알고리즘및실습

제8장 탐욕적 알고리즘 1부

1. 개요

최적해를 찾는 문제는 가능한 모든 경우를 조사하여 이들을 비교하면 항상 정확한 최적해를 찾을 수 있다. 하지만 조사해야 하는 경우의 수가 너무 많으면 전수조사 알고리즘을 이용하여 주어진 문제를 해결할 수 없다. 보통 분할정복은 시간 복잡도의 수준을 획기적으로 개선하지 못하므로 이때 사용할 수 있는 알고리즘 설계 기법이 아니다. 이와 같은 경우 우리가 사용할 수 있는 알고리즘 설계 기법 중 하나가 이 장에서 살펴보는 **탐욕적**(greedy) 알고리즘 설계 기법이다.

탐욕적 알고리즘은 전수조사하지 않고 반복적으로 해당 시점에서 가장 최선의 선택을 하여 최적해를 만든다. 따라서 탐욕적 알고리즘은 수 많은 경우 중 최적해에 해당하는 경우를 일련의 선택을 통해 구한다. 따라서 매우 효율적으로 문제를 해결할 수 있다. 하지만 모든 최적해 문제를 탐욕적 알고리즘으로 해결할 수 없다. 이 때문에 탐욕적 알고리즘을 고안한 경우 이 알고리즘의 정확성을 증명할 수 있어야 한다. 특정 입력에 대해서는 고안한 알고리즘이 최적해를 줄 수 있지만 실제 사용이 가능하기 위해서는 가능한 모든 입력에 대해 항상 최적해를 주어야 한다. 따라서 탐욕적 알고리즘은 설계하기 쉬울 수 있고, 이렇게 설계된 알고리즘의 시간 복잡도를 분석하기도 쉽지만 정확성 증명은 간단하지 않을 수 있다. 보통 알고리즘의 정확성 증명은 귀납법이나 모순을 이용하여 증명한다.

알고리즘 8.1 일반 탐욕적 알고리즘

```
1: function GREEDY()
2:     S := empty set
3:     while condition do                          ▷ 조건: 선택할 것이 있는지 여부
4:         s := SELECT()                           ▷ 선택 작업
5:         if ISFEASIBLE(s) then                   ▷ 타당성 조사
6:             S.ADD(s)
7:             if ISSOLVED(S) then break           ▷ 해답 조사
8:     return S
```

일반적 탐욕적 알고리즘은 알고리즘 8.1과 같다. 제시된 알고리즘처럼 탐욕적 알고리즘은 다음 3가지 요소로 구성된다.

- 선택 작업(selection procedure): 현 시점에서 최적에 해당하는 것을 선택
- 타당성 조사(feasibility test): 선정한 것을 사용할 수 있는지 여부 검사
- 해답 조사(solution check): 지금까지 선택한 것이 최종해가 되는지 검사

2. 거스름 문제

거스름 문제

- 입력. 동전 액면가 정보, 거스름액
- 출력. 거스름을 구성하기 위해 필요한 최소 동전/지폐의 수

거스름 문제에 대한 탐욕적 알고리즘을 만들어 보자. 탐욕적 알고리즘을 설계할 때 가장 핵심이 되는 것은 선택 작업이다. 이 문제에서는 거스름액이 주어졌을 때 거스름을 주기 위한 동전 또는 지폐를 선택해야 한다. 탐욕적으로 접근하면 거스름과 같거나 작은 가장 큰 액면가 동전을 이용하는 것이다. 이와 같은 선택을 통해 거스름 문제를 해결하는 알고리즘은 알고리즘 8.2와 같다. 이 알고리즘에서 $d[]$는 사용 가능한 동전/지폐의 액면가가 내림차순으로 정렬되어 있는 배열이다. 또 사용할 수 있는 특정 액면가 동전/지폐의 수에 대한 제한은 없다.

알고리즘 8.2 거스름 알고리즘

```
1: function COINCHANGE(d[], change)
2:     count := 0
3:     i := 1
4:     while i ≤ n do
5:         coin := d[i]
6:         if coin ≤ change then
7:             ++count
8:             change −= coin
9:             if change = 0 then break
10:        else++i
11:    return count if change = 0 else −1
```

알고리즘 8.2은 정확한가? 현재 유통되고 있는 동전만을 가지고 탐욕적 알고리즘을 적용하면 이 알고리즘은 정확하다. 예를 들어 100원, 50원, 10원 액면가 동전만 있을 때 210원에 대한 거스름의 최적해는 100원 짜리 2개, 10원 짜리 1개이다. 하지만 120원 짜리 액면가가 있다고 가정하면 위 알고리즘이 주는 답은 120원 짜리 1개, 50원짜리 1개, 10원 짜리 4개를 답으로 준다. 이것은 최적해가 아니므로 위 알고리즘이 정확하지 않다는 것을 알 수 있다. 다음 정리를 통해 상위 액면가 동전을 그다음 액면가 동전이 나눌 수 있는 경우에만 탐욕적 기법을 적용할 수 있다.

