**Modern Education Society’s**

**Wadia College of Engineering, Pune**

|  |
| --- |
| **NAME OF STUDENT: CLASS:** |
| **SEMESTER/YEAR: ROLL NO:** |
| **DATE OF PERFORMANCE: DATE OF SUBMISSION:** |
| **EXAMINED BY: EXPERIMENT NO: LP-III(DAA)-01** |

###### **TITLE: Fibonacci Numbers**

**AIM:** Write a non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity.

**OBJECTIVES:**

1. To find Fibonacci Number
2. Analyse time and space complexity

**PER-REQUISITES:**

1. Knowledge of programming language.

2. Knowledge of Time and space complexity

THEORY:

The Fibonacci numbers or Fibonacci series are the numbers in the following integer sequence: 0,1,1,2,3,5,8,13,21,.... .

By definition, the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two. In mathematical terms, the sequence F n of Fibonacci numbers is defined by the recurrence relation Fn= Fn-1 + Fn-2

ALGORITHM:-

1. Read n, the number of terms to be generated
2. Term 1 = 0, term2 = 1.
3. Display term1, term2.
4. Vary the counter in a loop from 0 to (n – 1) incremented by 1.
5. Term = term1 + term2
6. Display term.
7. Term 1 =term2.
8. Term 2 = term →
9. End loop

CONCLUSION:-Implemented Fibonacci series successfully.

**Questions for Review:**

1. What is Fibonacci series?
2. What is recursion?
3. Analyze time and space complexity of fibonacci series?

**CODE:**

arr = []

*# Recursive*

def **fibo\_rec**(n):

    if n <= 1:

        arr[n] = n

        return n

**next** = fibo\_rec(n - 1) + fibo\_rec(n - 2)

    arr[n] = **next**

    return **next**

*# non recursive*

def **fibo\_itr**(n):

    if n == 0:

**print**(0,end=" ")

        return 0

    a,b = 0,1

**print**(a,b,end=" ",sep=" ")

    for i in **range**(1, n):

        a,b = b,a + b

**print**(b,end=" ")

    return b

n = int(**input**("Enter Number Of Elements:"))

arr = [0 for i in **range**(n+1)]

**print**("Itretative")

fibo\_itr(n)

**print**()

**print**("Recursive")

fibo\_rec(n)

**print**(arr)

**print**("End")

"""

Analysis

Recursive:

    Time Complexity - O(2^n)

    Space Complexity - O(n)

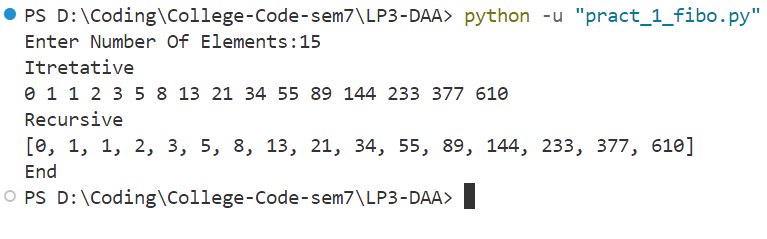
Iterative:

    Time Complexity - O(n)

    Space Complexity - O(1)

"""

**OUTPUT:**



**Modern Education Society’s**

**Wadia College of Engineering, Pune**

|  |
| --- |
| **NAME OF STUDENT: CLASS:** |
| **SEMESTER/YEAR: ROLL NO:** |
| **DATE OF PERFORMANCE: DATE OF SUBMISSION:** |
| **EXAMINED BY: EXPERIMENT NO: LP-III(DAA)-02** |

###### **TITLE:** Huffman coding algorithm

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

**OBJECTIVES:**

1. Analyze performance of an algorithm.
2. Learn how to implement algorithms that follow greedy strategy.

**PER-REQUISITES:**

1. Knowledge of programming language.

2. Knowledge of Time and space complexity

THEORY:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

ALGORITHM:-

*Steps to build Huffman Tree*

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root).
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat step 2 and 3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

CONCLUSION:-Implemented Huffman Encoding using a greedy strategy successfully.

