**Assignment No. 1**

**Aim :** Write a program non-recursive and recursive program to calculate Fibonacci numbers and

analyze their time and space complexity.

Theory :

The **Fibonacci sequence,**also known as Fibonacci numbers, is defined as the sequence of numbers in which each number in the sequence is equal to the sum of two numbers before it. The Fibonacci Sequence is given as:

**Fibonacci Sequence = 0, 1, 1, 2, 3, 5, 8, 13, 21, ….**

Here, the third term “1” is obtained by adding the first and second term. (i.e., 0+1 = 1)

Similarly,

“2” is obtained by adding the second and third term (1+1 = 2)

“3” is obtained by adding the third and fourth term (1+2) and so on.

For example, the next term after 21 can be found by adding 13 and 21. Therefore, the next term in the sequence is 34.

The Fibonacci sequence of numbers “Fn” is defined using the recursive relation with the seed values F0=0 and F1=1:

**Fn = Fn-1+Fn-2**

Here, the sequence is defined using two different parts, such as kick-off and recursive relation.

The kick-off part is F0=0 and F1=1.

The recursive relation part is Fn = Fn-1+Fn-2.

The sequence starts with 0 rather than 1. So, F5should be the 6th term of the sequence.

## Iterative Approach:

## Time Complexity:

The time complexity of the iterative code is linear, as the loop runs from 2 to n, i.e. it runs in O(n) time

Space Complexity :

For the iterative approach, the amount of space required is the same for fib(6) and fib(100), i.e. as N changes the space/memory used remains the same. Hence it’s space complexity is O(1) or constant.

Recursive approach:

Time Complexity :

T(n) = T(n-1) + T(n-2) + c  
 = 2T(n-1) + c //from the approximation T(n-1) ~ T(n-2)  
 = 2\*(2T(n-2) + c) + c  
 = 4T(n-2) + 3c  
 = 8T(n-3) + 7c  
 = 2^k \* T(n - k) + (2^k - 1)\*c

n - k = 0 k = n

T(n) = 2^n \* T(0) + (2^n - 1)\*c  
 = 2^n \* (1 + c) – c

T(n) = 2^n

Space Complexity :

For Fibonacci recursive implementation or any recursive algorithm, the space required is proportional to the maximum depth of the recursion tree, because, that is the maximum number of elements that can be present in the implicit function call stack.

**Input** :

No. of terms

**Output:**

Fibonacci series upto nth terms

**Program Implementation :** Code is written in C++ to implement program to implement Fibonacci series using iterative and recursive approach.

**Conclusion :** Thus , Fibonacci series using iterative and recursive approach is implemented using C++.

#include <iostream>

**using** **namespace** std;

**int** main() {

**int** n1=0,n2=1,n3,i,number;

 cout<<"Enter the number of elements: ";

 cin>>number;

 cout<<n1<<" "<<n2<<" "; //printing 0 and 1

**for**(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printed

 {

  n3=n1+n2;

  cout<<n3<<" ";

  n1=n2;

  n2=n3;

 }

**return** 0;

   }

#include <iostream>

using namespace std;

int fib(int x) {

   if((x==1)||(x==0)) {

      return(x);

   }else {

      return(fib(x-1)+fib(x-2));

   }

}

int main() {

   int x , i=0;

   cout << "Enter the number of terms of series : ";

   cin >> x;

   cout << "Fibonnaci Series : ";

   while(i < x) {

      cout << " " << fib(i);

      i++;

   }

}

**Assignment No.– 2**

**Aim :** Write a program to implement Huffman Encoding using a greedy strategy.

**Theory:**

Greedy Method:

In greedy method following activities are performed:

1. First we select some solution from input domain.

2. Then we check whether solution is feasible or not.

3. From set of feasible solution, particular solution that satisfies or nearly satisfies or nearly

satisfies objective of function such solution is called optimal solution.

4. As this method work in stages at each stage only one input is considered at each time. Based

on this input it is decided whether particular input gives the optimal solution or not.

**Characteristics of greedy method:**

1) Greedy choice property:

We solve sub problems & which ever choice. We find then solve subproblem arising after choice is made. This choice may depend upon previously made choices but it does not depend upon any future choice. Thus, here greedy choices are made one after another, reducing each problem into smaller one.

2) Optimal substructure:

A problem shows optimal substructure if an optimal solution to problem contains optimal solution to subproblem.

**Huffman tree:**

In Huffman coding method, data is inputted as a sequence of characters. Then a table of frequency of occurrence of each character in data is built.

