

5. 그리디 알고리즘

한국외국어대학교
고 석 훈

목차

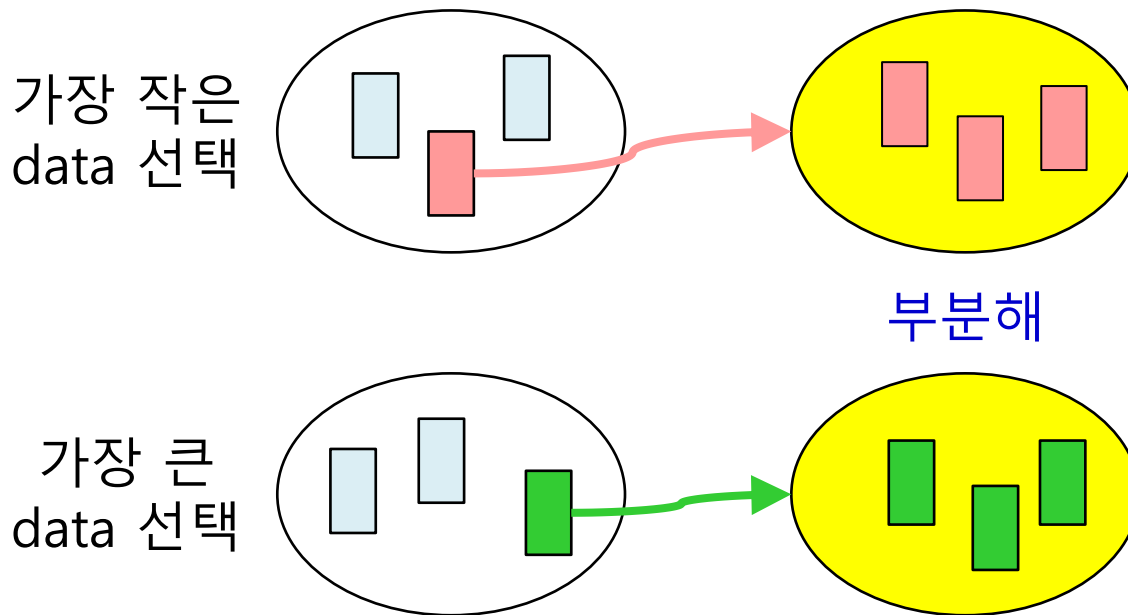
- 5.1 그리디 알고리즘
- 5.2 동전 거스름돈 문제
- 5.3 부분 배낭 문제
- 5.4 허프만 압축

5.1 그리디(Greedy) 알고리즘

- 그리디 알고리즘은 최적화 문제를 해결한다.
 - 최적화(optimization) 문제:
가능한 해들 중에서 가장 좋은 해 (최대 또는 최소)를 찾는 문제
 - 그리디 알고리즘은 입력 데이터 간의 관계를 고려하지 않고
수행 과정에서 '욕심 내어' 최소값 또는 최대값의 데이터를 선택
(이러한 선택을 '근시안적'인 선택이라 한다.)
 - 욕심쟁이 방법, 탐욕적 방법, 탐욕 알고리즘 등으로 불림
(실제로 매우 직관적으로 이해되는 방법임)

그리디 알고리즘 수행 과정

- 그리디 알고리즘은 근시안적인 선택으로 부분적인 최적해를 찾고, 이들을 모아서 문제의 최적해를 얻는다.



그리디 알고리즘의 특징

- 그리디 알고리즘은 일단 한번 부분 해를 선택하면, 이를 절대로 번복하지 않는 특징이 있다.
 - 즉, 선택한 데이터를 버리고 다른 것을 취하지 않는다.
 - 이러한 특성 때문에 대부분의 그리디 알고리즘들은 매우 단순하며, 제한적인 문제들만을 해결할 수 있다.

5.2 동전 거스름돈 문제

- 동전 거스름돈(Coin Change) 문제
 - 동전으로 거스름돈을 줄 때, 가장 적은 개수의 동전으로 거슬러 주는 방법을 찾는 문제
- 동전 거스름돈 문제의 최소 동전 수를 찾는 그리디 알고리즘
 - 동전 거스름돈 문제를 해결하는 가장 간단하고 효율적인 방법은 남은 액수를 초과하지 않는 조건하에 '욕심 내어' 가장 큰 액면의 동전을 취하는 것이다.
 - 단, 동전의 액면은 500원, 100원, 50원, 10원, 1원이라 가정한다.