정리 8.1. 동전의 액면가를 오름차순으로 d_1, d_2, ..., d_k로 표현하자. 즉, 모든 $1 \le i < k$에 대해 $d_i < d_{i+1}$이다. 이때 $d_1 = 1$이고, 모든 $1 \le i < k$에 대해 $d_i | d_{i+1}$이다. 이 경우에는 탐욕적 기법을 적용하여 거스름 문제를 해결할 수 있다.

증명 동전의 집합이 최소로 구성되어 있다고 하면 모든 $1 \le i < k$에 대해 d_i 액면가 동전이 d_{i+1}/d_i보다 적게 있는 경우를 말한다. 최소로 구성되어 있지 않으면 최적의 구성이 될 수 없다. 그것은 d_{i+1}/d_i개의 액면가 d_i의 동전을 d_{i+1}로 바꿀 수 있기 때문이다. 예를 들어 100원 짜리를 6개 사용하였다면 500원 짜리 1개와 100원 짜리 1개로 바꾸면 더 적은 수의 동전으로 거스름을 구성하게 된다. 이와 같은 조건에서 최소로 구성된 동전이 가질 수 있는 최대 합계는 다음과 같다.

$$\left(\frac{d_2}{d_1} - 1\right) d_1 + \left(\frac{d_3}{d_2} - 1\right) d_2 + \cdots + \left(\frac{d_j}{d_{j-1}} - 1\right) d_{j-1}$$

이 값은 $d_j - d_1 = d_j - 1$이다. 따라서 최소로 구성된 동전 집합이 d_j 이상의 액면가 동전을 하나도 포함하고 있지 않으면 이 집합의 합계는 $d_j - 1$과 같거나 작아야 한다.

문제에 입력이 X이고, $d_j \le X < d_{j+1}$라 하자. X의 최적의 거스름은 최소로 구성해야 한다. 당연한 것이지만 이 거스름 집합에는 d_{j+1}과 같거나 큰 액면가의 동전을 포함할 수 없다. d_{j+1}를 포함하고 있지 않으면 앞에서 증명한

바와 같이 X의 금액은 최대 $d_{j+1} - 1$이다. 따라서 d_j 액면가 동전을 최소 하나 포함해야 한다. 최소 하나 포함되기 때문에 X가 최적이 되기 위해서는 $X - d_j$가 최적이 되어야 한다. 이것은 탐욕적 기법과 정확하게 일치하기 때문에 주어진 조건에서는 탐욕적 기법으로 문제를 해결할 수 있다. □

3. 캐시 교체 문제

캐시는 느린 메모리와 빠른 메모리 사이에 위치하는 용량이 작은 빠른 메모리이며, 이를 통해 느린 메모리에 대한 접근 성능을 향상시켜 준다. 예를 들어 CPU와 주기억장치 사이에 CPU 캐시가 위치하며, 디스크와 주기억장치 사이에 디스크 캐시가 위치한다.

프로그램이 실행되면 프로그램이 사용하는 데이터는 메모리에 위치한다. 이 데이터를 CPU가 처리하기 위해서는 CPU 레지스터로 이 데이터를 가져와야 한다. 이때 해당 데이터를 메모리에서 CPU 레지스터로 바로 옮기지 않고, CPU 캐시에 먼저 저장한다. 데이터를 메모리에서 CPU 캐시로 이동할 때 CPU에서 요구하는 데이터 단위로 이동하지 않고 그보다 큰 단위(페이지 단위)로 옮긴다. 따라서 어떤 데이터를 접근한 다음에 바로 인접한 메모리에 있는 다른 데이터를 접근해야 하면 메모리에 대한 접근 없이 CPU 캐시에서 해당 데이터를 얻을 수 있다. 예를 들어 CPU가 접근하는 데이터가 배열이고, 배열에 있는 요소를 첫 번째 색인부터 차례로 접근할 경우 CPU 캐시 때문에 요소를 접근할 때마다 메모리에 대한 접근이 필요 없다. 배열은 임의 접근을 제공하기 위해 연속된 공간을 차지하고 있으므로 배열의 첫 번째 요소를 접근할 때 배열의 많은 부분이 캐시로 이동하게 되며, 배열에 대한 후속 접근은 메모리에 대한 접근 없이 캐시에서 이루어게 된다. 이 때문에 배열 기반 자료구조에 대한 접근은 연결구조와 달리 캐싱에 유리하기 때문에 더욱 빠르게 처리할 수 있다.

