Chapter 4-2

Greedy Algorithm

부분 배낭 문제

- 배낭 (Knapsack) 문제
  - n개의 물건이 각각 1개씩 있고,
  - 각 물건은 무게와 가치를 가지고 있으며,
  - 배낭이 한정된 무게의 물건들을 담을 수 있을 때,
  - 최대의 가치를 갖도록 배낭에 넣을 물건들을 정하는 문제
- 부분 배낭 (Fractional Knapsack) 문제
  - 물건을 부분적으로 담는 것을 허용
  - Greedy algorithm으로 해결

부분 배낭 문제

- 아이디어
  - 부분 배낭 문제에서는 물건을 부분적으로 배낭에 담을 수 있으므로, 최적해를 위해서 '욕심을 내어 ' 단위 무게 당 가장 값 나가는 물건을 배낭에 넣고, 계속해서 그 다음으로 값 나가는 물건을 넣음
  - 만일 물건을 '통째로' 배낭에 넣을 수 없으면, 배낭에 넣을 수 있을 만큼만 물건을 부분적으로 배낭에 담음

부분 배낭 문제

- 알고리즘
  - FractionalKnapsack
  - 입력: n개의 물건, 각 물건의 무게와 가치, 배낭의 용량 C
  - 출력: 배낭에 담은 물건 리스트 L과 배낭 속의 물건 가치의 합 v

1. 각 물건에 대해 단위 무게 당 가치를 계산함
2. 물건들을 단위 무게 당 가치를 기준으로 내림차순으로 정렬하고, 정렬된 물건 리스트를 S라고 정의
3. L= ∅, w=0, v=0 // L은 배낭에 담을 물건 리스트, w는 배낭에 담긴 물건들의 무게의 합, v는 배낭에 담긴 물건들의 가치의 합
4. S에서 단위 무게 당 가치가 가장 큰 물건 x를 가져옴

부분 배낭 문제

- 알고리즘
  - FractionalKnapsack

```
5. While w + (x의 무게) <= C
6.      x를 L에 추가
7.      w = w + (x의 무게)
8.      v = v + (x의 가치)
9.      x를 S에서 제거
10.     S에서 단위 무게 당 가치가 가장 큰 물건 x를 가져옴
11. If C-w > 0   // 배낭에 물건을 부분적으로 담을 여유가 있으면
12.     물건 x를 (C-w) 만큼만 L에 추가
13.     v = v + (C-w) 만큼의 x의 가치
14. return L, v
```

부분 배낭 문제

- 수행 과정
  - 배낭의 최대 용량 = 40그램

  - 단위 무게 당 가치로 정렬: S=[백금, 금, 은, 주석]

물건	단위 그램당 가치
백금	6만원
금	5만원
은	4천원
주석	1천원

부분 배낭 문제

- 수행 과정
  - 백금을 통째로 담는다.
  - 배낭에 담긴 물건(들)의 무게 w = 10, 얻는 가치 v = 60

  - 금을 통째로 담는다
  - 배낭에 담긴 물건(들)의 무게 w = 25, v = 60+75 = 135

부분 배낭 문제

- 수행 과정
  - 은을 통째로 담으려 하지만
  - 배낭에 담긴 물건(들)의 무게 w = 25 + 25 = 50이 되어 배낭 용량 초과

  - 은을 40 – 25 = 15 만큼만 담는다
  - 배낭에 담긴 물건(들)의 무게 w = 40,

    v = 135 + (0.4X15) = 141 만원

부분 배낭 문제

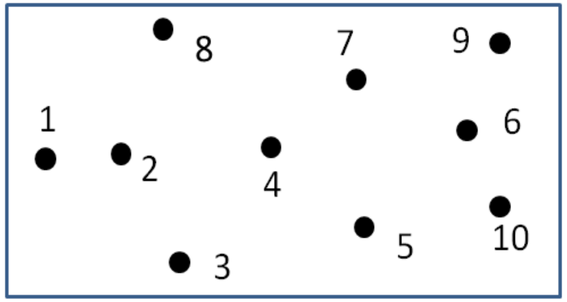
- 시간 복잡도
  - Line 1: n개의 물건 각각의 단위 무게 당 가치를 계산하는 데는 O(n) 시간 소요
  - Line 2: 물건의 단위 무게 당 가치에 대해서 정렬하기 위해 O(nlogn) 시간 소요
  - Line 5~10: while-루프의 수행은 n번을 넘지 않으며, 루프 내부의 수행은 O(1) 시간 소요
  - Line 11~14: 각각 O(1) 시간 소요
  - 알고리즘의 시간 복잡도: O(n) + O(nlogn) + n•O(1) + O(1) = O(nlogn)

집합 커버 문제

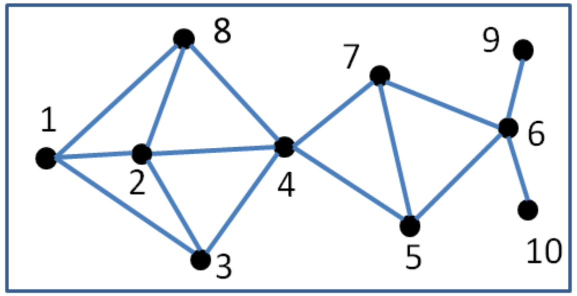
- 문제
  - n개의 원소를 가진 집합 U가 있고,
  - U의 부분집합들을 원소로 하는 집합 F가 주어질 때,
  - F의 원소들인 집합들 중에서 어떤 집합들을 선택하여 합집합하면 U와 같게 될까?
- 집합 커버 (Set Cover) 문제
  - 집합 F에서 선택하는 집합들의 수를 최소화하는 문제

집합 커버 문제

- 신도시 학교 배치
  - 신도시를 계획하는 데 있어서 학교 배치의 예
    - 10개의 마을이 신도시에 만들어질 계획
    - 다음 조건이 만족되도록 학교 위치를 선정해야 함
      - ✓ 학교는 마을에 위치해야 함
      - ✓ 등교 거리는 걸어서 15분 이내여야 함
    - 어느 마을에 학교를 신설해야 학교의 수가 최소가 되는가?