동전 거스름돈 알고리즘

CoinChange(W) {

입력: 거스름돈 액수 W

출력: 거스름돈 액수에 대한 최소 동전 수

$change = W;$

$n500 = n100 = n50 = n10 = n1 = 0;$

$while (change \geq 500)$

$change = change - 500, n500 ++;$

$while (change \geq 100)$

$change = change - 100, n100 ++;$

$while (change \geq 50)$

$change = change - 50, n50 ++;$

$while (change \geq 10)$

$change = change - 10, n10 ++;$

$while (change \geq 1)$

$change = change - 1, n1 ++;$

$return (n500 + n100 + n50 + n10 + n1);$

}

// 남은 거스름돈

// 각각의 동전 개수

// 500원짜리 동전 수를 1 증가

// 100원짜리 동전 수를 1 증가

// 50원짜리 동전 수를 1 증가

// 10원짜리 동전 수를 1 증가

// 1원짜리 동전 수를 1 증가

// 동전 개수의 합계를 리턴

CoinChange 알고리즘 분석

- 초기화
 - change를 입력인 거스름돈 액수 W로 놓고,
각 동전 카운트 $n_{500} = n_{100} = n_{50} = n_{10} = n_1 = 0$ 으로 초기화
- 부분해 계산
 - 차례로 500원, 100원, 50원, 10원, 1원의 액수가 큰 순서로
while 루프를 통해 change 한도 내에서 동전을 거슬러 주고
그 때마다 각각의 동전 카운트를 1 증가시킨다.
- 최종해 합산
 - 모든 동전 카운트 들의 합을 리턴 한다.

- 그리디 알고리즘의 근시안적 특성

- CoinChange 알고리즘은 남아있는 거스름돈인 change에 대해 가장 높은 액면의 동전을 거스르며, 500원짜리 동전을 처리하는 단계에서는 100원, 50원, 10원, 1원짜리 동전을 몇 개씩 거슬러 주어야 하는지에 대해 전혀 고려하지 않는다.

CoinChange 알고리즘 실행 예

- CoinChange(760) 수행 과정
 1. $\text{change}=760$, $n500=n100=n50=n10=n1=0$ 으로 초기화
 2. $\text{change}=760$, while ($\text{change} \geq 500$) 조건에 의해 $\text{change}=760-500=260$, $n500=1$ 이 된다.
 3. $\text{change}=260$, while ($\text{change} \geq 500$) 조건 실패
 4. $\text{change}=260$, while ($\text{change} \geq 100$) 조건에 의해 $\text{change}=260-100=160$, $n100=1$ 이 된다.
 5. $\text{change}=160$, while ($\text{change} \geq 100$) 조건에 의해 $\text{change}=160-100=60$, $n100=2$ 가 된다.
 6. $\text{change}=60$, while ($\text{change} \geq 100$) 조건 실패

7. $\text{change}=60$, while ($\text{change} \geq 50$) 조건에 의해 $\text{change}=60-50=10$, $n_{50}=1$ 이 된다.
8. $\text{change}=10$, while ($\text{change} \geq 50$) 조건 실패
9. $\text{change}=10$, while ($\text{change} \geq 10$) 조건에 의해 $\text{change}=10-10=0$, $n_{10}=1$ 이 된다.
10. $\text{change}=0$, while ($\text{change} \geq 10$) 조건 실패
11. $\text{change}=0$, while ($\text{change} \geq 1$) 조건 실패
12. $n_{500}+n_{100}+n_{50}+n_{10}+n_1=1+2+1+1+0=5$ 리턴



CoinChange 알고리즘은 항상 옳은가?

- 만일 한국은행에서 160원짜리 동전을 추가로 발행한다면, CoinChange 알고리즘이 최소 동전 수를 계산할 수 있을까?
 - 거스름돈이 200원이라면, CoinChange 알고리즘은 160원 동전 1개와 10원 동전 4개로서 총 5개를 리턴 한다.



