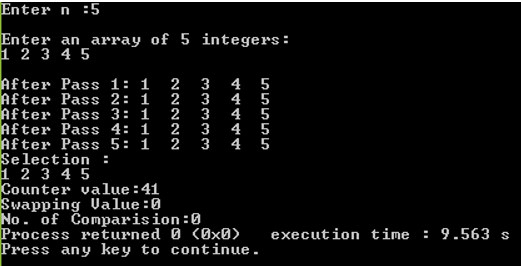
printf("\nCounter value:%d \nSwapping Value:%d \nNo. of Comparision:%d",ct,swap,comp);

return 0;

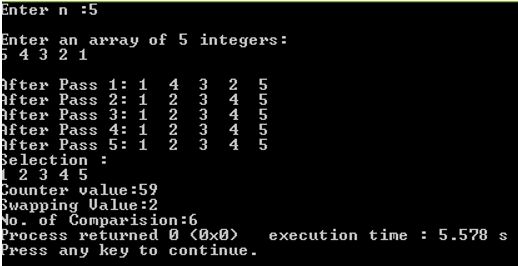
}

Output:

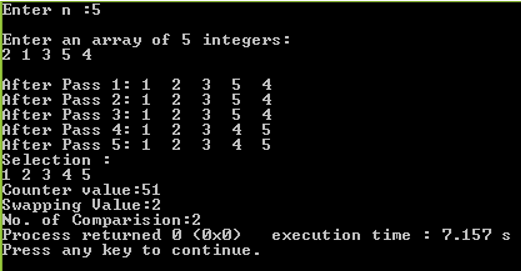
* Best case :



* Worst case :



* Average case :



Graph:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sr. No | No Of Inputs | Selection Sort | | |
| Best Case | Average Case | Worst Case |
| Counts | | |
| 1 | 5 | 41 | |  | | --- | | 59 | | 53 |
| 2 | 10 | 131 | 196 | 180 |
| 3 | 15 | 271 | 404 | 351 |
| 4 | 20 | 461 | 691 | 595 |
| 5 | 25 | 701 | 1049 | 923 |

* Best case :

Time Complexity : O(n^2)

* Worst case :

Time Complexity : O(n^2)

* Average case :

Time Complexity : O(n^2)

**2.3 Insertion sort**

Theory:

* Insertion sort is a simple [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that builds the final [sorted array](https://en.wikipedia.org/wiki/Sorted_array) (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as [quicksort](https://en.wikipedia.org/wiki/Quicksort), [heapsort](https://en.wikipedia.org/wiki/Heapsort" \o "Heapsort), or [merge sort](https://en.wikipedia.org/wiki/Merge_sort). However, insertion sort provides several advantages:
* Simple implementation: [Bentley](https://en.wikipedia.org/wiki/Jon_Bentley_(computer_scientist)) shows a three-line [C](https://en.wikipedia.org/wiki/C_(programming_language)) version, and a five-line [optimized](https://en.wikipedia.org/wiki/Program_optimization) version[[1]](https://en.wikipedia.org/wiki/Insertion_sort#cite_note-pearls-1):116
* Efficient for (quite) small data sets, much like other quadratic sorting algorithms
* More efficient in practice than most other simple quadratic (i.e., [O](https://en.wikipedia.org/wiki/Big_O_notation)(n2)) algorithms such as [selection sort](https://en.wikipedia.org/wiki/Selection_sort) or [bubble sort](https://en.wikipedia.org/wiki/Bubble_sort)
* [Adaptive](https://en.wikipedia.org/wiki/Adaptive_sort), i.e., efficient for data sets that are already substantially sorted: the [time complexity](https://en.wikipedia.org/wiki/Time_complexity) is O(nk) when each element in the input is no more than k places away from its sorted position
* [Stable](https://en.wikipedia.org/wiki/Stable_sort); i.e., does not change the relative order of elements with equal keys
* [In-place](https://en.wikipedia.org/wiki/In-place_algorithm); i.e., only requires a constant amount O(1) of additional memory space
* [Online](https://en.wikipedia.org/wiki/Online_algorithm); i.e., can sort a list as it receives it.

Example:

* Array list : 3 7 4 9 5 2 6 1
* Pass 1 : **3** 7 4 9 5 2 6 1
* Pass 2 : 3 **7** 4 9 5 2 6 1
* Pass 3 : 3 **4** 7 9 5 2 6 1
* Pass 4 : 3 4 7 **9** 5 2 6 1
* Pass 5 : 3 4 **5** 7 9 2 6 1
* Pass 6 : **2** 3 4 5 7 9 6 1
* Pass 7 : 2 3 4 5 **6** 7 9 1
* Pass 8 : **1** 2 3 4 5 6 7 9
* Sorted Array : 1 2 3 4 5 6 7 8 9

Algorithm:

1. For j←2 to n

do key ← A[j]

2. i ← j-1

3. while i>0 & A[i]>key

do A[i+1] ← A[i]

i←i-1

4. A[i+1]←key

Code:

#include<stdio.h>

void main(){

int \*data, i, j, n, insert, pos, k, swap=0, comp=0;

printf("Enter The Range:");

scanf("%d",&n);

data = (int\*) malloc(sizeof(int)\*n);

printf("\n");

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

printf("Enter element %d : ",i+1);

scanf("%d",(data + i));

}

for(i=1;i<n;i++){

insert=\*(data + i);

pos=i;

comp++;

while(pos>0 && \*(data+pos-1)>insert){

\*(data + pos)=\*(data+pos-1);

swap++;

pos--;

}

\*(data + pos)=insert;

printf("\n PASS %d : ",i+1);

for(k=0; k<n; k++) {

printf(" %d",\*(data + k));

}

printf("\n\nSorted Array:");

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

printf("%d ",\*(data + i));

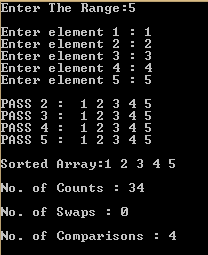
printf("\nNo. of Swaps : %d", swap);

printf("\nNo. of Comparisons : %d", comp);

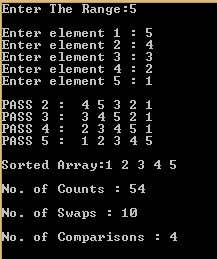
}

Output:

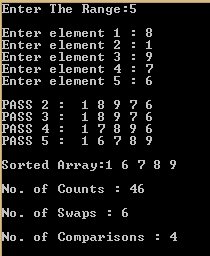
* Best case :



* Worst case :



* Average case :



Graph:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sr. No | No Of Inputs | Insertion Sort | | |
| Best Case | Average Case | Worst Case |
| Counts | | |
| 1 | 5 | 34 | |  | | --- | | 92 | | 73 |
| 2 | 10 | 50 | 357 | 103 |
| 3 | 15 | 74 | 797 | 133 |

* Best case :

Time Complexity: O(n)

* Worst case :

Time Complexity: O(n^2)

* Average case :

Time Complexity: O(n^2)

**Time Complexity:**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No. | Name | Theorical | Practical |
| 1 | Bubble Sort – Best case | O(n) | O(n) |
| 2 | Bubble Sort – Worst case | O(n^2) | O(n^2) |
| 3 | Bubble Sort – Average case | O(n^2) | O(n^2) |
| 4 | Selection Sort – Best case | O(n^2) | O(n^2) |
| 5 | Selection Sort – Worst case | O(n^2) | O(n^2) |
| 6 | Selection Sort – Average case | O(n^2) | O(n^2) |
| 7 | Insertion Sort – Best case | O(n) | O(n) |
| 8 | Insertion Sort – Worst case | O(n^2) | O(n^2) |
| 9 | Insertion Sort – Average case | O(n^2) | O(n^2) |

**Conclusion**: From this practical, we have learnt about various sorting algorithms and there complexities in various cases and comparisons between them.

**PRACTICAL – 3**

**Aim:** Divide and Conquer Strategy (Implement & Perform Analysis)

3.1 Merge Sort (Two way and Three way)

3.2 Quick Sort

**Software Required:** CodeBlocks

**Hardware Required:** NA

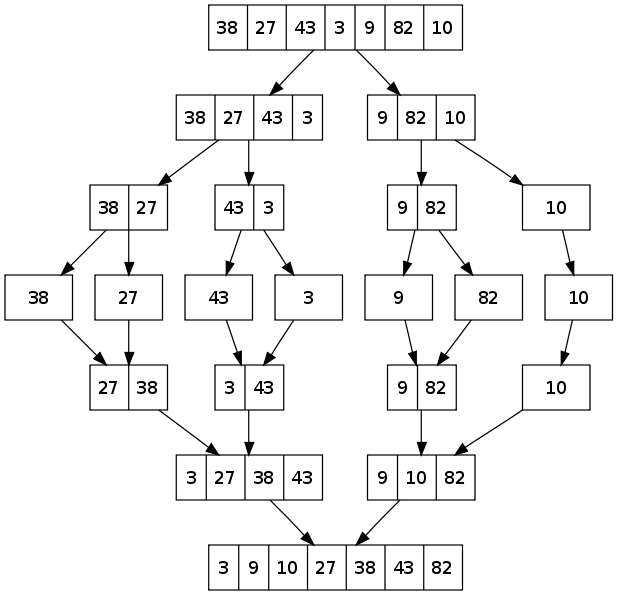
**Knowledge Required:** Basic knowledge of c, c++

**3.1 Merge Sort**

Theory:

* Merge sort is an efficient, general-purpose, [comparison-based](https://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm).
* Most implementations produce a [stable sort](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability), which means that the implementation preserves the input order of equal elements in the sorted output.