10개 마을에 위치

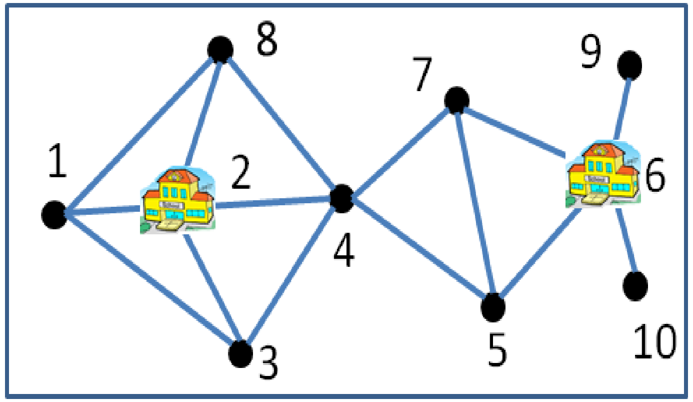


등교 거리가 15분 이내인 마을 간의 관계

집합 커버 문제

- 신도시 학교 배치의 최적 해 (optimal solution)
  - 어느 마을에 학교를 신설해야 학교의 수가 최소가 되는가?
    - 2번 마을에 학교를 만들면 1, 2, 3, 4, 8 마을의 학생들이 15분 이내의 등교 가능
      - 즉, 마을 1, 2, 3, 4, 8이 커버됨
    - 6번 마을에 학교를 만들면 마을 5, 6, 7, 9, 10이 커버됨
    - 2번과 6번 마을에 학교를 배치하면 모든 마을이 커버됨

최소의 학교 수 = 2개



집합 커버 문제

- 신도시 계획 문제를 집합 커버 문제로 변환

  $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ // 신도시의 마을 10개

  $F = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$

  // S_i는 마을 i에 학교를 배치했을 때 커버되는 마을의 집합

  $S_1 = \{1, 2, 3, 8\}$ $S_5 = \{4, 5, 6, 7\}$ $S_9 = \{6, 9\}$

  $S_2 = \{1, 2, 3, 4, 8\}$ $S_6 = \{5, 6, 7, 9, 10\}$ $S_{10} = \{6, 10\}$

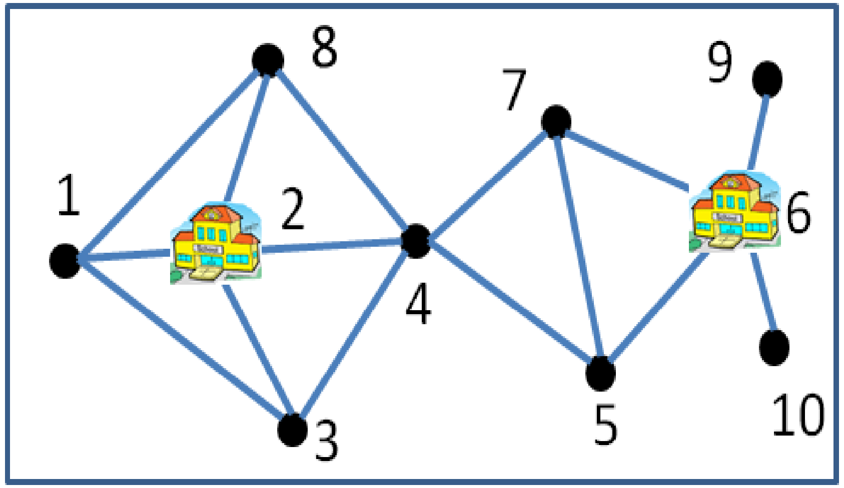
  $S_3 = \{1, 2, 3, 4\}$ $S_7 = \{4, 5, 6, 7\}$

  $S_4 = \{2, 3, 4, 5, 7, 8\}$ $S_8 = \{1, 2, 4, 8\}$

  - S_i 집합들 중에서 어떤 집합들을 선택해야 그들의 합집합이 U와 같은가?
    (단, 선택된 집합의 수는 최소이어야 함)

집합 커버 문제

- 신도시 학교 배치의 최적 해 (optimal solution)
  - $S_2 \cup S_6 = \{1, 2, 3, 4, 8\} \cup \{5, 6, 7, 9, 10\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} = U$



집합 커버 문제

- 단순한 해결 방법
  - 집합 커버 문제의 최적해는 어떻게 찾아야 되는가?
    - F에 n개의 집합들이 있다고 가정
  - 가장 단순한 방법
    - F에 있는 집합들의 모든 조합을 1개씩 합집합하여 U가 되는지 확인
    - U가 되는 조합의 집합 수가 최소인 것을 찾음
    - $F=\{S_1, S_2, S_3\}$일 경우의 모든 조합
      - $S_1, S_2, S_3, S_1 \cup S_2, S_1 \cup S_3, S_2 \cup S_3, S_1 \cup S_2 \cup S_3$
      - 집합이 1개인 경우 3개 = ${}_3C_1$
      - 집합이 2개인 경우 3개 = ${}_3C_2$
      - 집합이 3개인 경우 1개 = ${}_3C_3$, 총합은 3+3+1 = 7 = 2^3-1개

집합 커버 문제

- 단순한 해결 방법
  - n개의 원소가 있을 경우
    - 최대 (2^n-1)개를 검사해야,
    - n이 커지면 최적해를 찾는 것은 실질적 불가능
  - 이런 점에 대한 극복 방법
    - 최적해를 찾는 대신에 최적해에 근접한 근사해 (approximation solution)를 찾음

집합 커버 문제

- 알고리즘
  - SetCover

입력: U, F = $\{S_i\}$, i=1,…,n

출력: 집합 커버 C

1. C = Φ
2. while U ≠ Φ
3. U의 원소를 가장 많이 가진 집합 S_i를 F에서 선택
4. U = U – S_i
5. S_i를 F에서 제거하고, S_i를 C에 추가
6. return C

집합 커버 문제

- 수행과정

$U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$F = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$

$S_1 = \{1, 2, 3, 8\}$ $S_5 = \{4, 5, 6, 7\}$ $S_9 = \{6, 9\}$

$S_2 = \{1, 2, 3, 4, 8\}$ $S_6 = \{5, 6, 7, 9, 10\}$ $S_{10} = \{6, 10\}$

$S_3 = \{1, 2, 3, 4\}$ $S_7 = \{4, 5, 6, 7\}$

$S_4 = \{2, 3, 4, 5, 7, 8\}$ $S_8 = \{1, 2, 4, 8\}$

집합 커버 문제

- 수행과정
  - U의 원소를 가장 많이 커버하는 집합
    - S_4 = {2, 3, 4, 5, 7, 8} 선택
  - $U = U - S_4 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} - \{2, 3, 4, 5, 7, 8\} = \{1, 6, 9, 10\}$
  - S_4를 F에서 제거
    - $F = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\} - \{S_4\} = \{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}\}$
  - S_4를 C에 추가
    - $C = \{S_4\}$