- From table of frequencies Huffman tree is constructed.

- The Huffman’s tree is further used for encoding each character so that binary encoding

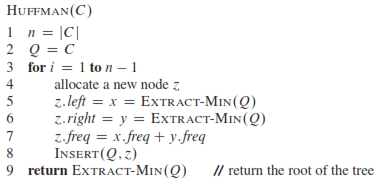
is obtained for the given data.

- In Huffman’s tree is further specific method of representing each symbol. This method

produces a code. In such a manner that no code coded in prefix of some other code

word such code word are called as prefix codes.

Algorithm :



**Input** :

No. of characters with frequencies.

**Output:**

Encoding of characters

**Program Implementation :** Code is written in C++ to implement program to implement Huffman coding using Greedy approach.

**Conclusion :** Thus , Huffman coding using Greedy approach is implemented using C++.

#include <bits/stdc++.h>

#include<iostream>

#include<string>

using namespace std;

// stucture of tree node

typedef struct treenode

{

float freq;

char data;

struct treenode \*left,\*right;

}treenode;

/\* structure of node of linked list \*/

typedef struct node

{

treenode \*data; //address of tree

struct node \*next;

}node;

node \*insert(node \*,treenode \*);

treenode \*create();

void encode();

void decode(treenode \*);

int n=0;

char alphabets[30];

char codes[30][10];

void preorder(treenode \*p,int i ,char word[])

{

if(p!=NULL)

{

if(p->left==NULL && p->right==NULL)

{

word[i]=0;

printf("\n%c --- %s",p->data,word);

alphabets[n]=p->data;

strcpy(codes[n++],word);

}

word[i]='0';

preorder(p->left,i+1,word);

word[i]='1';

preorder(p->right,i+1,word);

}

}

int main()

{

int op;

char word[10];

treenode \*root=NULL;

do

{

cout << "\n1)Create Huffman Tree";

cout << "\n2)Encode a Message ";

cout << "\n3)Decode a message ";

cout << "\nEnter Your Choice : ";

cin>>op;

switch(op)

{

case 1:

root=create();

cout << "\nPrefix codes : \n";

preorder(root,0,word); // create the encoding sequence

break;

case 2: encode(); break;

case 3: decode(root);break;

}

}while(op!=4);

}

treenode \*create()

{

treenode \*p,\*t1,\*t2;

node \*head;

int n,i;

char x;

float probability;

head=NULL; //empty linked list

cout <<"\nEnter No. of alphabets :";

cin >> n;

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

{

//flushall();

cout << "\n Enter alphabet :";

cin >>x;

cout << "\nEnter frequency :";

cin >> probability;

/\* create a new tree and insert it in the priority linked list \*/

p=(treenode\*)malloc(sizeof(treenode));

p->left=p->right=NULL;

p->data=x;

p->freq=probability;

head=insert(head,p);

}

/\* create the final tree by merging of two trees of smaller weights (n-1) merges will be required\*/

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

{

t1=head->data; //first tree

t2=head->next->data; //second tree

head=head->next->next; /\*remove first 2 trees from linked list\*/

/\*merge t1 and t2 with new tree in P\*/

p=(treenode \*)malloc(sizeof(treenode));

p->left=t1;

p->right=t2;

p->freq=t1->freq+t2->freq;

head=insert(head,p); /\*insert the new tree in the linked list\*/

}

return(head->data);

// preorder(head->data);

//getch();

}

node \*insert(node \*head,treenode \*t)

{

node \*p,\*q;

p=(node \*)malloc(sizeof(node));

p->data=t;

p->next=NULL;

if(head==NULL) //empty linked list

return(p);

if(t->freq<head->data->freq)

{

p->next=head;

return(p);

}

// locate the point of insertion

q=head;

while(q->next!=NULL && t->freq>q->next->data->freq)

q=q->next;

p->next=q->next;

q->next=p;

return(head);

}

void encode()

{

char word[30];int i,j;

//flushall();

cout << "\n Enter a Message : ";

cin>>word;

cout << "\n Encoded Message \n";

for(i=0;word[i]!='\0';i++)

{

for(j=0;alphabets[j]!=word[i] && j<n;j++);

if(j<n)

cout<<codes[j];

}

}

void decode(treenode \*p)

{

char word[90];int i;treenode \*q;

//shall();

cout<<"\nEnter an Encoded message : ";

cin >>word;

q=p;i=0;

cout << "\nDecoded Message = ";

while(word[i]!='\0')

{

if(word[i]=='0')

q=q->left;

else

q=q->right;

if(q->left==NULL && q->right==NULL)

{

printf("%c",q->data);

q=p;

}

i++;

}

}

**Assignment No. 3**

Aim : Write a program to solve a fractional Knapsack problem using a greedy method.

