# Unit 7.1 Backtracking

Algorithms

EE3980

May 21, 2020

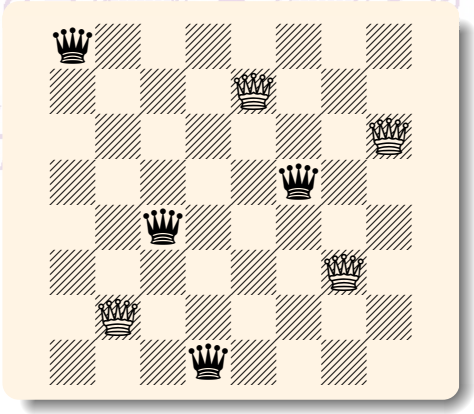


## Backtracking Algorithms

- The backtracking algorithms are to deal with problems to generate a desired solution expressible as an $n$-tuple, $(x_1, x_2, \cdots, x_n)$, where $x_i$ are chosen from a finite set $S_i$, and the solution satisfies or minimizes/maximizes a criterion function $P(x_1, x_2, \cdots, x_n)$.
- Suppose $m_i$ is the size of $S_i$. Then, there are $m = m_1 \times m_2 \times \cdots \times m_n$ possible candidates for satisfying the function $P$.
- The brute force approach is to form all $m$ candidates and evaluate criterion function on each of them, selects all (or the optimal) solutions.
- The backtracking method forms the $n$-tuple one component at a time, and then use the modified criterion function $P_i(x_1, x_2, \cdots, x_i)$ to see if the current vector can meet the overall criterion. If it cannot, the current vector is ignored immediately.
  - The number of tries is substantially smaller with backtracking methods.

# The 8-Queens Problem

- A queen in a chess game can attack any other piece if
  - It is in the same row
  - It is in the same column
  - It is in the same diagonal (two directions)
- The 8-queens puzzle is to place 8 queens on a chessboard such that they don't attack each other.





# The N-Queens Problem — Algorithms

- The problem is generalized to placing $N$-queens onto an $N \times N$ board.
- Note that each row can have only one queen.
  - Thus, one can use an array $Q[1:N]$ for column position for each queen.
- For 8-Queens puzzle this reduces the number of possible checks from $8^8$ (16,777,216), to $8!$ (40,320), 0.24%.

## Algorithm 7.1.1. N-Queen puzzle

```
   // Place k'th queen onward, if successful print out solution.
   // Input: Q[1 : n], k, n
   // Output: Solutions for placing n-Queens.
 1 Algorithm NQueens(k, n)
 2 {
 3       for i := 1 to n do { // all possible positions for Q[k]
 4           if Placeable(k, i) then { // placing Q[k] = i is legitimate
 5               Q[k] := i;
 6               if (k = n) then write (Q[1 : n]); // a solution found.
 7               else NQueens(k + 1, n); // place Q[k + 1]
 8           }
 9       }
10 }
```

# The N-Queens Problem — Algorithms, II

- The `Placeable` algorithm is shown below.

## Algorithm 7.1.2. Placeable

```
   // Test if it is legitimate to place a Queen at (k, i).
   // Input: Q[1 : k], i
   // Output: 1: if OK to place, 0: otherwise.
 1 Algorithm Placeable(k, i)
 2 {
 3       for j := 1 to k − 1 do { // check against Queens placed.
 4            if (Q[j] = i or |Q[j] − i| = |j − k|) then return false ;
 5       }
 6       return true ;
 7 }
```

- Note that if a queen is not placeable to $(k, i)$ then rows $k$ onwards are not checked, thus reducing even more checkings.
- An iterative version, `NQueens_I`, is given next
  - Same time complexity as the recursive version
  - The number of try-outs is identical in either case
  - Smaller heap space for function calls for iterative version
- Both `NQueens` and `NQueens_I` algorithms can still be improved.