집합 커버 문제

- 수행과정
  - $U = \{1, 6, 9, 10\}$을 가장 많이 커버하는 집합
    - $S_6 = \{5, 6, 7, 9, 10\}$ 선택
  - $U = U - S_6 = \{1, 6, 9, 10\} - \{5, 6, 7, 9, 10\} = \{1\}$
  - S_6를 F에서 제거
    - $F = \{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}\} - \{S_6\} = \{S_1, S_2, S_3, S_5, S_7, S_8, S_9, S_{10}\}$
  - S_6를 C에 추가
    - $C = \{S_4, S_6\}$

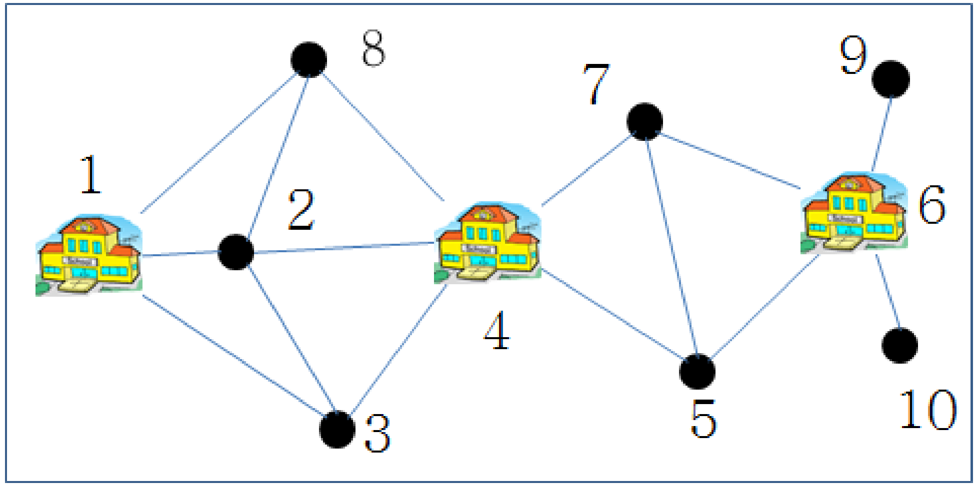
집합 커버 문제

- 수행과정
  - U = {1}을 가장 많이 커버하는 집합
    - S_1 = {1, 2, 3, 8} 선택
  - $U = U - S_1 = \{1\} - \{1, 2, 3, 8\} = \{\}$
  - S_1를 F에서 제거
    - $F = \{S_1, S_2, S_3, S_5, S_7, S_8, S_9, S_{10}\} - \{S_1\} = \{S_2, S_3, S_5, S_7, S_8, S_9, S_{10}\}$
  - S_1을 C에 추가
    - $C = \{S_4, S_6, S_1\}$

집합 커버 문제

- 수행과정
  - Line 6: C = {S_4, S_6, S_1} 리턴

SetCover 알고리즘의 최종해



집합 커버 문제

- 시간 복잡도
  - while-loop가 수행되는 횟수: 최대 n회
    - 루프가 1회 수행될 때마다 집합 U의 원소 1개씩만 커버된다면, 최악의 경우 루프가 n번 수행되어야 하기 때문
  - 루프가 1회 수행될 때
    - Line 3: U의 원소들을 가장 많이 포함하고 있는 집합 S를 찾으려면, 현재 남아있는 S_i들 각각을 U와 비교하여야 함
    - S_i들의 수가 최대 n이라면, 각 S_i와 U의 비교는 O(n) 시간이 걸리므로, line 3은 $O(n^2)$ 시간 걸림
    - 집합 U에서 집합 S_i의 원소를 제거하므로 O(n) 시간 걸림
    - S_i를 F에서 제거하고, S_i를 C에 추가하는 것은 O(1) 시간 걸림
  - 시간 복잡도: $n \cdot O(n^2) = O(n^3)$

작업 스케줄링

- 작업 스케줄링 (Job Scheduling) 문제
  - 작업의 수행 시간이 중복되지 않도록 모든 작업을 가장 적은 수의 기계에 배정하는 문제
  - 학술대회에서 발표자들을 강의실에 배정하는 문제와 같음
    - 발표 = '작업', 강의실 = '기계'
- 작업 스케줄링 문제에 주어진 요소
  - 작업의 수
    - 입력의 크기이므로 알고리즘을 고안하기 위해 고려되어야 하는 직접적인 요소는 아님
  - 각 작업의 시작시간과 종료시간
  - 작업의 길이
    - 작업의 시작시간과 종료시간은 정해져 있으므로 작업의 길이도 주어진 것

작업 스케줄링

- 시작시간, 종료시간, 작업 길이에 대한 Greedy 알고리즘
    - 빠른 시작시간 작업 우선 (Earliest start time first) 배정
    - 빠른 종료시간 작업 우선 (Earliest finish time first) 배정
    - 짧은 작업 우선 (Shortest job first) 배정
    - 긴 작업 우선 (Longest job first) 배정
- 위 4가지 중 첫 번째 알고리즘을 제외하고 나머지 3가지는 항상 최적해를 갖지 못함

작업 스케줄링 알고리즘

- JobScheduling

입력: n개의 작업 t_1, t_2, …, t_n

출력: 각 기계에 배정된 작업 순서

1. 시작 시간으로 정렬한 작업 리스트: L
2. while L ≠ Φ
3. L에서 가장 이른 시작 시간 작업 t_i를 가져옴
4. if t_i를 수행할 기계가 있으면
5. t_i를 수행할 수 있는 기계에 배정
6. else
7. 새 기계에 t_i를 배정
8. t_i를 L에서 제거
9. return 각 기계에 배정된 작업 순서

