ABC 134 解説

yuma000, drafear, DEGwer, evima, gazelle, potetisensei

2019 年 7 月 20 日

For International Readers: English editorial starts on page 7.

A. Dodecagon(writer : yuma000)

$3 * r * r$ を出力すればよいです。以下が C++ のサンプルコードです。

```
#include<iostream>
using namespace std;

int main() {
        int R;cin>>R;

        int area=R * R * 3;

        cout<<area<<endl;

        return 0;
}
```

B: Golden Apple

1 人の監視員を配置すると連続した $2D+1$ 本のりんごの木を監視できます。前から順に詰めて配置する (例えば最初の監視員には $1, 2, ..., 2D+1$ 番目の木を監視させる) のが最適なので、答えは N を $2D+1$ で割った切り上げになります。一般的に、整数 A, B に対して $\frac{A}{B}$ の切り上げは $\frac{A+B-1}{B}$ の商と等しいです。したがって、今回の場合は $\frac{N+2D}{2D+1}$ の商が答えになります。C++ での実装例を以下に挙げます。

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  int N, D; cin >> N >> D;
  int ans = (N + D * 2) / (D * 2 + 1);
  cout << ans << endl;
}
```

C: Exception Handling

N が数千以下 (言語によっては数万以下) なら、各問で A_i 以外の $N-1$ 個の要素すべてに対してループを回して最大値を直接求めても実行制限時間の 2 秒に間に合います。しかし実際には N は最大で 20 万であり、この方針では C++ でも間に合う望みはありません。計算時間を削減する方針を以下に二つ示します。

方針 1: 場合分け

ほとんどの場合、問いの答えは N 個すべての要素のうちの最大の値 $A_{\max}$ です。唯一の例外は問いで取り除かれる値 A_i が $A_{\max}$ と等しい場合で、この場合の答えはすべての要素のうち 2 番目に大きい値 A_{second} (数列中に $A_{\max}$ が複数回現れる場合は $A_{\mathrm{second}} = A_{\max}$ とします) です。問いの処理を始める前に $A_{\max}$ と A_{second} をあらかじめ求めておけば、各問を直ちに処理できます。なお、A_{second} を最も簡単な実装で求める方法は、与えられた数列をコピーして言語の標準ライブラリでソートすることでしょう (やや「余計」な計算をすることになりますが十分高速です)。

方針 2: 両端から攻める

$j = 0, 1, \ldots, N-1$ に対し、$A_1, A_2, \ldots, A_j$ のうちの最大の値を left_j とします (ただし $\mathrm{left}_0 = 0$ とします)。$j \geq 1$ のとき $\mathrm{left}_j = \max(\mathrm{left}_{j-1}, A_j)$ [*1] であることに注意すると、これらの値は一周のループですべて求められます。また、$j = 2, \ldots, N+1$ に対し、$A_j, A_{j+1}, \ldots, A_N$ のうちの最大の値を right_j とします (ただし $\mathrm{right}_{N+1} = 0$ とします)。$j \leq N$ のとき $\mathrm{right}_j = \max(\mathrm{right}_{j+1}, A_j)$ であることに注意すると、これらの値も一周のループですべて求められます。問いの処理を始める前にこれらの値をあらかじめ求めておけば、各問 i の答えを $\max(\mathrm{left}_{i-1}, \mathrm{right}_{i+1})$ として直ちに求められます。

[*1] $\max(a, b)$ は a と b のうち大きい方 (より厳密には小さくない方) を表します

D: Preparing Boxes

大きい数が書かれた箱からボールを入れるかを決めていくことにします。こうすると、整数 i が書かれた箱にボールを入れるか決めるとき、i 以外の i の倍数が書かれた箱については、すでにボールを入れるかが決まっています。それらの箱に入ったボールの総和の偶奇が a_i と異なる場合は箱にボールを入れて、そうでないときはボールを入れないことにします。このようにしてボールを入れるかを決めていくと、与えられた条件をすべて満たすようにボールを入れることができます。このアルゴリズムを愚直に実装すると $O(\frac{N}{1} + \frac{N}{2} + ... + \frac{N}{N})$ 程度の計算量がかかります。これは一見 $O(N^2)$ に見えますが、丁寧に解析すると $O(NlogN)$ で抑えることができます。よって十分高速です。

E: Sequence Decomposing

結論から言うと、答えは「与えられた数列の、**広義**単調減少列の長さの最大値 L」です。これを証明します。

