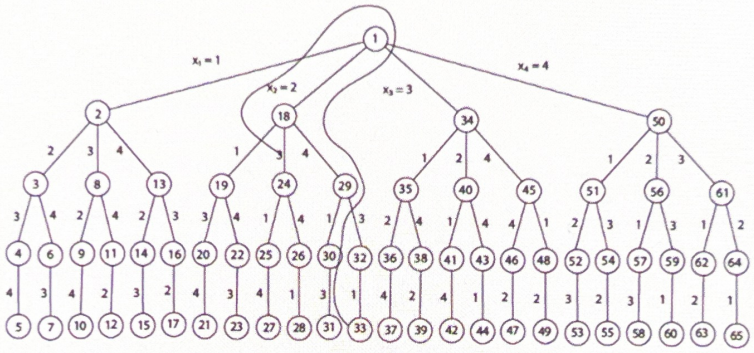
Few Pending

DAA MODULE 4

PART A

1) Build the state space tree generated by the 4 queen's problem.

Answer is (2,4,1,3) or its mirror image (3,1,4,2)



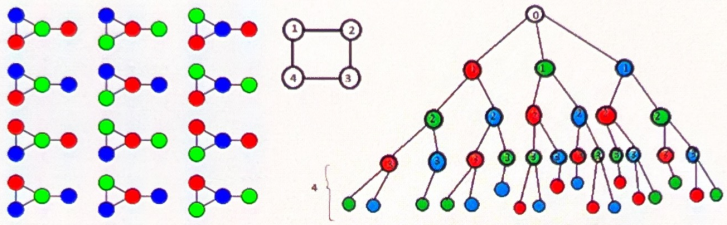
2) Apply the backtracking algorithm to solve the following instance of the sum of subsets problem S=5,10,12,13,15,18 and d=30

```
def sum_of_subset(s,k,rem):
    x[k]=1
    if s+l[k]==target_sum:
        list1=[]
        for i in range (0,k+1):
            if x[i]==1:
                list1.append(l[i])
        print( list1 )
    elif s+l[k]+l[k+1]<=target_sum :
        sum_of_subset(s+l[k],k+1,rem-l[k])
    if s+rem-l[k]>=target_sum and s+l[k+1]<=target_sum :
        x[k]=0
        sum_of_subset(s,k+1,rem-l[k])
```

```
        sum_of_subset(s+l[k],k+1,rem-l[k])
    if s+rem-l[k]>=target_sum and s+l[k+1]<=target_sum :
        x[k]=0
        sum_of_subset(s,k+1,rem-l[k])

l=[]
n=int(input("Enter number of elements"))
total=0
for i in range (0,n):
    ele=int(input())
    l.append(ele)
    total=total+ele
l.sort()
target_sum=int(input("Enter required Sum"))
x=[0]*(n+1)
sum_of_subset(0,0,total)
```

3) Build the state space tree and generate all possible 3-color,4-node graph.



4) Question incomplete

5) Solve the following instance of travelling salesperson problem using Least Cost Branch and Bound

5)

$$\begin{bmatrix} \infty & 12 & 5 & 7 \\ 11 & \infty & 3 & 6 \\ 4 & 9 & \infty & 18 \\ 10 & 3 & 2 & \infty \end{bmatrix} \begin{matrix} 5 \\ 3 \\ 4 \\ 2 \end{matrix} \rightarrow \begin{bmatrix} \infty & 7 & 0 & 2 \\ 8 & \infty & 0 & 3 \\ 0 & 5 & \infty & 14 \\ 8 & 1 & 0 & \infty \end{bmatrix}$$

14

0 1 0 2 = 3

Reduced Cost
$\Rightarrow 14+3=17$

Reduced Matrix :

$$\begin{matrix} & 1 & 2 & 3 & 4 \\ 1 & \infty & 6 & 0 & 0 \\ 2 & 8 & \infty & 0 & 1 \\ 3 & 0 & 4 & \infty & 12 \\ 4 & 8 & 0 & 0 & \infty \end{bmatrix}$$

