Homework #2

Due: 10/25

HW2 instructions

이번 HW2 는 이론 4 문항과, 2 개의 coding 및 analysis 실습 문항으로 이루어져 있습니다. 모든 제출 파일은 하나의 zip 파일로 묶어서 제출해 주세요. 코드는 Python(.py extension)으로 작성하시기 바랍니다.

파일 1: 학번_이름.pdf
파일 2: 학번_이름_5.py
파일 3: 학번_이름_6.py

→ **HW2_학번_이름.zip** 압축 후 제출

(Total 100 points)

1. **(5 points)** 아래 조건을 만족하는 hash family 에 대해 답하시오.

> $\mathbf{h_b}(\boldsymbol{x}) = \mathbf{(999}\boldsymbol{x} + \boldsymbol{b}\mathbf{)}$ **mod 11** where $b \in [0, 10]$
> and the argument x to the hash function is from the universe $\{0, 1, 2, \ldots, 21\}$

hash family 가 universal 한가? 답과 풀이과정을 적으시오.

Not Universal

서로 다른 x_i , x_j에 대하여 $\mathrm{h_b}(x)$에서 임의로 뽑은 hash function, h 가 주어졌을 때, $P\left(h(x_i) = h(x_j)\right) > \frac{1}{11}$ 인 반례를 찾으면 hash family 는 not universal 이다.

반례: $x_i = 0$, $x_j = 11$ 일 때, hash family 에 존재하는 어떠한 hash function 을 골라도 $h(x_i) = h(x_j)$이기 때문에, $P\left(h(0) = h(11)\right) = 1 > \frac{1}{11}$ 이다.

채점 기준: not universal 임을 반례를 들어 보이면 5 points, 그 외 0 point

2. **(15 points)** M 을 임의의 큰 자연수라고 가정하자. 이 때, unsorted integer array A 는 아래 조건을 만족한다. 쉽게 말해, A 는 1 과 M 사이의 n 개의 서로 다른 자연수를 원소로 갖는다.

$$A = [a_1, a_2, a_3, \cdots, a_i, \cdots, a_{n-1}, a_n]$$

모든 $1 \le i \le n, 1 \le j \le n$ 에 대해, $1 \le a_i \le M$ 와 $if\ i \ne j,\ a_i \ne a_j$을 만족한다.

위 사실을 기반으로, A 안에 수치적 간격 T 안에 존재하는 두 숫자가 있는지를 판단하는 알고리즘을 디자인하려고 한다. [Input: A, T/output: Yes or No]

간단한 예로, M = 2000, n = 7, A= [100, 2000, 60, 1000, 1755, 1400, 1] 이라고 하자.

이 때, 직관적으로 가장 가까운 두 수는 60 과 100 으로, 수치적 간격은 40 이다.

따라서, 해당 알고리즘은 T = 1~39 일 때에는 No, T >= 40 이라면 Yes 를 return 할 것이다.

(a) $O(n^2)$의 time complexity 를 갖는 해당 알고리즘을 디자인하시오. (2 Points)

$O(n^2)$: Array 내의 모든 element 들을 한 번씩 pair 로 묶어 T 간격 안에 있는지 확인한다.

(b) $O(n \log n)$의 time complexity 를 갖는 해당 알고리즘을 디자인하시오. (5 Points)

$O(n \log n)$: Merge Sort 등을 이용하여 $O(n \log n)$으로 원소들을 정렬한 후,

$O(n)$: 근접한 원소들을 순서대로 비교하여 T 이하의 간격이 나올 때까지 찾는다.

→ $O(n \log n)$

(c) $O(n)$ 알고리즘을 디자인하시오. (Hint: bucket sort 응용) (8 Points)

크기가 T+1 인 bucket 들을 $(n\ //\ (T+1)) + 1$ 개 만든다. 1 부터 T+1 까지의 원소를 첫번째 bucket, T+2 ~ 2T+2 의 원소들을 두번째 bucket 에 넣는 식으로, 각 원소들을 알맞은 bucket 에 할당한다. 모든 원소를 할당한 후, bucket 을 돌면서 아래 case 들을 check 한다.