数列の後ろの要素から順に、どの色で塗るかを決めていくことにします。最も後ろの要素に関しては、何らかの色で塗る必要があり、かつ他の要素に色はついていないので、適当に 1 色で塗っておきます。

後ろから 2 番目以降の数 A_i に関しては、「各色 c で塗られている要素の内、現在の最小の値 B_c はいくつか」を考えます。まず、$B_c <= A_i$ ならば、A_i を c で塗る事は条件に反するため出来ません。したがって、$B_c > A_i$ であるような色 c を用いるか、新しい色を使って A_i を塗る必要があります。

この時、$B_c > A_i$ であるような色 c の内、B_c が最小であるような色で A_i を塗るのが最適です。理由は以下のようにして分かります: まず、簡単のため、B_c を昇順に並べた列を考えます。ある色 c で A_i を塗る時、B_c の値が A_i に更新されると捉えることが出来ます。この時、B_c が最小であるような色で A_i を塗った際に更新された列 $\{B'_c\}$ と、それ以外の色で塗った際に更新された列 $\{B''_c\}$ を（ソートした状態で）考えると、同じインデックスに並んでいる B'_i および B''_i は値が等しいか、$B'_i > B''_i$ が成立するかの 2 通りしか考えられません。したがって、A_{i-1} 以降を色で塗る際には、B' を利用したほうが B'' を利用するよりも、新しい色を追加しなければならない状況に真に陥りにくいといえます。

結局、ソートされた $\{B_c\}$ を保持し、更新していく行為は $O(N \log N)$ の LIS の計算方法と同等であるため、$O(N \log N)$ で求める事が出来ます。

余談: この問題は Dilworth の定理を用いて考えると、より抽象的に議論・正当性の理解が出来ます。Dilworth の定理は、「DAG 上の最小 path 被覆 (全ての頂点を適当な path に属するようにして分割する時の必要な path の最小本数) は、最大 antichain（頂点の集合であって、集合に属する任意の 2 頂点に関して、それらを結ぶような path が存在しない）の要素数と一致する」という内容の定理です。今、この問題においては、与えられた数列の数を頂点に対応させ、$i < j$ かつ $A_i < A_j$ なる 2 点に有向辺を貼ることにすると、求める答えは最小 path 被覆に一致します。したがって、これは最大 antichain を求めれば良く、このグラフにおける antichain が満たすべき条件は、集合に含まれる任意の 2 要素について、$i >= j$ または $A_i >= A_j$ が成立するというものです。これは結局、数列上の**広義**単調減少列と一致するため、この問題の解法の正当性を示すことが出来ます。

F: Permutation Oddness

簡単のため、問題を以下のように言い換えます。

> うさぎ 1, うさぎ 2, ..., うさぎ N と、かめ 1, かめ 2, ..., かめ N がいる。うさぎとかめのペアを N 組作る方法のうち、ペアになったうさぎの添字とかめの添字の差の和 (これを「奇妙さ」と呼ぶ) が K になるものはいくつあるか。

結論から言うと、以下のように状態を定義することで、解を動的計画法で求めることができます。

- $dp[i][j][k][l]$ = うさぎ 1, ..., うさぎ i, かめ 1, ..., かめ i までを見て、ペアにするのを保留にしている (かめ $i+1$ 以降とペアにする) うさぎの数がj、ペアにするのを保留にしている (うさぎi+1以降とペアにする) かめの数が k、確定している奇妙さが l であるときの場合の数。

ここで確定している奇妙さとは、たとえばうさぎ i をかめ $i+1$ 以降とペアにする場合、少なくとも 1 の奇妙さが確定する、といったことを意味します。ところで、ペアを保留にしているうさぎの数とかめの数は一致します。よって DP テーブルの二次元目と三次元目は 1 つにまとめることができます。このとき DP の遷移は以下のように書けます。

- $dp[i][j][k] = (2*j+1)*dp[i-1][j][k-2*j]+(j+1)*(j+1)*dp[i-1][j+1][k-2*j]+dp[i-1][j-1][k-2*j]$

ここで右辺の第 1 項は、うさぎ i とかめ i をペアにする場合および、うさぎ i とかめ i のいずれかを保留にしておいた動物とペアにする場合を表します (ペアを保留にしている動物の数だけ、確定している奇妙さが増えます)。右辺の第 2 項は、うさぎ i とかめ i の両方を保留にしておいた動物とペアにする場合を表します。そして右辺の第 3 項は、うさぎ i とかめ i のどちらもペアにするのを保留する場合を表しています。

この動的計画法は状態数が $O(N^4)$、遷移が $O(1)$ なので、全体で $O(N^4)$ の計算量で解を求めることができます。

ABC 134 Editorial

yuma000, drafear, DEGwer, evima, gazelle, potetisensei

July 20, 2019

A. Dodecagon(writer : yuma000)

You can print $3 * r * r$. The following is the C++ sample code.