Cost = 17

$1 \rightarrow 2$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 \\ 0 & \infty & \infty & 12 \\ 8 & \infty & 0 & \infty \end{bmatrix}$$

0 0 0 1

$\Rightarrow$ Cost
$\Rightarrow c(1,2)+\hat{r}+\hat{r}$
$\Rightarrow 6+17+1$
$\Rightarrow 24$

$$\rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 0 \\ 0 & \infty & \infty & 11 \\ 8 & \infty & 0 & \infty \end{bmatrix}$$

$1 \rightarrow 3$

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & 12 \\ 8 & \infty & \infty & \infty \end{bmatrix}$$

$\Rightarrow$ Cost

$$\rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 0 \\ \infty & \infty & \infty & 11 \\ 8 & \infty & 0 & \infty \end{bmatrix}$$

5)

$$\begin{bmatrix} \infty & 12 & 5 & 7 \\ 11 & \infty & 3 & 6 \\ 4 & 9 & \infty & 18 \\ 10 & 3 & 2 & \infty \end{bmatrix}\begin{matrix}5\\3\\4\\2\end{matrix} \rightarrow \begin{bmatrix} \infty & 7 & 0 & 2 \\ 8 & \infty & 0 & 3 \\ 0 & 5 & \infty & 14 \\ 8 & 1 & 0 & \infty \end{bmatrix}$$

14

0 1 0 2 = 3

Reduced Cost
$\Rightarrow 14+3 = 17$

Reduced Matrix :

	1	2	3	4
1	∞	6	0	0
2	8	∞	0	1
3	0	4	∞	12
4	8	0	0	∞

Cost = 17

1 → 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 \\ 0 & \infty & \infty & 12 \\ 8 & \infty & 0 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 0 \\ 0 & \infty & \infty & 11 \\ 8 & \infty & 0 & \infty \end{bmatrix}$$

0 0 0 1

$\Rightarrow$ Cost
$\Rightarrow C(1,2) + r + \hat{r}$
$\Rightarrow 6 + 17 + 1$
$\Rightarrow 24$

1 → 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 8 & \infty & \infty & 1 \\ \infty & 4 & \infty & 12 \\ 8 & 0 & \infty & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 0 \\ \infty & 4 & \infty & 11 \\ 0 & 0 & \infty & \infty \end{bmatrix}$$

8 0 0 1

$\Rightarrow$ Cost
$\Rightarrow C(1,3) + r + \hat{r}$
$\Rightarrow 0 + 17 + 9$
$\Rightarrow 26$

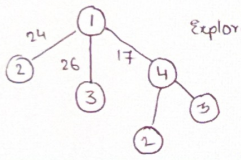
1 → 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & \infty \\ 0 & 4 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix}$$

0 0 0 0

$\Rightarrow$ Cost
$\Rightarrow C(1,4) + r + \hat{r}$
$\Rightarrow 0 + 17 + 0$
$\Rightarrow 17$

→ Same Matrix



Explore (4) now

Reduced Matrix : $\begin{bmatrix} \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & \infty \\ 0 & 4 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix}$

4 → 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

$TC = C(4,2) + r + \hat{r}$
$= 0 + 17 + 0$
$= 17$

4 → 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ 8 & \infty & \infty & \infty \\ \infty & 4 & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

