دانشگاه کردستان
University of Kurdistan

# Backtracking and Branch & Bound


**Sadoon Azizi**

s.azizi@uok.ac.ir

Department of Computer Engineering and IT


Spring 2019

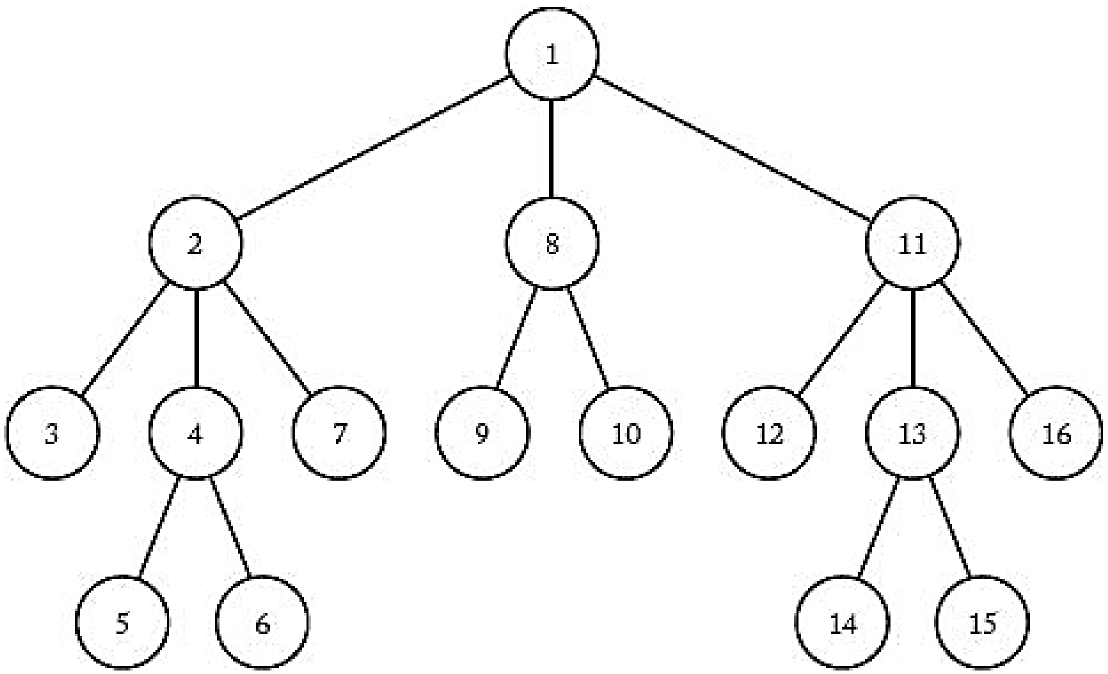
# Techniques for the design of Algorithms

- Divide and Conquer
- Dynamic Programming
- Greedy Algorithms
- **Backtracking Algorithms**
- **Branch and Bound Algorithms**

