

Algorithms HW1

Mechanical Engineering

2019-15838 주 기영

#1. Linear-search 함수 작성

알고리즘:

LinearSearch(A, v)

```

1: loc = NIL // NIL은 특정한 값
2: for i = 1 to A.length
3:   if A[i] = v
4:     loc = i
5:   return loc
6: return loc
    
```

loop에서 invariant 한 특성:

index i 를 시작하기 전에

$(1 \leq k \leq A.length, k < i, A[k] = v)$ 인 k 가 존재하지 않는다.

① 초기조건: $i=1$ 을 시작하기 전에 k

$1 \leq k \leq A.length, k < 1$ 인 k 는 존재하지 않는다.

② 유지조건: index i 에서 loop invariant가 성립하면

만약 loop가 실행된다면 다음에서 $A[i] = v$ 가

아닌 것이므로, 다음 loop에서 loop invariant가 성립한다.

③ 종료조건:

만약 line 3의 if문의 조건문이 참이어서

loop가 끝났다면 $(1 \leq k \leq A.length, A[k] = v)$

인 k 를 찾은 것이므로 목적을 이룬 것이다.

만약 $i=1 \dots A.length$ 까지 모든 loop에서

line 3의 조건문이 거짓이어서 $i=A.length$ 에서

loop를 빠져 나면

$(1 \leq k \leq A.length, k < A.length+1, A[k] = v)$

조건을 만족시키는 k 가 존재하지 않는다는 것이다.

따라서 이 경우에 NIL 값을 반환하여 목적을 이룬다.

#2. Selection Sort 함수 작성

Selection Sort (A)

```

1: for i = 1 to A.length-1
2:   min = i
3:   for j = i+1 to A.length
4:     if A[j] < A[min]
5:       min = j
6:   A[min] ↔ A[i]
    
```

loop invariant:

index i 를 시작하기 전에 i 보다 작은 값은 없다.

index로 갖는 배열의 원소 중 즉 A 배열 전체에서 k 번째로 작은 숫자이다.

① 초기조건: $i=1$ 보다 작은 값을 index로 갖는

배열의 원소 값이 없으므로 항상 성립한다.

② 유지조건: $A[1 \dots i-1]$ 은 A에서 차례로 $1 \dots i-1$

번째 작은 수이다.

line 3-5에서 $A[i \dots A.length]$ 까지

값 중 가장 작은 값의 index를 min 변수에

저장하므로, line 3-5의 for문이 끝났을 때

min 변수는 $A[i \dots A.length]$ 부분배열에서

가장 작은 값의 index를 가지고 있고, 이는

A 전체에서 i 번째 작은 값이다. 따라서 이제

만약 line 6에 의해서 $A[i]$ 는 A에서 i 번째

작은 값을 원소로 갖게 된다.

③ 종료조건: i 가 $A.length - 1$ 일 때 종료할

때 $A[1 \dots A.length - 1]$ 부분배열을 순서대로

1, 2, 3, ..., $A.length - 1$ 번째로 작은 수를 가지고

있다. 따라서 $A[A.length]$ 는 자동적으로 A에서

가장 큰 값을 갖게 된다. 이것이 $i = A.length$ 인

경우에 for loop를 수행하지 않아도 selection

sort가 정상적으로 수행되는 이유이다.

Running Time 계산

하나라도 i index에 대해서 $A[i+1 \dots A.length]$

부분 배열에서 최솟값의 index를 찾는 것은

$A.length - i$ 번 수행한다. 배열의 크기 $A.length$

를 n 이라고 하면 running time을 다음과

같이 계산할 수 있다.

$$\sum_{i=1}^{n-1} (n-i) = n(n-1) - \sum_{i=1}^{n-1} i$$

$$= n(n-1) - \frac{(n-1)n}{2} = \frac{n(n-1)}{2}$$

배열의 상태와 상관없이 항상 비교하는

횟수가 똑같으므로 (부분배열에서 최솟값을 찾기 위해서

항상 같은 횟수의 비교가 필요하기 때문이다.)

최선의 경우, 평균적 경우, 최악의 경우 running time

은 모두 $O(n^2)$ 으로 계산된다.

$$\#3. T(n) = T(n-1) + n$$

\Rightarrow Inductive hypothesis 이용.