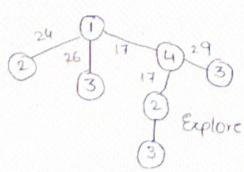
8 4

$TC = C(4,3) + r + \hat{r}$
$= 0 + 17 + 12$
$= 29$

2 → 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

$\Rightarrow C(2,3) + r + \hat{r}$
$\Rightarrow 0 + 17 + 0 = 17$

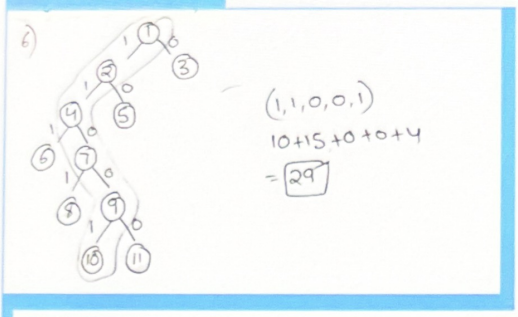


Explore

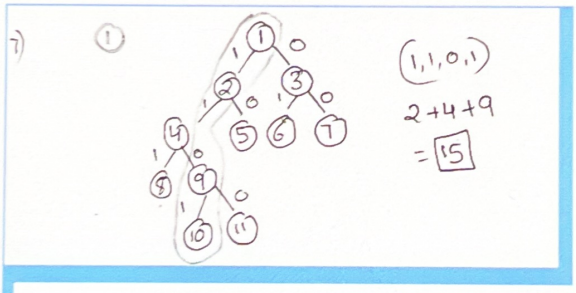
TOTAL COST = 17
PATH : 1 - 4 - 2 - 3

Select all

6) Build the state space tree generated by LCBB by the following knapsack problem n=5, (p1, p2, p3, p4, p5) = (10, 15, 6, 8,4), (w1, w2, w3, w4, w5) = (4, 6, 3, 4, 2) and m = 12.



7) Build the state space tree generated by FIFO knapsack for the instance N=4 , (P1, P2, P3, P4)= (10, 10, 12, 18) , (w1, w2,w3,w4) = (2, 4, 6, 9) , m=15

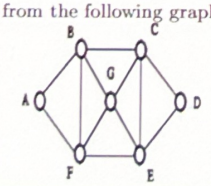


8) Solve the following instance of travelling salesperson problem using Least Cost Branch Bound

Same as 5th

9)

9	Identify Hamiltonian cycle from the following graph

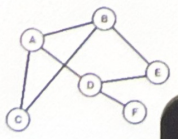


ABGCDEFA

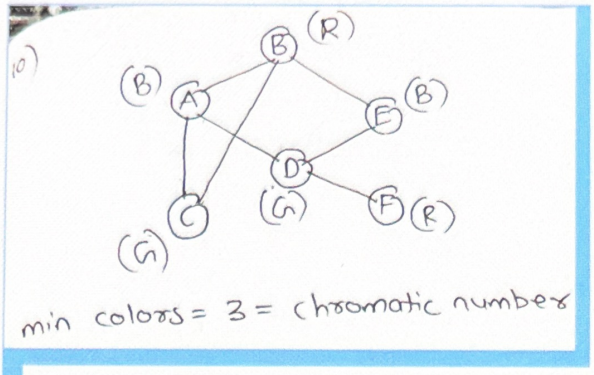
etc

10)

10	Apply the backtracking algorithm to find chromatic number for the following graph



Select all



PART B

1) Develop an algorithm for the N-queens problem using backtracking.

Link: N-queens problem using backtracking

```
#Number of queens
print ("Enter the number of queens")
N = int(input())

#chessboard
#NxN matrix with all elements 0
board = [[0]*N for _ in range(N)]

def is_attack(i, j):
    #checking if there is a queen in row or column
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonals
```

```
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queen(n):
    #if n is 0, solution found
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking                    een here or not
            queen will                     place is being
            attacked or
            if (not(is_attack(i,j))) and (board[i][j]!=1):
```

Select all

```
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False

def N_queen(n):
    #if n is 0, solution found
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking if we can place a queen here or not
            queen will not be placed if the place is being
            attacked or already occupied'''
            if (not(is_attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queen(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queen(N)
for i in board:
    print(i)
```

Explanation

is_attack(int i,int j) → This is a function to check if the cell (i,j) is under attack by any other queen or not. We are just checking if there is any other queen in the row 'i' or column 'j'. Then we are checking if there is any queen on the diagonal cells of the cell (i,j) or not. Any cell (k,l) will be diagonal to the cell (i,j) if k+l is equal to i+j or k-l is equal to i-j.

