챕터 9. Greedy Approach (탐욕 접근법) - (1) Deadline Scheduling

과제 정보:

Input

첫째 줄에 job의 개수 N이 주어진다.

둘째 줄에 N개의 deadlines 가 주어진다.

셋째 줄에 N개의 profits 가 주어진다.

단, 데드라인과 기대이익은 기대이익의 내림차순으로 정렬되어 있음이 보장된다.

Output

첫째 줄에 최적값(총 이익의 최댓값)을 출력한다.

둘째 줄에 최적해(데드라인 이내에 실행할 수 있는 feasible set)의 profits 를 출력한다.

단, 출력하는 최적해는 데드라인의 오름차순(알고리즘의 선택 순서)으로 정렬되어 있어야 한다.

Sample Input 1

```
7
3 1 1 3 1 3 2
40 35 30 25 20 15 10
```

Sample Output 1

```
100
35 40 25
```

Sample Input 2

```
7
2 1 3 1 3 4 2
70 60 50 40 30 20 10
```

Sample Output 2

```
200
60 70 50 20
```

마감시간 있는 스케줄 짜기 문제

마감시간과 함께 여러 개의 작업이 주어지고,

각 작업을 마감시간 전에 마치면 받을 수 있는 보상이 정해져 있다.

주어진 마감시간내에 얻을 수 있는 보상을 최대화하는 스케줄을 작상하라.

-> 최소화 x, 최대화 o

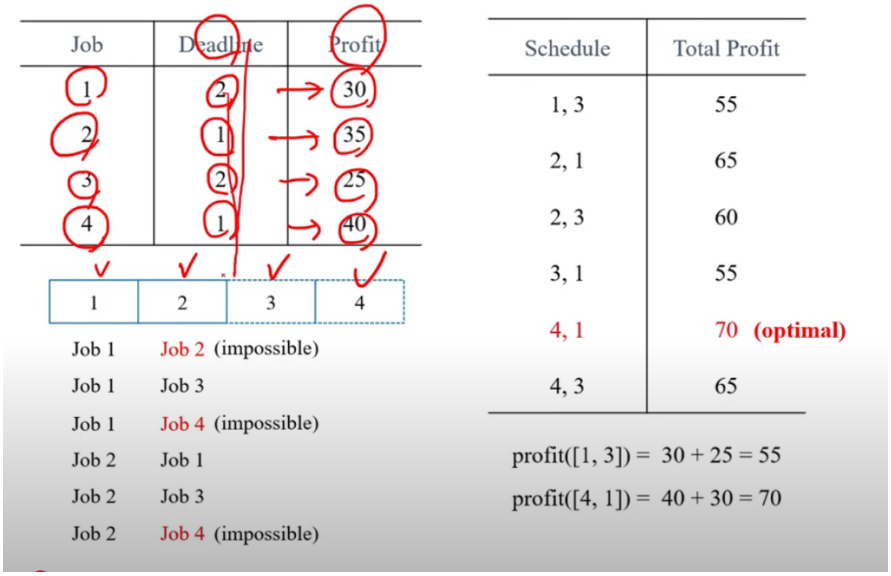
각 작업을 끝내는데 1의 단위 시간이 걸린다고 가정한다. (Greedy algorithm의 단순화)

보상은 마감시간 이전이나 마감시간에 끝내는 경우에만 주어진다.

마감시간 이후의 스케줄에 대해서는 고려할 필요가 없다.

1. 각각의 job에는 deadline과 profit이 주어졌는데, job간 deadline이 맞지 않으면 impossible한 job으로 볼 수 있다.

예를 들어 job1의 deadline은 2이고, job2의 deadline은 1이다. job2는 job1과 달리 단위시간 기준 1안에 끝내야되기에 스케줄이 서로 맞지 않아, job1과 job2는 impossible한 job이라 볼 수 있다.



Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

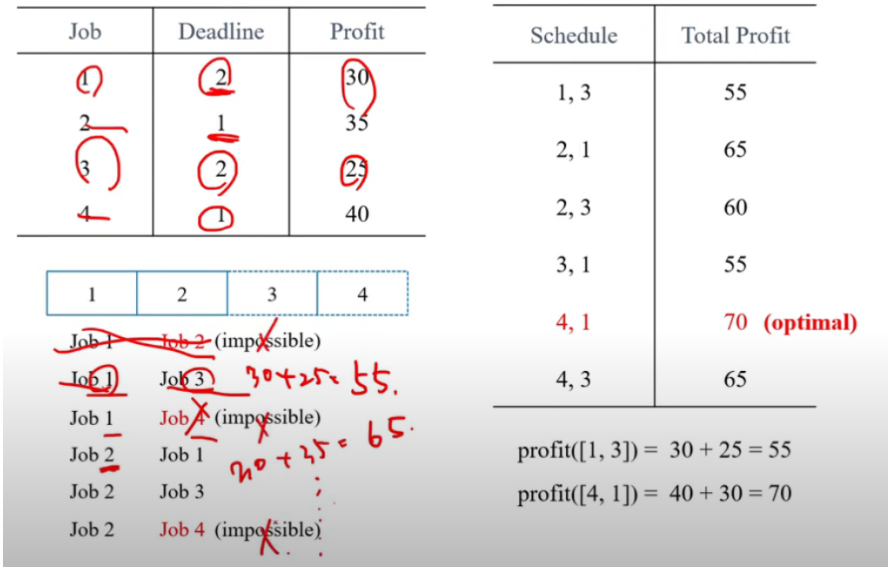
1	2	3	4

Job 1 Job 2 (impossible)
Job 1 Job 3
Job 1 Job 4 (impossible)
Job 2 Job 1
Job 2 Job 3
Job 2 Job 4 (impossible)

Schedule	Total Profit
1, 3	55
2, 1	65
2, 3	60
3, 1	55
4, 1	70 **(optimal)**
4, 3	65

profit([1, 3]) = 30 + 25 = 55
profit([4, 1]) = 40 + 30 = 70

2. 따라서 각 job들을 서로 비교해보면 impossible한 job sequence가 있을 것이고, possible한 job sequence가 있을 것이다.



Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

1	2	3	4

Job 1 Job 2 (impossible)
Job 1 Job 3
Job 1 Job 4 (impossible)
Job 2 Job 1
Job 2 Job 3
Job 2 Job 4 (impossible)

Schedule	Total Profit
1, 3	55
2, 1	65
2, 3	60
3, 1	55
4, 1	70 **(optimal)**
4, 3	65

profit([1, 3]) = 30 + 25 = 55
profit([4, 1]) = 40 + 30 = 70

3. 오른쪽은 possible한 job sequence를 나열한 것이다. 또한 각각의 frofit을 더하고 그 중 최대값(optimal)을 찾는다. 따라서 첫번째 시간에는 job4을 하고 두번째 시간에는 job1을 하는 것이 최상의 schedule인 것을 알 수 있다.

문제를 정리하자면,

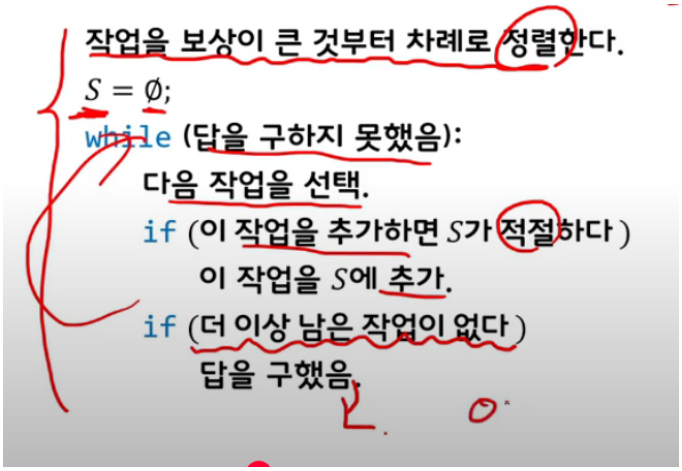
∴ 각각의 job에 deadline과 frofit이 주어지면, deadline안에 끝낼 수 있는 job sequence(작업의 순서)를 정하고, 보상의 최대화를 찾는 문제로 정의할 수 있다.