1. bucket 에 원소가 없다면: pass
2. bucket 에 2 개 이상의 원소가 있다면: return yes
3. bucket 에 하나의 원소만 있다면: 근접한 bucket 의 원소와 비교

채점 기준: 문제에 대한 적절한 설명을 제시할 경우 정답

3. **(10 points)** 밑 빠진 독에 물을 붓던 콩쥐에게 n 마리의 두꺼비가 다가와 도와주겠다고 이야기한다. 두꺼비는 두 종류로, 항상 진실만을 말하는 복두꺼비 n//2 + 1 마리, 거짓말을 하고 도망갈 수 있는 독두꺼비 n//2 - 1 마리가 있다. 다행히 콩쥐는 생물 전공의 두꺼비 전문가라서, 특정 두꺼비가 복두꺼비인지 판단이 가능하지만, 그 과정에는 $\Theta(n)$-time 이 소요된다. 아래 그림에 제시된 알고리즘들 모두, 한 마리의 복두꺼비를 찾아낼 때까지 시도하는 것이다. 아래 표를 채우고, 그 이유를 설명하시오.

Hint

1. **n 마리 중 한 마리를 뽑았을 때, 복두꺼비일 확률은 $\frac{1}{2}$ 이라고 가정**

2. **Run time 표현은 tightest bound, $big-\Theta$ notation을 사용**

3. **Randomized Algorithm 의 Expected Runtime 구하는 방법: input 을 x, choice sequence 를 c 라고 할 때, 알고리즘이 소요하는 시간을 $T(x,c)$라 하자. 그러면 input x 에 대한 expected runtime, $\overline{T}(x)$와 모든 가능한 choice sequence c 들을 포함하는 set C 에 대해, $\overline{T}(x) = E_c(T(x,c)) = \sum_{c \in C} P(c)T(x,c)$ 이다.**

```
Algorithm 1
-----------------------------------------
input: A (array of n toads)
for 100 iterations do
|  Choose a random index i in {0,...,n-1};
|  콩쥐가 i번째 두꺼비를 조사한다;
|  if A[i] == 복두꺼비:
|      return A[i]
|__
return A[0]
-----------------------------------------
```

```
Algorithm 2
-----------------------------------------
input: A (array of n toads)
while true do
|  choose a random index i in {0,...,n-1};
|  콩쥐가 i번째 두꺼비를 조사한다;
|  if A[i] == 복두꺼비:
|      return A[i]
|__
-----------------------------------------
```

```
Algorithm 3
-----------------------------------------
input: A (array of n toads)
for i = 0, ... , n-1 do
|  콩쥐가 i번째 두꺼비를 조사한다;
|  if A[i] == 복두꺼비:
|      return A[i]
|__
-----------------------------------------
```

Algorithm	Monte Carlo or Las Vegas or not Randomized?	Expected running time	Worst-case running time	Probability of returning a 복두꺼비
Algorithm 1	MC	$\Theta(n)$	$\Theta(n)$	$1 - \frac{1}{2^{100}}$
Algorithm 2	LV	$\Theta(n)$	∞	1
Algorithm 3	Not Randomized		$\Theta(n^2)$	1

Randomized Algorithm 의 Expected Runtime 구하기

input 을 x, choice sequence 를 c 라고 할 때, 알고리즘이 소요하는 시간을 $T(x,c)$라 하자. 그러면, $\bar{T}(x)$: expected running time on a particular input x 와 모든 가능한 choice sequence c 들을 포함하는 C 에 대해,

$\bar{T}(x) = E_c(T(x,c)) = \sum_{c \in C} P(c)T(x,c)$ 이다.

이 문제에 specific 한 form 으로 바꾸면, 아래와 같이 나타내 볼 수 있다.

$\bar{T}(x) = \frac{1}{2} \times \Theta(\mathrm{n}) + \frac{1}{2^2} \times 2\Theta(\mathrm{n}) + \frac{1}{2^3} \times 3\Theta(\mathrm{n}) + \ldots + \frac{1}{2^k} \times k\Theta(\mathrm{n})$

Algo 2: $\bar{T}(x) = \Theta(\mathrm{n}) \times \sum_{k=1}^{\infty} \frac{k}{2^k} = 2 \times \Theta(\mathrm{n}) = \Theta(\mathrm{n})$

Algo 1: $\bar{T}(x) = \Theta(\mathrm{n}) \times \sum_{k=1}^{100} \frac{k}{2^k} \cong 2 \times \Theta(\mathrm{n}) = \Theta(\mathrm{n})$

Appendix $\sum_{k=1}^{\infty} \frac{k}{2^k} = 2$

$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$ $(if\ x < 1)$ → 양변을 미분 → $\sum_{n=1}^{\infty} nx^{n-1} = \frac{1}{(1-x)^2}$

Let $x = \frac{1}{2}$, $\sum_{n=1}^{\infty} n \frac{1}{2}^{n-1} = \frac{1}{\left(1-\frac{1}{2}\right)^2} = 4$ $\quad \therefore \sum_{n=1}^{\infty} n \frac{1}{2}^{n} = \frac{1}{2} \times 4 = 2$