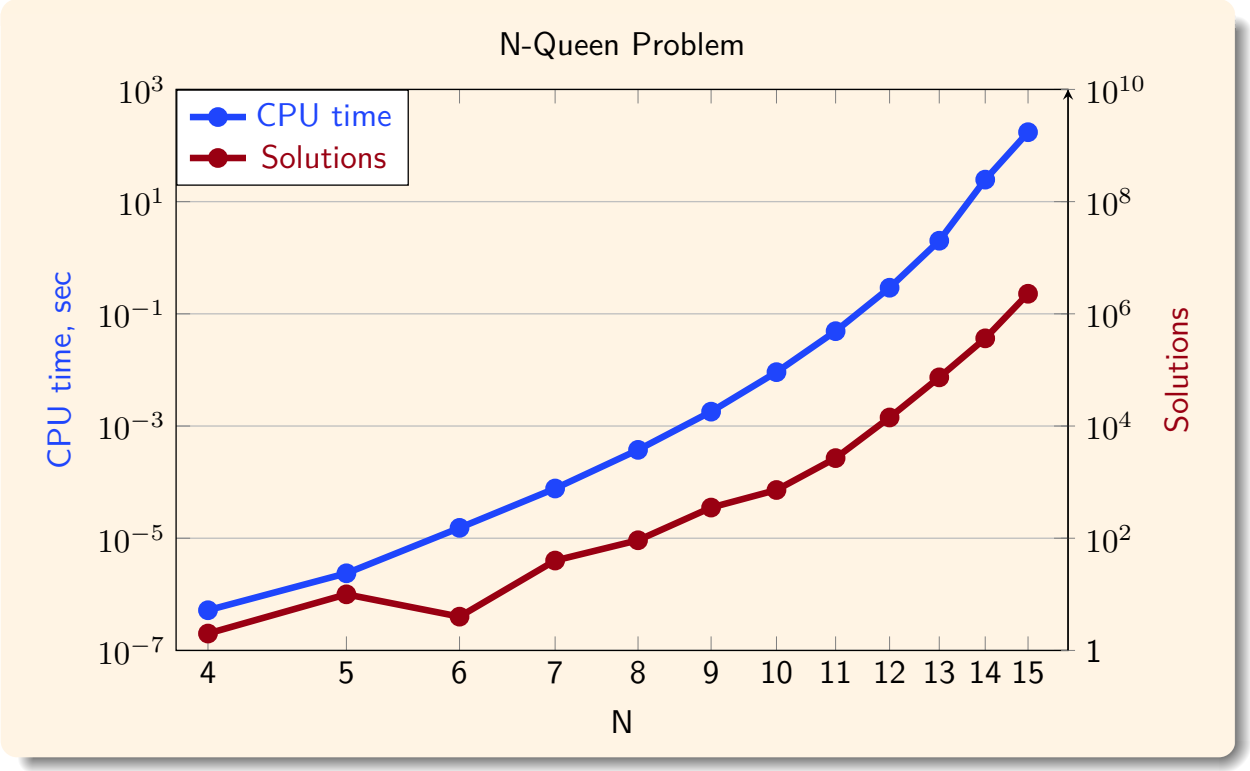


# The N-Queens Problem — Iterative Algorithms

## Algorithm 7.1.3. N-Queen puzzle, iterative solution

```
   // Find all solutions for N-Queen problem iteratively.
   // Input: number of Queens: n
   // Output: Solutions for placing n-Queens.
 1 Algorithm NQueens_I(n)
 2 {
 3       k := 1 ;
 4       Q[k] := 0 ;
 5       while (k > 0) do {
 6            Q[k] := Q[k] + 1 ;
 7            while (Q[k] ≤ n) do {
 8                 if Placeable(k, Q[k]) then {
 9                      if (k = n) then write (Q[1 : n]) ; // a solution is found
10                      else { // try next row and initialize
11                           k := k + 1 ;
12                           Q[k] := 0 ;
13                      }
14                 }
15                 Q[k] := Q[k] + 1 ;
16            }
17            k := k − 1 ; // done with this row, backtrack to previous row
18       }
19 }
```

# The N-Queen Problem – Solutions




- Both CPU time and number of solution appear to increase exponentially with N.
- The complexity of the algorithms, recursive and iterative, are $\mathcal{O}(N!)$.