```
#include<iostream>
using namespace std;

int main() {
        int R;cin>>R;

        int area=R * R * 3;

        cout<<area<<endl;

        return 0;
}
```

B: Golden Apple

If you deploy 1 inspector, $2D+1$ apple trees can be inspected. It is optimal to deploy from left to right without space, so the answer is N divided by $2D+1$, rounded up. In general, for any integers A, B, the rounded up $\frac{A}{B}$ is equal to the quotient of $\frac{A+B-1}{B}$. Therefore, the answer for this problem is $\frac{N+2D}{2D+1}$. The following is an implementation example in C++.

```cpp
#include <bits/stdc++.h>

using namespace std;

int main() {
  int N, D; cin >> N >> D;
  int ans = (N + D * 2) / (D * 2 + 1);
  cout << ans << endl;
}
```

C: Exception Handling

If N is less than around thousands (or around tens of thousands, dependening on languages), even if you tried to find the maximum value of $N-1$ elements except for A_i using loops for each query, it will meet the time limit of 2 seconds. But practically N is at most 200 thousand and this is not a good strategy. There are two ways to reduce the execution time:

Strategy 1: Case Splitting

In most case, the answer is the maximum value of all N elements, $A_{\max}$. The only exception is that the excluded value a_i is equal to $A_{\max}$, in which case the second largest value A_{second} (if $A_{\max}$ appears multiples times, let $A_{\text{second}} = A_{\max}$) is the answer. If you precalculate $A_{\max}$ and A_{second} before handling the queries, each queries can be immediately answered. The easiest implementation to find A_{second} would be copying the array and sort it with the language's standard library (it needs some "extra" computation but it's fast enough).

Strategy 2: Calculating from the Both Sides

For all $j = 0, 1, \ldots, N-1$, let left_j be the maximum value among $\text{left}_0 = 0$ (here, let $\text{left}_0 = 0$). Since $\text{left}_j = \max(\text{left}_{j-1}, A_j)$ [*1] when $j \geq 1$, those values can be obtained in a single loop. Also, for all $j = 2, \ldots, N+1$, let right_j be the maximum value among $A_j, A_{j+1}, \ldots, A_N$ (let $\text{right}_{N+1} = 0$). Since $\text{right}_j = \max(\text{right}_{j+1}, A_j)$ when $j \leq N$, these values can also be obtained in a single loop. By precalculating those values before hanlding the queries, you can respond to each query immediately with the answer $\max(\text{left}_{i-1}, \text{right}_{i+1})$.

[*1] $\max(a, b)$ represent the greater value of (more strictly, not less) value

D: Preparing Boxes

Let's decide whether to put the ball in a decreasing order of box's indices. This way, when you consider whether or not to put a ball into box i, the boxes whose index is multiple of i except for i itself are already confirmed whether to put balls. If the parity of the number of balls in such boxes are different than desired a_i, put a ball into the box; otherwise, don't put the box. If you determine the choices in this way, you can find the optimal way of putting balls that satisfies all the given conditions. A naive implementation of this algorithm needs about $O(\frac{N}{1} + \frac{N}{2} + ... + \frac{N}{N})$. At first glance it seems $O(N^2)$, but a careful analysis shows that it's at most $O(NlogN)$. So, it's fast enough.

E: Sequence Decomposing

To come to the point, the answer is "the maximum length L of **non-increasing** sequence of the given sequnce." Here is a proof.

Let's decide which color to paint the sequence, from back to front. The very last element has to be painted with some color and the other elements are not painted yet, so it can be painted with a arbitrary color.

For each of the other elements A_i, consider "among the elements painted with color c, what is the current minimum value B_c?" First, if $B_c <= A_i$, you can't paint A_i with c, since it violates the given conditions. So, you have to use color c such that $B_c > A_i$, or new color, to paint A_i.

