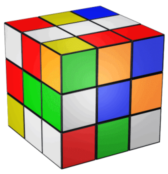
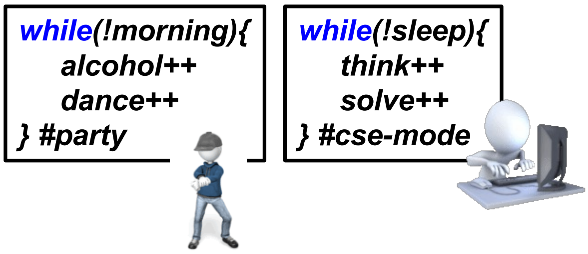
분기한정법

한국기술교육대학교 컴퓨터공학부 김상진





전수조사

- 기본적인 전수조사: 경우의 수가 제한적인 경우에만 사용 가능
  - 보통 재귀 호출을 이용하여 구현
  - 종료조건 중요, 조사 순서 중요 (중복 답 제거 포함)
- 특정 경우만 조사하여 답을 찾을 수 있으면 탐욕적 기법
- 중복 계산이 많음 ⟹ 메모이제이션
- 하향식이 아니라 상향식 ⟹ 테뷸레이션
  - 점화식을 찾는 것이 핵심
  - 최적화 문제는 최적 원칙을 만족해야 함
- 전수조사하지만 노드가 유망하지 않으면 배제 ⟹ 되추적
  - 유망 여부를 검사할 수 있어야 함
  - DFS (트리를 실제 만들어 사용하지 않음)
- 가장 유망한 노드부터 검사 ⟹ 분기 한정 (최적화 문제)
  - 우선순위 큐 기반 BFS (실제 트리를 만들어야 함)

교육목표

- 분기한정법(branch-and-bound)
  - 최적화 문제를 해결하기 위해 되추적 기술을 향상시킨 기법
- 0-1 배낭 채우기 문제
- 외판원 문제
- 작업 할당 문제

분기한정법

- 분기한정법(branch-and-bound): 최적화 문제를 해결하기 위해 되추적 기술을 향상시킨 기법
  - 되추적 기술과 마찬가지로 상태 공간 트리를 사용함
  - 상태 공간 트리를 순회하는 방법이 제한되어 있지 않음
    - 비고. 되추적은 항상 깊이 우선 검색(재귀 호출을 이용한 전수조사)을 함
  - 최적화 문제를 해결하기 위해서만 사용함 (nQueens 문제는 해당하지 않음)
    - 한계치를 계산할 수 있어야 함
- 분기한정 알고리즘은 되추적과 같이 노드를 방문할 때마다 어떤 수를 계산하여 노드의 유망성 여부를 검사함
  - 이 수는 그 노드 아래로 탐색하였을 때 얻을 수 있는 한계치를 나타냄
  - 이 한계치는 지금까지 찾은 최적의 해답보다 좋지 않으면 더 이상 그 노드의 후손들은 검색하지 않음
  - 차이점은 검색하는 노드 순서가 되추적과 달리 가장 유망한 노드부터 검색함
- 상태 공간 트리를 순회하므로 최악의 경우는 지수시간 알고리즘임

0-1 배낭 채우기 문제

- 동적 프로그래밍, 되추적으로 해결하였음
  - 동적 프로그래밍: $O(nW)$
    - 점화식: $V_{i,x} = max(V_{i-1,x}, v_i + V_{i-1,x-w_i})$
  - 되추적: $currWeight + w_i > W$, $bound \leq maxProfit$ 기준을 이용
    - 빈틈없이 채우기 문제를 이용하여 bound 계산
- 분기 한정을 이용한 해결책
  - 한계치를 계산하여 노드의 유망 여부를 검사할 뿐만 아니라 유망 노드의 한계치를 비교하여 가장 유망한 노드부터 검사함
    - 답을 더 빨리 찾으면 가지치기를 더 많이 할 수 있음
  - 이와 같은 알고리즘을 최고 우선 검색 분기 한정 가지치기(best-first search with branch-and-bound pruning)이라 함
    - 이 가지치기는 너비 우선 검색을 수정하여 구현함
    - 이 기법은 우선순위 큐를 이용하여 구현함

