**알고리즘 1 학기말 과제 보고서**

장길진 교수님  
2012104208 전자공학부 윤종민

**(1번 문제) :**

**설명 :**  
문제를 읽고 다음 몇 가지를 알 수 있었다.

1. 각각의 짐마다 한 개만을 얹을 수 있다.
2. 각각의 짐 중에서 딱 하나만을 반으로 갈라 얹을 수 있다. 이 때 무게와 이득은 반으로 줄어든다.
3. 완전히 쪼개거나 하는 행동은 할 수 없으나, 반으로 갈라 얹은 것이 두 개 얹혀지게 되면 하나로 다시 뭉칠 수 있다.

우선 0/1 knapsack 을 2차원 배열을 만들어 동적 계획법을 통해 푸는 것은 같으나, 각 짐 중에서 딱 하나만을 반으로 갈라 얹을 수 있기 때문에 어떻게 해야 되도록 빨리 풀 수 있을지 생각을 해야했다. (여기서 공간을 최적화할 수 있는 방법도 존재하나 구현이 매우 어렵기 때문에 생략한다)

가장 맨 먼저 떠오른 생각은, 0/1 Knapsack DP 을 약간 변형해서 솔루션을 구하는 것이었다. 각 DP 순회마다 한 짐을 반으로 떼어내 일반 0/1 Knapsack 문제를 푸는 것을 반복한다. 만약 짐의 개수가 N 개이고 가방의 최대 하중이 W kg 일 때, 이 경우에는 각 DP 순회마다 O((N + 1)W) 만큼의 공간을 쓴다. 공간 면에서는 효율이 좋지만, 시간 복잡도에서는 매우 효율이 좋지 않다는 것을 알 수 있었다. 짐 개수가 N개 일 때 N 번 DP 처리를 반복해야 최종 값이 나올 수 있기 때문이다. 이 때는 O(N(N+1)W) = O(WN^2 + N) = O(N^3) 임을 알 수 있다.

그래서 좀 더 생각해본 결과 구현이 복잡하나 시간 복잡도를 O(4NW) 으로 줄일 수 있는 방법을 찾아냈다. 우선 각 짐 중 하나만 반으로 가르지 않고, 모든 짐을 반으로 나눠서 처리하게 한다. 이 때 무게와 이득은 반으로 나누지 않는다. 그리고 최대 하중을 2배로 늘린다. 그리고 각 무게에 대해서 현재 짐에 반으로 나뉜 짐이 들어와 있는지 검색해 최종 결과로 반밖에 들어온 짐이 하나 이하로만 존재하도록 구현한다.

최종 공간 복잡도는 최초안보다 매우 커서 O(4NW) + O(4NW) + O(4NW) = O(12NW) 가 되지만, 짐의 개수가 매우 크면 첫번째보다 매우 빠른 속도를 보장할 수 있다.

※ 실은 첫번째도 실제 총 사용하는 배열의 양을 대략 NO((N+1)W) = O(WN^2 + N) 으로 볼 수 있다.

그래서 다음과 같이 풀면 문제가 해결될 것이라 생각했다.

1. 파일을 읽어 짐의 무게와 이득을 받아 객체를 생성한다.  
   만약 이득이 -1 이면, 무게 값을 가방의 하중이라고 본다.
2. 이득 값을 저장하기 위한 2차원 배열과, Dirty 인가 Clean 인가를 확인하기 위한 2차원 Flag 배열을 만들어 초기화시킨다. 이 때 행의 크기는 짐의 원래 개수의 두배가 되야하며, 열의 크기는 가방의 하중의 두배가 되야 한다.
3. 0/1 Knapsack DP 연산을 수행한다. 일반 연산과는 다르게 Flag 2차원 배열을 확인하면서 수행한다.
4. 현재 위치가 [y][x] 이고, y 값에 해당되는 아이템의 무게가 weight 이라고 할 때, [y – 1][x – weight] 에 해당되는 위치의 Flag 가 Dirty (반절 노드가 하나) 인가, Double Dirty (서로 다른 반절 노드가 두개) 인가에 따라 적절한 분기를 취한다.
5. [y – 1][x – weight] 를 취한 값을 더한 값이 [y][x] 에 갱신이 되면, 나중에 아이템 사용 수를 따오기 위한 별도의 플래그 배열의 [y][x] 위치를 True 로 갱신한다.
6. 처리가 끝나면, 2차원 캐시 값 배열의 맨 마지막 위치에서 위로 올라가면서 플래그가 True 인 것만을 골라 뒤로 점프하면서 아이템 사용 수를 센다.
7. 아이템 사용 수와, 총 이득 / 2 을 출력한다. 왜냐면 아이템을 2개씩 생성 할 때, 이득을 반절로 하지 않았기 때문이다.

