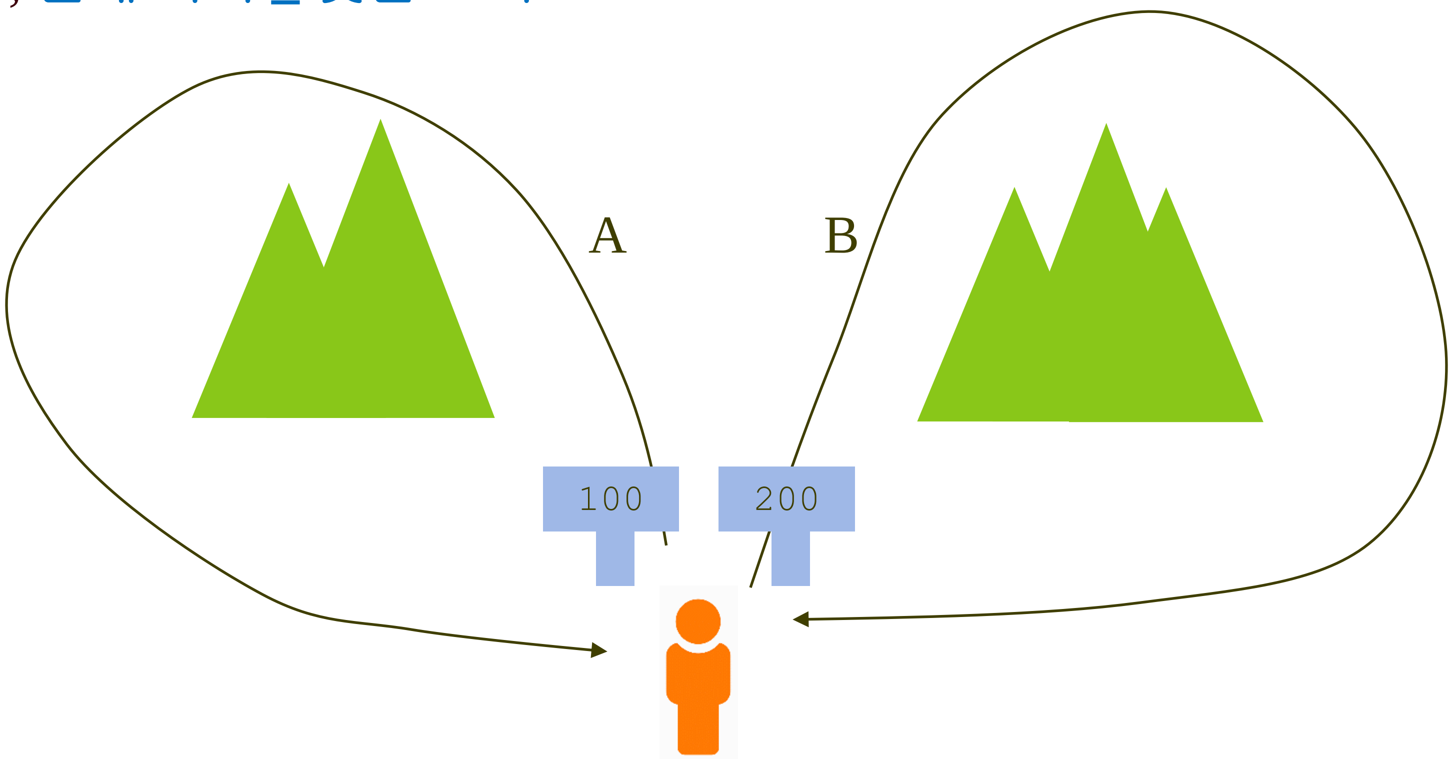
6장　분기한정법
(Branch-and-Bound)

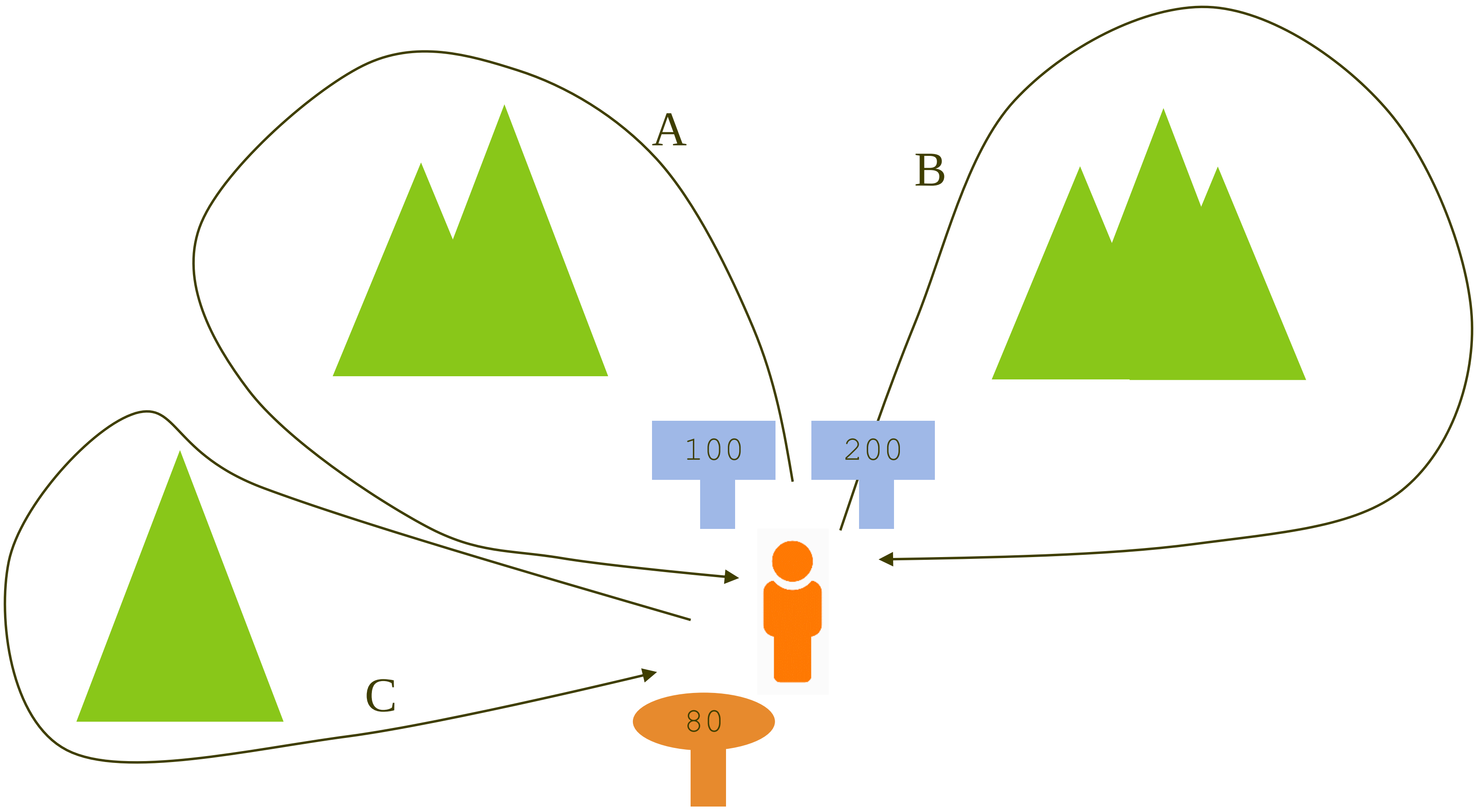
• 최소화

문제: 최소 거리의 트래킹코스를 찾는다.

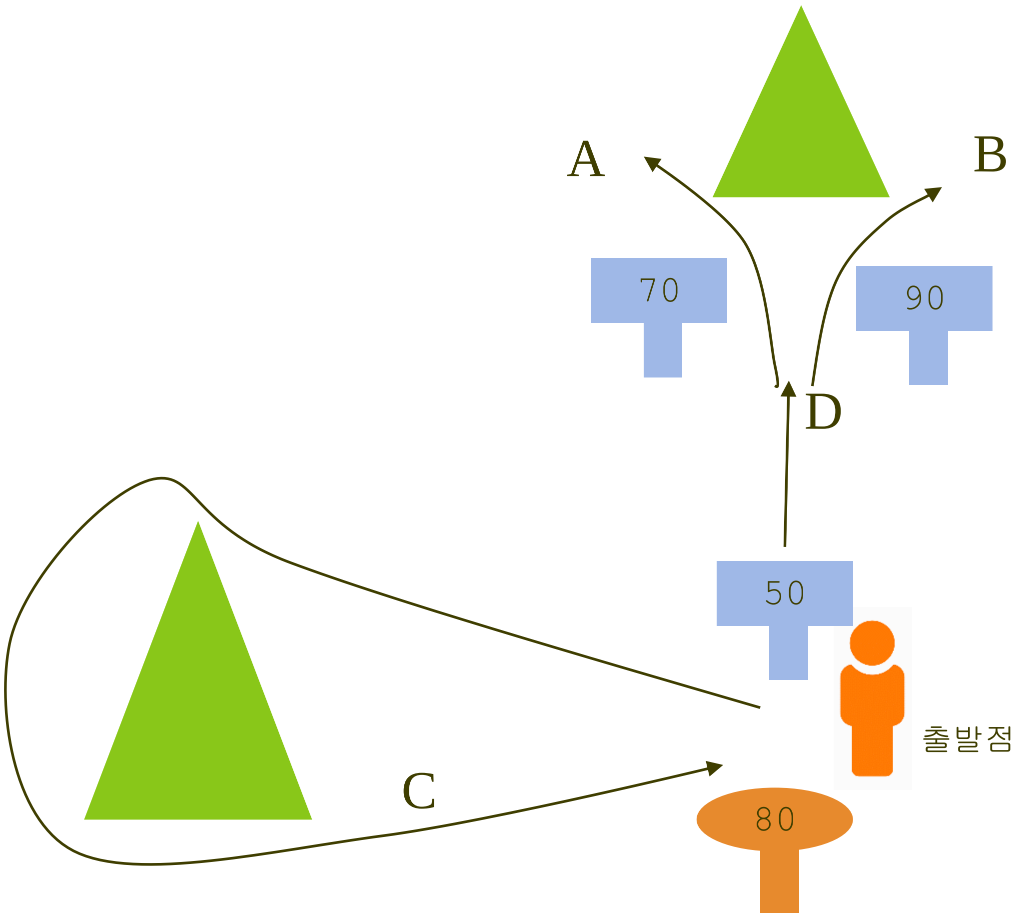
- 각 팻말은 그 코스로 갔을 때의 최소트래킹 코스 거리를 알려 준다.
- 그러나, 그 거리의 트래킹 코스가 있다는 것을 의미하지 않고, 트래킹 코스 거리의 하한을 보여 준다.
- 즉, 실제 거리 ≥ 팻말 표시



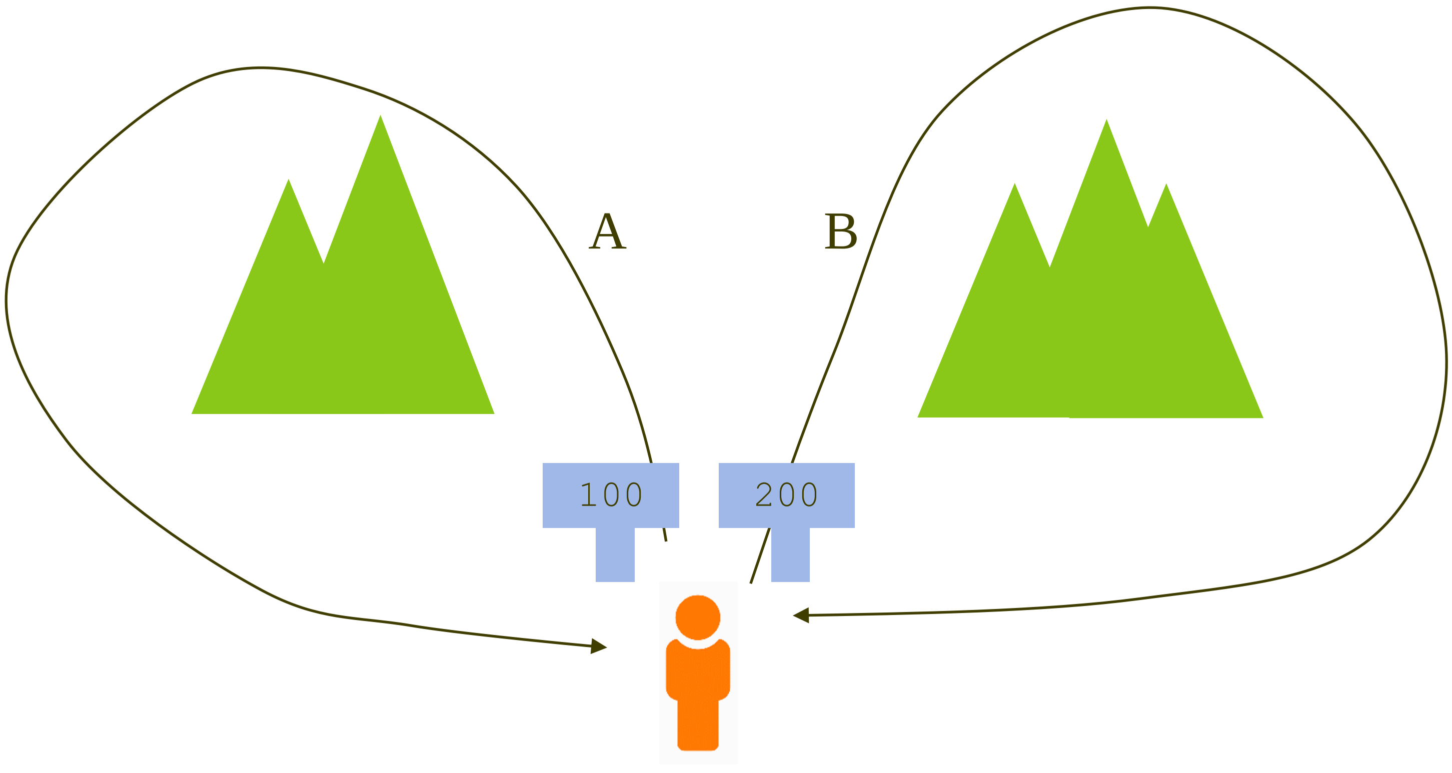
- ✓ Q: 만일 이미 C코스(실제 거리 80)을 안내자가 알고 있다면, A와 B를 탐사할까?



- Q: 이미 C코스(실제 거리 80)을 안내자가 알고 있다고 가정.
  출발점에서 시작해서 D로 이동하니, A와 B의 갈림길에서 팻말 확인.
  A는 70, B는 90. 어느 곳을 탐사할까?



Q: 안내자는 A와 B 중 어느 곳을 먼저 탐색하는 것이 유리할까?