N_queen → This is the function where we are really implementing the backtracking algorithm.

if(n==0) → If there is no queen left, it means all queens are placed and we have got a solution.

if((!is_attack(i,j)) && (board[i][j]!=1)) → We are just checking if the cell is available to place a queen or not. is_attack function will check if the cell is under attack by any other q is making sure that the cell is vacant. If these conditio put a queen in the cell – board[i][j] = 1.

Select all

diagonal cells of the cell (i,j) or not. Any cell (k,l) will be diagonal to the cell (i,j) if k+l is equal to i+j or k-l is equal to i-j.

N_queen → This is the function where we are really implementing the backtracking algorithm.

if(n==0) → If there is no queen left, it means all queens are placed and we have got a solution.

if((!is_attack(i,j)) && (board[i][j]!=1)) → We are just checking if the cell is available to place a queen or not. is_attack function will check if the cell is under attack by any other queen and board[i][j]!=1 is making sure that the cell is vacant. If these conditions are met then we can put a queen in the cell – board[i][j] = 1.

if(N_queen(n-1)==1) → Now, we are calling the function again to place the remaining queens and this is where we are doing backtracking. If this function (for placing the remaining queen) is not true, then we are just changing our current move – board[i][j] = 0 and the loop will place the queen in some other position this time.

2) Apply a backtracking method and solve subset-sum problems and discuss the possible solution strategies.

Same as Part A 2nd q

3) Apply graph colouring technique and write an algorithm for m-colouring problems.

In this problem, an undirected graph is given. There are also provided m colors. The problem is to find if it is possible to assign nodes with m different colors, such that no two adjacent vertices of the graph are of the same colors. If the solution exists, then display which color is assigned on which vertex.

Starting from vertex 0, we will try to assign colors one by one to different nodes. But before assigning, we have to check whether the color is safe or not. A color is not safe whether adjacent vertices are containing the same color.

Algorithm

isValid(vertex, colorList, col)

Input – Vertex, colorList to check, and color, which is trying to assign.

Output – True if the color assigning is valid, otherwise false.

```
Begin
   for all vertices v of th
      if there is an edge t                olorList[i], then
         return false
   done
   return true
End
```

Select all

such that no two adjacent vertices of the graph are of the same colors. If the solution exists, then display which color is assigned on which vertex.

Starting from vertex 0, we will try to assign colors one by one to different nodes. But before assigning, we have to check whether the color is safe or not. A color is not safe whether adjacent vertices are containing the same color.

Algorithm

isValid(vertex, colorList, col)

Input – Vertex, colorList to check, and color, which is trying to assign.

Output – True if the color assigning is valid, otherwise false.

```
Begin
   for all vertices v of the graph, do
      if there is an edge between v and i, and col = colorList[i], then
         return false
   done
   return true
End
```

4) Explain an algorithm for the Hamiltonian cycle with an example.

In an undirected graph, the Hamiltonian path is a path that visits each vertex exactly once, and the Hamiltonian cycle or circuit is a Hamiltonian path, where there is an edge from the last vertex to the first vertex.

In this problem, we will try to determine whether a graph contains a Hamiltonian cycle or not. And when a Hamiltonian cycle is present, also print the cycle.

Algorithm

isValid(v, k)

Input – Vertex v and position k.

Output – Checks whether placing v in the position k is valid or not.

```
Begin
   if there is no edge between node(k-1) to v, then
      return false
   if v is already taken, then
      return false
   return true; //otherwise it is valid
End
```

4) Explain an algorithm for the Hamiltonian cycle with an example.

In an undirected graph, the Hamiltonian path is a path that visits each vertex exactly once, and the Hamiltonian cycle or circuit is a Hamiltonian path, where there is an edge from the last vertex to the first vertex.