**Theory:**

Greedy Method:

In greedy method following activities are performed:

1. First we select some solution from input domain.

2. Then we check whether solution is feasible or not.

3. From set of feasible solution, particular solution that satisfies or nearly satisfies or nearly

satisfies objective of function such solution is called optimal solution.

4. As this method work in stages at each stage only one input is considered at each time. Based

on this input it is decided whether particular input gives the optimal solution or not.

**Characteristics of greedy method:**

1) Greedy choice property:

We solve sub problems & which ever choice. We find then solve subproblem arising after choice is made. This choice may depend upon previously made choices but it does not depend upon any future choice. Thus, here greedy choices are made one after another, reducing each problem into smaller one.

2) Optimal substructure:

A problem shows optimal substructure if an optimal solution to problem contains optimal solution to subproblem.

## **Fractional Knapsack**

In this case, items can be broken into smaller pieces.

According to the problem statement,

* There are **n** items in the store
* Weight of **ith** item wi>0
* Profit for **ith** item pi>0 and
* Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction ***xi*** of **ith** item.

0⩽xi⩽1

The **ith** item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.

Hence, the objective of this algorithm is to

maximize∑n=1n(xi.pi)maximize∑n=1n(xi.pi)

subject to constraint,

∑n=1n(xi.wi)⩽W

Algorithm :

**Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)**

for i = 1 to n

do x[i] = 0

weight = 0

for i = 1 to n

if weight + w[i] ≤ W then

x[i] = 1

weight = weight + w[i]

else

x[i] = (W - weight) / w[i]

weight = W

break

return x

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

**Input** :

No. of Objects, Capacity of sack, weights and profits of all objects.

**Output:**

Maximum profit value

**Program Implementation :** Code is written in C++ to implement program to implement fractional knapsack problem using Greedy approach.

**Conclusion :** Thus , fractional knapsack problem using Greedy approach is implemented using C++.

#include <bits/stdc++.h>

#include <iostream>

using namespace std;

struct Item {

int value;

int weight;

};

class Solution {

public:

bool static comp(Item a, Item b) {

double r1 = (double) a.value / (double) a.weight;

double r2 = (double) b.value / (double) b.weight;

return r1 > r2;

}

// function to return fractionalweights

double fractionalKnapsack(int W, Item arr[], int n) {

sort(arr, arr + n, comp);

int curWeight = 0;

double finalvalue = 0.0;

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

if (curWeight + arr[i].weight <= W) {

curWeight += arr[i].weight;

finalvalue += arr[i].value;

cout << 1 <<"\t";

} else {

int remain = W - curWeight;

finalvalue += (arr[i].value / (double) arr[i].weight) \* (double) remain;

cout<<((double)remain/arr[i].weight);

break;

}

}

return finalvalue;

}

};

int main() {

int n = 3, weight = 50;

Item arr[n] = { {100,20},{60,10},{120,30} };

Solution obj;

double ans = obj.fractionalKnapsack(weight, arr, n);

cout << "\n The maximum value is " << setprecision(2) << fixed << ans;

return 0;

}

-----

**Assignment No. 4**

Aim : Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Theory :

The dynamic programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of caching subproblem solutions and appealing to the "principle of optimality."

There are two basic elements that characterize a dynamic programming algorithm:

**1. Optimal Substructure  :**

A problem is said to have optimal substructure if the globally optimal solution can be constructed from locally optimal solutions to subproblems.

**2. Overlapping Subproblems :**

A problem is said to have overlapping subproblems if it can be broken down into subproblems which are reused multiple times. This is closely related to recursion.

**The Principle of Optimality**

The dynamic programming  relies on a principle of optimality. This principle states that in an optimal sequence of decisions or choices, each subsequence must also be optimal.

Steps of dynamic programming :

1) Characterize the structure of an optimal solution.

2) Recursively defined the value of an optimal solution.

3) Compute the value of an optimal solution in a bottom-up fashion.

4) Construct an optimal solution from computed information.

In case of 0-1 Knapsack, the value of ***xi*** can be either ***0*** or ***1***, where other constraints remain the same. 0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

**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

if wi ≤ w then

if vi + c[i-1, w-wi] then

c[i, w] = vi + c[i-1, w-wi]

else c[i, w] = c[i-1, w]

else

c[i, w] = c[i-1, w]

**Input** :

No. of Objects, Capacity of sack, weights and profits of all objects.

