Analysis

- The total number of nodes in the state space tree

$$1 + n + n^2 + n^3 + \cdots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

19,173,961 nodes for n=8

- The promising nodes are at most

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

109,601 nodes for n=8

- Actual number of nodes

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

Trying n! possible solutions.

DFS without backtracking. The number of nodes in the state space tree.

- The Sum-of-Subsets Problem
  - Input : n positive integers w_i (weight) and a positive integer W.
  - Question : Find all subsets of integers that sum to W.
  - Example:

Suppose that $n = 5$, $W = 21$, and

$$w_1 = 5 \quad w_2 = 6 \quad w_3 = 10 \quad w_4 = 11 \quad w_5 = 16.$$

[0/1,...]
Brute Force
$O(2^n)$

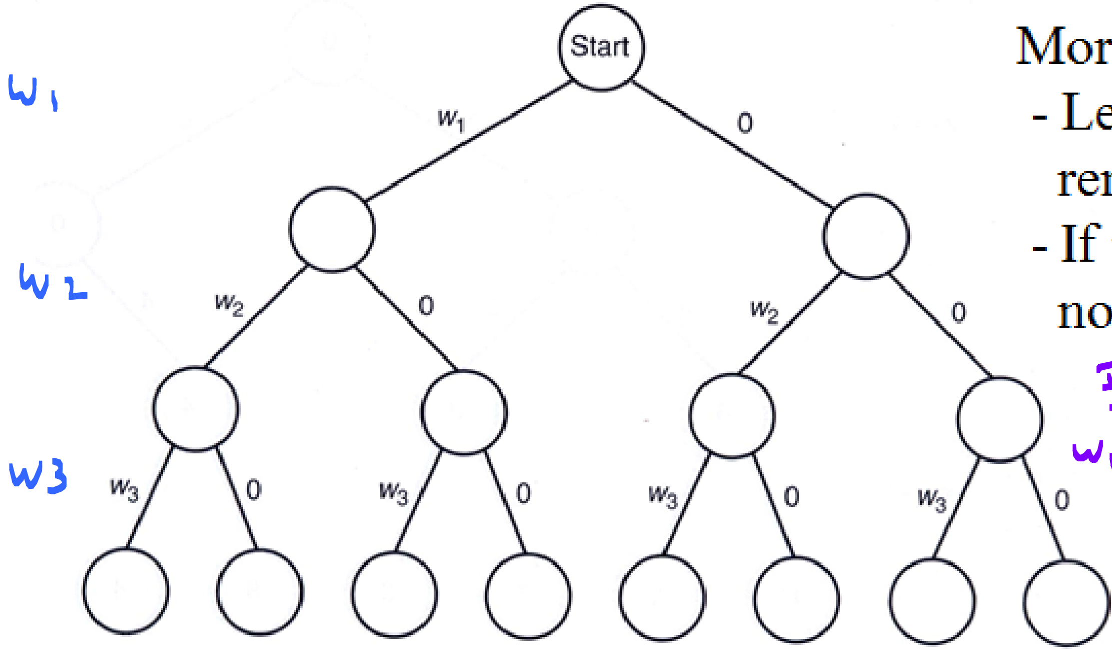
Because $w_1 + w_2 + w_3 = 5 + 6 + 10 = 21$,

$$w_1 + w_5 = 5 + 16 = 21, \text{ and}$$

$$w_3 + w_4 = 10 + 11 = 21,$$

the solutions are $\{w_1, w_2, w_3\}$, $\{w_1, w_5\}$, and $\{w_3, w_4\}$.

