DESIGN AND ANALYSIS OF ALGORITHMS...

ASSIGNMENT 1

NAME : NIVETHA.B.S

CLASS : II CSE B

REG NO : 727822TUCS132

DATE : 26.10.2023

1. Explain the subset sum problem and provide an algorithm in c++ to solve it using backtracking. Discuss the time and space complexity of your solution.

Subset sum problem:

The subset sum problem is to find a subset 's' of the given set S = {S1, S2, S3, ... Sn} where the elements of the set S are n positive integers in such a manner that s ∈ S and sum of the elements of subset 's' is equal to some positive integer 'x'.

Example: Given a set S = {3, 4, 5, 6} and sum x = 9

subset {6, 3} and {4, 5}

Algorithm:

- Construct a state space tree - it is a type of Binary tree.
- The root of the tree is selected in such a way that represents that no decision is yet taken on any input.
- So consider the root node as 0.
- The set should be in ascending order.
- Left child includes S_i right child excludes S_i.
- Each node stores the partial solution.
- Checks the child is less than the sum, if it is less than sum then proceed to include

the array values.

- If the child is greater than sum exclude the value which you have added before then proceed.
- Any stage the sum is equal to x then your search is successfull and terminates.
- If the last end is not equal to sum then it is considered as dead end.
- Dead end represents the sum of s' is too large or too small.
- We are using backtracking to solve this problem. So it checks for other possible solution.

TIME COMPLEXITY:

Time complexity of this backtracking solution is exponential. In the worst case, it explores all possible subsets, which is O(2^n) where n is the number of elements in the set.

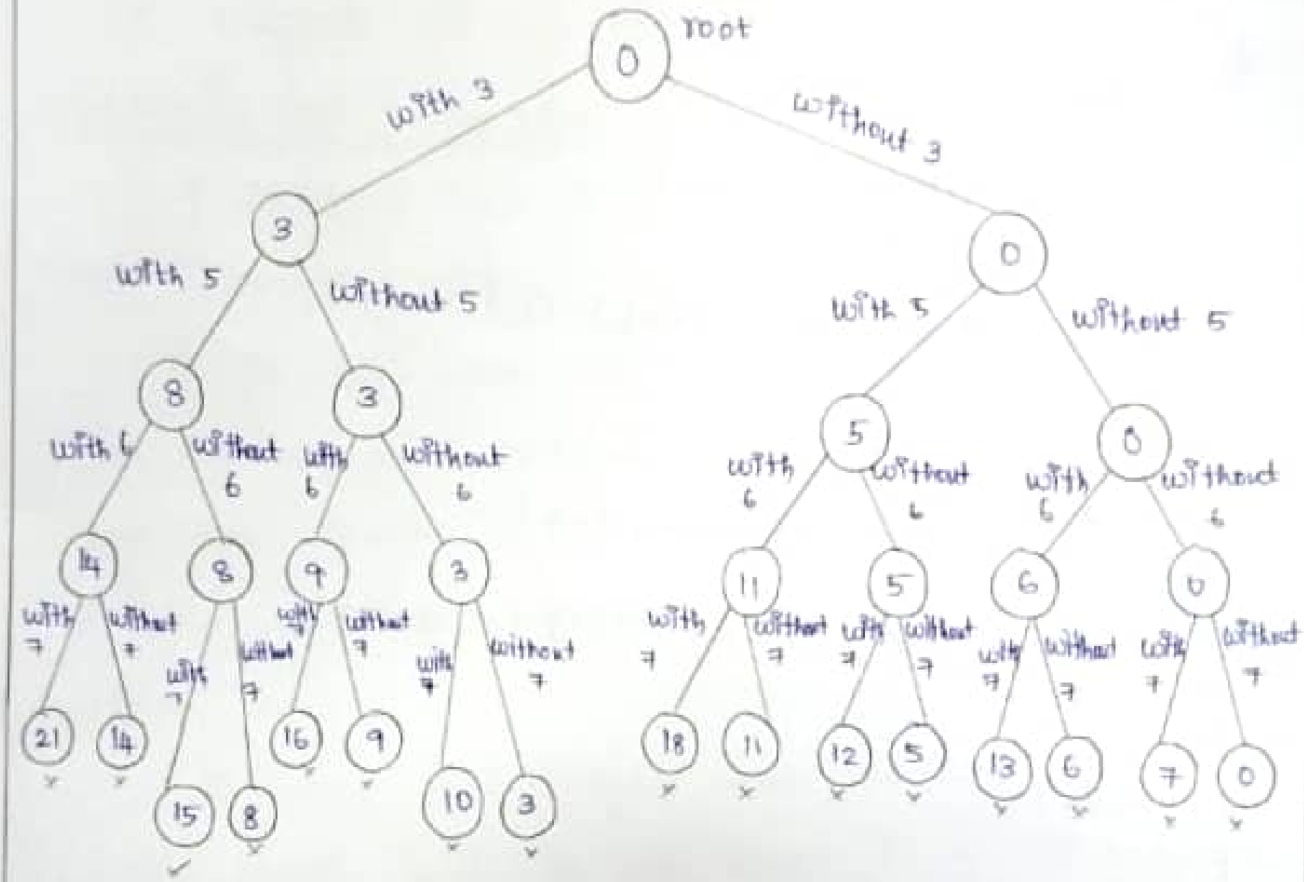
SPACE COMPLEXITY:

Space complexity of this algorithm is O(n) for the function call, as it makes recursive calls.

CONSTRUCTION OF STATE SPACE TREE

S = {3,5,6,7} Sum = 15

set consist of 3,5,6,7 the sum should be 15



↓
solution end - 1

This sum has only one solution end and other ends are considered as dead end.

solution → {3, 5, 7} ∵ 3 + 5 + 7 = 15

CODING:

```
# include <iostream>
 using namespace std;
 static int count = 0;
 void subset ( int arr[], int s, int l, int sum, int n)
 {
        if (s == sum)
        {
             count ++;
             if (l<n)
             {
              subset (arr, s-arr[l-1], l, sum, n);
             }
        }
        else
        {    for (int i=l; i<n; i++)
             {
              subset (arr, s+arr[i], i+1, sum, n);
          }  }
 }
 int main ()
 {
      int n,sum;
      cin>>n;
       int arr[n];
       for( int i=0;i<n;i++)
       {
           cin>>arr[i];
       }
       cin>> sum;
       subset (arr,0,0,sum,n);
       cout << count ;
  }
```

2. Implement a C++ program to find a knight's Tour on an nxn chessboard, starting from a given initial position. Ensure that the knight visits each square exactly once. Display the solution if possible, or indicate if it's not solvable.

KNIGHT's TOUR PROBLEM:

- knight's tour is a problem in which we are provided with a NxN chessboard and a knight.

- It can move either two squares horizontally and one square vertically or two squares vertically and one square horizontally in each direction. So the complete movement looks like English letter 'L'.

- Make the knight cover all the cells of the board and it can move to a cell only once.

- knight cannot move outside the board.

It is possible to find all the possible locations the knight can move to from the given location by using the array that stores the knight movement's relative position from any location.

If the current location is (i, j) we can move to (i+ row[k], j+ col[k]) for 0<=k<=7 using the following array.

row[] = [2, 1, -1, -2, -2, -1, 1, 2]

col[] = [1, 2, 2, 1, -1, -2, -2, -1]

so from a position (i, j) in chessboard, the moves are (i+2, j+1), (i+1, j+2), (i-1, j+2), (i-2, j+1), (i-2, j-1), (i-1, j-2), (i+1, j-2), (i+2, j-1).

	(i-2, j-1)		(i-2, j+1)		
(i-1, j-2)				(i-1, j+2)	
		(0,0)			
(i+1, j-2)				(i+1, j+2)	
	(i+2, j-1)		(i+2, j+1)		