**Questions for Review:**

1. Explain greedy approach?

2. Explain huffman coding with example using greedy approach?

3. Analyze time complexity of huffman coding?

**CODE:**

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

class Node{

    public:

    bool is\_leaf;

    int freq\_sum;

    char c;

    Node\* left;

    Node\* right;

    Node(char c,int f){

        this->c = c;

        this->freq\_sum = f;

        this->left = NULL;

        this->right = NULL;

        this->is\_leaf = true;

    }

    Node(int f,Node\* l=NULL,Node\* r=NULL){

        this->is\_leaf = false;

        this->freq\_sum = f;

        this->left = l;

        this->right = r;

    }

};

void build\_code\_table(Node\* root,string code,map<char,string>& code\_table){

    if(root == NULL) return;

    if(root->is\_leaf){

        code\_table[root->c] = code;

        return;

    }

    build\_code\_table(root->left,code+'0',code\_table);

    build\_code\_table(root->right,code+'1',code\_table);

}

int main(){

    // input string

    string s;

    cout<<"Enter string To Encode:";

    cin>>s;

    // build frequency table

    map<char,int> freq;

    for(char c:s) freq[c]++;

    cout<<"Frequency Of Chars:"<<endl;

    for(auto p:freq){

        cout<<p.first<<" - "<<p.second<<endl;

    }

    // build huffman tree

    vector<Node\*> nodes;

    for(auto p:freq){

        nodes.push\_back(new Node(p.first,p.second));

    }

    sort(nodes.begin(),nodes.end(),[](Node\* a,Node\* b){return a->freq\_sum < b->freq\_sum;});

    while(nodes.size()>1){

        int sum = nodes[0]->freq\_sum + nodes[1]->freq\_sum;

        Node\* parent = new Node(sum,nodes[0],nodes[1]);

        nodes.erase(nodes.begin(),nodes.begin()+2);

        nodes.push\_back(parent);

        sort(nodes.begin(),nodes.end(),[](Node\* a,Node\* b){return a->freq\_sum < b->freq\_sum;});

    }

    Node\* root = nodes[0];

    // build code table

    map<char,string> codes;

    string code\_str = "";

    build\_code\_table(root,code\_str,codes);

    // print code table

    cout<<"Code Table:"<<endl;

    for(auto p:codes){

        cout<<p.first<<" - "<<p.second<<endl;

    }

    // encode string

    string encoded\_str = "";

    for(char c:s){

        encoded\_str += codes[c];

    }

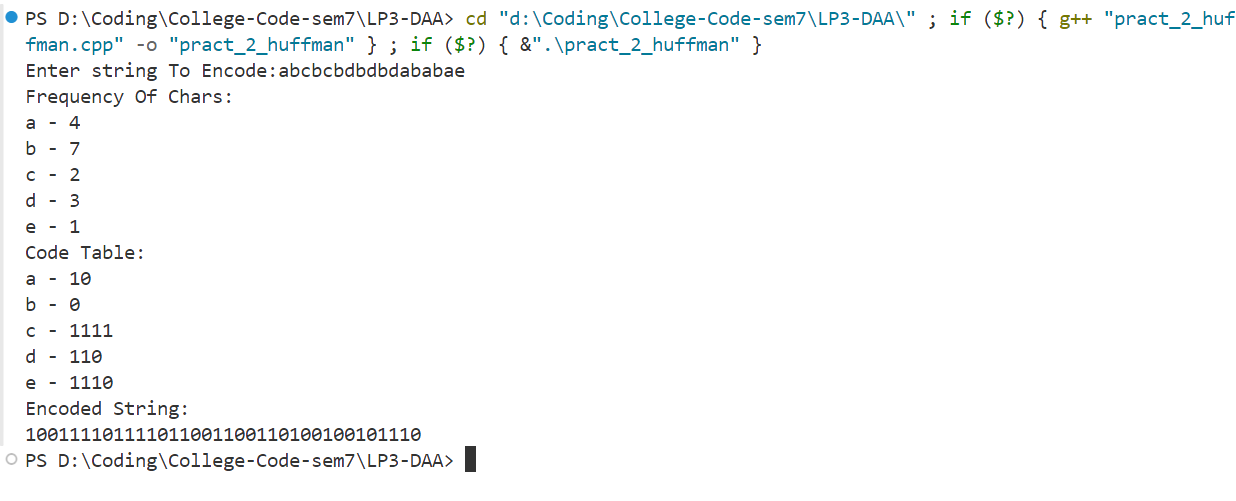
    cout<<"Encoded String:"<<endl;

    cout<<encoded\_str<<endl;

    return 0;

}

**OUTPUT:**

****

**Modern Education Society’s**

**Wadia College of Engineering, Pune**

|  |
| --- |
| **NAME OF STUDENT: CLASS:** |
| **SEMESTER/YEAR: ROLL NO:** |
| **DATE OF PERFORMANCE: DATE OF SUBMISSION:** |
| **EXAMINED BY: EXPERIMENT NO: LP-III(DAA)-03** |

###### **TITLE:** Knapsack Problem

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

**OBJECTIVES:**

Learn how to implement algorithms that follow greedy strategy.

