Unit 6: Backtracking

6.1. Concept of Backtracking, Recursion vs Backtracking

6.2. Backtracking Algorithms:

- Subset-sum Problem
- Zero-one Knapsack Problem
- N-queen problem

Prepared by: Shiv Raj Pant

What is Backtracking?

- Backtracking is one of the most general techniques of algorithm design.
- Backtracking can be used to solve two types of problems:
  - Problems which search for a set of solutions.
  - Problems that try to find optimal solution satisfying some condition.
- In such problems, the desired solution is expressed as n-tuple (x_1, x_2,.. x_n) where x_i belongs to some finite set of elements.
- e.g., consider 0-1 knapsack problem. The solution is tuple ($i_1, i_2, \ldots i_k$) such that for j=1,2,..k, $\sum v[i][j]$ is maximized and $\sum w[i][j] \leq W$.
- Here, we have many feasible solutions and we are interested in some (or one) of them.
- One way to solve these kinds of problems is to try all possible solution and select desired ones. However this approach is very inefficient.

- Backtracking is efficient way of solving problems which avoids trying all possible solutions.

Backtracking Vs. Recursion

Recursion	Backtracking
1. Recursion divides whole problem into subproblems 2. Recursion is when a function calls itself 3. In recursion, function calls itself until it reaches base case. 4. Recursion is just implementation concept. 5. Recursion is like a bottom-up process. We can solve the problem just by using the result of the sub-problem.	1. Backtracking also divides whole problem into subproblems. 2. Backtracking is when the algorithm makes opportunistic decision. If the decision is wrong then the algorithm goes back to previous state and tries another solution. 3. In backtracking, the algorithm explores possible solutions until it reaches best solution. 4. Backtracking may or may not use recursion 5. Backtracking is not strictly bottom-up. Sometimes, it may go top-down. Sometimes we can't solve the problem just by using the result of the sub-problem, we need to pass the information you already got to the sub-problems.

How backtracking works?

- Backtracking algorithm determines solution by systematically searching solution space using tree representation.
  - Each node represents a problem state.
  - All paths from root to other nodes is called state space of the problem
  - Solution is the path from root to state s that defines solution tuple.
- The algorithm starts building the tree from root node in depth-first manner.
- At each step, the algorithm examines if current branch leads to desired solution. If not, the algorithm backtracks and explores other branch.
- Depth-first node generation with bounding function(checking if the solution is good) is called backtracking.

Subset sum problem

"Given n positive numbers w_i $(1 \leq i \leq n)$ and another number m, find all subsets of the numbers whose sums are m".

Example:

W = {11,13,24,7} , m = 31

Then the solutions are {11,13,7} and {24,7}

Here, the solutions can also be represented using index notation {1,2,4} and {3,4}. The numbers in these sets are the indices of actual numbers in the problem

Or

We can also represent the solution as n-tuple of binary numbers {1,1,0,1} and (0,0,1,1} representing whether a particular element is included or not.

How to solve?

Let us construct the solution tree (solution space) for above problem as shown in figure.

The paths from root to other nodes are the possible solutions.

(),(1,2,3,4),(1,2) ….

During, the node generation process, we explore a path to see if it leads to desired solution. Otherwise we stop and backtrack.

We also backtrack when we have found one solution and want to find another solution.