# Backtracking Algorithms

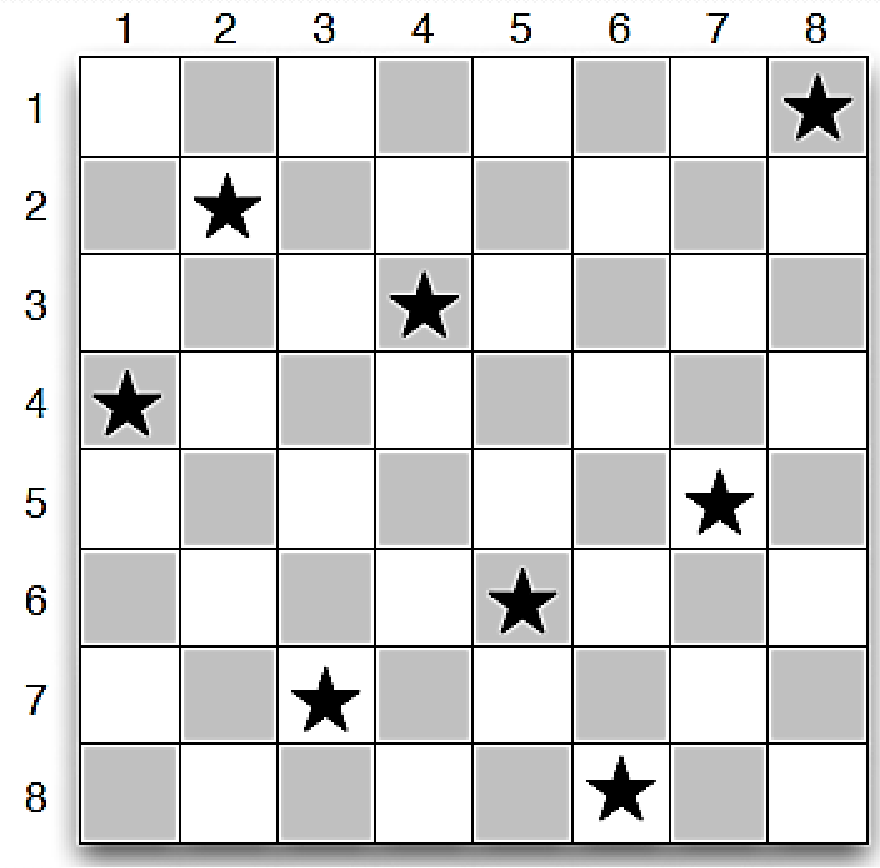
- In general, we assume our solution is a vector **$s=(a_1,a_2,\ldots,a_n)$**.
- At each step, we try to **extend** a partial solution $s_k=(a_1,a_2,\ldots,a_k)$ by adding another element at the end.
- Then we test whether what we now have is a solution: if so, we should print it or count it.
- If not, we check whether the partial solution is still potentially extendible to some complete solution.
- Backtracking algorithm is modeled by a **tree** of partial solutions, where each node represents a partial solution.

# Backtracking Algorithms (space state tree)



# n-Queens Problem

- The *n* Queen is the problem of **placing** *n* chess queens on an n×n chessboard so that no two queens attack each other.



