Ограничена памет

Петър Петров

February 23, 2020

1 Събиране

Задача 1. *Дадена е пермутация на числата от 1 до n без последното число - $a_1, a_2, ..., a_{n-1}$. Намерете кое е липсващото число. Ограничения:* $1 \leq n \leq 10^6$.

Решение. В тази задача може да използваме информацията за дадените числа. Може да използваме някаква обща комутативна функция, например сумата, на всички числа и да сравним с дадените числа. Ако липсващото число е x, то $x + (a_1 + ... + a_{n-1}) = 1 + ... + n$. Така намираме $x = (1 + ... + n) - (a_1 + ... + a_{n-1})$. Сумата $1 + ... + n$ няма значение дали ще я сметнем с формула или ще обходим всички числа. Сумата $a_1 + ... + a_{n-1}$ трябва да я смятаме докато четем числата.
Ще отбележим, че сумата трябва да е от тип *long long*. □

Задача 2. *Дадена е пермутация на числата от 1 до n без последните две - $a_1, a_2, ..., a_{n-2}$. Намерете двете липсващи числа. Ограничения:* $1 \leq n \leq 10^6$.

Решение. Отново се насочваме към сумата. Ако липсващите числа са x и y, то $x + y = (1 + ... + n) - (a_1 + ... + a_{n-2})$. Това е добро начало, но не е достатъчно за да ги намерим двете числа.
Един вариант е да намерим още една подобна формула. Първата комутативна функция за която може да се сетим е произведението. Ясно е, че то ще е твърде голямо за да го запазим. Алтернативата е да го смятаме по модул. Ако използваме модул m може да пресметнем $p\_curr = (a_1 * ... * a_{n-2})\%m$ и $p\_exp = (1 * ... * n)\%m$. Сега обхождаме всички варианти за x, смятаме y от сумата и проверяваме дали $(x * y * p\_curr)\%m = p\_exp$. Понеже произведението е сметнато по модул не е гарантирано, че ще получим еднозначно отговор. За по-сигурно може да добавим втори модул, но за сега няма да навлизаме толкова надълбоко.
Всъщност има друга стандартна функция, която върши работа. Това е сумата от квадратите на числата. Така пресмятаме $x^2 + y^2 = (1^2 + ... + n^2) - (a_1^2 + ... + a_{n-2}^2)$.
Получаваме системата

$$\begin{cases} x + y = (1 + ... + n) - (a_1 + ... + a_{n-2}) \\ x^2 + y^2 = (1^2 + ... + n^2) - (a_1^2 + ... + a_{n-2}^2) \end{cases}$$

Сега ако положим $s = (1+...+n)-(a_1+...+a_{n-2})$ и $s2 = (1^2+...+n^2)-(a_1^2+...+a_{n-2}^2)$, получаваме $x + y = s, x^2 + y^2 = s2$. Ако приемем, че $x \geq y$, намираме $x - y = \sqrt{x^2 + y^2 - 2xy} = \sqrt{s2 - 2s}$.
Имаме $x + y$ и $x - y$, събираме и изваждаме и намираме самите числа. □

2 Броене на битове

Задача 3. *Дадена е редица от $3n+1$ цифри - $a_1, a_2, ..., a_{3n+1}$. Всички се повтарят кратен на три брой пъти с изключение на една. Намерете тази цифра.*
Ограничения: $1 \leq n \leq 10^6, 0 \leq a_i \leq 9$.

Решение. Тук нещата не изглеждат сложни. Ще преброим всяка цифра колко пъти се среща и ще видим коя бройка не е кратна на 3. □

Задача 4. *Дадена е редица от $3n+1$ числа - $a_1, a_2, ..., a_{3n+1}$. Всички се повтарят точно по три пъти с изключение на едно. Намерете числото, което не се повтаря.*
Ограничения: $1 \leq n \leq 10^6, 1 \leq a_i \leq 10^18$.

Решение. Нека да ограничим малко нещата. Да пробваме да намерим само последната цифра на числото. Гледаме всички цифри на последни позиции и броим коя колко пъти се среща. Тук нещата са аналогични на предната задача. Ако една цифра е от число, което се повтаря три пъти, то ще я преброим три пъти. Така бройката която не е кратна на три ще издаде последната цифра на търсеното число. По аналогичен начин може да намерим всяка една цифра. В $br[i][d]$ ще пазим колко пъти цифрата d се среща на i-та позиция. Позициите ще ги гледаме отзад напред, последната цифра е нулева позиция, предпоследната първа и т.н. За всяко число обикаляме по позициите на цифрите и попълваме бройките. Може да обикаляме докато има цифри или да използваме, че броя цифри е до 10.

```
void fillBr(int a) {
    for (int i = 0; i < 10; i++) {
        int d = a%10;
        br[i][d]++;
        a /= 10;
    }
}
```

След като минем през всички числа, за всяка позиция i ще видим коя цифра не се среща кратен на 3 брой пъти и ще си конструираме резултата.

```
int findAnswer() {
    int ans = 0;
    for (int i = 9; i >= 0; i--) {
        for (int d = 0; d < 10; d++) {
            if (br[i][d]%3 == 1) {
                ans = 10*ans + d;
                break;
            }
        }
    }
    return ans;
}
```

Друг подобен вариант на решение при който може да избегнем двумерния масив е да разглеждаме числото по битове - в двоична бройна система. Така за всеки бит е достатъчно да пазим в колко числа той е единици - ясно е че в останалите случай ще е нула, ако трябва може да го сметнем понеже имаме броя на числата. Нека $br[i]$ показва колко числа имат бит единица на

i-та позиция. Например $br[0]$ показва колко числа имат 1 за последна цифра(нулева позиция). Запълването на масива и конструирането на решението са аналогични на случая с двумерния масив.