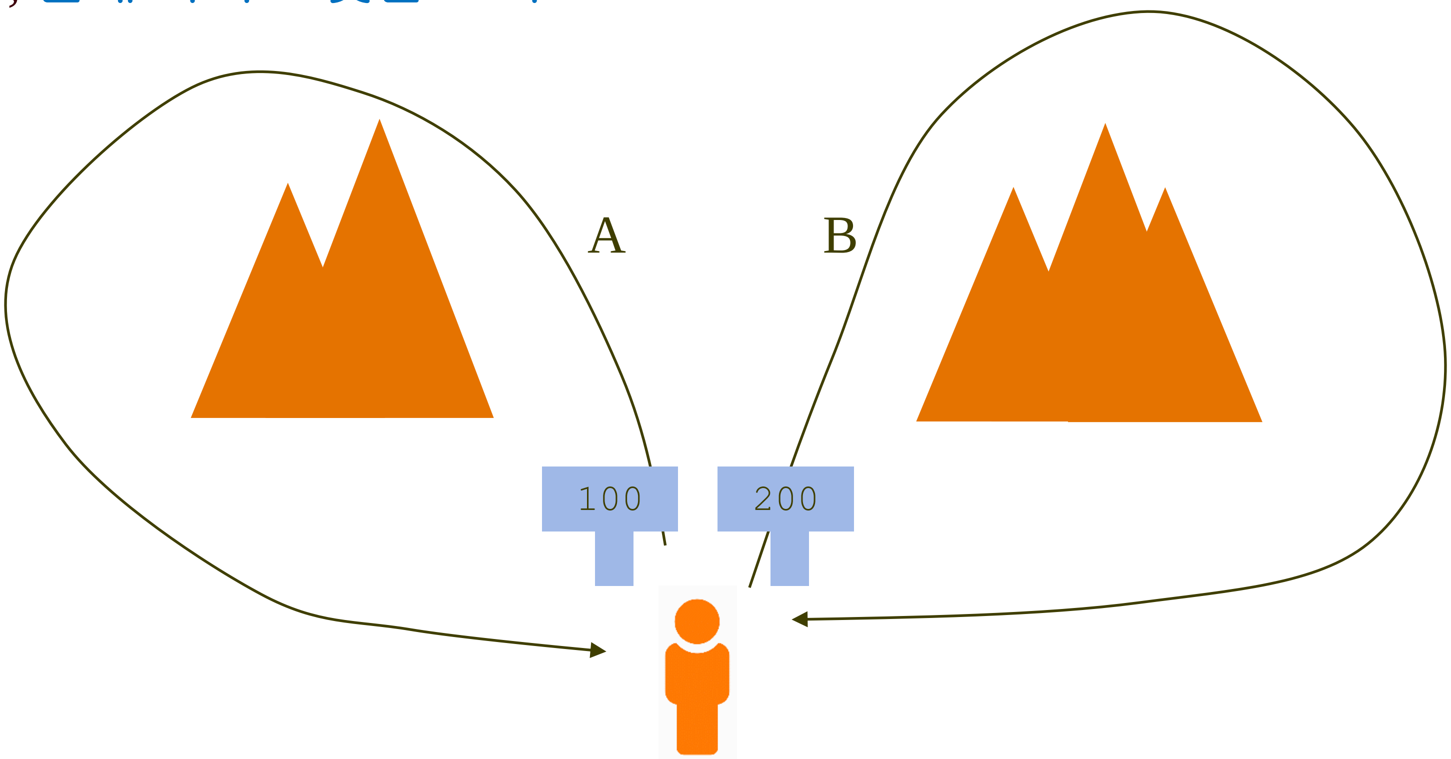


팻말: bound

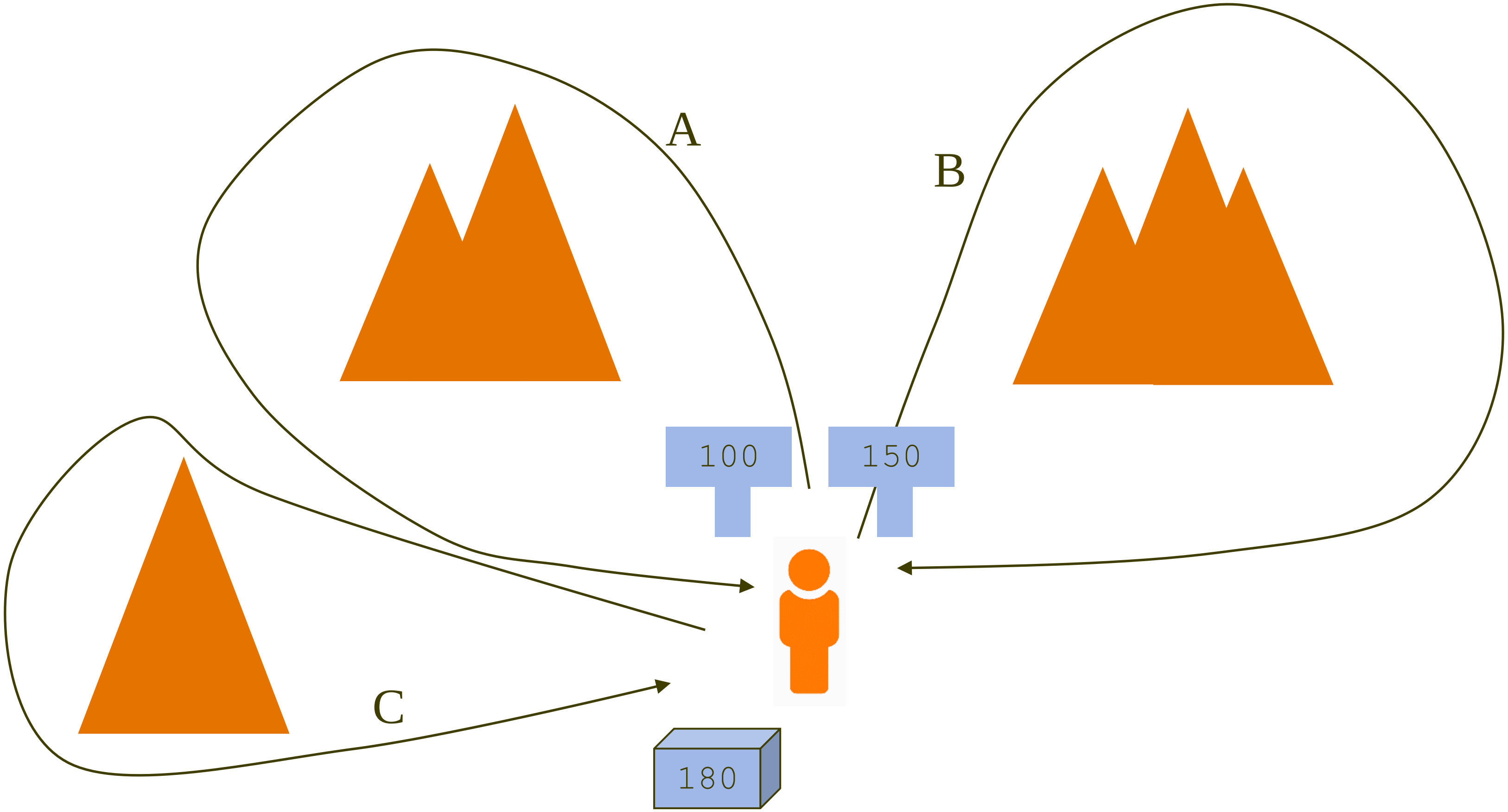
• 최대화

문제: 최대가치의 보물을 찾는다.

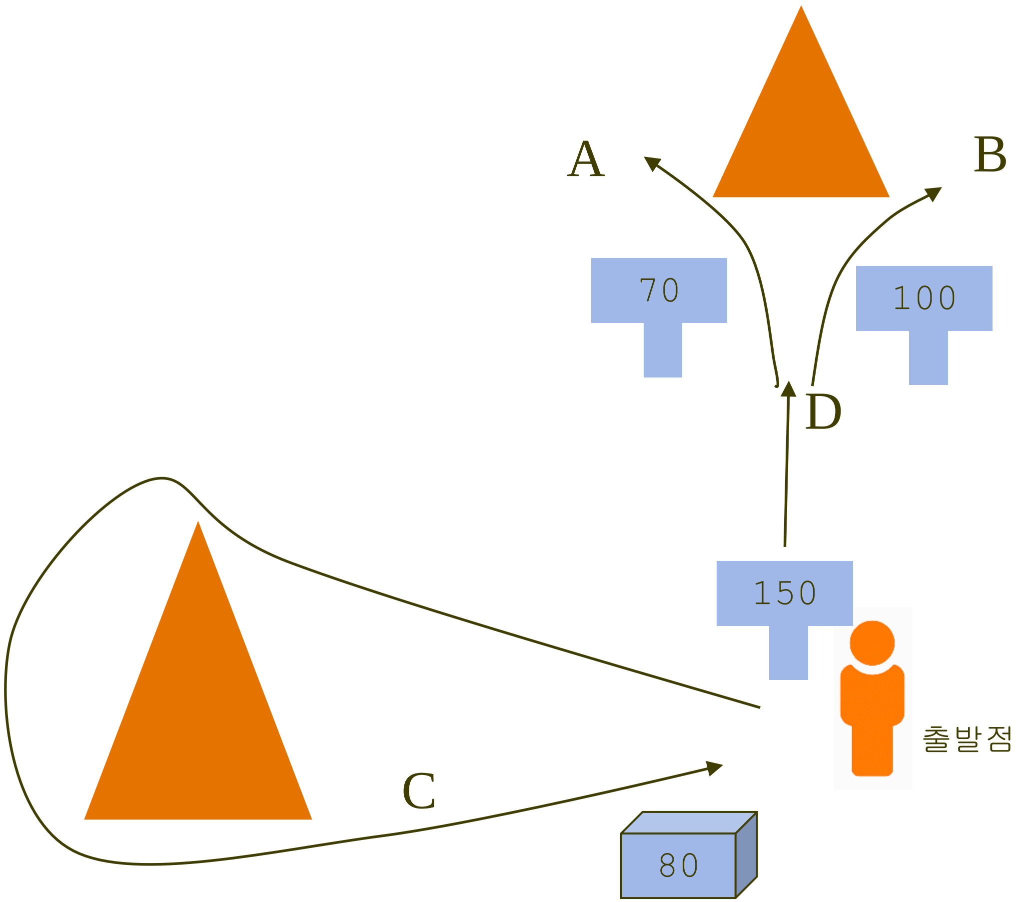
- 각 팻말은 그 코스로 갔을 때의 가능한 보물의 최대가치를 알려 준다.
- 그러나, 그 가치의 보물이 있다는 것을 의미하지 않고, 보물 가치의 상한을 보여 준다.
- 즉, 실제 가치 ≤ 팻말 표시



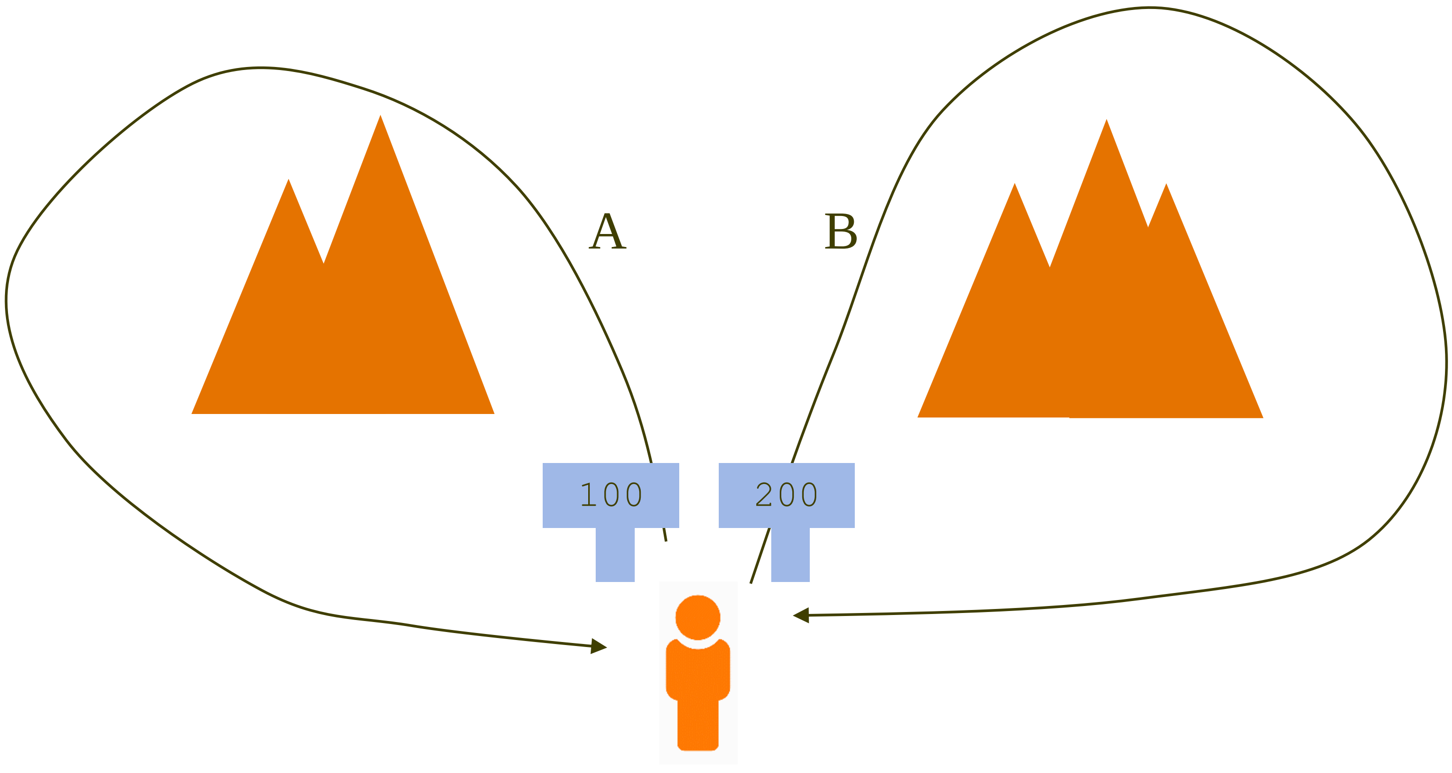
- ✓ Q: 만일 이미 C코스(실제 보물 180)을 안내자가 알고 있다면, A와 B를 탐사할까?



- Q: 이미 C코스(실제 보물 80)을 알고 있다고 가정. 출발점에서 시작해서 D로 이동하니, A와 B의 갈림길에서 팻말 확인. A는 70, B는 100. 어느 곳을 탐사할까?



Q: 안내자는 A와 B 중 어느 곳을 먼저 탐색하는 것이 유리할까?



팻말: bound

분기한정법(branch-and-bound)

- 특징:
  - ✓ 되추적 기법과 같이 상태공간트리를 구축하여 문제를 해결.
  - ✓ 최적의 해를 구하는 문제(optimization problem)에 적용할 수 있음.
  - ✓ 최적의 해를 구하기 위해서는 모든 해를 다 고려해 보아야 하므로 트리의 마디를 순회(traverse)하는 방법에 구애 받지 않음.
- 분기한정 알고리즘의 원리
  - ✓ 각 마디를 검색할 때 마다, 그 마디가 **유망(promising)**한지의 여부를 결정하기 위해서 한계값(**bound**)을 계산한다.
  - ✓ 그 한계치는 그 마디로부터 가지를 뻗어나가서(**branch**) 얻을 수 있는 해답값의 한계를 나타낸다.
  - ✓ 따라서 만약 그 한계값이 지금까지 찾은 최적의 해답값 보다 좋지 않은 경우는 더 이상 가지를 뻗어서 검색을 계속할 필요가 없으므로, 그 마디는 **유망하지 않다(nonpromising)**고 할 수 있다.