**Output:**

Maximum profit value

**Program Implementation :** Code is written in C++ to implement program to implement fractional knapsack problem using dynamic programming.

**Conclusion :** Thus , fractional knapsack problem using dynamic programming is implemented using C++.

#include <iostream>

using namespace std;

int max(int x, int y) {

return (x > y) ? x : y;

}

int knapSack(int W, int w[], int v[], int n) {

int i, wt;

int K[n + 1][W + 1];

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

for (wt = 0; wt <= W; wt++) {

if (i == 0 || wt == 0)

K[i][wt] = 0;

else if (w[i - 1] <= wt)

K[i][wt] = max(v[i - 1] + K[i - 1][wt - w[i - 1]], K[i - 1][wt]);

else

K[i][wt] = K[i - 1][wt];

}

}

return K[n][W];

}

int main() {

cout << "Enter the number of items in a Knapsack:";

int n, W;

cin >> n;

int v[n], w[n];

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

cout << "Enter value and weight for item " << i << ":";

cin >> v[i];

cin >> w[i];

}

cout << "Enter the capacity of knapsack";

cin >> W;

cout << knapSack(W, w, v, n);

return 0;

}

**Assignment -5**

Aim: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining

Queens to generate the final n-queen‘s matrix.

**Theory** :-

**Backtracking**

In backtracking method :-

* The desired solution is expressible as an n tuple(x1,x2,x3,..,xn) which is chosen from finite set Si.
* The solution maximizes or minimizes or satisfies a certain function C(x1,x2,x3...xn).
* The problem can be categorized into 3 categories :-
* Optimization problem.
* Decision problem.
* Enumeration problem.
* The basic idea of backtracking is to build up a vector, one component at a time.
* The major advantage of backtracking algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be generated.
* Backtracking algorithm determines the solution by systematically learning the solution for the given problem.
* Backtracking is the DFS (Depth First Search) with lname bounding function.
* Backtracking algorithm solves the problem using 2 types of constraints :-
* Explicit constraint.
* Implicit constraint.
* Explicit constraints are rules which restrict each vector element to be chosen from given set.
* Implicit constraints are rules which determine which of tuple in solution space satisfy the criteria function.

**Algorithm for n-queens :-**

1. Algorithm Queen(n)
2. //Input : n-queens
3. // output : Solution Vector
4. for column <-- 1 to n do
5. {
6. if(place(row, column)) then
7. {
8. board [row]=column;
9. if(row=n) then
10. print\_board(n);
11. else
12. Queen(row t1,n)
13. }
14. }

**Algorithm to place the queen :-**

1. Algorithm place(row, column)
2. // Input : row, column
3. //Output : boolean
4. for i <-- 1 to row-1 do
5. {
6. if (board[i] = column) then
7. return 0;
8. else
9. if (abs(board[i]-column) = abs(i-row)) then
10. return 0;
11. }
12. //no conflicts hence Queen can be placed.
13. return 1;
14. }

**Input** :

No. of queens.

**Output:**

Solution vector showing the column position of all the queens.

**Program Implementation :** Code is written in C++ to implement program to find the solution of placing n queens on chessboard using backtracking.

**Conclusion :** Thus , N-Queens algorithm using backtracking is implemented using C++.

#include <iostream>

using namespace std;

//#include<stdio.h>

//#include<math.h>

int board[20];

//int count=0;

int main()

{

int n,i,j;

void Queen(int row,int n);

cout<<"\n\t Program for n-queen using Backtracking";

cout<<"\n Enter the no of queens";

cin >> n;

Queen(1,n);

return 0;

}

void print\_board(int n)

{

int i,j;

int count=0;

cout <<"\n Solution :\n";

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

{cout<<"\t"<<i;

}

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

{

cout<< "\n \n"<<i; //printf("\n\n%d",i);

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

{

if(board[i]==j)

cout <<"\t"<< "Q" ;

else

cout << "\t" << "-";

}

}cout<<"\n Press any key to continue...";

}

int place(int row,int column)

{

int i;

for(i=1;i<=row-1;i++)

{

if(board[i] == column)

return 0;

else

if(abs(board[i]-column)==abs(i-row))

return 0;

}

return 1;

}

void Queen(int row,int n)

{

int column;

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

{

if(place(row,column))

{

board[row]=column;

if(row == n)

print\_board(n);

else

Queen(row+1,n);

}

}

}