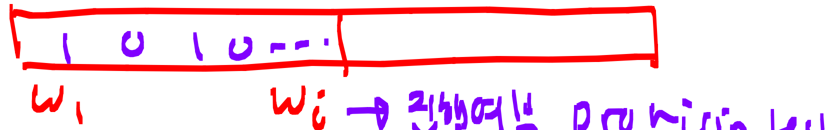
A State Space Tree for n = 3



Checking if promising

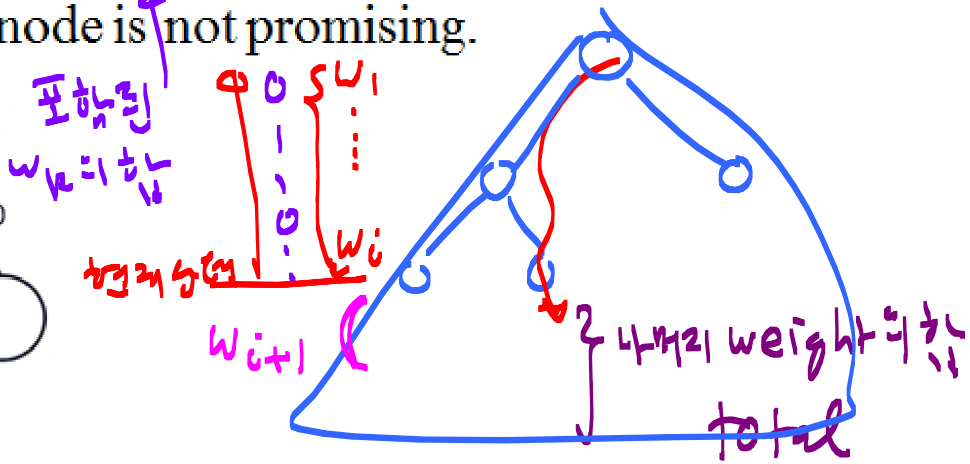
- Sort the weights in non-decreasing order.
- Let ***weight*** be the sum of weights up to level i.
- If $\boxed{\textbf{weight} + w_{i+1}} > W$, the node at ith level can not be promising. ②

구현



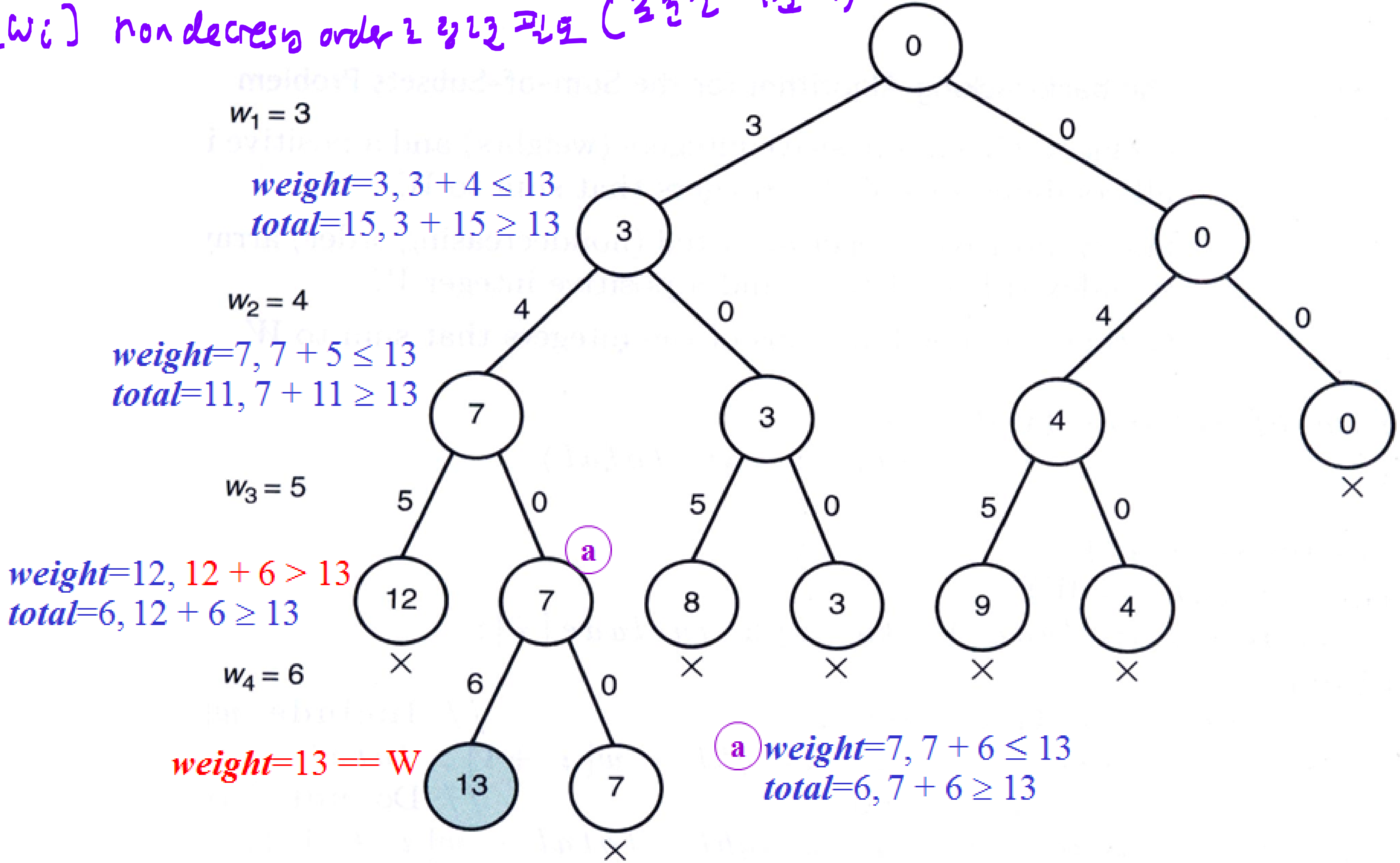
More on promising check :

- Let ***total*** be the sum of remaining weights. ①
- If ***weight*** + ***total*** < W, the node is not promising.