CODING:

```
# include <iostream>
using namespace std;
bool issafe (int x, int y, int **cb, int n)
{
  return (x>=0 && x<n && y>=0 && y<n && cb[x][y]==-1);
}
```

```
bool KTmove (int x, int y, int movei, int **cb, int xmove[], int ymove[],
                                                              int n)
{
      int k, nextx, nexty;
       If (movei == n*n)
       {
         return true;
        }
       for (int k=0; k<8; k++)
       {
          nextx = x + xmove[k];
          nexty = y + ymove[k];
         If (issafe (nextx, nexty, cb, n))
         {
            cb[nextx][nexty] = movei;
           If (KTmove (nextx, nexty, movei+1, cb, xmove, ymove, n))
           {
             return true;
            }
           else
            {
              cb[nextx][nexty] = -1;
            }
         }
      }
     return false;
  }
bool KTsol (int **cb, int n)
{   int xmove[8] = {2,1,-1,-2,-2,-1,1,2};
     int ymove[8] = {1,2,2,1,-1,-2,-2,-1};
      cb[0][0] = 0;
      If (!KTmove (0,0,1, cb, xmove, ymove, n))
      {
         return false;
       }
      else
       {
        return true;
 }     }
```

```
int main()
{
    int n;
    cin>>n;
    int **Cb=new int*[n];
    for (int i=0; i<n; i++)
    {
        cb[i] = new int[n];
        for (int j=0;j<n;j++)
        {
            cb[i][j] = -1;
        }
    }
    if (KTSol(cb,n))
    {
        cout << "Solution exists: " << endl;
        for (int i=0; i<n; i++)
        {
            for (int j=0;j<n;j++)
            {
                cout << cb[i][j] << " ";
            }
            cout << endl;
        }
    }
    else
    {
        cout << " No solution exists.";
    }
}
```

INPUT: 8

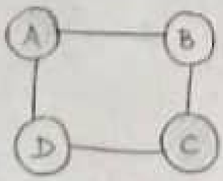
OUTPUT: Solution exists:

```
0   59  38  33  30  17  8   63
37  34  31  60  9   62  29  16
58  1   36  39  32  27  18  7
35  48  41  26  61  10  15  28
42  57  2   49  40  23  6   19
47  50  45  54  25  20  11  14
56  43  52  3   22  13  24  5
51  46  55  44  53  4   21  12
```

3. Develop a C++ program that solves the M-coloring problem for a given graph. Determine if it is possible to color the vertices with at most M colors such that no two adjacent vertices have the same color. Display the coloring solution if possible.

M coloring problem:

- Given an undirected graph and a number m, determine if the graph can be coloured with atmost m colours such that no two adjacent vertices of the graph are colored with same colour.
- Here coloring of a graph means the assignment of colours to all vertices.
- It is also known as M-colourability decision problem.
- The M colorability optimization problem deals with the smallest Integer m for which the graph G can be coloured.
- The Integer M is knaon as chromatic number of the graph.
- If d is the degree of the given graph, then it can be colored with d+1 color.



Steps to color graph:

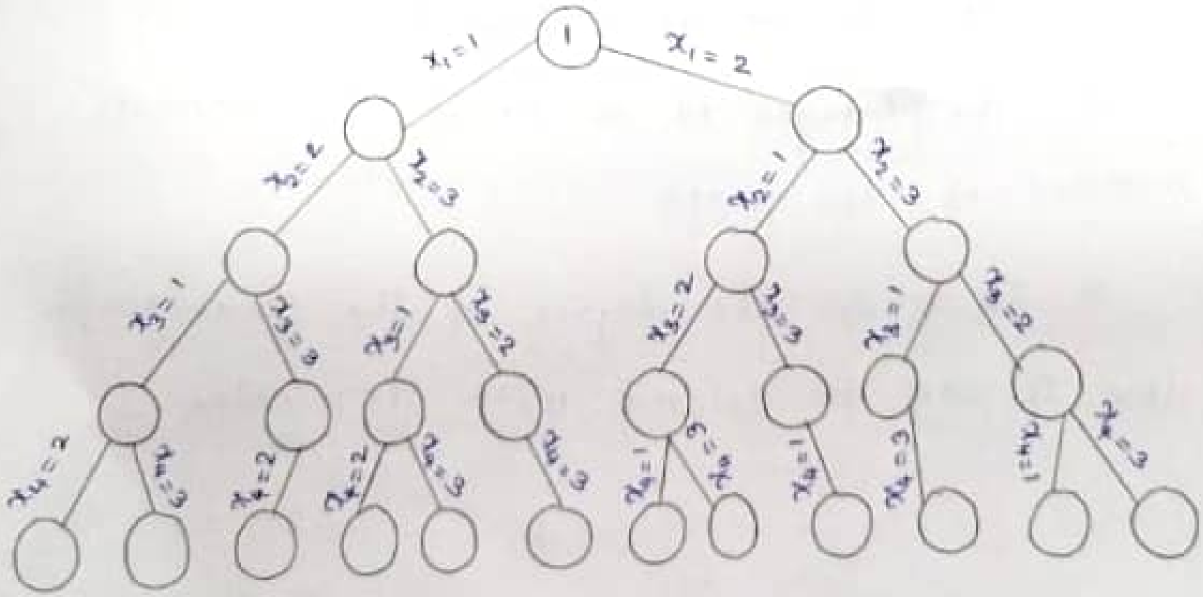
- Confirm whether it is valid to colour the vertex with current color ? by checking whether any of its adjacent vertices are coloured with the same color.

