ABC 150 解説

writers: camypaper, DEGwer, kyopro_friends, gazelle, satashun, tozangezan, yuma000

2020 年 1 月 10 日

For International Readers: English editorial starts on page 7.

A: 500 Yen Coins

高橋君が持っているお金の総額は $500 \times K$ 円です。これを X と比較し、$500 \times K$ が X 以上ならば Yes、そうでないならば No です。

Listing 1　C++ での実装例

```
#include <stdio.h>

int main(){
        int K,X;
        scanf("%d%d",&K,&X);
        if(500*K>=X)printf("Yes\n");
        else printf("No\n");
}
```

B: Count ABC

任意の i $(0 \leq i \leq n-3)$ について、S_i 以下 3 文字が ABC であるかを確認すればいいです。C++ による実装例を以下に示します。

Listing 2　C++ による実装例

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

int main() {
    int n;
    string s;
    cin >> n >> s;
    int ans = 0;
    for(int i = 0; i <= n - 3; i++) {
        if(s[i] == 'A' && s[i + 1] == 'B' && s[i + 2] == 'C') ans++;
    }
    cout << ans << endl;
    return 0;
}
```

C: Count Order

順列 X が辞書順で k 番目であることと、X より辞書順で小さい順列が $k-1$ 個あることは同値です。よって、大きさ N の順列をすべて生成して、そのうちに P, Q それぞれより小さいものがいくつあるかを数えることで、この問題を解くことができます。

順列の生成には、深さ優先探索などの探索を行う他に、C++ の next_permutation 関数のような順列生成用の関数を利用することも可能です。

D.SemiCommonMultiple(writer : yuma000)

X が A の半公倍数である条件は、以下のように言い換えられます。

- 任意の $k(1 \leq k \leq N)$ に対して、$X = (a_k/2) * (2p + 1)$ を満たす負でない整数 p が存在する。

X が $a_k/2$ の奇数倍であることから、これは以下のような二つの条件に分割できます。

- 任意の $k(1 \leq k \leq N)$ に対して、X は $a_k/2$ の倍数である。
- 任意の $k(1 \leq k \leq N)$ に対して、X が 2 で割り切れる回数と $a_k/2$ が 2 で割り切れる回数は同じである。

a_i が 2 で割り切れる回数と a_j が 2 で割り切れる回数が異なるような i, j が存在するとき、条件を満たす X はありません。

そのような i, j が存在しないときについて考えます。$a_1/2, a_2/2, ..., a_N/2$ の最小公倍数を T とします。すると、T を奇数倍したものが条件を満たすので、条件を満たす X の個数が求まります。

E: Change a Little Bit

まず、S,T の組を 1 つ固定して、$f(S,T)$ がどうなるかを考えてみましょう。

明らかに S,T で等しい要素を変更する必要はありません。ここで、$p_1,p_2,\cdots,p_k$ が S,T で異なるところの添字であり、この順番に変更するとします。このときのコストは $k*C_{p_1}+(k-1)*C_{p_2}+\cdots+1*C_{p_k}$ であるので、C の小さい方から変更するのが最適です。これは、ある i があって $C_{p_i}>C_{p_{i+1}}$ となるとき、この順番を swap してもコストが悪化しないことからわかります。

以上の考察と対称性より、C は広義単調減少であると仮定します。全ての $S\neq T$ の組について、$S_{i_1}\neq T_{i_1},S_{i_2}\neq T_{i_2}\cdots S_{i_k}\neq T_{i_k}(1\leq i_1<i_2<\cdots<i_k\leq N)$ のとき $\sum_{t=1}^{k}C_{i_t}*t$　を求めたいです。

総和の寄与を分解して、各 C_i が何回足されるかを考えます。これは $j\leq i$ で $S_j\neq T_j$ となる j の個数に依存します。これが k 箇所であるとすると、$\binom{i-1}{k-1}*2^k*2^{i-1-k}*4^{N-i}*C_i*k$ 寄与します。(各要素が異なる場合も等しい場合も、2 通りずつであることに注意してください)

これを k について足し合わせると $C_i*4^{N-i}*2^{i-1}*\sum_{k=1}^{i}\binom{i-1}{k-1}*k$ ですが、総和の中身も各要素が何回カウントされるかと読み替えることで $2^{i-1}*(i+1)$ であることがわかります。

以上より結局のところ答えは $4^{N-1}*\sum_{i=1}^{N}C_i*(i+1)$ と表せます。

ソート部分がボトルネックとなり、 $O(NlogN)$ でこの問題が解けました。

F:XOR Shift

数列の添え字は断りなく $\mathrm{mod} N$ で考えるものとし、XOR を $\oplus$ で表すこととします。

$$(1)\ \text{すべての } i \text{ で } a'_i = b_i$$

となるためには

$$(2)\ \text{すべての } i \text{ で } a'_i \oplus a'_{i+1} = b_i \oplus b_{i+1}$$

が成立していることが必要です。ここで $a'_i \oplus a'_{i+1} = (a_{i+k} \oplus x) \oplus (a_{i+k+1} \oplus x) = a_{i+k} \oplus a_{i+k+1}$ であることから、(2) の条件は k のみによって決まります。この条件を満たす k は、KMP 法などを用いることで、$O(N)$ で列挙することができます。（$c_i = a_i \oplus a_{i+1}$、$d_i = b_i \oplus b_{i+1}$ として、d の中から c を検索します）

逆に (2) を満たすような k に対しては、x が $b_i \oplus a_{i+k}$ であるとき、その時に限り (1) の条件を満たします。したがって、各 k に対してこのような x は定数時間で求めることができます。以上によりこの問題は $O(N)$ で解けました。

XOR になじみがない人は、まず XOR の代わりに足し算で同様のことを考えてみるとわかりやすいかもしれません。その場合 (2) の条件は $a'_i - a'_{i+1} = b_i - b_{i+1}$ となります。

A: 500 Yen Coins

Takahashi has $500 \times K$ yen. Compare this with X, and if $500 \times K$ is greater than or equal to X then the answer is Yes, otherwise No.

Listing 3 Sample code in C++