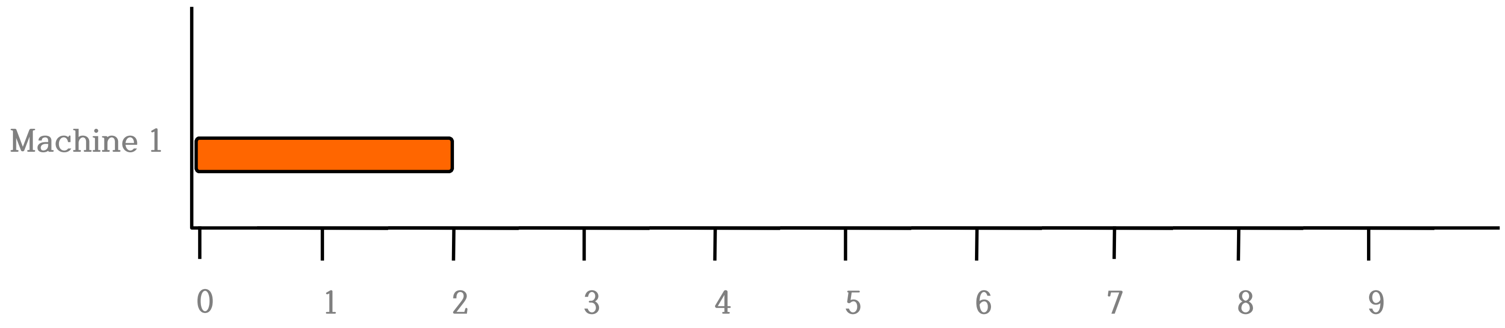
작업 스케줄링

- 수행 과정
  - t_1=[7, 8], t_2=[3, 7], t_3=[1, 5], t_4=[5, 9], t_5=[0, 2], t_6=[6, 8], t_7=[1, 6]
    - [s, f]에서 s는 시작 시간, f는 종료 시간
  - 정렬: L = {[0, 2], [1, 6], [1, 5], [3, 7], [5, 9], [6, 8], [7, 8]}

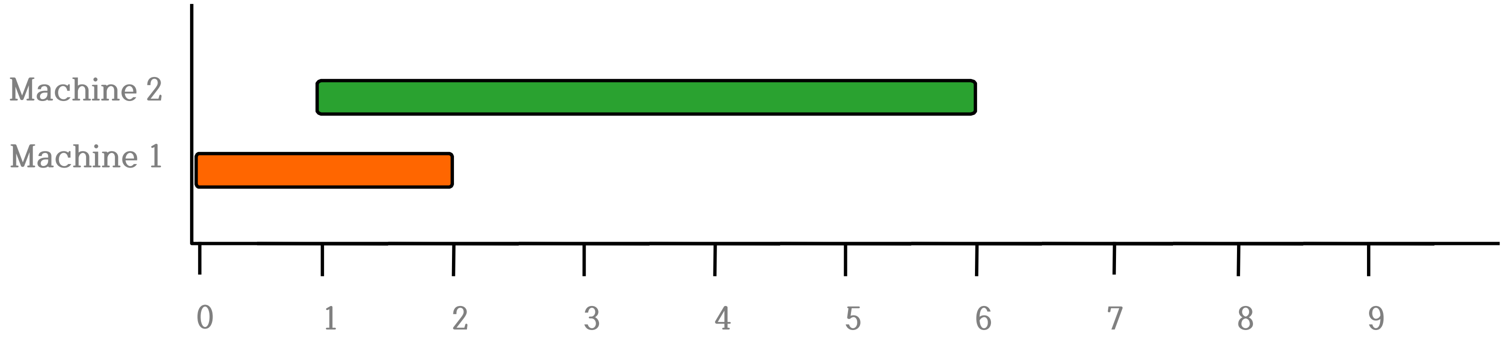
작업 스케줄링

- 수행 과정

[0,2]



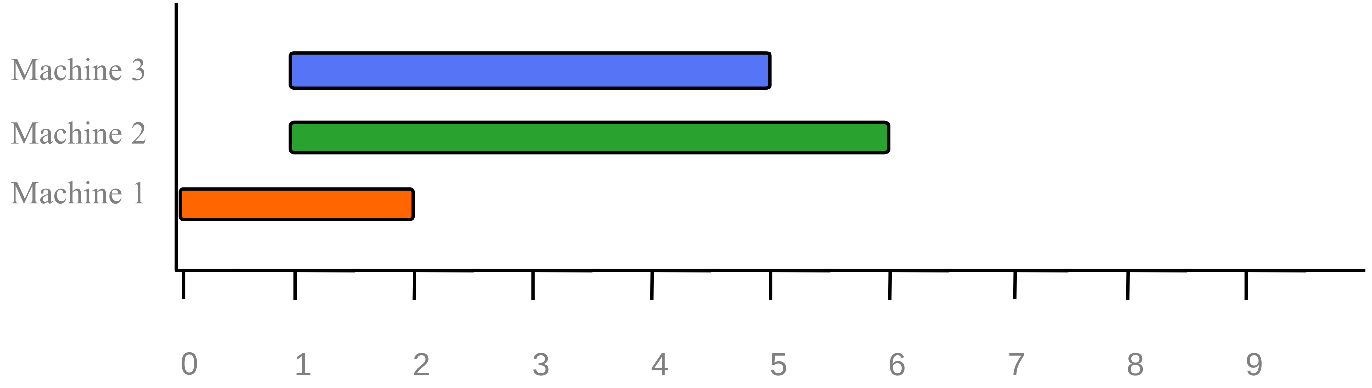
[0,2], [1,6]



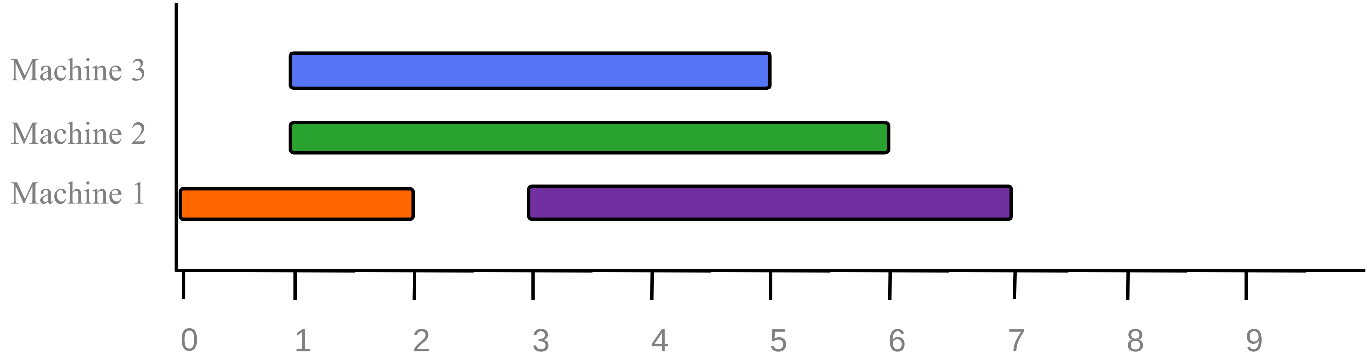
작업 스케줄링

- 수행 과정

[0,2], [1,6], [1,5]