Analyze performance of an algorithm

**PER-REQUISITES:**

1. Knowledge of Fractional Knapsack problem using a greedy method.

THEORY:

Given the weights and values of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack

Note: In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don’t take it.

## **Greedy approach for fractional knapsack problem:**

An efficient solution is to use the Greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on the basis of this ratio. Then take the item with the highest ratio and add them until we can’t add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

ALGORITHM:-

Steps to solve the problem using the above approach:

1) Calculate the ratio(value/weight) for each item.

2) Sort all the items in decreasing order of the ratio.

3) Initialize res =0, curr\_cap = given\_cap.

4) Do the following for every item “i” in the sorted order:

* + If the weight of the current item is less than or equal to the remaining capacity then add the value of that item into the result
  + else add the current item as much as we can and break out of the loop

5)Return res

CONCLUSION:-Implemented fractional knapsack using a greedy strategy successfully.

**Questions for Review:**

1) Write realistic applications of this experiment in brief (at least two applications).?

2) Explain Kanpsack with example using greedy approach?

|  |  |  |
| --- | --- | --- |
| Item | Weight | Value |
| 1 | 5 | 30 |
| 2 | 10 | 40 |
| 3 | 15 | 45 |
| 4 | 22 | 77 |
| 5 | 25 | 90 |

3)Analyse time complexity of knapsack ?

**CODE:**

#include <iostream>

#include <bits/stdc++.h>

using namespace std;

struct weights{

    int weight;

    int profit;

    double pw\_ratio;

    weights(int w,int p){

        weight = w;

        profit = p;

        pw\_ratio = (double)p/(double)w;

    }

};

void print(vector<weights> a){

    cout<<"-----------"<<endl;

    cout<<" W "<<" P "<<"P/W"<<endl;

    for(int i=0;i<a.size();i++){

        cout<<a[i].weight<<" "<<a[i].profit<<" "<<a[i].pw\_ratio<<endl;

    }

    cout<<"-----------"<<endl;

}

bool cmp(weights a,weights b){

    return a.pw\_ratio > b.pw\_ratio;

}

int f\_knapsack(vector<weights>arr,int capacity){

    int rem = capacity;

    int profit = 0;

    for(weights w:arr){

        cout<<"Selected -> Weight:"<<w.weight<<", Profit:"<<w.profit<< ", P/W Ratio:"<<w.pw\_ratio<<endl;

        if(w.weight <= rem){

            rem-= w.weight;

            profit += w.profit;

        }else{

            // factional

            int pf = w.pw\_ratio \* rem;

            profit += pf;

            rem = 0;

            return profit;

        }

        cout<<"Capacity Remained:"<<rem<<", Profit:"<<profit<<endl;

    }

    return profit;

}

int main(){

    int n;

    cout<<"Enter Number of Weights:";

    cin>>n;

    vector<weights> arr;

    for(int i=0;i<n;i++){

        int a,b;

        cout<<"Enter Weight "<<i+1<<":";

        cin>>a;

        cout<<"Enter Profit "<<i+1<<":";

        cin>>b;

        weights w(a,b);

        arr.push\_back(w);

    }

    cout<<endl;

    sort(arr.begin(),arr.end(),cmp);

    print(arr);

    int capacity;

    cout<<"Enter Capacity:";

    cin>>capacity;