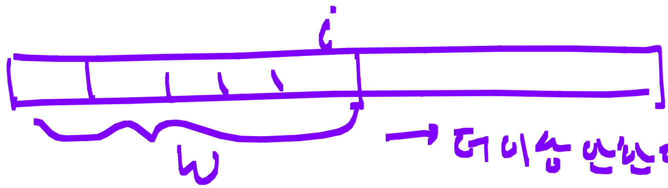


✦ An Example (W = 13)

[w_i] nondecresing order로 정렬 필요 (조건2 때문에)



✦ The Algorithm



→ 더이상 안쓴다. why?

```
void sum_of_subsets (index i, int weight, int total){
  if (promising(i))
     if (weight == W)
        cout << include[1] through include[i];
     else{
        include[i + 1] = "yes";  // Include w[i + 1].
        sum_of_subsets(i + 1, weight + w[i +1], total - w[i + 1]);
        include[i + 1] = ''no"; // Do not include w[i + 1].
        sum_of_subsets(i + 1, weight, total - w[i + 1]);
     }
}

bool promising (index i) {
  return (weight + total >= W) &&
           (weight == W || weight + w[i + 1] <= W);
}
```

The 0-1 Knapsack Problem

Problem Definition

Suppose there are n items. Let

$S = \{item_1, item_2, ..., item_n\}$

$w_i =$ weight of $item_i$ $\quad p_i =$ profit of $item_i$

$W =$ maximum weight the knapsack can hold,

where w_i, p_i, and W are positive integers.

Determine a subset A of S such that $\sum_{item_i \in A} p_i$

is maximized subject to $\sum_{item_i \in A} w_i \le W$.

Example

i	p_i	w_i	$\frac{p_i}{w_i}$
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2

W=16

The optimal solution : {item1, item3}.
The profit = \$90.

- Searching Strategy
  - The same state space tree as the sum-of-subsets problem.
  - However, since the problem is an optimization problem, we do not know if an optimal solution is obtained until the entire state space tree is searched.
  - We need a clever promising checking method.
  - ***best*** : the profit of the best solution found so far. Will be used for promising test.
  - A backtracking procedure

```
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);
}
```

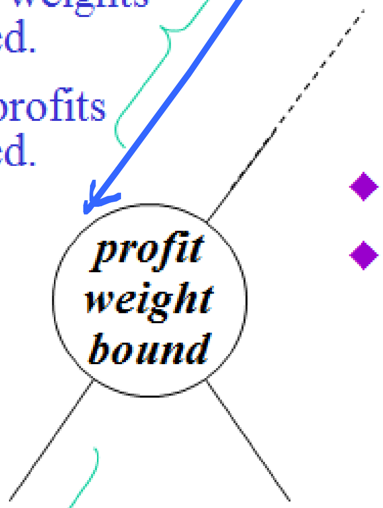
◆ Checking if promising

- ✦ Sort items in nondecreasing order by the values of p_i/w_i.

weight : sum of weights of items included.

profit : sum of profits of items included.

w_i



w_{i+1}

.

.

.

w_{k-1}

w_k

2

- If $\boxed{\boldsymbol{weight} \geq W}$, the node is not promising.
- Let ***maxprofit*** is the value of the profit in the best solution found so far. If $\boxed{\boldsymbol{bound} \leq \boldsymbol{maxprofit}}$, the node is not promising. 1

$\boldsymbol{toweight} = \boldsymbol{weight} + w_{i+1} + w_{i+2} + \ldots + w_{k-1}$

Greedily add items like fractional knapsack until toweight + w_k > W. Then define

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j\right)}_{\text{Profit from first } k-1 \text{ items taken}} + \underbrace{(W - totweight)}_{\text{Capacity available for } k\text{th item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}}.$$

이 방법은 $\frac{p_i}{w_i}$ 의 nonincreasing order로 정렬 필요.

$$\underbrace{\begin{matrix} w_1 & w_2 & \cdots & w_i \\ 1 & 0 & 1\ 1 & \cdots \end{matrix}}_{} \quad \Bigg| \quad \overbrace{w_{i+1} \quad w_{i+2} \quad w_{k-1}}^{\text{모두 포함}} \quad \overbrace{w_k}^{\text{일부 포함}} \quad \cdots$$

weight = 선택된 item의 weight 합

profit = 이득

에 대한 optimal 을 예측.

⇒ fractional knapsack.

toweight $= w_{i+1} + w_{i+2} \cdots w_{k-1}$

조건 ① weight + toweight + $w_k \geq W$ 인 k를 잡는다.