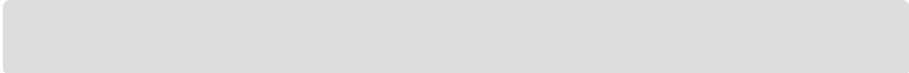
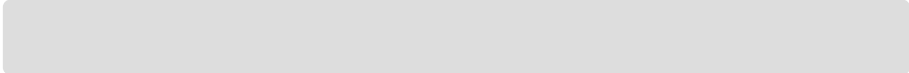
CoinChange의 해



정답

- CoinChange 알고리즘은 항상 최적의 답을 주지 못한다.
 - 다음 장에서 어떤 경우에도 최적해를 찾는 동전 거스름돈을 위한 동적 계획 알고리즘을 소개한다.

5.3 배낭(Knapsack) 문제

- 배낭(Knapsack) 문제
 - 한정된 무게의 물건을 담을 수 있는 배낭과 고유한 무게와 가치를 가지는 n 개의 물건이 있을 때, 최대의 가치를 갖도록 배낭에 넣을 물건을 정하는 문제
- 두 가지 배낭 문제
 - 
물건을 쪼갤 수 있어서, 배낭에 물건을 부분적으로 넣는 것을 허용
 - 
물건을 나눌 수 없어서, 배낭에 하나의 물건을 통째로 넣어야 함

- 부분 배낭 문제의 해법

- 물건을 부분적으로 배낭에 담을 수 있으므로,
'욕심을 내어' 단위 무게 당 가장 값나가는 물건을 배낭에 넣고,
계속해서 차례대로 그 다음으로 값나가는 물건을 넣는다.
- 만일 값나가는 물건을 '통째로' 배낭에 넣을 수 없게 되면,
배낭에 넣을 수 있을 만큼만 부분적으로 배낭에 담는다.

예제: 부분 배낭 문제

- 50kg 용량의 배낭과 다음의 5가지 음식재료가 있을 때, 최대 가치가 되도록 배낭에 넣는 방법은?



부분 배낭 문제

- 최대 용량 $40g$ 의 배낭과 다음의 4가지 금속 분말이 있을 때, 최대 가치가 되도록 배낭에 넣는 방법은?
 - 단, 금속 분말은 일부만 배낭에 넣을 수 있다.



부분 배낭 알고리즘

FractionalKnapsack(S, C) {

입력: n 개의 물건 리스트 S , 각 물건의 무게와 가치, 배낭의 용량 C

출력: 배낭에 담은 물건 리스트 L 과 배낭에 담은 물건의 가치 합 v

S 의 각 물건의 단위 무게당 가치를 계산한다

S 를 단위 무게당 가치를 기준으로 내림차순 정렬;

$L = \emptyset$; // 배낭에 담을 물건 리스트

$w = 0$; // 배낭에 담긴 물건들의 무게의 합

$v = 0$; // 배낭에 담긴 물건들의 가치의 합

S 에서 단위 무게당 가치가 가장 큰 물건 x 를 가져온다.

while ($(w + x$ 의 무게) $\leq C$) {

x 를 L 에 추가, $w = w + x$ 의 무게, $v = v + x$ 의 가치, x 를 S 에서 제거;

S 에서 단위 무게당 가치가 가장 큰 물건 x 를 가져온다.

}

if ($(C - w) > 0$) { // 배낭에 물건을 부분적으로 담을 여유가 있으면

물건 x 를 $(C - w)$ 만큼만 L 에 추가;

$v = v + (C - w)$ 만큼의 x 의 가치;

}

return L, v ;