**Pseudo Code :**// Input   
ReadFile("input.txt", read\_mode);  
ItemList.Insert(DummyObject);  
Loop (파일을 다 읽을 때까지) {  
 if (Second value is -1) {  
 input First value to backpack\_max  
 Eㅌxit Loop;  
 }  
 else {  
 Make Item Object has id, weight, value;  
 PositionArray[layer].Insert(Object) for twice.   
 }  
}  
  
Create(CacheValue[2 \* itemNumber + 1][2 \* backpack\_max + 1]);  
Create(CacheFlag[2 \* itemNumber + 1][2 \* backpack\_max + 1]);  
Create(CheckFlag[2 \* itemNumber + 1][2 \* backpack\_max + 1]);  
  
// Procedure  
for (A = ItemList[1] ~ ItemList[end]) {  
 for (B = CacheValue[i][0] ~ CacheValue[i][end]) {  
 when (A.weight <= j) {  
 갱신 값 C = CacheValue[i-1][j-weight] + A.value;  
 플래그 D = CacheFlag[i-1][j-weight];  
 윗행 값 E = CacheValue[i-1][j];  
   
 when (D) {  
 clean -> {  
 if (C > D) { Target = C; TargetFlag = dirty;  
CheckFlag = true; }  
 else { Target = E; TargetFlag = UpRowFlag; }  
 }  
 dirty -> {  
 if (C > D) {   
 Target = C;   
 TargetFlag = double\_dirty;   
 CheckFlag = true; }  
 else if (C same D) {  
 if (UpRowFlag == clean) Target = E;  
 else { Target = C; TargetFlag =  
double\_dirty; }  
 }  
 else {  
 Target = E;  
 TargetFlag = UpRowFlag;  
 }  
 }  
 double\_dirty -> {  
 if (Upper row has same id to present row   
 And C > D) {  
 Target = C;  
 TargetFlag = clean;  
 CheckFlag = true;  
 }  
 else { Target = E; TargetFlag = UpRowFlag; }  
 }  
 }  
 }  
 }  
}  
  
// Print  
var x = CacheValue 의 마지막 열  
for (y = CacheValue 의 마지막 행 ~ 0) {  
 if (CheckFlag is true) {  
 x -= 현재 위치의 Item 의 weight;  
 print Item.id;  
 }  
}

**수행시간 분석 :**

수행 시간은 대략 O(4NW) 로 분석된다. CacheValue 의 2차원 배열을 하나씩 훑어 가면서, 분기에 따라 적절한 연산 O(1) 을 수행하는데, 이 때 2차원 배열의 행과 열의 크기를 따져보면, 행은 아이템의 총 개수 + 1 이고, 열은 가방의 최대 무게치 + 1 이다.

여기서 파일로 가져온 아이템의 개수가 N 이라고 할 때, 모든 종류의 아이템이 하나씩 복제되야 하기 때문에 2N 개가 되며, 더미가 포함되므로 결과로는 2N + 1 이라고 할 수 있다. 가방의 최대 무게치가 W 이라고 하면, 0.5 아이템을 가려내기 위해서 2W 가 되야한다. 이 때 열에도 더미 열이 하나 존재하기 때문에 2W + 1 이라고 할 수 있다.

따라서 좀 더 자세한 수행 시간은 O((2N + 1) \* (2W + 1)) = O (4NW + 2W + 2N + 1) 이다.

**(2번 문제) :**

**설명 :**  
문제를 읽고 다음을 알 수 있었다.

1. 각각의 짐마다 한 종류의 짐에 한해서 하나를 더 얹을 수 있다.
2. 기존 1번의 문제와 매우 유사하다.

따라서 기존 1번 문제의 0/1 knapsack 을 처리하는 코드 부분만을 고쳐서 구현해보기로 했다. 우선 1번 문제와는 다르게 이 문제에서는 이득에 관한 문제를 해결하기 위해 가방의 최대 중량을 2배로 늘이지 않아도 됬다. 그러나 N 개의 아이템 종류 중 한 종류가 두 번씩 올 가능성이 있기 때문에 실제 들어가는 아이템의 수는 2N 개로 1번과 동일하다. 따라서 DP 에서 값을 저장하는 차원 배열의 공간 복잡도는 O(2NW) 이다.

