全国統一プログラミング王決定戦予選/NIKKEI Programming Contest 2019 解説

writer : maroonrk

2019 年 1 月 28 日

For International Readers: English editorial starts on page 6.

A : Subscribers

両方の新聞を購読している人数としてありうる最大値は A と B のうち小さい方であり、ありうる最小値は $A+B \leq N$ ならば 0、そうでなければ $A+B-N$ です。

`if` 文を使う代わりに、多くの言語に標準で実装されている関数 `min` や `max` を使うこともできます。以下に Python3 でのコードを示します。

```
N, A, B = map(int, input().split())
print(min(A, B), max(0, A + B - N))
```

B : Touitsu

文字列中の各位置 $1, 2, ..., N$ について、個別に 3 つの文字 A_i, B_i, C_i を揃えます。3 つの文字がすべて異なれば 2 回、3 つのうち 2 つが等しく残り 1 つのみが異なれば 1 回の操作がその位置について必要になります。

実装にはおそらく `for` 文を使うことになり、さらにそれぞれの位置について処理するために `if` 文か別の何らかの手段を使うことになります。

```
N = int(input())
A, B, C = input(), input(), input()
ans = 0
for i in range(N):
    ans += len(set([A[i], B[i], C[i]])) - 1
print(ans)
```

C : Different Strokes

問題の値: 『「最終的に高橋くんが得る幸福度の総和」から「最終的に青木さんが得る幸福度の総和」を引いた値』を X とします。高橋くんは X を最大化しようとし、青木さんは X を最小化しようとします。

もし青木さんがすべての料理を食べたと仮定すると、$X = -(b_1 + \ldots + b_N)$ となります。しかし、実際には高橋くんも料理を何皿か食べ、高橋くんが料理 i を食べるごとに X はここから $a_i + b_i$ ずつ増えていきます。よってこの問題は、高橋くんのみが料理 i を食べることで $a_i + b_i$ ポイントの幸福度を得て、青木さんは料理を食べても何も得られないとしても同等です (得られた答えに $-(b_1 + \ldots + b_N)$ を足す必要はあります)。これは、料理を $a_i + b_i$ の降順にソートすることで解けます。

D : Restore the Tree

制約により入力が不正でないことが保証されているため、頂点に 1 から N までの番号を振り直して次の条件を満たすようにすることがトポロジカルソートにより $\mathrm{O}(N+M)$ 時間で行えます: グラフの各辺 $u \to v$ について $u < v$ である。

この新しいグラフにおいて、頂点 1 が元の木の根であり、それ以外の各頂点 v について、元の木における v の親は辺 $w \to v$ が存在するような最大の w であることがわかります。なぜなら、辺 $x \to v$ が存在するような各頂点 x ($x \neq p_v$: 元の木における v の親) について、x は p_v の祖先であり、したがって $x < p_v$ であるからです。

E : Weights on Vertices and Edges

「最終状態における連結成分内で、最も重みの大きい辺」の候補になる辺を探します。辺 e が候補になる必要条件は、辺 e から、辺 e 以下の重みの辺のみを辿って行ける頂点を列挙したときに、列挙した頂点の重みの総和が辺 e の重み以上になることです。

各辺について、候補になるかをチェックします。まず、元のグラフとおなじ N 頂点を持ち、辺のないグラフを新たに用意します。次に、元のグラフの辺を重みの小さい順にこのグラフへ追加していきます。辺 e を追加し終えた直後に辺 e の属する連結成分の重みの総和が分かれば辺 e が候補になるかどうかのチェックが行なえます。そしてこれは、Union-Find などを用いると高速に行えます。

次に、辺をコストの降順に見ていきます。いま辺 e を見ているとします。e を残すことが決定している場合は、特に何もしません。e を残すことが決定していないが、先程の候補に上がっている場合は、e を使い、そこから e 以下の重みの辺のみを辿って行ける辺もすべて残すことにします。そうでない場合、e を残すことはできないとわかります。

