ABC 128 解説

writer: gazelle, tozangezan, satashun, drafear, potetisensei, yuma000

2019 年 5 月 26 日

A: Apple Pie

見通しをよくするため、あらかじめ林檎はすべて砕いて林檎の欠片にしておきます。こうすることで、合計で $3A + P$ 個の林檎の欠片を用意できます。これらの林檎の欠片を使って、作れるだけアップルパイを作ればよいです。求める最大数は、$3A + P$ が偶数のときは $\frac{3A+P}{2}$ 個、奇数のときは $\frac{3A+P-1}{2}$ 個となります。以下に C++ での実装例を示します。

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int A, P;
    cin >> A >> P;
    cout << (3 * A + P) / 2 << endl;
    return 0;
}
```

B: Guidebook

多くのプログラミング言語では、string 型の配列を標準ライブラリのソート関数 (例: C++ の `std::sort`) によって辞書順に並べることができます。

問題文に合うようにレストランを並べるには、C++ でいう pair 型を利用すると簡単に実装できます。具体的には、`pair<pair<string,int>,int>` の配列を用意し、レストランの情報を格納します。first.first (1 番目のソート基準) に string 型で市名を入れ、first.second (2 番目のソート基準) に int 型で点数の -1 倍を入れ (点数が高い順に並べたいので)、second (3 番目のソート基準) にレストランの番号を入れます。

この pair 型をソートした後、 i 番目の要素の second には、本で i 番目に紹介するレストランの番号が格納されています。

pair 型がない言語などでは、まだ本で紹介されていないレストランのうち市名が辞書順で最も早く、その中で最も点数が高いレストランを順次ループを回して探していくことでも求めることができます。この場合は計算量は $O(N^2)$ となります。これでも間に合います。

C++ での実装例は以下の通りです (include 等は省いています)。