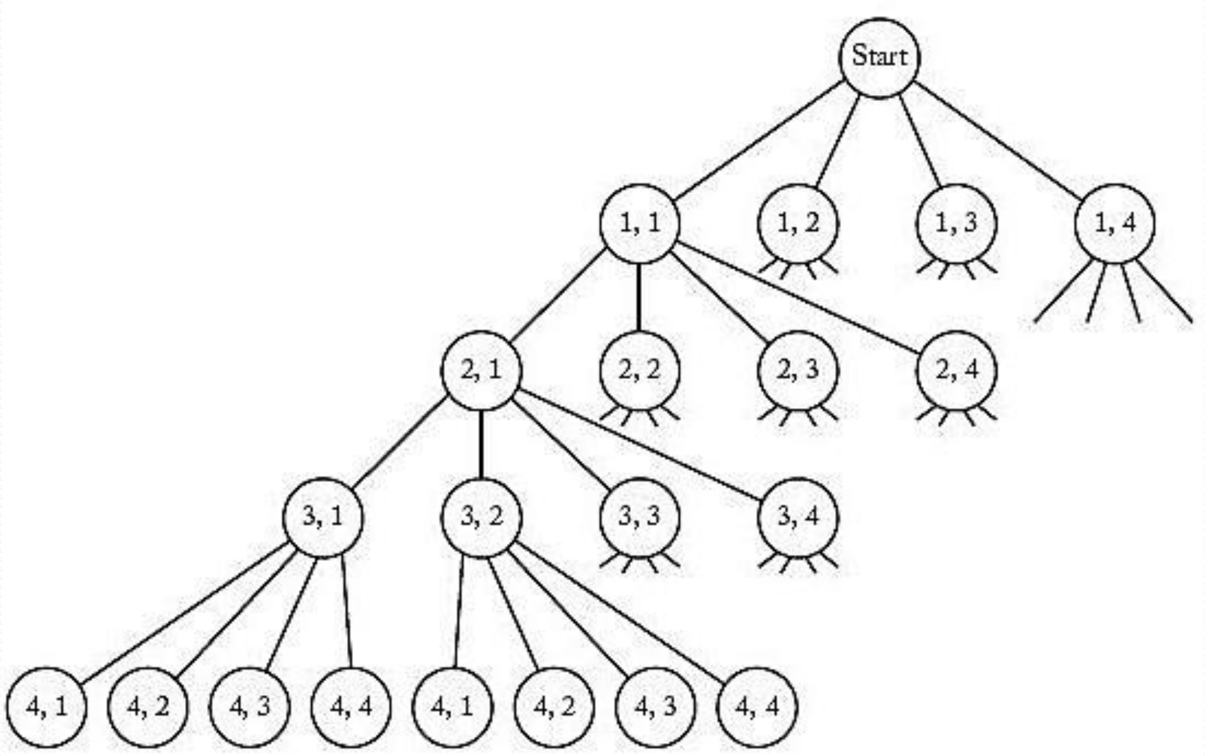
# n-Queens Problem

We can use different approaches:

- Search all the solution space of size: $\binom{n^2}{n}$
- Using eight loops, each is inside the other: $n^n$
- Using 1-dimensional array in order to remove more conflicts and reducing the search space.

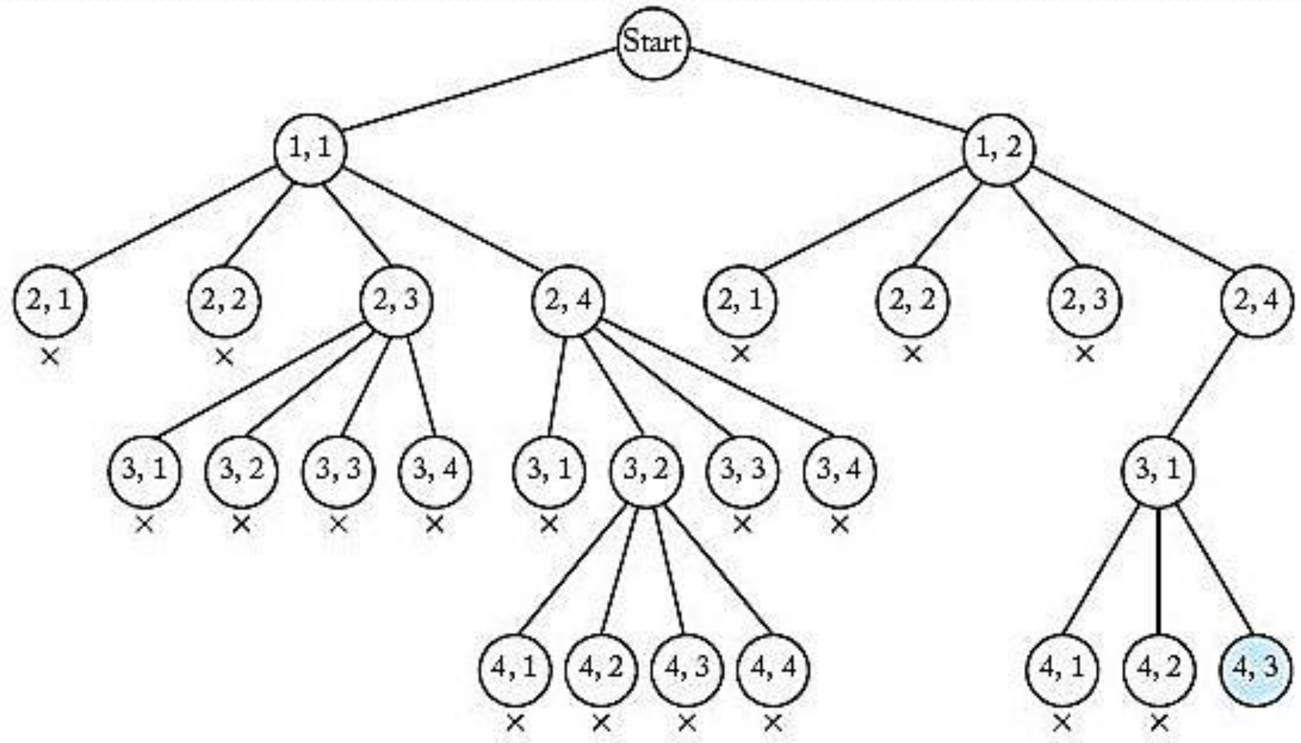
# n-Queens Problem

- A portion of the state space tree for the instance of the *n*-Queens problem in which *n*=4.