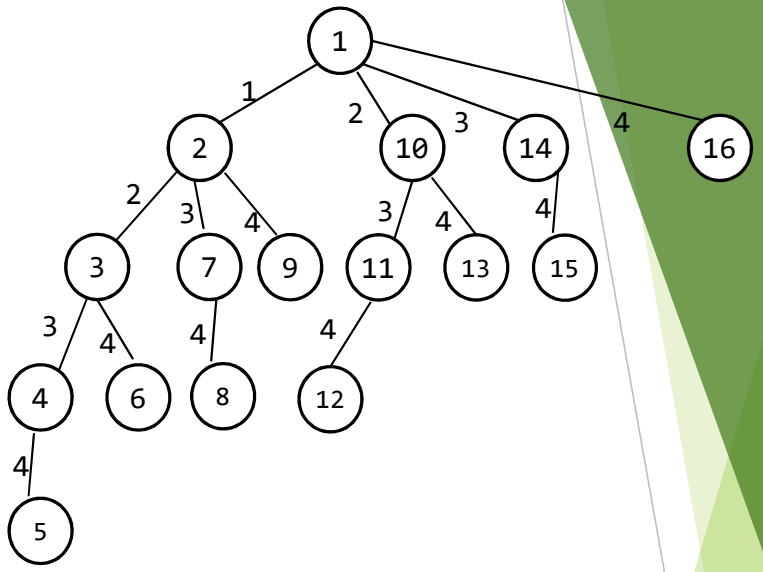


Fig: The solution space for subset sum problem W=(11,13,24,7},m=31.

Here, the edges represent indices of numbers in W. the root node represents empty solution (no element taken). The edge from node 1 to node 16 represents that only 4^{th} element is taken for solution. Nodes are numbered in depth-first order. Note: we can also generate a tree for fixed-tuple form.

Pseudocode:

```
subset_sum(W,sum,i,m,sol)
{
   if(sum == m )
    print sol
   if(sum < m)
    {
     for(j = i; j < n; j++ )    //generate childrens
       {
        subset_sum(W, sum + W[j], j + 1,m,sol U {W[j]} )
       }
    }
}
```

W: input array of numbers
sum: currently evaluated
m: target sum
Sol: solution tuple
i: start index of input
Initially i=0, sol=ϕ

Working:

Above algorithm starts from an empty solution and recursively adds the elements to the solution tuple "sol". Whenever it finds correct solution, it displays the solution and backtracks (last line) to find another solution. If not, it explores the children of current state.

Note: Because of the inherent nature of recursive procedure, the children are always explored in depth-first order (using stack) and the procedure automatically backtracks whenever it finds a solution (sum=m)or the current solution does not lead to solution (sum>m). The generation of feasible child is controlled by the "for" loop and backtracking is controlled by the recursion itself.

Analysis:

How many tree nodes need to be computed ?

Worst case: $O(2^n)$

Prepared by: Shiv Raj Pant

0-1 Knapsack problem

- Problem definition: you know that already!
- Let v[i] and w[i] be the value and weight of i^{th} item. W be the knapsack capacity.
- We can solve the problem by backtracking as follows:
  1. Start from an empty state.
  2. Add items one by one and create child states in depth-first manner, if the total weight of the items in a state do not exceed W
  3. If the weight of the total items added equal to the knapsack capacity, then record the solution and backtrack to find another solution.
  4. If the total weight of the items exceed W, then record the parent state solution and backtrack.
  5. Finally, we find the maximum-valued solution among the recorded solutions.