Example:



Algorithm:

* Simple Merge :

SIMPLE\_MERGE\_SORT(K,FIRST,SECOND,THIRD)

1. [INITIALIZE]

2. [FIND SMALLEST ELEMENT]

I ← FIRST

J ← SECOND

L ← 0

Repeat While I&lt;SECOND and J≤THIRD

If K[I] ≤ K[J]

Then L ← L+1

Else L ← L+1

TEMP[L] ← K[I]

I ← I+1

TEMP[L] ← K[I]

J ← J+1

3. [COPY THE REMAINING ELEMENTS]

If I≥SECOND

Then Repeat while J≤THIRD

L ← L+1

TEMP[L] ← K[J]

J ← J+1

Else Repeat while I&lt;SECOND

L ← L+1

TEMP[L] ← K[I]

I ← I+1

4. [COPY ELEMENTS]

Repeat for I=1,2…L

K[FIRST-1+I] ← TEMP[I]

5. [FINISHED] Return

Code:

#include<stdio.h>

void divide(int \*a, int min, int max, int \*tmp);

void merge(int \*a, int min, int mid, int max, int \*tmp);

int count=0,comp=0;

int x=1;

void main(){

int \*a, i, j, n,\*tmp;

printf("Enter the size of array : ");

scanf("%d", &n);

a = (int \*) malloc(sizeof(int)\*n);

tmp = (int \*) malloc(sizeof(int)\*n);

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

printf("Enter element %d : ",i+1);

scanf("%d", &a[i]);

}

count++;

divide(a,0,n-1,tmp);

printf("\n\nSorted array is : \n");

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

printf(" %d", tmp[i]);

}

printf("\n\nNo. of Counts: %d", count);

printf("\nNo. of Comparisons: %d" ,comp);

}

void divide(int \*a, int min, int max, int \*tmp){

int mid;

if(min!=max){

mid=((min + max)/2);

divide(a, min, mid, tmp);

divide(a, mid+1,max,tmp);

merge(a, min, mid, max, tmp);

count+=4;

}

count++;

}

void merge(int \*a, int min, int mid, int max, int \*tmp){

int i=min, j=mid+1, k=min;

count++;

while(i<=mid && j<=max){

count++; comp++;

if(a[i]<a[j]) { tmp[k]=a[i]; i++; }

else { tmp[k]=a[j]; j++; }

k++;

count+=4;

}

count++;

while(j<=max) { tmp[k]=a[j]; k++; j++; count+=4; }

count++;

while(i<=mid) { tmp[k]=a[i]; k++; i++; count+=4; }

printf("\n AFTER PASS %d : ",x); x++;

for(i=min; i<=max; i++){

count++;

a[i]=tmp[i];

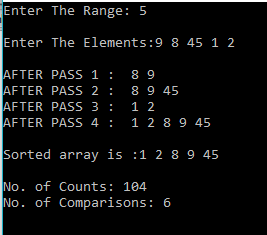
printf(" %d",a[i]);

}

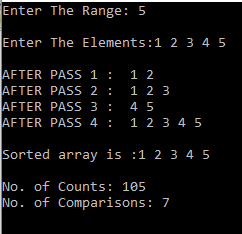
}

Output:

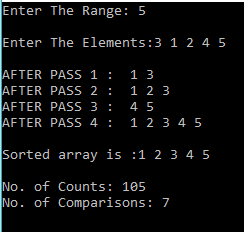
* Best case :



* Average case :



* Worst case :



Graph:

* Best case :

|  |  |  |  |
| --- | --- | --- | --- |
| Merge Sort (Best case) | | | |
| Sr No. | No Of Inputs | Count | Comparisons |
| 1 | 5 | 105 | 7 |
| 2 | 7 | 167 | 11 |
| 3 | 10 | 272 | 19 |

Time complexity :O(n log(n))

* Average case :

|  |  |  |  |
| --- | --- | --- | --- |
| Merge Sort(Average case) | | | |
| Sr No. | No Of Inputs | Count | Comparisons |
| 1 | 5 | 106 | 8 |
| 2 | 7 | 169 | 13 |
| 3 | 10 | 275 | 22 |

Time complexity : O(n log(n))

* Worst case :

|  |  |  |  |
| --- | --- | --- | --- |
| Merge Sort (Worst case) | | | |
| Sr No. | No Of Inputs | Count | Comparisons |
| 1 | 5 | 105 | 7 |
| 2 | 7 | 168 | 12 |
| 3 | 10 | 274 | 21 |

Time complexity : O(n log(n))

**3.2 Quick Sort**

Theory:

* Quicksort (sometimes called partition-exchange sort) is an [efficient](https://en.wikipedia.org/wiki/Algorithm_efficiency) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm), serving as a systematic method for placing the elements of an [array](https://en.wikipedia.org/wiki/Array_data_structure) in order.
* Quicksort is a [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort), meaning that it can sort items of any type for which a "less-than" relation (formally, a [total order](https://en.wikipedia.org/wiki/Total_order)) is defined.
* In efficient implementations it is not a [stable sort](https://en.wikipedia.org/wiki/Stable_sort), meaning that the relative order of equal sort items is not preserved.
* Quicksort can operate [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) on an array, requiring small additional amounts of [memory](https://en.wikipedia.org/wiki/Main_memory) to perform the sorting.

Example:

Input: 65 70 75 80 85 60 55 50 45

P: 65 i

* Pass 1: 65 70 75 80 85 60 55 50 45

i j swap (A[i], A[j])

(i) 65 45 75 80 85 60 55 50 70

i j swap (A[i], A[j])

(ii) 65 45 50 80 85 60 55 75 70

i j swap (A[i], A[j])

(iii) 65 45 50 55 85 60 80 75 70

i j swap (A[i], A[j])

(iv) 65 45 50 55 60 85 80 75 70

j i if (i>=j) break 60 45 50 55 65 85 80 75 70 swap (A[left], A[j])

Result of Pass 1: 3 60 45 50 55 65 85 80 75 70

* Pass 2a (left sub-block): 60 45 50 55 (P = 60)

i j

1. 60 45 50 55

j i if (i>=j) break 55 45 50 60 swap (A[left], A[j])

* Pass 2b (right sub-block): 85 80 75 70 (P = 85)

i j

1. 85 80 75 70

j i if (i>=j) break 70 80 75 85 swap (A[left], A[j])

sorted series : 45 50 55 60 70 75 80 85

Algorithm:

QUICK\_SORT(K,LB,UB)

1. [INITIALIZE]

FLAG ← true

2. [PERFORM SORT]

If LB&lt;UB

Then I ← LB

J ← UB+1

KEY ← K[LB]

Repeat while FLAG

I ← I+1

Repeat while K[I]&lt;KEY

I ← I+1

J ← J-1

Repeat while K[J]&gt;KEY

J ← J-1

If I&lt;J

Then K[I] ↔ K[J]

Else FLAG ← false

K[LB] ↔ K[J]

Call QUICK\_SORT(K,LB,J-1)

Call QUICK\_SORT(K,J+1,UB)

3. [FINISHED]

Return

Code:

#include<stdio.h>

int count=0, swap=0, comp=0;

void main(){

int \*a, i, j, n, start ,end;

printf("Enter The Range:");

scanf("%d",&n);

a = (int \*) malloc(sizeof(int)\*n);

printf("\nEnter The Elements:");

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

scanf("%d", (a + i));

}

start=0;

end=n-1;

printf("\n Sublists: \n");

quicksort(a, start, end);

printf("\nNo. of Counts: %d", count);

printf("\nNo. of Swaps: %d", swap);

printf("\nNo. of Comparisons: %d", comp);

printf("\n\nSorted Array:");

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

printf(" %d",\*(a + i));

}

}

void quicksort(int \*a, int start, int end){

int flag=0, i, j, pivot, temp, m;

pivot = \*(a + start);

i=start+1;

j=end;

count+=5;

if(start<end){

count++;

while(flag==0){

count++;

while(pivot>\*(a + i)) { i++;count++; }

count++;

while(pivot<\*(a + j)) { j--;count++; }

count++;

comp++;

if(i<j){

temp=\*(a + i);

\*(a + i)=\*(a + j);

\*(a + j)=temp;

count+=3;

swap++;

}

else { flag=1;count+=2; }

}

swap++;

temp=\*(a + start);

\*(a + start)=\*(a + j);

\*(a + j)=temp;

count+=3;

printf("Pivot: %d\n", pivot);

printf("List 1: [ ");

for(m=start; m<=j-1; m++) { printf("%d ",\*(a + m)); }

printf("]\n");

printf("List 2: [ ");

for(m=j+1; m<=end; m++) { printf("%d ",\*(a + m)); }

printf("]\n\n");

quicksort(a,start,j-1);

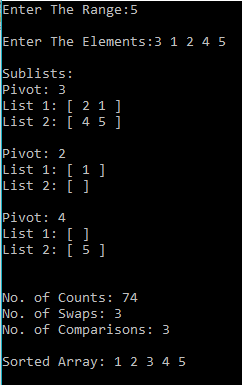
quicksort(a,j+1,end);

count+=2;

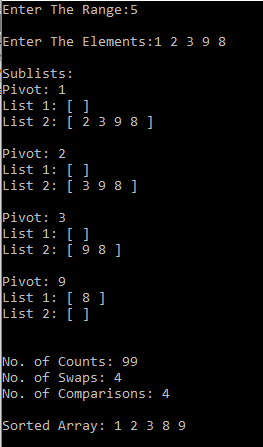
}

Output:

