CS202: 동적 계획법 보고서

20160385 오진영

1. Tomato

기본적인 문제 해결 방법은 Cake2 문제와 동일하다. 먼저 토마토의 수(상자의 크기와 층 수를 고려하여) 만큼의 배열을 만든 후, 이 배열을 맨 처음 토마토(1 층의 1 열 1 행에 있는 토마토)부터 각 인덱스까지의 토마토들의 가격의 합으로 채워야 한다. 동적 계획법을 그대로 따라서, 먼저 1 층 1 열의 1 행부터 차근차근 채워 나가는데, 이때 각 x 층 y 열 z 행의 토마토까지의 가격의 합은 인접한 위치인 (x-1)층 y 열 z 행, x 층 (y-1)열 z 행, x 층 y 열 (z-1)행의 토마토의 가격의 합으로부터 유도할 수 있다. 하지만, 이 경우를 직접 3 차원상에 그려 보면, 중복되는 부분이 있는 것을 볼 수 있다. Cake2 문제처럼 중복되는 부분들을 빼 주고, 또 중복해서 빼진 부분을 다시 더해주면 1 층 1 열 1 행의 토마토부터 각 인덱스까지의 토마토들의 가격의 합을 구할 수 있다.

이 상태에서, 주문에 해당하는 토마토들의 가격을 구하는 것은 쉽다. 먼저 (1, 1, 1) 부터 (x2, y2, z2)까지의 토마토들의 가격의 합을 이미 구해 두었기 때문이다. 여기에서 주문에 해당하지 않는 토마토들을 빼 주면 되는데, 먼저 xy 평면에서 생각하면 [1, x1 – 1]*[1, y2]와 [1, x2]*[1, y1 - 1]을 빼 주고 중복해서 빼진 [1, x1-1]*[1, y1-1]을 다시 더해 주어야 한다. 이 때 오버플로우를 방지하기 위해 더하는 것을 먼저 해야 한다. 이제, 주문의 x, y 에 해당하는 토마토들의 가격의 합을 얻어 내었지만, [1, z1-1]에 해당하는 토마토들의 가격을 빼 주어야 한다. 위와 같은 방법으로 하되, 참조하는 z 인덱스를 z2 가 아닌 (z1-1)로 하면 쉽게 구할 수 있고, 앞서 구한 값에서 이 값을 빼 주면 된다.

2. Big Data

문제의 조건이 너무 복잡하기 때문에 문제를 단순화시킬 필요가 있다. 먼저, 클럭에 대해서 고려해 보자. 주어진 의뢰들 중 가장 낮은 클럭을 요구하는 의뢰의 요구 클럭보다 낮은 클럭을 가진 컴퓨터는 필요가 없다. 또, 대여 가능한 컴퓨터들 중 가장 높은 클럭보다 더 높은 클럭을 요구하는 의뢰는 해결할 수 없다. 이를 조금 더 일반화해서, 의뢰와 컴퓨터를 함께 vector 에 넣은 후 각각의 클럭에 따라서 오름차순으로 정렬되어 있다고 생각해 보자. 이 때, 뒤에 있는 의뢰들은 앞에 있는 컴퓨터로 해결할 수 없고, 반드시 의뢰 자신보다 뒤에 있는 컴퓨터로만 해결할 수 있다. **즉, 클럭에 따라 의뢰, 컴퓨터 구별 없이 오름차순으로 정렬되어 있을 때, 의뢰는 필요한 코어 수를 늘리고 그 뒤에 있는 컴퓨터들은 각각의 코어 수만큼 필요한 코어 수를 줄인다.**

이제 이 문제를 knapsack 문제로 생각하여 풀 수 있다. 먼저, 위에서 설명한 대로 컴퓨터와 의뢰를 벡터 안에 넣고 오름차순 정렬을 한 다음에, 필요한 코어 수를 인덱스로 하고 이때의 수익을 그 값으로 하는 배열 `revenue` 를 만든다. 이때, `revenue` 의 크기는 최악의 경우에 대응해서 의뢰마다 필요로 하는 코어 수의 총 합으로 해야 한다. 이후, 정렬된 벡터를 순서대로 순회하면서, 컴퓨터라면 필요한 코어의 수를 늘리고, 의뢰라면 필요한 코어의 수를 줄이면 된다.