このアルゴリズムの正当性は、重み最大の辺について残すか否かを正しく判定できることから、辺数に関する帰納法によって示せます。また、e 以下の重みの辺のみを辿って行ける辺もすべて残す、という操作については、一度探索された頂点や辺は別の辺から探索されることがないため、探索は全体で $O(N+M)$ になります。

このアルゴリズムは、辺を重みでソートする部分がボトルネックになり、$O(N+MlogM)$ で動作します。

F : Jewels

各色について、その色の価値が最も高い 2 つの宝石をペアにします。そして、ペアの宝石の価値を、それらの価値の平均としておきます。

適当に摂動して、すべての宝石の価値が異なるとしておきます。すべての宝石を価値の降順にソートし、前から見ていきます。前から x 個の宝石をそのまま選ぶことができる場合とそうでない場合があります。できないのは、ちょうど x 番目と $x+1$ 番目の宝石がペアになっているときです。前から x 個の宝石をそのまま選ぶことができるなら、それが明らかに最適な選び方なので、そうでない場合について考えます。

前から $x-1$ 個の宝石は、そのまま選ぶことができます。そこで、前から $x-1$ 個の宝石を選んだ状態を基準にして、1 個宝石を増やす方法を考えます。ここで、以下のような集合を考えます。

- P: 基準状態で 2 個以上選ばれている色の宝石であって、現在選ばれており、その色の価値 top2 には含まれないもの、の集合
- Q: 基準状態で 2 個以上選ばれている色の宝石であって、現在選ばれていないもの、の集合
- R: 基準状態でちょうど 2 個以上選ばれている色の、価値 top2 の宝石の組、の集合
- S: 基準状態で 0 個選ばれている色の、価値 top2 の宝石のペア、の集合
- T: 基準状態で 0 個選ばれている色の、価値 top3 の宝石の 3 つ組、の集合

基準状態から 1 個宝石を増やして価値を最大化する方法は、以下の 3 パターンを考えれば良いです。

- Q から 1 個選んで増やす
- P から 1 個選んで消し、S から 1 ペア選んで増やす
- R から 1 ペア選んで消し、T から 1 つ 3 つ組を選んで増やす

細かい証明は省略しますが、先頭の $x-1$ を価値の大きい順に取っているという事実を使うと、この 3 パターン以外はうまく変形することで上記の 3 パターンのうちいずれかを超えることができないことがわかります。

あとは、先頭から宝石を取りつつ、集合 P, Q, R, S, T を管理すればよいです。P, Q, R, S, T に必要な操作は、要素の挿入、削除、最大、最小の取得なので、これは例えば std::set などを用いることで行なえます。

よってこのアルゴリズム $O(NlogN)$ 時間で動作し、この問題を解くのに十分高速です。

NIKKEI Programming Contest 2019 Editorial

maroonrk

Jan 27, 2019

A : Subscribers

The maximum possible number of respondents subscribing to both newspapers is the smaller of A and B, and the minimum possible number of such respondents is 0 if $A+B \leq N$, and $A+B-N$ otherwise.

Instead of using `if` statements, you can also use the functions `min` and `max` that are implemented in many languages by default. Python3 code follows:

```
N, A, B = map(int, input().split())
print(min(A, B), max(0, A + B - N))
```

B : Touitsu

For each position $1, 2, ..., N$ in the strings, match the three characters A_i, B_i and C_i separately. If the three characters are all different, we need two operations for that position; if two of the three are the same but the other one is different, we need one operation.

You will most likely use a `for` statement in the implementation, and `if` statements or some other means to process each position. Python3 code follows:

```
N = int(input())
A, B, C = input(), input(), input()
ans = 0
for i in range(N):
    ans += len(set([A[i], B[i], C[i]])) - 1
print(ans)
```

C : Different Strokes

Let X be the value in question: "the sum of the happiness Takahashi earns in the end" minus "the sum of the happiness Aoki earns in the end". Takahashi tries to maximize X, while Aoki tries to minimize it.

If Aoki would eat all the dishes, $X = -(b_1 + \ldots + b_N)$. Actually, however, Takahashi also eats some dishes, and each time Takahashi eats Dish i, X increases by $a_i + b_i$ from here. Thus, the problem is equivalent to the case where only Takahashi earns $a_i + b_i$ points happiness by eating Dish i, and Aoki earns nothing from eating the dishes (we need to add $-(b_1 + \ldots + b_N)$ to the answer). This can be solved by sorting the dishes in ascending order of $a_i + b_i$.