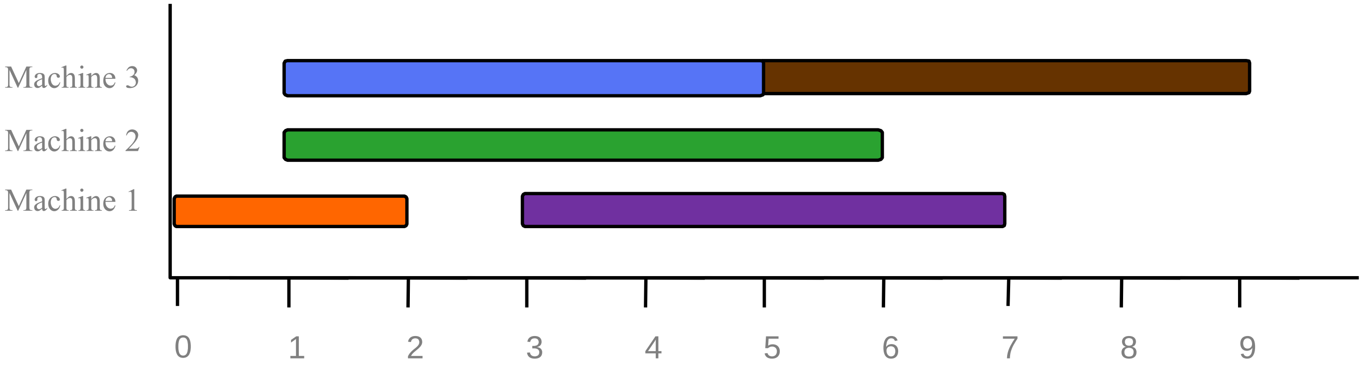
[0,2], [1,6], [1,5], [3,7]



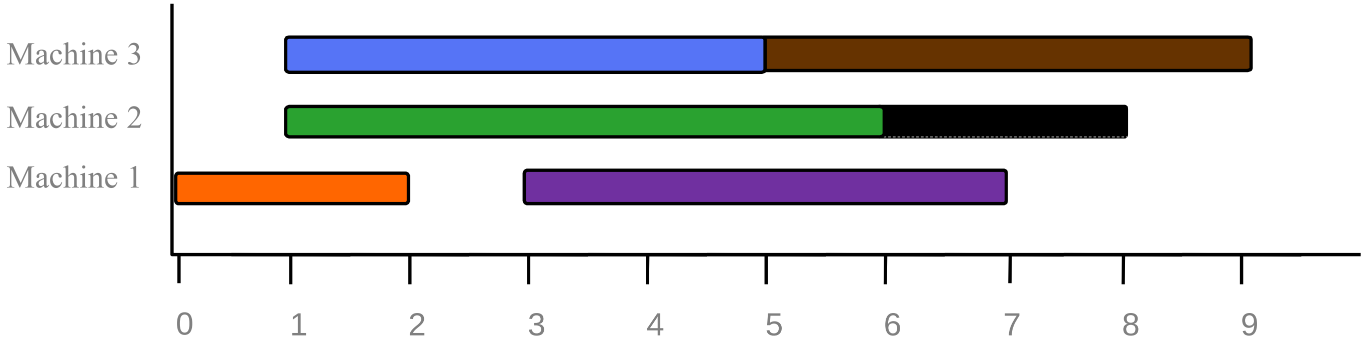
작업 스케줄링

- 수행 과정

[0,2], [1,6], [1,5], [3,7], [5,9]



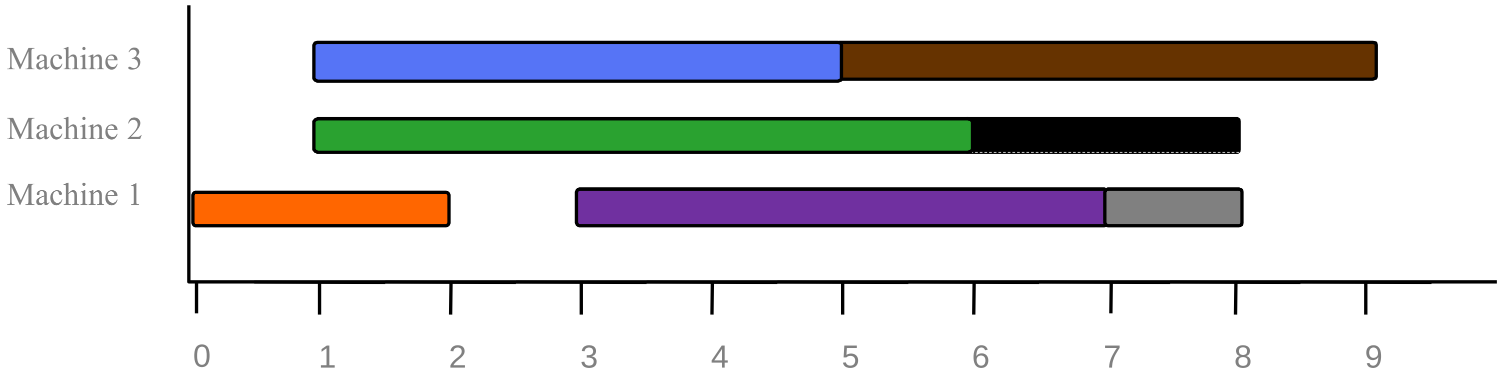
[0,2], [1,6], [1,5], [3,7], [5,9], [6,8]



작업 스케줄링

- 수행 과정

[0,2], [1,6], [1,5], [3,7], [5,9], [6,8], [7,8]



작업 스케줄링

- 시간 복잡도
  - Line 1: 정렬 시간 O(nlogn)
  - while-loop
    - 작업을 L에서 가져다 수행 가능한 기계를 찾아서 배정하므로 O(m) 시간 소요
      - m: 사용된 기계의 수
    - while-loop가 수행된 총 횟수는 n번이므로, line 2-9까지는 O(m)•n = O(mn) 시간 소요
  - 시간 복잡도: O(nlogn)+O(mn)