② w_k = 잘라서 일부만 포함

→ 이때의 profit : $w_{i+1} \cdots w_k$ 까지

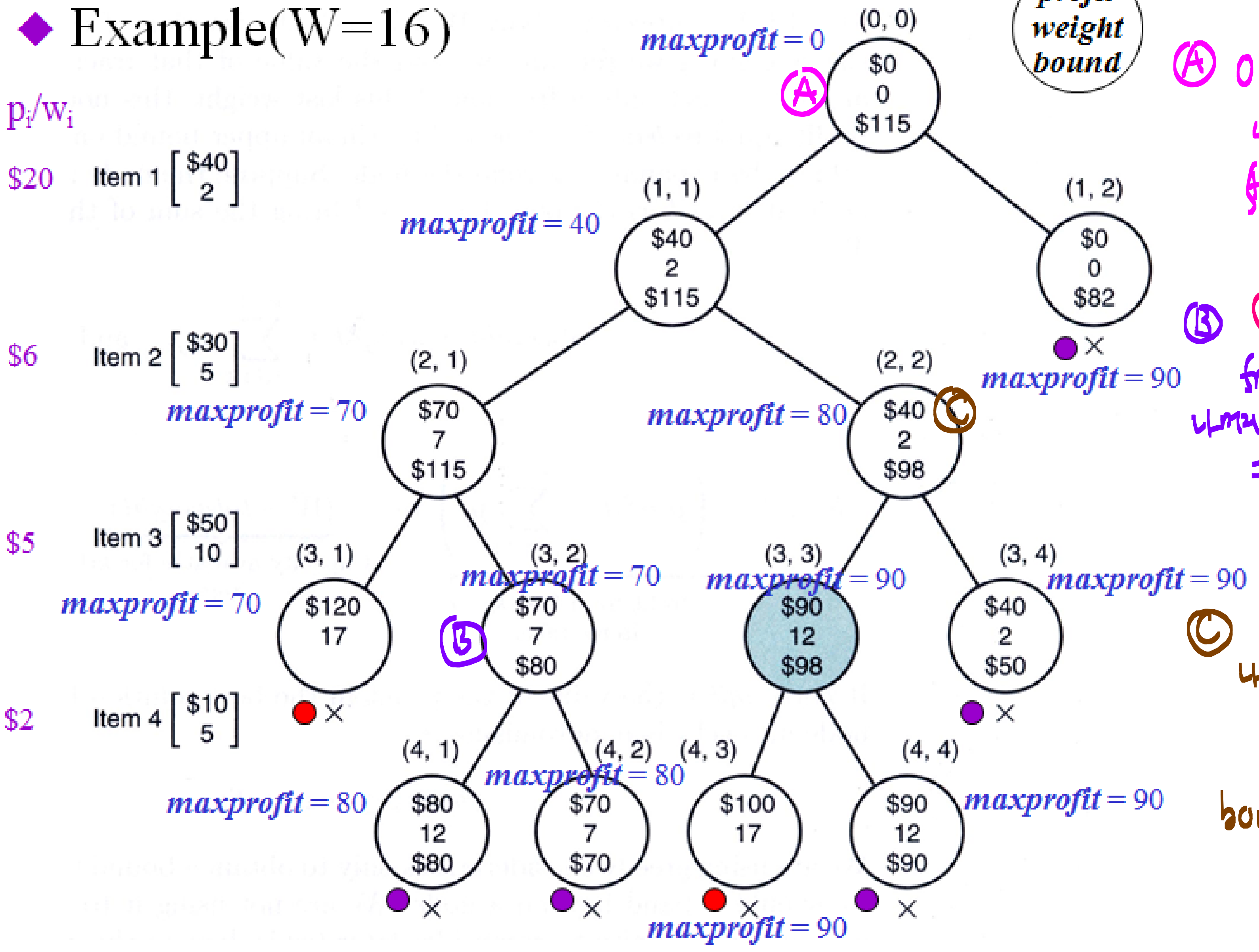
$$\boxed{bound = \underbrace{profit}_{\text{현재까지 이득}} + \underbrace{\sum_{p=i+1}^{k-1} p_i + ((W - toweight)/w_k)\, p_k}_{\text{앞으로 얻을 수 있는 이득의 최대 가능한 값(upper bound)}}}$$

max profit의 upper bound 이다.

만일 bound $\leq$ $\underset{\text{max profit}}{\underline{best}}$ 이면 non promising.