0-1 Knapsack Problem

- problem: $S = \{item_1, item_2, \ldots, item_n\}$

  w_i = weight of $item_i$

  p_i = profit of $item_i$

  W = maximum weight the knapsack can hold.

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

$$\sum_{item_i \in A} w_i \leq W$$

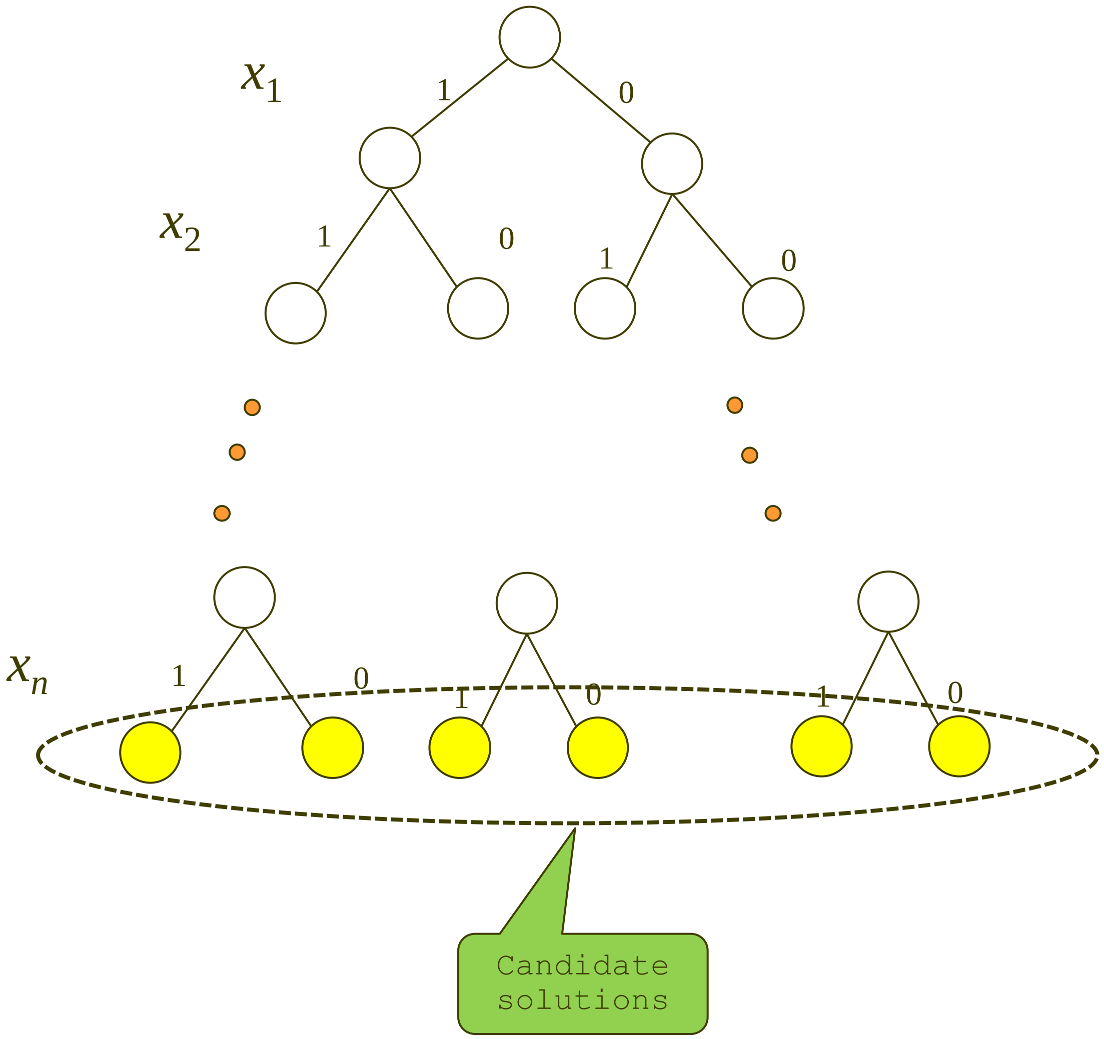
$$\begin{aligned} &MAX \quad \sum_{item_i \in A} p_i \\ &subject \;\; \text{to} \;\; \sum_{item_i \in A} w_i \leq W \end{aligned}$$

or

$$\begin{aligned} &MAX \quad \sum_{i=1}^{n} p_i x_i \\ &subject \;\; \text{to} \;\; \sum_{i=1}^{n} w_i x_i \leq W \\ &\qquad x_i = 0 \;\; \text{or} \;\; 1, \quad for \;\; i = 1, n \end{aligned}$$

0-1 배낭채우기 문제

- 되추적 (bound 개념이 아직 없음)
  - ✓ 상태공간트리를 구축하여 되추적 기법으로 문제를 푼다.
  - ✓ 뿌리마디에서 왼쪽으로 가면 첫번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 첫번째 아이템을 배낭에 넣지 않는 경우이다.
  - ✓ 동일한 방법으로 트리의 수준 1에서 왼쪽으로 가면 두 번째 아이템을 배낭에 넣는 경우이고, 오른쪽으로 가면 그렇지 않는 경우이다.
  - ✓ 이런 식으로 계속하여 상태공간트리를 구축하면, 뿌리마디로부터 잎마디까지의 모든 경로는 해답후보가 된다.
  - ✓ 이 문제는 최적의 해를 찾는 문제(optimization problem)이므로 검색이 완전히 끝나기 전에는 해답을 알 수가 없다. 따라서 검색을 하는 과정 동안 항상 그 때까지 찾은 최적의 해를 기억해 두어야 한다.


x_1
1
0
x_2
1
0
1
0
x_n
1
0
1
0
1
0
Candidate solutions

최적화 문제를 풀기 위한 일반적인 되추적 알고리즘

```
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);
}
```

최적화문제이므로 최적값을 유지

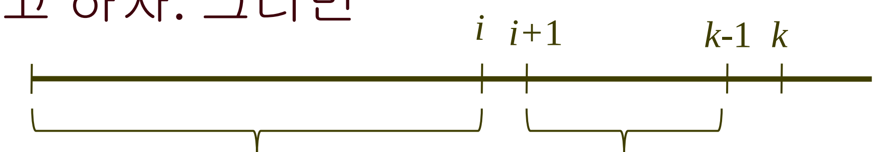
feasible solution 인지 확인하는 절차 포함. 즉 자신이 유효한지도 확인

자식으로 확장 가능한지, 즉, 앞으로 뻗어 나갈 수 있는 가능성 확인: bound 이용

- ✓ best : 지금까지 찾은 제일 좋은 해답값.
- ✓ value(v) : v 마디에서의 해답값.

0-1 배낭채우기: 알고리즘

- 알고리즘 스케치:아이템은 무게당 가치가 감소하는 순서로 정렬 가정
  - *profit* : 그 마디에 오기까지 넣었던 아이템의 값어치의 합.
  - *weight* : 그 마디에 오기까지 넣었던 아이템의 무게의 합.
  - *bound* : 마디가 수준 i에 있다고 하고, 수준 k에 있는 마디에서 총무게가 W를 넘는다고 하자. 그러면



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

이론적으로 현 상태에서 얻을 수 있는 최대 이익

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j\right)}_{\text{처음 } k\text{-1개를 이용한 아이템의 값어치}} + \underbrace{(W - totweight)}_{k\text{번째 아이템에 가용한 용량}} \times \underbrace{\frac{p_k}{w_k}}_{k\text{번째 아이템의 단위 무게당 값어치}}$$

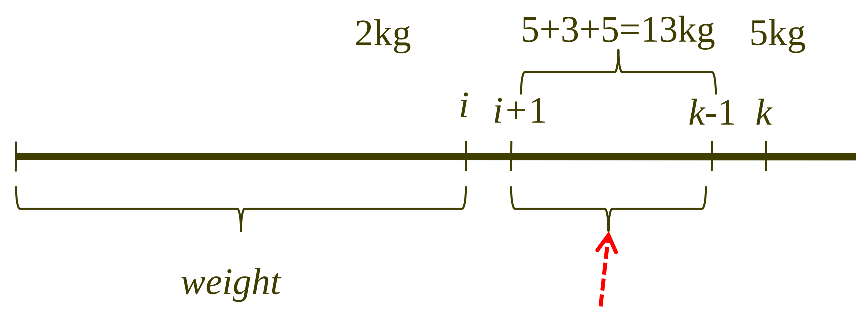
i	p_i	w_i	p_i/w_i
1	\$40	2	\$20
2	\$35	5	\$7
3	\$18	3	\$6
4	\$25	5	\$5
5	\$10	5	\$2

W=16

첫번째 아이템을 넣은 상태의 bound 계산 예

- bound = \$40 + \$35+\$18+ \$25+1*\$10/5=\$118+\$2 =\$120

다섯 번째 아이템이 들어갈 수 있는 공간은 1kg=16-2-13



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

이론적으로 현 상태에서 얻을 수 있는 최대 이익

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{처음 } k\text{-1개를 이용한 아이템의 값어치}} + \underbrace{(W - totweight)}_{k\text{번째 아이템에 가용한 용량}} \times \underbrace{\frac{p_k}{w_k}}_{k\text{번째 아이템의 단위 무게당 값어치}}$$

- ✓ *maxprofit* : 지금까지 찾은 최선의 해답이 주는 값어치
- ✓ bound ≤ maxprofit 이면 수준 i에 있는 마디는 유망하지 않다.
- ✓ w_i와 p_i를 각각 i번째 아이템의 무게와 값어치라고 하면, p_i / w_i 의 값이 큰 것부터 내림차순으로 아이템을 정렬한다. (일종의 탐욕적인 방법이 되는 셈이지만, 알고리즘 자체는 탐욕적인 알고리즘은 아니다.)
- ✓ 초기값:

  maxprofit := $0;   *profit* := $0;   *weight* := 0

- ✓ 깊이우선순위로 각 마디를 방문하여 다음을 수행한다:
  1. 그 마디의 *profit*과 *weight*를 계산한다.
  2. 그 마디의 *bound*를 계산한다.
  3. $weight < W$ and $bound > maxprofit$이면, 검색을 계속한다;
     그렇지 않으면, 되추적.
- ✓ 고찰: 최선이라고 여겼던 마디를 선택했다고 해서 실제로 그 마디로부터 최적해가 항상 나온다는 보장은 없다.

- (예 5.6) $n = 4$, $W = 16$

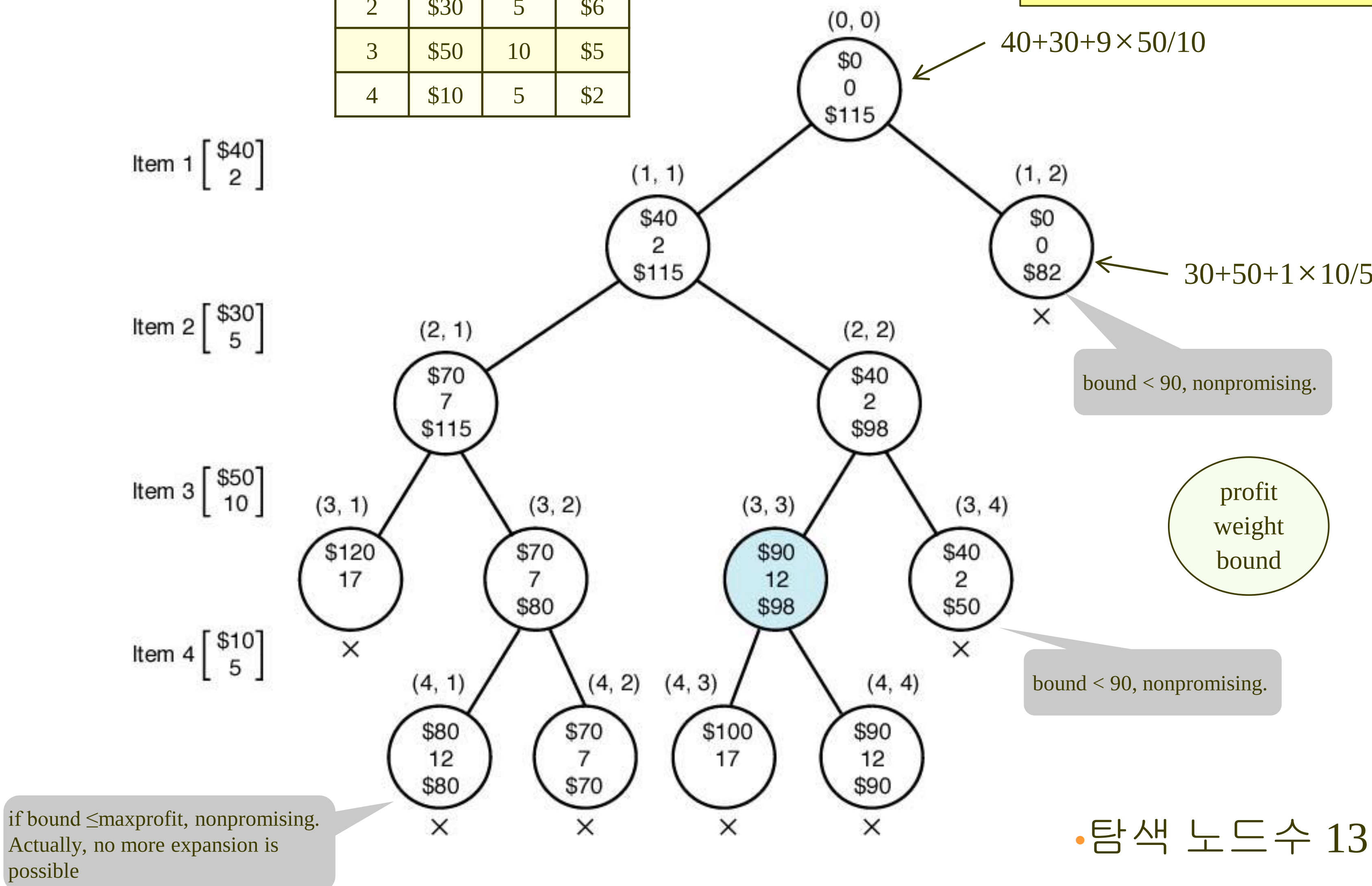
i	p_i	w_i	p_i/w_i
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2

일 때, 되추적을 사용하여 구축되는 가지친 상태공간트리를 그려 보시오.