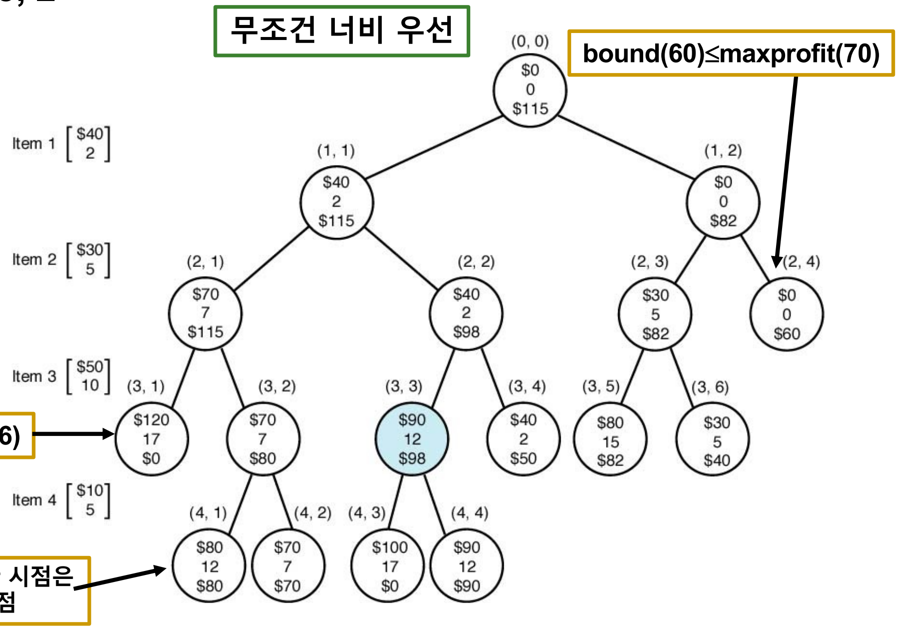
너비 우선 검색 분기 한정 알고리즘 (1/4)

- maxProfit: 지금까지 찾은 가장 최적의 이익
- currWeight: 현 시점에서 배낭에 포함된 물건들의 무게 합
- currProfit: 현 시점에서 배낭에 포함된 물건들의 이익 합
- bound: profit의 상한값 (빈틈없는 배낭 채우기 문제를 통해 계산)
- 노드의 유망여부
  - $bound > maxProfit$
  - $currWeight + w_i \leq W$

너비 우선 검색 분기 한정 알고리즘 (2/4)

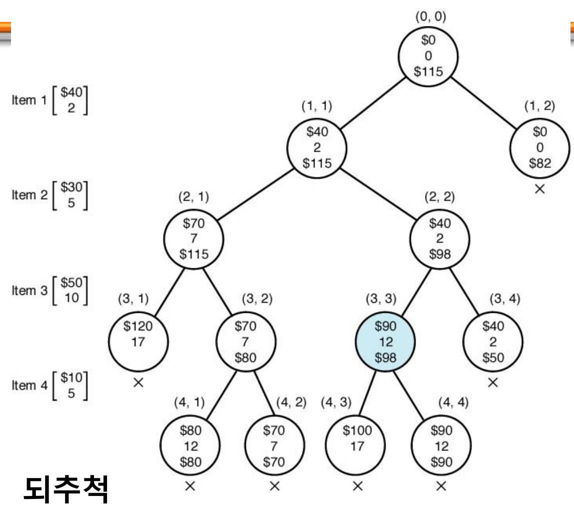
- 예) {(2, 40), (5, 30), (10, 50), (5, 10)}, 16
  - score = 20, 6, 5, 2