그러나 다른 종류의 짐(아이템)이 2개 이상 들어가는 것을 방지하기 위해서는 기존 1번과는 다른 플래그 (clean, dirty, double\_dirty 와는 다른) 가 필요했다.

1. 아무 짐도 들어가 있지 않을 때
2. 아무 종류의 짐이 하나씩 들어가 있을 때
3. 한 종류의 짐이 두 개 들어가 있을 때
4. 한 종류의 짐이 두 개 들어가 있고 다른 종류의 짐 또한 들어가 있을 때

그리고 check\_flag 2차원 배열 역시 역으로 아이템의 개수를 파악하기 위해 필요했다.  
생각한 수행 순서는 다음과 같다.

1. 1번 문제와 동일
2. 이득 값을 저장하기 위한 2차원 배열과, Dirty 인가 Clean 인가를 확인하기 위한 2차원 Flag 배열을 만들어 초기화시킨다. 이 때 행의 크기는 짐의 원래 개수의 두배가 되야한다. 열의 크기는 읽어온 가방의 하중과 동일하다.
3. 0/1 Knapsack DP 연산을 수행한다.
4. DP 연산에서 2차원 배열 안의 현재 위치에 해당되는 플래그가…
   1. 1번 플래그 (NONE) 인 경우  
      값을 갱신하고 2번 플래그로 갱신한다.
   2. 2번 플래그 (NORMAL) 인 경우  
      값을 비교해서, 만약 같은 아이템이 연속으로 들어올 경우 3번 플래그로 갱신한다.  
      아닌 경우는 플래그를 유지한다.  
      값을 비교했을 때, 윗 행의 동일 열 위치의 값이 클 경우는 윗 위치의 플래그에 따라서 적절한 연산을 행한다.
   3. 3번 플래그 (FILLED) 인 경우  
      값을 비교해서, 클 경우는 값을 갱신하고 4번 플래그로 갱신한다. 왜냐면 FILLED 플래그가 성립되는 조건이 이전에 같은 종류의 아이템이 연속으로 두 번 들어왔을 때이기 때문이다.  
      값이 크지 않을 경우에는 B의 경우와 마찬가지로 윗 위치의 플래그에 따라서 적절한 연산을 행한다.
   4. 4번 플래그 (LIMIT) 인 경우  
      값이 윗 위치의 값보다 크면서, 현재 행의 위치가 홀수인 경우에만 더해진 값을 갱신하도록 했다. 플래그는 유지한다. 왜냐하면 행의 위치가 홀수일 때는 이전에 참조한 아이템 종류와 지금 참조하고 있는 아이템 종류가 다르기 때문에, 같은 종류의 아이템이 다시 중복되서 들어가는 경우가 배제되기 때문이다.  
      그렇지 않은 경우에는 윗 위치의 값과 플래그를 가져와 갱신한다.

**Pseudo Code :**

구조는 1번 문제의 의사 코드와 같기 때문에, 변경된 부분만 올리기로 했다.

Create(CacheValue[2 \* itemNumber + 1][backpack\_max + 1]);  
Create(CacheFlag[2 \* itemNumber + 1][backpack\_max + 1]);  
Create(CheckFlag[2 \* itemNumber + 1][backpack\_max + 1]);  
  
// Procedure  
for (A = ItemList[1] ~ ItemList[end]) {  
 for (B = CacheValue[i][0] ~ CacheValue[i][end]) {  
 when (A.weight > j) {  
 Target = E; TargetFlag = UpRowFlag;  
 }  
 when (A.weight <= j) {  
 갱신 값 C = CacheValue[i-1][j-weight] + A.value;  
 플래그 D = CacheFlag[i-1][j-weight];  
 윗행 값 E = CacheValue[i-1][j];  
   