캐시의 용량을 나타낼 때 사용하는 단위는 슬롯이다. 캐시는 슬롯 수만큼의 페이지를 유지할 수 있다. 캐시는 용량이 제한적이기 때문에 필요한 데이터가 캐시에 없으면 기존 페이지 중 하나를 새 페이지로 교체해야 한다. 이처럼 데이터가 있는 페이지가 캐시에 없으면 페이지 부재(page fault)가 발생하였다고 하며, 페이지 부재가 적을수록 캐시 사용에 따른 성능은 더 향상될 수 있다. 캐시 교체 알고리즘 또는 정책은 페이지 부재가 발생하였을 때 캐시에 있는 기존 페이지 중 어떤 페이지 위에 새 페이지를 덮어 쓸지 결정하는 알고리즘을 말한다.

캐시 교체 알고리즘 중 가장 널리 사용하는 알고리즘 중 하나가 LRU(Least Recently Used) 정책이다. 이 알고리즘은 사용한지 가장 오래된 페이지와 새 페이지를 교체한다. 예를 들어 캐시의 슬롯 수가 4라 하자. 필요한 페이지를 알파벳 소문자로 나타낸다고 가정하고, 캐시가 빈 상태에서 ($a\ b\ c\ d\ e\ f\ a\ b$) 순으로 해당 페이지에 있는 데이터를 접근하였다고 하자. 그러면 첫 4개의 요청에 의해 캐시에 [$a\ b\ c\ d$] 4개의 페이지를 유지하게 된다. 그다음 요청인 페이지 e는 캐시에 없으므로 캐시에 있는 기존 페이지 대신에 e를 저장해야 한다. LRU 정책은 [$a\ b\ c\ d$] 중 가장 오래 전에 접근한 a 페이지 대신에 e를 저장하게 된다. 따라서 e와 f에 대한 접근에 의해 캐시의 모습은 [$c\ d\ e\ f$]가 된다. 그다음 요청은 a이므로 다시 페이지 부재가 발생한다. 하지만 페이지 e를 캐시에 저장할 때 a 대신에 c나 d와 교체했으면 페이지 a를 접근할 때 페이지 부재 없이 처리가 가능하다. 그러면 어떤 캐시 교체 알고리즘은 가장 효과적일까?

가장 효과적인 캐시 교체 알고리즘은 FIF(Furthest-In-Future) 알고리즘이다. 이 알고리즘은 페이지 부재가 발생하였을 때 가장 나중에 필요한 페이지와 교체를 한다. 캐시의 슬롯 수가 6이고, 현재 캐시 모습이 [$a\ b\ c\ d\ e\ f$]일 때, ($g\ a\ b\ c\ e\ d\ a\ b\ b\ f$) 순으로 페이지 요청이 일어난다고 가정하자. 이 경우 FIF는 g에 대한 페이지 부재 때 가장 미래에 접근하는 f 페이지와 교체를 하게 된다. 요청이 ($g\ a\ b\ f\ c\ e\ d\ a\ b\ b$) 순이면 d가 가장 미래에 요구하는 페이지이기 때문에 d와 교체를 하게 된다. 요청 순서를 보면 b가 가장 뒤에 있지만 b가 그때 처음 요청되는 것이 아니기 때문에 캐시에 있는 페이지 중 가장 늦게 요청하는 페이지는 d이다. 이 알고리즘은 탐욕적 알고리즘이다. 앞서 살펴본 LRU도 탐욕적 알고리즘이다. 매 순간 특정 기준을 바탕으로 최선의 선택을 한다. 실제 FIF는 가장 효과적인 캐시 교체 알고리즘이다. 하지만 이 알고리즘은 실제 사용할 수 있는 알고리즘은 아니다. 이 알고리즘은 미래의 접근 순서를 알아야 한다. 더욱이 미래의 접근 순서의 끝이 있는 것도 아니다. 따라서 실제 사용할 수 있는 캐시 정책은 아니다. 이처럼 실용적으로 사용할 수 없는 알고리즘은 쓸모가 없는 알고리즘이라고 생각할 수 있는데,

이 알고리즘은 성능 분석할 때 이상적인 기준으로 활용할 수 있기 때문에 전혀 활용도가 없는 알고리즘은 아니다.

4. 스케줄링

이전 절에서 살펴본 캐시 교체 문제와 이 절에서 살펴보는 스케줄링 문제는 컴퓨터공학의 중요 교과인 컴퓨터구조, 운영체제에서 실제 학습하는 문제이며, 컴퓨터 동작에 꼭 필요한 요소이다. 물론 스케줄링은 운영제체에서만 필요한 것은 아니며, 실 생활에서 여러 가지 형태로 나타나는 문제이다. 예를 들어 한 기업에 회의실이 하나 밖에 없는데, 여러 회의를 진행해야 하면 한 번에 여러 회의를 동시에 진행할 수 없으므로 어떤 형태의 스케줄링은 반드시 필요하다. 컴퓨터의 경우 실행해야 할 작업이 여러 개 있지만 CPU는 하나밖에 없으면 이 작업을 실행할 순서를 결정해야 하며, 어떤 순서로 하는 것이 가장 효과적인지는 목적에 따라 다를 수 있다. 하지만 n개 작업이 있으면 가능한 작업 스케줄은 $n!$이다. 따라서 n이 조금만 커도 전수조사하여 최적의 순서를 찾는 것은 가능하지 않다.