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



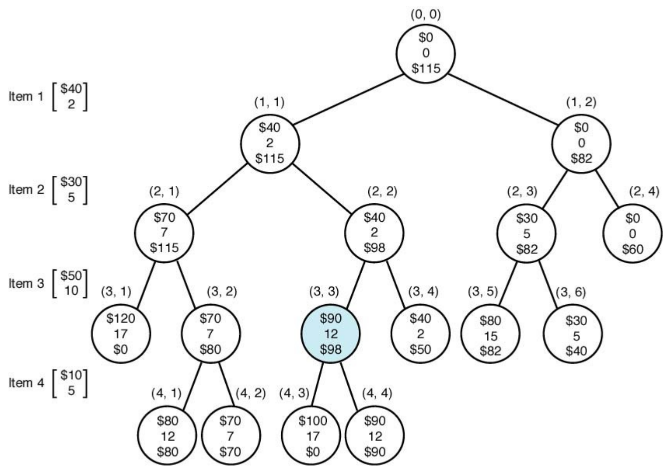
weight(17)>W(16)

주의. 자식을 검사하기로 한 시점은 부모 노드를 방문하였을 시점



되추적

너비 우선 검색 분기 한정



너비우선 검색 분기 한정 알고리즘 (3/4)

- 깊이 우선과 달리 재귀 방식으로 구현하지 않으므로 실제 노드를 만들어 검색해야 함

```
Node{
    level          // 트리의 레벨, item[]의 색인정보
    currProfit     // 지금까지 포함한 물건들의 이익 합
    currWeight     // 지금까지 포함한 물건들의 무게 합
    bound          // 향후 추가할 수 있는 것을 바탕으로 최대 이익
    include[]      // 지금까지 포함한 물건 정보
} // 큐에 포함할 노드 구조체
```


너비 우선 검색 분기 한정 알고리즘 (4/4)

```
knapsack(items[], W)
Q := Node를 유지할 수 있는 queue
root := 루트 노드
Q.push(root)
while Q is not empty do
    node := Q.pop()
    node.level += 1
    if(node.level > n) continue;
    node.currWeight += items[node.level].weight
    node.currProfit += items[node.level].profit
    if node is promising then // items[node.level]을 포함하는 경우
        if node.currProfit > maxProfit then
            maxProfit := node.currProfit
            solution := node.include
        Q.push(node)
    node.currWeight −= items[node.level].weight
    node.currProfit −= items[node.level].profit
    if node is promising then // items[node.level]을 포함하지 않는 경우
        Q.push(node)
```

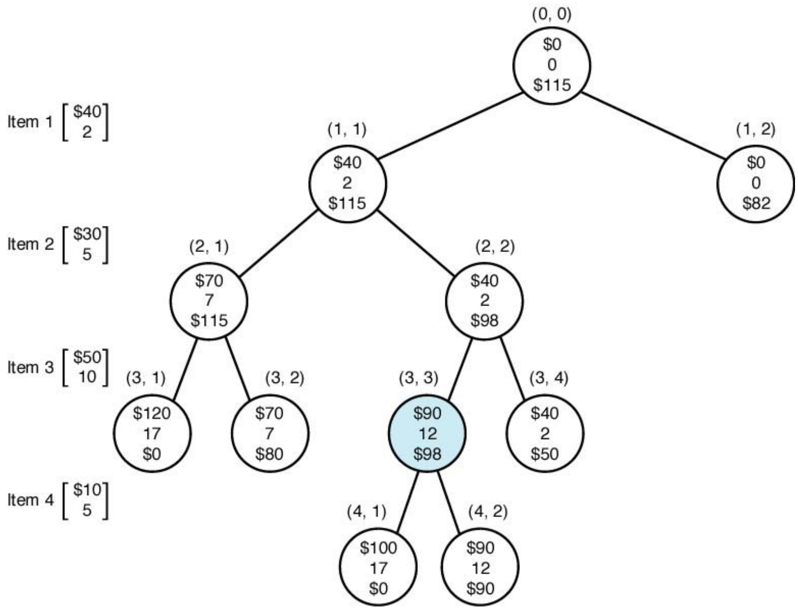
최고 우선 검색 분기 한정 가지치기 (1/3)