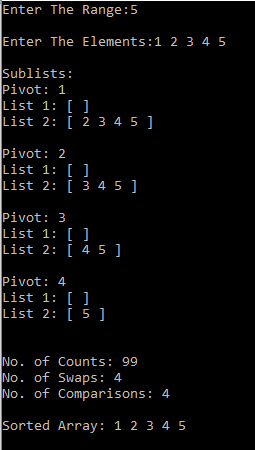
* Best case :



* Average case :



* Worst case :



Graph:

* Best case :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Sort (Best case) | | | | |
| Sr No. | No Of Inputs | Count | Swap | Comparisons |
| 1 | 5 | 74 | 3 | 3 |
| 2 | 7 | 78 | 3 | 3 |
| 3 | 10 | 162 | 7 | 7 |

Time complexity : O(n log(n))

* Average case :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Sort (Average case) | | | | |
| Sr No. | No Of Inputs | Count | Swap | Comparisons |
| 1 | 5 | 86 | 5 | 5 |
| 2 | 7 | 110 | 5 | 5 |
| 3 | 10 | 170 | 9 | 9 |

Time complexity : O(n log(n))

* Worst case:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Quick Sort (Worst case) | | | | |
| Sr No. | No Of Inputs | Count | Swap | Comparisons |
| 1 | 5 | 99 | 4 | 4 |
| 2 | 7 | 152 | 6 | 6 |
| 3 | 10 | 239 | 9 | 9 |

Time complexity : O(n^2)

**Time complexity:**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No. | Name | Theorical | Practical |
| 1 | Merge Sort – Best case | O(n log(n)) | O(n log(n)) |
| 2 | Merge Sort – Worst case | O(n log(n)) | O(n log(n)) |
| 3 | Merge Sort – Average case | O(n log(n)) | O(n log(n)) |
| 4 | Quick Sort – Best case | O(n log(n)) | O(n log(n)) |
| 5 | Quick Sort – Worst case | O(n^2) | O(n^2) |
| 6 | Quick Sort – Average case | O(n log(n)) | O(n log(n)) |

**Conclusion**: From this practical, we have learnt time complexity of merge sort and quick sort and their implementations.

**PRACTICAL – 4**

**Aim:** Greedy Approach

4.1 Making change problem (Implement)

4.2 0/1 and Fractional Knapsack problem (Study/Implement)

4.3 Activity Selection Problem (Study/Implement)

**Software Required:** CodeBlocks

**Hardware Required:** NA

**Knowledge Required:** Basic knowledge of c, c++

**4.1 Making change problem**

Theory:

* Coin values can be modeled by a set of *n* distinct positive [integer](https://en.wikipedia.org/wiki/Integer) values (whole numbers), arranged in increasing order as *w*1 = 1 through *wn*.
* The problem is: given an amount *W*, also a positive integer, to find a set of non-negative (positive or zero) integers {*x*1, *x*2, ..., *xn*}, with each *xj* representing how often the coin with value *wj* is used, which minimize the total number of coins

Σ xi subject to Σ wi xi = W

Example:

* Available coins are : 5,10,25,50
* Required amount is 140
* Coins required 50-2 , 25-1, 10-1 , 5-1

Algorithm:

1. C ← {100, 25, 10, 5, 1}
2. Sol ← {};
3. Sum ← 0 sum of item in solution set
4. WHILE sum not = n
5. x = largest item in set C such that sum + x ≤ n
6. IF no such item THEN
7. RETURN    "No Solution"
8. S ← S {value of x}
9. sum ← sum + x
10. RETURN S

Code:

#include<stdio.h>

#include<conio.h>

int main(){

int c[50],i,q[50],o,n,m,s=0,w[50];

printf("Enter no. of coins:");

scanf("%d",&n);

printf("Enter coins available:");

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

scanf("%d",&c[i]);

}

printf("Enter quantity of each coin:");

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

scanf("%d",&q[i]);

}

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

w[i]=0;

}

printf("enter amount:");

scanf("%d",&o);

if((o%5)==0){

printf("valid");

i=0;

while(s!=o&&i<=n){

if((o-s)>=c[i]){

s=s+c[i];

w[i]++;

}

else{

i++;

}

}

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

printf("\ncoins of %d is %d",c[i],w[i]);

}

}

else{

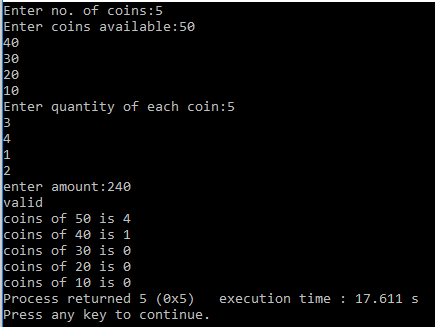
printf("\ninvalid amount");

}

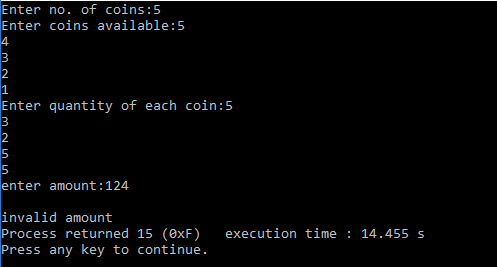
}

Output:

* Valid



* Invalid



**4.2 0/1 and Fractional Knapsack problem**

* 0/1 knapsack

Theory:

* The most common problem being solved is the 0-1 knapsack problem, which restricts the number xi of copies of each kind of item to zero or one. Given a set of n items numbered from 1 up to n, each with a weight wi and a value vi, along with a maximum weight capacity W,

Maximize Σ vi xi subject to Σ wi xi ≤ W and xi = {0, 1}

* Here xi represents the number of instances of item i to include in the knapsack. Informally, the problem is to maximize the sum of the values of the items in the knapsack so that the sum of the weights is less than or equal to the knapsack's capacity.

Example:

|  |  |  |
| --- | --- | --- |
| I | P | W |
| 1 | 12 | 3 |
| 2 | 8 | 2 |
| 3 | 10 | 3 |
| 4 | 20 | 4 |
| 5 | 18 | 1 |

Total weight=10

Maximum Profit Minimum Weight

I=4 P=20 W=10-4=6 I=5 P=18 W=10-1=9

I=5 P=20+18=38 W=6-1=5 I=2 P=18+8=26 W=9-2=7

I=1 P=38+12=50 W=5-3=2 I=1 P=26+12=38 W=7-3=4

I=2 P=50+8=58 W=2-2=0 I=3 P=38+10=48 W=4-3=1

Knapsack = {1,1,0,1,1} Knapsack = {1,1,1,0,1}

Algorithm:

1. FOR w = 0 TO W
2. DO  c[0, w] = 0
3. FOR i=1 to n
4. DO c[i, 0] = 0
5. FOR w=1 TO W
6. DO IF wi ≤ w
7. THEN IF  vi + c[i-1, w-wi] > c[i-1,w]
8. THEN c[i, w] = vi + c[i-1, w-wi]
9. ELSE c[i, w] = c[i-1, w]
10. ELSE
11. c[i, w] = c[i-1, w]

Code:

#include<stdio.h>

void main(){

int \*w,\*p,\*no,temp,n,i,j,limit,weight=0,profit=0,\*flag;

printf("Enter no of items : ");

scanf("%d",&n);

no=(int \*) malloc(sizeof(int)\*n);

w=(int \*) malloc(sizeof(int)\*n);

p=(int \*) malloc(sizeof(int)\*n);

flag=(int \*) malloc(sizeof(int)\*n);

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

printf("Enter weight and profit of item %d : ",i+1);

scanf("%d %d",&w[i],&p[i]);

no[i]=i+1;

}

printf("\nEnter the Maximum Weight : ");

scanf("%d",&limit);

for(i=0;i<n-1;i++){

for(j=0;j<n-i-1;j++){

if(p[j]<p[j+1]){

temp=p[j];

p[j]=p[j+1];

p[j+1]=temp;

temp=w[j];

w[j]=w[j+1];

w[j+1]=temp;

temp=no[i];

no[i]=no[i+1];

no[i+1]=temp;

}

}

}

i=0;

while(weight<limit && i<n){

if(weight+w[i]<=limit){

weight+=w[i];

profit+=p[i];

}

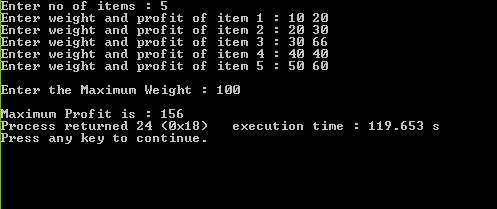
i++;

}

printf("\nMaximum Profit is : %d",profit);

}

Output:



* Fractional knapsack

Theory:

* An instance of either the fractional knapsack problems may be specified by the numerical capacity W of the knapsack, together with a collection of materials, each of which has two numbers associated with it: the weight wi of material that is available to be selected and the value per unit weight vi of that material. The goal is to choose an amount xi ≤ wi of each material, subject to the capacity constraint

Σ xi ≤ W and maximizing total benefit Σ xi.vi

* In the fractional knapsack problem, each of the amounts xi must be either zero or wi the continuous knapsack problem differs by allowing xi to range continuously from zero to wi.Some formulations of this problem rescale the variables xi to be in the range from 0 to 1.