Example: v[] = {3,4,5,6} w= {2,3,4,5} W = 5

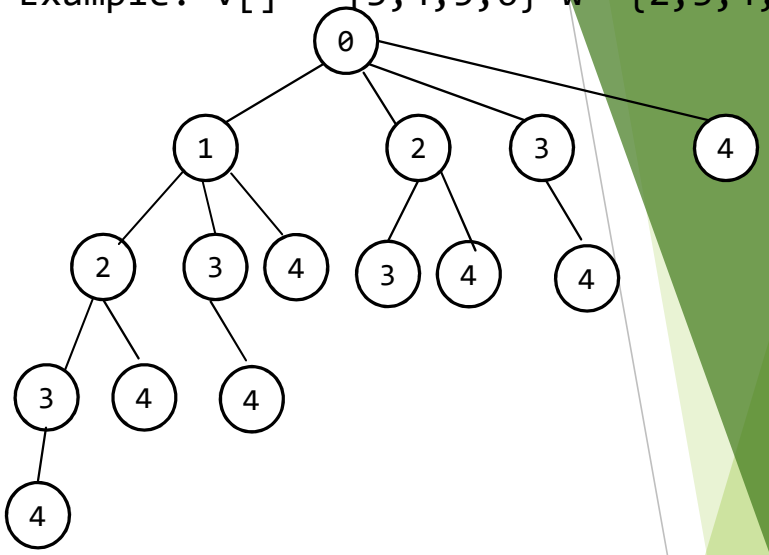
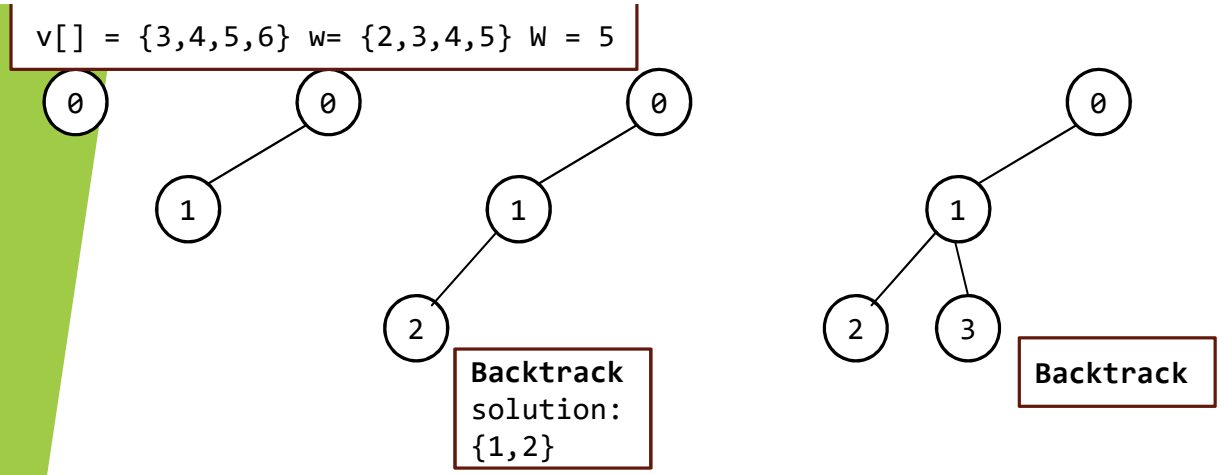


Fig: *The possible solution space for 0-1 Knapsack problem given in the example. The root node represents empty solution (no item taken). The numbers in the nodes represent the item picked at that time.*

Using backtracking, we search the tree in depth-first order. (see the next slide)


v[] = {3,4,5,6} w= {2,3,4,5} W = 5
0
0
0
1
1
2
Backtrack
solution:
{1,2}
0
1
2
3
Backtrack
continue

Algorithm

```
knapsack(v,w,W,sum,i,sol) //initially i=0,sum=0
{
  if(sum == W )
   {
    record sol
    return
   }
  if(sum < W)
   {
    for(j = i; j < n; j++ )   //generate children
    {
     knapsack(v,w,W, sum + W[j], j + 1,sol U {j} )
    }
   }
 record sol-{w[i-1]}
}
print sol with maximum value among all sols.
```

Note:

The algorithm provided in horowitz book quite complex to understand

The above algorithm is simple but not much better than naïve approach.

However, it will do well for examp purpose.

n-queen problem

- Place n queens on a nxn board so that no queen attacks other queen.
- Two queens attack each other if they are on the same row, column, or diagonal.
- Consider a 2D board where each position is determined by (row,column) tuple. Suppose two queens are placed at positions (i,j) and (k,l). Then two queens attack each other if :
  - $i=k$ (same row)
  - $J=l$ (same column)
  - $|i-k| = |j-l|$ (diagonal)
- **Example**: following are the solutions for a 4x4 board (n=4).