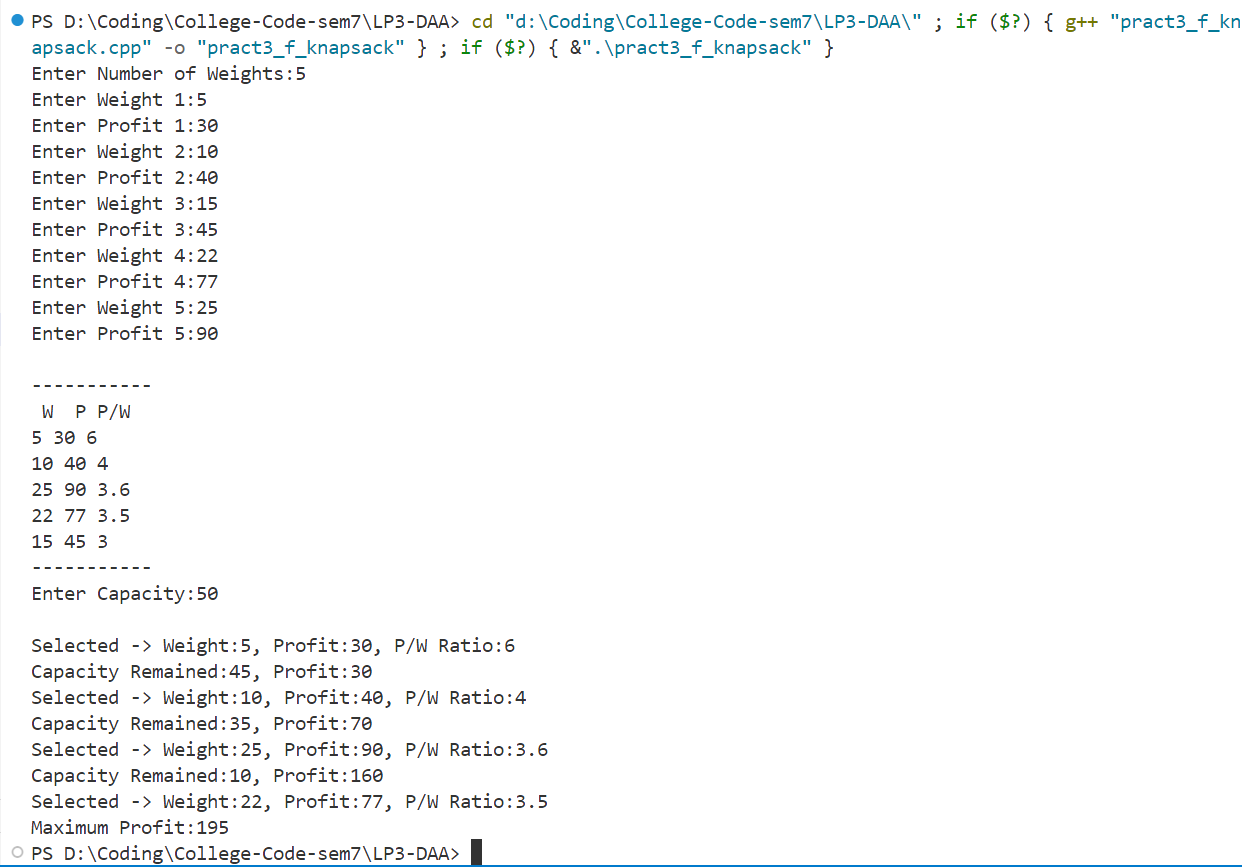
    cout<<endl;

    cout<<"Maximum Profit:"<<f\_knapsack(arr,capacity)<<endl;

    return 0;

}

**OUTPUT:**



`

**Modern Education Society’s**

**Wadia College of Engineering, Pune**

|  |
| --- |
| **NAME OF STUDENT: CLASS:** |
| **SEMESTER/YEAR: ROLL NO:** |
| **DATE OF PERFORMANCE: DATE OF SUBMISSION:** |
| **EXAMINED BY: EXPERIMENT NO: LP-III(DAA)-04** |

###### TITLE: 0/1 Knapsack

AIM: - 0/1 Knapsack Problem using Dynamic Programming.

**OBJECTIVES:**

* To study dynamic programming.
* To implement 0/1 knapsack problem.
* Calculate time complexity of algorithm.

**THEORY:**

In the Dynamic programming we will work considering the cases as:-

Case 1: The item is included in the optimal subset.

Case 2: The item is not included in the optimal set.

In a DP[][] table let’s consider all the possible weights from ‘1’ to ‘W’ as the columns and weights that can be kept as the rows. The state DP[i][j] will denote maximum value of ‘j-weight’ considering all values from ‘1 to ith’. So if we consider ‘wi’ (weight in ‘ith’ row) we can fill it in all columns which have ‘weight values > wi’. Now two possibilities can take place:

Fill ‘wi’ in the given column.

Do not fill ‘wi’ in the given column.