Nonpromising condition : *weight* ≥ W, *bound* ≤ *maxprofit*

◆ Example(W=16)



bound = profit + fractional knapsack profit (계산)

A 0 + $\boxed{}$

w_i = 2 5 10 5

fractional profit 일부

20 6 $50 \times \frac{9}{10}$ = 20 + 6 + 45 = 115

B 70 + fractional

남은 weight = 9

5 fractional prof 10

bound = 80

C 40 +

남은 w 14

10 5 일부 (4)

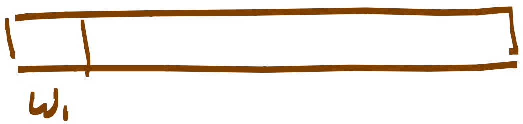
bound = $40 + 50 + \frac{4}{5} 10$

= 98

◆ Algorithm Implementation

Calling Functon

```
numbest = 0;
maxprofit = 0;
knapsack(0 , 0 , 0);
cout << maxprofit;                  // Write the maximum profit.
for (j = 1; j <= numbest; j++)  // Show an optimal set of items.
  cout << bestset[i];
```



Knapsack function

```
void knapsack (index i, int profit, int weight){
  if (weight <= W && profit > maxprofit){
     maxprofit = profit;  // This set is best so far.
     numbest = i;   // Set numbest to number of items considered.
     bestset = include;   // Set bestset to this solution.
  }
  if (promising(i)){
     include[i + 1] = "yes";   // Include w[i + 1].
     knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
     include[i + 1] = "no";    // Do not include w[i + 1].
     knapsack(i + 1, profit , weight);
  }
}
```

best 저장.

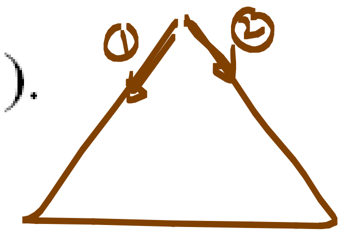
넣는경우

안 넣는경우.

◆ Algorithm Implementation(cont'd)

```
bool promising (index i) {
  index j, k;   int totweight;   float bound;
  if (weight >= W)            // Node is promising only
     return false;            // if we should expand to
  else{                       // its children. There must
     j = i + 1;               // be some capacity left for
     bound = profit;          // the children.
     totweight = weight;
     while (j <= n && totweight + w[j] < = W){
        totweight = totweight + w[j];  // Grab as many items as
        bound = bound + p[j];          // possible.
        j++;
     }
     k = j;          // Use k for consistency
     if (k <=n)      // with formula in text.
        bound = bound + (W - totweight) * p[k]/w[k];
     return bound > maxprofit;   // Grab fraction of kth item.
  }
}
```

- Comparison between the Dynamic Programming Algorithm and the Baktracking Algorithm
  - The dynamic programming algorithm : $O(\min(2^n, nW))$.
  - The backtracking algorithm : $O(2^n)$.
  - Difficult to compare theoretially.
  - By many experiments, the backtracking algorithm is found to be more efficient.
  - Horowitz and Sahni (1974) : $O(2^{n/2})$ algorithm
    - Combination of the divide and conquer approach and the dynamic programming approach.



Branch-and-Bound

$w_1 \quad w_2 \quad w_3 \quad - \quad w_{k-1} \mid w_k \quad - \quad w_n$

backtra'

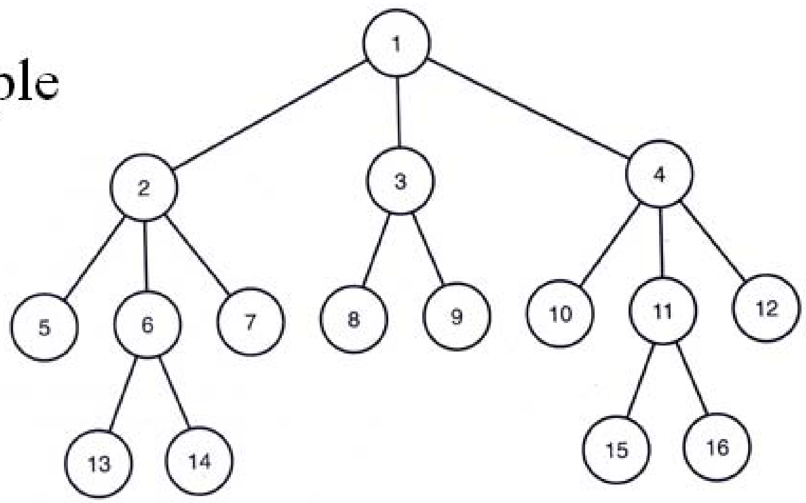
모두 넣는 조합

$> W$

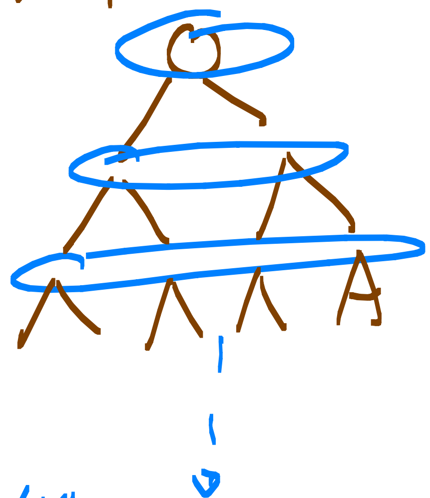
- **Breath First Tree Searching**
  - Visits nodes level by level (from low to high)
  - void *breadth_first_tree_search* (tree *T*){

```
    queue_of_node Q;   node u, v;
    initialize(Q); // Initialize Q to be empty.
    v = root of T;   visit v;   enqueue(Q, v);
    while (! empty(Q)) {
        dequeue(Q, v);
        for (each child u of v){
            visit u;   enqueue(Q, u);
 } } }
```