허프만 압축

- 파일의 각 문자가 8 bit 아스키 (ASCII) 코드로 저장되면, 그 파일의 bit 수는 8 X (파일의 문자 수)
- 파일의 각 문자는 일반적으로 고정된 크기의 코드로 표현
- 고정된 크기의 코드로 구성된 파일을 저장하거나 전송할 때 파일의 크기를 줄이고, 필요시 원래의 파일로 변환할 수 있으면, 메모리 공간을 효율적으로 사용할 수 있고 파일 전송 시간을 단축
- 파일의 크기를 줄이는 방법을 파일 압축 (file compression)이라 함

허프만 압축

- 아이디어
  - 허프만 (Huffman) 압축은 파일에 빈번히 나타나는 문자에는 짧은 이진 코드를 할당하고, 드물게 나타나는 문자에는 긴 이진 코드를 할당
  - 허프만 압축 방법으로 변환시킨 문자 코드들 사이에는 접두부 특성 (prefix property)이 존재
    - 각 문자에 할당된 이진 코드는 어떤 다른 문자에 할당된 이진 코드의 접두부 (prefix)가 되지 않음
    - [예제] 문자 'a'에 할당된 코드가 '101'이라면, 모든 다른 문자의 코드는 '101'로 시작되지 않으며 또한 '1'이나 '10'도 아님

허프만 압축

- 접두부 특성의 장점은 코드와 코드 사이를 구분할 특별한 코드가 필요 없는 것
  - 101#10#1#111#0#… 에서 ‘#’가 인접한 코드를 구분 짓고 있는데, 허프만 압축에서는 이러한 특별한 코드 없이 파일을 압축/해제 가능
- 허프만 압축은 입력 파일에 대해 각 문자의 빈도수 (문자가 파일에 나타나는 횟수)에 기반을 둔 이진 트리를 만들어서, 각 문자에 이진 코드 할당
  - 이러한 이진 코드를 ‘허프만 코드’라고 함

허프만 압축

- 알고리즘
  - HuffmanCoding

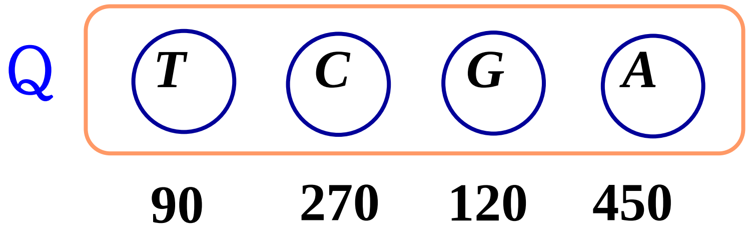
입력: 입력 파일의 n개의 문자에 대한 각각의 빈도수

출력: Huffman tree

1. 각 문자 당 노드를 만들고, 그 문자의 빈도수를 노드에 저장
2. n 노드의 빈도수에 대해 우선 순위 큐 Q를 만듬
3. while Q에 있는 노드 수 >= 2
4. 빈도수가 가장 적은 2개의 노드 (A와 B)를 Q에서 제거
5. 새 노드 N을 만들고, A와 B를 N의 자식 노드로 만듬
6. N의 빈도수 = A의 빈도수 + B의 빈도수
7. 노드 N을 Q에 삽입
8. return Q // Huffman tree의 루트를 리턴

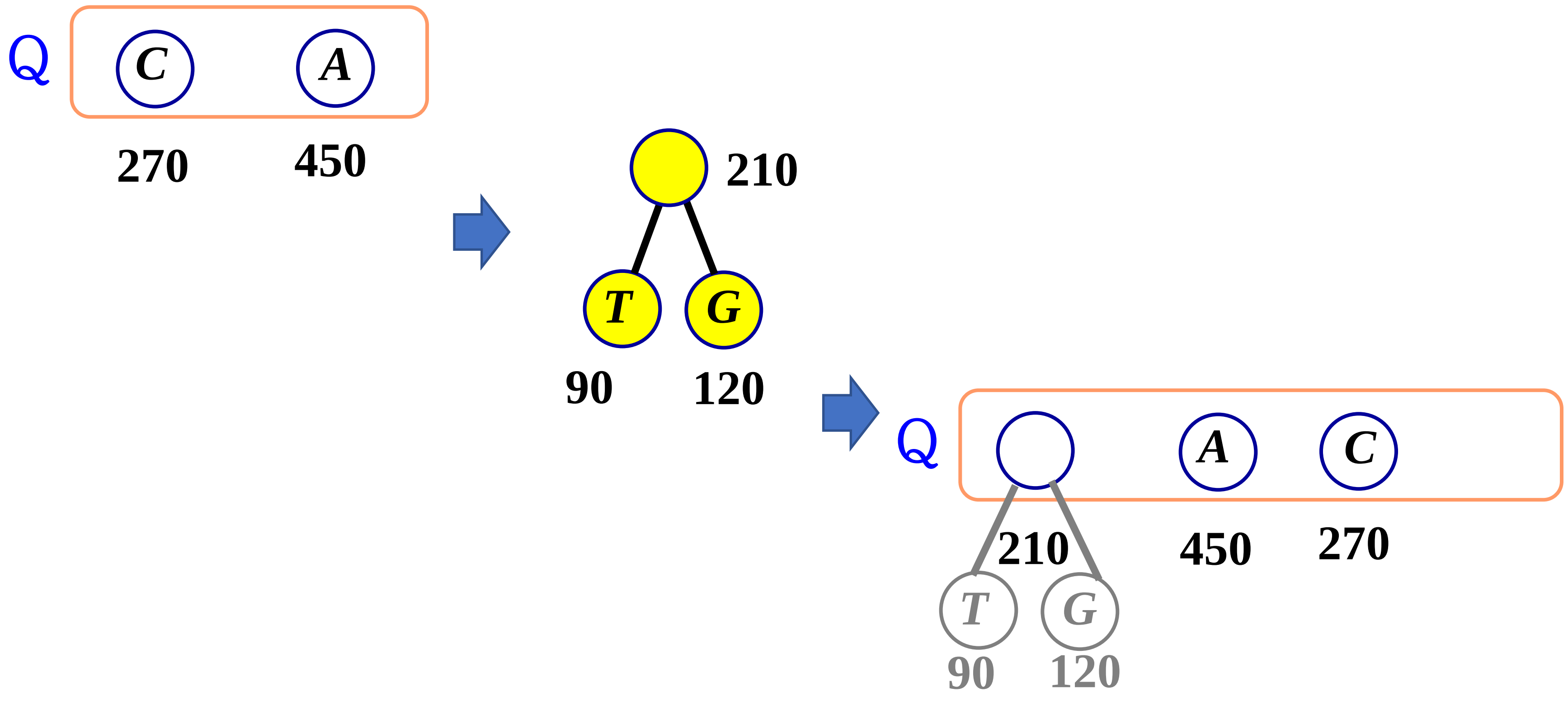
허프만 압축

- 수행과정
  - 각 문자의 빈도수에 대해
    - A: 450, T: 90, G: 120, C: 270
  - Line 2를 수행한 후의 Q
    - 우선 순위 큐 Q를 생성