# Sum of Subsets Problem

- Given a set of $n$ distinct positive numbers, $\{w_i, 1 \leq i \leq n\}$ and $m$, $m > 0$, the sum of subsets problem is to find all the combinations of those $n$ numbers whose sum is $m$.
- Example, given the set $\{4, 11, 15, 24\}$ and $m = 15$.
  - Two subsets, $\{4, 11\}$ and $\{15\}$, have the sum equals to 15.
- It is assume that the set is ordered in nondecreasing order, $w_i \leq w_{i+1}, 1 \leq i < n$, and

$$w_1 \leq m, \tag{7.1.1}$$

$$\sum_{i=1}^{n} w_i \geq m. \tag{7.1.2}$$

  otherwise, there is no solution possible.
- Let $\{x_i | x_i = 0 \text{ or } x_i = 1, 1 \leq i \leq n\}$ be the solution, then

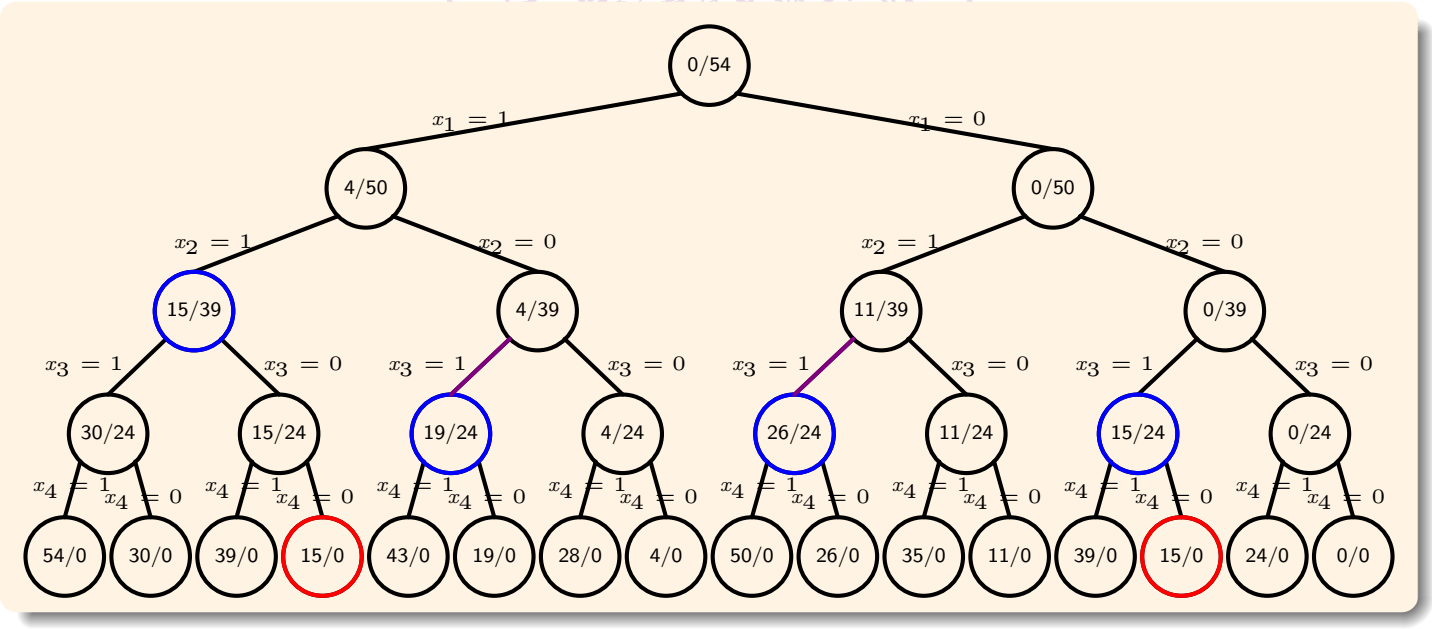
$$\sum_{i=1}^{n} x_i w_i = m. \tag{7.1.3}$$

- To find the solution, all combinations are to be tested.
  - Backtracking approach can be applied.

# Sum of Subsets, Example

- Example, $w = \{4, 11, 15, 24\}$, $m = 15$
- Two numbers shown in each node $s/r$

$$s = \sum_{i=1}^{k} x_k w_k \qquad r = \sum_{i=k+1}^{n} w_k$$






# Sum of Subsets, Algorithm

- A recursive algorithm to find all the solutions.