}

알고리즘 실행 과정

- 최대 용량 $C = 40g$ 의 배낭과 다음의 4가지 금속 분말이 있을 때, 부분 배낭 알고리즘 수행 과정

- 각 물건의 단위 무게당 가치를 계산

물건	단위 그램당 가치
주석	1,000원
백금	60,000원
은	4,000원
금	50,000원



- 단위 무게당 가치로 정렬

$$S = [\text{백금}, \text{금}, \text{은}, \text{주석}]$$

- $L = \emptyset, w = 0, v = 0$ 로 각각 초기화

- $S = [\text{백금}, \text{금}, \text{은}, \text{주석}]$ 에서 백금을 가져온다.
- $\text{while } ((w + \text{백금의 무게}) \leq C) = ((0 + 10) < 40)$ 으로 '참'이다.
- 백금을 배낭 L 에 추가시킨다. 즉, $L = [\text{백금 } 10g]$ 이 된다.
- $w = w + (\text{백금의 무게}) = 0 + 10 = 10g$
- $v = v + (\text{백금의 가치}) = 0 + 60 = 60$ 만원
- S 에서 백금을 제거한다. $S = [\text{금}, \text{은}, \text{주석}]$
- S 에서 금을 가져온다.
- $\text{while } ((w + \text{금의 무게}) \leq C) = ((10 + 15) < 40)$ 으로 '참'이다.
- 금을 배낭 L 에 추가시킨다. 즉, $L = [\text{백금 } 10g, \text{금 } 15g]$ 이 된다.
- $w = w + (\text{금의 무게}) = 10 + 15 = 25g$
- $v = v + (\text{금의 가치}) = 60 + 75 = 135$ 만원

- S 에서 금을 제거한다. $S = [\text{은}, \text{주석}]$
- S 에서 은을 가져온다.
- $\text{while } ((w + \text{은의 무게}) \leq C) = ((25 + 25) > 40)$ 으로 '거짓'이다.
- $\text{if } ((C - w) > 0) = (40 - 25) > 0$ 으로 '참'이다.
- 은을 $(C - w) = 40 - 25 = 15g$ 만큼 L 에 추가한다.
즉, $L = [\text{백금 } 10g, \text{금 } 15g, \text{은 } 15g]$ 이 된다.
- $v = v + (C - w)$ 만큼의 은의 가치 $= 135 + 15 * 4\text{천} = 141\text{만원}$
- 배낭 $L = [\text{백금 } 10g, \text{금 } 15g, \text{은 } 15g]$ 과 가치의 합 $v = 141\text{만원}$ 리턴



시간복잡도 분석

- 단계 별 시간복잡도

- n 개의 물건 각각의 단위 무게 당 가치 계산 시간: $O(n)$
- 물건의 단위 무게 당 가치를 기준으로 정렬 시간: $O(n \log_2 n)$
- while 루프의 수행은 n 번을 넘지 않으며,
루프 내부의 수행은 $O(1)$ 시간이 걸린다.
- 마지막 물건을 담는 if 문 실행 시간: $O(1)$
- 따라서 전체 알고리즘의 시간복잡도는

응용 사례

- 버리는 부분을 최소화하는
원자재 자르기(Raw Material Cutting)
- 금융 포트폴리오(Financial Portfolio)에서의 최선의 선택

5.4 허프만(Huffman) 압축

- 파일 압축(file compression)
 - 파일의 각 문자는 일반적으로 고정된 크기의 코드로 표현된다.
 - 파일의 각 문자가 8 bit 아스키 (ASCII) 코드로 저장되면, 그 파일의 bit 수는 $8 \times$ (파일의 문자 수)이다.
 - 고정된 크기의 코드로 구성된 파일 들을 저장하거나 전송할 때 크기를 줄이고 필요 시 원래의 파일로 변환할 수 있으면 저장 공간을 효율적으로 사용할 수 있고 전송 시간을 단축시킬 수 있다.

- 허프만(Huffman) 압축

- 기본 아이디어:

- 파일에 빈번히 나타나는 문자에는 짧은 이진 코드를 할당하고,
드물게 나타나는 문자에는 긴 이진 코드를 할당하는 압축 방법

- 좀 더 구체적으로, 입력 파일에 대해

- 각 문자의 출현 빈도수(문자가 파일에 나타나는 횟수)에 기반으로
이진트리(binary tree)를 만들어서 각 문자에 이진 코드를 할당

- 허프만 압축 방법으로 변환시킨 문자 코드들 사이에는
접두부 특성(prefix property)이 존재

- 접두부 특성(prefix property)

- 각 문자에 할당된 이진 코드는 어떤 다른 문자에 할당된 이진 코드의 접두부(prefix)가 되지 않는다는 것을 의미
- 예를 들어, 문자 'a'에 할당된 코드가 '101'이라면, 모든 다른 문자의 코드는 '1', '10', '101'로 시작되지 않는다.
- 이러한 특성으로 코드 사이를 구분하는 인접코드가 필요 없어진다. 예를 들어, 101#10#1#111#...에서 '#'이 인접 코드를 구분 짓는데, 허프만 압축에서는 이러한 코드 없이 파일을 압축/해제할 수 있다.