$T(n) = O(n^2)$ 이라고 가정하자

$\Rightarrow T(n) \leq cn^2$ 인 c 를 찾아야 한다.

먼저 $T(1) \leq c$ 인 조건이 있다 ($n=1$ 인 경우)

$$T(n) = T(n-1) + n \leq c(n-1)^2 + n$$

$$= c(n^2 - 2n + 1) + n = cn^2 + (1-2c)n + 1$$

$$cn^2 + (1-2c)n + 1 \leq cn^2 \text{인 } c \text{와 } n \text{의}$$

조건을 확인해보자.

$$\Rightarrow (1-2c)n + 1 \leq 0$$

$$\boxed{(2c-1)n \geq 1} \quad n \text{이 자연수이므로,}$$

$2c-1 \geq 1$ 이면 n 이 모든 수를 성립한다.

처음에 $T(1) \leq c$ 의 조건이 있었으므로

$c = \max(1, T(1))$ 으로 설정하면 모든 n 에 대하여

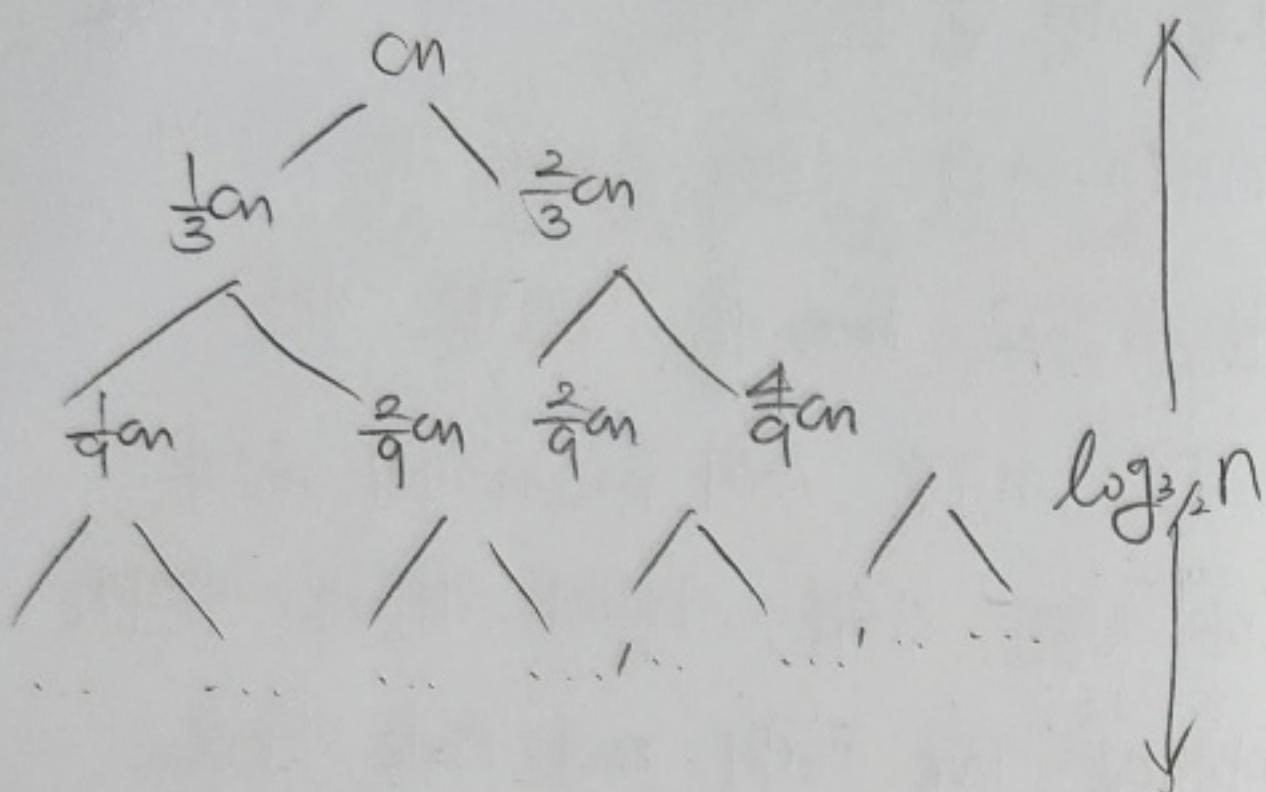
$T(n) \leq cn^2$ 인 c 가 존재함을 보일 수 있다.