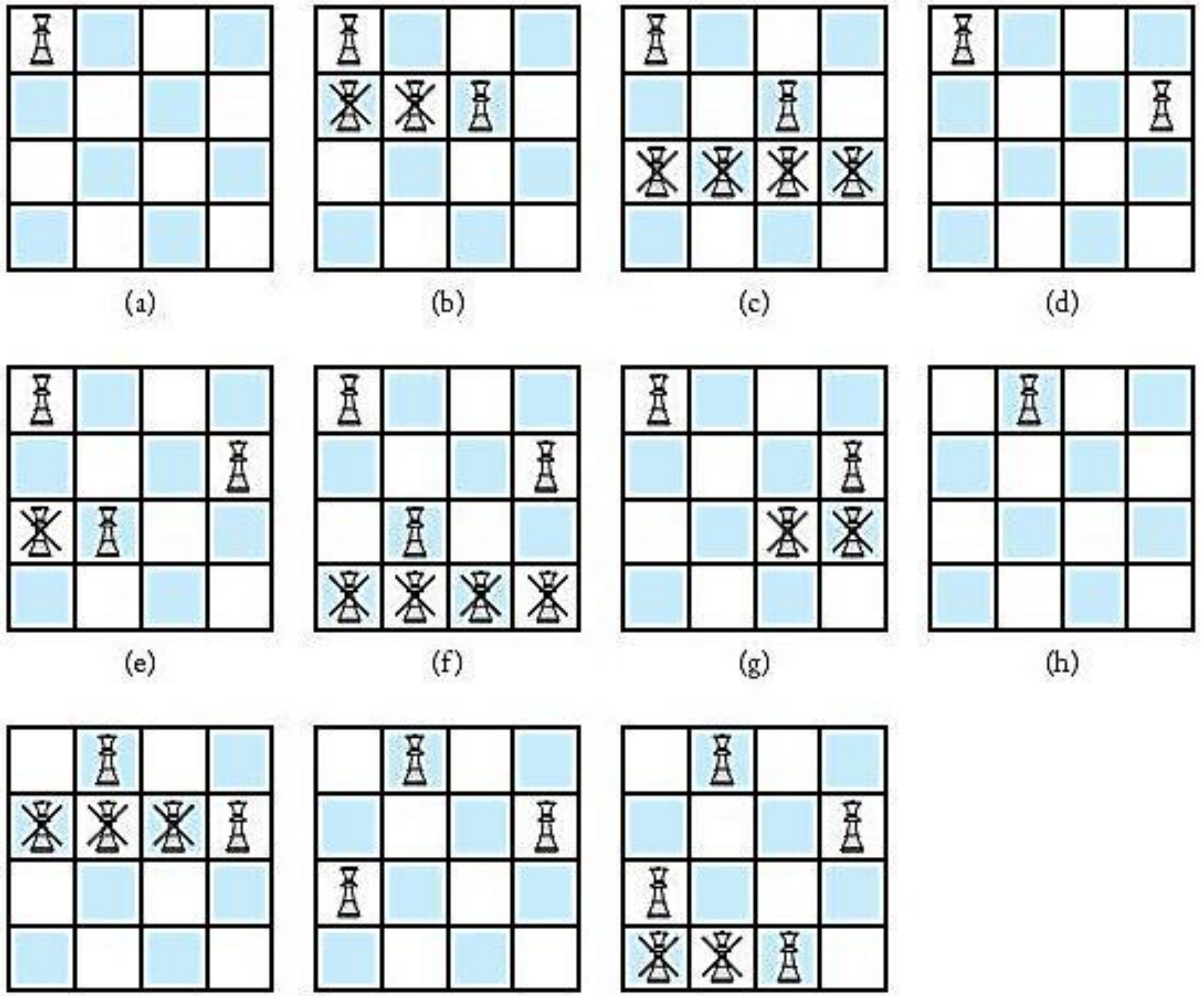


# n-Queens Problem

- A portion of the pruned state space tree produced when backtracking is used to solve the instance of the *n*-Queens problem in which *n*=4.