4.1 시스템 내부 총 시간 최소화하기

시스템 내부 총 시간 최소화 문제

- 입력. n개의 작업의 작업 길이 $l_1, l_2, \ldots, l_n$
- 출력. 시스템 내부 총 시간을 최소화하는 작업 실행 순서

여기서 시스템 내부 총 시간이란 작업의 대기 시간과 처리 시간을 합한 시간을 말하며, 다른 말로 완료 시간(completion time)이라 한다. 이 문제에서 모든 작업은 현재 모두 대기중인 상태이다. 즉, 모든 작업은 실행이 준비된 상태이다.

예를 들어 $l_1 = 1, l_2 = 2, l_3 = 3$이면 j_1, j_2, j_3 순서로 실행하면 $c_1 = 1, c_2 = 1 + 2 = 3, c_3 = 3 + 3 = 6$이므로 전체 시간은 10이 소요된다. 순서를 j_3, j_2, j_1 순서로 실행하면 $c_1 = 5 + 1 = 6, c_2 = 3 + 2 = 5, c_3 = 3$이므로 전체 시간은 14가 소요된다. 이 문제는 길이가 가장 짧은 순서로 실행하면 시스템 내부 총 시간을 최소화할 수 있다. 길이가 가장 짧은 순서로 실행하는 알고리즘은 알고리즘 8.3과 같다.

알고리즘 8.3 시스템 내부 총 시간 최소화 알고리즘

```
1: function SCHEDULE(l[])
2:     SORT(l)                          ▷ 작업 길이를 기준으로 오름차순으로 정렬
3:     total := 0
4:     c := 0
5:     for i := 1 to n do
6:         c+ = l[i]
7:         total += c
8:     return total
```

이 알고리즘의 시간 복잡도는 정렬 비용 때문에 $O(n \log n)$이다. 알고리즘에서 **for** 문은 n 번 반복되며, 반복할 때 일정 비용이 소요되므로 $O(n)$이므로 전체 비용은 정렬 비용에 좌우된다. 공간 복잡도는 $O(n)$이다. 앞서 언급한 바와 같이 탐욕적 알고리즘은 항상 정확성을 증명해야 한다. 이 알고리즘의 정확성은 다음 정리를 이용하여 증명할 수 있다.

정리 8.2. 알고리즘 8.3을 이용하면 전체 시스템 내부 시간이 최소화된다.

증명 작업의 길이를 기준으로 리스트를 오름차순으로 정렬한 다음에 가장 작은 길이의 작업부터 차례로 작업 번호를 부여하여 수행한 스케줄을 $\sigma = [j_1, j_2, ..., j_n]$라 하자. 이 스케줄과 다른 전체 시스템 내부 시간을 더 최소화하는

최적 스케줄링 σ^*가 있다고 하자. σ^*는 σ와 다른 스케줄이기 때문에 색인이 $k > i$이지만 j_k, j_i 순으로 실행된 두 개의 작업이 존재할 수밖에 없다. 이때 j_k와 j_i 사이에 있는 작업의 집합이 S라 하고, 이 두 작업의 순서를 바꾼 스케줄을 σ'이라 하자. 그러면 $c'_k = c^*_k + l_i + \sum_{s \in S} l_s$이고, $c'_i = c^*_i - \sum_{s \in S} l_s - l_k$이므로 전체 시스템 내부 시간은 $T_{\sigma^*} = T_{\sigma'} - l_k + l_i$이다. 그런데 $l_k > i_i$이므로 $T_{\sigma^*} > T_{\sigma'}$이다. 따라서 σ^*가 최적의 스케줄이라는 것은 모순이다. 그러므로 σ가 최적의 스케줄링일 수밖에 없다. □

4.2 가중 완료 총 시간 최소화

가중 완료 총 시간 최소화 문제

- 입력. n개의 작업 (l_i, w_i). $l_i(>0)$는 작업의 길이이고, $w_i(>0)$는 작업의 가중치이며, 가중치가 클수록 우선순위가 높은 작업에 해당함
- 출력. $\sum_{i=1}^{n} w_i c_i$를 최소화하는 작업 실행 순서, c_i는 작업의 완료 시간임