Example:

|  |  |  |  |
| --- | --- | --- | --- |
| I | P | W | P/W |
| 1 | 14 | 3 | 4.67 |
| 2 | 22 | 5 | 4.4 |
| 3 | 10 | 3 | 3.33 |
| 4 | 20 | 4 | 5 |
| 5 | 18 | 2 | 9 |

Maximum Weight = 10

I=5 P=18 W=10-1=9

I=4 P=18+20=38 W=9-4=5

I=1 P=38+14=42 W=5-3=2

I=2 P=42+((2/5)22)=50.8 W=2-2=0

Maximum Profit = 50.8

Fractional Knapsack = {1,1,0,0.4,1}

Algorithm:

1. FOR i =1 to n
2. do x[i] =0
3. weight = 0
4. while i ≤ n & weight < W
5. do i = best remaining item
6. IF weight + w[i] ≤ W
7. then x[i] = 1
8. weight = weight + w[i]
9. Else
10. x[i] = (w - weight) / w[i]
11. weight = W
12. i++
13. return x

Code:

#include<stdio.h>

#include<conio.h>

int main(){

int n,m,i,j;

float p[100],w[100],pw[100],a[100],temp,pv=0,x[100];

printf("enter no. of process:");

scanf("%d",&n);

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

x[i]=0;

}

printf("enter processes:");

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

scanf("%f",&a[i]);

}

printf("enter profit values:");

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

scanf("%f",&p[i]);

}

printf("enter weights:");

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

scanf("%f",&w[i]);

}

printf("\n enter max weight:");

scanf("%d",&m);

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

pw[i]=(p[i]/w[i]);

}

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

for(j=0;j<n;j++){

if(pw[j]<pw[j+1]){

temp=pw[j];

pw[j]=pw[j+1];

pw[j+1]=temp;

temp=p[j];

p[j]=p[j+1];

p[j+1]=temp;

temp=a[j];

a[j]=a[j+1];

a[j+1]=temp;

temp=w[j];

w[j]=w[j+1];

w[j+1]=temp;

}

}

}

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

if(w[i]>m){

pv=pv+(p[i]\*(m/w[i]));

x[i]=m/w[i];

break;

}

else{

m=m-w[i];

pv=pv+p[i];

x[i]=1.0;

}

}

printf("\nweight\tprofit\tratio");

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

printf("\n%.2f\t%.2f\t%.2f",w[i],p[i],pw[i]);

}

printf("\n knapsack:");

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

printf("%.2f\n",x[i]);

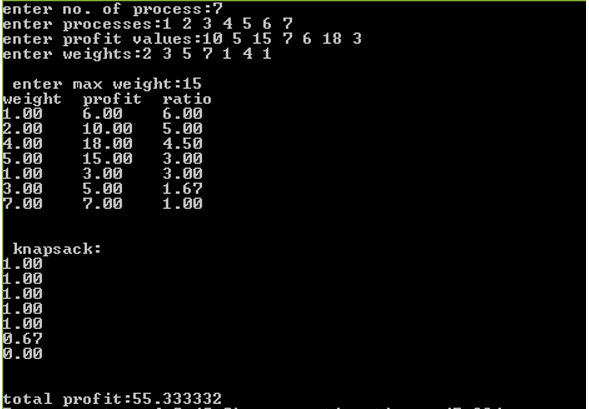
}

printf("\n\ntotal profit:%f",pv);

return 0;

}

Output:



* Activity Selection Problem:

Theory:

* The activity selection problem is a [combinatorial optimization](https://en.wikipedia.org/wiki/Combinatorial_optimization) problem concerning the selection of non-conflicting [activities](https://en.wikipedia.org/wiki/Task_(project_management)) to perform within a given [time frame](https://en.wikipedia.org/wiki/Time_frame), given a set of activities each marked by a start time (si) and finish time (fi). The problem is to select the maximum number of activities that can be performed by a single person or [machine](https://en.wikipedia.org/wiki/Machine), assuming that a person can only work on a single activity at a time.
* Assume there exist *n* activities with each of them being represented by a start time *si* and finish time *fi*. Two activities *i* and *j* are said to be non-conflicting if *si* ≥ *fj* or *sj* ≥ *fi*. The activity selection problem consists in finding the maximal solution set (S) of non-conflicting activities, or more precisely there must exist no [solution set](https://en.wikipedia.org/wiki/Solution_set) S' such that |S'| > |S| in the case that multiple maximal solutions have equal sizes.

Algorithm:

1 Greedy-Iterative-Activity-Selector(A, s, f):

2 Sort A by finish times stored in f'

3 S = {A[1]}

4 k = 1

5 n = A.length

6 for i = 2 to n:

7 if s[i] ≥ f[k]:

8 S = S U {A[i]}

9 k = i

10 return S

Code:

#include<stdio.h>

int main(){

int n,s[10],f[10],i,a[50],temp,j,k=0;

printf("Enter number of activities::\n");

scanf("%d",&n);

printf("Enter start time and finish time for each activity::\n");

printf("Activity\tStart Time\t Finish Time\n");

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

a[i]=i+1;

printf("A[%d]\t",a[i]);

scanf("%d",&s[i]);

scanf("%d",&f[i]);

}

for(i=n-2;i>=0;i--){

for(j=0;j<=i;j++){

if(f[j]>f[j+1]){

temp=f[j+1];

f[j+1]=f[j];

f[j]=temp;

temp=s[j+1];

s[j+1]=s[j];

s[j]=temp;

temp=a[j+1];

a[j+1]=a[j];

a[j]=temp;

}

}

}

printf("\nArranging the activity in increasing order of finish time\n\n");

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

printf("A[%d]\t",a[i]);

printf("%d\t",s[i]);

printf("%d\n",f[i]);

}

printf("The final activity schedule is ::\n");

printf("\nA[%d]\t",a[0]);

for(i=1;i<n;i++){

if(s[i]>=f[k]){

printf("A[%d]\t",a[i]);

k=i;

}

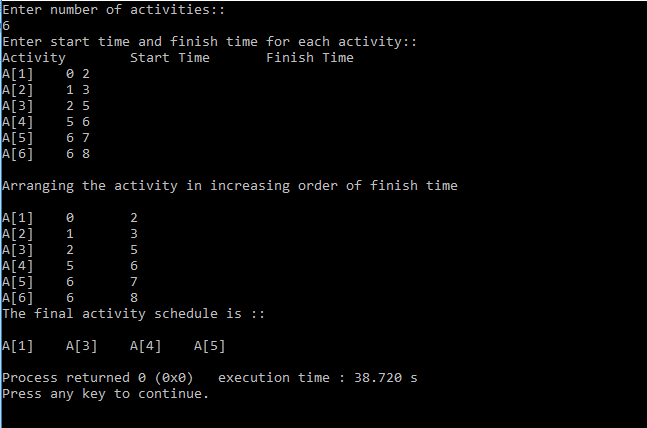
}

printf("\n");

return 0;

}

Output:



**Conclusion**: From this practical we have learnt about Making change problem, 0/1 and Fractional Knapsack problem and Activity Selection problem.

**PRACTICAL – 5**

**Aim:** Dynamic Programming

5.1 Find binomial Co-efficient using Dynamic Programming (Implement)

5.2 0/1 Knapsack Problem (Implement)

5.3 Matrix Chain Multiplication (Implement)

5.4 Longest Common Subsequence (Implement)

**5.1 Find binomial Co-efficient using Dynamic Programming. (Implement)**

Theory:

* A binomial coefficient is any of the positive integers that occur as coefficients in the binomial theorem. They are indexed by two nonnegative integers; the binomial coefficient indexed by n and k is usually written nCc . It is the coefficient of the x k term in the polynomial expansion of the binomial power (1 + x)n.
* Under suitable circumstances the value of the coefficient is given by the expression n! / k! (n-k)!. Arranging binomial coefficients into rows for successive values of n, and in which k ranges from 0 to n, gives a triangular array called Pascal's triangle.
* Time complexity of binomial co-efficient is O(n k).
* C(n, k), where n ≥ k ≥ 0

Example:

C(3,2) = 3

|  |  |  |
| --- | --- | --- |
|  | 1 | 2 |
| 1 | 1 |  |
| 2 | 2 | 1 |
| 3 | 3 | 3 |

Algorithm:

1. C(n , k) = C (n-1, k-1) + C (n-1, k) for n > k > 0
2. C(n , 0) = 1,
3. C(n , n) = 1 for n ≥ 0

Code:

#include<stdio.h>

int main(){

int k,n;

printf("Enter n: ");

scanf("%d",&n);

printf("Enter k: ");

scanf("%d",&k);

printf("C(%d,%d) id %d",n,k,B\_C(n,k));

return 0;

}

int B\_C(int n,int k){

if(k==0||k==n){

return 1;

}

else{

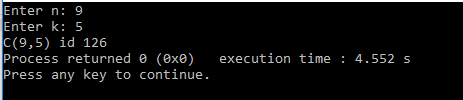
return B\_C(n-1,k-1)+B\_C(n-1,k);

}

return 0;

}

Output:



**5.2 0/1 Knapsack Problem. (Implement)**

Theory:

* The knapsack problem or rucksack problem is a problem in [combinatorial optimization](https://en.wikipedia.org/wiki/Combinatorial_optimization): Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
* It derives its name from the problem faced by someone who is constrained by a fixed-size [knapsack](https://en.wikipedia.org/wiki/Knapsack) and must fill it with the most valuable items.
* The problem often arises in [resource allocation](https://en.wikipedia.org/wiki/Resource_allocation) where there are financial constraints and is studied in fields such as [combinatorics](https://en.wikipedia.org/wiki/Combinatorics" \o "Combinatorics), [computer science](https://en.wikipedia.org/wiki/Computer_science), [complexity theory](https://en.wikipedia.org/wiki/Computational_complexity_theory), [cryptography](https://en.wikipedia.org/wiki/Cryptography), [applied mathematics](https://en.wikipedia.org/wiki/Applied_mathematics), and [daily fantasy sports](https://en.wikipedia.org/wiki/Daily_fantasy_sports).
* Time complexity is O(W n).