먼저, 컴퓨터일 경우에는 필요한 코어의 수를 줄이므로 `revenue` 배열의 앞 인덱스의 값들을 변경하게 될 것이므로 앞에서부터 뒤로 순회해야 한다. 또, 현재 필요한 코어의 수가 0 일 경우에는 굳이 컴퓨터를 새로 대여할 이유가 없다. 어차피 이 컴퓨터로는 나중에 나올 의뢰들을 해결할 수 없기 때문이다. 따라서, 필요한 코어 수의 인덱스를 1 부터 뒤로 돌면서 아래 코드를 수행한다.

```
if(revenue[i] > min64){
                    currentrevenue = revenue[i] - it->price;
                    currentcore = it->core > i ? 0 : i - it->core;
                    // 필요한 코어 수는 0 이상이어야만 함

                    revenue[currentcore] = revenue[currentcore] > currentrevenue
                        ? revenue[currentcore]
                        : currentrevenue;
                    // 이 상태에서의 수익 값 업데이트
                }
```

`min64` 는 `revenue` 배열의 초기값으로, 아직 인덱스 `i` 만큼의 코어를 필요로 하는 경우의 수가 발견되지 않았을 때 이 값을 갖게 된다. 따라서, 현재 `revenue[i]`, 즉 인덱스 `i` 만큼의 코어를 필요로 하는 경우의 수가 있을 경우 지금 관심의 대상인 컴퓨터를 대여하게 되면 이 경우의 수의 수익에서 대여료만큼이 감소하게 될 것이고, 필요한 코어 수는 그만큼 감소하게 될 것이다. 단 0 미만으로 감소할 수는 없으므로 이 경우 0 으로 설정해 주어야 한다. 이제, 감소한 필요 코어 수 만큼을 필요로 하는 기존의 경우의 수 `revenue[currentcore]` 와 비교해서 이 컴퓨터를 대여하는 경우의 수가 더 큰 이득을 가져온다면 이 컴퓨터를 대여하는 경우의 수익으로 업데이트 한다.

의뢰일 경우에도 마찬가지 방법으로 revenue 배열을 업데이트 해야 하는데, 새로운 의뢰를 받게 될 경우 필요한 코어의 수를 증가시키게 되므로 revenue 배열의 나중 인덱스의 값을 변경시키게 되어 뒤에서부터 순회해야 한다.

```
if(revenue[i] > min64){
                    // 일을 하면 수익은 증가하지만 필요한 코어 수도 증가할 것이다
                    currentrevenue = revenue[i] + it->price;
                    currentcore = i + it->core;

                    // 어쨋던 증가한 수익 업데이트
                    revenue[currentcore] = revenue[currentcore] > currentrevenue
                        ? revenue[currentcore]
                        : currentrevenue;
                }
```

의뢰를 받는 경우에는 수익 currentrevenue 는 이 의뢰의 보상만큼 증가할 테지만, 필요한 코어 수는 이 의뢰의 요구 만큼 증가할 것이다. 따라서, 증가한 필요 코어 수의 인덱스에 해당하는 경우의 수보다 이 경우의 수가 더 나은 이익을 가져다 줄 경우 값을 업데이트하면 된다.

정렬된 벡터를 끝까지 순회하고 나면, 이제 더 이상 컴퓨터를 대여할 수 없다. 따라서, 이 시점에서 더이상 코어를 필요로 하지 않는 currentrevenue[0]의 값을 출력하면 그것이 답이다.