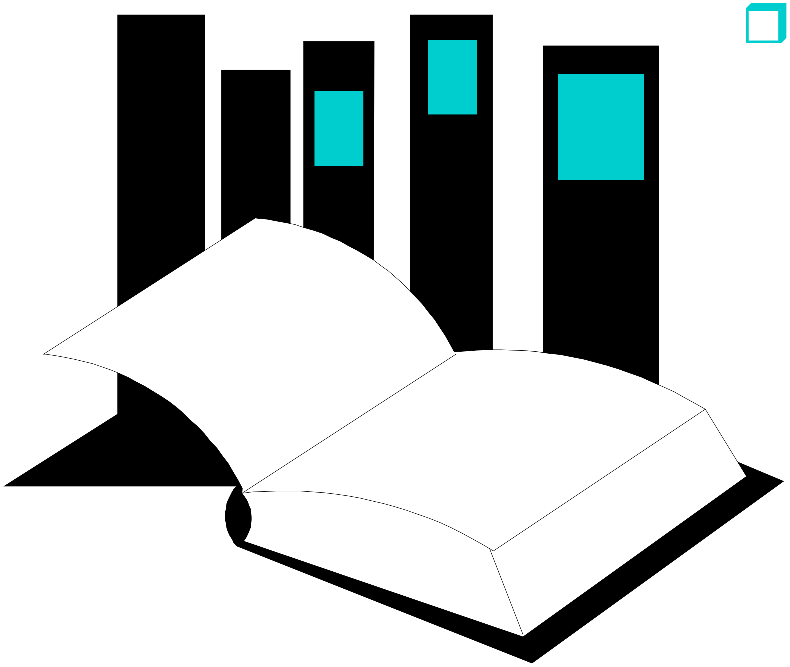
Chapter 5. Backtracking



CHAPTER 5

Foundations of Algorithms

- Chapter 5 introduces an algorithm design technique called "***Backtracking***".

5.1 The Backtracking Technique

- ***Backtracking***

  \- used to solve problems in which a ***sequence*** of objects is chosen from a specified ***set*** so that the sequence satisfies some ***criterion.***

  \- after ***each choice*** has been made and added to a partial solution, it ***can be retracted*** from the solution set later by backtracking.

5.1 The Backtracking Technique

- ***Example: N queens problem***

  \- problem of placing N queens on an N×N chessboard so that no two queens threaten each other

  sequence:

  n positions where the queens are placed

  set:

  n^2 possible positions on the chessboard

  criterion:

  no two queens threaten each other

5.1 The Backtracking Technique

- ***4 queens problem***

  - Number of all possible configurations

  C(16,4) = 1820

  - Since no two queens can be in the same row, we can eliminate ***some*** possibilities:

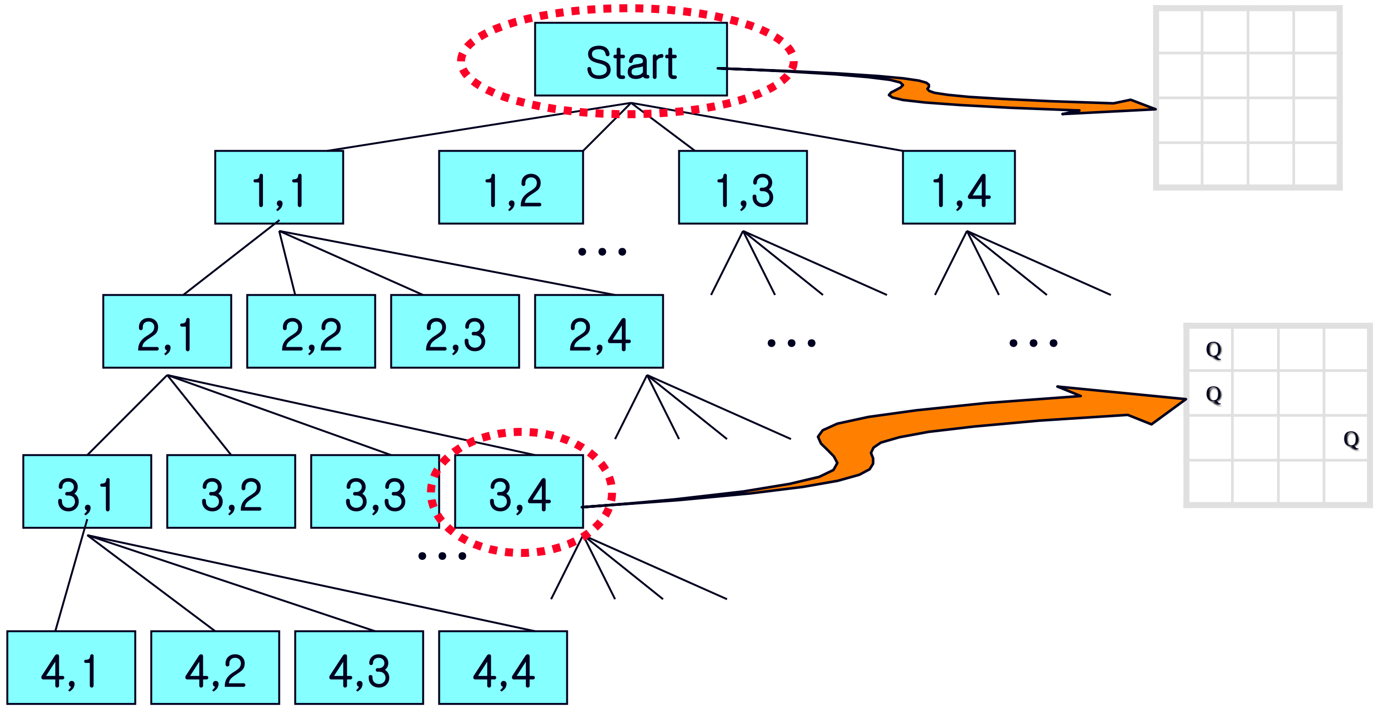
  $$\mathbf{4 \times 4 \times 4 \times 4 = 256}$$

  - State representation <i,j>:

  Queen in the i-th row is in the j-th column

5.1 The Backtracking Technique

- ***4 queens problem - the State Space Tree***



5.1 The Backtracking Technique

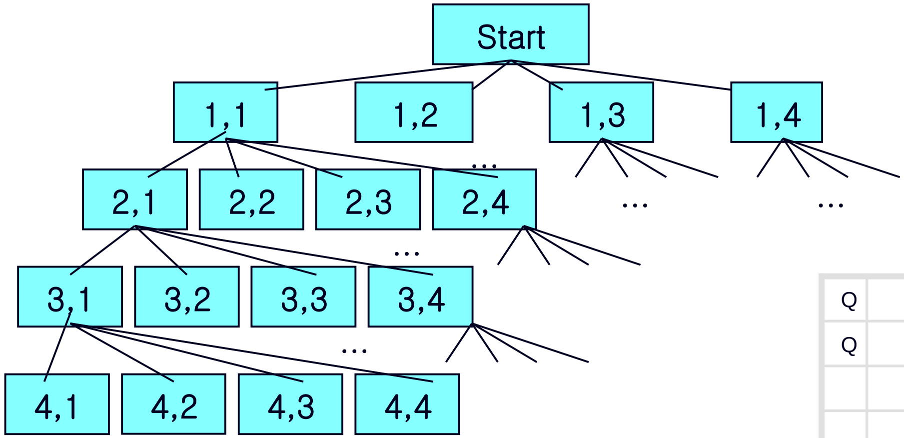
4 queens problem - the State Space Tree

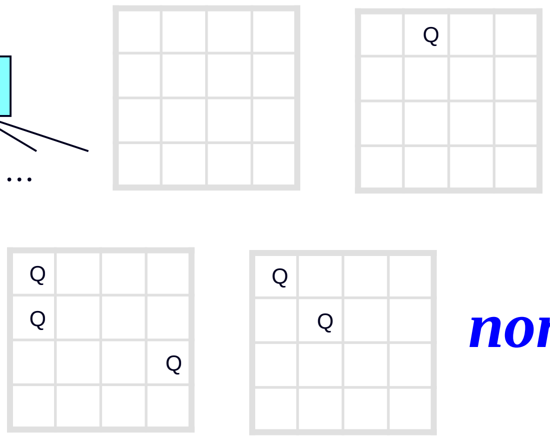
NonPromising **Node:**

- a node that cannot possibly lead to a solution

Promising **Node:**

- a node that is ***not nonpromising***





promising

nonpromising

5.2 The N-queens problem

- ***N-queens problem***

  Backtracking

  - do a ***depth-first search*** of a state space tree checking whether each node is promising

  - if ***nonpromising, backtrack*** to the node's parent and try other path

  ➔ Backtracking can be implemented by a recursive depth-first search algorithm.

5.2 The N-queens problem