Example:

W=5

|  |  |  |
| --- | --- | --- |
| Item | Weight | Value |
| 1 | 2 | 3 |
| 2 | 3 | 4 |
| 3 | 4 | 5 |
| 4 | 5 | 6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** |
| **0** | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 0 | 0 | 3 | 3 | 3 | 3 |
| **2** | 0 | 0 | 3 | 4 | 4 | 7 |
| **3** | 0 | 0 | 3 | 4 | 5 | 7 |
| **4** | 0 | 0 | 3 | 4 | 5 | 7 |

Algorithm:

1 for j from 0 to W do:

2 m[0, j] := 0

3 for i from 1 to n do:

4 for j from 0 to W do:

5 if w[i] > j then:

6 m[i, j] := m[i-1, j]

7 else:

8 m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])

Code:

#include<stdio.h>

#include<conio.h>

int sum=0;

int max(int a,int b){

if(a>b)

return a;

else

return b;

}

void knapsack(int m,int n,int w[],int p[]){

int v[100][200],x[10],i,j,count=0;

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

v[0][i]=0;

for(i=1;i<=n;i++){

for(j=0;j<=m;j++){

if(j>=w[i])

v[i][j]=max(v[i-1][j],v[i-1][j-w[i]]+p[i]);

else

v[i][j]=v[i-1][j];

}

}

for(i=1;i<=n;i++)

x[i]=0;

i=n;

j=m;

while(i>0 && j>0){

if(v[i][j]!=v[i-1][j]){

x[i]=1;

j=j-w[i];

}

i--;

}

printf("\n Items in the KnapSack solution are : \n\n");

printf("\tSr.no \tweight profit\n");

printf(" ----------------------------------------\n");

for (i = 1; i <= n; i++)

if (x[i])

printf("\t %d \t %d \t %d\n", ++count, w[i], p[i]);

printf("\n The knapsack solution : ");

for(i=1;i<=n;i++){

if(x[i]==1){

printf("1 ",i);

sum=sum+p[i];

}

else

printf("0 ",i);

}

printf("\n\n Total profit = %d",sum);

}

void main()

{

int w[10],p[10],i,m,n;

printf("\n Enter the number of items : ");

scanf("%d",&n);

printf("\n Enter the weight : ");

for(i=1;i<=n;i++)

scanf("%d",&w[i]);

printf("\n Enter the profits for items : ");

for(i=1;i<=n;i++)

scanf("%d",&p[i]);

printf("\n Enter the capacity of knapsack : ");

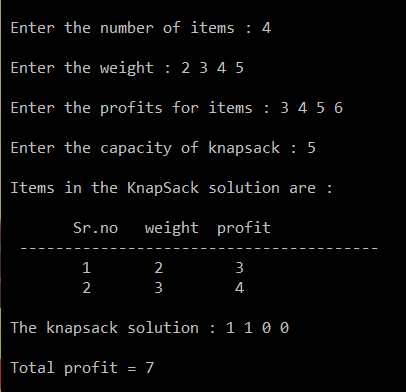
scanf("%d",&m);

knapsack(m,n,w,p);

getch();

}

Output :



**5.3 Matrix Chain Multiplication (Implement)**

Theory:

* Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an [optimization problem](https://en.wikipedia.org/wiki/Optimization_problem) that can be solved using [dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming). Given a sequence of matrices, the goal is to find the most efficient way to [multiply these matrices](https://en.wikipedia.org/wiki/Matrix_multiplication). The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved.
* Ways to multiply A, B, C and D matrix:

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

* Answer will be the same, but the efficiency of the calculations will be different.
* Time Complexity is O(n3).

Example:

Dimensions: 5 10 6 5 3 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 300/1 | 450/2 | 420/1 | 310/1 |
| 2 | -1 | 0 | 300/2 | 270/2 | 210/2 |
| 3 | -1 | -1 | 0 | 90/3 | 90/3 |
| 4 | -1 | -1 | -1 | 0 | 30/4 |
| 5 | -1 | -1 | -1 | -1 | 0 |

Solution: (A(B(C(D E))))

Algorithm:

1 MatrixChainOrder (int dims[])

2 n = dims.length - 1;

3 for (i = 1; i <= n; i++) m[i, i] = 0;

4 for (len = 2; len <= n; len++)

5 for (i = 1; i <= n - len + 1; i++)

6 j = i + len - 1;

7 m[i, j] = MAXINT;

8 for (k = i; k <= j - 1; k++)

9 cost = m[i, k] + m[k+1, j] + dims[i-1]\*dims[k]\*dims[j];

10 if (cost < m[i, j])

11 m[i, j] = cost;

12 s[i, j] = k;

Code:

#include<stdio.h>

#define max\_int 10000

int s[10][10];

void matrix(int p[], int n){

int l,i, j,k, temp;

int m[n][n];

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

for(j=0; j<n; j++){

m[i][j] = 0;

s[i][j] = 0;

}

}

for(l=2; l<n; l++){

for(i=1; i<n-l+1; i++){

j = i+l-1;

m[i][j] = max\_int;

for(k=i; k<j; k++){

temp = m[i][k] + m[k+1][j] + p[i-1] \* p[k] \* p[j];

if(temp < m[i][j]){

m[i][j] = temp;

s[i][j] = k;

}

}

}

}

printf("\n Value Matrix:\n\n");

for(i=1; i<n; i++){

printf("\t");

for(k=1; k<n; k++){

printf("%d ", m[i][k]);

}

printf("\n");

}

printf("\n K values:\n\n");

for(i=1; i<n; i++){

printf("\t");

for(k=1; k<n; k++){

printf("%d ", s[i][k]);

}

printf("\n");

}

}

void solution(int i,int j){

if (i == j)

printf(" A%d",i);

else{

printf(" (");

solution(i, s[i][j]);

solution(s[i][j] + 1, j);

printf(" )");

}

}

int main(){

int n,p[5],m,i,k,j;

printf("\n Enter size of matrix : ");

scanf("%d",&m);

k=m+1;

printf("\n Enter values of p : ");

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

scanf(" %d",&p[i]);

matrix(p,k);

printf("\n Sequence:");

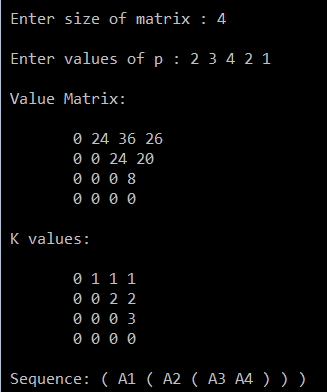
i=1,j=m;

solution(i,j);

return 0;

}

Output:



**5.4 Longest Common Subsequence (Implement)**

Theory:

* The longest common subsequence (LCS) problem is the problem of finding the longest [subsequence](https://en.wikipedia.org/wiki/Subsequence) common to all sequences in a set of sequences (often just two sequences).
* It differs from problems of finding common [substrings](https://en.wikipedia.org/wiki/Substring): unlike substrings, subsequences are not required to occupy consecutive positions within the original sequences.
* The longest common subsequence problem is a classic [computer science](https://en.wikipedia.org/wiki/Computer_science) problem, the basis of [data comparison](https://en.wikipedia.org/wiki/Data_comparison) programs such as the [diff utility](https://en.wikipedia.org/wiki/Diff_utility), and has applications in [bioinformatics](https://en.wikipedia.org/wiki/Bioinformatics).
* It is also widely used by [revision control systems](https://en.wikipedia.org/wiki/Revision_control) such as [git](https://en.wikipedia.org/wiki/Git_(software)" \o "Git (software)) for [reconciling](https://en.wikipedia.org/wiki/Merge_(revision_control)) multiple changes made to a revision-controlled collection of files.

Example:

String 1: LMNOP

String 2: LNOP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1(L) | 2(M) | 3(N) | 4(O) | 5(P) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1(L) | 0 | 1 ↖ | 1 ← | 1 ← | 1 ← | 1 ← |
| 2(N) | 0 | 1 ↑ | 1 ↑ | 2 ↖ | 2 ← | 2 ← |
| 3(O) | 0 | 1 ↑ | 1 ↑ | 2 ↑ | 3 ↖ | 3 ← |
| 4(P) | 0 | 1 ↑ | 1 ↑ | 2 ↑ | 3 ↑ | 4 ↖ |

LCS: LNOP

Algorithm:

1 LCS (X,Y)

2 m <- length[X]

3 n <- length[Y]

4 for I <- 1 to m

5 do c[i, 0] <- 0

6 for j <- 0 to n

7 do c[0, j] <- 0

8 for i <- 1 to m

9 do for j <- 1 to n

10 do if xi = yj

11 then c [i, j] <- c [i-1, j-1] + 1

12 b [i, j] <- “↖”

13 else if c [i-1, j] ≥ c [i, j-1]

14 then c [i, j] <- c [i-1, j]

15 b [i, j] <- “↑”

16 else c [i, j] <- c [i, j-1]

17 b [i, j] <- “←”

18 return c and b

Code:

#include<stdio.h>

int cmatrix[100][100];

char bmatrix[100][100], str[50], str1[50];

void main(){

int i,j,m,n,temp,temp1;

printf("\n Enter First Sequence : ");

scanf("%s",str);

fflush(stdin);

printf("\n Enter Second Sequence : ");

scanf("%s",str1);

fflush(stdin);

m=strlen(str);

n=strlen(str1);

for(i=0;i<=m;i++){

for(j=0;j<=n;j++){

if(i==0 || j==0){

cmatrix[i][j]=0;

bmatrix[i][j]='n';