저것을 brute-force방법으로 풀면은 시간이 아주 많이 걸린다. 대신 선형시간 안에 끝낼 수 있는 알고리즘이 있는데..

마감시간 있는 스케줄 짜기: 탐욕법(The Greedy approach)

보상에 따라서 비오름차순으로 작업을 정렬한다.

각 작업을 순서대로 하나씩 가능한 스케줄에 포함시킨다.



적절함(feasibility)의 정의

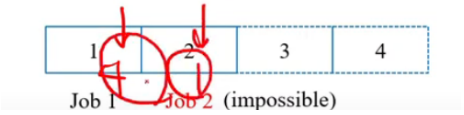
어떤 순서(sequence)는 그 순서 내의 모든 작업이

데드라인 이전에 시작될 때 **적절한 순서(feasible sequence)**라 한다.

- [4, 1]은 적절한 순서
- [1, 4]는 부적절한 순서

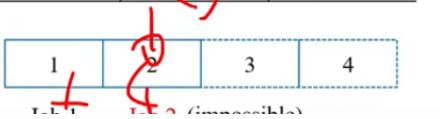
(job4는 deadline이 1인데, job1의 deadline때문에 2시간만에 끝나는건 적절하지 않기 때문이다.)

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40



Job 1 Job 2 (impossible)

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40



어떤 집합(set)은 그 집합의 원소들로

하나 이상의 적절한 순서를 만들 수 있을 때 적절한 집합(feasible set)이라 한다.

- {1, 4} 는 적절한 집합: [4, 1]이라는 적절한 순서를 만들 수 있음.
- {2, 4} 는 부적절한 집합: 적절한 순서를 만들 수 없음.

[2, 4] : X, [4, 2] : X

1. job들은 profit을 기준으로 비내림차순 정렬이 되어있다. 또한 S집합에 job을 추가하여 각각의 집합들이 feasible한지 검사하고, feasible하지 않으면은 squence를 rejected(거부)한다.

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

The jobs are already sorted by the profit

- $S = \phi$
- $S = \{1\}$, [1] is feasible

2. 집합에 job2를 추가할 시 집합 {1,2}는 [2,1]이라는 sequence가 feasible하니까 rejected하지 않고 S집합에 그대로 추가한다.

- $S = \{1, 2\}$, [2, 1] is feasible

3. 집합에 job3을 추가할 시, 모든 sequence가 no feasible이므로, 집합 S에 job3을 rejected한다.

- $S = \{1, 2, 3\}$, rejected
  there is no feasible sequence for this set : $S = \{1, 2\}$

4. 집합에 job4를 추가할 시, [2,1,4] 라는 feasible한 sequence를 찾을 수 있으므로 S집합에 그대로 추가한다.

- $S = \{1, 2, 4\}$, [2, 1, 4] is feasible

5. 집합에 job5를 추가할 시, feasible한 sequence가 없으므로 집합 S로부터 rejected한다.

- $S = \{1, 2, 4, 5\}$, rejected
  $S = \{1, 2, 4\}$

6. 집합에 job6를 추가할 시, 위와 마찬가지이기에 집합 S로부터 rejected한다.

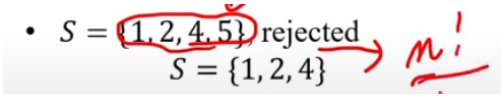
- $S = \{1, 2, 4, 6\}$, rejected
  $S = \{1, 2, 4\}$

7. 집합에 job7을 추가할 시, 또한 위와 마찬가지이기에 집합 S로부터 rejected한다.

- $S = \{1, 2, 4, 7\}$, rejected
  $S = \{1, 2, 4\}$ **(feasible set)**

∴ 따라서 더이상 남는 job이 없으므로 feasible set은 {1, 2, 4}인 것을 알 수 있다.