```
char in[120];
pair<pair<string,int>,int> p[110];
int main(){
  int a;scanf("%d",&a);
  for(int i=0;i<a;i++){
    int t;scanf("%s%d",in,&t);
    string tmp=in;
    p[i]=make_pair(make_pair(in,-t),i);
  }
  std::sort(p,p+a);
  for(int i=0;i<a;i++)printf("%d\n",p[i].second+1);
}
```

C: Switches

まず条件を無視すると、スイッチの on/off の組み合わせは 2^N 通りあります。条件を満たすスイッチの状態を直接数えるのは難しいので、この組み合わせを 1 つ固定して考えてみましょう。すると、それぞれの電球が点灯しているかどうかチェックし、全て点灯しているかどうか判定することで答えが求まります。この時間計算量は $\mathcal{O}(MN * 2^N)$ です。

なおスイッチの状態を固定する方法としては、DFS を用いる方法や、スイッチの状態を N bit の整数にエンコードする方法などがあります。

今回は全列挙することで解ける制約でしたが、mod 2 での連立方程式の解を数える問題と見なせるので、rank 等を計算することでより高速に求めることもできます。

解答例: https://atcoder.jp/contests/abc128/submissions/5640467

D: equeue

操作 C, D は最後に行っても構いません。すると、操作 C, D は等価で、「宝石を捨てる」操作になります。

操作 A を A 回、操作 B を B 回行うとします。$A+B \leq \min\{N, K\}$ です。このとき、左から A 個、右から B 個の宝石を手に入れ、捨てる操作は $K-(A+B)$ 回までできます。負の価値の宝石を捨てるとその絶対値分だけ合計価値が上がるので、価値が負でできるだけ絶対値が大きいものから $K-(A+B)$ 個まで捨てるのが最適です。価値が 0 や正の宝石を捨てる必要はありません。

A, B を全探索することで、$R = \min\{N, K\}$ として $O(R^3 logR)$ の時間計算量で解くことができます。

余談ですが、この探索を高速化して $O(R^2 logR)$ で解くこともできます[*1]。

[*1] ヒント: B を 1 ずらすことによって捨てる宝石は高々 2 個しか変わらないことを利用します

E: Roadwork

Q 人の人全てが整数時間に出発するので、時刻 $S_i - 0.5$ から $T_i - 0.5$ までの通行止めは時刻の区間 $[S_i, T_i)$ について通行止めされていると考えて差し支えありません (簡単のためです)。

i 番目の道路工事は、$[S_i, T_i)$ の間、座標 X_i を通行止めにします。したがって、逆算して考えると、この通行止めの影響を受ける可能性があるのは $[S_i - X_i, T_i - X_i)$ の間に座標 0 を出発した人だけです (それらの時刻以外に出発した人は、その通行止めに出くわすことは絶対にありません)。

よって、i 番目の人に対する答えは、その人が影響を受ける可能性がある道路工事を全て求め、それらの道路工事が通行止めにする座標の内、最も小さいものです。

Q 人それぞれについて、N 個の道路工事がその人に影響するかどうかを調べてしまうと、時間計算量が $O(NQ)$ となってしまい間に合いません。そこで、イベントソートを利用します。具体的には、「イベントを入れる配列」および「現時点で通行止めにされている座標を持つセット」を 1 つずつ用意し、以下の 2 種類のイベントを定義しておきます:

1. 追加イベント $(t, 1, x)$ - セットに x を追加する。
2. 削除イベント $(t, -1, x)$ - セットから x を削除する。

そして、N 個の道路工事全てについて、以下のようにしてイベントを配列へ追加します:

i 番目の道路工事については、

1. 追加イベント $(S_i - X_i, 1, X_i)$
2. 削除イベント $(T_i - X_i, -1, X_i)$

を配列に追加する。

配列に追加したイベントは t の値に従ってソートし、順番に処理します。i 番目の人に対して答えを求める際は、t の値が D_i 以下であるようなイベント全てを処理し終わったタイミングで、セットの最小値を調べれば良いです。

こうして、$O(N \log N)$ でイベントをソートし、各イベントについて $O(\log N)$ でセットを処理、各 Q 人について $O(\log N)$ で最小値を調べることができるので、全体で $O((N + Q) \log N)$ でこの問題を解くことができます。

F. Frog Jump

まず明らかに、溺れずに座標 $N-1$(以下ゴール) に辿り着く方が溺れるよりも最終点数が高いので、座標 -1 以下や N 以上に移動するような移動経路は考慮に入れる必要がありません。つまり、$0 < B < A <= N-1$ としてもよいです。

A と B の値を全部試して愚直にシミュレーションするやり方では $O(N^2 logN)$ かかってしまい、TLE してしまいます。色々な高速化の手法が存在すると思いますが、ここでは「最後はゴールに辿り着く」という特性を使った解法を説明します。

ゴールにたどり着くならば、それは奇数回の移動後です。ある A と B の値に対して、$2k+1$(k は非負整数) 回の移動でゴールにたどり着いたとき、その移動経路は、

$$0, A, A-B, 2A-B, 2A-2B, 3A-2B, ..., kA-(k-1)B, kA-kB, (k+1)A-kB$$

のようになります。このままでは分かりにくいので、$C = A - B(> 0)$ とおくと、

$$0, A, C, A+C, 2*C, A+2C, ..., A+(k-1)C, kC, A+kC$$

$A + kC = N-1$ なので、さらに変形すると、

$$0, (N-1)-kC, C, (N-1)-(k-1)C, 2C, (N-1)-(k-2)C, ..., N-1-C, kC, N-1$$

k と C さえ決めれば、経路は一意に定まります。その得点を f(k,C) とおきます。この時、

$$f(k+1, C) = f(k, C) + S_{N-1-kC} + S_{kC} (k >= 0)$$

が成り立つので、動的計画法を用いることにより、それぞれの k, C に対しての $f(k, C)$ は $O(1)$ で求まります。また、$kC < N-1$ より、考えるべき k, C の通り数は $O(NlogN)$ です。よって、$O(NlogN)$ で解けます。途中で溺れないために、同じ座標を二度通ってはいけないということに注意して下さい。