Algo 1: 최대 100 번의 루프를 돌 수 있기에 worst-case runtime 은 $100 \times \Theta(\mathrm{n}) = \Theta(\mathrm{n})$으로 bound 할 수 있다. 하지만, 100 번째 loop 까지도 복두꺼비를 찾지 못할 수 있기에 correctness 를 보장하지는 않는다. 이 경우, $\frac{1}{2^{100}}$의 확률로 복두꺼비를 찾지 못한다. 따라서, 이 알고리즘은 정확도를 보장하지 않고, 실행시간을 보장하는 randomized algorithm 으로, Monte Carlo 알고리즘이다.

Algo 2: 최악의 경우 영원히 복두꺼비를 찾지 못한다. 하지만, 시행 횟수가 늘어날수록 복두꺼비를 return 할 확률은 1 에 수렴한다. 따라서 이 알고리즘은 정확도를 보장하고, 실행시간을 보장하지 못하는 randomized algorithm, Las Vegas 알고리즘이다.

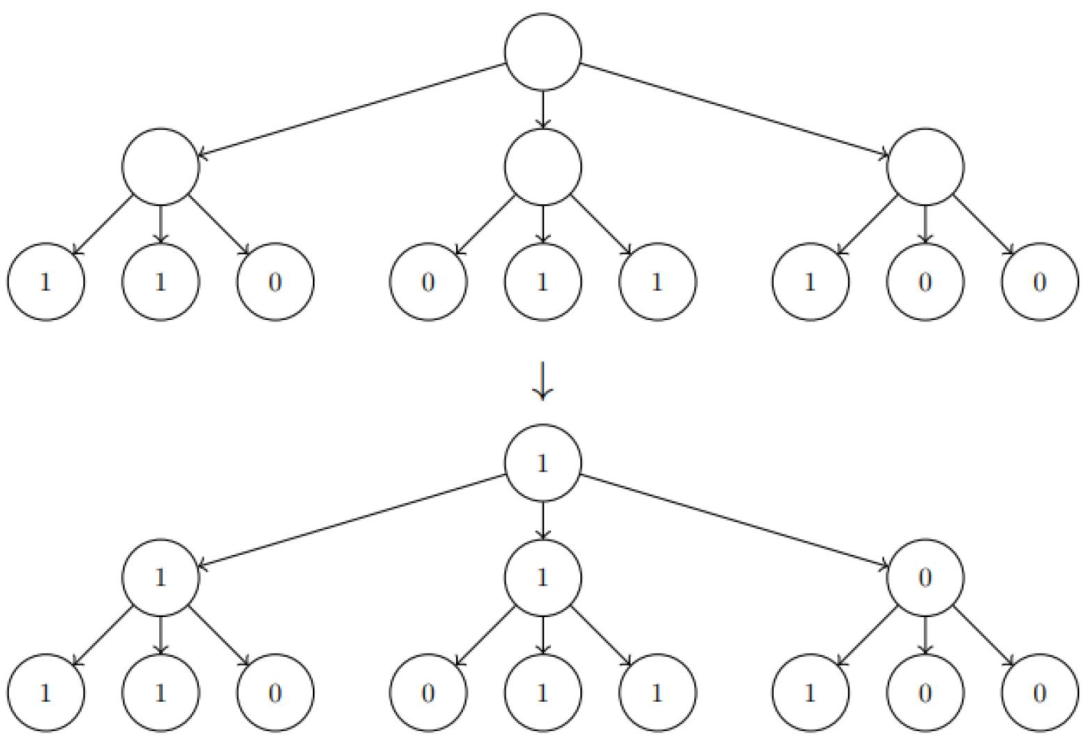
Algo 3: Not Randomized (별도 풀이 없어도 됨)

예외 사항: Algo3 Expected Run Time 을 비우는 것이 출제 의도였지만, $\Theta(\mathrm{n})$ 이라고 해도 정답처리

채점 기준: 표의 각 알고리즘 3 점(한 칸이라도 틀리면 해당 알고리즘은 0 점), 풀이 1 점

4. **(25 points)** Complete ternary tree T는 다음 조건을 만족한다.

1. T는 아래의 2 종류의 vertex(node)로 구성되어 있다.
   1) internal (not-leaf) vertices: have exactly three children
   2) leaf vertices: distance h from the root, where h is height of the tree.
2. Height를 높이는 방법으로만 vertices가 추가될 수 있다.
   즉, Complete tree이므로 항상 leaf node의 수는 3^h개다.
3. T와 관련된 자료형, map M은 leaf node $n = 3^h$개 각각을 n = 3^h개의 Boolean Value로 매핑한다. 특정 leaf node의 pointer, v 에 대해 Boolean value를 조회하기 위해서는, $M[v]$를 사용한다.
4. root 를 포함한 internal vertex 의 Boolean value 를 결정하는 데에는 해당 vertex 의 child vertices 의 과반수를 이용한다. (아래 예시 사진 참고)
5. 특정 노드가 leaf 노드인지 확인하는 방법은, $v.left,\ v.middle, v.right$가 None 을 return 하는 경우이다.



이 정보들을 토대로, root 의 Boolean value 를 return 하는 함수, ***majority_tree(root, M)***을 구현하려 한다. 이 때, *root* 는 *T.root* 로, 트리 *T* 의 root 를 가리키는 pointer 라고 생각하면 된다. 아래 질문들에 답하시오.

(a) *majority_tree* 함수를 divide-and-conquer 알고리즘으로 디자인하시오. 이 때, 모든 leaf node 들은 정확히 한 번씩 조회되어야 한다(i.e. for each leaf, your algorithm should index into `M` exactly once). pseudocode 를 작성하시오. [Hint: HW1 의 3way merge sort]

```
def majority_tree (root, M):
  if root.left is nil:
  return M[root]
  return majority_tree(root.left, M)
         + majority_tree(root.middle, M)
         + majority_tree(root.right, M) >= 2
```

(b) (a)에서 제시한 알고리즘은 recursive call 의 수를 줄일 수 있기에, 개선의 여지가 있다. 예를 들어, 예시 그림에서 root 는 left 와 middle 만 확인하면 이미 과반수가 1 이기 때문에 right 를 조회할 필요가 없다. (a) 알고리즘을 개선한 short-circuiting algorithm 을 디자인하여 pseudocode 를 작성하시오.

```
def majority_tree(root, M):
  if root.left is nil:
  return M[root]
  left = majority_tree(root.left, M)
  middle = majority_tree(root.middle, M)
  if left == middle:
  return left
  return majority_tree(root.right, M)
```

(c) (b)의 아이디어를 응용하여, Worst-case input 이 들어왔을 때, 평균적으로 $O(n^{0.9})$개의 leaf node 조회를 하는 randomized algorithm 을 디자인하려 한다. pseudocode 를 작성하시오.

Same as part (b), except choose a random set of two children to explore first instead of deterministically choosing the left and middle children. Part (b) was intended to be a strong hint for part (c).

```
def majority_tree(root, M):
   if root.left is nil:
   return M[root]
   first_child, second_child, third_child = random_order({root.left, root.middle, root.right})
   first_result = majority_tree(first_child, M)
   second_result = majority_tree(second_child, M)
   if first_result == second_result:
   return first_result
   return majority_tree(third_child, M)
```

(d) (c)에서 본인이 디자인한 알고리즘이 worst-case input 일 때, 평균적으로 $O(n^{0.9})$개의 leaf node 를 조회함을 증명하시오.

In the worst case, at level i, we have inputs with 2 of one value and 1 of the other. In these cases, we have a 1/3 chance of picking the correct two children as `first_child` and `second_child` and we only need to recurse twice on trees at level $i-1$. Otherwise, we have a 2/3 chance of needing to recurse three times on trees at level $i-1$. This produces the following recurrence relation:

$$T(i) = 1/3 * 2 * T(i-1) + 2/3 * 3 * T(i-1) = 8/3T(i-1)$$

To determine the value of the root, we expect to inspect at most 8/3 of its children, and 8/3 of their children, etc. Since $n = 3^h$, we have $h = log_3(n)$. So starting at level h, we expect to inspect at most $(8/3)^h = (8/3)^{log_3(n)} = n^{log_3(8/3)} = O(n^{0.9})$ leaves. This is an upper bound since we assumed a worst-case scenario for our tree.

(e) (c)에서 제시한 randomized algorithm 이 (b)에서 제시한 divide-and-conquer algorithm 과 비교했을 때 worst-case input 처리 측면에서 어떠한 장점이 있는지 간략히 설명하시오.

Same as the advantage of randomized quicksort vs. quicksort; an adversary cannot construct a worst-case input i.e. the randomized algorithm reduces the expected number of leaves checked by the algorithm for a worst-case input.

채점 기준: 각각 5points / (a) , (b), (c): 로직 차이가 명확히 드러나면 만점 / (d), (e): 적절한 설명 시 만점