# n-Queens Problem



# n-Queens Problem

```
void expand(node v)
{
   node u;

   for (each child u of v)
       if (promising(u))
          if (there is a solution at u)
             write the solution;
          else
             expand(u);
}
```

# Knapsack Problem (Review)

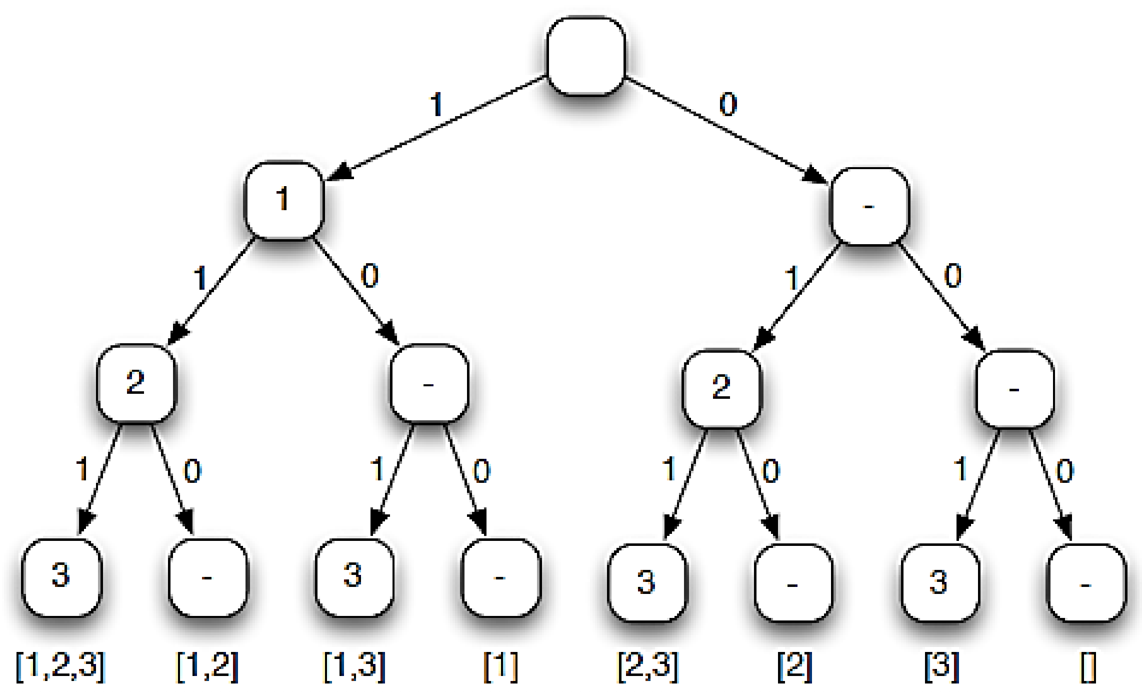
## Definition

Suppose that we have $n$ objects, say $o_i$ ($i = 1, 2, \cdots, n$), each with corresponding weight ($w_i$) and profit ($p_i$), and a weight bound $b$. The goal of this problem is to find an $X = (x_1, x_2, \cdots, x_n)$ that maximize $\sum_{i=1}^{n} x_i p_i$ with respect to $\sum_{i=1}^{n} x_i w_i \leq b$.

- if $x_i \in \{0, 1\}$ the this problem is called 0/1-Knapsack.
- if $x_i \in [0, 1]$ the this problem is called fractional-Knapsack.