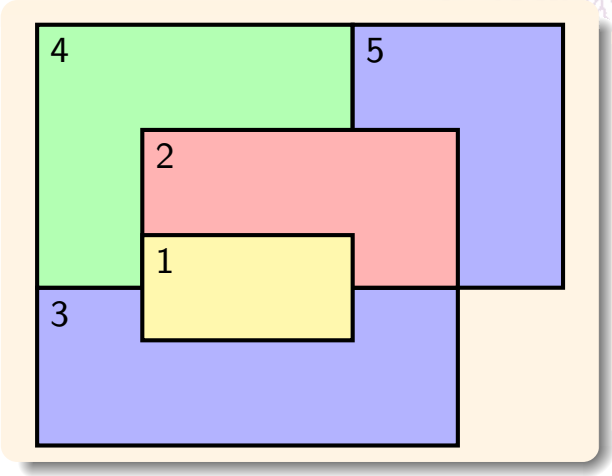
## Algorithm 7.1.4. Sum of Subsets

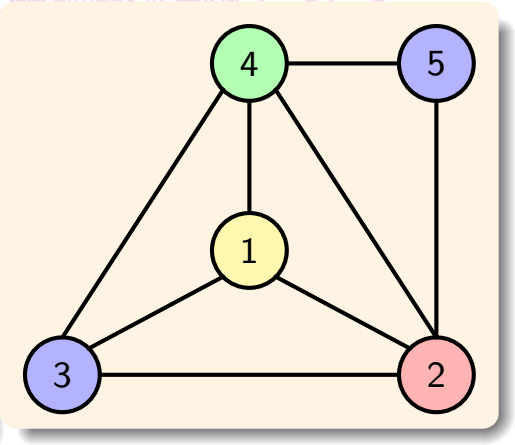
```
   // To test if x[k] = 1 for sum of subset problem.
   // Input: s = Σ_{i=1}^{k-1} w[i]x[i], r = Σ_{i=k}^{n} w[i], k, w[1 : n]
   // Output: x[1 : n].
 1 Algorithm SumOfSub(s, k, r)
 2 {
 3       x[k] := 1 ; // try to include w[i]
 4       if (s + w[k] = m) then write (x[1 : k]) ; // one solution found
 5       else if (s + w[k] + w[k + 1] ≤ m) then
 6             SumOfSub(s + w[k], k + 1, r − w[k]) ;
 7       if ((s + r − w[k] ≥ m) and (s + w[k + 1] ≤ m)) then { // x[i] = 0 case
 8             x[k] := 0 ;
 9             SumOfSub(s, k + 1, r − w[k]) ;
10       }
11 }
```

- The definition of $s$ is different from the preceding page.
- Note the termination condition of this algorithm
- With proper checking, lines 5 and 7, number of unsuccessful search is significantly reduced, but the complexity remains to be $\mathcal{O}(2^n)$.

# Graph Coloring

- Given a map with $n$ regions, the $m$-colorability decision problem is to find if one can assign $m$ colors to the map such that each region has a color and no two adjacent regions have the same color.
- Note that the map with $n$ regions can be transformed into a graph.
  - Each region is represented by a node,
  - Adjacent regions are connected by an edge between the nodes.
- The adjacency relationship can also be represented by the adjacency matrix.




$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$



# Graph Coloring, Algorithm

- The following algorithm solves for the $m$-coloring problem for a graph, $G$, with $n$ vertices and adjacency matrix $A$.
  - Global Array $x$ is the solution found, $x[i]$ is the color for vertex $i$.
- The algorithm should be invoked by
  `mColoring`$(n, m, 1)$;

## Algorithm 7.1.5. $m$-Color Algorithm

```
// Recursively assign all possible, at most m, colors to node k.
// Input: int n, m, k, adjacent matrix A
// Output: All acceptable solutions.
1  Algorithm mColoring(n, m, k)
2  {
3       for x[k] := 1 to m do {
4            i := 1 ; // check for colored and adjacent nodes with the same color
5            while (i < k and ((A[i, k] = 0) or (x[i] ≠ x[k]))) do i := i + 1 ;
6            if (i = k) then { // color acceptable
7                 if (k = n) then write (x[1 : n]) ; // a solution is found
8                 else mColoring(n, m, k + 1);
9            }
10      }
11 }
```