  - An Example



Knapsack



queue 필요

FIFO

priority queue 사용

- Branch-and-Bound
  - Search the state space tree in BFS like fashion.
  - Use the same strategy of promising checking to stop or continue searching as the backtracking.
  - If no preference in selecting a node at each level (just search FIFO based), we call the method breath-first search with branch-and-bound pruning.
  - We may give some preference in selecting a node at each level for searching. In this case we call the method best-fit search with branch-and-bounding pruning.

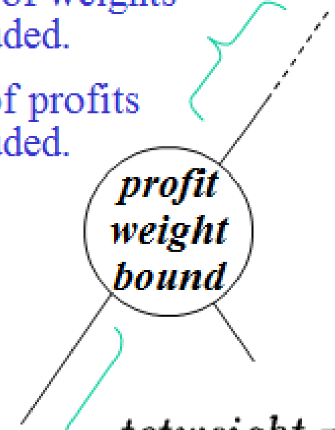
Breath-First-Search with B&B for 0-1 Knapsack

Promising Check

w_i

weight : sum of weights of items included.

profit : sum of profits of items included.



w_{i+1}

.

.

.

w_{k-1}

w_k

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

An Instance

W=16

i	p_i	w_i	$\frac{p_i}{w_i}$
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2

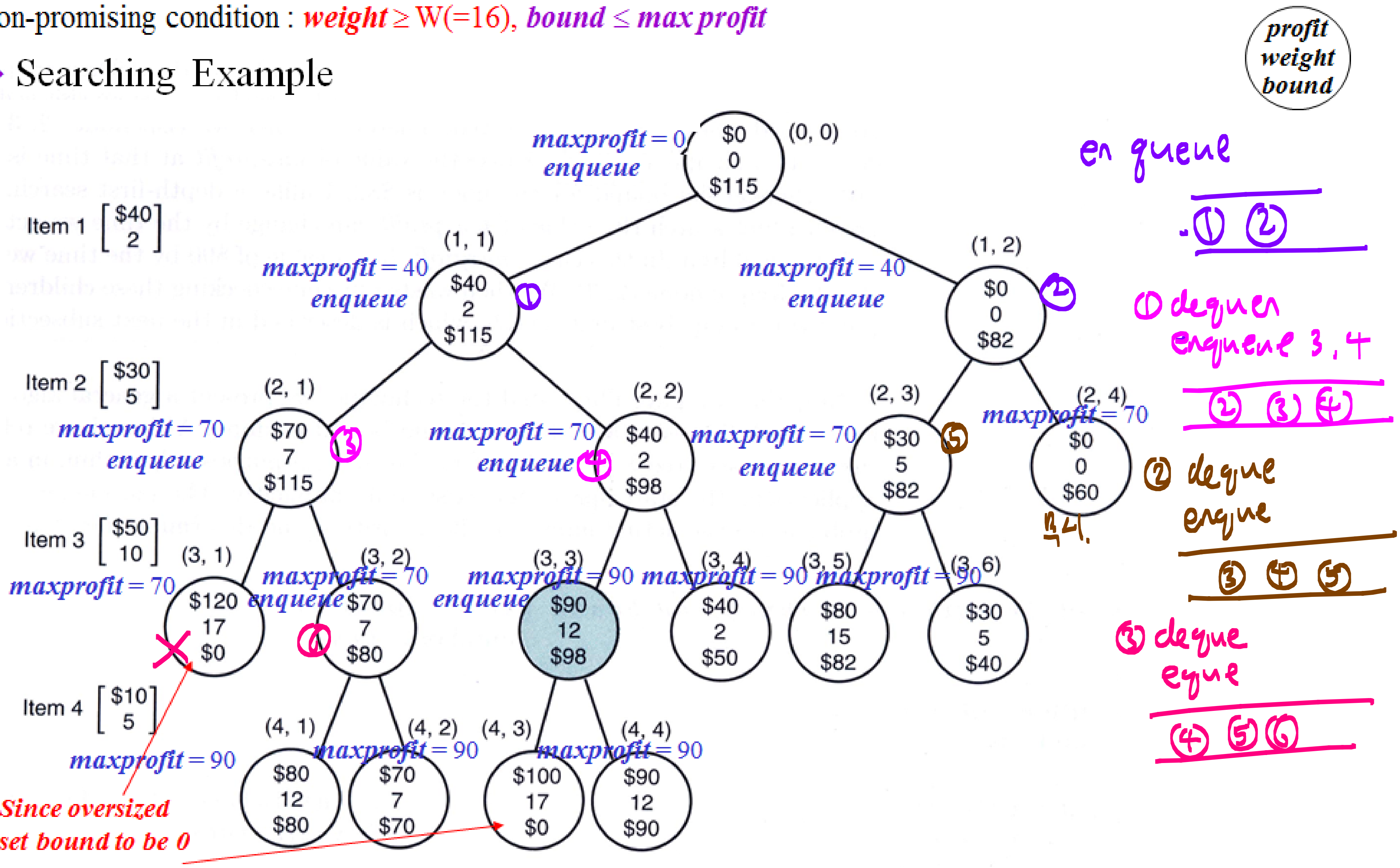
- If ***weight*** ≥ W, the node is not promising.
- Let ***maxprofit*** is the value of the profit in the best solution found so far.
  If ***bound*** ≤ ***maxprofit***, the node is not promising.

Greedily add items like fractional knapsack until ***toweight*** + w_k > W. Then define

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j\right)}_{\text{Profit from first } k-1 \text{ items taken}} + \underbrace{(W - totweight)}_{\text{Capacity available for } k\text{th item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}}.$$

Non-promising condition : *weight* ≥ W(=16), *bound* ≤ *max profit*

◆ Searching Example



◆ The BFS with B&B Pruning Algorithm

```
struct node {
  int level;
  int profit;
  int weight;
};
```

```
void knapsack2 (int n, const int p[], const int w[],
                int W, int& maxprofit) {
  queue_of_node Q;    node u, v;
  initialize(Q); // Initialize Q to be empty.
  v.level = 0; v.profit = 0; v.weight = 0; // Initialize
  maxprofit = 0; enqueue(Q, v);    // v to be the root.
  while (! empty(Q)){
     dequeue(Q, v);
     u.level = v.level + 1;         // Set u to a child of v.
     u.weight = v.weight + w[u.level]; // Set u to the child
     u.profit = v.profit + p[u.level]; // that includes the
                                       // next item.
     if (u.weight <= W  && u.profit > maxprofit)
        maxprofit = u.profit;
     if (bound(u) > maxprofit)
        enqueue(Q, u);
     u.weight = v.weight;           // Set u to the child that
     u.profit = v.profit;           // does not include the
     if (bound(u) > maxprofit)      // next item.
        enqueue(Q, u);
  }
}
```

◆ The bound function

```
struct node {
  int level;
  int profit;
  int weight;
};
```

```
float bound (node u) {
   index j, k;   int totweight;  float result;
   if (u.weight >= W) return 0;
   else{
      result = u.profit;
      j = u.level + 1;
      totweight = u.weight;
      while (j <= n && totweight + w[j] <= W){
        totweight = totweight + w[j]; // Grab as many items
        result = result + p[j];  j++; // as possible.
      }
      k = j;        // Use k for consistency
      if (k <=n)  // with formula in text.
        result = result + (W - totweight) * p[k]/w[k];
      return result;      // Grab fraction of kth item.
   }
}
```

Best-First-Search with B&B for 0-1 Knapsack

- In general, the BFS strategy has little advantage over backtracking.
- Improvement : after visiting all the children of a given node, look at all the promising, unexpanded nodes and expand beyond the one with the best bound.
- The generic procedure