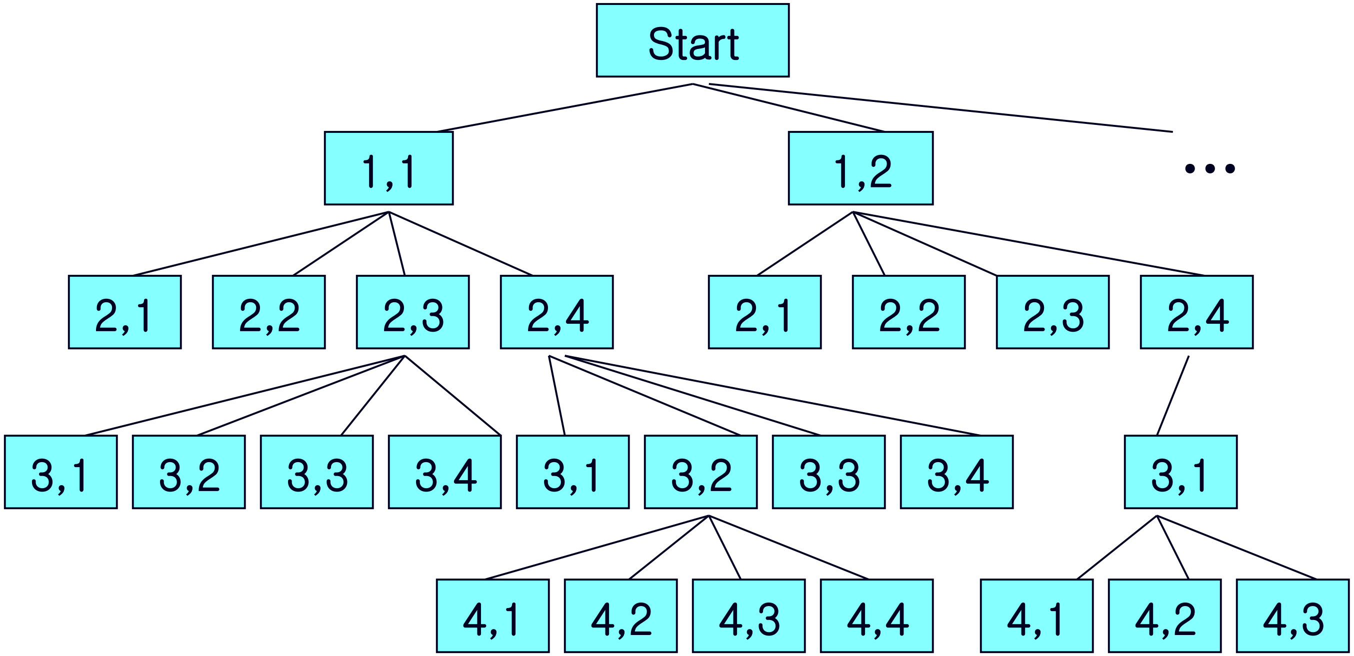
- ***N-queens problem***

```
public static void checknode(node v)
{
        node u;

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

5.2 The N-queens problem

- ***N-queens problem***



A portion of the *pruned* state space tree

5.2 The N-queens problem

❑ *N-queens problem*

- How to check whether two queens are in the same column or diagonal:

➔ Let **Col(i)** be the column where the queen in the i-th row is located.

1. To check whether two queens are in the ***same column***:

   check whether ***Col(i) = Col(k)***

2. To check whether two queens are in the ***same diagonal***:

   check whether Col(i) – Col(k) = i-k or

   Col(i) – Col(k) = k-i

   that is, ***|Col(i) – Col(k)| = i-k*** for i >k

5.2 The N-queens problem

N-queens problem