In this problem, we will try to determine whether a graph contains a Hamiltonian cycle or not. And when a Hamiltonian cycle is present, also print the cycle.

Algorithm

isValid(v, k)

Input – Vertex v and position k.

Output – Checks whether placing v in the position k is valid or not.

```
Begin
   if there is no edge between node(k-1) to v, then
      return false
   if v is already taken, then
      return false
   return true; //otherwise it is valid
End
```

5) Explain properties of LC search.

- Least Cost Branch and Bound is a way of finding an optimal solution from the state space tree.
- The search for an answer node can be fastened by using an "intelligent" ranking function c(.) for live nodes. The next E-node is selected on the basis of a ranking function used in LC Search.
- The node x is assigned a rank using:
  `c( x ) = f(h(x)) + g( x )`
  - where, c(x) is the cost of x.
  - h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.
  - g(x) is an estimate of the additional effort needed to reach an answer node from x.
- LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.
- An LC-search coupled with bounding functions is called an LC-branch and bound search.

6) Explain control abstraction for LC Search.

6) Explain control abstraction for LC Search.

Control Abstraction for LC-search

Let **t** be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the sub tree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.

LC search uses ĉ to find an answer node. The algorithm uses two functions

1. Least-cost()
2. Add_node().

Least-cost() finds a live node with least c(). This node is deleted from the list of live nodes and returned.
Add_node() to delete and add a live node from or to the list of live nodes.
Add_node(x)adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

7 and 8) Explain the principle of FIFO and LIFO branch and bound.

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node.

However branch and Bound differs from backtracking in two ways:

- It has a branching function, which can be a depth first search, breadth first search or based on the bounding function.
- It has a bounding function, which goes far beyond the feasibility test as a means to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node Branch and Bound is the generalisation of both graph search strategies, BFS and DFS.
A BFS-like state space search is called a FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).

FIFO Branch and bound

- FIFO Branch and Bound is a BFS.
- In FIFO Branch and Bound , children of E-Node (or Live nodes) are inserted in a queue.
- Implementation of list of live nodes as a queue
  Least() Removes the head of the Queue
  Add() Adds the node to the end of the Queue

A DFS-like state space search is called a LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

LIFO Branch and bound

FIFO Branch and bound

- FIFO Branch and Bound is a BFS.
- In FIFO Branch and Bound , children of E-Node (or Live nodes) are inserted in a queue.
- Implementation of list of live nodes as a queue
  Least() Removes the head of the Queue
  Add() Adds the node to the end of the Queue

A DFS-like state space search is called a LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

LIFO Branch and bound

LIFO Branch and Bound is a D-search (or DFS).
In LIFO Branch and Bound children of E-node (live nodes) are inserted in a stack
Implementation of List of live nodes as a stack
Least() Removes the top of the stack
ADD() Adds the node to the top of the stack

9) Apply the method of reduction to solve travelling salesperson problems using branches and bonds.

Refer this link: Travelling Salesman Problem | Branch & Bound | Gate Vidyalay

10) Explain TSP using branch and bound method with example

Same as the 9th question.

11) Explain the basic principle of Backtracking and list the applications of Backtracking.

Backtracking is a technique based on algorithms to solve problems. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems:

- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In a backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example:
Let's use this backtracking problem to find the solution to the N-Queen

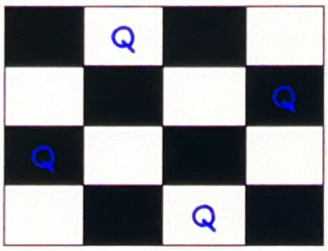
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In a backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example:
Let's use this backtracking problem to find the solution to the N-Queen Problem.

In the N-Queen problem, we are given an NxN chessboard and we have to place n queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. Here, we will do the 4-Queen problem.