# Graph Coloring, Complexity

- In algorithm `mColoring`, Algorithm (7.1.5), the `for` loop, lines 3-10, is executed $m$ times at each recursive call
  - And `mColoring` is called recursively for $n$ times
- Again in algorithm `mColoring`, the `while` loop, line 5 is executed at most $n$ times
- Thus the total time complexity is $\mathcal{O}(nm^n)$
- An alternative algorithm is shown on the next page.
  - The color to be tested for node $k$ is reduced – no repeated colors.
  - But the complexity is $\mathcal{O}(m^n)$.

- Note that given a graph $G$ with degree $d$, then $G$ can be colored using $d+1$ colors.
- The smallest $m$ that can color a graph $G$ is also called the chromatic number of $G$.
- Note that $m \leq d+1$ and $m$ can be found by using the Algorithm `mColoring` using different $m$.



# Graph Coloring, An Alternative Algorithm

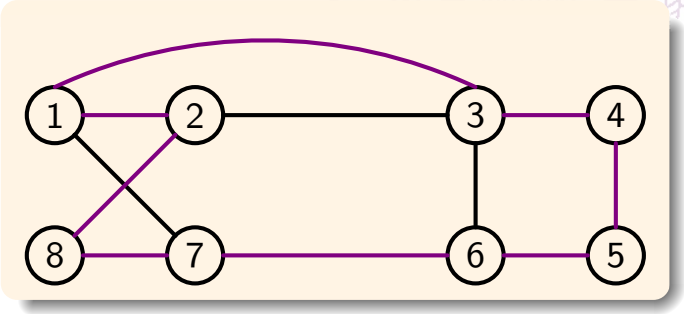
- An alternative algorithm for the $m$-coloring problem.

## Algorithm 7.1.6. $m$-Color Algorithm – Alternative

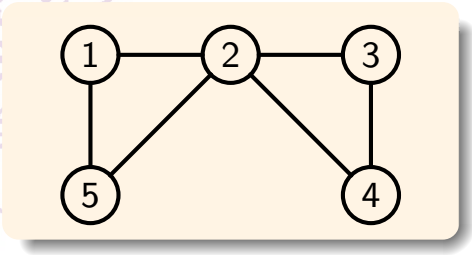
```
   // Recursively assign all possible, at most m, colors to node k.
   // Input: int n, m, k, adjacent matrix A
   // Output: All acceptable solutions.
 1 Algorithm mColoring_A(n, m, k)
 2 {
 3       for i := 1 to m do c[i] := 1; // Let all colors be available.
 4       for i := 1 to k − 1 do
 5           if A[i, k] = 1 then c[x[i]] := 0; // Color used by adj. nodes.
 6       for i := 1 to m do {
 7           if c[i] = 1 then { // Use all available colors.
 8               x[k] := i;
 9               if (k = n) then write (x[1 : n]); // A solution is found
10               else mColoring_A(n, m, k + 1);
11           }
12       }
13 }
```

# Hamiltonian Cycles

- Let $G = (V, E)$ be a connected graph with $n$ vertices. A Hamiltonian cycle is a closed path along $n$ edges of $G$ that visits every vertex once and returning to its starting position.
  - If a Hamiltonian cycle begin at a vertex $v_1 \in V$ and the vertices are visited in the order $(v_1, v_2, \cdots, v_{n+1})$, then the edge $(v_i, v_{i+1}) \in E$, $1 \leq i \leq n$, and the $v_i$ are distinct except $v_1 = v_{n+1}$.




$G_1$ with Hamiltonian cycles.




$G_2$ No Hamiltonian cycle.