```
public static void queens( index i) {
    index j;

    if (promising(i))
        if (i==n)
            system.out.print ( col[1] .. col[n] )
        else
            for (j=1; j<=n; j++) {
                col[i+1] = j;
                queens(i+1) ;
            }
}
```

➔ n and Col[1..n] are globally defined

➔ Top-level call

queens(0) ;

5.2 The N-queens problem

- ***N-queens problem***

```
public static boolean promising (index i)
{
    index k;   boolean switch ;

    k = 1;
    switch = true ;
    while (k<i && switch) {
        if (col[i]==col[k]) || abs(col[i]-col[k]) == i-k)
            switch = false ;
        k++;
    }
    return switch ;
}
```

5.2 The N-queens problem

- ***N-queens problem***

  - It is hard to analyze this algorithm because we have to determine the number of nodes checked as a function of n.

  ***Maximum Upper* Bound:**

  $$1 + n + n^2 + n^3 + \dots + n^n = (n^{n+1} - 1)/(n-1)$$

  Upper Bound on number of *promising* nodes:

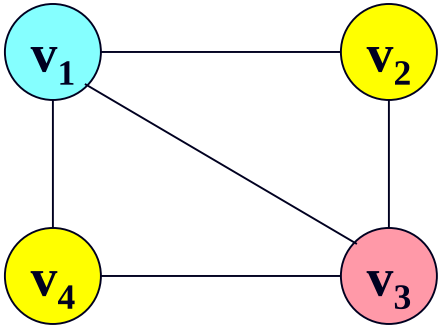