이 문제는 각 작업의 우선순위가 있는 경우이다. $\sum_{i=1}^{n} w_i c_i$을 최소화한다는 것은 우선순위가 높은 것은 가급적 먼저 수행하면서 전체 시스템 내부 시간을 최소화하고 싶다는 것을 의미한다. 예를 들어 $l_1 = 1$, $l_2 = 2$, $l_3 = 3$이고, $w_1 = 3$, $w_2 = 2$, $w_3 = 1$일 때 j_1, j_2, j_3 순서로 작업을 실행하면 $w_1 c_1 = 1 \times 3 = 3$, $w_2 c_2 = 2 \times 3 = 6$, $w_3 c_3 = 1 \times 6 = 6$이므로 $\sum_{i=1}^{n} w_i c_i = 15$이다. 반면에 j_3, j_2, j_1 순서로 작업을 실행하면 $w_1 c_1 = 1 \times 6 = 6$, $w_2 c_2 = 2 \times 5 = 10$, $w_3 c_3 = 1 \times 3 = 3$이므로 $\sum_{i=1}^{n} w_i c_i = 19$이다.

이 문제에서 가중치가 모두 같으면 시스템 내부 총 시간 최소화와 같은 문제가 된다. 또 작업 시간이 모두 같으면 우선순위가 높은 것을 먼저 실행하여 목표한 값을 최소화할 수 있다. 따라서 이 문제는 작업 시간이 작은 것과 우선순위가 높은 것을 우선해야 한다.

이 문제를 탐욕적 기법을 이용하여 해결하기 위해서는 작업을 선택하는 기준이 필요하다. 앞서 설명한 바와 같이 작업 시간이 작을수록, 우선 순위는 높을수록 우선해야 한다. 이를 바탕으로 작업 선택 기준 함수를 설계하면 후보로 $f(w_i, l_i) = w_i - l_i$와 $f(w_i, l_i) = w_i / l_i$ 등을 생각해 볼 수 있다. 둘 다 작업 시간이 작을수록, 우선 순위가 높을수록 함수의 결과 값이 커진다.

간단한 예에 두 함수를 적용하여 문제를 해결하여 보자. $l_1 = 2$, $l_2 = 1$이고, $w_1 = 5$, $w_2 = 2$인 경우 첫 번째 방법을 적용하면 $f(w_1, l_1) = 5 - 3 = 2$이고, $f(w_2, l_2) = 2 - 1 = 1$이다. 따라서 j_2, j_1 순서로 실행하게 되며 $sum_{i=1}^{n} w_i c_i = 3 \times 7 + 1 = 23$이다. 같은 예에 두 번째 방법을 적용하면 $f(w_1, l_1) = \frac{3}{5}$이고, $f(w_2, l_2) = \frac{1}{2}$이다. 따라서 j_1, j_2 순서로 실행하게 되며 $sum_{i=1}^{n} w_i c_i = 3 \times 5 + 1 = 22$이다. 결과적으로 첫 번째 방법은 최적해를 주지 않는다는 것을 알 수 있다. 한 가지 반례만 찾으면 해당 방법은 정확하지 않다는 것이 증명된다. 그러면 두 번째 방법을 사용하면 알고리즘은 정확한가? 이것을 확신하기 위해서는 증명이 필요하다. 증명하기 전에 두 번째 방법을 이용한 알고리즘은 알고리즘 8.4와 같다. 이 알고리즘의 시간 복잡도는 시스템 내부 총 시간 최소화 알고리즘과 마찬가지로 $O(n \log n)$이다.

알고리즘 8.4 가중 완료 총 시간 최소화 알고리즘

```
1: function SCHEDULE(J[])                    ▷ J은 (w_i, l_i)쌍 배열
2:     SORT(J)                               ▷ w_i/l_i 기준으로 내림차순으로 정렬
3:     total := 0
4:     c := 0
5:     for i := 1 to n do
6:         c += J[i].l
7:         total += J[i].w × c
8:     return total
```

정리 8.3. 알고리즘 8.4을 이용하면 $\sum_{i=1}^{n} w_i c_i$ 값이 최소화된다.

증명 w_i/l_i를 기준으로 리스트를 내림차순으로 정렬한 다음에 w_i/l_i 값이 가장 큰 작업부터 차례로 작업 번호를 부여하여 수행한 스케줄을 $\sigma = [j_1, j_2, ..., j_n]$라 하자. 이 스케줄과 다른 $\sum_{i=1}^{n} w_i c_i$를 더 최소화하는 최적 스케줄링 σ^*가 있다고 하자. σ^*는 σ와 다른 스케줄이기 때문에 색인이 $k > i$이지만 j_k, j_i 순으로 실행된 작업들이 존재할 수밖에 없다. 이때 j_k와 j_i 사이에 있는 작업의 집합이 S라 하고, 이 두 작업의 순서를 바꾼 스케줄을 σ'이라 하자. 그러면 작업의 완료시간 $c'_k = c^*_k + l_i + \sum_{s \in S} l_s$이고, $c'_i = c^*_i - \sum_{s \in S} l_s - l_k$이므로 전체 시스템 내부 시간 $T_{\sigma^*} = T_{\sigma'} + w_i l_k - w_k l_i$이다. 그런데 $w_i/l_i > w_k/l_k$이므로 $w_i l_k > w_k l_i$이다. 따라서 $T_{\sigma^*} > T_{\sigma'}$이므로 σ^*가 최적의 스케줄이라는 것은 모순이다. 그러므로 σ가 최적의 스케줄링일 수밖에 없다. □

