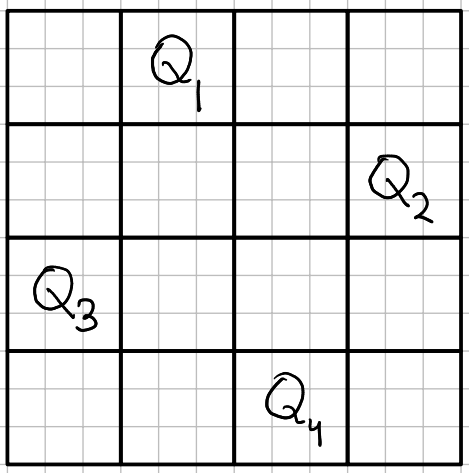
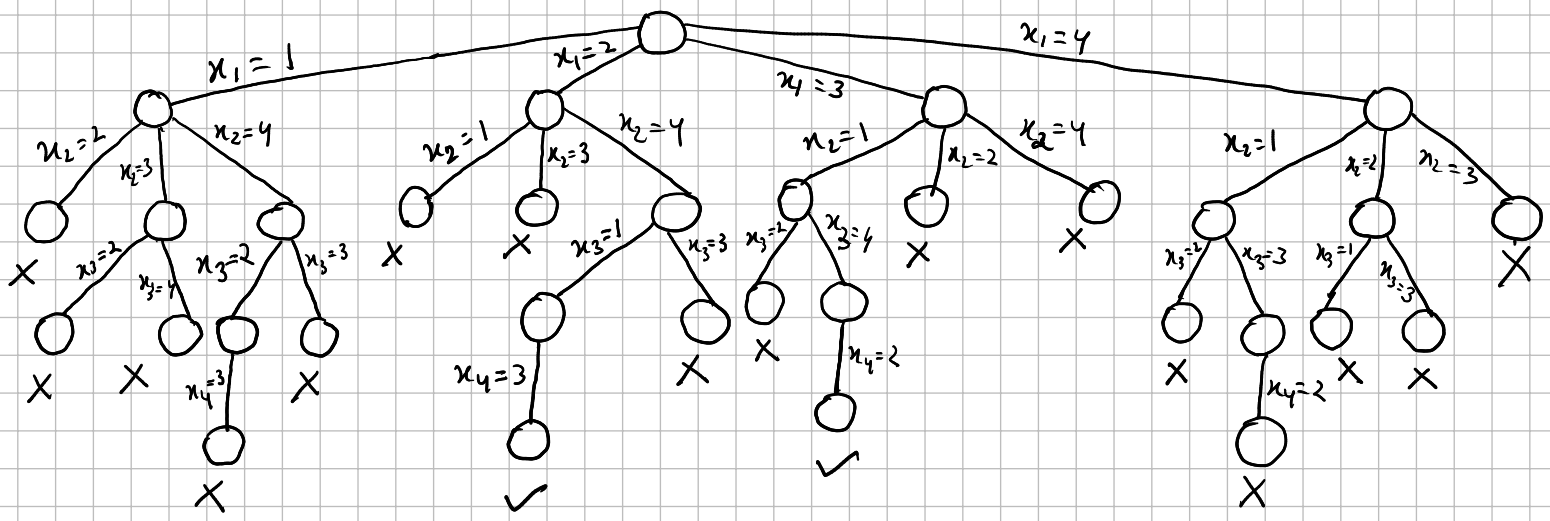
N-QUEENS PROBLEM USING BACKTRACKING