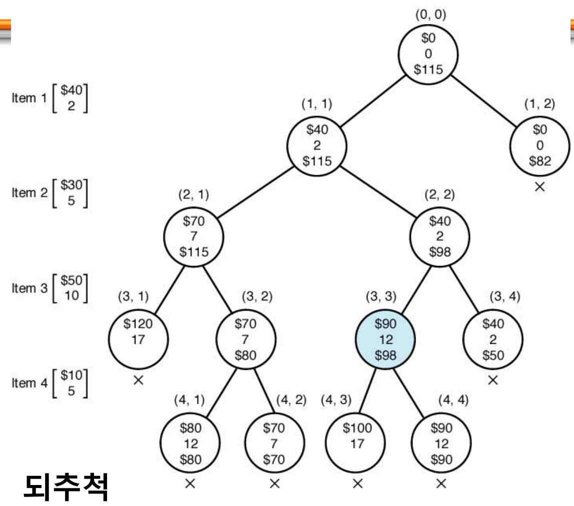
- 일반 너비 우선 분기 한정을 이용한 0-1 배낭 채우기 문제는 기존에 되추적을 이용한 알고리즘에 비해 개선된 점이 별로 없음
- 최고 우선 검색 분기 한정 가지치기
  - 노드의 모든 자식 노드를 검색한 후에 유망하면서 아직 확장하지 않은 노드 중에서 가장 좋은 한계치를 가진 마디를 먼저 확장함
  - 이를 위해 일반 큐 대신에 우선 순위 큐(priority queue)를 사용함

최고 우선 검색 분기 한정 가지치기 (2/3)

- 예) {(2, 40), (5, 30),(10, 50), (5, 10)}, 16
  - score = 20, 6, 5, 2

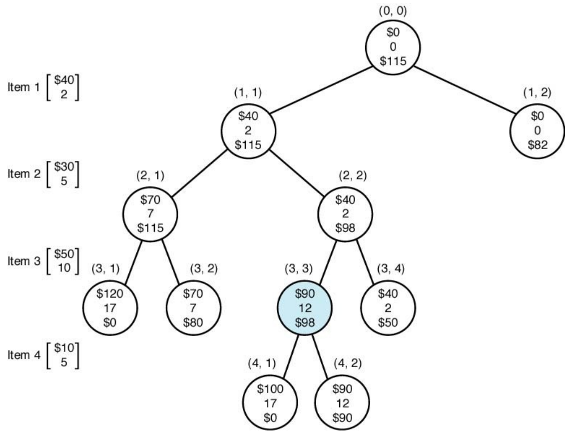
i	p_i	w_i	p_i/w_i
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2