}

else if(str[i-1]==str1[j-1]){

cmatrix[i][j]=cmatrix[i-1][j-1]+1;

bmatrix[i][j]='c';

}

else if(str[i-1]!=str1[j-1]){

if(cmatrix[i][j-1]>cmatrix[i-1][j]){

cmatrix[i][j]=cmatrix[i][j-1];

bmatrix[i][j]='l';

}

else{

cmatrix[i][j]=cmatrix[i-1][j];

bmatrix[i][j]='u';

}

}

}

}

printf("\n\n String Matrix:\n\n");

for(i=0;i<=m;i++){

printf("\t");

for(j=0;j<=n;j++){

printf("%d ",cmatrix[i][j]);

}

printf("\n");

}

printf("\n\n Arrow Matrix:\n\n");

for(i=0;i<=m;i++){

printf("\t");

for(j=0;j<=n;j++){

printf("%c ",bmatrix[i][j]);

}

printf("\n");

}

printf("\n LCS : ");

lcs(m,n);

}

void lcs(int i,int j){

if(i==0 || j==0){

return 0;

}

if (bmatrix[i][j]=='c'){

lcs(i-1,j-1);

printf("%c",str[i-1]);

}

else if (bmatrix[i][j]=='u'){

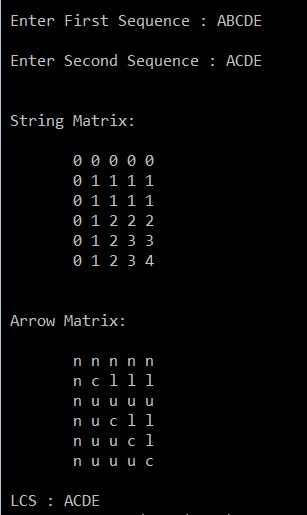
lcs(i-1,j);

}

else { lcs(i,j-1); }

}

Output :



**Conclusion:** From this practical, we have learnt about dynamic programming concepts for different problems and their implementation.

**PRACTICAL – 6**

**Aim:** Graph

6.1 Print all the nodes reachable from a given starting node in a diagraph

using BFS method.

6.2 Check whether the graph is connected or not using DFS method.

6.3 From a given vertex in a weighted connected graph, find shortest paths to

other vertices using Dijkstra’s algorithm (Implement)

6.4 Find Minimum Cost spanning tree of a given undirected graph using

Kruskal’s algorithm.

6.5 Find Minimum Cost spanning tree of a given undirected graph using

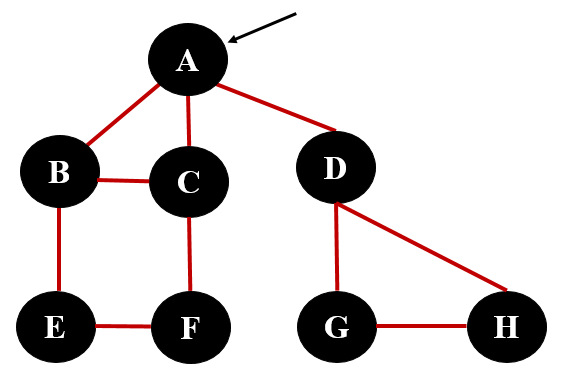
Prim’s algorithm.

**6.1 Print all the nodes reachable from a given starting node in a diagraph using BFS method.**

Theory:

* Breadth-first search (BFS) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures.
* It starts at the [tree root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) and explores the neighbor nodes first, before moving to the next level neighbors.
* The time complexity can be expressed as O(|V|+|E|), whereas the space complexity is O(|V|).
* If the graph is represented by an [adjacency list](https://en.wikipedia.org/wiki/Adjacency_list) it occupies O(|V|+|E|) space in memory, while an [adjacency matrix](https://en.wikipedia.org/wiki/Adjacency_matrix) representation occupies O(|V2|).

Example:



* BFS Sequence is : A B C D E F G H

Algorithm:

Algorithm BFS (G)

{

//Problem Description: This algorithm is for finding BFS.

Queue Q; // create a queue for storing the adjacent vertices

// visit [] is an array that keeps track of all the visited nodes.

// initially, the visit [] is initialized to 0

1. While ( G has an unvisited node) do
2. v <- an unvisited node;
3. visit [v] <- 1
4. en\_queue (v, Q); // add an element to the queue
5. While ( Q is not empty) do
6. x <- del\_queue ( Q) ; // delete an element from queue
7. For ( unvisited neighbor y of x ) do
8. visit [y] <- 1;
9. en\_queue ( v, Q) ; // add adjacent vertices of X to queue

Code:

#include<stdio.h>

#include<conio.h>

int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;

void bfs(int v)

{

for(i=1;i<=n;i++)

if(a[v][i] && !visited[i])

q[++r]=i;

if(f<=r)

{

visited[q[f]]=1;

bfs(q[f++]);

}

}

void main()

{

int v;

printf("\n Enter the number of vertices:");

scanf("%d",&n);

for(i=1;i<=n;i++)

{

q[i]=0;

visited[i]=0;

}

printf("\n Enter adjacency Matrix:\n\n");

for(i=1;i<=n;i++)

{

printf("\t");

for(j=1;j<=n;j++)

scanf("%d",&a[i][j]);

}

printf("\n Enter the starting vertex:");

scanf("%d",&v);

bfs(v);

printf("\n The sequence is:\n");

for(i=1;i<=n;i++)

if(visited[i])

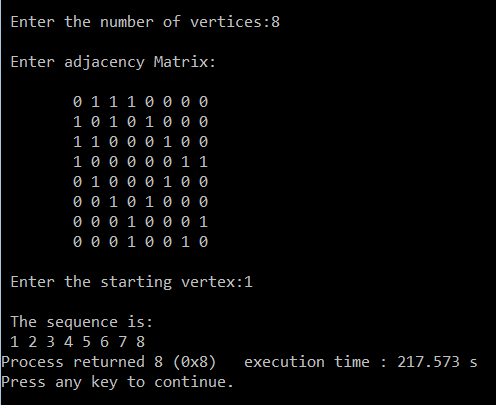
printf(" %d",i);

else

printf("\n BFS is not possible");

}

* Output :

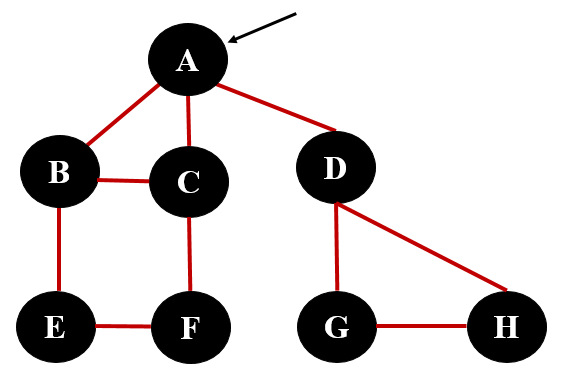


**6.2 Check whether the graph is connected or not using DFS method.**

Theory:

* Depth-first search (DFS) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures. One starts at the [root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before [backtracking](https://en.wikipedia.org/wiki/Backtracking).
* DFS is typically used to traverse an entire graph, and takes time O(|V| + |E|), linear in the size of the graph. In these applications it also uses space O(|V|) in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices.
* DFS may suffer from non-termination, in this case search is performed for a limited depth.

Example:



* DFS Sequence is : A B C F E D G H

Algorithm:

1. visit [v] = 1;
2. for ( each vertex x adjacent from v)
3. if (visit [x] = 0 ) then
4. DFS(x)

Code:

#include<stdio.h>

int TOP=-1;

int stack[10];

void push(int a);

int pop();

void main()

{

int n,in[10][10],i,j,x=0,q,flag=0,start,current,visit[10]={0},connected[10]={0},count=0,path[10];

printf("\n Enter The No Of Nodes : ");

scanf("%d",&n);

printf("\n Enter The Start Node : ");

scanf("%d",&start);

printf("\n Enter The Adjacency Matrix :\n\n");

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

{

printf("\t");

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

{

scanf(" %d",&in[i][j]);

}

}

q=n;

push(start-1);

visit[start-1]=1;

connected[start-1]=1;

while(q>0)

{

current=pop();

path[x]=current;

x++;

for(i=n-1;i>=0;i--)

{

if(in[current][i]==1 && visit[i]==0)

{

visit[i]=1;

connected[i]=1;

push(i);

}

}

q--;

}

while(TOP>=0)

{

current=pop();

visit[current]=1;

x--;

path[x]=current;

x++;

}

if(TOP==-1)

{

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

{

if(connected[i]==1)

{

count++;

}

}

}

if(count==n)

{

printf("\n Graph is Connected.\n\n The Sequence is : ");

for(i=0;i<n;i++) printf(" %d ",path[i]);

}

else

{

printf("\n Graph is Not Connected.");

}

}

void push(int a)

{

TOP++;

stack[TOP]=a;

}

int pop()

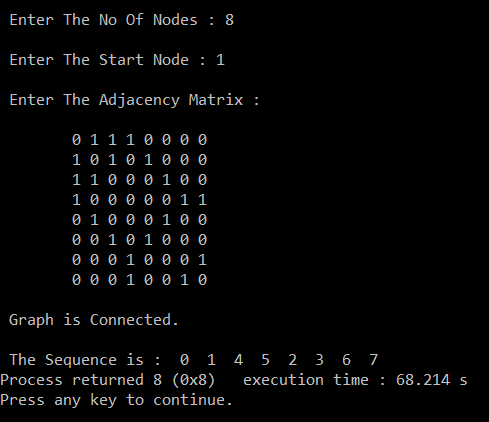
{

TOP--;

return stack[TOP+1];

}

Output:

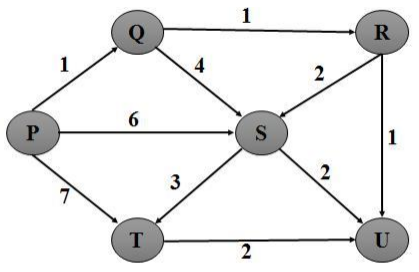


**6.3 From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra’s algorithm. (Implement)**

Theory:

* Dijkstra's algorithm is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for finding the [shortest paths](https://en.wikipedia.org/wiki/Shortest_path_problem) between [nodes](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) in a [graph](https://en.wikipedia.org/wiki/Graph_(abstract_data_type)), which may represent, for example, road networks.
* It was conceived by [computer scientist](https://en.wikipedia.org/wiki/Computer_scientist) [Edsger W. Dijkstra](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra" \o "Edsger W. Dijkstra).
* The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a [shortest-path tree](https://en.wikipedia.org/wiki/Shortest-path_tree).
* Time complexity is O(|V2|).