Here, the solution is –



To solve the queen's problem, we will try placing the queen into different positions in one row. And checks if it clashes with other queens. Current

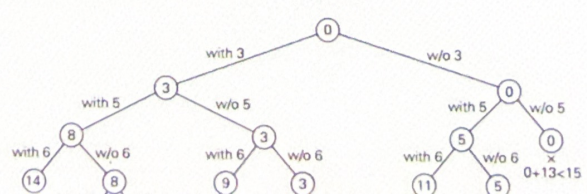
positioning of queens if there are any two queens attacking each other. If they are attacking, we will backtrack to the previous location of the queen and change its positions. And check the clash of the queens again.

Algorithm

```
Step 1 - Start from 1st position in the array.
Step 2 - Place queens in the board and check. Do,
   Step 2.1 - After placing the queen, mark the position as a part of the solution
and then recursively check if this will lead to a solution.
   Step 2.2 - Now, if placing the queen doesn't lead to a solution and trackback
and go to step (a) and place queens to other rows.
   Step 2.3 - If placing queen returns a lead to solution return TRUE.
Step 3 - If all queens are placed return TRUE.
Step 4 - If all rows are tried and no solution is found, return FALSE.
```

12) Explain Backtracking technique and solve the following instance for the subset problem s=(1,3,4,5) and d=11.

Backtracking explained in the previous question.



12) Explain Backtracking technique and solve the following instance for the subset problem s=(1,3,4,5) and d=11.

Backtracking explained in the previous question.

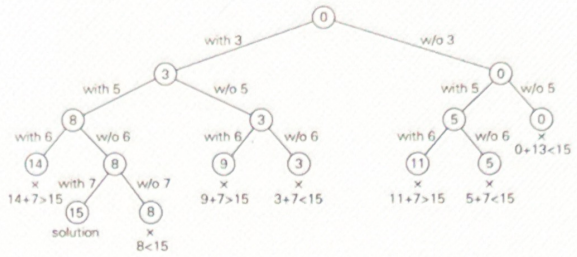


FIGURE 12.4 Complete state-space tree of the backtracking algorithm applied to the instance $A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

Solve it like this question.

13) Construct the portion of the state space tree generated by LCBB for the knapsack instance: n=5,(p1,p2,p3,p4,p5)=(w1,w2,w3,w4,w5)=(4,4,5,8,9) and m=15

14) Explain an algorithm for 4-queens problem using backtracking

Refer Part B 11th q

15) Apply Backtracking technique to solve the following instance for the subset problem s=(6,5,3,7) and d=15.

Same as 12th question

16) Build the portion of state space tree generated by FIFOBB for the job sequencing with deadlines instance n=5 , (p1,p2,..,p5) =(6,3,4,8,5), (t1,t2,..t5) = (2,1,2,1,1) and (d1,d2,..,d5)=(3,1,4,2,4).What is the penalty corresponding to an optimal solution.

17) Solve the solution for 0/1 knapsack problem using dynamic programming N=3 , m=6 profits (p1,p2,p3) = (1,2,5) weights (w1,w2,w3) = (2,3,4)

Refer this link : 0/1 Knapsack Problem | Dynamic Programming | Example | Gate Vidyalay

19) Solve knapsack problem by Dynamic Programming method n=6, (p1, p2,...p6)= (w1,w2,....w6)= (100,50,20,10,7,3) and m=165.

Same as 17th

20) Define greedy method.

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Advantages of Greedy Approach

- The algorithm is easier to describe.

- This algorithm can perform better than other algorithms (but, not in all cases).

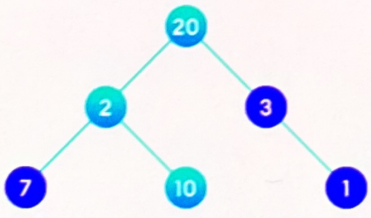
Greedy Approach

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.

2. Our problem is to find the largest path. And, the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.

3. Finally the weight of an only child of **3** is **1**. This gives us our final result `20 + 3 + 1 = 24`.

However, it is not the optimal solution. There is another path that carries more weight (`20 + 2 + 10 = 32`) as shown in the image below.



Longest path