  $$1 + n + n(n-1) + n(n-1)(n-2) + \dots + n!$$

  ➔ May have to actually run the algorithm on a computer and count how many nodes are checked.

5.5 Graph Coloring

The m-coloring problem

- problem of finding all ways to color an undirected graph using ***at most m different colors***, so that ***no two adjacent vertices are the same color***

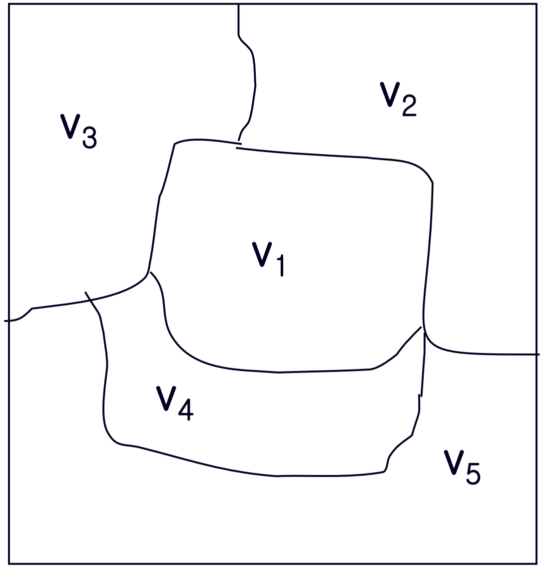


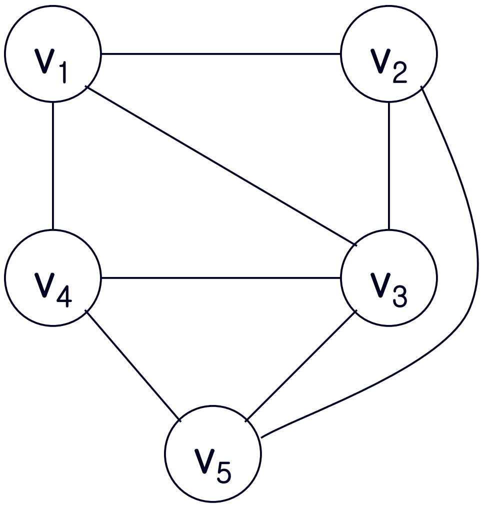
→ can be applied to Coloring of Maps

5.5 Graph Coloring

- ***The m-coloring problem***

  Every map can be converted to a ***planar*** graph, where no two edges cross each other.

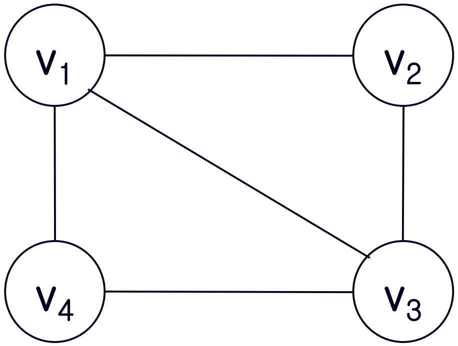


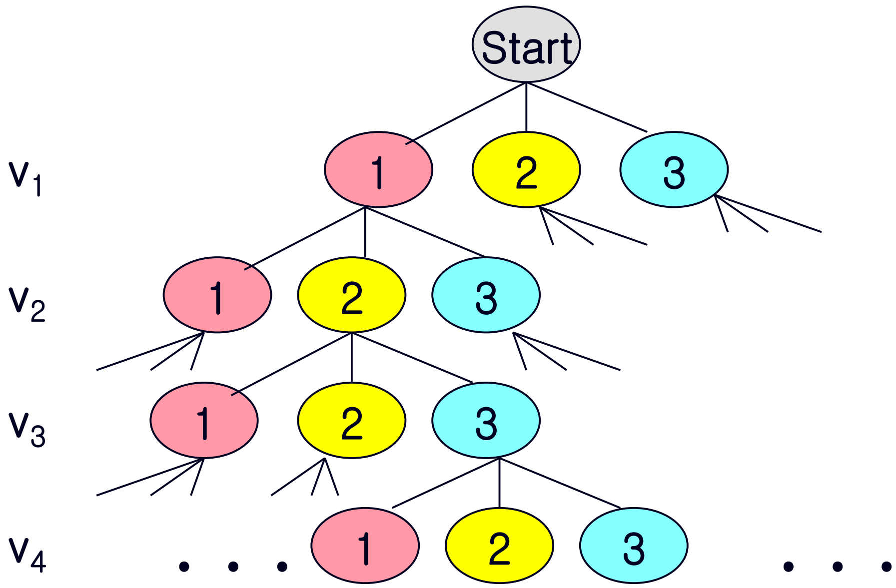


5.5 Graph Coloring

❑ *The m-coloring problem*

Example:



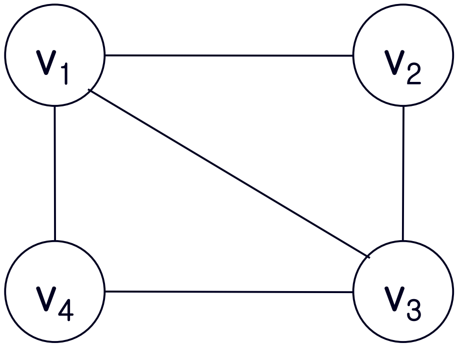


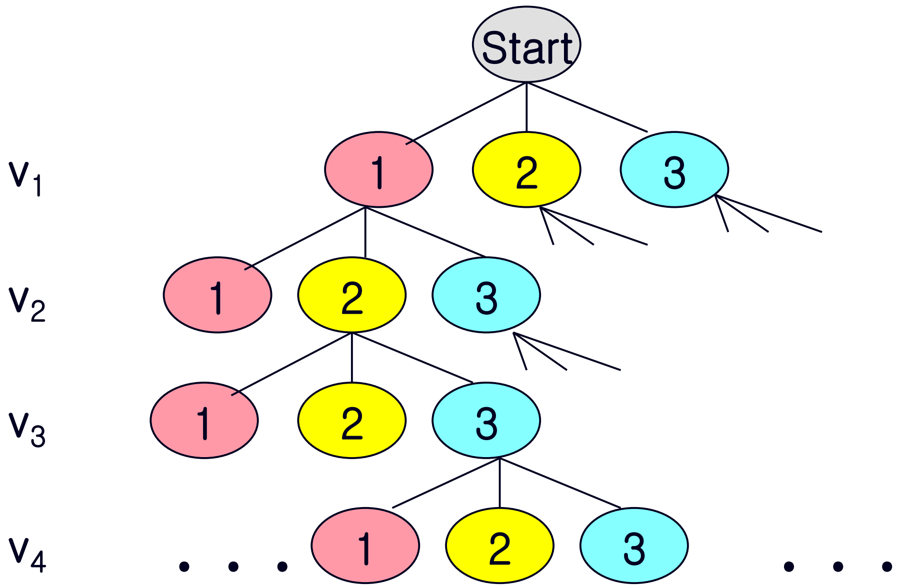
A ***full*** state space tree for 3-coloring problem

5.5 Graph Coloring

❑ *The m-coloring problem*

Example:





A portion of the ***pruned*** state space tree

5.5 Graph Coloring

The m-coloring problem

