CS300: Homework #3

Due on October 18, 2016 at 10:30am

Prof. Sunghee Choi

20160051 Ohjun Kwon

Problem 1

(a)

2	3	4	5
8	9	12	∞
14	16	∞	∞
∞	∞	∞	∞

(b)

Part A

$m \times n$인 영 타블로는 아래와 같은 성질들을 만족한다.
$i < x \leq m$, $j < y \leq n$인 임의의 i, j, x, y에 대하여,
(1) $Y[i, j] \leq Y[x, j]$
(2) $Y[i, j] \leq Y[i, y]$
이가 의미하는 바는 각 열과 행이 왼쪽에서 오른쪽, 위에서 아래로 오름차순으로 정렬되어 있다는 것이다.

$Y[1, 1] = \infty$라고 하면 영 타블로의 임의 칸의 숫자 $Y[i, j]$에 대하여 영 타블로의 성질을 이용해 아래와 같은 식을 세울 수 있다.

$$\infty = Y[1, 1] \leq Y[1, j] \leq Y[i, j]$$

임의의 i, j에 대하여 $Y[i, j] = \infty$이므로, 영 타블로 Y는 비어있다.

Part B

위와 비슷한 방법으로, 영 타블로의 임의 칸의 숫자 $Y[i, j]$에 대하여 영 타블로의 성질을 이용해 아래와 같은 식을 세울 수 있다.

$$\infty > Y[m, n] \geq Y[m, j] \geq Y[i, j]$$

임의의 i, j에 대하여 $Y[i, j] < \infty$이므로, 영 타블로 Y는 꽉차있다.

(c) $m \times n$ 영 타블로에서 가장 작은 숫자를 추출하는 알고리즘은 아래 **Extract-Min**에서 보이는 것처럼 영 타블로의 1행 1열의 원소가 영 타블로에서 가장 작으므로 이를 리턴하고 나머지 영 타블로를 다시 조건에 맞게 적용하는 과정인 **Corr-Young**으로 가능하다.

```
1: function EXTRACT-MIN(Y)
2:     min ← Y[1, 1]
3:     Y[1, 1] ← ∞
4:     CORR-YOUNG(Y, i, j)
5:     return min
6: end function
```

Corr-Young은 $Y[1, 1]$가 빠져 영 타블로의 성질을 잃은 $m \times n$ 행렬을 다시 영 타블로의 성질을 만족하도록 변경해주는 함수이다. 이는 $Y[1, 1]$에서 차례로 오른쪽에 들어있는 값과 아래에 들어있는 값을 비교해 작은 값으로 변경하도록 한다. 초기의 영 타블로의 성질을 만족하던 행렬에서 $Y[1, 1]$만 변경되었으므로 그를 제외한 $Y[2...m, 2...n]$은 영 타블로의 성질을 그대로 만족한다는 사실을 가지고 알고리즘은 진행한다. 오른쪽과 아래에 들어있는 값 중에서 작은 값으로 변경한 후에는 다시 변경된 곳에서 재귀적으로 실행함으로 결과적으로는 영 타블로의 성질을 만족하도록 변경된다.
이때, **Extract-Min**의 시간복잡도를 계산해보면, **Corr-Young**를 제외한 Statement들은 상수 시간에 연산이 가능하여 **Corr-Young**의 시간복잡도가 가장 큰 영향을 끼치게 되는데 이 **Corr-Young**는 $m \times n$ 행렬의 오른쪽 방향과 아랫쪽 방향만을 움직이게 되므로 최대 $m \times n$ 번 실행된다고 볼 수 있다. 오른쪽 방향으로 진행하게 될 때는 $i \times (j - 1)$의 subproblem을 풀게 되는 것이고, 아랫쪽 방향으로 진행하게 될 때는 $(i - 1) \times j$의 subproblem을 풀게 되는 것이다. $i + j$의 값을 생각해보면 한 번의 실행마다 오른쪽 혹은 아랫쪽으로 진행하므로 1씩 증가하게

```
1: function CORR-YOUNG(Y, i, j)
2:     right ← ∞
3:     down ← ∞
4:     if j + 1 < n then
5:         right ← Y[i, j + 1]
6:     end if
7:     if i + 1 < m then
8:         down ← Y[i + 1, j]
9:     end if
10:    if right < Y[i, j] OR down < Y[i, j] then
11:        if down < right then
12:            Y[i, j] ↔ Y[i + 1, j]
13:            CORR-YOUNG(Y, i + 1, j)
14:        else
15:            Y[i, j] ↔ Y[i, j + 1]
16:            CORR-YOUNG(Y, i, j + 1)
17:        end if
18:    end if
19: end function
```

된다. 그리고 이는 $m+n$이 될 때 멈춘다. 이를 문제에서 주어진 $T(p)$에 대해서 나타내면 식 (1)와 같이 나타낼 수 있다.