되추척

최고 우선 검색 분기 한정



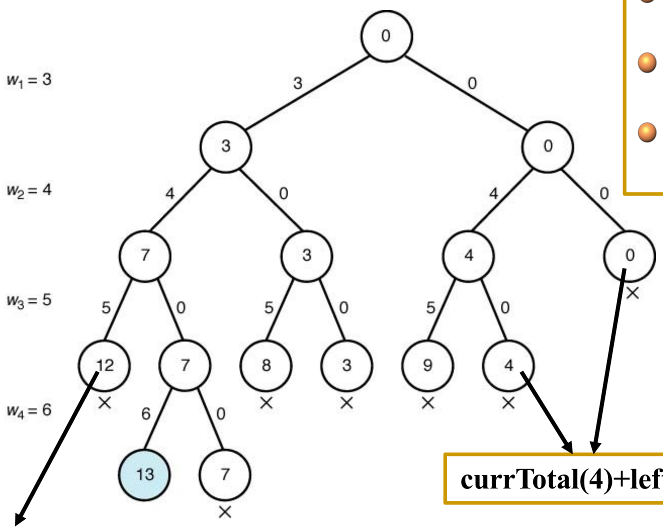
최고 우선 검색 분기 한정 가지치기 (3/3)

```
knapsack(items[], W)
Q := Node를 유지할 수 있는 priority queue (bound maxHeap)
root = 루트 노드
Q.enqueue(root)
maxProfit := 0
while Q is not empty do
    node := Q.pop()
    node.level += 1
    node.currWeight += items[node.level].weight
    node.currProfit += items[node.level].profit
    if node.currWeight ≤ W and node.currProfit > maxProfit then
        maxProfit := node.currProfit
        solution := node.include
    bound := computeBound(node)
    if bound > maxProfit then Q.push(node)
    node.currWeight −= items[node.level].weight
    node.currProfit −= items[node.level].profit
    bound := computeBound(node)
    if bound > maxProfit then Q.push(node)
```

- 다음 노드를 만들 때 Node의 복제 필요

부분집합의 합 구하기

- 이 문제도 최고 우선 분기 한정 적용 가능?
  - 예) {3,4,5,6}, 13



- 각 노드를 살펴보았을 때 우선할 노드를 결정할 수 없음
- 최고 우선 분기 한정은 최적화 문제에만 적용 가능
- n-Queens 문제, 그래프 색칠하기도 해당하지 않음

currTotal(4)+leftTotal(6)<W

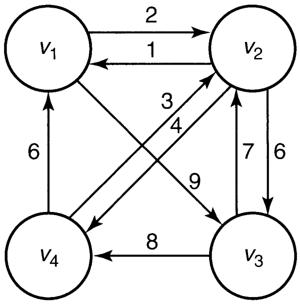
currTotal(12)+w_{i+1}(6)>W

외판원 문제 (1/6)

- 외판원 문제(traveling salesperson problem)
  - 동적 프로그래밍에서 살펴본 문제
    - $\min_{2 \le k \le n} (d[1][k] + D[v_k][V - \{v_1, v_k\}])$
      - $D[v_k][A]$: A에 속한 노드를 한 번씩 거치며 v_k에서 v_1로 가는 최단 경로의 길이
      - $D[v_k][\emptyset] = d[k][1]$
    - 시간 복잡도: $O(n^2 2^n)$
  - 가중치 방향 그래프로 표현
    - 가중치는 음이 아닌 정수임
    - 한 도시에서 다른 도시로 가는 가중치와 그 반대방향의 가중치가 다를 수 있음
    - 보통 완전 그래프를 가정함

외판원 문제 (2/6)

- 예)

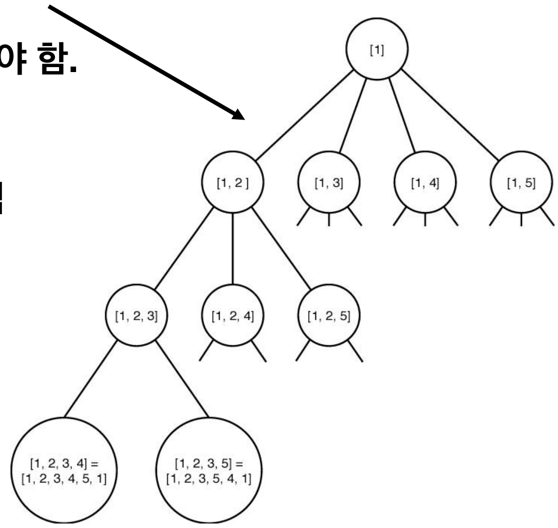


length$[v_1,v_2,v_3,v_4,v_1] = 22$
length$[v_1,v_3,v_2,v_4,v_1] = 26$
length$[v_1,v_3,v_4,v_2,v_1] = 21$