- If yes then color it or else try with another color.

- If no other colour is available then backtrack i.e un-colour the last coloured vertex and return false.

- After each coloring check all vertices are colored or not. If yes then end the program by returning tree.

- Now select any one of the uncoloured adjacent vertices of the currently coloured vertex x and repeat the whole process.



CODING:

```
# include <iostream>
using namespace std;
int V=4;
bool issafe(int ve,int **graph,int *color,int c)
{
    for (int i=0; i<V; i++)
    {
        if (graph[ve][i]==1 && color[i]==c)
        { return false;
        }
    }
    return true;
}
bool assigncolor (int **graph, int *color, int m,int ve)
{
    if (ve==V)
    { return true;
    }
    else
    {
        for (int c=1; c<=m; c++)
        { if (issafe (ve,graph,color,c))
          {
            color [ve]=c;
            if (assigncolor(graph,color,m,ve+1))
            {
              return true;
            }
            else
            {
              return color [ve]=0;
            }
          }
        }
        return false;
}
bool graphcolor (int **graph, int *color, int m)
{
    if (! assigncolor (graph,color,m,0))
    { return false;
    }
    else {
          return true;
        }
 }

int main()
{
        int i,j,m;
        int *color = new int [V];
        int **graph = new int *[V];
        for (i=0; i<V; i++)
        {
            graph[i] = new int [V];
             color [i] = 0;
           for (j=0; j<V; j++)
           {
              cin >> graph[i][j];
        }  }
        cin >>m;

        if (graphcolor (graph, color,m))
        {
            cout << "Solution exists:" << endl;
            for (i=0; i<V; i++)
            {
               cout << color [i] << " ";
          } }
         else
          {
            cout << "Solution does not exist";
          }
        return 0;
 }
```

INPUT :

```
0 1 1 1
1 1 1 0
1 1 1 1
1 1 1 1
5
```

OUTPUT :

```
Solution Exists:
1 2 3 4
```

4. Design an algorithm to find a Hamiltonian Cycle in a given graph, if one exists. Provide an implementation in C++ and determine it on a sample graph.

Hamiltonian Cycle:

- Hamiltonian path is an undirected graph is a path that visits each vertex exactly once.

- A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian path such that there is an edge in the graph from the last vertex to the first vertex of the Hamiltonian path.

Steps:

- Given a graph G = (V, E) we have to find the Hamiltonian circuit using backtracking approach.

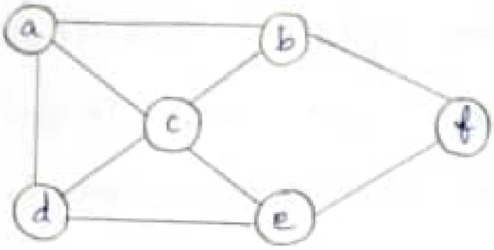
- start to search from any arbitrary vertex say 'a'. this vertex 'a' becomes the root of our implicit tree.

- the first element of our partial solution is the first intermediate vertex of the Hamiltonian cycle that is to be constructed.