In such cases, among all color c such that $B_c > A_i$, when painting A_i it is optimal to choose one such that B_c is minimum. The following is the reason: first, for simplicity, consider a sequence in which B_c are sorted in a increasing order. Here, if you compare an updated sequence $\{B'_c\}$ where A_i was painted with the color with minimum B_c, and another updated sequence $\{B'_c\}$ where it was painted by any other color (both sequence being sorted), the two elements at the same position B'_i and B''_i are either equal or $B'_i > B''_i$. Therefore, when painting A_{i-1} and from then on, it is strictly less likely that you will need to add a new color if you prefer B' to B''.

Therefore, keeping and updating sorted $\{B_c\}$ is equivalent to $O(N \log N)$ LIS finding algorithm, so it can be solved in $O(N \log N)$.

Just for aside: This problem's discussion and validity can be understood in more abstract manner by using Dilworth's theorem. Dilworth's theorem is that "on any DAG, the minimum path cover (which is the size of set of path on the graph so that any vertex on it belongs to exactly one of them) is equal to the size of its maximum antichain (a vertex set such that any pair of vertices in the set does not have a path between them)." For this problem, let the elements of the given sequence be corresponding to vertices, and add directed edge between a pair of vertices such that $i < j$ and $A_i < A_j$, the answer is equal to its minimum path cover. Therefore, you can solve it by finding the maximum antichain, and the condition that the antichain on this graph should hold is that, for any pair of vertices in the set, $i >= j$ or $A_i >= A_j$ holds. This is eventually equal to the **non-increasing** subsequence on the sequence, and here the validity of this solution has been shown.

F: Permutation Oddness

There are Rabbit 1, Rabbit 2, ..., Rabbit N, Turtle 1, Turtle 2, ... and Turtle N. How many ways are there to make N pairs of Rabbits and Turtles, so that the sum of the difference between the index of Rabbit and Turtle of each pair (which we call "oddness") is equal to K?

To come to the point, you can solve this problem using DP by defining the state as follows:

- $dp[i][j][k][l]$ = The way of making pairs, where you have seen Rabbit 1, ..., Rabbit i, Turtle 1, ... and Turtle i, where there are j Rabbits that aren't still paired (so that they will be paired with Turtles later than i), where there k Turtles that aren't still paired (so that they will be paired with Rabbits later than i), and where the confirmed oddness so far is l.

Here, "confirmed oddness so far" means that, for instance, if you decided to make a pair of Rabbit i and Turtle later than i, at least 1 more oddness is additionally confirmed. By the way, the number of Rabbits and Turtles that aren't still paired are the same. So the second and third dimensions can be merged into one. Here, the DP recurrence equation can be written as follows:

- $dp[i][j][k] = (2*j+1)*dp[i-1][j][k-2*j]+(j+1)*(j+1)*dp[i-1][j+1][k-2*j]+dp[i-1][j-1][k-2*j]$

Here, the first term of the rhs (right-hand side) represents that Rabbit i is paired with Turtle i, or either Rabbit i or Turtle i is paired with an animal that aren't still paired (in which case the confirmed oddness increases by the number of animals that aren't still paired). The second term of rhs represents that each of Rabbit i and Turtle i is paired with animal that aren't still paired. The third term of rhs represents that both Rabbit i and Turtle i were withheld from being paired.

This DP has $O(N^4)$ states and needs $O(1)$ for each transition, so this can be solved in time complexity of $O(N^4)$.