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

W=16

```
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);}
```

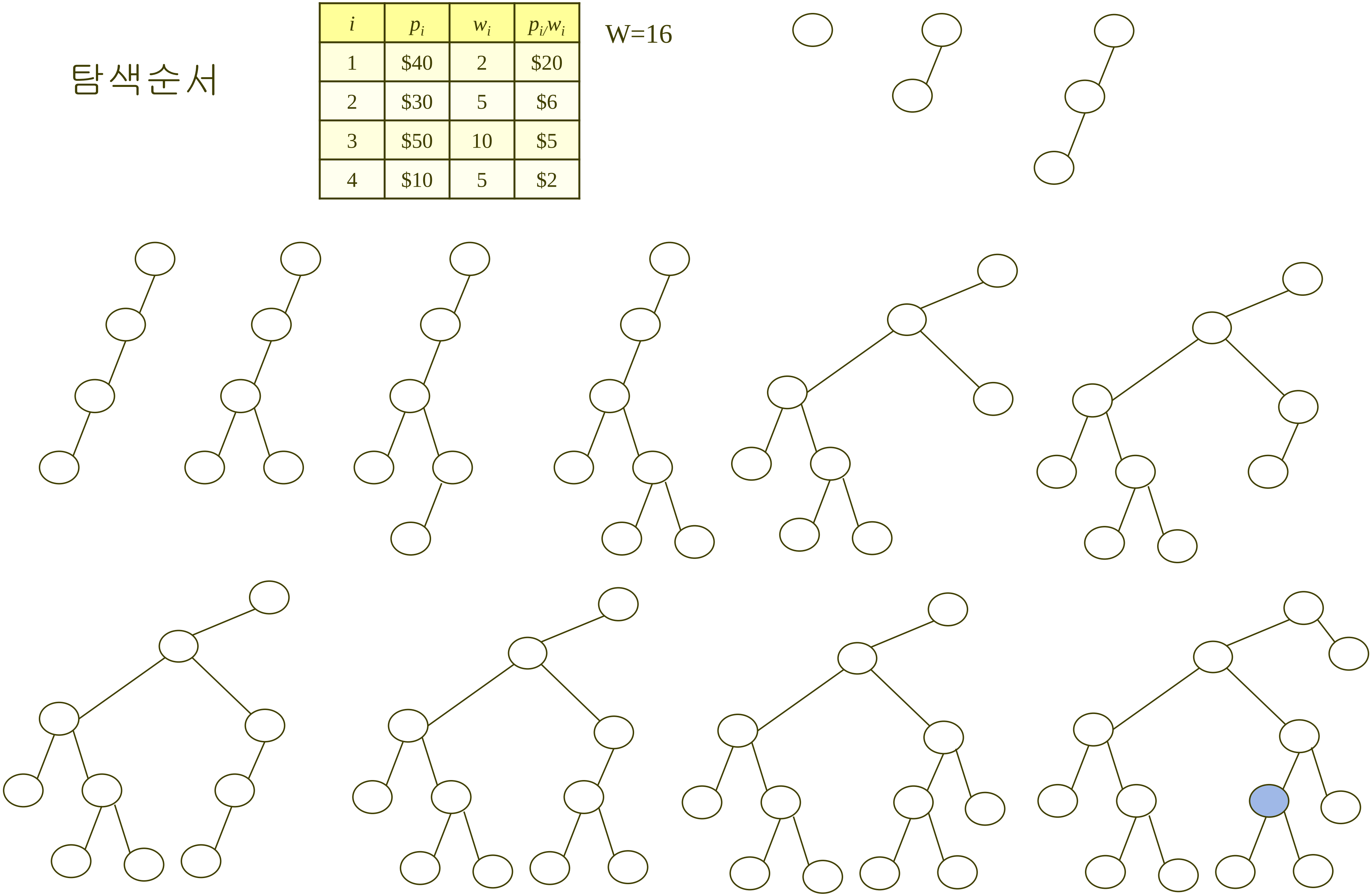


- 탐색 노드수 13개

탐색순서

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

W=16



- 탐색 노드수 13개

분기한정 가지치기로 깊이우선검색(Depth-First Search with Branch-and-Bound)

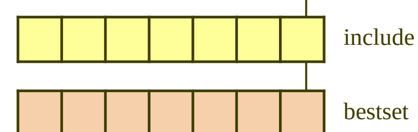
- n개의 아이템. 양의 정수 W. 배열 w와 p, 인덱스는 1부터 n.
  각 배열은 $p[i]/w[i]$ 값의 내림차순으로 정렬
- 출력: 배열 bestset, 1부터 n. bestset[i]의 값은 i번째 아이템이 최적의 해에 포함되어 있으면 Y, 아니면 N.

```
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);
    }
}
```

자신이 유효한지, 지금까지의 최대 값보다 큰지

마지막으로 포함되는 아이템

배열간의 복사



i+1 포함

남은 공간이 있고 확장 시 bound 값이 현재 최대값 보다 크면

i+1 불포함

```
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;
    }
}
```

남은 공간이 있고 확장 시 bound 값이 현재 최대값 보다 크면 true

남은 공간 확인

아직 안 채운 item이 있으면

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

$$bound = \left(profit + \sum_{j=i+1}^{k-1} p_j \right) + (W - totweight) \times \frac{p_k}{w_k}$$

- ✓ 자신이 유효한지, 그리고 확장이 가능한지 확인
- ✓ 초기 호출: numbest=0, maxprofit=0, knapsack(0,0,0);

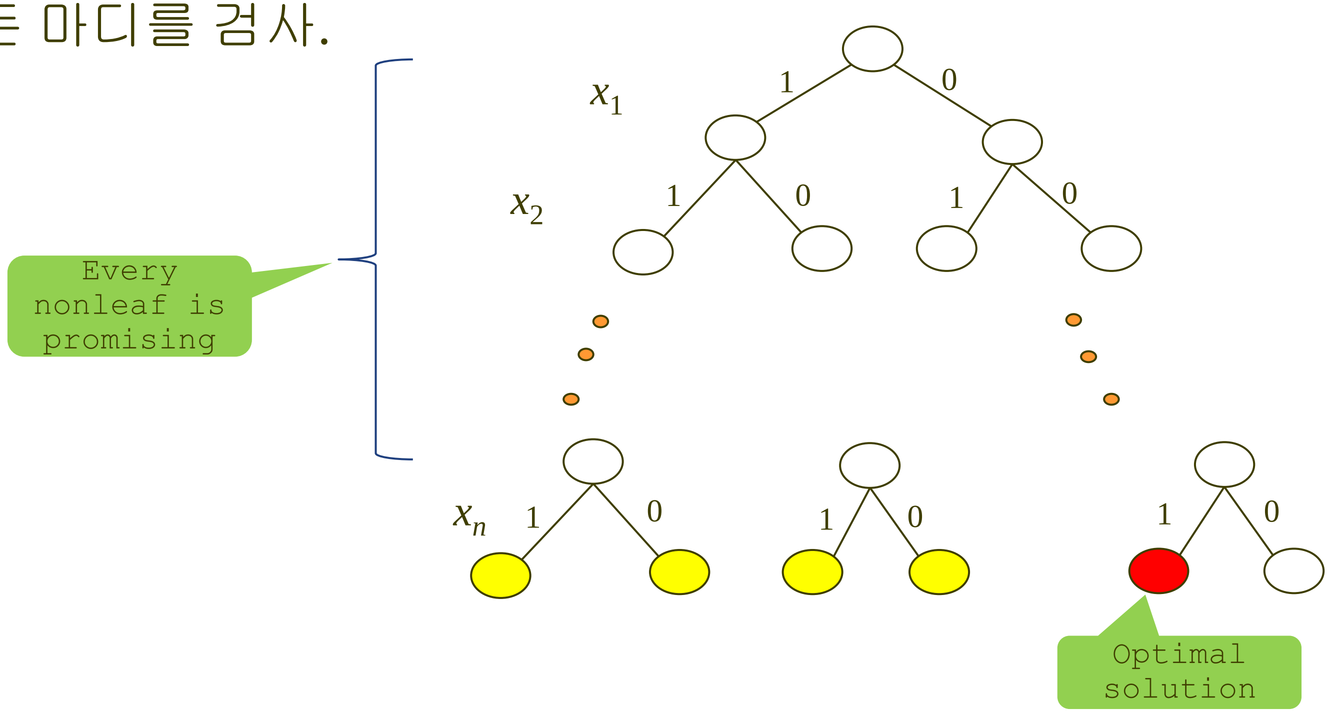
```
numbest=0;
maxprofit=0;
knapsack(0,0,0);
cout << maxprofit;
for (j=1; j<= numbest; j++)
   cout << bestset[j];
```

0-1 배낭채우기 알고리즘: 분석

- 이 알고리즘이 점검하는 마디의 수는 $2^{n+1} - 1 = \Theta(2^n)$이다.

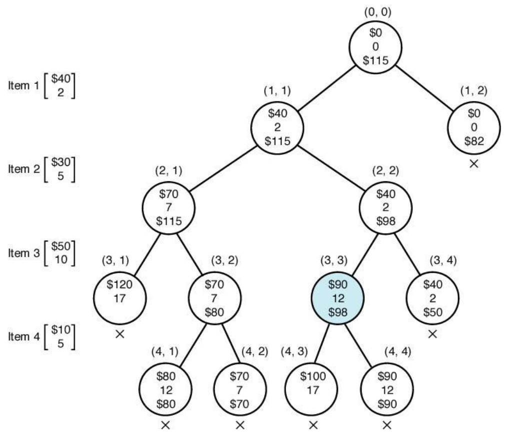
$W=n$, $p_i=1$, $w_i=1$, $(1 \leq i \leq n\text{-}1)$, $p_n=n$, $w_n=n$ 이면

- the optimal solution: $x_1=x_2=\ldots=x_{n\text{-}1}=0$, $x_n=1$. 즉 n번째 아이템을 선택하는 것이 해답.
- 모든 마디를 검사.



0-1 배낭채우기 알고리즘: 분석

- 위 보기의 경우의 분석: 점검한 마디는 13개이다. 이 알고리즘이 동적계획법으로 설계한 알고리즘 보다 좋은가?
  - ✓ 확실하게 대답하기 불가능 하다.
  - ✓ Horowitz와 Sahni(1978)는 Monte Carlo 기법을 사용하여 되추적 알고리즘이 동적계획법 알고리즘 보다 일반적으로 더 빠르다는 것을 입증하였다.
  - ✓ Horowitz와 Sahni(1974)가 분할정복과 동적계획법을 적절히 조화하여 개발한 알고리즘은 $O(2^{n/2})$의 시간복잡도를 가지는데, 이 알고리즘은 되추적 알고리즘 보다 일반적으로 빠르다고 한다.



분기한정 가지치기로 너비우선검색

- 너비우선검색(breadth-first search)순서:

  (1) 뿌리마디를 먼저 검색한다.

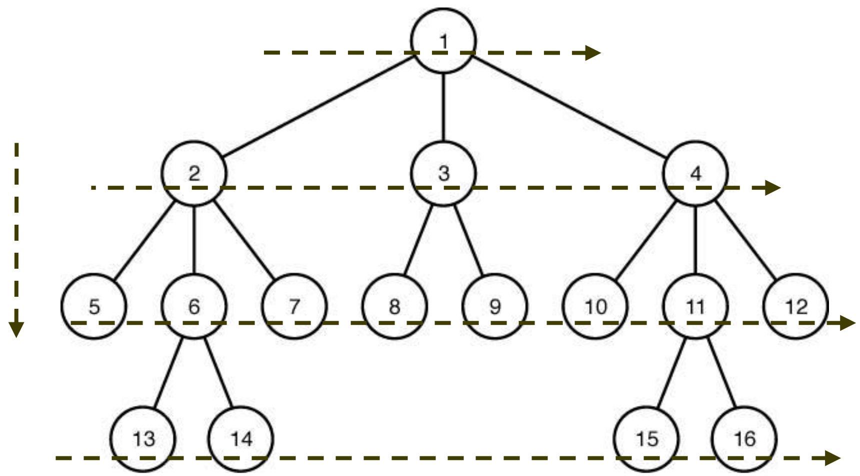
  (2) 다음에 수준 1에 있는 모든 마디를 검색한다.

  (왼쪽에서 오른쪽으로)

  (3) 다음에 수준 2에 있는 모든 마디를 검색한다

  (왼쪽에서 오른쪽으로)

  (4) ...



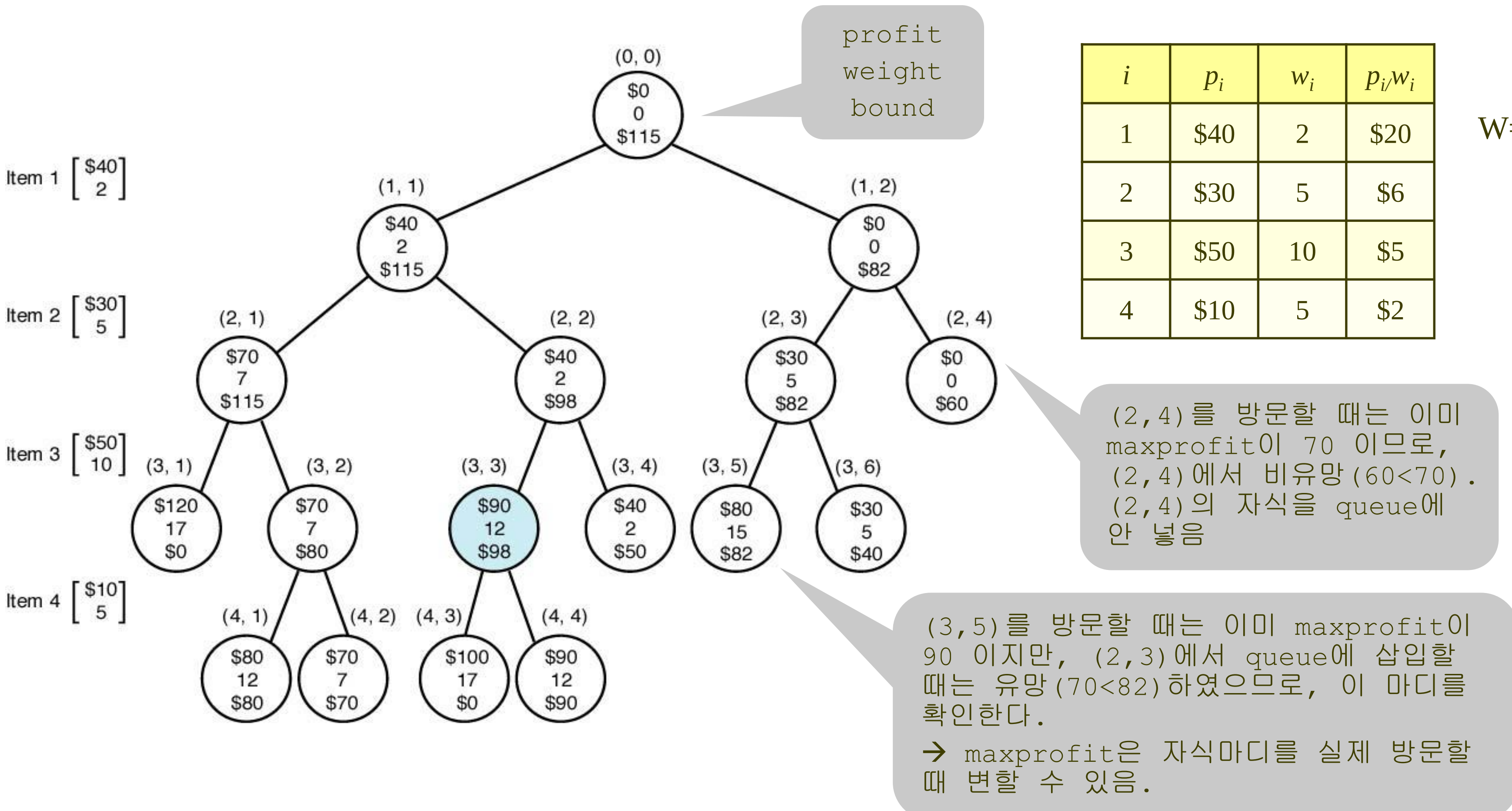
일반적인 너비우선검색 알고리즘

- **대기열(queue)**을 사용하여 구현

```
void breadth_first_search(tree T) {
        queue_of_node Q;
        node u, v;

        initialize(Q);
        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);
          }
        }
    }
```

(예 6.1)

- 앞의 예를 분기한정 가지치기 너비우선검색한 가지친 상태공간트리
- 검색하는 마디의 개수는 17. 깊이우선 알고리즘(13개)보다 좋지 않다!