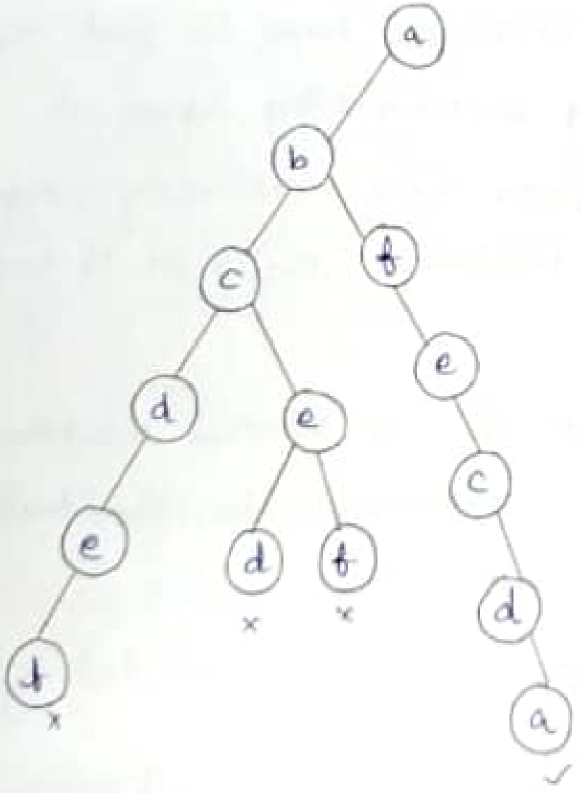
- The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a', then we say that dead end is reached.

- In this case, we backtrack one step and again the search begins by selecting another vertex and backtrack the element from the partial and solution must be removed.

- The search using backtracking is successful if a Hamiltonian cycle is obtained.



Hamiltonian path:



solution: a b f e c d a

CODING:

```
# include <iostream>
using namespace std;
int V, graph[100][100];
void printSolution (int path[])
{
    cout << "Hamiltonian Path:";
    for (int i=0; i<V; i++)
    {   cout << path[i] << " ";
    }
    Cout << path[0] << endl;
}
bool isSafe (int v, int pos, int path[])
{
    if (graph[path [pos-1]][V]==0)
    return false;
    for (int i=0; i<V; i++)
    {
        if (path[i]==V)
        return false;
    }
    return true;
}
bool hamiltonianPathUtil (int path[], int pos)
{
    if (pos==V)
    {
        return true;
    }
    for (int ve=1; ve<V; ve++)
    {
        if (isSafe (ve, pos, path))
        {   Path [pos]= ve;
            if (hamiltonianPathUtil (path, pos+1))
        {
          return true;
        }
        else
         {
          path [pos] = -1;
         }
      }
    }
    return false;
  }

void  findHamiltonianPath()
  {
        int path[V];
        for (int i=0; i<V; i++)
         {
           path[i] = -1;
         }
         path[0]=0;
         if (! hamiltonianPathUtil(path,1))
         {
         cout << "No Hamiltonian Path exists.";
          return;
         }
         cout << " Hamiltonian Path : ";
          for (int i=0; i<V; i++)
          {
           cout << path [i] << " ";
           }
         cout << path [0];
   }
```

```
int main()
{
    int i, j;
    cin >> V;
    for (i=0; i<V; i++)
    {
        for (j=0; j<V; j++)
        {
            cin >> graph[i][j];
        }
    }
    findHamiltonianPath();
    return 0;
}
```

INPUT:

```
5
0 1 0 1 0
1 0 1 1 1
0 1 0 0 1
1 1 0 0 1
0 1 1 1 0
```

OUTPUT:

```
Hamiltonian Path:
0 1 2 4 3 0
```

5. Create a C++ program to solve a sudoku puzzle using a backtracking algorithm. Given an incomplete sudoku grid, find the solution by filling in the missing numbers. Display the solved sudoku grid.

Sudoku puzzle:

- A sudoku puzzle consists of an incomplete 9x9 grid.
- There are nine 3x3 sub grids with some cells already filled with degits 1 to 9.
- The goal is to fill in the remaining cells with the appropriate degits, following the rules of the game.

Constraints:

- A row should not contain the same number more than once.
- A column should not contain the same number more than once.
- Any 3x3 sub matrix cannot have the same number more than once.