따라서 $T(n) = O(n^2)$ 이다.

#4. $T(n) = T(\frac{n}{3}) + T(\frac{2}{3}n) + cn$ 의

해가 $\Omega(n \lg n)$ 임을 보여라.

책의 4.6의 그림을 간략하게 그려라



여기서 관찰할 점은 만약 tree의 같은 level에 모든 node가 존재한다면 이 모든 node들이 cn 이라는 것이다.

이는 $T(n) = T(\frac{n}{3}) + T(\frac{2}{3}n) + cn$ 에서

$n = \frac{1}{3}n + \frac{2}{3}n$ 이 성립하기 때문임을 간단히 확인할

수 있다. tree의 같은 level에 node가 모두

존재하는 곳까지의 코이를 생각해 보자.

이는 root에서 시작하여 left node로 계속 이동하라

가 leaf node를 만날 때까지, 이 path의

길이는 $\log_3 n$ 이다. 따라서 알고리즘의 비용

대수와 같이 나타낼 수 있다, 그에 따라 이 알고리즘이

$\Omega(n \lg n)$ 임을 증명할 수 있다.

$$cn(\log_3 n + 1) \geq cn \log_3 n = \frac{c}{\lg 3} n \lg n = \Omega(n \lg n)$$

#5. $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$

$$T(n) = 2T(\frac{n}{4}) + n^{\frac{1}{2}}$$

master theorem 적용 꼴을 생각하자.

$$T(n) = aT(\frac{n}{b}) + f(n)$$

이 문제의 경우 $a=2, b=4, f(n)=n^{\frac{1}{2}}$

$$f(n) = n^{\frac{1}{2}} = \Theta(n^{\log_4 2}) = \Theta(n^{\frac{1}{2}}) \text{ 이 성립}$$

master theorem의 2번째 case

<2번째 case>

$$f(n) = \Theta(n^{\log_b a}) \text{ 이면 } T(n) = \Theta(n^{\log_b a} \lg n)$$

$$\therefore T(n) = \Theta(n^{\frac{1}{2}} \lg n)$$

$$\text{Ans. } T(n) = \Theta(n^{\frac{1}{2}} \lg n)$$

#6

$$X_i = \begin{cases} 1 & \text{주사위에서 } i\text{-번째 눈이 나오는 경우} \\ 0 & \text{주사위에서 } i\text{-번째 눈이 나오지 않는 경우} \end{cases}$$

주사위 하나를 굴렸을 때, 나오는 눈을 X 라고 하면

$$X = X_1 + 2X_2 + 3X_3 + 4X_4 + 5X_5 + 6X_6$$

$$= \sum_{i=1}^6 iX_i \text{ 로 표현이 가능하}$$

$$E(X) = \sum_{i=1}^6 iE[X_i] = \sum_{i=1}^6 iP_i[X_i] = \frac{1}{6} \sum_{i=1}^6 i$$

$$= \frac{1}{6} \times \frac{6 \times 7}{2} = \frac{7}{2}$$

주사위 n 개를 굴릴 때 각 주사위 눈은 $X_1, X_2, X_3, \dots, X_n$ 이라고 하면 총합 X 는

$$X = X_1 + X_2 + \dots + X_n \text{ 이고, 각각의 기대값은}$$

위와 계산했으므로

$$E(x) = E(x_1) + E(x_2) + \dots + E(x_n)$$

$$= \underbrace{\frac{1}{2} + \frac{1}{2} + \dots + \frac{1}{2}}_{n\text{개}} = \frac{1}{2}n$$

Ans. 기댓값 : $\frac{1}{2}n$

#7.