허프만 코드 알고리즘

HuffmanCoding {

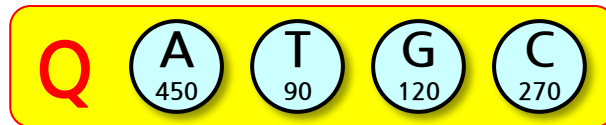
입력: 입력 파일의 n 개의 문자에 대한 각각의 빈도수

출력: 허프만 트리

1. 각 문자의 노드를 만들고, 그 문자의 빈도수를 노드에 저장
2. n 개의 노드의 빈도수에 대한 우선순위 큐 Q 를 만든다.
3. *while* (Q 에 있는 노드 수 ≥ 2) {
4. 빈도수가 가장 작은 2개의 노드 A 와 B 를 Q 에서 제거
5. 새 노드 N 을 만들고, A 와 B 를 N 의 자식 노드로 만든다.
6. N 의 빈도수 = A 의 빈도수 + B 의 빈도수
7. 노드 N 을 Q 에 삽입한다.
- }
8. *return* Q // 허프만 트리의 루트를 리턴
- }

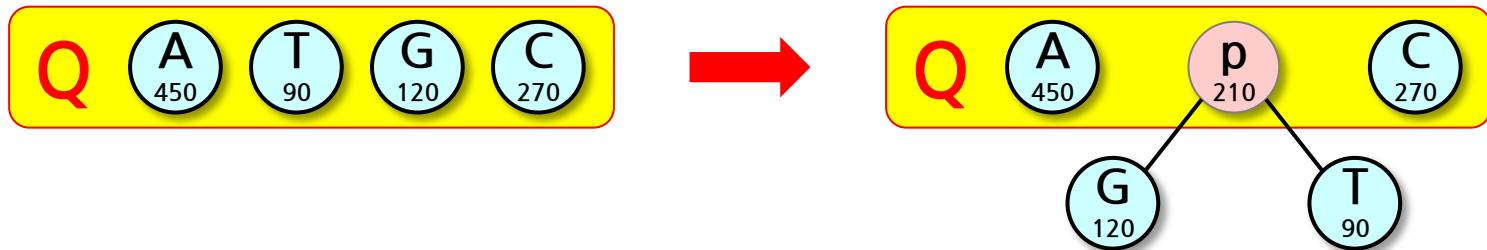
허프만 코드 알고리즘 실행 과정

- 입력 파일은 4개의 문자 A, T, G, C 로 되어 있고, 각 문자의 빈도수는 $A: 450, T: 90, G: 120, C: 270$ 이다.
 - Line 1. 각 문자의 노드를 생성하고 빈도수 저장
 - Line 2. 노드 들을 우선순위 큐 Q 에 저장