	Q_1		
			Q_2
Q_3			
		Q_4	

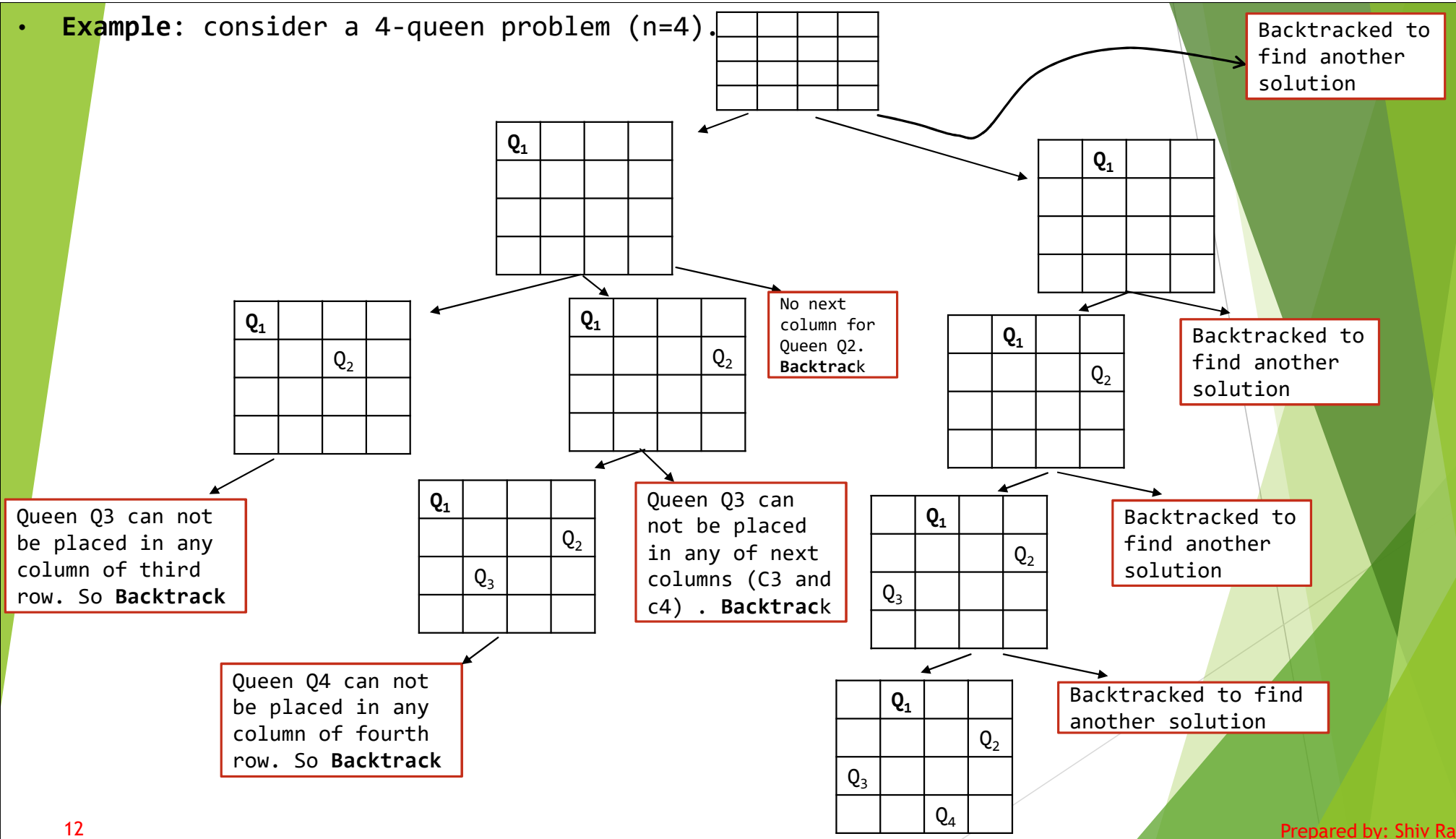
		Q_1	
Q_2			
			Q_3
	Q_4		

How to solve ?

- The problem can be solved by backtracking.
- Idea:

i. For each queen Q_i see if it can be put in any column j of row i.
ii. If yes, then put the queen in jth column of row i proceed to put queen Q_{i+1} in $(i+1)^{th}$ row.
iii. If no, then backtrack and change the column of queen Q_{i-1}.

- **Example:** consider a 4-queen problem (n=4).