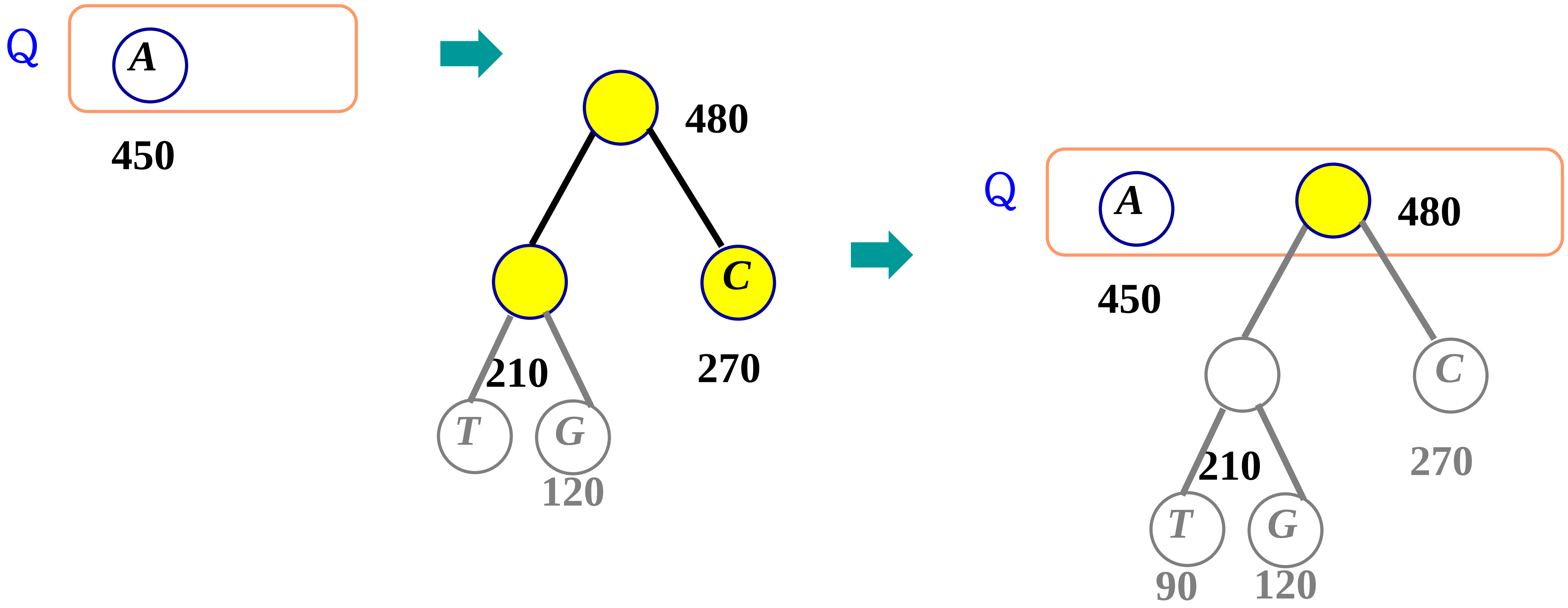
허프만 압축

- 수행과정
  - Line 3: Q에서 'T'와 'G'를 제거한 후, 새 부모 노드를 Q에 삽입



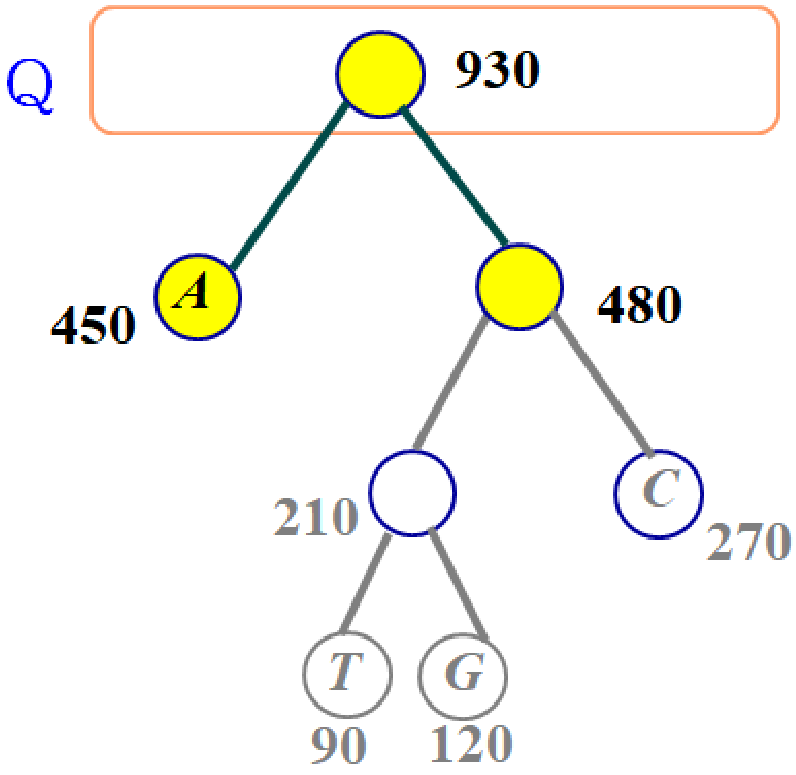
허프만 압축

- 수행과정
  - Line 3: Q에서 'T'와 'G'의 부모 노드와 'C'를 제거한 후, 새 부모 노드를 Q에 삽입



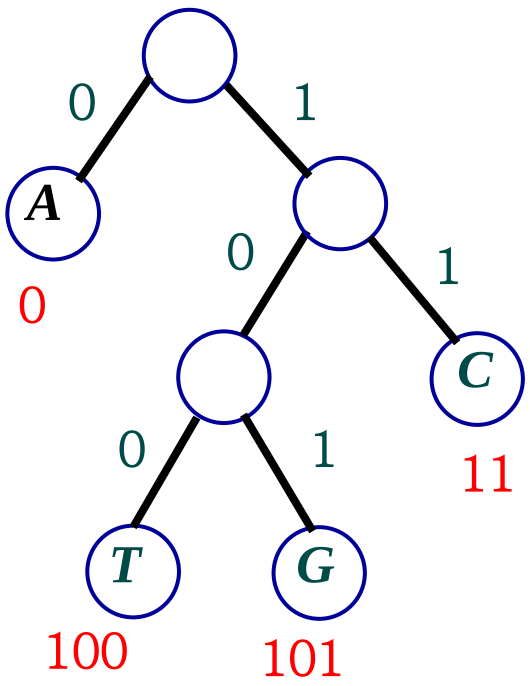
허프만 압축

- 수행과정
  - Line 3: Q에서 'C'의 부모 노드와 'A'를 제거한 후, 새 부모 노드 Q에 삽입



허프만 압축

- 수행과정
  - 반환된 트리를 살펴보면 각 잎 (단말)의 노드에만 문자가 있음
    - 루트로부터 왼쪽 자식 노드로 내려가면 '0'을, 오른쪽 자식 노드로 내려가면 '1'을 부여하면서, 각 잎에 도달할 때까지의 이진수를 추출하여 문자의 이진 코드를 얻음



허프만 압축

- 압축률
  - 예제에서 'A'는 '0', 'T'는 '100', 'G'는 '101', 'C'는 '11'의 코드가 각각 할당됨
    - 할당된 코드들을 보면, 가장 빈도수가 높은 'A'가 가장 짧은 코드를 가지고, 따라서 루트의 자식이 되어 있고, 빈도수가 낮은 문자는 루트에서 멀리 떨어지게 되어 긴 코드를 가짐
    - 이렇게 얻은 코드는 접두부 특성을 가짐
  - 압축된 파일의 bit 수
    - (450 X 1) + (90 X 3) + (120 X 3) + (270 X 2) = 1,620 bits
  - 아스키 코드로 된 파일 크기
    - (450 + 90 + 120 + 270) X 8 = 7,440 bits
  - 파일 압축률
    - (1,620/7,440) X 100 = 21.8 %이며, 원래의 약 1/5 크기로 압축

허프만 압축

- 복호화
  - 예제에서 얻은 허프만 코드로 아래의 압축된 부분에 대해서 압축을 해재하여 보자
    - 1011001000111010101010100
    - 101 / 100 / 100 / 0 / 11 / 101 / 0 / 101 / 0 / 100

→ G T T A C G A G A T

허프만 압축

- 시간 복잡도
  - Line 1: n개의 노드를 만들고, 각 빈도수를 노드에 저장하므로 O(n) 시간
  - Line 2: n개의 노드로 우선순위 큐 Q를 만듬
    - 여기서 우선 순위 큐로서 이진 힙 자료구조를 사용하면 O(n) 시간 소요

허프만 압축

- 시간 복잡도
  - Line 3~7
    - 최소 빈도수를 가진 노드 2개를 Q에서 제거하는 힙의 삭제 연산과 새 노드를 Q에 삽입하는 연산을 수행하므로 O(logn) 시간 소요