State Space Tree

Solutions are : $x_1 = 2, x_2 = 4, x_3 = 1, x_4 = 3$

$x_1 = 3, x_2 = 1, x_3 = 4, x_4 = 2$

SUM of SUBSETS PROBLEM USING BACKTRACKING

$w[1:6] = \{ 5, 10, 12, 13, 15, 18 \}$

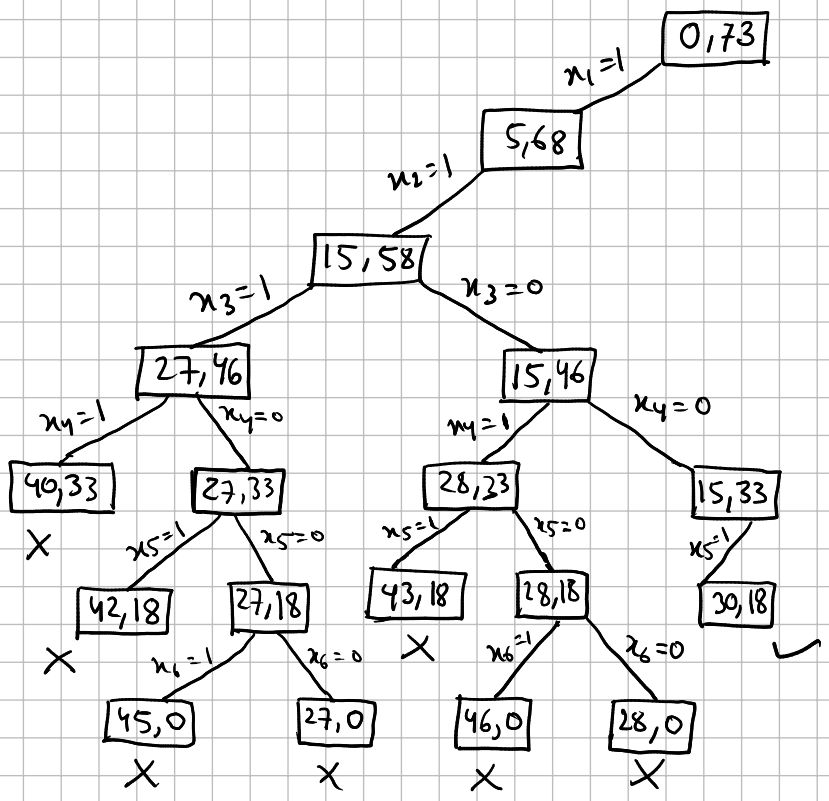
$n = 6$

$m = 30$

Bounding Conditions :

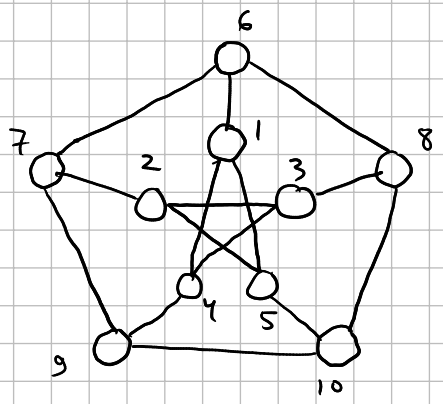
① $\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$

② $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$



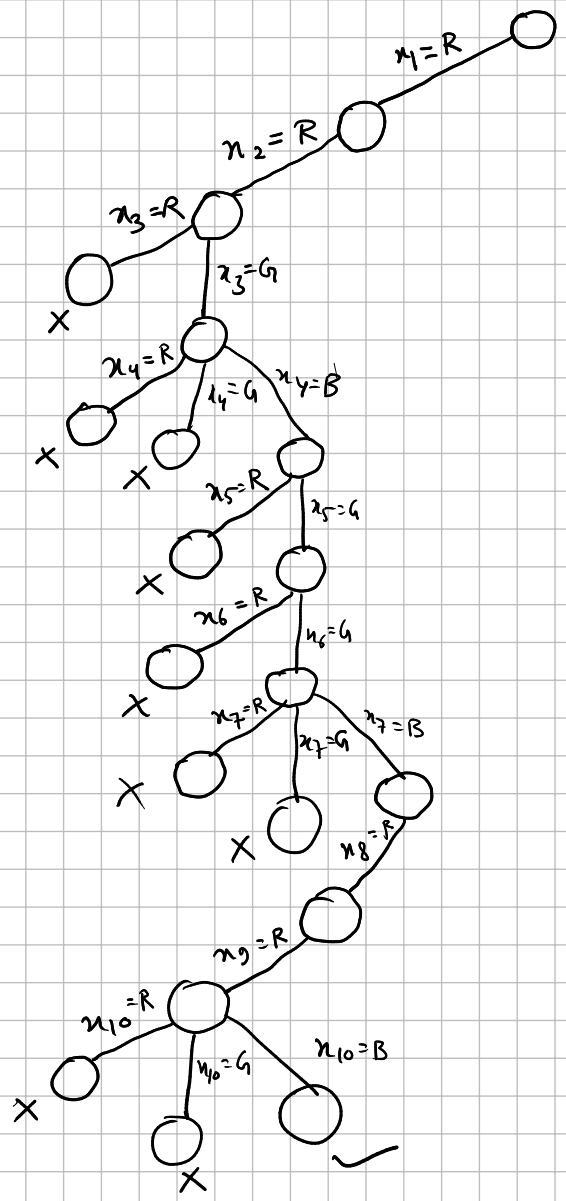
One of the possible solutions = {1,1,0,0,1,0}

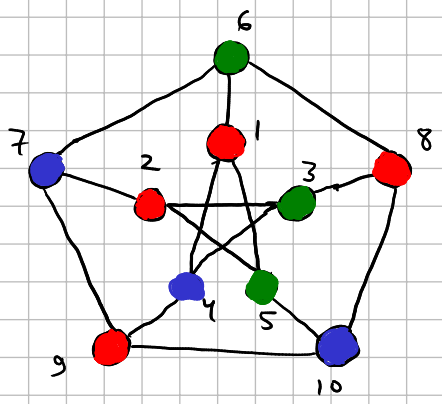
GRAPH COLORING PROBLEM USING BACKTRACKING



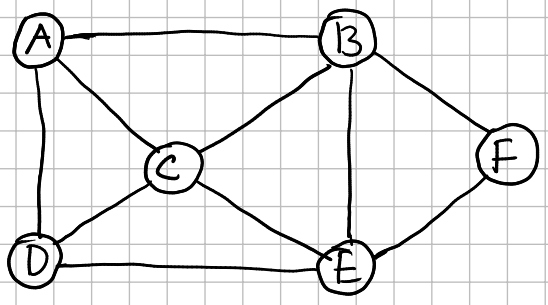
Find at least one solution using Backtracking