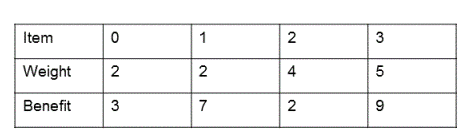
Now we have to take a maximum of these two possibilities, formally if we do not fill ‘ith’ weight in ‘jth’ column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of ‘wi’+ value of the column weighing ‘j-wi’ in the previous row. So we take the maximum of these two possibilities to fill the current state.

**CONCLUSION:** Implemented 0/1/ knapsack using dynamic programming successfully.

**Questions for Review:**

1.Write time & space complexity of 0/1 knapsack algorithm using dynamic programming.

2.Write realistic applications of this experiment in brief (at least two applications).

3. The weight limit for this knapsack is 10 find solution using dynamic programming? 

**Modern Education Society’s**

**Wadi College of Engineering, Pune**

|  |
| --- |
| **NAME OF STUDENT: CLASS:** |
| **SEMESTER/YEAR: ROLL NO:** |
| **DATE OF PERFORMANCE: DATE OF SUBMISSION:** |
| **EXAMINED BY: EXPERIMENT NO: LP-III(DAA)-05** |

### TITLE: N Queen’s Problem using Backtracking.

**AIM:** Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen‘s matrix

**OBJECTIVES:**

* To understand the concept of Backtracking.
* To understand the concept of State Space Tree.
* To compare the space & time complexity of Recursive & Non-Recursive techniques of Backtracking.

**PRE-REQUISITES:** Backtracking formulation is used to solve problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints.

Definition of state space tree:-

The tree organization of the solution space is referred to as state space tree.

If the tree organization is independent of the problem instance being solved, they are called static trees.

If the tree organization is dependent of the problem instance being solved, they are called dynamic trees.

**THEORY:**

**What is Backtracking?**

Backtracking formulation is used to solve problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints.

Constraint satisfaction problems:-

1. These are problems with complete solution, where the order of elements does not matter.

2. The problem consists of set of variables each of which must be assigned a value, subject to particular constraints of problem.

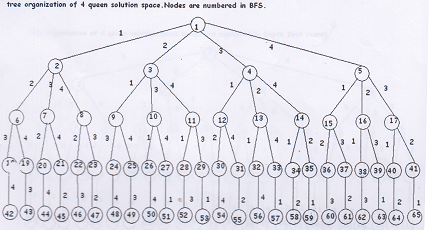
3. Backtracking attempts to try all the combinations in order to obtain a solution.

4. Its strength is that many implementations avoid trying many partial combinations, thus speeding up the running time.

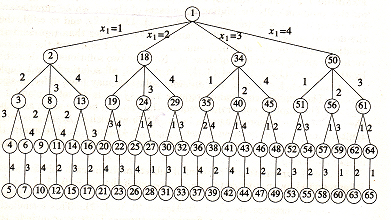
**State space trees:-**Backtracking algorithms determined problems solution by systematically searching the solution space for the given problem instance.This search is facilitated by using a tree organization for the solution space. For a given problem space many tree organizations are possible.

Consider a 4 \* 4 chessboard. We have to place 4 queens such that no two queens are on the same row, column or diagonal. For the solution to this problem ,we place each of the 4 queens on a separate row. Now we have to place each queen on a unique column too such that no 2 queens are on same diagonals. Various permutations of queens position are possible but only the permutations that satisfy the constraints are valid. The Various permutations of queens positions can be depicted by a tree organization .Let the level of the tree denote the row and edge denote the column. Nodes denote the states reached. Definition of state space tree:-

The tree organization of the solution space is referred to as state space tree. If the tree organizations are independent of the problem instance being solved, they are called static trees. If the tree organization is dependent of the problem instance being solved, they are called dynamic trees.

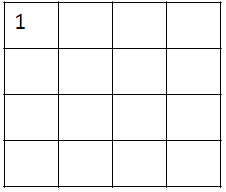


**tree organization of 4 queen space. Nodes are numbered in DFS**

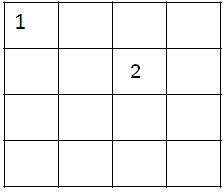


Implementation of backtracking algorithm for 4 queen’s problem:-

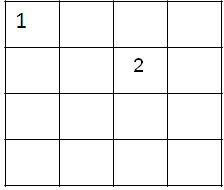
Step1: We place the first queen in row 1, column 1 as shown.