HeapSort(A)

1. BUILD MAXHEAP(A)
2. for $i = A.length$ downto 2
3. $A[i] \leftrightarrow A[1]$
4. $A.heap-size = A.heap-size - 1$
5. MAX-HEAPIFY(A, 1)

2-5의 for loop의 loop invariance 증명하여
Heap Sort가 A를 정렬하는지 증명

loop invariance: index i 에 대한 loop를
진행하기 전 $A[1 \dots i]$ 는 A 전체에서 작은
원소들로 구성된 최대 힙이며 $A[i+1 \dots n]$ 은
A에서 역순으로 큰 원소가 (즉, 이미 정렬된 상태에서)의
자리에 위치) $n-i$ 개 정렬되어 있는 배열이다.

① 초기 조건: $i = A.length$ 일 때, line 1에서
A가 Build max heap을 하였으므로, $A[1 \dots n]$
이 Max Heap이며, 순서대로 부분 배열은 원소가
존재하지 않으므로, loop invariant를 만족함.
(또한 $A.length$ 를 n 으로 쓰게 됨)
즉, $n = A.length$

② 유지 조건: index i 에서

loop의 시작 시 loop invariant가 만족한 값
가정. 이 때 $A[i]$ 와 $A[1]$ 를 바꾸는 line 3을 보자.
바꾸기 전 $A[1 \dots i]$ 는 max heap이므로, $A[i]$ 은
이중 가장 큰 값을 가지고 있으며 A 전체에서는
 $n - (n - i)$ 의 i 번째 들어야 하는 원소이다.

이를 수행하고, line 4를 수행하고 나면

$A[i \dots n]$ 은 A에서 $n - i + 1$ 개의 원소가
이미 정렬된 상태로, 자리에 위치하는 배열이다.
마지막으로 line 5에서 root node 기준으로,
HEAPIFY를 진행하면 heap-size를 1 줄였으므로,
 $A[1 \dots i-1]$ 은 MAX HEAP을 형성한다.

③ 종료 조건:

$i = 2$ 인 경우의 loop를 수행하고 나면
loop invariant에 의해 $A[2 \dots n]$ 은 정렬되었
으며 이미 최종 정렬 배열에서 자신이 위치할 곳에
존재한다. 따라서 자연스럽게 $A[1]$ 은 A에서 가장
작은 원소가 위치해있다는 의미가 되고, 즉,
 $A[1 \dots n]$ 이 정렬되어 있음을 증명할 수 있다.
 \Rightarrow Heap Sort가 A를 Sort한다.

#6. 가수정렬 (RADIX-Sort)의

Correctness 증명

RADIX-SORT(A, d)

1. for $i = 1$ to d
2. \rightarrow use a stable sort to sort array A on digit i

loop invariance

index i 가 시작 전 A 의 모든 수의

$i-1$ 번째 digit 이하를 취하여 정렬한 상태이다.

① 초기조건: $i=1$ 인 경우

trivial한 경우이다.

② 유계조건 (stable sort라는 가정의 필요한 부분) 가 A 를 정렬한다는 것이 증명 가능하냐,

index i 로 시작하였을 때 $i-1$ 번째 이하를

취한 수들로 정렬이 되어 있는 상태

이 상태에서 i 번째 digit로 A 를 stable sort

하면 같은 i 번째 digit를 갖는 숫자들은 stable

sort에 의해서 $i-1$ 번째 이하 숫자들은 정렬되어

있는 상태일 것이다. 따라서 바뀐 A 는

i 번째 이하 digit를 취한 수로 정렬이 된 상태로

만들어지며, 이는 loop invariance를 유지한다.

만약 stable sort가 아니라면

(ex)

912		914
913	\longrightarrow	912
914	3번째 digit으로 정렬	913

이러한 예시로, 가장 $i-1$ 번째 이하 숫자들로 정렬하면

A 가 망쳐지게 되므로, RADIX Sort에서 digit i 를 sorting하려면 반드시 stable sort가 필요하다.

③ 종료조건

$i = d$ 인 loop가 끝난 후에 A 는 d 이하의

digit을 취한 수로 정렬이 완료된 상태이다.

이때 d 는 highest-order digit이므로,

d 이하의 digit을 취한 수가 즉 A 의 원래

요소들로 판단이 가능하냐, 따라서 RADIX Sort

가 A 를 정렬한다는 것이 증명 가능하냐,