```
#include <stdio.h>

int main(){
        int K,X;
        scanf("%d%d",&K,&X);
        if(500*K>=X)printf("Yes\n");
        else printf("No\n");
}
```

B: Count ABC

For all i ($0 \leq i \leq n-3$), check if 3 letters starting from S_i is equal to ABC, and it's sufficient. The following is a sample code in C++.

Listing 4 Sample Code in C++

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

int main() {
    int n;
    string s;
    cin >> n >> s;
    int ans = 0;
    for(int i = 0; i <= n - 3; i++) {
        if(s[i] == 'A' && s[i + 1] == 'B' && s[i + 2] == 'C') ans++;
    }
    cout << ans << endl;
    return 0;
}
```

C: Count Order

X is the k-th in a lexicographical order if and only if there exist $k-1$ permutations that are lexicographically smaller than X. Therefore, you can solve this problem by generating all the permutations of length N, and counting how many permutations are smaller than P, Q.

When generating permutations, you can perform searching such as Depth-First Search, or you can also utilize functions generating permutations such as next_permutation function in C++.

D.SemiCommonMultiple(writer : yuma000)

The condition that X is a semi-common multiple of A can be rephrased as follows:

- For all $k(1 \leq k \leq N)$, there exists a non-negative integer p such that $X = (a_k/2) * (2p+1)$.

Since X is an odd-number-times multiple of $a_k/2$, it can be divided into the following two conditions:

- For all $k(1 \leq k \leq N)$, X is a multiple of $a_k/2$.
- For all $k(1 \leq k \leq N)$, the number of times X is divided by 2 is equal to the number of times $a_k/2$ is divided by 2.

If there exist i, j such that the number of times a_i is divided by 2 and the number of times a_j is divided by 2, then there does not exists X such that satisfies the conditions.

Assume that there doesn't exist such i, j. Let T be the least common multiple of T. Then, odd-number-times multiples of T satisfies the conditions, so you can find the number of X that meets the conditions.

E: Change a Little Bit

First, let's fix a pair of S, T and consider $f(S, T)$.

Obviously you don't need to change the same elements between S, T. Here, assume that $p_1, p_2, \cdots, p_k$ are the indices of S and T where they differs each other, and we change them in that order. In such case the total cost is $k * C_{p_1} + (k-1) * C_{p_2} + \cdots + 1 * C_{p_k}$, so it is optimal to change in the ascending order of C. This is followed by the fact that if there exists an i such that $C_{p_i} > C_{p_{i+1}}$, then swapping their order does not worsen the cost.

By the observations above and symmetry, we assume that C is monotonically non-increasing. For all pairs of $S \neq T$, we want to find $\sum_{t=1}^{k} C_{i_t} * t$ where $S_{i_1} \neq T_{i_1}, S_{i_2} \neq T_{i_2} \cdots S_{i_k} \neq T_{i_k} (1 \leq i_1 < i_2 < \cdots < i_k \leq N)$.

Decompose the contributions of sums, and consider how many times C_i are added. This depends on the number of j such that $j \leq i$ and $S_j \neq T_j$. If there are k such positions, then contributions are $\binom{i-1}{k-1} * 2^k * 2^{i-1-k} * 4^{N-i} * C_i * k$. (Note that there are always two patterns no matter each pair of elements is different or the same.)

By summing up this for each k we obtain $C_i * 4^{N-i} * 2^{i-1} * \sum_{k=1}^{i} \binom{i-1}{k-1} * k$. By regarding the contents of summation as how many times each elements are counted, it appears to be $2^{i-1} * (i+1)$.

Therefore, the answer can be represented as $4^{N-1} * \sum_{i=1}^{N} C_i * (i+1)$.

The bottleneck is the sort, and this problem could be solved in a total of $O(NlogN)$ time.

F:XOR Shift

We assume suffixes to be mod$N$, and $\oplus$ denotes XOR.

$$(1)\ a'_i = b_i \text{for all } i$$

implies that

$$(2)\ a'_i \oplus a'_{i+1} = b_i \oplus b_{i+1} \text{for all } i$$

. Here, it holds that $a'_i \oplus a'_{i+1} = (a_{i+k} \oplus x) \oplus (a_{i+k+1} \oplus x) = a_{i+k} \oplus a_{i+k+1}$, so condition (2) depends on only k. Such k can be enumerated by KNP algorithm or any other in a total of $O(N)$ time. (Let $c_i = a_i \oplus a_{i+1}$, $d_i = b_i \oplus b_{i+1}$, and find c in d)

Conversely, for such k that satisfies (2), condition (1) is satisfied if and only if x is $b_i \oplus a_{i+k}$. Therefore, for each k you can find such x in a constant time. As above this problem can be solved in a total of $O(N)$ time.

If you are not familiar with XOR, it may be easier to understand if you first consider the same things with sums instead of XOR. In such case condition (2) will be $a'_i - a'_{i+1} = b_i - b_{i+1}$.