보조정리 4.3



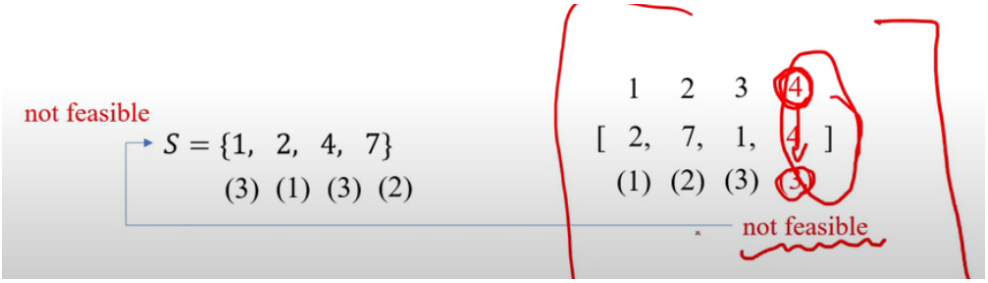
feasible한 set인지 아닌지 확인할려면 모든 경우의 수를 구해야하니까 n!이라는 시간복잡도를 가지게 된다.

해결 방법은 수학적으로 증명하는 것이다.

S를 작업의 집합이라고 할 때,

"S가 적절하다"는

"S에 속한 작업을 마감시간이 감소하지 않게 나열하여 얻은 순서가 적절하다"의 필요충분조건이다.



1. deadline을 비내림차순으로 정렬하였을때 job2, job7, job1은 모두 deadline을 지켰다. 하지만 job4는 deadline을 넘겼으므로 not feasible한 squence라하면 집합 S도 not feasible한 set이라 할 수 있다.

// profit은 현재로선 필요 없고, deadline과 job으로 feasible한 set을 찾아보자.

1. 일단 J sequence는 job1부터 넣어서 feasible한 status로 시작한다.

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

- $J = [1]$

The jobs are already sorted by the profit
The profits need not to be listed (Lemma 4.3)

2. 각 job들의 profit이 내림차순으로 정렬된 기준으로 K sequence에 job2를 deadline기준 비내림차순으로 insert하고, 이 sequence가 feasible한지 조사해본다. feasible하기에 J sequence를 [2,1]로 2를 추가한다.

- $K = [2,1]$, K is feasible
  $J = [2,1]$

3. 다음 정렬된 job3을 선택하여 K sequence에 deadline이 비내림차순으로 정렬되게 insert해보니 not feasible하여 reject된다. 따라서 J sequence에 추가하지 않는다.

- $K = [2,3,1]$ is rejected, because
  K is not feasible

4. 다음 정렬된 job4를 선택하여 K sequence에 deadline이 비내림차순으로 정렬되게 insert해보니 feasible하여 j sequence에 추가한다.

- $K = [2,1,4]$. K is feasible
  $J = [2,1,4]$

5. 다음 정렬된 job5를 선택하여 K sequence에 deadline이 비내림차순으로 정렬되게 insert해보니 not feasible하여 reject된다. 따라서 J sequence에 추가하지 않는다.

- $K = [2,5,1,4]$ is rejected

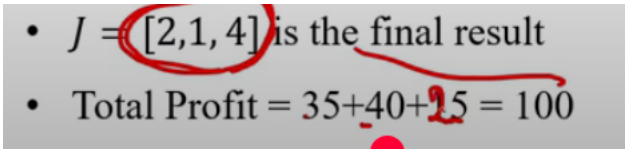
6. 다음 정렬된 Job6를 선택하여 K sequence에 deadline이 비내림차순으로 정렬되게 insert해보니 not feasible하여 reject된다. 따라서 J sequence에 추가하지 않는다.

- $K = [2,1,4,6]$ is rejected

7. 다음 정렬된 job7을 마지막으로 선택하여 K sequence에 deadline이 비내림차순으로 정렬되게 insert해보니 not feasible하여 reject된다. 따라서 J sequence에 추가하지 않으며,

J sequence는 최종적으로 [2, 1, 4]로 마무리짓는다.