# 0/1-Knapsack

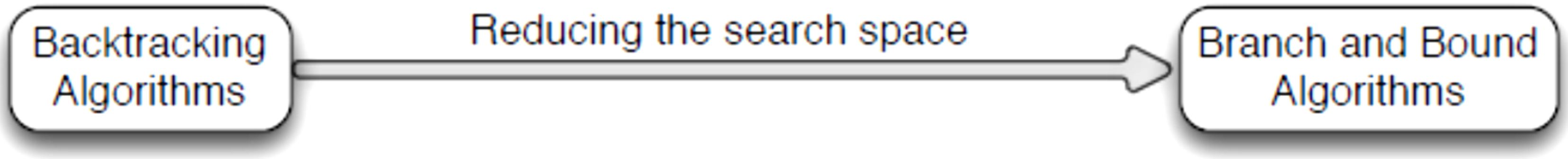
- To design a backtracking algorithm for this problem, we should generate all subsets of $\{0,1\}^n$ and check which one is optimal.



# 0/1-Knapsack (Backtracking Algorithm)

```
Backtrack-Knapsack(X, optX, optP, ℓ){
    if ℓ = n+1 then{
        if ∑_{i=1}^{n} x_i w_i ≤ b then{
            curP ← ∑_{i=1}^{n} x_i p_i;
            if curP ≥ optP then{
                optP ← curP;
                optX ← [x_1, x_2, ··· , x_n];
            }
        }
    }
    else{
        x_l ← 1;
        Backtrack-Knapsack(X, optX, optP, ℓ+1);
        x_l ← 0;
        Backtrack-Knapsack(X, optX, optP, ℓ+1);
    }
}
```

# Branch & Bound Algorithms



# Branch & Bound Algorithms

- **Branch-and-Bound** is based on backtracking, which is an exhaustive searching technique in the space of all feasible solutions.
- The cardinality of the sets of feasible solutions are typically as large as $2^n$, $n!$, or even $n^n$ for inputs of size n.
- The idea of the branch-and-bound technique is to **speed up** backtracking by omitting the search in some parts of the space of feasible solutions, because one is already able to recognize that these parts do not contain any optimal solution in the moment when the exhaustive search would start to search in these parts.
- The branch-and-bound is based on some **pre-computation of a bound** on the cost of an optimal solution (a lower bound for maximization problems and an upper bound for minimization problems).

# 0/1-Knapsack (B&B-Knapsack1)

```
B&B-Knapsack1(X, optX, optP, ℓ, curW){
    if ℓ = n+1 then{
        if ∑_{i=1}^{n} x_i p_i ≥ optP then{
            optP ← ∑_{i=1}^{n} x_i p_i;
            optX ← [x_1, x_2, ··· , x_n];
        }
    }
    else{
        if curW + w_ℓ ≤ b then C_ℓ ← {1,0};
        else C_ℓ ← {0};
    }
    for each x ∈ C_ℓ do {
        x_l ← x;
        Backtrack-Knapsack(X, optX, optP, ℓ+1, curW + x_ℓ w_ℓ);
    }
}
```

# 0/1-Knapsack (B&B-Knapsack2)

B&B-Knapsack2$(X, optX, optP, \ell, curW)\{$

  if $\ell = n+1$ then$\{$

    if $\sum_{i=1}^{n} x_i p_i \geq optP$ then$\{$

      $optP \leftarrow \sum_{i=1}^{n} x_i p_i$;

      $optX \leftarrow [x_1, x_2, \cdots, x_n]$;

    $\}$

  $\}$

  else$\{$

    if $curW + w_\ell \leq b$ then $C_\ell \leftarrow \{1, 0\}$;

    else $C_\ell \leftarrow \{0\}$;

  $\}$

  $B \leftarrow \sum_{i=1}^{l-1} x_i p_i + GFK(p_\ell, p_{\ell+1}, \cdots, p_n, w_\ell, w_{\ell+1}, \cdots, w_n, b - curW)$;

  if $B \leq optP$ then return;

  for each $x \in C_\ell$ do $\{$

    $x_l \leftarrow x$;