4.3 마감 시간이 있는 스케줄링

마감 시간이 있는 최적 스케줄링 문제

- 입력. n개의 작업 (d_i, w_i). $d_i(> 0)$는 작업의 마감 시간이고, $w_i(> 0)$는 작업의 이득임
- 출력. $\sum_{i \in \sigma} w_i$의 최대가 되는 작업 실행 순서. 여기서 σ는 스케줄된 작업 목록임

이 문제에서 모든 작업은 대기중인 상태이며, 모든 작업의 길이는 1이다. 이 문제는 스케줄링 가능한 것 중 총 이득을 최대화하는 것이다. 작업의 실행을 통해 작업의 이득을 얻기 위해서는 작업의 마감 시간을 포함하여 그 이전에 작업을 실행해야 한다. 마감 시간 이후로 처리된 작업은 이득이 0이 되므로 고려할 필요가 없다. 따라서 모든 작업을 실행해야 하는 것은 아니며, 각 작업의 마감 시간 이전에 최대한 많은 작업을 실행하여 가능한 많은 이득을 얻는 것이 목적이다.

예를 들어 $(2, 30)$, $(1, 35)$, $(2, 25)$, $(1, 40)$ 4개의 작업이 주어진 경우 가능한 스케줄은 총 다음 6가지가 존재한다.

- j_1, j_3: $30 + 25 = 55$
- j_2, j_1: $35 + 30 = 65$
- j_2, j_3: $35 + 25 = 60$
- j_3, j_1: $25 + 30 = 55$
- j_4, j_1: $40 + 30 = 70$
- j_4, j_3: $40 + 25 = 65$

이 중에 j_4, j_1 순으로 실행하는 스케줄이 최적이다. 가장 이득이 높은 작업은 최적해에 포함되어 있지만 그다음 이득이 높은 것은 포함되어 있지 않다. 이 관찰을 통해 이득을 기준으로 내림차순한 다음, 가장 이득이 높은 것부터 차례로 스케줄 가능한지 검사하여 작업을 선택하는 탐욕적 방법을 통해 이 문제의 해결이 가능하다고 생각할 수 있다. 지금까지의 모든 탐욕적 알고리즘처럼 어떤 탐욕적 알고리즘을 설계하면 그다음에는 정확성을 증명해야 한다.

정확성을 증명하기 위해 타당한 순서(feasible sequence)와 타당한 집합이라는 개념을 이용한다. 타당한 순서란 주어진 순서로 실행 가능한 스케줄을 말한다. 이전 예에서 $[4, 1]$은 타당한 순서이지만 $[1, 4]$는 타당한 순서가 아니다. 타당한 순서가 주어지면 이 순서에 포함된 작업 목록이 있을 것이다. 이를 타당한 집합이라 하자. $\{1, 4\}$는 타당한 집합이지만 $\{2, 4\}$는 타당한 집합이 아니다. $\{2, 4\}$는 어떤 순서로 실행하여도 타당한 순서가 아니다. 주어진 작업 목록이 타당한 집합이 되기 위한 필요충분조건은 이 집합에 있는 작업을 마감 시간을 기준으로 오름차순으로 스케줄링하였을 때 이 순서가 타당한 순서이어야 한다.

정리 8.4. 주어진 작업 집합 S가 타당한 집합이기 위한 필요충분조건은 이 집합에 있는 작업을 마감 시간을 이용하여 오름차순으로 정렬하여 얻은 실행 순서가 타당한 순서이어야 한다.

증명 마감 시간을 이용하여 오름차순으로 정렬하여 얻은 실행 순서가 타당하면 이 집합은 타당한 집합이다. S가 타당한 집합이면 이 작업에 대한 타당한 순서가 존재한다. $x, y \in S$에 대해 $\sigma = [\ldots, x, y, \ldots]$가 실행 가능한 순서라 하고, x의 마감 시간이 y보다 크다고 하자. 이 경우 두 작업의 순서를 바꾸어도 이 순서로 여전히 실행이 가능하다. 그 이유는 y는 x보다 마감 시간이 작으므로 x가 실행된 위치에서 실행할 수 있으며, x는 y보다 마감 시간이 크므로 y가 실행된 위치에서 실행할 수 있다. 따라서 이 과정을 반복하면 오름차순으로 정렬된 스케줄을 얻을 수 있으며, 이 스케줄은 실행 가능한 스케줄이다. □