  - 모든 경로를 고려한 다음 최적의 일주여행경로를 찾는 것은 시간 복잡도가 계승 시간임. (완전 그래프: $(n-1)!/2$)

외판원 문제 (3/6)

- 되추적 기법을 이용한 해밀토니안 순환경로는 모든 순환경로를 찾아줌
  - 모든 순환 경로를 다 찾는 것은 지수 시간이 요구될 수 있음
  - 특히, 외판원 문제는 보통 완전 그래프를 많이 가정함
    - 5개의 노드로 구성된 그래프에서 모든 노드가 서로의 인접 노드인 경우 상태 공간 트리
  - 효율을 높이기 위해서는 가지치기를 해야 함. 어떻게?
  - 최적화 문제이므로 가지치기를 할 수 있다면 분기한정법이 더 효과적임



외판원 문제 (4/6)

- 분기한정을 사용하기 위해서 한계치를 계산할 수 있어야 함
  - 노드의 한계치는 그 노드를 확장하여 얻을 수 있는 일주여행경로 길이의 하한값으로 정의됨
  - 노드의 한계치가 지금까지 계산된 최적의 일주여행경로 길이보다 크면 이 노드는 유망하지 않음
- 한계치를 계산하는 법
  - $v_1 = (14, 4, 10, 20) = 4$
  - $v_2 = (14, 7, 8, 7) = 7$
  - $v_3 = (4, 5, 7, 16) = 4$
  - $v_4 = (11, 7, 9, 2) = 2$
  - $v_5 = (18, 7, 17, 4) = 4$
    - 일주여행경로의 하한값: $4 + 7 + 4 + 2 + 4 = 21$
    - 노드에서 진출하는 간선의 가중치 중 가장 작은 것들의 합으로 한계치를 추정함
    - 길이가 21인 일주여행경로가 있다는 것은 아님. 이 보다는 작을 수 없다는 것은 확실함

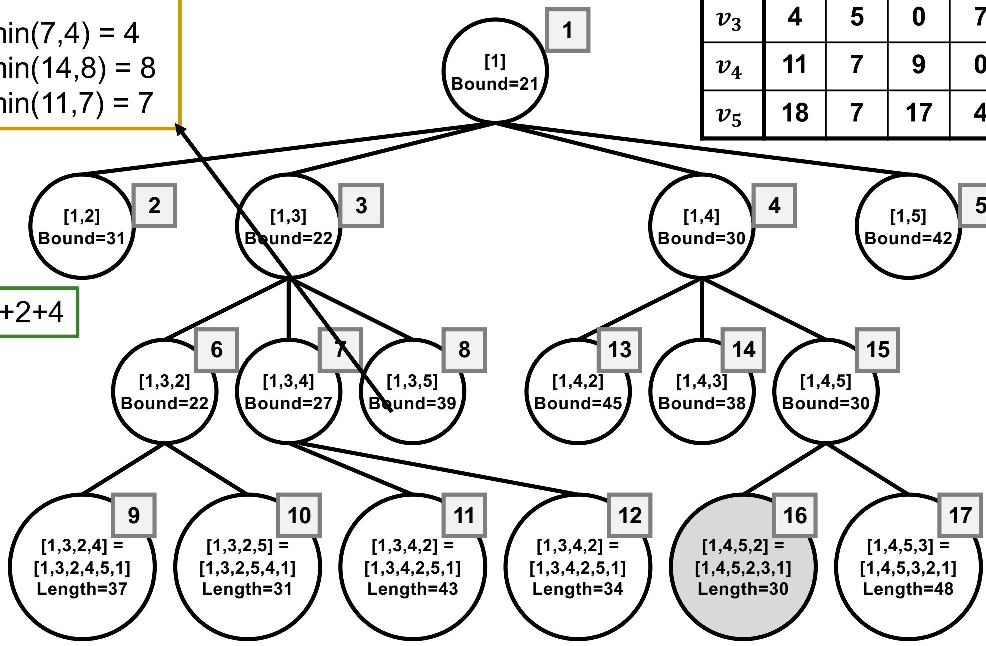
	v_1	v_2	v_3	v_4	v_5
v_1	0	14	4	10	20
v_2	14	0	7	8	7
v_3	4	5	0	7	16
v_4	11	7	9	0	2
v_5	18	7	17	4	0