# Hamiltonian Cycles — Algorithm

## Algorithm 7.1.7. Hamiltonian Cycle

```
// Recursively find the k-th vertex of a Hamiltonian cycle.
// Input: graph G(V, E), int n, k
// Output: All possible Hamiltonian cycles.
1  Algorithm Hamiltonian(n, k)
2  {
3      for x[k] := 1 to n do { // All possible vertices.
4          if (E[x[k − 1], x[k]] = 1) then { // Connecting to x[k − 1].
5              i := 1;
6              while ((i < k) and (x[i] ≠ x[k])) i := i + 1; // Check if x[k] distinct
7              if (i = k) then // x[k] has not been used
8                  if (k < n) Hamiltonian(n, k + 1); // Move to the next vertex
9                  else {
10                     if (E[k, 1] = 1) then write (x[1 : n]); // Print solution
11                 }
12         }
13     }
14 }
```

# Hamiltonian Cycles — Algorithm, II

- Backtracking approach to solve the Hamiltonian cycle problem.
- $x[1:n]$ is the solution vector.
- $E[1:n, 1:n]$ is the adjacency matrix
  - $E[i,j] = 1$ if $(i,j) \in E$ is an edge in $G$
  - Otherwise, $E[i,j] = 0$.
- `Hamiltonian` should be invoked by
  `Hamiltonian`$(n, 2)$;
  with $x[1] = 1$.
- Thus, this algorithm always find the Hamiltonian cycle starting from vertex 1.
- Note that the depth of the recursive call is $n$
  - The maximum number of `Hamiltonian` recursive call at level $k$ is $n - k$ since each vertex on the path must be distinct
  - Thus, the number of function call is bounded above by $(n-1)!$
- The `while` loop of line 6 is executed at most $n$ times
- The worst case time complexity of `Hamiltonian` algorithm is $\mathcal{O}(n!)$
  - Due to the sparsity of the adjacency matrix, this algorithm has much lower complexity in practice.



# 0/1 Knapsack Problem

- Given $n$ objects, each with profit $p_i$ and weight $w_i$, $1 \le i \le n$, to be placed into a sack that can hold maximum of $m$ weight. However, there is an additional constraint that each object must be placed as a whole into the sack, or not at all. That is, find $x_i$, $1 \le i \le n$, such that

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} p_i x_i, \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \le m, \\ \text{and} \quad & x_i = 0 \text{ or } 1, \quad 1 \le i \le n. \end{aligned} \tag{7.1.4}$$

- Note that $x_i = 0$ or 1 and the solution space can be expanded as a tree.
- The solution can be found by traversing the tree.
- In the following, we assume the objects are ordered as

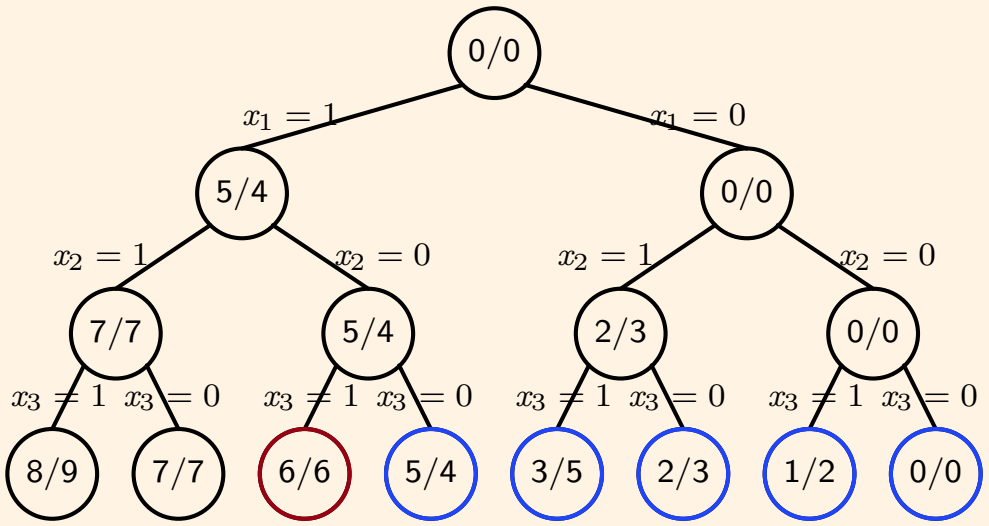
$$\frac{p_1}{w_1} \ge \frac{p_2}{w_2} \ge \cdots \ge \frac{p_n}{w_n}. \tag{7.1.5}$$

And, $fp$ is the final profit, $fw$ is the final weight. Both are global variables.