알고리즘 8.5 마감 시간이 있는 최적 스케줄링 알고리즘

```
1: function SCHEDULE(J[])                    ▷ J은 (d_i, w_i)쌍 배열
2:     SORT(J)                               ▷ d_i 기준으로 내림차순으로 정렬
3:     S := {1}
4:     for j := 2 to n do
5:         K := S.ADD(j)
6:         if K is feasible then S := K
7:     return S
```

이 증명 자체가 알고리즘 8.5에 제시된 탐욕적 알고리즘이 정확하다는 것을 증명해 주지는 않는다. 하지만 탐욕적 알고리즘의 구현 방법은 제공해 준다. 위 알고리즘에서 작업을 S 집합에 포함한 다음 이 집합이 타당한 집합인지 검사할 수 있어야 한다. 이 검사를 하는 방법을 위 증명이 제시해 주고 있다. 타당한 집합이 되기 위한 필요충분조건은 집합에 있는 작업을 마감 시간을 이용하여 오름차순으로 정렬한 다음 스케줄이 가능한지 검사하면 된다. 이것을 직관적으로 구현하면 **for** 문이 반복할 때마다 정렬을 해야 한다.

하지만 매번 정렬하지 않고 더 간단히 구현하는 방법이 있다. 마감 시간을 기준으로 버킷 정렬을 하는 것이다. 주어진 작업의 마감 시간 중 가장 늦은 시간이 m이면 m 크기의 **boolean** 배열을 준비한다. 이 배열의 모든 항을 **false**로 초기화한 후에 새 작업을 선택할 때마다 그 작업의 마감 시간부터 1까지 내려가면서 **false**인 항이 있으면 해당 시간에 작업을 실행하는 것으로 할당하면 된다. 이때 이 작업에 할당할 작업 수행 시간이 없으면 이 작업은 타당한 집합에 포함할 수 없는 작업이 된다.

이와 같은 방법으로 타당성 조사를 할 경우 전체 시간 복잡도를 분석하여 보자. J를 최초로 정렬하는 비용은 $O(n \log n)$이다. 그다음 **for** 문은 $n-1$개 작업마다 최대 m 슬롯을 검사해야 할 수 있다. 따라서 **for** 문의 비용은 $O(nm)$이다. 따라서 전체 시간 복잡도는 $O(nm + n \log n)$이다.

이제 이 알고리즘의 정확성을 증명하여 보자. 참고로 타당한 집합이 주어졌을 때 스케줄 순서는 전체 이득에 영향을 주지 않는다. 해당 집합에 있는 작업을 어떤 타당한 순서로 실행하더라도 전체 이득은 변하지 않는다.

정리 8.5. 알고리즘 8.5을 이용하면 $\sum_{i \in \sigma} w_i$ 값이 최대화된다.

증명 귀납법을 이용하여 증명한다.

- 귀납 출발: 작업이 하나 밖에 없으면 이 정리는 당연히 성립한다.
- 귀납 가정: 첫 k개 작업에 대해 이 알고리즘이 주는 답은 최적이다.
- 귀납 절차: $k+1$개 작업에 대해 이 알고리즘이 주는 답은 최적임을 귀납 가정을 이용하여 증명해야 한다.

σ가 $k+1$개 작업에 대해 이 알고리즘이 주는 결과 집합이라 하고, σ^*가 최적의 집합이라 가정하자. 또 j_i는 이득 순으로 정렬된 집합에서 i번째 작업이라 하자. 두 가지 경우에 대한 고려가 필요하다.

- 경우 1. $j_{k+1} \notin \sigma^*$
- 경우 2. $j_{k+1} \in \sigma^*$

경우 1에서 σ는 귀납 가정에 의해 첫 k 작업에 대한 최적해를 포함하고 있으므로 $\sigma >= \sigma^*$가 성립한다. 따라서 σ는 최적해이다. 경우 2는 다시 $j_{k+1} \in \sigma$와 $j_{k+1} \notin \sigma$인 경우로 나누어질 수 있다. 전자의 경우에는 귀납 가정에 의해 σ는 j_{k+1}를 고려하지 않았을 때 최적이고, 둘 다 j_{k+1}를 포함하고 있으므로 $\sigma >= \sigma^*$가 성립한다. 후자의 경우는 σ^*은 j_{k+1}를 포함하였고, σ는 포함하지 않은 경우이다. σ^*에서 j_{k+1}를 할당한 타임 슬롯을 생각하여 보자. σ에서 해당 슬롯이 비어 있다면 당연히 알고리즘에 의해 σ도 j_{k+1}를 해당 슬롯에 할당하였을 것이다. 따라서 σ는 해당 슬롯에 다른 작업을 할당한 상태가 된다. 이 작업을 j_1이라 하자. j_1이 σ^*에 포함되어 있다면 σ^*에서 j_1이 차지하는 타임 슬롯은 같은 이유로 σ에서 비어있지 않고 어떤 작업이 차지하고 있을 것이다. 이 작업을 j_2라 하자. 이 과정은 아래 그림처럼 계속 반복될 수 있다.