외판원 문제 (5/6)

	v_1	v_2	v_3	v_4	v_5
v_1	0	14	4	10	20
v_2	14	0	7	8	7
v_3	4	5	0	7	16
v_4	11	7	9	0	2
v_5	18	7	17	4	0

1→3: 4
3→5: 16
5→(2, 4): min(7,4) = 4
2→(1, 4): min(14,8) = 8
4→(1, 2): min(11,7) = 7

14+7+4+2+4



외판원 문제 (6/6)

tsp(G[][])

Q ≔ Node를 유지할 수 있는 *priority queue (bound minHeap)*
$root.level, root.tour, root.bound$ ≔ $0, [1]$, computeBound(root)
Q.push(root)
$minLength$ ≔ ∞
while Q **is not empty do**
    $node$ ≔ Q.pop()
    if $node.bound < minLength$ **then**
        for i ≔ 2 **to** n **do**
            if i in $node.tour$ **then continue**
            if $G[node.tour[node.level]][i] = \infty$ **then continue**
            $next$ ≔ $node$.clone()
            $next.level$ ≔ $node.level + 1$
            put i at the end of $next.tour$
            if $next.level = n - 2$ **then**
                complete the tour
                if $next$.length() $< minLength$ **then**
                    $minLength$ ≔ $next$.length()
                    $solution$ ≔ $next.tour$
            else
                $next.bound$ ≔ computeBound($next$)
                if($next.bound < minLength$) Q.push($next$)
return $minLength$

```
Node{
    level   // 트리 레벨
    bound   // 가능한 최솟값
    tour[]  // 일주여행경로
}
```

- Node에 아직 tour에 포함되지 않은 노드를 유지하면 편리함
- computeBound의 계산이 간단하지 않음

작업 할당 문제 (1/3)

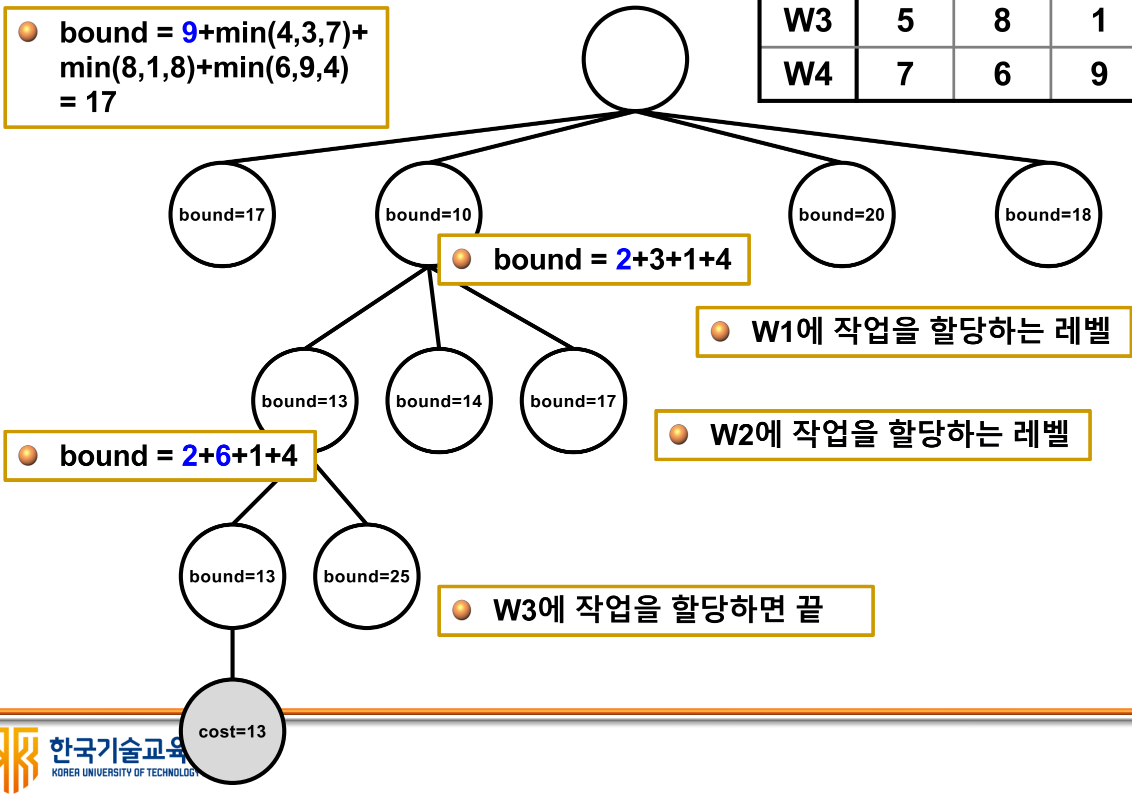
- 입력. N명과 N개 작업, 각 사람이 각 작업을 수행할 때 소요되는 시간
- 출력. 전체 작업 완료 시간을 최소화하도록 작업 할당 결과
    - 각 사람에게 하나의 작업을 할당해야 함