Example:



Source Vertex : 1

P Q R S T U

P 0 ∞ ∞ ∞ ∞ ∞

Q 1 ∞ 6 7 ∞

R 2 5 7 ∞

U 4 7 3

S 4 7

T 7

Algorithm:

Dijkstra (Graph, source)

step1: create vertex set Q

step2: for each vertex v in Graph: // Initialization

dist[v] ← INFINITY // Unknown distance from source to v

prev[v] ← UNDEFINED // Previous node in optimal path from source

add v to Q // All nodes initially in Q (unvisited nodes)

step3: dist[source] ← 0 // Distance from source to source

step4: while Q is not empty:

u ← vertex in Q with min dist[u] // Source node will be selected first

remove u from Q

step5: for each neighbor v of u: // where v is still in Q.

alt ← dist[u] + length (u, v)

if alt < dist[v]: // A shorter path to v has been found

dist[v] ← alt

prev[v] ← u

step6: return dist[], prev[]

Code:

#include<stdio.h>

void main()

{

int in[10][10], i, j, k=0, n, src, q, select[10], ans[10], dist=0,

solution[10][10], l=0, min=10000, p, count=0, f, flag=0, temp, count1=0;

printf(" Enter number Of Nodes:");

scanf("%d", &n);

src=0;

printf("\n Enter The Adjacency Matrix:\n\n");

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

{ printf("\t");

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

{

scanf("%d", &in[i][j]);

solution[i][j]=-5;

}

}

q=n;

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

{

solution[l][i]=-1;

select[i]=0;

}

select[src]=1;

solution[l][src]=0;

l++;

ans[k]=src+1;

k++;

while(q>0)

{

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

{

if(in[src][i]>0)

{

if(select[i]==0)

{

temp = dist + in[src][i];

if(temp<min)

{

min=temp;

p=i;

}

solution[l][i]=temp;

}

}

else

{

count++;

if(solution[l-1][i]==-1)

{

solution[l][i]=-1;

}

else if(select[i]==0)

{

solution[l][i]=solution[l-1][i];

}

}

}

ans[k]=p+1;

min=1000;

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

{

if(select[f]==1)

{

count1++;

}

}

if(count==n && count1==n-2)

{

src=ans[k-2]-1;

select[src]=0;

dist=dist-in[l-2][ans[k]-1];

count=0;

flag=1;

}

else

{

src=p;

dist=solution[l][p];

k++;

l++;

}

if(flag==0)

{

select[p]=1;

flag=0;

}

else

{

select[src]=1;

}

count=0;

count1=0;

q--;

}

printf("\n\n Solution:\n");

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

{

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

{

if(solution[i][j]!=-5)

{

printf("%4d",solution[i][j]);

}

else

{

printf(" ");

}

}

printf("\n");

}

printf("\n\n Path : ");

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

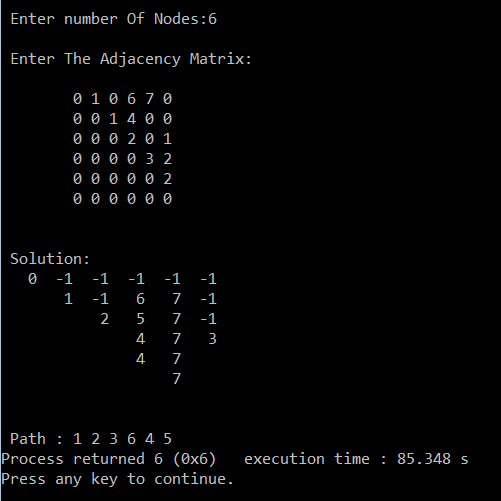
{

printf("%d ",ans[i]);

}

}

Output:

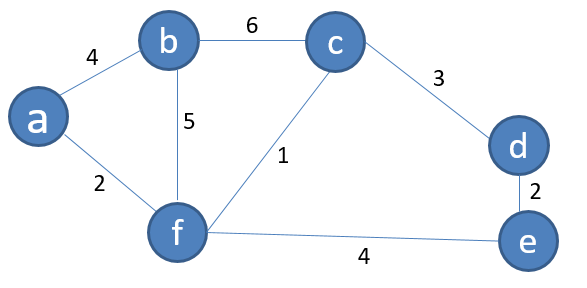


**6.4 Find Minimum Cost spanning tree of a given undirected graph using Kruskal’s algorithm.**

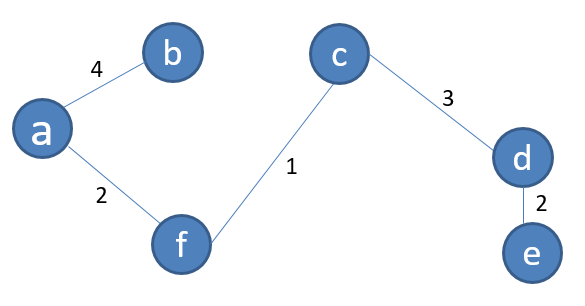
Theory:

* Kruskal's algorithm is a [minimum-spanning-tree algorithm](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Algorithms) which finds an edge of the least possible weight that connects any two trees in the forest.
* It is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) in [graph theory](https://en.wikipedia.org/wiki/Graph_theory) as it finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [connected](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)) [weighted graph](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#Weighted_graphs_and_networks) adding increasing cost arcs at each step. This means it finds a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a tree that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the total weight of all the edges in the tree is minimized.
* Time Complexity is O(E log V).

Example:



* Minimal Spanning Tree



Algorithm:

**MST-KRUSKAL *(G,w)***

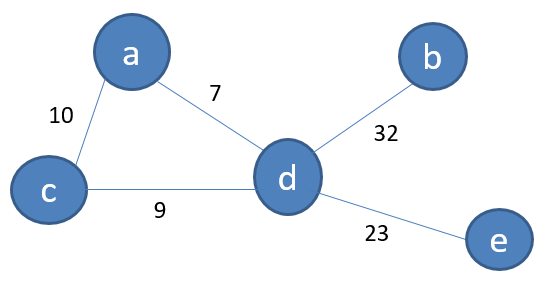
1. *A* ← ∅
2. **for** each vertex *v* ∈ *V*[*G*]
3. **do** MAKE-SET*(v)*
4. sort the edges of *E* into non-decreasing order by weight *w*
5. **for** each edge *(u, v)* ∈ *E*, taken in non-decreasing order by weight
6. **do if** FIND-SET*(u)* ≠ FIND-SET*(v)*
7. **then** *A* ← *A* ∪ {*(u, v)*}
8. UNION*(u, v)*
9. **return** *A*

**6.5 Find Minimum Cost spanning tree of a given undirected graph using Prim’s algorithm.**

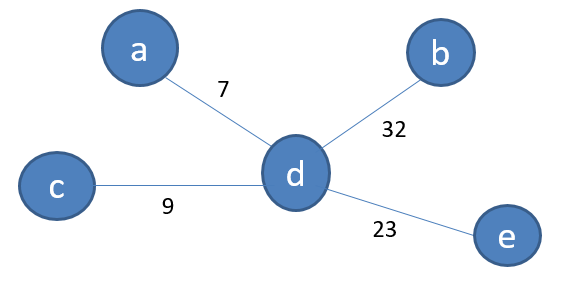
Theory:

* Prim's algorithm is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) that finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree) for a [weighted](https://en.wikipedia.org/wiki/Weighted_graph) [undirected graph](https://en.wikipedia.org/wiki/Undirected_graph).
* This means it finds a subset of the [edges](https://en.wikipedia.org/wiki/Edge_(graph_theory)) that forms a [tree](https://en.wikipedia.org/wiki/Tree_(graph_theory)) that includes every [vertex](https://en.wikipedia.org/wiki/Vertex_(graph_theory)), where the total weight of all the [edges](https://en.wikipedia.org/wiki/Graph_theory) in the tree is minimized.
* The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.
* Time Complexity is O(E log V).

Example:



* Minimal Spanning Tree



Algorithm:

**MST-PRIM*( G, w, r)***

1. ***A={}***
2. **for** each *u* ∈ *V*[*G*]
3. **do** *key*[*u*]←∞
4. *π*[*u*]← NIL
5. *key*[*r*] ← 0
6. *Q* ← *V*[*G*]
7. **while** *Q* = ∅
8. **do** *u* ← EXTRACT-MIN*(Q)*
9. *Q=Q-u*
10. *If(π*[*u*]!=NIL)
11. *A=AU(u, π*[*u*]*)*
12. **for** each *v* ∈ *Adj*[*u*]
13. **do if** *v* ∈ *Q* and *w(u, v) < key*[*v*]
14. **then** *π*[*v*]← *u*
15. *key*[*v*] ← *w(u, v)*

**Conclusion:** From this practical, we have learnt about various algorithms for finding MST, finding shortest path, searching in finite space and their complexities and implementations of them.