3	1	6	5	7	8	4	9	2
5	2	9	1	3	4	7	6	8
4	8	7	6	2	9	5	3	1
2	6	3	4	1	5	9	8	7
9	7	4	8	6	3	1	2	5
8	5	1	7	9	2	6	4	3
1	3	8	9	4	7	2	5	6
6	9	2	3	5	1	8	7	4
7	4	5	2	8	6	3	1	9

CODING:

```
# include <iostream>
using namespace std;
# define n 9
bool issafe (int ** grid, int row, int col, int no)
{
    for (int i=0; i<n; i++)
    {
        if (grid [row] [i]==no)
        {
            return false;
        }
    }
    for (int i=0; i<n; i++)
    {
        if (grid [i] [col]==no)
        {
            return false;
```

```
          }
      }
      int startrow = row - row%3;
      int startcol = col - col%3;
      for (int i=0; i<3; i++)
      {
         for (int j=0; j<3; j++)
         {
            if (grid [i+startrow][j+startcol]==no)
            {
             return false;
            }
         }
      }
      return true;
   }
bool solvesudoku (int **grid, int row, int col)
{
      if (row==n-1 && col==n)
      {
        return true;
      }
      if (col ==n)
      {
         row++;
         col=0;
      }
      if (grid [row][col] >0)
      {
        return solvesudoku(grid, row, col+1);
      }
      for (int no=1; no<=n; no++)
      {
         if (issafe (grid, row, col, no))
         {
           grid [row][col] = no;
```

```
            if (solve sudoku (grid, row, col+1))
            {
              return true;
             }
            else
             {
              grid [row][col] = 0;
             }
           }
         }
          return false;
       }

  int main()
 {
      int ** grid = new int * [n];
       for (int i = 0; i<n; i++)
        {
          grid [i] = new int [n];
          for (int j = 0; j<n; j++)
          {
           cin >> grid [i][j];
        }  }
       if (solvesudoku (grid, 0, 0))
        {
          for (int i = 0; i<n; i++)
          {
           for (int j = 0; j<n; j++)
          {
           cout << grid [i][j] << " ";
           }
          cout << endl;
        }
      }
     else {
         cout << "No solution exists";
       }
 }
```

INPUT:

```
3 0 6 5 0 8 4 0 0
5 2 0 0 0 0 0 0 0
0 8 7 0 0 0 0 3 1
0 0 3 0 1 0 0 8 0
9 0 0 8 6 3 0 0 5
0 5 0 0 9 0 6 0 0
1 3 0 0 0 0 2 5 0
0 0 0 0 0 0 0 7 4
0 0 5 2 0 6 3 0 0
```

OUTPUT:

```
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
```

6. Write a c++ program that uses the sieve of Sundaram algorithm to find all prime numbers less than or equal to n. Implement and test the algorithm, displaying the list of prime numbers.

Sieve of Sundaram:

- It is an efficient algorithm used to find all the prime numbers till a specific numbers say n.

  $m = (n-1)/2;$

- From a list of integers from 1 to remove all integers of the form $i+j+2ij$ where integers i and j range from $1 \leq i \leq j$ and $i+j+2ij \leq m$.

- For each remaining integer k, the integer $2k+1$ is prime and the list gives all the odd primes.

Algorithm:

```
• m = n-1/2;
• L = list of numbers from 1 to m.
• For i in 1 to m:
      Initialize j from i and increment
  till (i+j+2ij) <= m.
```

- Remove $i+j+2ij$ from L.

- For each k remaining in L,

$2k+1$ is prime.

Example:

If $n = 40$ then,

$$m = \frac{(n-1)}{2} = \frac{39}{2}$$

$$m = 19$$

On removing $i+j+2ij$ from L

then $k = 1, 2, 3, 5, 6, 8, 9, 11, 14, 15, 18$

$2k+1$ is prime.

3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 are prime numbers less than 40.

Coding:

```
# include <iostream>
using namespace std;
int main()
{
        int n;
        cin >> n;
        int m = (n-1)/2;
        int arr[n];
```

```
      for (int i=1; i<=n; i++)
      {
          arr[i]=i;
      }
      for (int i=1; i<=m; i++)
      {
        for (int j=i; (i+j+2ij)<=m; j++)
        {
            arr[i+j+(2*i*j)] = 0;
        }
      }
     cout << 2 <<" ";
    for (int i= 1; i<=m; i++)
    {
       if (arr[i]!=0)
        {
           int k = 2* arr[i]+1;
           cout << k <<" ";
        }
     }
    return 0;
 }
```

INPUT:

20

OUTPUT:

```
2 3 5 7 11 13 17 19
```

7. Develop a C++ function to find a prime number greater than a given value 'p' with the condition that the sum of its digits is equal to 's'. Implement this using a backtracking algorithm and return the found prime number.