타임슬롯	$\cdots$							$\cdots$	
σ^*	$\cdots$		j_3	j_1		j_2	j_{k+1}	$\cdots$	
σ	$\cdots$		j_4	j_2		j_3	j_1	$\cdots$	
σ'	$\cdots$		j_4	j_2		j_3	j_1	$\cdots$	

하지만 작업의 수는 유한하기 때문에 이것이 무한히 반복될 수는 없다. 결국에는 $j_s \in \sigma$이지만 $j_s \notin \sigma^*$를 만나게 된다. 만약 만나지 않게 된다면 $\sigma \subseteq \sigma^*$이 성립한다는 것을 의미하므로 이 반복과정에서 만난 σ의 작업들을 σ^*와 동일하게 위치를 바꾸어 j_{k+1}를 σ에 할당할 수 있다는 것을 의미하므로 $j_s \in \sigma$이지만 $j_s \notin \sigma^*$가 존재할 수밖에 없다. 그러면 σ^*에서 j_{k+1}를 제거하고 σ와 동일하게 j_1부터 j_{s-1}까지 재배치하고, j_s를 σ^*에 할당할 수 있다(위 그림에서 $j_s = j_4$라고 가정). 그런데 $w_s > w_{j+1}$이므로 σ^*가 최적해라는 것에 모순된다. 따라서 σ는 최적해일 수밖에 없다. □

퀴즈

1. 탐욕적 알고리즘과 관련된 다음 설명 중 틀린 것은?

   ① 매 단계 선택 작업에서 사용할 선택 기준을 결정해야 한다.
   ② 탐욕적 알고리즘은 반드시 정확성 증명이 필요하다.
   ③ 선택 기준이 잘못된 것인지는 반례를 하나만 찾으면 된다.
   ④ 시간 복잡도를 분석하기 어렵다.

2. 캐시 교체 알고리즘 중 Furthest-in-future 알고리즘이 있다. 이 알고리즘은 가장 나중에 필요한 것을 교체 대상으로 선정한다. 이 알고리즘은 미래를 알아야 하기 때문에 실제 활용할 수 있는 알고리즘은 아니다. 하지만 실제 사용하는 알고리즘의 성능을 비교 분석하기 위해 널리 사용한다. 캐시 슬롯이 6개이고 [a b c d e f] 6개 페이지가 캐시에 저장되어 있다고 가정하자. 다음 요청하는 순서가 [g a b c e f d a b] 순일 때 g와 교체되는 페이지는?

   ① b
   ② d
   ③ f
   ④ a

3. 마감 시간이 있는 스케줄 문제에서 다음과 같이 4개의 작업이 주어졌다.

   $$j_1 : (2, 25), j_2 : (1, 40), j_3 : (2, 20), j_4 : (1, 30)$$

   작업 정보 (a, b) 중 a가 마감 시간이고, b가 마감 시간 전에 작업을 수행하였을 얻어지는 이득이다. 다음 중 타당한 집합이 아닌 것은?

   ① $\{1, 4\}$
   ② $\{1, 3\}$
   ③ $\{1, 2\}$
   ④ $\{2, 4\}$

연습문제

1. 음이 아닌 일련의 정수가 주어졌을 때, 이 수들을 조합하여 만들 수 있는 가장 큰 수를 찾아라. 예를 들어 $[10, 2]$가 주어지면 102, 210 두 개의 수를 만들 수 있는데, 이 중 가장 큰 수는 210이다.

2. 일렬로 나열된 정수로 표현되는 행성이 주어진다. 주어진 정수의 절대값이 행성의 크기를 나타내고, 부호가 행성의 이동 방향을 나타낸다. 양수이면 오른쪽으로 이동하고, 음수이면 왼쪽으로 이동한다. 모든 행성은 같은 속도로 이동한다고 가정한다. 두 행성이 충돌하면 작은 크기의 행성은 폭파하여 사라진다. 두 행성의 크기가 같으면 둘 다 폭파한다. 같은 방향으로 이동하는 행성은 절대 충돌하지 않는다. 주어진 일련의 행성이 주어졌을 때 모든 충돌이 일어난 후 남은 행성이 어떻게 되는지 제시하라.

3. 영소문자로만 구성된 문자열이 주어진다. 이 문자열에서 각 문자가 정확하게 한번만 등장하도록 중복된 모든 문자를 제거하라. 제거된 결과는 사전 순서에서 가장 앞에 위치하는 문자열이어야 한다. 예를 들어 "bcabc"가 주어지면 각 문자가 정확하게 하나만 남도록 중복된 문자를 제거한 결과로 "bca", "cab", "bac", "abc"를 얻을 수 있는데, "abc"가 사전 순서에서 가장 앞에 등장하므로 "abc"를 결과로 주어야 한다.