D : Restore the Tree

Since the constraints guarantee that the input is valid, by topological sorting we can renumber the vertices 1 to N so that $u < v$ for each edge $u \to v$ in the graph, in $\mathrm{O}(N + M)$ time.

In this new graph, it can be seen that Vertex 1 is the root of the original tree, and for each of the remaining vertices, v, the parent of v in the original tree is the largest w such that there is an edge $w \to v$. This is because, for each of the vertices x ($x \neq p_v$: the parent of x in the original tree) such that there is an edge $x \to v$, x is an ancestor of p_v, and thus $x < p_v$.

E : Weights on Vertices and Edges

Let us find all the edges that can possibly be the heaviest edge in a connected component at the final state. Edge e can be such an edge if the total weight of the vertices that can be reached from e by traversing only edges with weights not greater than that of e.

For each edge, let us check if this condition is satisfied. First, we have a new graph with N vertices in the original graph and no edges. Then, we add the edges in the original graph in ascending order of weight. We can see if the condition is satisfied for edge e if we can find the total weight of the connected component that contains e just after e is added. This can be found efficiently by using data structures such as Union-Find.

Then, let us go over the edges in descending order of weight. Assume that we are now looking at edge e. If it is already determined that e remains in the final graph, we do nothing particularly. If it is not already determined but e can possibly remain in the final graph as we discussed earlier, we decide to use e in the final graph and also the edges that can be reached from there by traversing only edges with weights not greater than that of e. In the remaining case, we see that e cannot remain in the final graph.

The validity of this algorithm can be shown by induction on the number of edges, since it can correctly determine if the heaviest edge can remain in the final graph or not. Regarding the operation where we "decide to use the edges that can be reached from e by traversing only edges with weights not greater than that of e", the total time taken by these searches is $\mathrm{O}(N + M)$, since a vertex or edge visited once during these searches will never be visited again.

The slowest part of this algorithm is sorting the edges by weight, and works in $\mathrm{O}(N + M \log M)$ time.

F : Jewels

For each color, let us pair the two most valuable jewels, and set the values of these jewels as the average of the two.

By giving perturbation, we can assume that all the jewels have different values. Let us sort the jewels in descending order of value, and go over from front to back. In some cases we can just choose the x jewels from the front, and in some other cases we cannot. More specifically, it is when the x-th and $(x+1)$-th jewels are paired that we cannot just choose the x jewels from the front. Below, we will only consider this case, since it is optimal to just choose the x jewels from the front if it is possible.

We can choose to take the $x-1$ jewels from the front. With respect to the case where the $x-1$ jewels from the front are taken (we will call it "the base case"), let us consider how we have one more jewel. Let us define the following sets: (Below, we will call a color *used* if two or more jewels of that color are chosen in the base case, and *unused* otherwise.)

- P: the set of the jewels of a used color that are chosen now but not among the two most valuable jewels of the color
- Q: the set of the jewels of a used color that are not chosen now
- R: the set of the pairs of the two most valuable jewels of a used color
- S: the set of the pairs of the two most valuable jewels of an unused color
- T: the set of the triplets of the three most valuable jewels of an unused color

In order to maximize the total value, we only need to consider the following three ways to have one more jewel:

- Add one more jewel from Q.
- Remove one jewel that is in P, and add a pair of jewels from S.
- Remove a pair of jewels that is in R, and add a triplet of jewels from T.

We will omit the detailed proof, but it can be seen that any other way to have one more jewel is not better than one of these three ways by applying proper transformations, using the fact that the $x-1$ jewels from the front is taken in descending order of value.

Now, we only need to maintain the sets P, Q, R, S and T while going over the jewels. The operations required are insertions and deletions of elements and finding the minimum and maximum elements in sets, and this can be done by data structures such as std::set.

Thus, we have an algorithm that works in $\mathrm{O}(N \log N)$ time, which is fast enough.