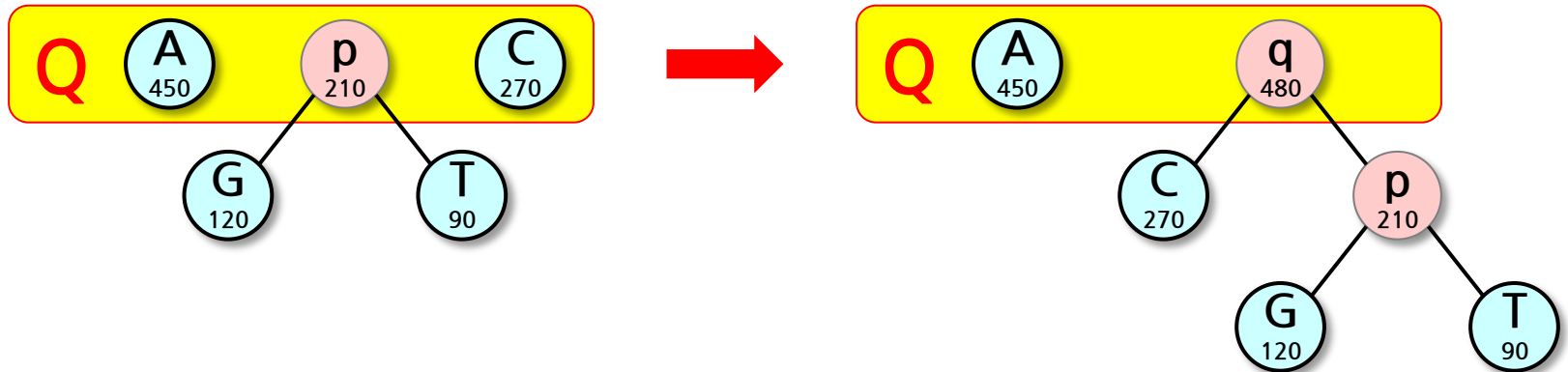


허프만 코드 알고리즘 실행 과정

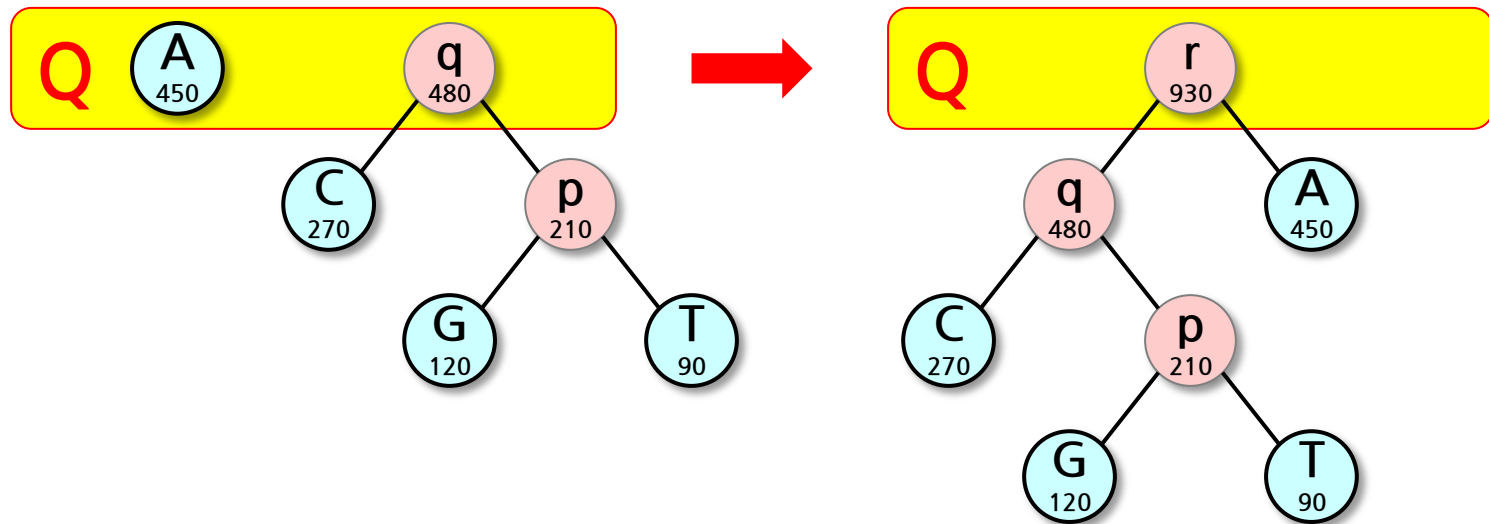
- Line 3. *while* (Q 의 노드 수 ≥ 2) = $(4 \geq 2)$ 이므로 '참'
- Line 4. 빈도 수가 가장 작은 노드 T 와 G 를 Q 에서 제거
- Line 5. 새 노드 p 를 만들고, 노드 T 와 G 를 자식 노드로 연결
 - ◆ 이때, 노드 T 와 G 는 노드 p 의 어느 쪽 자식 노드로 연결되어도 압축률은 동일하지만, 일반적인 트리 알고리즘이 left를 먼저 방문하므로 빈도가 높은 노드를 left child로 연결하자.
- Line 6. p 의 빈도수 = G 의 빈도수 + T 의 빈도수 = $120 + 90 = 210$
- Line 7. 노드 p 를 Q 에 삽입