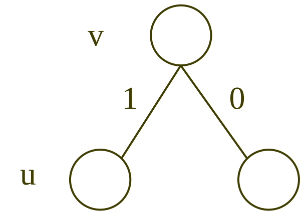
i	p_i	w_i	p_i/w_i
1	$40	2	$20
2	$30	5	$6
3	$50	10	$5
4	$10	5	$2

W=16

분기한정 너비우선검색 알고리즘

```
void breadth_first_branch_and_bound(state_space_tree T, number& best) {
          queue_of_node Q;
          node u, v;

          initialize(Q);                            // Q는 빈 대기열로 초기화
          v = root of T;                            //뿌리마디를 방문
          enqueue(Q,v);
          best = value(v);
          while(!empty(Q)) {
              dequeue(Q,v);
              for(each child u of v) {              // 각 자식마디를 방문
                if(value(u) is better than best)
                  best = value(u);
                if(bound(u) is better than best)
                  enqueue(Q,u);
              }
          }
  }
```

```
void knapsack2(int n, const int p[], const int w[], int W, int& maxprofit){
    queue_of_node Q;
    node u,v;

    initialize(Q);
    v.level=0; v.profit=0; v.weight=0;
    maxprofit=0;
    enqueue(Q,v);
    while(!empty(Q)){
        dequeue(Q,v);
        u.level = v.level + 1;

        u.weight = v.weight + w[u.level];
        u.profit = v.profit + p[u.level];
α {     if(u.weight <= W && u.profit > maxprofit)
              maxprofit = u.profit;
        if(bound(u) > maxprofit)
              enqueue(Q,u);

        u.weight = v.weight;
        u.profit = v.profit;
        if(bound(u) > maxprofit)
             enqueue(Q,u);
        }
}
```



u를 v의 자식마디로 만듦. u를 다음 아이템을 포함하는 자식마디로 놓음

u를 v의 자식마디로 만듦. u를 다음 아이템을 포함하지 않는 자식마디로 놓음. profit과 weight의 변화가 없으므로, α 부분 없음.

- ✓ dequeue(Q,v)후 v가 아직 유망한 지 점검가능. 여기서는 생략
- ✓ 만일 dequeue 후 유망점검이 있다면 (3,5) 이후는 바로 대상이 아닌 것으로 판단가능

```
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];
          result = result + p[j];
          j++;
     }
     k=j;
     if(k<=n)
          result = result + (W-totweight)*p[k]/w[k];

     return result;
   }
}
```

분기한정 가지치기로 최고우선검색 (best-first search)

- 최적의 해답에 더 빨리 도달하기 위한 전략:

    1. 주어진 마디의 모든 자식마디를 검색한 후,

    2. 유망하면서 확장되지 않은(unexpanded) 마디를 살펴보고,

    3. 그 중에서 가장 좋은(최고의) 한계치(bound)를 가진 마디를 확장한다.

- 최고우선검색(Best-First Search)은 너비우선검색에 비해서 좋아짐

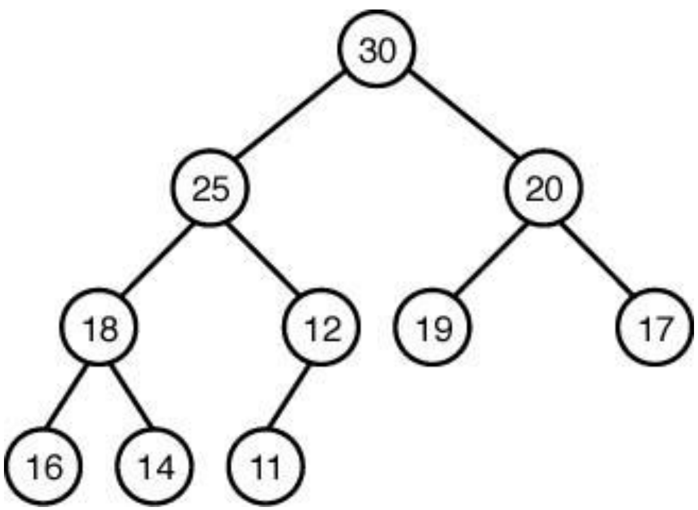
최고우선검색 전략

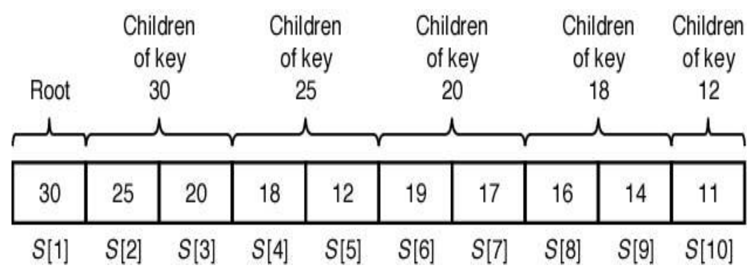
- 최고의 한계를 가진 마디를 우선적으로 선택하기 위해서 우선순위 대기열 (priority queue)을 사용한다.

- 우선순위 대기열은 힙(heap)을 사용하여 효과적으로 구현할 수 있다.

Heap

- A heap is an essentially complete binary tree such that
  - ✓ the values stored at the nodes come from an ordered set.
  - ✓ the values stored at each node is greater than or equal to the values stored at its children. [max heap property]:





array representation of the heap

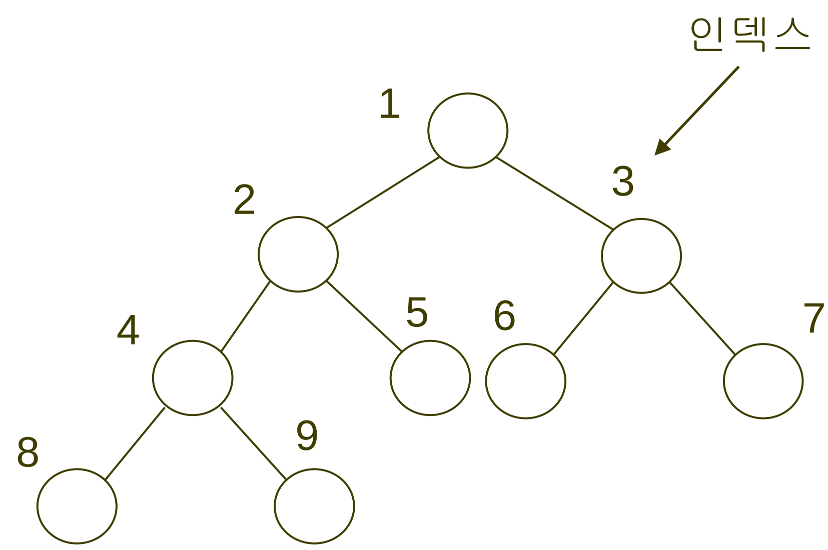
- Characteristics of Heap
  1. Find a Max – O(1)
  2. Remove the Max and rebuilding – O(log n)
  3. Add(delete or modify) a data - O(log n)

- Suitable to maintain a Max of Queue – priority queue

- Interpretation of Heap Structure

  For a node with index i

  - left child index = $2 \times i$
  - right child index = $2 \times i + 1$
  - parents node index = $\lfloor n/2 \rfloor$



분기한정 최고우선검색 알고리즘

```
void best_first_branch_and_bound(state_space_tree T, number& best) {
    priority_queue_of_node PQ;
    node u,v;

    initialize(PQ);                        // PQ를 빈 대기열로 초기화
    v = root of T;
    best = value(v);
    insert(PQ,v);
    while(!empty(PQ)) {                // 최고 한계값을 가진 마디를 제거
      remove(PQ,v);
      if(bound(v) is better than best)     // 마디가 아직 유망한 지 점검
         for(each child u of v) {
            if(value(u) is better than best)
               best = value(u);
            if(bound(u) is better than best)
               insert(PQ,u);
         }
    }
}
```

추가부분

```
void breadth_first_branch_and_bound( -----) {
     = = = =
   while(!empty(Q)) {
      dequeue(Q,v);
      for(each child u of v) {
         if(value(u) is better than best)
            best = value(u);
         if(bound(u) is better than best)
            enqueue(Q,u);
      } }}
```

```
void knapsack3(int  n, const  int p[], const  int w[],
               int W, int& maxprofit){
    priority_queue_of_node PQ;
    node u,v;

    initialize(PQ);
    v.level = 0; v.profit = 0; v.weight = 0;
    maxprofit = 0;
    v.bound = bound(v);
    insert(PQ,v);

    while(!empty(PQ)){
         remove(PQ,v);
         if(v.bound > maxprofit){  /* 마디가 아직 유망한지 확인
           u.level = v.level + 1;
           u.weight = v.weight + w[u.level];
           u.profit = v.profit + p[u.level];
           if(u.weight <= W && u.profit > maxprofit)
               maxprofit = u.profit;
           u.bound = bound(u);
           if(u.bound > maxprofit)
               insert(PQ,u);
           u.weight = v.weight;
           u.profit = v.profit;
           u.bound = bound(u);
```

최고의 한계 값을 가진 마디 추출

u를 다음 아이템을 포함하는 자식마디로 놓음

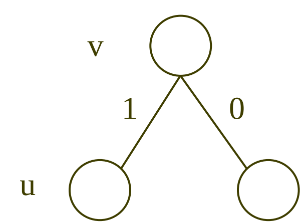
지금까지 발견한 해 중 가장 좋은 것이면

u가 유망하면 PQ에 입력

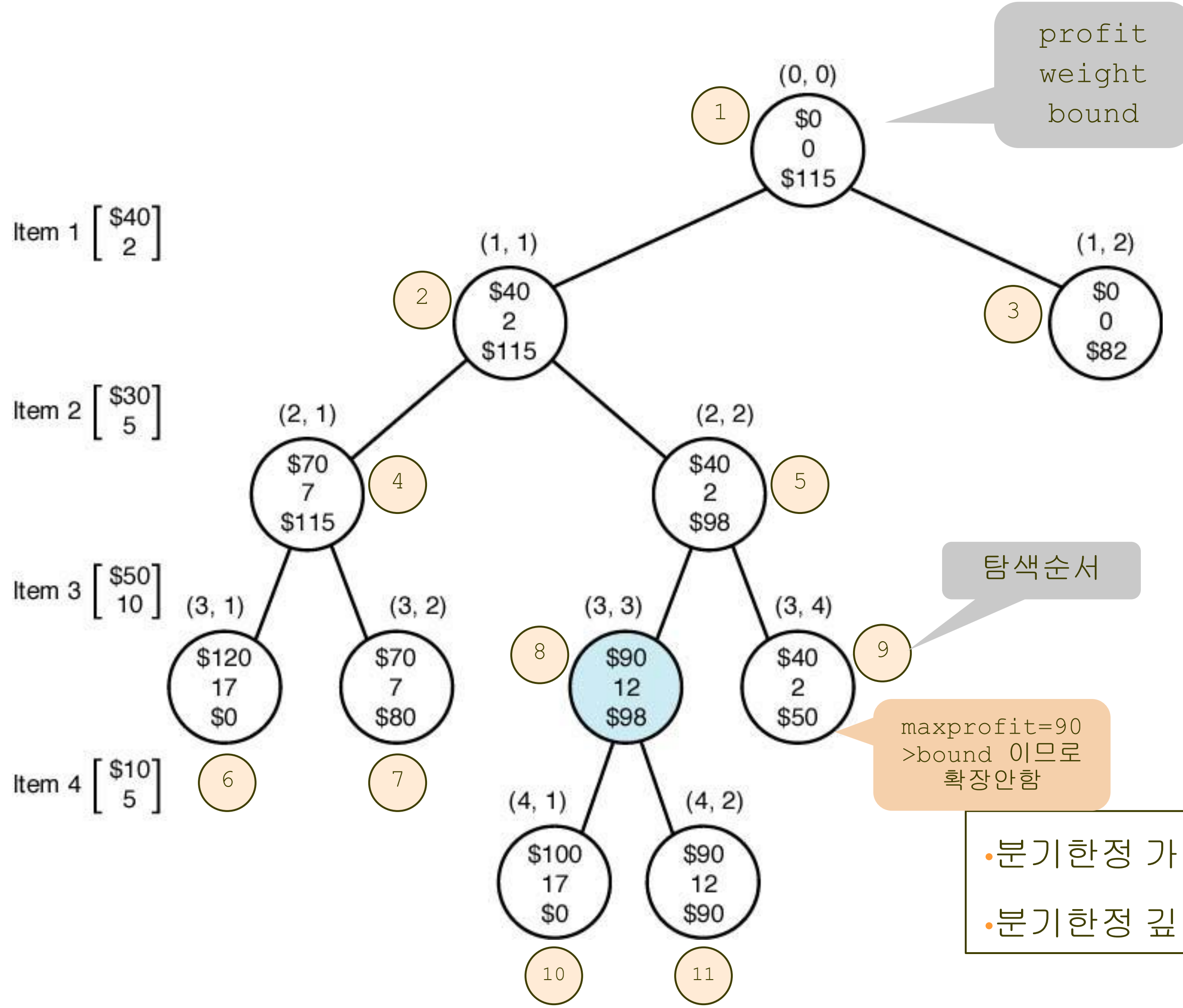
u를 다음 아이템을 포함하지 않는 자식마디로 놓음

u가 유망하면 PQ에 입력

```
           if(u.bound > maxprofit)
               insert(PQ,u);
         } // if
       } // while
}


float bound(node u)
{
     same as before
}
```



•(그림 6.3) 분기한정 가지치기로 최고우선검색을 하여 가지친 상태 공간트리. 검색하는 마디는 **11개**.



i	p_i	w_i	p_i/w_i
1	$40	2	$20
2	$30	5	$6
3	$50	10	$5
4	$10	5	$2

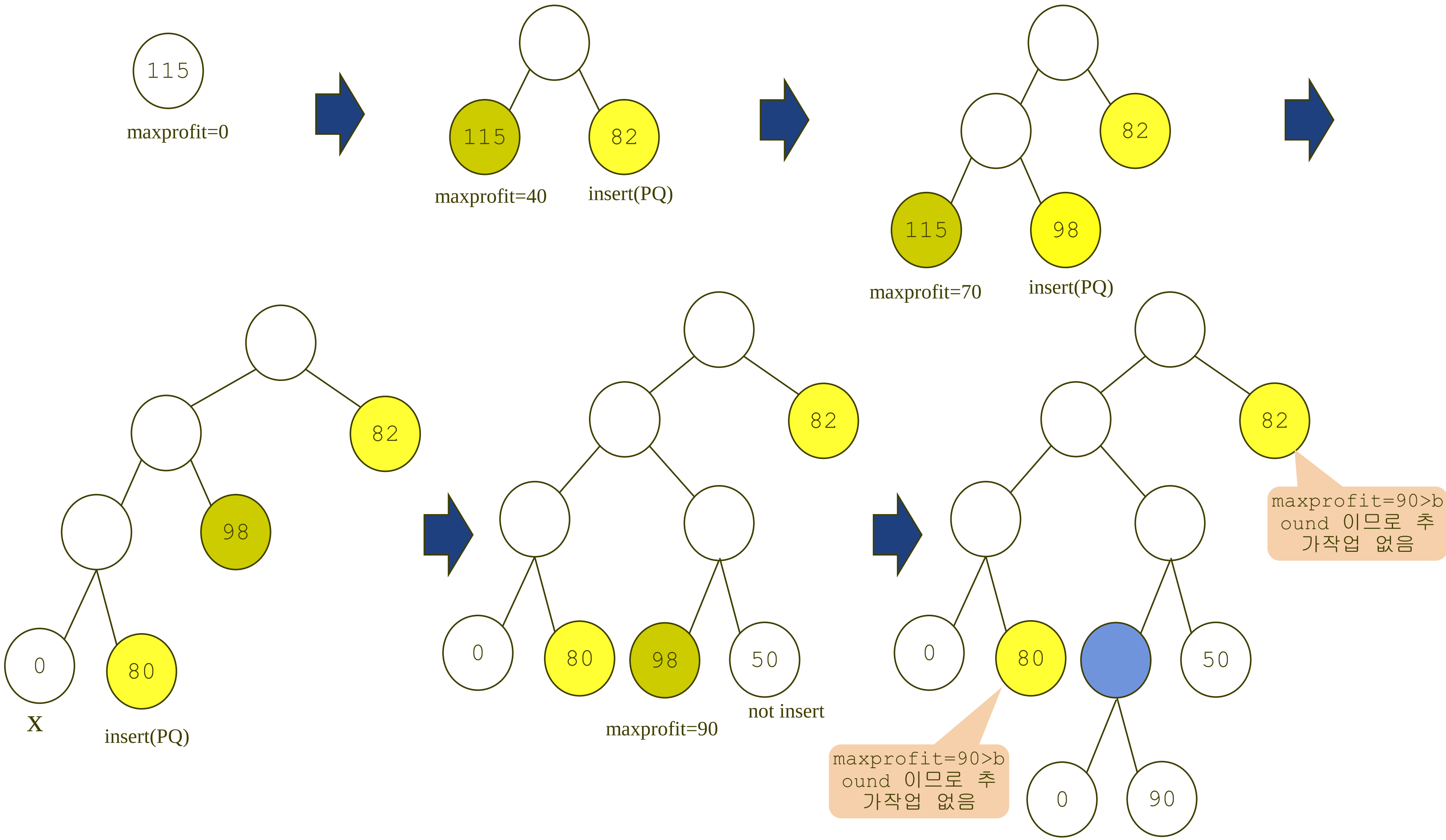
W=16

- 분기한정 가지치기 너비우선:17개
- 분기한정 깊이우선: 13개

탐색순서

- bound는 노드 내 표시

- 분기한정 가지치기 너비우선:17개
- 분기한정 깊이우선: 13개



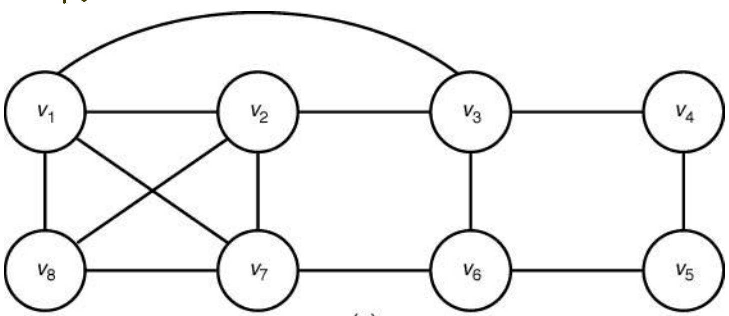
최종적으로 PQ 내에 82,80 이 순차적으로 remove(PQ)에 의해 확인되지만, maxprofit>bound 이므로, 추가 작업 없음

외판원 문제
(Traveling Salesman(person) Problem)

- 하나의 노드에서 출발하여 다른 노드들을 각각 한번씩 만 방문하고, 다시 출발노드로 돌아오는 가장 짧은 일주여행경로(tour)를 결정하는 문제
- 이 문제는 음이 아닌 가중치가 있는, 방향성 그래프로 나타낼 수 있다.
- 그래프 상에서 일주여행경로(tour, Hamiltonian circuit)는 한 정점을 출발하여 다른 모든 정점을 한번씩 만 거쳐서 다시 그 정점으로 돌아오는 경로.
- 여러 개의 일주여행경로 중에서 길이가 최소가 되는 경로가 최적일주여행경로(optimal tour)가 된다.
- 무작정 알고리즘: 가능한 모든 일주여행경로를 다 고려한 후, 그 중에서 가장 짧은 일주여행경로를 선택한다. 가능한 일주여행경로의 총 개수는 $(n - 1)!$.

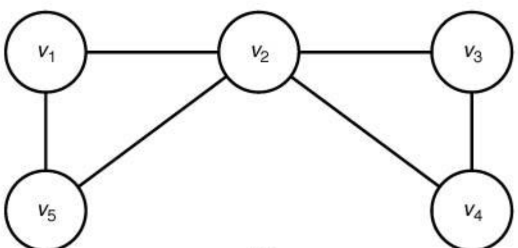
해밀토니안 회로

- 연결된 비방향성 그래프에서, 해밀토니안 회로(Hamiltonian circuits, (cycle))/ 일주여행경로(tour)는 어떤 한 마디에서 출발하여 그래프 상의 각 정점을 한번씩 만 경유하여 다시 출발한 정점으로 돌아오는 경로이다.



(a)

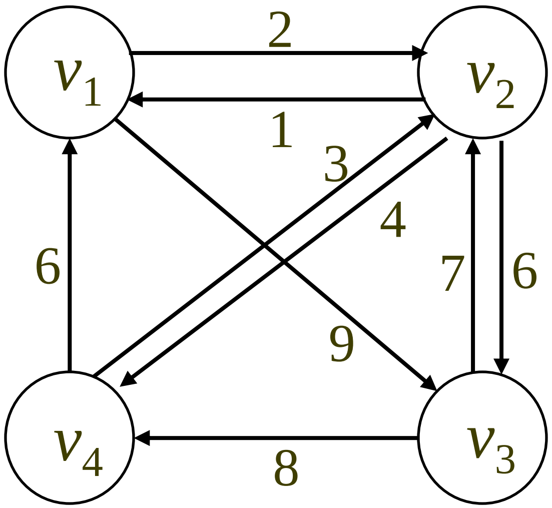
1-2-8-7-6-5-4-3-1　HC 존재



(b)

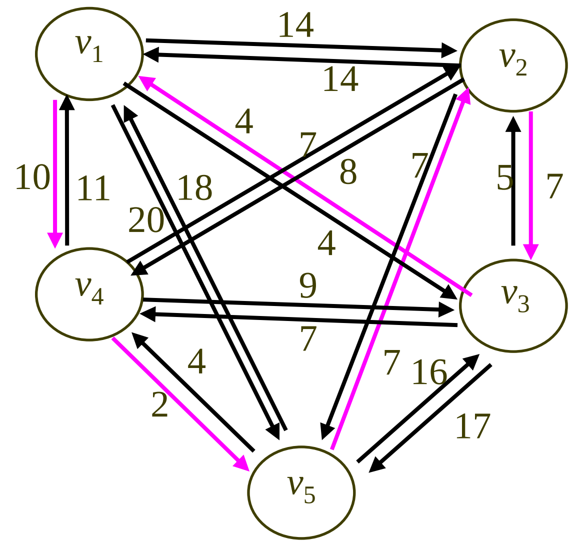
HC 없음

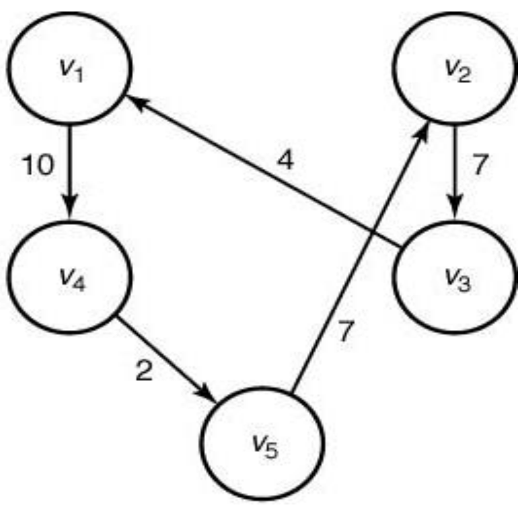
(예) 가장 최적이 되는 일주여행경로는?



(그림 6.4) 모든 노드간에 에지가 있는 그래프의 인접행렬 표현

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$





TSP의 최적해

외판원문제: 분기한정법

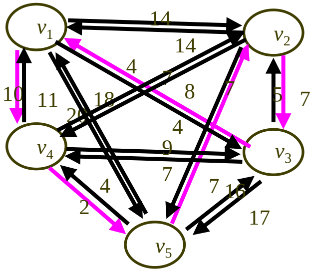
- 동적계획법 알고리즘의 복잡도는 $(n-1)(n-2)2^{n-3} \in \Theta(n^2 2^n)$
- 동적계획법 알고리즘의 기본동작을 수행하는데 걸리는 시간을 1μsec이라고 할 때,

✓ n = 20일 때, T(20) = (20 - 1)(20 - 2)2^{20-3}μsec = 45초

✓ n = 40일 때, T(40) = (40 - 1)(40 - 2)2^{40-3}μsec > 6년 이상.

→ 분기한정법을 시도.

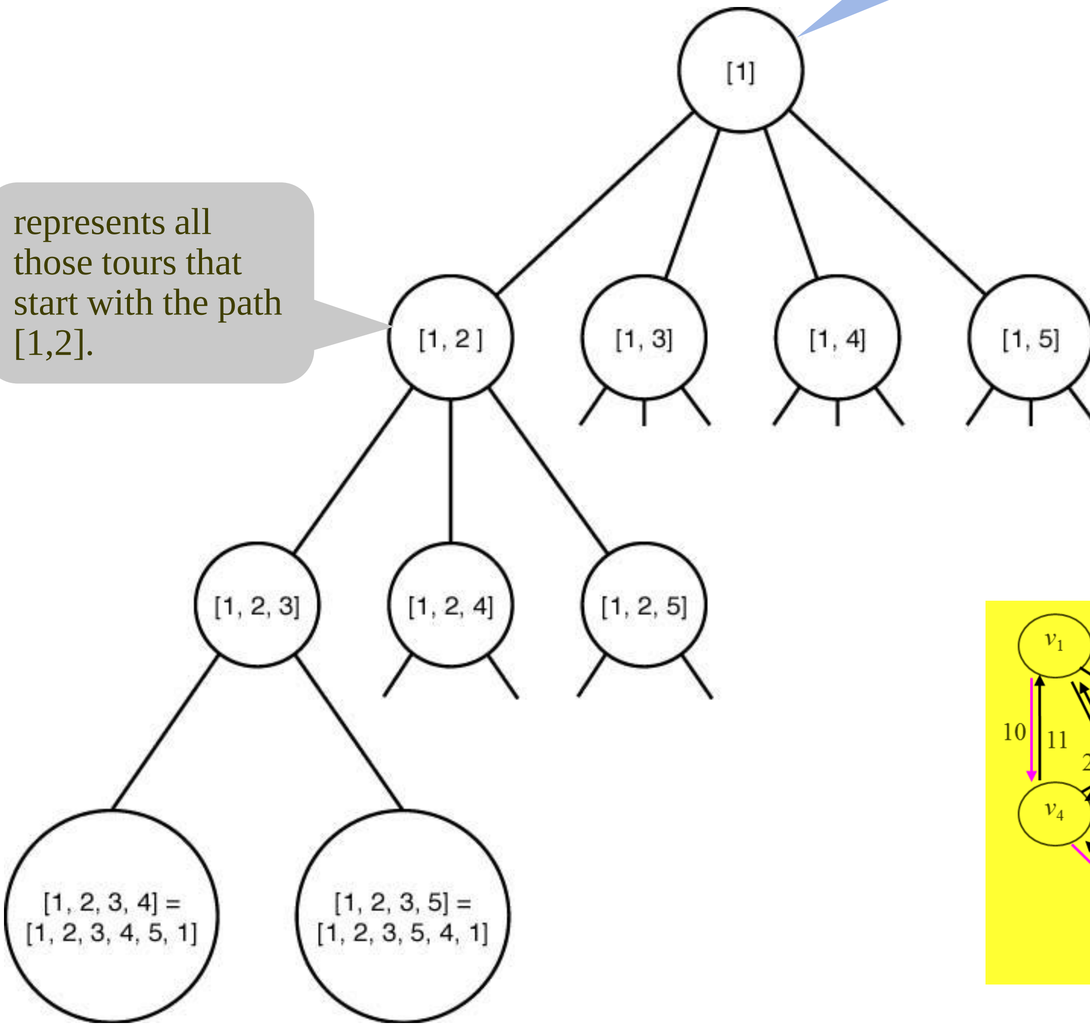
상태공간트리 구축방법



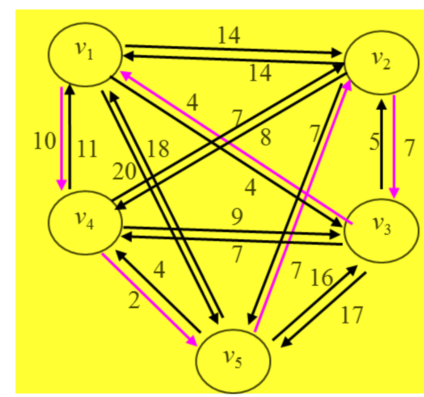
- 각 마디는 출발마디로부터의 일주여행경로를 나타냄
- 뿌리마디의 여행경로는 [1]이 되고, 뿌리마디에서 뻗어 나가는 수준 1에 있는 여행경로는 각각 [1,2], [1,3], …, [1,5]가 되고, 마디 [1,2]에서 뻗어 나가는 수준 2에 있는 마디들의 여행경로는 각각 [1,2,3],…,[1,2,5]가 되고, 이런 식으로 뻗어 나가서 잎마디에 도달하게 되면 완전한 일주여행경로를 가지게 된다.
- 따라서 최적일주여행경로를 구하기 위해서는 잎마디에 있는 일주여행경로를 모두 검사하여 그 중에서 가장 길이가 짧은 일주여행경로를 찾으면 된다.
- 참고: 위 예에서 각 마디에 저장되어 있는 마디가 4개가 되면 더 이상 뻗어 나갈 필요가 없다. 5번째 마디는 자동 결정.

Root = start

represents all those tours that start with the path [1,2].

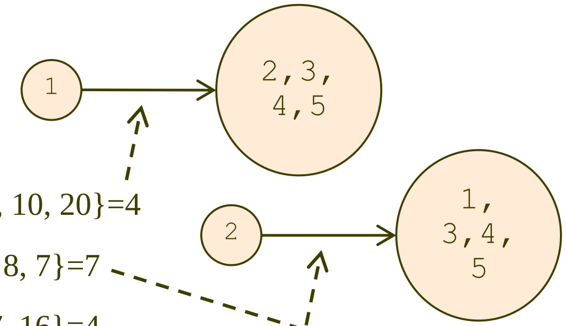


Leaf is a tour, solution



- 한계값 구하기
  - ✓ 초기

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



v1에서 떠나는 비용의 하한= min{14, 4, 10, 20}=4

v2에서 떠나는 비용의 하한= min{14, 7, 8, 7}=7

v3에서 떠나는 비용의 하한= min{4, 5, 7, 16}=4

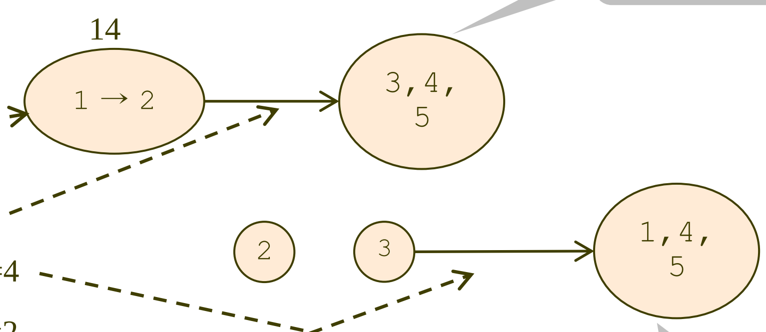
v4에서 떠나는 비용의 하한= min{11, 7, 9, 2}=2

v5에서 떠나는 비용의 하한= min{18, 7, 17, 4}=4

일주여행경로의 하한=4+7+4+2+4=21

1은 직전 방문노드 이므로 바로 갈 수는 없다

  - ✓ [v1→v2] 경로가 결정된 경우



[v1 → v2] 경로 길이 = 14

v2에서 떠나는 비용의 하한= min{7, 8, 7}=7

노드 4,5,1,로 떠날 수 있다

v3에서 떠나는 비용의 하한= min{4, 7, 16}=4

v4에서 떠나는 비용의 하한= min{11, 9, 2}=2

v5에서 떠나는 비용의 하한= min{18, 17, 4}=4

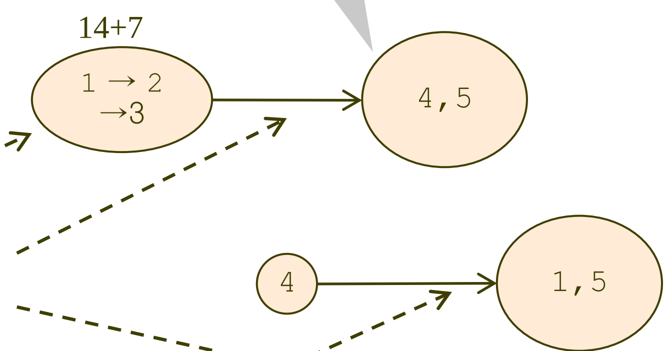
[v1,v2]경로를 포함하는 일주여행경로의 하한=14+7+4+2+4=31

2는 이미 방문한 노드 이므로 고려하지 않는다.

✓ [v1 → v2 → v3] 경로가 결정된 경우

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

3은 직전 방문노드 이므로 바로 갈 수 는 없다



[v1 → v2 → v3] 경로 길이 = 14+7

v3에서 떠나는 비용의 하한= min{7, 16}=7

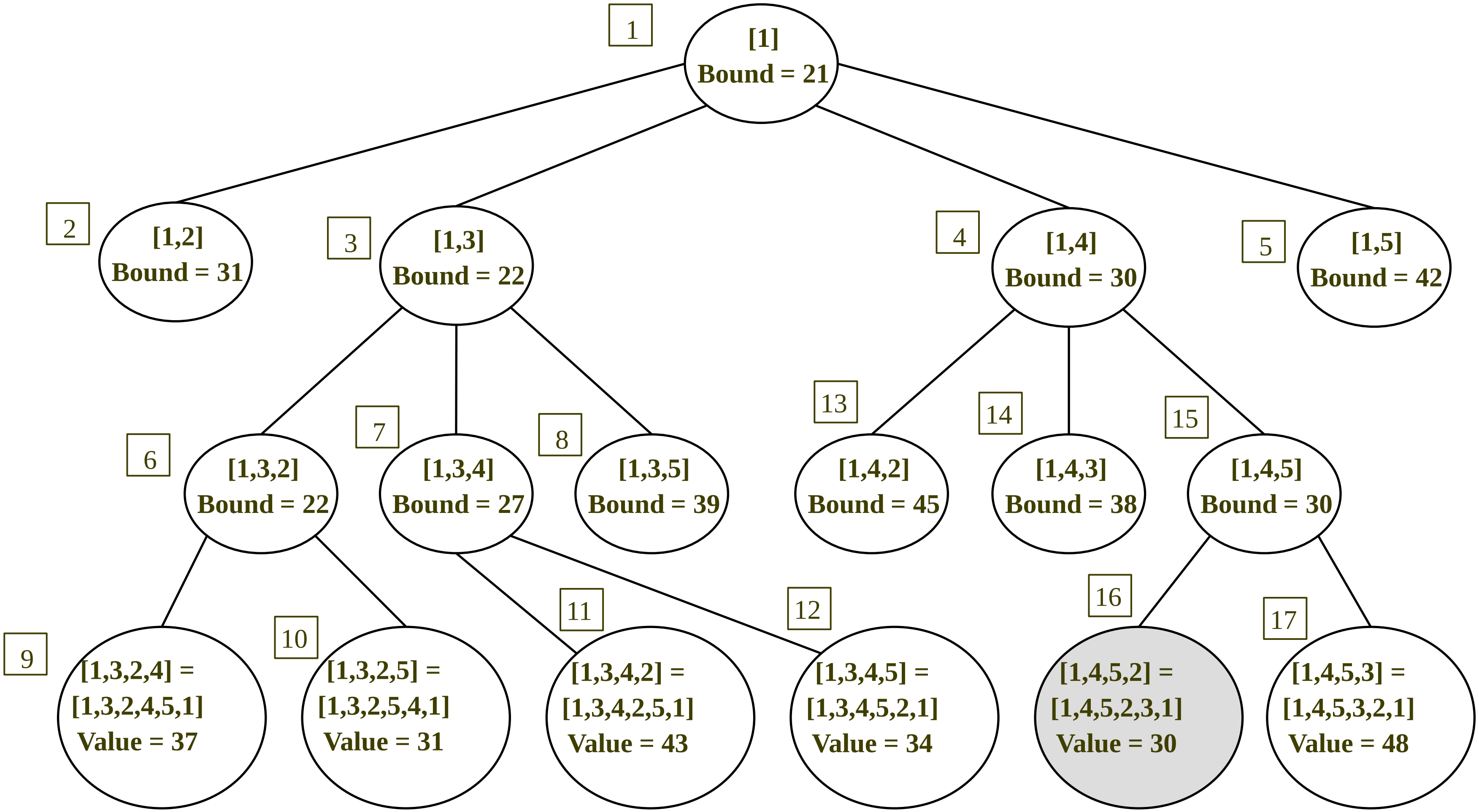
v4에서 떠나는 비용의 하한= min{11, 2}=2

v5에서 떠나는 비용의 하한= min{18, 4}=4

[v1 → v2 →v3] 경로를 포함하는 일주여행경로의 하한=14+7+7+2+4=34

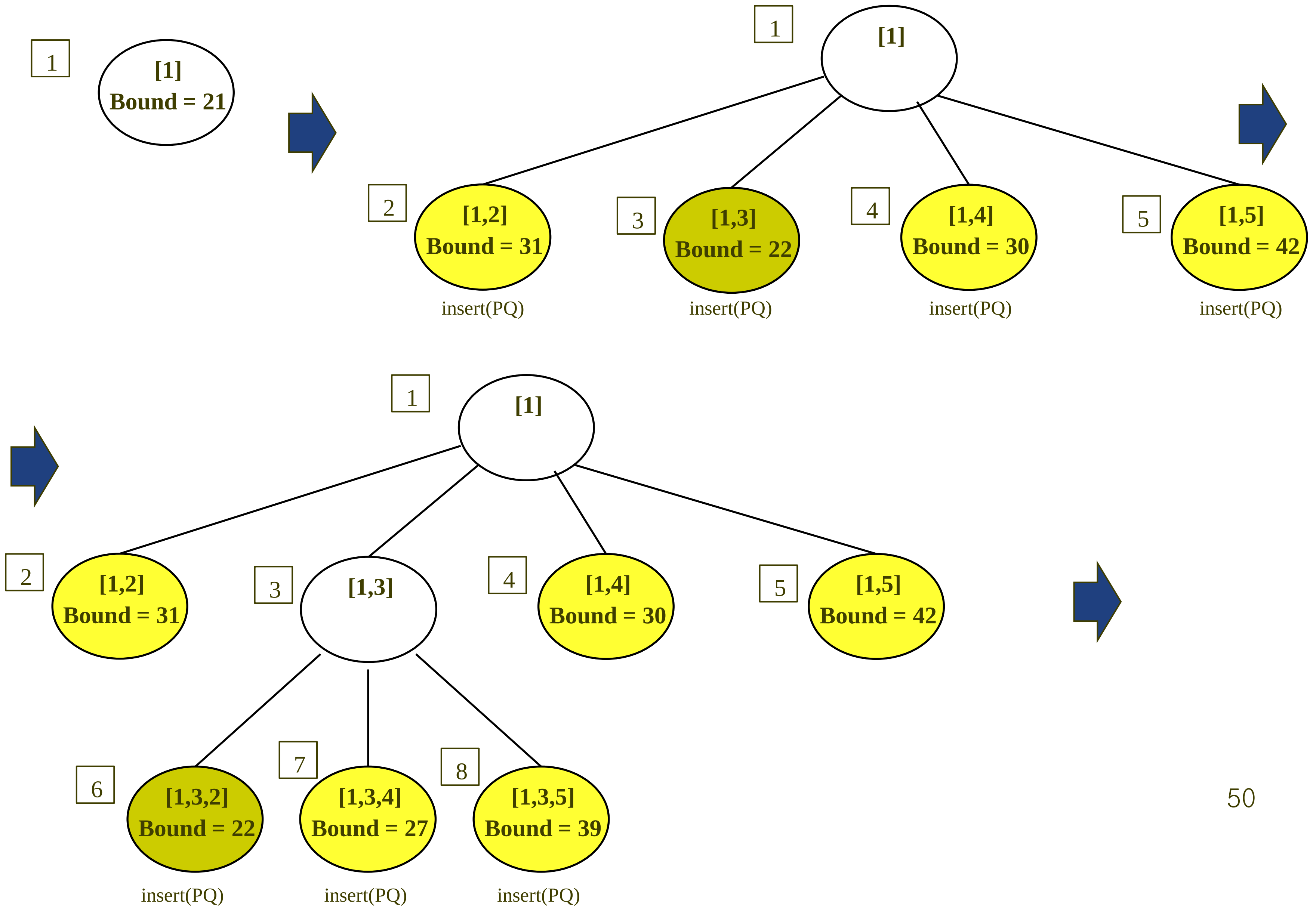
2와 3은 이미 방문 한 노드 이므로 고려 하지 않는다.