Prime numbers after P with sum S:

- Given three numbers, sum S, prime P and N.
- Task is to find all N prime numbers greater than P whose sum is equal to S.
- Find all prime numbers between P and S.
- Using backtracking find N prime numbers which sum upto S.

Example:

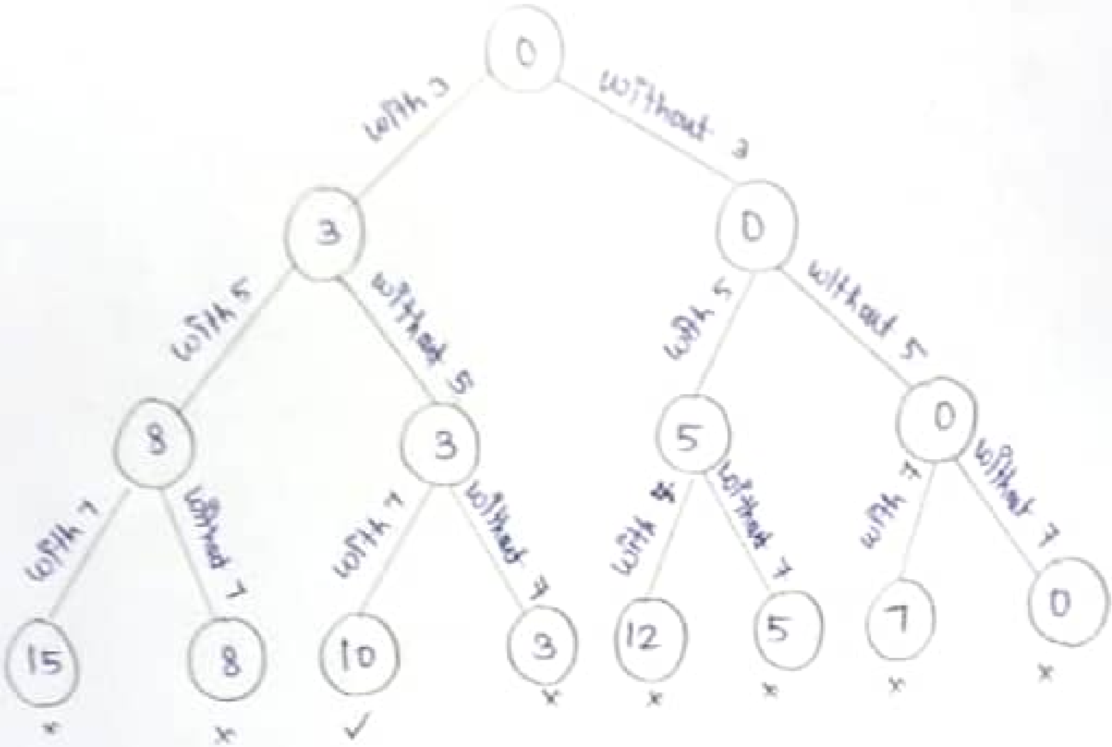
N = 2, P = 5, S = 18

Solution: 7 11

prime numbers greater than 5 less than 18 are: 7 11 13

(7 + 11) = 18

N = 2   P = 2   S = 10   {3,5,7}



solution : 3 + 7 = 10

CODING:

```
# include <iostream>
using namespace std;
bool isprime (int no)
{
   for (int i=2; i<=no/2; i++)
   {
        if (no%i==0)
        {
          return false;
   }    }
   return true;
}
```

```
void ps(int s, int n, int p, int ans[], int ind)
{
        if (s==0  &&  n==0)
         {
             for (int i=0; i<ind; i++)
              {
                cout << ans[i] << "\t";
              }
             cout << endl;
             return;
          }

        for (int i= p+1; i<=s ; i++)
        {
            if (isprime(i))
            {
               ans[ind]=i;
               ps(s-i, n-1, i, ans, ind+1);
            }
        }
   }
int  main()
{
        int s,n ,p;
        cin >> s>>n>>p;
        int ans[n];
        cout << n << " Prime numbers greater than " << p <<
               " with sum = " << s << " are: " << endl;
        ps(s, n, p, ans, 0);
        return 0;
   }
```

INPUT :

```
23   3   2
```

OUTPUT :

```
3 prime numbers greater than 2 with sum=23 are:
 3   7   13
 5   7   11
```