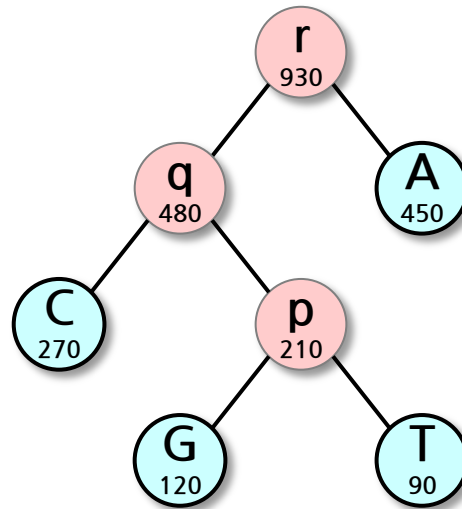
- Line 3. *while* (Q 의 노드 수 ≥ 2) = $(3 \geq 2)$ 이므로 '참'
- Line 4. 빈도 수가 가장 작은 노드 p 와 c 를 Q 에서 제거
- Line 5. 새 노드 q 를 만들고, 노드 p 와 c 를 자식 노드로 연결
 - ◆ 이때, 빈도가 높은 c 를 left child로 연결한다.
- Line 6. q 의 빈도수 = c 의 빈도수 + p 의 빈도수 = $270 + 210 = 480$
- Line 7. 노드 q 를 Q 에 삽입



- Line 3. *while* (Q 의 노드 수 ≥ 2) = ($2 \geq 2$)이므로 '참'
- Line 4. 빈도 수가 가장 작은 노드 A 와 q 를 Q 에서 제거
- Line 5. 새 노드 r 을 만들고, 노드 A 와 q 를 자식 노드로 연결
- Line 6. r 의 빈도수 = q 의 빈도수 + A 의 빈도수 = $480 + 450 = 930$
- Line 7. 노드 r 을 Q 에 삽입

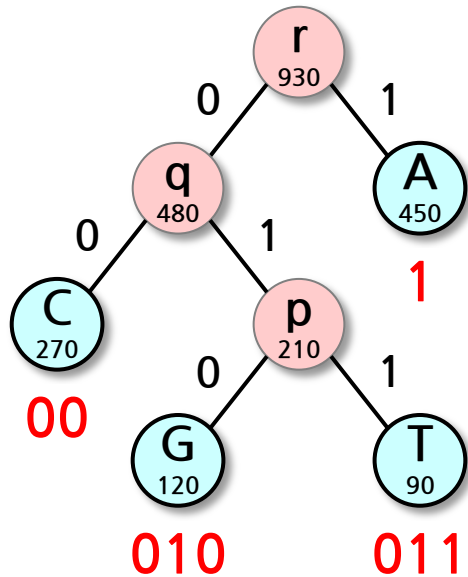


- Line 3. *while* (Q 의 노드 수 ≥ 2) \neq ($1 < 2$)이므로 '거짓'
- Line 8. 큐 Q 에 있는 루트 노드 r 을 리턴



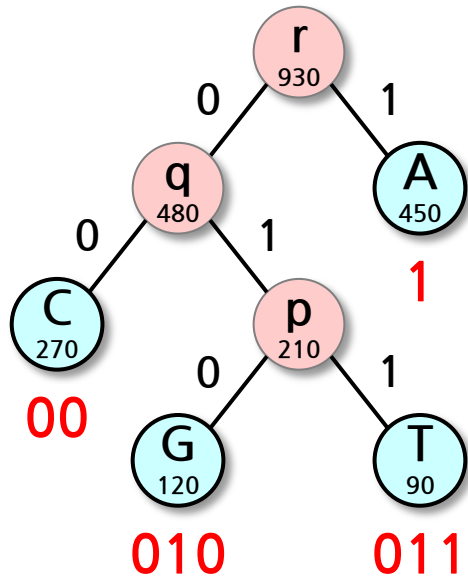
압축코드 생성

- 이진 압축 코드 생성
 - 리턴된 트리를 살펴보면 각 단말 노드에만 문자가 있다.
 - 루트로부터 왼쪽 자식 노드로 내려가면 '0'을, 오른쪽 자식 노드로 내려가면 '1'을 부여하면서, 각 단말 노드에 도달할 때까지의 이진 수를 추출하여 문자의 이진 코드를 구한다.