# 0/1 Knapsack Problem — Search Space

- Given 3 objects, $(p_1, p_2, p_3) = (5, 2, 1)$, $(w_1, w_2, w_3) = (4, 3, 2)$, and $m = 6$. Find the optimal 0/1 knapsack solution, $(x_1, x_2, x_3)$, $x_i = 0$ or $x_i = 1$, $1 \le i \le 3$, that maximizes the profit.
- Two numbers shown in each node $cp/cw$

$$cp = \sum_{i=1}^{k} x_k p_k \qquad cw = \sum_{i=1}^{k} x_k w_k$$




Red: optimal solution; Blue: feasible solutions



# 0/1 Knapsack Problem — Algorithm

## Algorithm 7.1.8. 0/1 Knapsack

```
   // Find solution of 0/1 knapsack problem.
   // Input: int k, n, cp/cw/cx: current profit/weight/sol
   // Output: Solution x[1 : n].
 1 Algorithm BKnap(k, cp, cw)
 2 {
 3       if (cw + w[k] ≤ m) then { // Add k-th object.
 4             cx[k] := 1;
 5             if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]); // Check next.
 6             else if ((cp + p[k] > fp) and (k = n)) then { // Record solution
 7                   fp := cp + p[k];
 8                   fw := cw + w[k];
 9                   for i := 1 to n do x[i] := cx[i];
10             }
11       }
12       if ( Bound(cp, cw, k) ≥ fp) { // Continue traversing only if needs to.
13             cx[k] := 0; // Not placing k-th object.
14             if (k < n) then BKnap(k + 1, cp, cw); // Check next object.
15             else if ((cp > fp) and (k = n)) then { // Record solution.
16                   fp := cp;
17                   fw := cw;
18                   for i := 1 to n do x[i] := cx[i];
19             }
20       }
21 }
```

# 0/1 Knapsack Problem — Bound Algorithm

- Due to Eq. (7.1.5), Bound function can estimate the maximum profit quickly.

## Algorithm 7.1.9. Bounding function

```
    // Estimate maximum profit for k+1 to n objects.
    // Input: int k, n, cp/cw: current profit/weight
    // Output: maximum profit mp.
 1 Algorithm Bound(cp, cw, k)
 2 {
 3       mp := cp; // Init to current values.
 4       mw := cw;
 5       for i := k+1 to n do { // Evaluate all possible.
 6           mw := mw + w[i]; // Update maximum weight.
 7           if (mw < m) then mp := mp + p[i]; // Within limit.
 8           else return mp + (1 - (mw - m)/w[i]) * p[i]; // Exceeding limit.
 9       }
10       return mp;
11 }
```

- Note that Bound function returns a floating number instead of an integer.



# 0/1 Knapsack Problem — Example

- Given 3 objects, $(p_1, p_2, p_3) = (5, 2, 1)$, $(w_1, w_2, w_3) = (4, 3, 2)$, and $m = 6$. Find the optimal 0/1 knapsack solution, $(x_1, x_2, x_3)$, $x_i = 0$ or $x_i = 1$, $1 \le i \le 3$, that maximizes the profit.
- The calling sequence of BKnap algorithm

BKnap$(k = 1, cp = 0, cw = 0)$
- test $cx[1] = 1$, $cw + w[1] \le m$
- BKnap$(k = 2, cp = 5, cw = 4)$
  - test $cx[2] = 1$, $cw + w[2] > m$
  - test $cx[2] = 0$, Bound$= 6$
  - BKnap$(k = 3, cp = 5, cw = 4)$
    - test $cx[3] = 1$, $cw + w[3] = m$, feasible solution: $fp = 6$, $x = (1, 0, 1)$
    - test $cx[3] = 0$, Bound$= 5$, terminates
- test $cx[1] = 0$, Bound$= 3$, terminates

- Function Bound helps to reduce the number of evaluations

# Summary

- Backtracking algorithm
- 8-queens problem
- Sum of subsets problem
- Graph coloring problem
- Hamiltonian cycles
- 0/1 knapsack problem