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

int br[32];

void fillBr(int a) {
    for (int i = 0; i < 32; i++) {
        if (a%2 == 1) {
            br[i]++;
        }
        a /= 2;
    }
}

int findAnswer() {
    int ans = 0;
    for (int i = 31; i >= 0; i--) {
        ans = 2*ans;
        if (br[i]%3 == 1) {
            ans++;
        }
    }
    return ans;
}

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < 3*n+1; i++) {
        int a;
        cin >> a;
        fillBr(a);
    }
    cout << findAnswer() << endl;

    return 0;
}
```

□

Задача 5. *Дадена е редица от n числа - $a_1, a_2, ..., a_n$. Едно от тях се среща поне $\lfloor n/2 \rfloor + 1$ пъти, т.е. повече от половината пъти. Намерете го.*
Ограничения: $1 \leq n \leq 10^6, 1 \leq a_i \leq 10^{18}$.

Решение. Нека пак да намерим коя е последната цифра на търсеното число. Може да преброим всяка цифра колко пъти е последна в дадените числа. След това ще видим коя се

среща повече от половината пъти. Съвсем аналогично правим за всички позиции и решението е почти като предното. Единствената разлика е че в предната задача проверявахме дали бройката и кратна на 3, тук ще проверяваме дали е по-голяма от половината □

3 Изключващо ИЛИ

Дефиниция 1. *Да вземем два бита $b1$ и $b2$. Изключващото или показва дали тези два бита са различни. $b1\hat{}b2$ ще върне едно, когато са различни и нула, когато са равни. Ако приложим изключващо или на две числа, то операцията ще се приложи поотделно за битовете на различни позиции.*

Задача 6. *Дадена е редица от $2n+1$ числа - $a_1, a_2, ..., a_{2n+1}$. Всички се повтарят точно по два пъти с изключение на едно. Намерете го.*

Решение. Тази задача може да я решим с броене на битове - разглеждайки за всяка позиция коя цифра се среща нечетен брой пъти.
Има възможност вместо да гледаме всяка позиция поотделно да използваме изключващо или. Така за всеки бит, всеки две нули или единици ще се убиват една друга.

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

int main() {
    int n;
    cin >> n;
    int ans = 0;
    for (int i = 0; i < 2*n+1; i++) {
        int a;
        cin >> a;
        ans ^= a;
    }
    cout << ans << endl;

    return 0;
}
```

□

Задача 7. *Дадена е редица от $2n+2$ числа - $a_1, a_2, ..., a_{2n+2}$. n от числата се повтарят точно по два пъти. Намерете двете числа, което не се повтарят. Ограничения:* $1 \leq n \leq 10^6, 1 \leq a_i \leq 10^{18}$.

4 Повторно четене на входа

В процеса на решаване на една задача четем входните данни от стандартния вход. Когато за решаването на задачата ни трябва да минаваме повторно през тях ние ги записваме по време на четенето. При задачите с ограничена памет това е проблем, понеже няма достатъчно място да ги запишем. Има един трик обаче, който ни позволява да ги прочетем още един път от стандартния вход. Той работи поради факта, че реално към стандартния вход е пренасочен

файл. Когато се опитваме да се върнем в началото на входа се връщаме в началото на този файл. По тази причина ако тестваме с вход от конзолата, магията няма да сработи.

Повторно четене

С лек хак може да се върнем в началото на входа, след което може да го прочетем повторно. За целта използваме следната команда - $cin.seekg(0)$, която ни връща в началото.

Когато се налага да четем входа повторно бързото четене става още по-важно.

Задача 8. *Как четем бързо вход?*

Решение. Обикновено така:

```
ios::sync_with_stdio(false);
cin.tie(nullptr);
```

Ако четем числата символ по символ може да забързаме нещата още повече. Може да използваме *getchar* за целта. Ще четем символи докато стигнем първата цифра. След това ще четем докато имаме цифри, като междувременно си пазим в променлива резултатното число.

```
long long readLong() {
    char c = getchar();
    while (!isdigit(c)) {
        c = getchar();
    }

    long long ans = 0;
    while (isdigit(c)) {
        ans = 10*ans+c-'0';
        c = getchar();
    }
    return ans;
}
```

Може да отидем още една стъпка по напред, ако вместо $getchar()$ използваме $getchar\_unlocked()$. Проблем обаче е, това може да не се компилира ако използваме *codeblocks* с *mingw*. Все пак ако има система може да го използваме и да очакваме обратна връзка.

```
long long readLong() {
    char c = getchar_unlocked();
    while (!isdigit(c)) {
        c = getchar_unlocked();
    }

    long long ans = 0;
    while (isdigit(c)) {
        ans = 10*ans+c-'0';
```

```
        c = getchar_unlocked();
    }
    return ans;
}
```

□

5 Задачи

Задача 9. *Летен Турнир 2019, C група, C1. Популярен рейтинг*

Задача 10. *НОИ 3 2018, D група, D2. Болшинство*

Задача 11. *Есенен Турнир 2007, C група, C3. Имена*

Задача 12. *НОИ 3 2019, C група, C4. Липсващи числа*

Задача 13. *НОИ 2 2011, A група, A3. Подслушване*