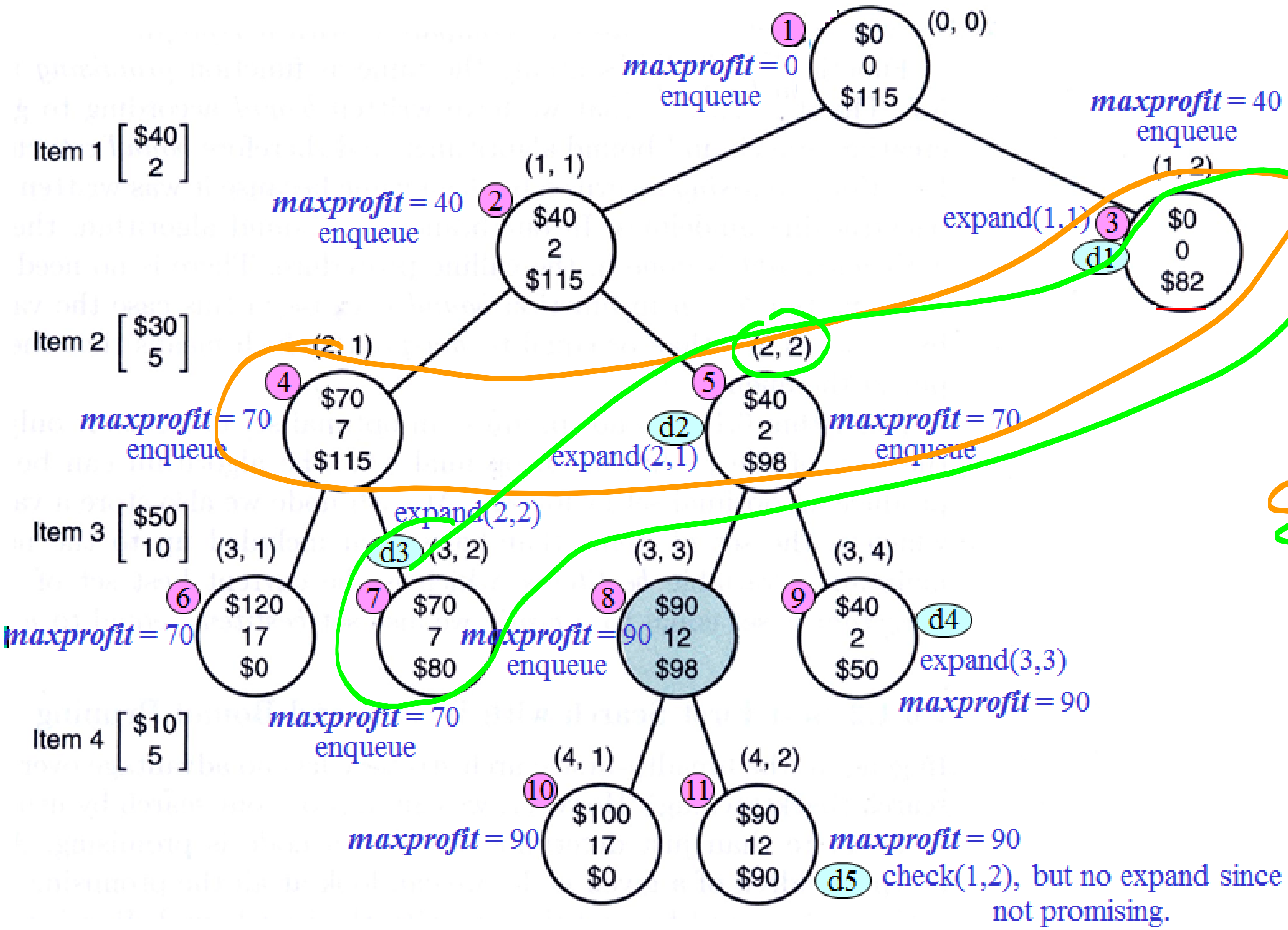
```
void best_first_branch_and_bound (state_space_tree T,
                                  number& best){
  priority_queue_of_node PQ;  node u, v;
  initialize(PQ); // Initialize PQ to be empty.
  v = root of T; best = value(v); insert(PQ, v);
  while (! empty(PQ)){
    remove(PQ, v); // Remove node with best bound.
    if (bound(v) is better than best) // Check if node is
        for (each child u of v){      // still promising.
           if (value(u) is better than best) best = value(u);
           if (bound(u) is better than best) insert(PQ, u);
        }
  }
}
```

Nonpromising condition : *weight* ≥ W(=16), *bound* ≤ *maxprofit*

ⓓ **decision**

profit
weight
bound

◆ Best-First-Search Example



PQ : (0,0)

d1 : (1,1) (1,2)

d2 : (2,1) (2,2) (1,2)

d3 : (2,2) (1,2) (3,2)

d4 : (3,3) (1,2) (3,2)

d5 : (1,2) (3,2)

(3,2) : not promising

ϕ

◆ Best-First-Search with B&B Pruning Algorithm

```
void knapsack3 (int n, const int p[], const int w[],
                int W, int& maxprofit){
  priority_queue_of_node PQ; node u, v;
  initialize(PQ); // Initialize PQ to be empty.
  v.level = 0; v.profit = 0; v.weight = 0;
  maxprofit = 0;      // Initialize v to be the root.
  v.bound = bound(v);
  insert(PQ, v);
  while (! empty(PQ)){
    remove(PQ, v);  // Remove node with best bound.
    if (v.bound > maxprofit){ // Check if node is still promising.
      u.level = v.level + 1;
      u.weight = v.weight + w[u.level]; // Set u to the child
      u.profit = v.profit + p[u.level]; // that includes the
      if (u.weight <= W && u.profit > maxprofit) // next item.
        maxprofit = u.profit;
      u.bound = bound(u);
      if (u.bound > maxprofit)
        insert(PQ, u);
      u.weight = v.weight;  // Set u to the child
      u.profit = v.profit;  // that does not include
      u.bound = bound(u);   // the next item.
      if (u.bound > maxprofit) insert(PQ, u);
    }
  }
}
```