따라서 job들의 Total Profit을 계산하면 35 + 40 + 25 로 100이 나오는 것을 알 수 있다.



이상 알고리즘을 만들기까지의 과정이며 아래는 코드들을 나열할 것.

Codes

```cpp
// schedule 함수: 이익 순으로 정렬 후 스케줄 구성
vector<int> schedule(const scheduleList &schedules)
{
    int N = schedules.size();
    // 작업의 원래 인덱스를 저장할 벡터
    vector<int> original_indices(N);
    // 0부터 N-1까지 인덱스로 채우기
    iota(original_indices.begin(), original_indices.end(), 0);

    // 원래 인덱스들을 이익(schedules[index].second) 내림차순으로 정렬
    sort(original_indices.begin(), original_indices.end(), [&](int a, int b)
         { return schedules[a].second > schedules[b].second; });

    // 스케줄된 작업들의 원래 인덱스를 시간 순서대로 저장할 벡터
    vector<int> J;

    // 이익이 큰 작업부터 순회하며 스케줄에 추가 시도
    for (int i = 0; i < N; i++)
    {
        int current_job_original_index = original_indices[i]; // 이익 순으로 고려할 현재 작업의 원래 인덱스

        // 현재 스케줄 J에 이 작업을 삽입했을 때의 가상 스케줄 K 생성
        vector<int> K = insert(J, current_job_original_index, schedules);

        // K가 실행 가능한 스케줄인지 확인
        if (feasible(K, schedules))
        {
            J = K; // 가능하면 J를 K로 업데이트
        }
    }
    return J;
}
```

```cpp
// feasible 함수: 주어진 스케줄 K가 유효한지 확인
bool feasible(const vector<int> &K, const scheduleList &schedules)
{
    // K는 시간 순서대로 정렬된 작업들의 '원래 인덱스' 목록
    // K[i-1]는 시간 i에 스케줄된 작업의 '원래 인덱스'를 의미
    for (int i = 1; i <= K.size(); i++)
    {
        int original_index_at_time_i = K[i - 1]; // 시간 i에 스케줄된 작업의 원래 인덱스
        // 시간 i에 스케줄된 작업의 마감일(schedules[original_index_at_time_i].first)이
        // 현재 시간 i보다 작으면 (즉, 마감일을 넘겼으면) 불가능
        if (i > schedules[original_index_at_time_i].first)
        {
            return false;
        }
    }
    return true; // 모든 작업이 마감일 안에 가능하면 참
}
```

```cpp
// insert 함수: 현재 스케줄 J에 새로운 작업(original_index)을 삽입하여 가상 스케줄 K 생성
vector<int> insert(const vector<int> &J, int current_job_original_index, const scheduleList &schedules)
{
    vector<int> K = J; // J를 복사하여 K 생성
    int j = K.size();  // K의 현재 크기 (새 작업이 들어갈 수 있는 최대 시간 + 1)

    // 삽입할 위치를 찾기 위한 루프
    // K는 시간 순서대로 정렬된 상태라고 가정하고,
    // 현재 작업의 마감일과 K에 있는 작업들의 마감일을 비교하며 삽입 위치를 찾으려는 로직 같아 보임.
    // j는 K의 끝(K.size())부터 1까지 내려가면서 비교.
    // j가 1일 때 K[j-1]은 K[0] (시간 1의 작업)을 의미.
    while (j > 0 && schedules[current_job_original_index].first < schedules[K[j - 1]].first)
    {
        j--; // 현재 작업 마감일이 K[j-1] 작업 마감일보다 작으면, 더 앞쪽 시간으로 이동해서 삽입할 위치를 찾아.
    }
    // 루프가 끝나면 j는 새로운 작업이 삽입될 위치의 인덱스
    // (0 <= j <= K.size())

    // 찾은 위치(인덱스 j)에 현재 작업의 원래 인덱스를 삽입
    K.insert(K.begin() + j, current_job_original_index);

    return K; // 새로운 가상 스케줄 K 반환
}
```