```
public static void m_coloring(index i)
{
    int color ;

    if (promising(i) )
        if (i==n)
            system.out.print( vcolor[1] .. vcolor[n] )
        else
            for (color=1; color<= m; color++) {
              vcolor[i+1] = color ;
              m_coloring(i+1) ;
            }
}
```

➔ Vcolor[] is globally defined:

➔ **Top level call: *m_coloring*(0)**

5.5 Graph Coloring

- ***The m-coloring problem***

```
public static boolean promising(index i)
{
      index j ;  bool  switch ;

      switch = true;
      j = 1 ;
      while ( j < i && switch ) {
         if ( W[i][j] && vcolor[i] == vcolor[j] )
            switch = false ;
         j++ ;
      }
      return switch ;
}
```

5.7 The 0-1 Knapsack Problem

The 0-1 Knapsack Problem

➔ an optimization problem, so need to find the ***best*** solution

```
public static void checknode (node v)
{
        node u;

        if (value(v) is better than best)
            best = value(v) ;

        if (promising(v))
            for (each child u of v)
                checknode(u) ;
}
```

5.7 The 0-1 Knapsack Problem

- ***The 0-1 Knapsack Problem***
  - State of the knapsack

    - Current profit

    - Current weight

    - The upper bound on the maximum profit

      ➔ the maximum profit that can be obtained by allowing ***fraction*** of an item in the knapsack

      ➔ That is, the maximum profit that can be obtained for the ***fractional knapsack problem***

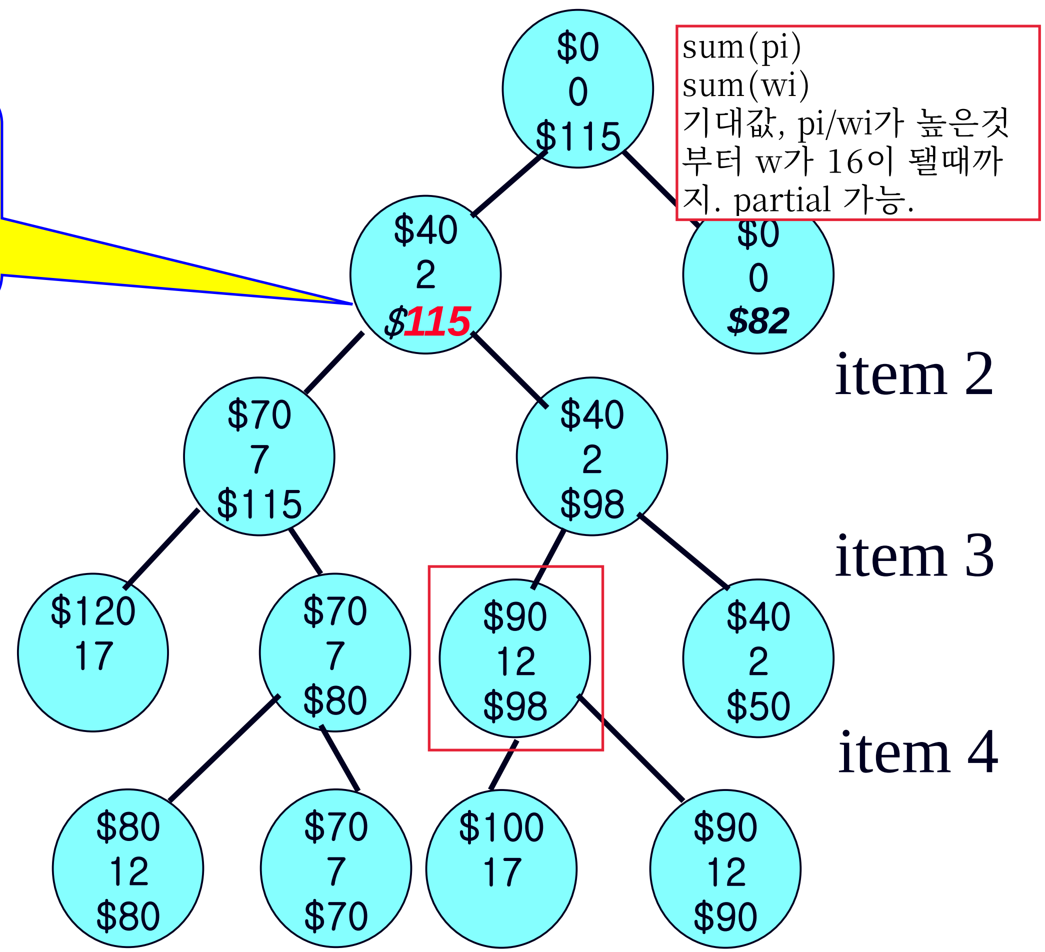
5.7 The 0-1 Knapsack Problem

❑ *Example*

*40*2/2*
*+30*5/5*
+50(9/10)*
= 115

	p_i	w_i	p_i/w_i
item 1 :	40	2	20
item 2 :	30	5	6
item 3 :	50	10	5
item 4 :	10	5	2

W = 16



5.7 The 0-1 Knapsack Problem

```
public static void knapsack(index i, int profit, int weight)
{
    if ( weight <= W && profit > maxProfit ) {
        maxProfit = profit ;
        numBest = i ;
        bestSet = include ;
    }

    if (promising(i)) {
        include[i+1] = “yes” ;
        knapsack(i+1,profit+p[i+1],weight+w[i+1]);
        include[i+1] = “no” ;
        knapsack(i+1,profit, weight);
    }
}
```

Global Variables:

maxProfit,
numBest,
bestSet ,
include

➔**Top level call:**

numBest = 0;
maxProfit = 0 ;
knapsack(0,0,0) ;

5.7 The 0-1 Knapsack Problem

```
public static bool promising(index i)  {
    index j,k ;    int totWeight ;    float bound ;

    if (weight >=W) return false ;
    else {
                j = i+1 ;  bound = profit ; totWeight = weight ;
                while ( j<=n  &&  totWeight + w[j] <= W ) {
                       totWeight = totWeight + w[j] ;
                       bound = bound + p[j] ;
                       j++ ;
                }
                k = j ;
                if (k <= n)
                       bound=bound+(W–totWeight)*p[k]/w[k];
                return bound > maxProfit ;
    }
 }
```