문자	이진 코드	빈도
A	1	450
C	00	270
G	010	120
T	011	90

- 할당된 이진 코드를 보면,
빈도수가 가장 높은 'A'가 가장 짧은 코드를 가지고,
빈도수가 낮은 문자는 긴 코드를 가진다.
- 이진 트리를 통해 얻은 이진 코드가
접두부 특성을 가지고 있음을 쉽게 확인할 수 있다.



문자	이진 코드	빈도
A	1	450
C	00	270
G	010	120
T	011	90

압축

- 압축: 문자를 이진 코드로 변환
 - 변환 테이블을 참조하여 아스키 문자를 이진 코드로 변환
 - 이진 코드를 모두 연결한 비트 시퀀스 생성
 - 비트 시퀀스에서 8비트씩 끊어서 아스키 문자로 파일 저장

G T T A C G A G A T
010 **011** 011 **1** 00 **010** 1 **010** 1 **011**
01001101110001010101011 ...
01001101 **11000101** 0101011...

문자	이진 코드	빈도
A	1	450
C	00	270
G	010	120
T	011	90

압축 해제

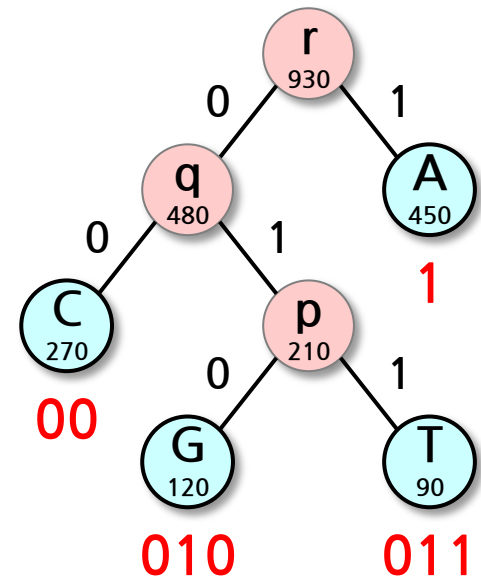
- 압축 해제: 비트 시퀀스에서 이진 코드 추출
 - 파일의 문자를 읽어 비트 시퀀스 생성
 - 비트 순서대로 트리를 따라가며 이진 코드 및 아스키 문자 추출

01001101 11000101 0101011...

0100110111000101010101011...

010 011 011 1 00 010 1 010 1 011

G T T A C G A G A T



- 파일 크기 비교

- 아스키 코드 파일 크기

- $$= (450 + 90 + 120 + 270) \times 8 = 7,440 \text{ bit}$$

- 압축 파일 크기

- $$= (450 \times 1) + (90 \times 3) + (120 \times 3) + (270 \times 2) = 1,620 \text{ bit}$$

- 파일 압축률

- $$= 1,620/7,440 = 21.8\%, \text{ 약 } 1/5 \text{ 크기로 압축}$$

시간복잡도

- 단계 별 시간복잡도

- n 개의 노드를 만들고, 각 빈도수를 노드에 저장하는데 $O(n)$
- 힙을 이용하여 n 개의 노드로 우선순위 큐 Q 를 만드는데 $O(n)$
- 노드 2개를 Q 에서 제거하는 힙의 삭제 연산과 새 노드를 Q 에 삽입하는 연산에 $O(\log n)$, 그리고 while 루프가 $(n - 1)$ 번 반복된다.
따라서 트리 생성에 $(n - 1) \times O(\log n) = O(n \log n)$
- 트리의 루트를 리턴하는 것은 $O(1)$ 시간이 걸린다.
- 시간복잡도:

Q&A