**PRACTICAL-7**

**Aim:** Backtracking

7.1 Eight Queen Problem (Implement)

**Software Required:** CodeBlocks

**Hardware Required:** NA

**Knowledge Required:** Basic knowledge of c, c++

Theory:

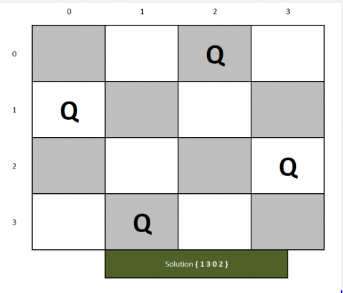
* Backtracking is a general [algorithm](https://en.wikipedia.org/wiki/Algorithm) for finding all (or some) solutions to some [computational problems](https://en.wikipedia.org/wiki/Computational_problem), notably [constraint satisfaction problems](https://en.wikipedia.org/wiki/Constraint_satisfaction_problem), that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.
* Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution.
* It is useless, for example, for locating a given value in an unordered table.
* Backtracking is an important tool for solving [constraint satisfaction problems](https://en.wikipedia.org/wiki/Constraint_satisfaction_problem), such as [crosswords](https://en.wikipedia.org/wiki/Crosswords), [verbal arithmetic](https://en.wikipedia.org/wiki/Verbal_arithmetic), [Sudoku](https://en.wikipedia.org/wiki/Algorithmics_of_sudoku), and many other puzzles. It is often the most convenient technique for [parsing](https://en.wikipedia.org/wiki/Parsing), for the [knapsack problem](https://en.wikipedia.org/wiki/Knapsack_problem) and other [combinatorial optimization](https://en.wikipedia.org/wiki/Combinatorial_optimization) problems. It is also the basis of the so-called [logic programming](https://en.wikipedia.org/wiki/Logic_programming) languages such as [Icon](https://en.wikipedia.org/wiki/Icon_programming_language), [Planner](https://en.wikipedia.org/wiki/Planner_programming_language) and [Prolog](https://en.wikipedia.org/wiki/Prolog).

Algorithm:

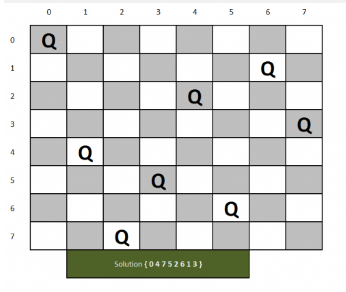
1. Place the queens col­umn wise, start from the left most column
2. If all queens are placed.
   1. Return true and print the solu­tion matrix.
3. Else
   1. Try all the rows in the cur­rent column.
   2. Check if queen can be placed here safely if yes mark the cur­rent cell in solu­tion matrix as 1 and try to solve the rest of the prob­lem recursively.
   3. If plac­ing the queen in above step leads to the solu­tion return true.
   4. If plac­ing the queen in above step does not lead to the solu­tion, BACKTRACK, mark the cur­rent cell in solu­tion matrix as 0 and return false.
4. If all the rows are tried and noth­ing worked, return false and print NO SOLUTION.

Example:

* 1. 4 Queens Problem Solution:



* 1. 8 Queens Problem Solution:



Code:

#include <iostream>

#include <cstdio>

#include <cstdlib>

#define N 8

using namespace std;

void printSolution(int board[N][N]){

for (int i = 0; i < N; i++) {

for (int j = 0; j < N; j++)

cout<<board[i][j]<<" ";

cout<<endl;

}

}

bool isSafe(int board[N][N], int row, int col){

int i, j;

for (i = 0; i < col; i++){

if (board[row][i])

return false;

}

for (i = row, j = col; i >= 0 && j >= 0; i--, j--){

if (board[i][j])

return false;

}

for (i = row, j = col; j >= 0 && i < N; i++, j--){

if (board[i][j])

return false;

}

return true;

}

bool solveNQUtil(int board[N][N], int col){

if (col >= N)

return true;

for (int i = 0; i < N; i++){

if ( isSafe(board, i, col) ){

board[i][col] = 1;

if (solveNQUtil(board, col + 1) == true)

return true;

board[i][col] = 0;

}

}

return false;

}

bool solveNQ(){

int board[N][N] = {0};

if (solveNQUtil(board, 0) == false){

cout<<"Solution does not exist"<<endl;

return false;

}

printSolution(board);

return true;

}

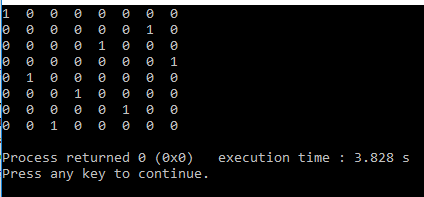
int main(){

solveNQ();

return 0;

}

Output:



**Conclusion:** From this practical, we have learnt how to solve an eight queen’s problem using backtracking.

**PRACTICAL-8**

**Aim:** 1) Naïve String Matching (Implement)

2) Rabin Karp (Implement)

**Software Required:** CodeBlocks

**Hardware Required:** NA

**Knowledge Required:** Basic knowledge of c, c++

Theory:

**8.1 Naïve String Matching. (Implement)**

- A simple but inefficient way to see where one string occurs inside another is to check each place it could be, one by one, to see if it’s there. So first we see if there’s a copy of the needle in the first character of the haystack; if not, we look to see if there’s a copy of the needle starting at the second character of the haystack; if not, we look starting at the third character, and so forth.

- In the normal case, we only have to look at one or two characters for each wrong position to see that it is a wrong position, so in the average case, this takes O(n + m) steps, where n is the length of the haystack and m is the length of the needle; but in the worst case, searching for a string like “aaaab” in a string like “aaaaaaaaab”, it takes O(nm).

Example:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | A | B | A | A | B | C | A | B | A | A |

* Text

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | A | A |

* Pattern
* Valid Shifts: s=3, s=9

Algorithm:

NAIVE-STRING- MATCHER (T, P)

step1: n length[T]

step2: m length[P]

step3: for s 0 to n - m

do if P[1 . . m] = T[s + 1 . . s + m]

then print &quot;Pattern occurs with shift&quot; s

Code:

#include<stdio.h>

void main(){

char t[100],p[100];

int tn,pn,shift[20]={0},s=0,i,j=0,count=0,m=0;

printf(&quot;\n Enter The Text : &quot;);

scanf(&quot;%s&quot;,t);

fflush(stdin);

printf(&quot;\n Enter The Pattern : &quot;);

scanf(&quot;%s&quot;,p);

tn = strlen(t);

pn = strlen(p);

while(s!=(tn-pn+1)){

j=0;

for(i=s;i&lt;pn+s;i++){

if(p[j]==t[i]){

count++;

if(count==pn){

count=0;

shift[m]=s;

m++;

}

}else{

count=0;

break;

}

j++;

}

s++;

}

if(m&gt;0){

printf(”\n\n Valid Shifts : “);

for(i=0;i&lt;m;i++) printf(“%d”,shift[i]);

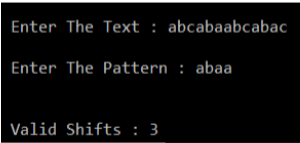
}else{

printf(“\n\n No Valid Shifts”);

}

}

Output:



* 1. **Rabin Karp. (Study)**

- The Rabin–Karp algorithm or Karp–Rabin algorithm is a string searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find any one of a set of pattern strings in a text.

- For text of length n and p patterns of combined length m, its average and best case running time is O(n + m) in space O(p), but its worst-case time is O(nm).

Example:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 | 3 | 5 |

Text

|  |  |
| --- | --- |
| 2 | 6 |

Pattern

- 26 mod 11 = 4

- Spurious hit : 15, 59, 92

- Valid hit : 26

Algorithm:

Rabin\_Karp (string s [1...n], string pattern [1...m])

step1: hpattern: = hash (pattern [1...m]); hs: = hash (s [1...m])

step2: for i from 1 to n-m+1

if hs = hpattern

if s [i... i+m-1] = pattern [1...m]

return i

hs: = hash (s [i+1...i+m])

step3: return not found

Code:

#include<stdio.h>

#include<conio.h>

#include<string.h>

#include<math.h>

#define d 10

void RabinKarpStringMatch(char [], char [], int);

void main(){

char Text[20],Pattern[20];

int Number = 11; //Prime Number

printf("\nEnter Text String : ");

gets(Text);

printf("\nEnter Pattern String : ");

gets(Pattern);

RabinKarpStringMatch(Text,Pattern,Number);

getch();

}

void RabinKarpStringMatch(char Text[], char Pattern[], int Number){

int M,N,h,P=0,T=0, TempT, TempP;

int i,j;

M = strlen(Pattern);

N = strlen(Text);

h = (int)pow(d,M-1) % Number;

for(i=0;i<M;i++) {

P = ((d\*P) + ((int)Pattern[i])) % Number;

TempT = ((d\*T) + ((int)Text[i]));

T = TempT % Number;

}

for(i=0;i<=N-M;i++) {

if(P==T) {

for(j=0;j<M;j++)

if(Text[i+j] != Pattern[j])

break;

if(j == M)

printf("\nPattern Found at Position : %d",i);

}

TempT =((d\*(T - Text[i]\*h)) + ((int)Text[i+M]));

T = TempT % Number;

if(T<0)

T=T+Number;

}

}

Output:

