Homework #2

Due: 10/23

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.

> $\mathbf{h_b}(x)$ = **(999x + b) 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.

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 $1 \leq i \leq n$, $1 \leq j \leq n$ 에 대해, $1 \leq a_i \leq M$ 와 $if\ i \neq j,\ a_i \neq a_j$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.

(답안은 pseudocode 작성 또는 접근법에 대한 상세 설명)

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

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

(c) $O(n)$의 time complexity 를 갖는 해당 알고리즘을 디자인하시오. (Hint: bucket sort 응용)

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				
Algorithm 2				
Algorithm 3				

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

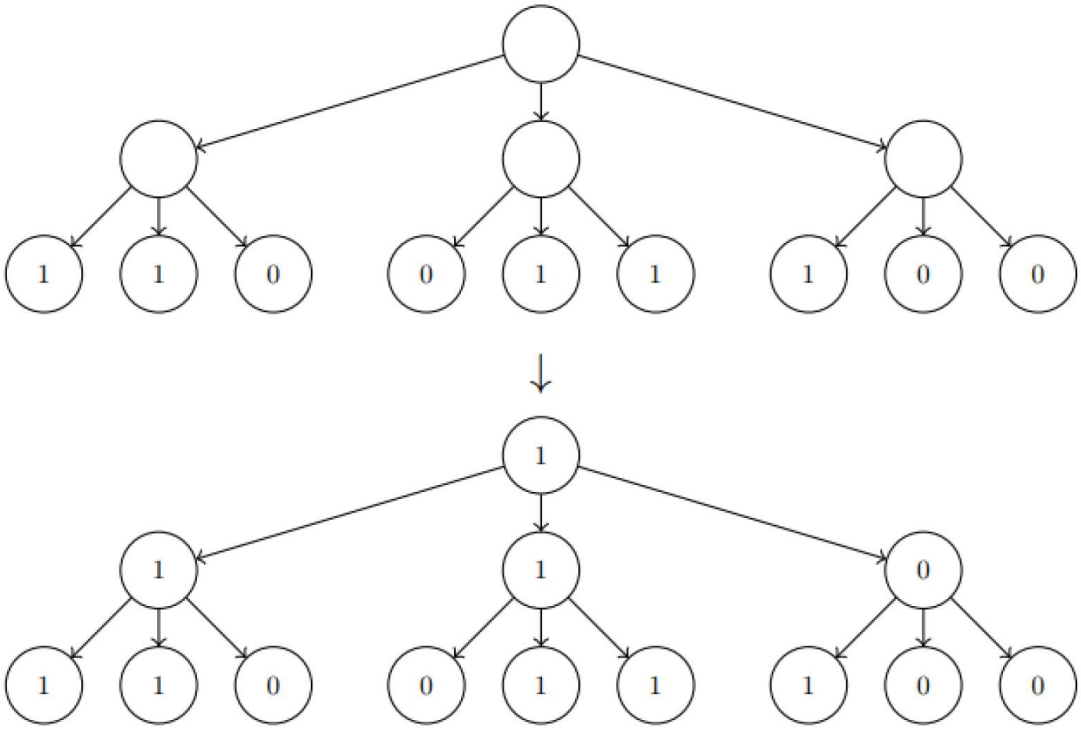
1. T 는 아래의 2 종류의 vertex(node)로 구성되어 있다.
   1) internal (not-leaf) vertices: have exactly three children
   2) leaf vertices: distance h from the root, where h is height of the tree.

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]

(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.

(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.

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

(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.

5. **(15 points)**
달구는 이번 달빛제에서 동아리 구성원들과 함께 주점을 꾸렸다. 정신없는 주점 운영시간이 끝나고, 현금 정산을 위해 모든 지폐를 한데 모아보니 분류되지 않은 지폐들이 총합 N 개, 동전이 총합 M 개 있었다. 이에 한 선배가 이야기한다.

> 어! 지폐가 N 개, 동전이 M 개라고? 내가 컴퓨터 알고리즘 때 들었는데, 그거 분류하려면 N+M 번이면 될 꺼 같지? 근데 무조건 $O((N+M)\log(N+M))$만큼은 걸려! 머지소트라고 들어봤냐? 신기하지 달구야?

(a) 해당 알고리즘은 $O(\max(N,M))$으로 구현할 수 있다. 아래 포맷에 맞춰 $O(\max(N,M))$의 알고리즘을 구현하시오. **(10 points)**
(제출 파일 형식: 학번_이름_5.py)

Input Format

첫째 줄에 지폐의 개수 N 이 주어진다.

둘째 줄에 동전의 개수 M 이 주어진다.

셋째 줄에 분류되지 않은 N 개의 지폐들이 주어진다.

(지폐의 종류는 1,000 원, 5,000 원, 10,000 원, 50,000 원 4 개로 한정)

편의를 위해, 1:1,000, 2:5,000, 3:10,000, 4:50,000 으로 대체한다.

넷째 줄에 분류되지 않은 M 개의 동전들이 주어진다.

(동전의 종류는 100 원과 500 원 2 개로 한정)

편의를 위해, 1:100, 2:500 으로 대체한다.

Output Format

출력은 아래 예시와 같이

첫째 줄에는 큰 금액부터 금액과 개수를 pair 로 묶어서 출력하고,

둘째 줄에는 sum 과 총금액을 pair 로 묶어서 출력한다.

(채점 시 띄어쓰기 차이로 오답이 될 수 있으므로 정확히 명시한 포맷으로 출력되도록 유의)

(₩50000, 2), (₩10000, 6), (₩5000, 6), (₩1000, 16), (₩500, 16), (₩100, 32)

(Sum, ₩217200)

Sample Input

13

20

4 3 4 1 4 3 4 1 2 2 2 3 1

1 1 1 1 1 1 1 2 1 1 1 1 2 2 2 1 1 1 1 1

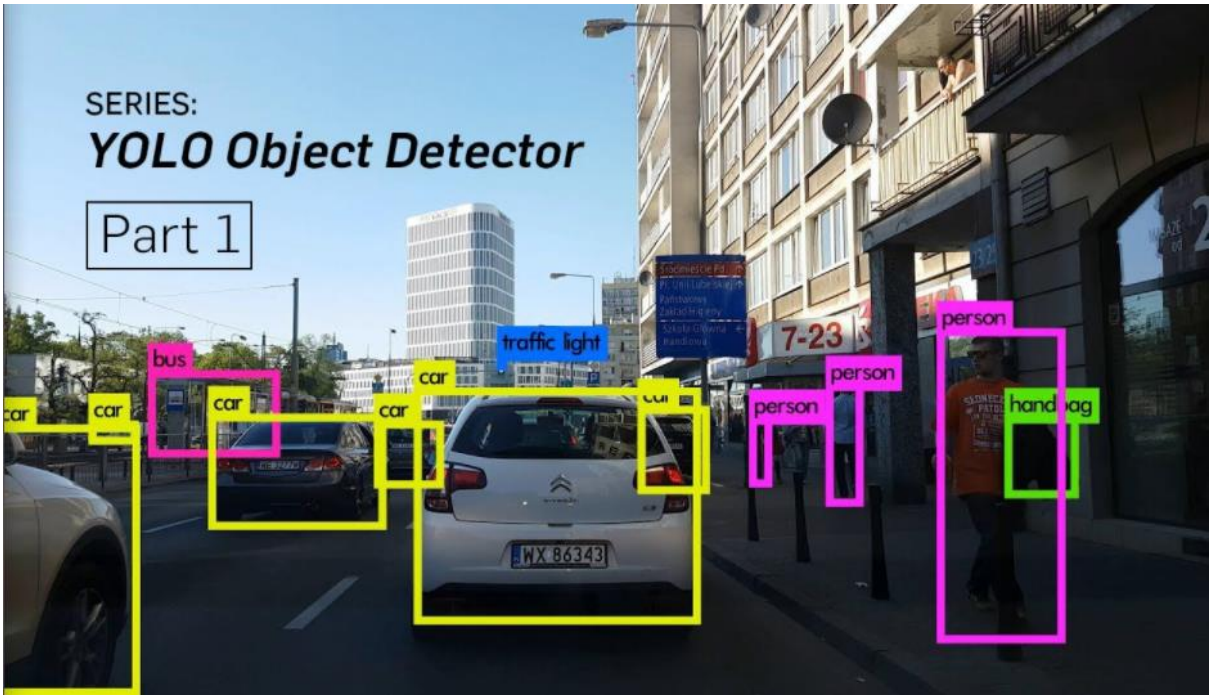
Sample Output

(₩50000, 4), (₩10000, 3), (₩5000, 3), (₩1000, 3), (₩500, 4), (₩100, 16)

(Sum, ₩251600)

(b) 앞서 작성한 알고리즘이 $O(\max(N, M))$임을 보이시오. **(5 points)**

6. **(30 points)** 3 학년이 된 달구는 UGRP 연구의 일환으로 **자율주행 알고리즘**을 개발하려 한다.



위의 예시에는 car, bus, traffic light, person, handbag 총 5 가지 종류의 12 개의 객체가 등장한다.

우선 달구는 객체들을 가로축 순서대로 아래와 같이 array 로 분류하는 프로그램을 미리 개발했다.

CaseInput: car | car | bus | car | car | car | traffic light | car | person | person | person | handbag

이 때, 달구는 위 array 를 input 으로 받아서, 특정 구간의 객체들을 chunk 로 묶어, 그 곳의 정보를 대략적으로 특정하는 프로그램을 짜려고 한다. (설명하는 예시들이 index 가 1 부터 시작함에 유의)

예를 들어, 1 번째 객체부터 5 번째 객체까지 묶고 싶다면, 아래와 같은 chunk 가 생길 것이다.

Chunk01: car | car | bus | car | car

또 다른 예로, 7 번째에서 9 번째까지 묶으면, 아래와 같은 chunk

Chunk02: traffic light | car | person

11 번째에서 12 번째까지 묶으면, 아래와 같은 chunk 가 형성될 것이다.

Chunk03: person | handbag

이 때, 각 chunk 는 객체 중 절반보다 많은 객체가 같은 객체라면 특정에 성공한 것이다. 즉, chunk 에 객체가 n 개 있고, n/2 보다 많은 객체가 같다면 특정에 성공한 것이다. 이 방법을 위 예시에 대입하면 Chunk01 은 car 로 특정 성공, Chunk02 와 Chunk03 은 특정 불가능이라고 할 수 있다.

Chunk C 개와 각 Chunk 에 포함된 객체가 주어졌을 때, 특정 성공 여부를 판단하는 프로그램을 작성하시오.

Input Format

첫째 줄에 객체의 수 N 과 객체 종류의 수 M 이 주어진다. ($3 \le N \le 30{,}000$, $1 \le M \le 1{,}000$, $M \le N$)

둘째 줄에 객체인식 결과의 가로축 순서대로, 각 객체의 종류를 번호로 구분하여 주어진다.

(앞 예시의 CaseInput 을 숫자로 변환했다고 이해하면 됨)

셋째 줄에 chunk 의 수 C 가 주어진다. ($1 \le C \le 1{,}000$)

다음 C 개의 줄에 두 정수 Start 와 End 가 주어진다. ($1 \le Start \le End \le N$)

이 줄은 Chunk 의 정보를 의미하고, Start 번째 객체부터 End 번째 객체까지 해당 Chunk 에 담겼다는 뜻이다.

(일반적인 인덱스 개념과 달리 인덱스가 1 부터 시작한다는 점에 유의)

Output Format

출력은 총 C 줄이다. 각 Chunk 가 특정되지 않았다면 "no"를 출력하고, 특정되었다면 "yes X"를 출력한다. 특정된 Chunk 의 경우 X 는 Chunk 의 절반이 넘는 객체의 종류이다.

Sample Input

12 5
1 1 2 1 1 1 3 1 4 4 4 5
7
1 2
1 3
7 9
7 10
7 11
11 12
1 12

Sample Output

yes 1
yes 1
no
no
yes 4
no
no

달구는 위의 기능을 수행하는 $O(CN)$의 알고리즘을 아래의 접근법으로 어렵지 않게 만들어냈다.

Algorithm 01.

각 chunk 마다 포함된 객체 종류를 key 로 갖고 그에 상응하는 인덱스들을 리스트형태의 value 로 갖는 dictionary 자료형을 만들어 주는데 $O(CN)$이 걸렸다. (아래 예시처럼)

⋮

Chunk4 = [1 1 2 1 1 1 3 1 4 4 4 5] Chunk_Dict4 = {1: [5, 6, 8], 3: [7]. 4: [9]}

Chunk5 = [1 1 2 1 1 1 3 1 4 4 4 5] Chunk_Dict5 = {1: [1, 2], 2: [3]}

⋮

이어서, 각 Dictionary 마다 가장 array 의 길이가 긴 key 를 찾아 그 길이가 과반수에 해당하는지 판별하였고, 그 과정은 최악의 경우 $O(CM)$이 걸렸다. 따라서, 전체 알고리즘은 $O(CN)$이 걸렸다.

하지만, 자율주행의 특성상, 안전을 보장하기 위해 worst-case runtime 을 최대한 줄여야 했고, 위의 $O(CN)$의 알고리즘은 너무 오래 걸려서 채택되지 못했다. 억울했던 달구는 컴퓨터 알고리즘 수업을 열심히 들어서 본인이 생각하는 최선의 worst-case runtime 을 갖는 다음과 같은 binary search with randomization algorithm 을 구현했다.

Algorithm 02.

우선, input list 를 아래 예시처럼 각 객체 종류마다의 인덱스를 분류하는 데에는 O(N)이 소요된다.

Input = [1 1 2 1 1 1 3 1 4 4 4 5] Dict = {1: [1,2,4,5,6,8], 2: [3], 3: [7], 4: [9, 10, 11], 5: [12]}

Chunk 속의 특정 객체를 뽑았을 때, 그것이 chunk 내에서 과반수인지 판별하는 logic 은 앞서 만든 Dictionary 를 binary search 로 탐색하면 최악의 경우에도 $O(\log N)$에 가능하다.

랜덤 뽑기 최대 횟수를 k 라고 설정하자.

Case 1: chunk 에 과반수인 객체가 있는 경우

Case 2: chunk 에 과반수인 객체가 없는 경우

Case 1 이면, 랜덤하게 어떤 객체를 뽑았을 때 그것이 답일 확률은 $\frac{1}{2}$ 이상이다. 그에 따라, 랜덤하게 k 번까지 뽑아도 단 한 번도 과반수인 수를 뽑지 못할 확률은 $\frac{1}{2^k}$ 이하가 된다. (문제 3. algo 1 과 같은 원리)

Case 2 이면 k 번 뽑고 항상 과반수를 찾지 못할 것이다.

최악의 경우, $O(\log N)$의 과반수 판별 과정을 $C \times k$번 수행해야 하므로, O(kC$\log N$)이 소요된다.

즉, 이 알고리즘의 runtime 은 k, C, N 에 대한 정확한 대소비교가 없으므로, O(max (N, kC$\log N$))이다.

(optional)

추가로, 최적화 옵션으로 이미 한번 판별하여 과반수가 아니라고 판단된 객체종류에 대해서는 다음 iteration 에서 뽑혔더라도, pass 한다. 더불어, 이미 판별했던 chunk 와 동일한 chunk 가 들어오면 이전의 기록을 이용하고 pass 한다.

Instruction

6. (a)

이 문제는 위의 달구의 예시처럼 다양한 접근법이 존재하는 문제이다. 우선, 접근법에 상관없이 Input 에 따른 Output 출력이 올바르다면 30 점 중 10 점을 부여한다. 스스로 해당 문제를 생각해보고, 그 python 파일을 제출하시오. **(10 points)**

(제출 파일 형식: 학번_이름_6.py)

6. (b)

Algorithm 의 정확도 측면에서, Algorithm 1 과 2 를 비교하시오. **(5 points)**

6. (c)

자신이 제출한 python 파일의 big-O notation 을 분석하여 Algorithm 2 와 비교하시오. 이 때, 분석한 big-O notation 은 Algorithm2 와 같거나 개선되어야 하며, 아래 case 로 분류하여 채점을 진행합니다.

1. Algo 2 보다 big-o notation 이 더 클 경우 **(3 points)**
2. 동일한 경우 **(15 points)**
3. 개선된 방법 제시 (세세한 최적화 옵션은 인정 X) **(25 points: additional 10 points)**

더불어, (c)의 경우, 보다 엄밀한 채점을 위해 N 과 C 가 큰 testcase 들에 대해

terminal 에서 아래의 command 로 실행시간을 측정하여, 위에서 설명한 Algo2 와 비교할 것이고, 제출한 분석 결과와 일치하는지 교차 검증할 예정입니다.

```
$ time python3 학번_이름_6.py < massive1.txt
```

Note that:

참고: Algo2 에서는 k 를 20 으로 설정하였음.

개선의 기준: 직관적으로, 본인이 제시한 알고리즘이 개선된 big-O notation 을 갖는다면 개선된 알고리즘임. 예를 들어, 본인의 알고리즘이 O(Clog*N*)의 time complexity 를 갖는다면, 명확히 개선된 알고리즘이라고 볼 수 있음.

하지만, 다음과 같은 애매한 경우가 존재함. 예를 들어 본인의 알고리즘이 O(N + CM)의 time complexity 를 갖는다면, Algo2 의 O(max (N, kClog*N*))와 대소비교는 불가능함. 이 때, 단 하나의 testcase 라도 실행시간이 악화되면 개선으로 인정하지 않음. 하지만, 일반적인 testcase 들에 대해서는 big-O notation 이 같아서 실행 시간이 비슷하지만, 특정 case 를 개선하였다면, 개선으로 인정함.

예를 들어, 본인의 알고리즘(이하 AlgoYours)이 N=256, M = 5, C = 100 (N +CM <= 20Clog*N*) 일 때에는 Algo2 에 비해 개선되었지만, N=4096, M = 5, C = 10 (N +CM > 20Clog*N*)일 때에는 오히려 악화되었다면, 개선으로 보기 힘듬.

하지만, N=256, M = 5, C = 100 (N +CM <= 20Clog*N*) input 에 대해서는 확실히 개선점을 제시하였기 때문에 아래 예시와 같은 포맷으로 big-O notation 비교를 통한 case 분류로 개선된 알고리즘을 제시하였음을 명시하면 개선된 방법으로 인정하도록 함.

```
if(case1): #for example, N +CM <= 20Clog N
    AlgoYours
else:
    Algo2
```

끝으로, "세세한 최적화 옵션은 인정 X"란 O(max (N, kClog*N*))의 동일한 time complexity 를 갖지만, 자료형 변경에서 오는 이점이나 특정 case 를 추가로 처리하여 빨라진 경우 등을 뜻하고, 그에 대해서는 추가점수를 부가하지 않는다는 뜻임.