- Let $(x_1, x_2, .. x_n)$ represents a solution in which x_i is the column of the i^{th} row where i^{th} queen is placed.

	Q_1		
			Q_2
Q_3			
		Q_4	

		Q_1	
Q_2			
			Q_3
	Q_4		

e.g. in above example, the first solution is (2,4,1,3) and the second solution is (3,1,4,2). Here, the position of a number represents the row (and queen number) and the value represents column number of that row. In (2,4,1,3) , the number 3 represents 4^{th} queen is placed in 3^{rd} column of 4^{th} row.

- Let *place(k,i)* represents a function which returns **true** if the k^{th} queen can be placed in i^{th} column. It tests if a queen to be placed is in same column and diagonal as all previous queens.

Algorithm:

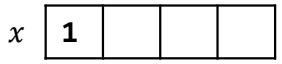
```
Nqueen(k,n)        //initially k=1
{
  for i = 1 to n
  {
    if (place(k,i)=true)
     {
       x[k]= i
       if(k=n)
         Print(x)
         else
          Nqueen(k+1,n)
     }
  }}

place(k,i)
{
  for j = 1 to k-1
  {
    if(x[j]=i) or (|x[j}-i| = |j-k|)
      return false;

   }

 return true;
}
```

Two queens in same column

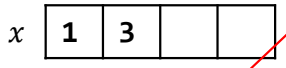
Two queens in same diagonal

Example: trace the algorithm for n=4

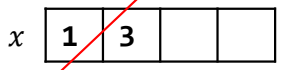
Function call: Nqueen(1,4)
Place(1,1) = true



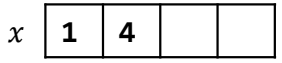
Function call: Nqueen(2,4)
Place(2,1) = false
Place(2,2) = false
Place(2,3) = true



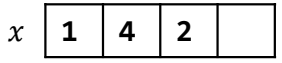
Function call: Nqueen(3,4)
Place(3,1) = false
Place(3,2) = false
Place(3,3) = false
Place(3,4) = false
3rd queen can not be placed in any column of third row.
Backtrack
Return to function Nqueen(2,4)



Function call: Nqueen(2,4)
Place(2,4) = true

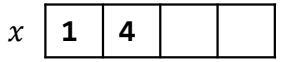


Function call: Nqueen(3,4)
Place(3,1) = false
Place(3,2) = true



Function call: Nqueen(4,4)
Place(4,1) = false
Place(4,2) = false
Place(4,3) = false
Place(4,4) = false

Backtrack



Function call: Nqueen(3,4)
Place(3,3) = false
Place(3,4) = false
Backtrack

Function call: Nqueen(2,4)
Nothing to do
Backtrack to function Nqueen(1,4)

Function call: Nqueen(1,4)
Place(1,2) = true

Continue the process…..

Note: Above trace is just to understand the pseudocode line by line. This is exactly the same process we did in earlier (slide 11) by graphical (tree) representation.

Caution: Do not trace like this in exam. In exam, make the tree as in slide 11

Prepared by: Shiv Raj Pant

Analysis:

- With naïve approach, we need to consider $\binom{n^2}{n}$ possible ways to put n queens
- With Backtracking, we need only n! tuples need to be examined. Because we put queens always in distinct rows and columns. This approach greatly reduces the number of steps of algorithm.

- Example: consider 8-queens problem

Naive approach: $\binom{64}{8}$ = 4.4 billion tuples

Backtracking: 8! = 40,320 tuples

Note: If you have difficulty with the previous pseudocode, then here is the algorithm in high-level structured English.

```
Nqueen(A,k,n)      //A is the nxn solution array. Initially k=1 and A is empty
{
  for i = 1 to n
  {
    if (putting queen Qk in A[k][i] position does not attack any of Q1, Q2, ..Qk-1)
     {
       put queen Qk in (k,i) position
       if(k=n)                                  //all queens are put in the board
         print A and return
        else
           Nqueen(A,k+1,n)                    //process (k+1)th queen recursively
     }
  }
}
```

End of unit 6

Thank You!