 when (D) { NONE -> {  
 if (C > D) {   
 Target = C; TargetFlag = NORMAL;   
 CheckFlag = true;   
 } else { Target = E; TargetFlag = UpRowFlag; }  
 }  
 NORMAL -> {  
 if (C > D) {   
 Target = C;   
 if (Upper row has same id to present row )  
 TargetFlag = FILLED;  
 else TargetFlag = NORMAL;  
 CheckFlag = true;   
 } else if (C same D) {  
 Target = E;  
 when (UpRowFlag) {  
 // 플래그에 따른 적절한 연산을 한다.  
 }  
 } else {  
 Target = E; TargetFlag = UpRowFlag;  
 }  
 }  
 FILLED -> {  
 if (C > D) {  
 Target = C; TargetFlag = LIMIT;  
 CheckFlag = true;  
 } else if (C same D) {   
 Target = E;  
 when (UpRowFlag) {  
 // 플래그에 따른 적절한 연산을 한다.  
 }  
 }  
 }  
 LIMIT -> {  
 if (현재 row is odd And C > D) {  
 Target = C; TargetFlag = LIMIT;  
 CheckFlag = true;  
 } else {  
 Target = D; TargetFlag = UpRowFlag;  
 }  
 }  
 }  
 }  
 }  
}

**수행시간 분석 :**

기본적인 수행 시간은 대략 O(2NW) 으로 예상 된다. 왜냐면 조회해야할 아이템의 수가 2N 개이며, 배열의 열 크기를 담당하는 가방의 최대 하중이 W 이기 때문이다. 1번 과는 달리 가방의 최대 용량이 2배가 될 필요가 없다.

좀 더 자세히 분석하면, 1번 DP 문제처럼 2번도 2차원 배열의 행과 열을 하나씩 다 훑어가면서 O(1) 의 연산을 수행한다. 이 때 파일에서 가져온 아이템 개수는 N 이며, 중량의 크기는 W 이다.

중량의 크기는 2배가 될 필요가 없으나, 아이템 중복을 처리하기 위해 모든 종류의 아이템이 복제되야 하므로, 실제 아이템 개수는 2N 개이다. 그리고 여기에서도 각 행과 각 열에 더미 행과 더미 열이 들어가기 때문에 총 행과 총 열은 2N + 1, W + 1 이다.

따라서 DP 를 수행하는데 필요한 총 시간은 O((2N + 1) \* (W + 1)) = O(2NW + 2N + W + 1) 이다.

**(3번 문제) :**

**설명 :**   
문제를 읽고 다음 몇 가지를 알 수 있었다.

1. 물이 떨어지는 위치는 주어진 폭의 절반이 되는 위치에서 떨어진다.
2. 물이 밑으로 완전히 떨어지기 까지 중간에 N 개의 계층이 있으며, 각 계층 마다 물줄기를 흘려보내는 슬릿이 M(N) 개 존재한다.
3. 위에서 아래로 물 줄기를 흘려 보낼 때 슬릿을 넘어서 멀리 존재하는 슬릿에서 흘러 내려가는 경우는 없다.

따라서 각 슬릿에 고유 번호를 정하고, 이 슬릿을 그래프로 만들어 입력 슬릿에서 출력 슬릿까지의 최단 경로를 구하면 될 것이다라고 생각했다. 이 때 위층 슬릿에서 아래층 슬릿까지의 거리는 항상 양수이기 때문에 우선순위 큐 (Min Heap) 을 이용한 Dijkstra Algorithm 을 이용해 정답을 풀어내는 것이 좋다 생각했다.

또한 마지막 층에서 떨어지는 물줄기는 어디서 떨어지든 간에 상관이 없기 때문에 마지막 계층의 슬릿들과 가상으로 만들어진 출력 슬릿 간의 거리는 0이라고 가정했다.

그래서 결과적으로 다음과 같은 과정을 거쳐서 풀면 될 것이라고 생각했다.

1. 입력을 받아 각 계층에서 위치를 가지는 슬릿 객체를 만들어 2차원 배열에 집어넣는다.
2. 2차원 배열의 양 끝줄에 입력과 출력 슬릿을 만들어 집어넣는다.
3. 2차원 배열을 토대로 위 계층의 임의 슬릿에서 가장 가까운 1~2개의 아래층 슬릿을 연결짓고 Edge 객체를 만들어 저장한다.
4. 입력 슬릿의 최종 거리를 0으로 초기화한 다음에 우선순위 큐에 넣고 데이크스트라 알고리즘을 수행한다.
5. 수행이 끝난 후에 출력 슬릿의 총 거리 (최단 거리) 와 경로를 파악해 출력한다.