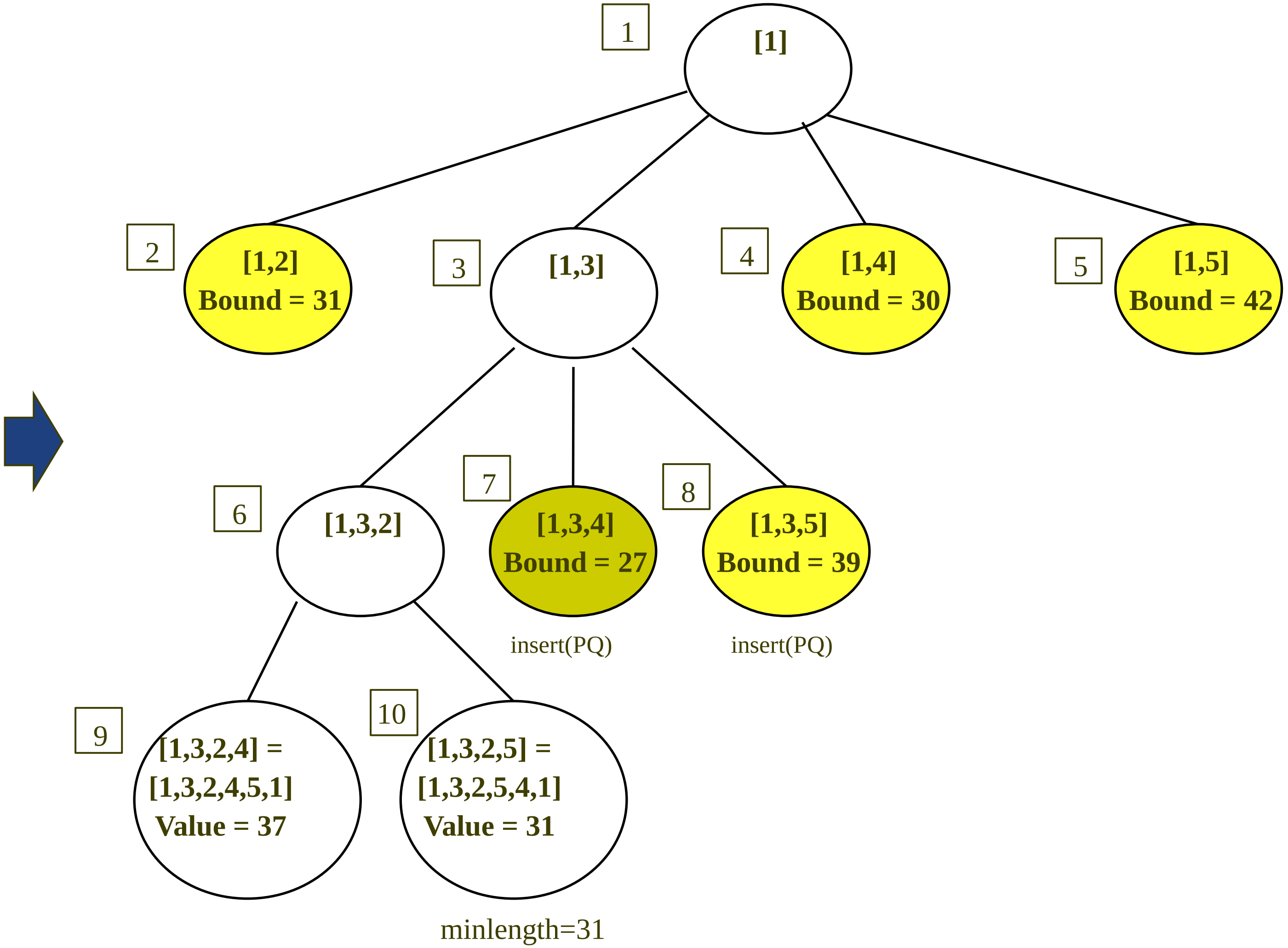
- 분기한정 가지치기로 최고우선검색을 하여 상태공간트리를 구축

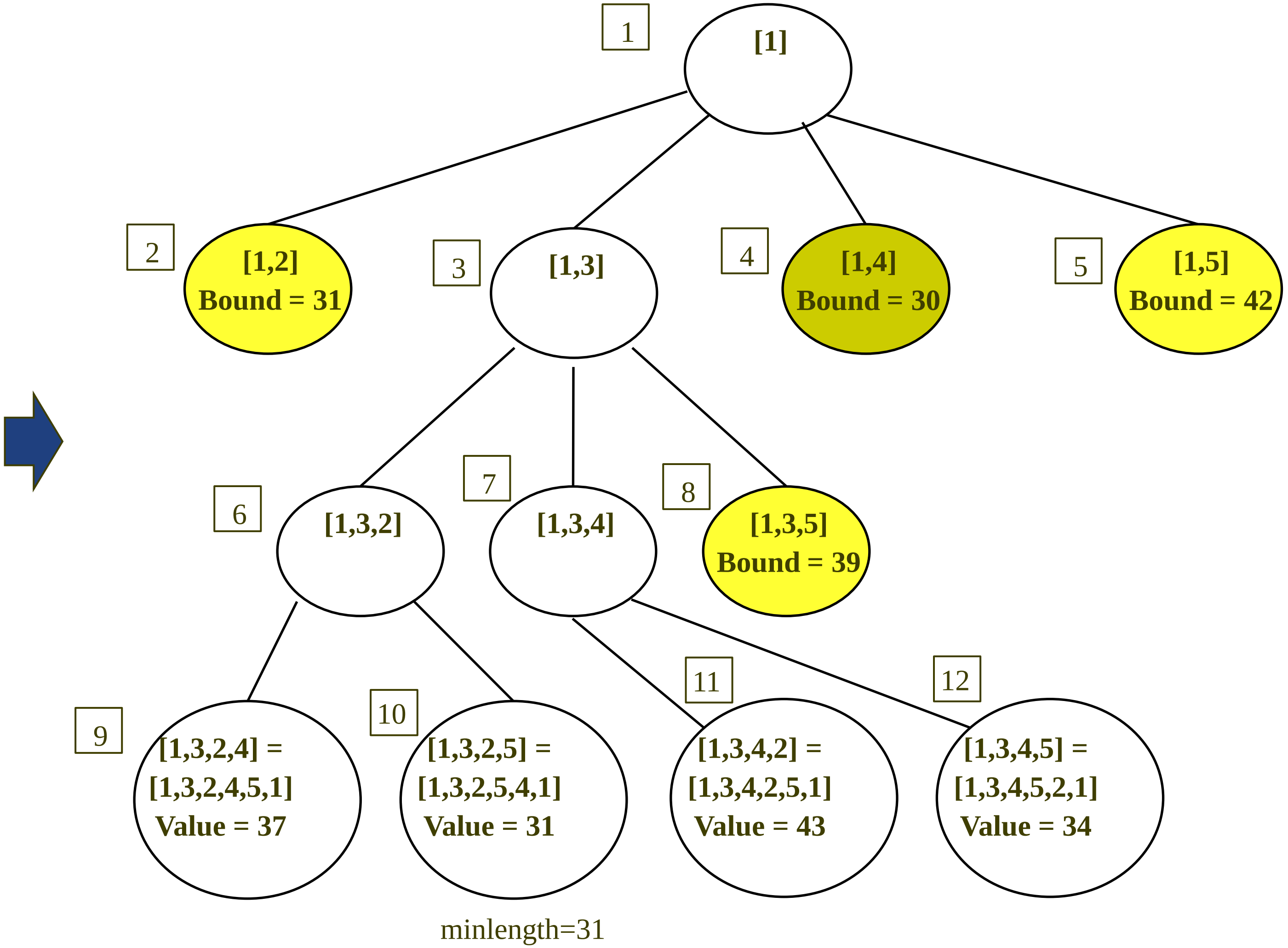


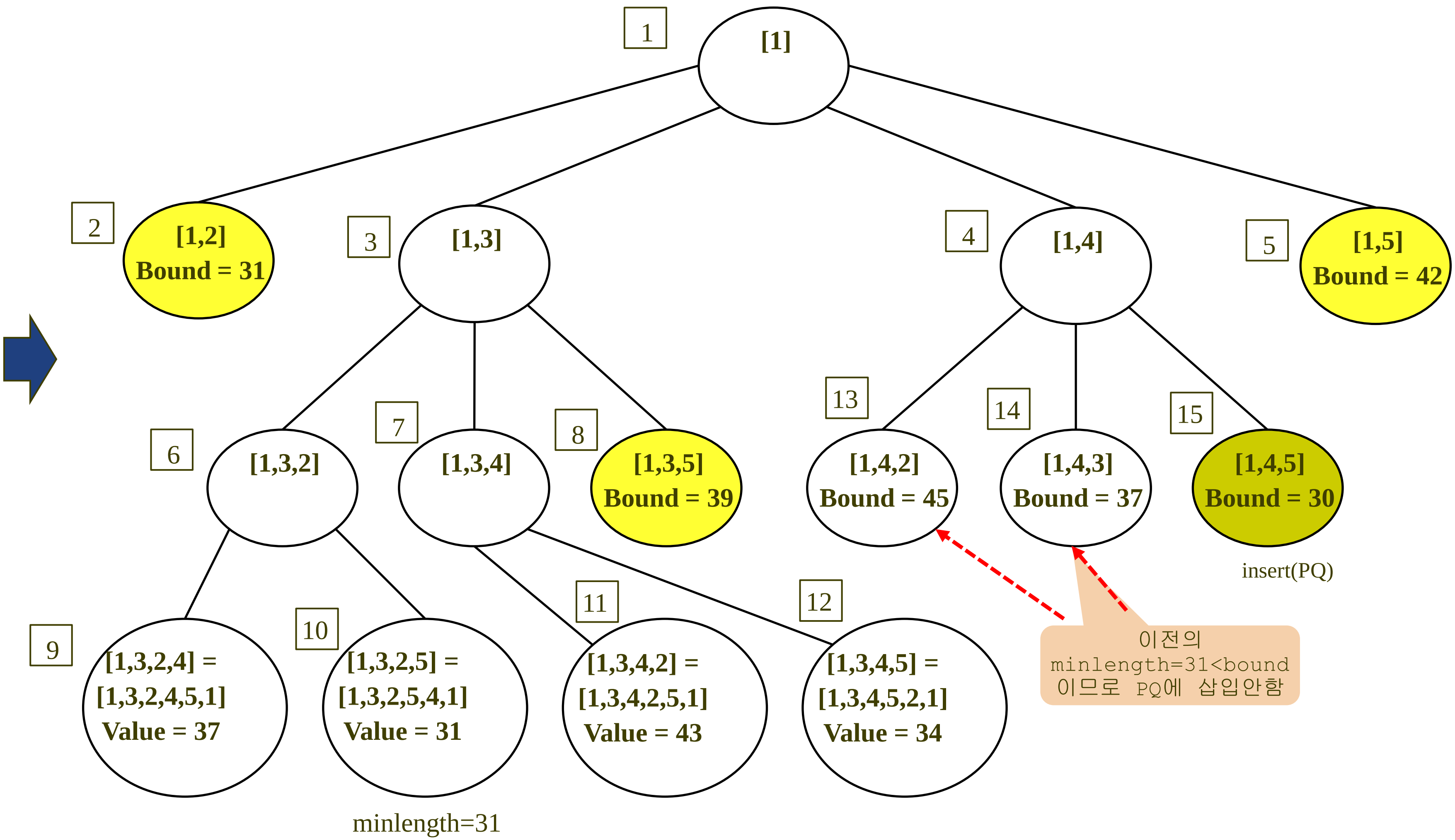
- 총 방문노드 수: 17개
- 생성 가능한 노드 수: 1+4+4×3+4×3×2=41

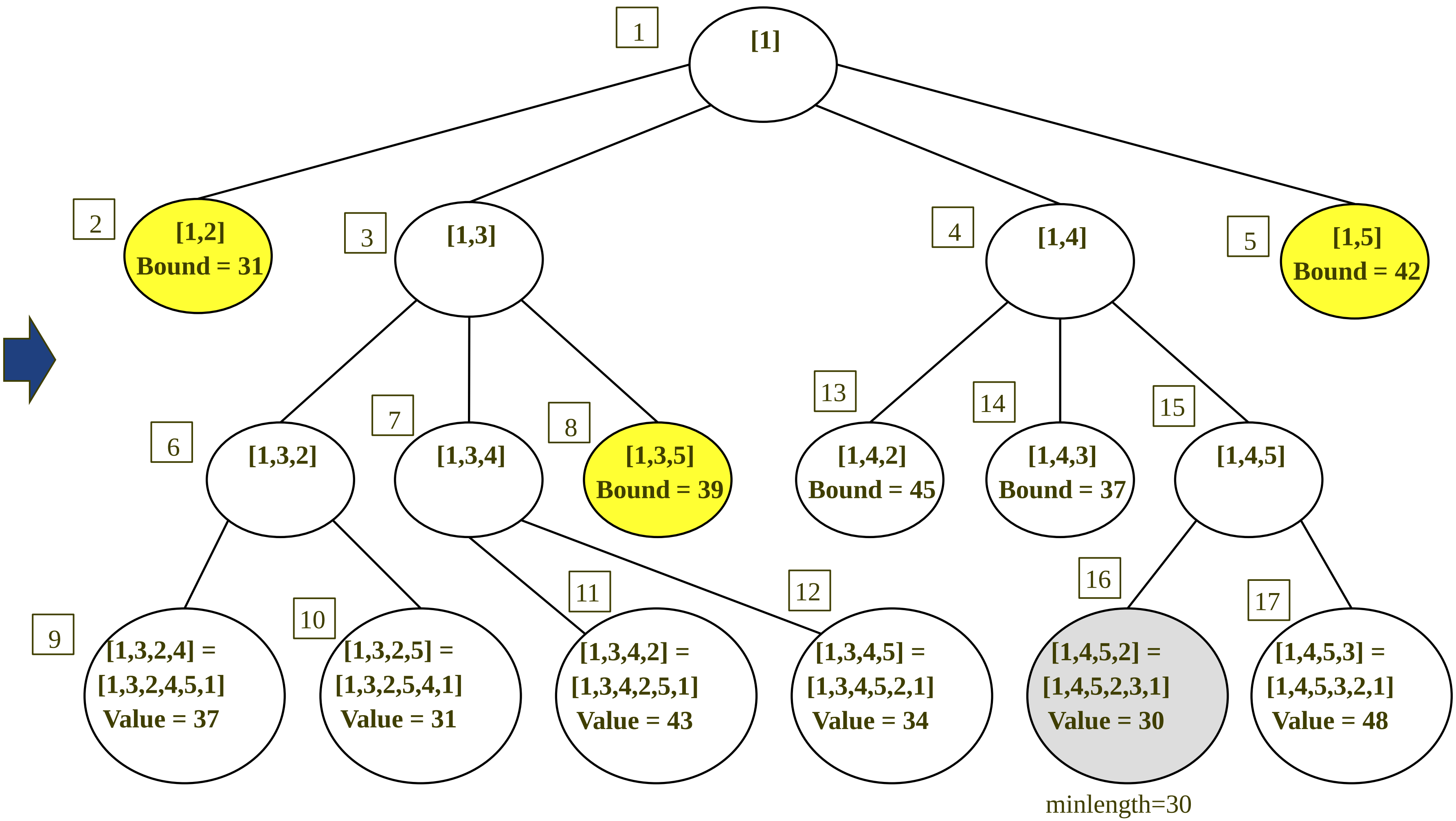
탐색순서


1
[1]
Bound = 21
1
[1]
2
[1,2]
Bound = 31
insert(PQ)
3
[1,3]
Bound = 22
insert(PQ)
4
[1,4]
Bound = 30
insert(PQ)
5
[1,5]
Bound = 42
insert(PQ)
1
[1]
2
[1,2]
Bound = 31
3
[1,3]
4
[1,4]
Bound = 30
5
[1,5]
Bound = 42
6
[1,3,2]
Bound = 22
insert(PQ)
7
[1,3,4]
Bound = 27
insert(PQ)
8
[1,3,5]
Bound = 39
insert(PQ)


1
[1]
2
[1,2]
Bound = 31
3
[1,3]
4
[1,4]
Bound = 30
5
[1,5]
Bound = 42
6
[1,3,2]
7
[1,3,4]
Bound = 27
8
[1,3,5]
Bound = 39
insert(PQ)
insert(PQ)
9
[1,3,2,4] =
[1,3,2,4,5,1]
Value = 37
10
[1,3,2,5] =
[1,3,2,5,4,1]
Value = 31
minlength=31


1
[1]
2
[1,2]
Bound = 31
3
[1,3]
4
[1,4]
Bound = 30
5
[1,5]
Bound = 42
6
[1,3,2]
7
[1,3,4]
8
[1,3,5]
Bound = 39
9
[1,3,2,4] =
[1,3,2,4,5,1]
Value = 37
10
[1,3,2,5] =
[1,3,2,5,4,1]
Value = 31
11
[1,3,4,2] =
[1,3,4,2,5,1]
Value = 43
12
[1,3,4,5] =
[1,3,4,5,2,1]
Value = 34
minlength=31


1
[1]
2
[1,2]
Bound = 31
3
[1,3]
4
[1,4]
5
[1,5]
Bound = 42
6
[1,3,2]
7
[1,3,4]
8
[1,3,5]
Bound = 39
13
[1,4,2]
Bound = 45
14
[1,4,3]
Bound = 37
15
[1,4,5]
Bound = 30
insert(PQ)
9
[1,3,2,4] = [1,3,2,4,5,1]
Value = 37
10
[1,3,2,5] = [1,3,2,5,4,1]
Value = 31
11
[1,3,4,2] = [1,3,4,2,5,1]
Value = 43
12
[1,3,4,5] = [1,3,4,5,2,1]
Value = 34
이전의 minlength=31<bound 이므로 PQ에 삽입안함
minlength=31



최종적으로 PQ 내에 31,39,42가 순차적으로 remove(PQ)에 의해 확인되지만, minlength<bound 이므로, 추가 작업 없음

```
void travel2(int  n, const  number W[][],
            ordered-set& opttour, number& minlength)
{  priority_queue_of_node PQ;
    node u,v;

    initialize(PQ);
    v.level = 0; v.path=[1];
    v.bound = bound(v);
    minlength= ∞;
    insert(PQ,v);

    while(!empty(PQ)){
         remove(PQ,v);
         if(v.bound < minlength){
           u.level=v.level + 1;
           for(all i such that 2≤i≤n && i is not in v.path){
             u.path = v.path;
             put i at the end of u.path;
             if(u.level == n-2){
               put index of only vertex not in u.path at the
end of u.path;
               put 1 at the end of u.path;
               if (length(u) < minlength){
                      minlength = length(u);
                      opttour = u.path;
               }
             }
```

```
 else{
      u.bound = bound(u);
      if(u.bound < minlength)
              insert(PQ,u);
    } /* else*/
   } /* for */
  } /* if */
 } /* while */
}

float bound(node u)
{
    homework
}
```

분석

- 분기한정 가지치기로 최고우선검색 알고리즘의 시간복잡도는 지수적이거나 그보다 못하다!
- 다시 말해서 $n = 40$이 되면 문제를 풀 수 없는 것과 다름없다고 할 수 있다.
- 다른 방법이 있을까?
  - ✓ 근사(approximation) 알고리즘: 최적의 해답을 준다는 보장은 없지만, 무리 없이 최적에 가까운 해답을 주는 알고리즘.


6장 끝
수고하셨습니다.