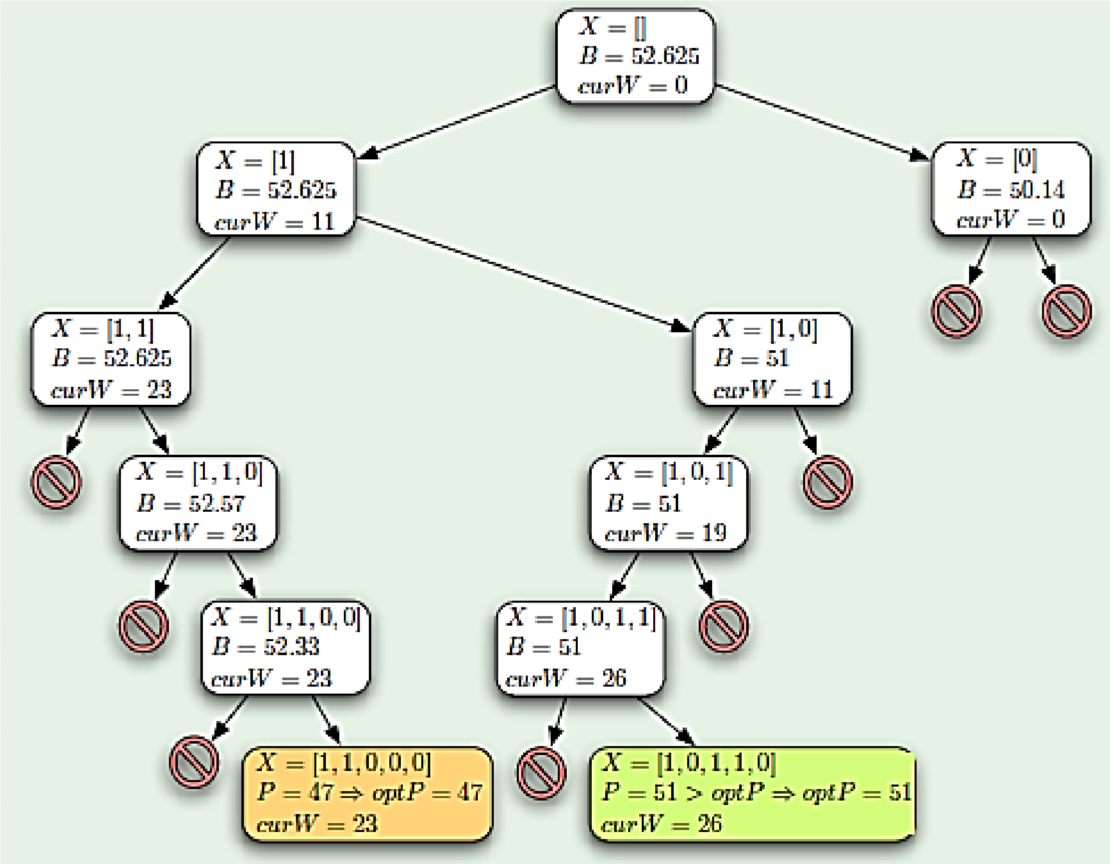
    Backtrack-Knapsack$(X, optX, optP, \ell + 1, curW + x_\ell w_\ell)$;

  $\}$

$\}$

# 0/1-Knapsack (Branch&Bound Algorithm-2)

- **Example:** Suppose that P=[23,24,15,13,16], W=[11,12,8,7,9], and b = 26. The algorithm B&B-Knapsack2 works as follows:



# Comparison of the algorithms

- The following table represents the worse case size of search space of random instances executed for 5 times.

| *n* | Backtrack-Knapsack | B&B-Knapsack1 | B&B-Knapsack2 |
|---|---|---|---|
| 8 | 511 | 333 | 78 |
| 12 | 8191 | 4988 | 195 |
| 16 | 131071 | 78716 | 601 |
| 20 | 2097151 | 1257745 | 480 |
| 24 | 33554431 | 19814875 | 755 |