$$T(p) = T(p-1) + \Theta(1) \tag{1}$$

식 (1)를 대입법을 이용하여 풀게 되면,

$$\begin{aligned} T(p) &= \Theta(1) + T(p-1) \\ &= \Theta(1) + \Theta(1) + T(p-2) \\ &\vdots \\ &= \Theta(p) \\ &= O(p) \\ &= O(m+n) \end{aligned}$$

(d) 가득 차지 않은 영 타블로는 가장 마지막 원소인 $Y[m, n]$의 값이 ∞이다. 그렇다면 이 값에 새로 입력하고자 하는 값을 집어넣고 **(c)**의 **Corr-Young**에서 했던 것과 비슷한 방법으로 index를 줄여가면서 끼워 넣을 수 있다.

```
1: function INSERT(Y, a)
2:     Y[m, n] ← a
3:     CORR-YOUNG'(Y, n, m)
4: end function
```

Corr-Young'은 위의 **Corr-Young**의 정확히 반대 방향으로 동일한 방식으로 동작하게 된다.
그러므로 **Insert**의 시간 복잡도도 위의 **Extract-Min**과 동일한 $O(m+n)$이다.

(e) 위에서 작성한 **Extract-Min**과 **Insert**을 이용하여 정렬 알고리즘을 구상할 수 있다. n^2개의 수를 정렬하기 위해서 먼저 $n \times n$의 행렬 A를 만든다. 이 행렬에 **Insert**를 이용하여 원소를 집어넣는다. 이때의 시간복잡도는 각각의 수를 넣는데 $O(n+n) = O(n)$의 시간이 걸리고, 총 n^2개의 수가 있으므로 총 $n^2 O(n) = O(n^3)$의

```
1: function CORR-YOUNG'(Y, i, j)
2:     up ← −∞
3:     left ← −∞
4:     if i − 1 ≥ 0 then
5:         up ← Y[i − 1, j]
6:     end if
7:     if j − 1 ≥ 0 then
8:         left = Y[i, j − 1]
9:     end if
10:    if Y[i, j] < up OR Y[i, j] < left then
11:        if left < up then
12:            Y[i, j] ↔ Y[i − 1, j]
13:            CORR-YOUNG'(Y, i − 1, j)
14:        else
15:            Y[i, j] ↔ Y[i, j − 1]
16:            CORR-YOUNG'(Y, i, j − 1)
17:        end if
18:    end if
19: end function
```

시간을 소요해 주어진 n^2개의 원소를 영 타블로 꼴로 나타낼 수 있다. 또, 이렇게 만들어진 영 타블로 A에서 **Extract-Min**을 이용해 숫자를 하나씩 영 타블로에서 뽑게되면 작은 원소부터 차례로 뽑혀 나오게 된다. 이때의 시간복잡도는 위와 마찬가지로 각 뽑은 원소마다 $O(n)$, 총 원소의 개수 n^2해서 $O(n^3)$의 시간이 걸린다. 이를 종합하면 결과적으로 $O(n^3)$의 시간이 걸린다.

(f) **Corr-Young**을 작성했을 때와 비슷한 방법으로 작성하면 된다. 다만 이때에는 왼쪽 상단에서 시작하게 되면 오른쪽과 아랫쪽 방향에 모두 자기보다 큰 원소가 위치하게 되므로 두 방향 모두 확인해보야야 하므로 $O(m+n)$만에 탐색을 끝낼 수 없게 된다. 여기서 약간의 아이디어가 추가되는데 반대로 오른쪽 상단에서 시작하게 되면 왼쪽으로 가면 현재보다 작은 값이, 아랫쪽으로 가면 현재보다 큰 값이 위치하게 되어 곧바로 하나의 길을 선택할 수 있게 된다. 이를 의사코드를 이용하여 영 타블로 Y 내부에 k를 포함하는지 체크하기 위한 함수를 **IsIn**와 같이 나타낼 수 있다. 처음에는 오른쪽 상단에서 시작해야하므로 **IsIn**$(Y, 0, n, k)$를 호출해주면 된다.

```
1: function ISIN(Y, i, j, k)
2:     if Y[i, j] < k then
3:         if i + 1 < m then
4:             return ISIN(Y, i + 1, j, k)
5:         else
6:             return False
7:         end if
8:     else if Y[i, j] > k then
9:         if j − 1 ≥ 0 then
10:            return ISIN(Y, i, j − 1, k)
11:        else
12:            return False
13:        end if
14:    else
15:        return True
16:    end if
17: end function
```

이 **IsIn** 알고리즘의 시간복잡도는 위에서 했던 것과 마찬가지로 최대 $m+n$번 움직이게 되므로 $O(m+n)$이다.