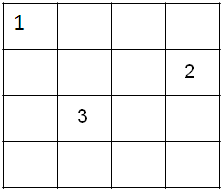
Step 2: Now we have to place queen 2 in row 2.We choose column 3 because no queens should be in same column or diagonal.



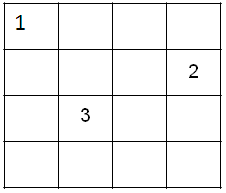
Step 3: Now we have to place queen 3 in row 3.We can’t choose column 2, 1 or 3. So we backtrack and shift the queen to column 4. Then we place queen 3 in row 3, column 2 as shown.



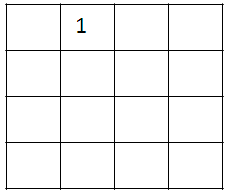
Step 4: now we have to place queen 4 in row 4. We cannot place queen 4 in column 1, 2, 3 or 4 since no queen should be in same column or diagonal.



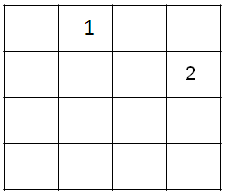
Step 5: Thus we backtrack. We cannot shift queen 3 and still satisfy implicit constraints. We cannot move queen 2 and still satisfy constraints. Thus we shift queen 1 to column 2, row 1.



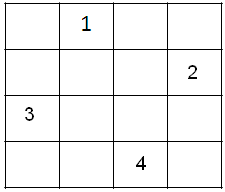
Step 6: Now we place queen 2 in column 4, row 2.



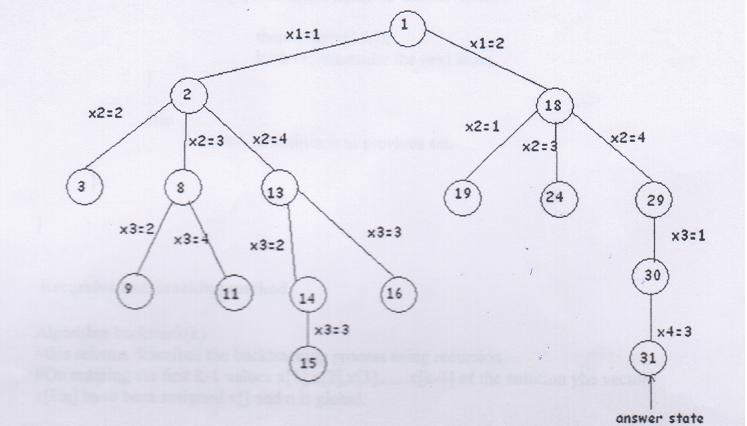
Step 7: Now we place queen 3 in column 1, row 3.



Step 8: At last, we place queen in column 3, row 4.



The tree generated during the backtracking is as follows. The edges denote the column. The level(1) denotes the row of the queen(1).



**ALGORITHM:-**

**Iterative backtracking Algorithm:**

Algorithm Backtrack (n)

//This schema describes the backtracking process.

// all solutions are generated in x[1:n] and printed as soon as they are determined.

{

k:= 1;

While(k!=0)do

{

if(there remains an untried x[k] belongs t(x[1],x[2],……, x[k- 1]) and

Bk(x[1],…...x[k] )is true) then

{

if(x[1],….x[k] is a path to answer node)

then write(x[1:k]);

k:=k.+1; //consider the next state.

}

else

k:=k-1; //backtrack to previous set.

}

}

**Recursive backtracking method.**

Algorithm backtrack(k)

//this schema describes the backtracking process using recursion.

//On entering the first k-1 values x[1],x[2],x[3],......x[k-1] of the solution the vector x[1:n] have been assigned x[] and n is global.

{For (each x[k]belongs t(x[1],x[k-1]))

do{if Bk(x [1], x[2],....x[k]!=0)then{

IfBk (x[1],x[2],……x[k] is a path to answer node)

then wrie (x[1:k]);

if(k<n) then backtrack(k+1);

}

}

}

**CONCLUSION:** Implemented 8 queen successfully.

**Questions for Review:**

* 1. What is backtracking? What are the peculiar characteristics & applications of this approach?
  2. Explain Explicit & Implicit constraints with respect to 8 queen’s problem?

3) Compare the space & time complexity of Recursive & Non-Recursive techniques of Backtracking.

1. Write realistic applications of this experiment in brief (at least two applications).