	작업 1	작업 2	작업 3	작업 4
일꾼 1	9	2	7	8
일꾼 2	6	4	3	7
일꾼 3	5	8	1	8
일꾼 4	7	6	9	4

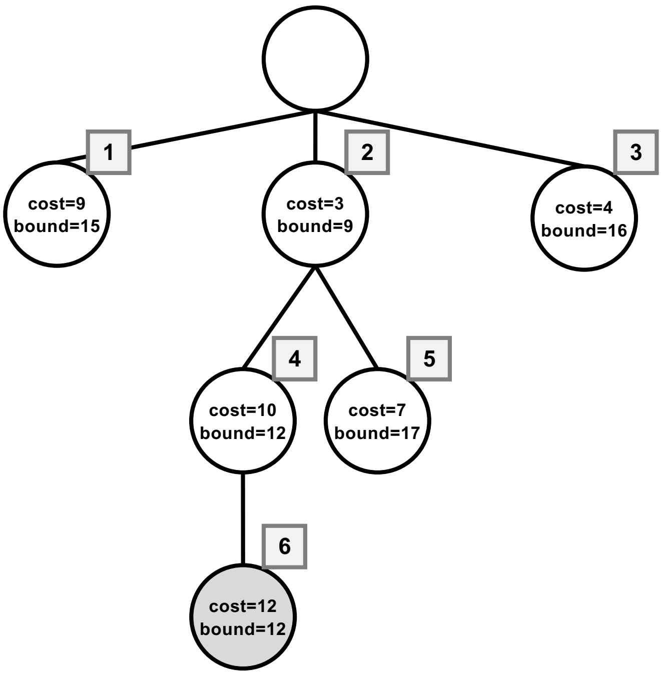
- 전수조사: $O(n!)$
- 그런데 왠지 TSP와 유사한 것 같음
    - 노드 1을 시작 노드로 하여 그다음 가능한 모든 노드를 고려함
    - 일꾼 1을 작업 1, 작업 2, 작업 3, 작업 4에 할당하는 것을 고려
    - 최소 한계는 $2 + 3 + 1 + 4 = 9$임 (이것도 비슷)

작업 할당 문제 (2/3)

	J1	J2	J3	J4
W1	9	2	7	8
W2	6	4	3	7
W3	5	8	1	8
W4	7	6	9	4



작업 할당 문제 (3/3)



	J1	J2	J3
W1	9	3	4
W2	7	8	4
W3	10	5	2