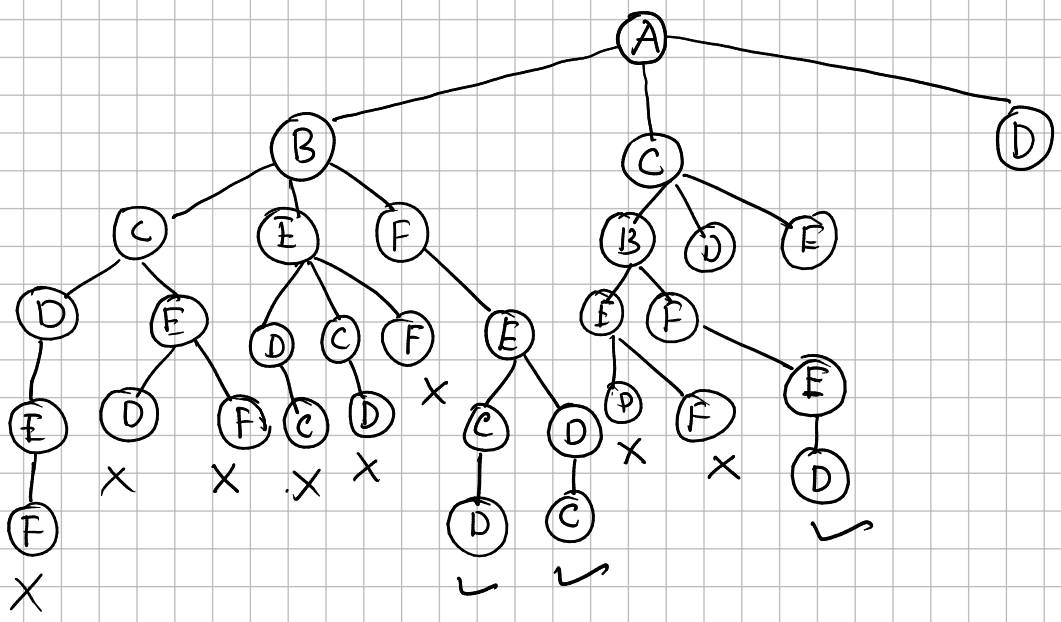
Available colors = R, G, B





HAMILTONIAN CYCLE USING BACKTRACKING





3 possible solutions (there are more) → A B F E D C A
A B F E C D A
A C B F E D A

BACKTRACKING Vs Branch & Bound

① Backtracking is used to solve Decision Problems.

Branch & Bound is used to solve Minimization Problems.

② Backtracking traverses the state space tree in DFS manner.
Branch & Bound traverses the state space tree in BFS manner.

③ In backtracking, the state space tree is searched until the solution is obtained.

In Branch & Bound, as the optimum solution may be present anywhere in the state space tree, the tree needs to be searched completely.

GREEDY vs Dynamic Programming

① In a greedy algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal sdution.

In DP, we make decision at each step considering current problem and solution to previously solved subproblem to calculate optimal solution.

② Greedy → no gurantee of getting optimal solution
DP → guarantees optimal solution

③ Greedy → more efficient in terms of memory as it never looks back to revise previous choices
DP → not efficient in terms of memory as it requires DP table for memoization.

④ Greedy → generally faster
DP → generally slower

Permutations with Duplicates Allowed

```
permute (char arr[], int l, int r)
{
    if (l>=r)
    {
        cout << arr << endl;
    }
    else
    {
        for (int i=l; i<r; i++)
        {
            swap (arr[l], arr[i])
            Permute (arr, l+1, r)
            swap (arr[l], arr[i])
        }
    }
}
```

Permutation with no duplicates allowed (Distinct)

```
swapper(char arr[], int start, int curr)
{
    for (int i=start, i<curr, i++)
        if ( arr[i] == arr[curr])
        return 0;
    return 1;
}
permute(char arr[], int l, int r)
{
    if (l>=r)
    {
        cout << arr << endl;
    }
  else
    {
      for (int i=l; i<r; i++)
        {
            bool check = swapper( arr, l, i);
            if(check)
            {
                swap (arr[l], arr[i]);
                permute ( arr, l+1, r);
                swap(arr[l], arr[i]);
            }
        }
    }
}
```