Problem 2

검색을 상수시간에 한다는 뜻은 값을 바로 읽어 일정한 과정을 거쳐 결과를 도출 할 수 있어야 한다는 이야기이다. 이를 위해서는 전처리 과정을 거친 후 상수 번의 접근을 통해 유의미한 값을 얻어낼 수 있어야 한다는 소리이다. 카운팅 소트를 이용하면 이들을 만족할 수 있다.

```
1: function HowMany(D, a, b)
2:     ▷ preprocessing
3:     for i ← 0 to k do
4:         C[i] ← 0 ▷ initialization of the array
5:     end for
6:     for i ← 1 to n do
7:         C[D[i]]+ = 1 ▷ counts
8:     end for
9:     for i ← 1 to k do
10:        C[i]+ = C[i − 1] ▷ accumulates
11:    end for
12:    ▷ quering
13:    return C[b] − C[a − 1]
14: end function
```

전처리시에는 $k+1$, n, k번 반복문을 돌기 때문에 총 $O(n+2k+1) = O(n+k)$의 시간복잡도를 가지고, 쿼리시에는 상수시간복잡도를 갖는다.

Problem 3

0부터 n^3-1까지의 수는 n진법으로 3자리 수로 나타낼 수 있는 값이다. $an^2+bn^1+cn^0$은 $\overline{abc}_{(n)}$로 나타나진다 ($a,b,c \in \mathbb{Z}$ s.t. $0 \le a,b,c \le n-1$). 여기서 1의 자리, n의 자리, n^2의 자리 별로 기수 정렬을 이용하여 정렬하면 $O(n)$만에 정렬이 가능하다.

Problem 4

버킷 정렬은 입력이 균등 분포를 하고 있음을 가정하고 실행되는 정렬이다. 입력이 균등 분포를 띤다는 것을 가정하기 때문에 실행시간의 기댓값이 $O(n)$이 될 수 있는 것이다. 그렇다면 우리는 문제에서 주어진 원에 점이 들어갈 확률이 같은 영역으로 원을 나누어 각각을 버킷으로 지정해야 각각의 버킷에 각각의 점들이 최대한 퍼져서 들어갈 것이고, 이렇게 분포 되어야 선형 시간에 정렬이 가능할 것이라고 기대할 수 있다. 그렇다면 우리는 어떻게 원을 쪼개어 각각을 버킷으로 나눌지를 고민해야 한다. 각각의 점의 거리를 이용하여 쪼개는 것이 가장 효율적인 방법일 것이다. 각각의 점의 거리를 구하는 연산은 상수 시간에 완료할 수 있다. 그리고 각각의 점이 어떤 버킷에 포함될지 결정하는 연산은 버킷을 몇 개로 쪼개는가에 따라 달라지게 되는데, 버킷 정렬에서의 버킷의 수는 입력 자료의 크기에 비례하여 증가하는 것이 각 버킷에 들어가는 자료의 개수가 줄어드므로 여기서는 n개의 버킷으로 나눈다고 하면 버킷에 포함하는 연산은 선형 시간에 완료할 수 있다. 버킷 안에서의 삽입 정렬은 입력이 균등 분포를 하고 있고 버킷이 입력 자료의 크기에 맞추어 변화하기 때문에 극히 양이 적다고 생각하여 (예를 들어, n개의 자료가 n개의 버킷을 이용해 버킷 정렬을 한다고 생각하고 자료가 균등 분포를 하고 있음을 가정한다면 각 버킷에는 1개 정도의 자료 밖에 들어가지 않는다) 상수 시간으로 근사할 수 있다고 생각한다. 자세한 증명은 수업시간에 했으므로 생략한다.

Problem 4 continued on next page...

그러면 이제 각각의 버킷을 설정해주어야 한다. 위에서 설명했듯이 원점으로부터 떨어진 거리를 이용하여 입력 자료들을 분배하는 것이 가장 효율적이고 타당한 방법이므로, 각각의 반지름으로 구분된 영역의 넓이가 동일하게 버킷을 설정해주면, 각각의 버킷에 데이터가 들어갈 확률이 동일하게 된다. $r_0 = 0$, $r_n = 1$이고, d를 원점으로부터 각 점까지의 거리라고 하고, 각각의 버킷을 $r_{i-1} < d \leq r_i$처럼 표현하면 r_i $(i = 0, 1, \ldots, n)$ Equation 2와 같은 식을 만족한다.

$$\pi(r_n^2 - r_{n-1}^2) = \pi(r_{n-1}^2 - r_{n-2}^2) = \cdots = \pi(r_1^2 - r_0^2) \tag{2}$$

$r_0 = 0$, $r_n = 1$이고, 제곱의 차가 일정하며, 각 항간은 1씩 차가 일정하게 난다는 점을 이용하면 Equation 3와 같이 만들어 낼 수 있을 것이다.

$$r_i = \sqrt{\frac{i}{n}} \tag{3}$$