**Pseudo Code :**// Input  
ReadFile("input.txt", read\_mode);  
var layer = 0;  
PositionArray[layer].Insert(InputObject);  
Loop (파일을 다 읽을 때까지) {  
 if (First Value is -1) Exit Loop;  
 else {  
 Make Position Object has layer, position, unique index value;  
 PositionArray[layer].Insert(Object);  
 }  
 ++layer;  
}  
PositionArray[layer].Insert(OutputObject);  
  
// Make Graph  
for (PositionArray 의 입력 계층 ~ 출력 계층의 바로 위 계층) {  
 for (A = 한 계층의 Position Object) {  
 nullable var left\_object, right\_object;  
 for (B = 아랫 계층의 Position Object 리스트) {  
 var distance = |A.position - B.position|;  
 if (B 가 A 보다 왼쪽에 있거나 중앙이며 가장 가까운가?)  
 left\_object = B;  
 else if (B 가 A 보다 오른쪽에 있고 가장 가까운가?)  
 right\_object = B;  
 }  
   
 // 만약 left\_ 혹은 right\_ 중 하나 이상이 NULL 이면 해당 코드는 무시한다.  
 A.edgeArray.Insert(Edge(left\_object?, distance\_1));  
 A.edgeArray.Insert(Edge(right\_object?, distance\_2));  
 }  
}  
  
// Dijkstra Algorithm  
InputObject.distance = 0;  
PriorityQueue.Push(InputObject);  
while (PriorityQueue.NotEmpty()) {  
 S = PriorityQueue.Top(); PriorityQueue.Pop();  
 for (EdgeItem in S.edgeArray) {  
 D = EdgeItem.destination\_object;  
 if (D.distance > EdgeItem.distance + S.distance) {  
 D.distance = EdgeItem.distance + S.distance;  
 D.parent = S;  
   
 PriorityQueue.Push(D);  
 }  
 }  
}  
  
// Print  
Nullable var T = OutputObject.parent  
if (T is not Null) {  
 print T.pos reversely  
 T = T.parent  
}

**수행시간 분석 :**

우선 계층의 수를 N 이라고 하며 각 계층의 슬릿의 수를 M(N) 이라고 한다.

그러면 Input 에서 각 계층의 슬릿 정보를 가져와 객체화 해서 저장하기 위해서는 다음의 시간이 필요할 것이다.

따라서 Input 에서는 P 시간이 걸린다.

그래프를 만드는 단계에서는 어림잡아 O(P^2) 시간이 걸리는 것을 알 수 있다. 왜냐면 각 계층의 슬릿에 대해 다음 계층 리스트의 모든 슬릿에 대해 비교를 하여 가장 가까운 슬릿을 추려내기 때문이다.

자세히 살펴보면 다음과 같다. ( prob\_3.cc 참고 )

67 | for (auto it = positions.begin(); it < positions.end() – 1; it)++  
68 | for (auto& source : \*it) {

여기서는 각 계층의 모든 슬릿에 대해서 한번씩은 참조를 하게 되기 때문에, 우선 P 의 시간이 걸리는 것을 알 수 있다.

또한 73번 줄에서

73 | for (auto& dest : \*(it + 1)) {

에서도 첫 계층을 제외한 모든 계층 ( N – 1 계층) 을 조회하기 때문에 N 번째 계층 당 M(N+1) 의 시간이 걸릴 수 있음을 알 수 있다.

따라서 임을 알 수 있다. 마지막에 – 1 이 된 이유는 마지막 계층은 다음 계층에 대해 그래프를 만들지 않기 때문이다.

그래서 현재까지의 시간은 다음과 같을 것이다. F(t) = P + P^2 – 1

그리고 Priority Queue 을 활용한 Dijkstra Algorithm 은 모든 슬릿이 한번은 조회된다고 가정하면 ( P ), Priority Queue 에서 Push 가 될 때 전체 소요시간은 이다.

그리고 129 번째 줄에서

129 | for (const auto& edge : source.edges) {

으로 우선순위 큐에서 뽑아낸 위치 Object (vertex) 마다 연결된 노드의 수는 만약 최악을 상정할 시에 P가 될 것이다.

따라서 Dijkstra Algorithm 에서 최악의 시간 소요는 대략 이다.

결과를 출력하는 데는 N 만큼의 시간 (레이어 계층의 수)이 사용될 것이다.   
따라서 최악일 경우 총 소요 시간은