    - While-loop는 (n-1)번 반복
      - 루프가 1번 수행될 때마다 Q에서 2개의 노드를 제거하고 1개를 Q에 추가하기 때문
    - (n-1) X O(logn) = O(nlogn)
  - Line 8
    - 트리의 루트를 반환하는 것이므로 O(1) 시간
  - 시간 복잡도는 O(n)+O(n)+O(nlogn)+O(1) = O(nlogn)

요약

- 그리디 알고리즘은 입력 데이터 간의 관계를 고려하지 않고 수행과정에서 ‘욕심내어 ' 최적값을 가진 데이터를 선택하며, 선택한 값들을 모아서 문제의 최적해를 찾는다
- 그리디 알고리즘은 문제의 최적해 속에 부분 문제의 최적해가 포함되어 있고, 부분 문제의 해 속에 그 보다 작은 부분 문제의 해가 포함되어 있다. 이를 최적 부분 구조 (Optimal Substructure) 또는 최적성 원칙 (Principle of Optimality)이라고 한다
- 동전 거스름돈 문제를 해결하는 가장 간단한 방법은 남은 액수를 초과하지 않는 조건하에 가장 큰 액면의 동전을 취하는 것이다. 단, 일반적인 경우에는 최적해를 찾으나 항상 최적해를 찾지는 못한다.

요약

- Kruskul 알고리즘은 가중치가 가장 작으면서 사이클을 만들지 않는 간선을 추가시키어 트리를 만든다. 시간 복잡도는 O(mlogm) (m: 그래프의 간선 수)
- Prim 알고리즘은 최소의 가중치로 현재까지 만들어진 트리에 연결되는 간선을 트리에 추가시킴. 시간 복잡도는 $O(n^2)$
- Dijkstra 알고리즘은 출발점으로부터 최단 거리가 확정되지 않은 점들 중에서 출발점으로부터 가장 가까운 점을 추가하고, 그 점의 최단 거리를 확정함. 시간 복잡도는 $O(n^2)$

요약

- 부분 배낭 (Fractional Knapsack) 문제에서는 단위 무게 당 가장 값나가는 물건을 계속해서 배낭에 담는다. 마지막엔 배낭에 넣을 수 있을 만큼만 물건을 부분적으로 배낭에 담는다. 시간 복잡도는 $O(n\log n)$
- 집합 커버 (Set Cover) 문제는 근사 (Approximation) 알고리즘을 이용하여 근사해를 찾는 것이 보다 실질적임. U의 원소들을 가장 많이 포함하고 있는 집합을 항상 F에서 선택함. 시간 복잡도는 $O(n^3)$
- 작업 스케줄링 (Job Scheduling) 문제는 빠른 시작시간 작업을 먼저 (Earliest start time first) 배정하는 Greedy 알고리즘으로 최적해를 찾음. 시간 복잡도는 $O(n\log n)+O(mn)$ (n: 작업 수, m: 기계 수)

요약

- 허프만 압축은 파일에 빈번히 나타나는 문자에는 짧은 이진 코드를 할당하고, 드물게 나타나는 문자에는 긴 이진 코드를 할당
- n이 문자의 수일 때, 시간 복잡도는 O(nlogn)