АиСД_дз2

CovarianceMomentum

1term

1

```
increment():
    i = 0
    while i < k and a[i] == 1:
        a[i] = 0
        i++
    if i < k:
        a[i] = 1
```

```
decrement():
    i = 0
    while i < k and a[i] == 0:
        a[i] = 1
        i++
    if i < k:
        a[i] = 0
```

```
get(i):
    return a[i]
```

```
setZero():
    i = 0
    while i < k:
        a[i] = 0
        i++
```

Предварительно положим, что за единицу времени выполняются операции сравнения, присваивания и арифметические действия.

a) Как нетрудно заметить, если весь массив a[] заполнен единицами, то истинное время работы increment будет равняться $1 + 4 \cdot k$. Поймём, что в случае, если не весь массив заполнен единицами, а только первые $m < k$ позиций, то время работы будет равно $1 + 4 \cdot m + 2 < 1 + 4 \cdot k$.

b)

Лемма 1.1. *В арифметических преобразованиях в этой задаче мы будем пользоваться следующим фактом:*

$$\sum_{j=0}^{n} j \cdot 2^j \ = \ 2^{n+1} \cdot (n-1) + 2$$

Поймём, что вариантов, когда первые $m < k$ элементов массива заполнены единицами и $\mathrm{a}[m] = 0$ будет ровно 2^{k-m-1}, так как способов заполнить каждую из оставшихся позиций ровно 2. Таким образом, суммарное время работы increment на всех возможных входных данных будет равняться:

$$\left(\sum_{j=0}^{k-1} 2^{k-j-1} \cdot (3 + 4 \cdot j) \right) + 1 + 4 \cdot k \ =$$
$$= \ 3 \cdot (2^k - 1) + 4 \cdot \sum_{i=0}^{k-1} (2^i \cdot ((k-1) - i)) + (1 + 4 \cdot k) \ =$$
$$= \ (3 + 4 \cdot k - 4) \cdot (2^k - 1) - 4 \cdot \sum_{i=0}^{k-1} 2^i \cdot i + (1 + 4 \cdot k) \ =$$
$$= \ (4 \cdot k - 1) \cdot (2^k - 1) - 4 \cdot (2^k \cdot (k-2) + 2) + (1 + 4 \cdot k) \ =$$
$$= \ 7 \cdot 2^k - 6$$

Так как всего вариантов входных данных 2^k, то среднее время работы increment равняется $7 - \dfrac{6}{2^k} = 7$.

c) Рассмотрим массив, весь заполненный единицами, кроме последнего элемента. Будем поочередно применять к нему операции increment и decrement. Тогда массив будет менять состояния между вышеописанным и всеми нулями с последним элементом единицей. increment и decrement в таком случае будут работать за $1 + 4 \cdot k$. Теперь очевидно, что n операций будут выполняться за $\Omega(nk)$. Осталось показать, что они не могут выполняться за большее время. Поймём, что время выполнения decrement на массиве a[] равно времени выполнения increment на инвертированном массиве (каждый элемент $\mathrm{a}[i] = 1 - \mathrm{a}[i]$), таким образом получаем, что decrement в худшем случае не может работать дольше, чем increment в худшем случае. Таким образом, показали, что рассмотренный случай действительно наихудший, $\Rightarrow$ время выполнения этих операций в худшем случае равно $\Theta(nk)$.

d) В дополнение к уже имеющимся переменным заведем ещё одну — номер самого старшего ненулевого бита нашего числа. Тогда setZero юудет просто обнулять эту переменную, increment проходить по элементам массива строго ДО элемента с таким индексом, а get() будет возвращать ноль, если указывает за старший бит. Ниже приведён псевдокод этих функций.

Весьма очевидно, что они все работают за $\mathcal{O}(1)$.

```
increment():
    i = 0
    while i < top_border and a[i] == 1:
        a[i] = 0
        i++
    if i < top_border:
        a[i] = 1
    if i == top_border and top_border < k:
        a[i] = 1
        top_border++
```

```
get(i):
    if i < top_border:
        return a[i]
    else:
        return 0
```

```
setZero():
    top_border = 0
```

2

Назовём cap длину стека, n — количество элементов в нём. Тогда определим потенциал так:

$$\begin{cases} \Phi(n, cap) = \dfrac{cap}{2} - n, & n \leq \dfrac{cap}{2} \\ \Phi(n, cap) = 2 \cdot \left(n - \dfrac{cap}{2}\right), & n \geq \dfrac{cap}{2} \end{cases}$$

Покажем, что этот потенциал подходит. Заведём следующие операции:

- lesser_push — push, когда в массиве меньше $\dfrac{cap}{2}$ элементов. Стоимость операции $a_{lps} = 1 + \Phi(n, cap) - \Phi(n-1, cap) = 1 + \dfrac{cap}{2} - n - \dfrac{cap}{2} + n - 1 = 0$.

- greater_push — push, когда в массиве больше или равно $\dfrac{cap}{2}$ элементов. Стоимость операции $a_{gps} = 1 + \Phi(n, cap) - \Phi(n-1, cap) = 1 + 2 \cdot \left(n - \dfrac{cap}{2} - n + 1 + \dfrac{cap}{2}\right) = 3$.

- lesser_pop — pop, когда в массиве меньше или равно $\dfrac{cap}{2}$ элементов. Стоимость операции $a_{lpp} = 1 + \Phi(n, cap) - \Phi(n+1, cap) = 1 + \dfrac{cap}{2} - n - \dfrac{cap}{2} + n + 1 = 2$.

- greater_pop — pop, когда в массиве больше или равно $\dfrac{cap}{2}$ элементов. Стоимость операции $a_{gpp} = 1 + 2 \cdot \left(n - \dfrac{cap}{2} - n - 1 + \dfrac{cap}{2}\right) = -1$.

- lesser_move — копирование массива при сужении, то есть когда $n = \dfrac{cap}{4}$. Стоимость операции $a_{lm} = n + \Phi(n, \dfrac{cap}{2}) - \Phi(n, cap) = \dfrac{cap}{4} + 0 - \dfrac{cap}{2} + \dfrac{cap}{4} = 0$.

- greater_move — копирование массива при расширении, то есть когда $n = cap$. Стоимость операции $a_{gm} = n + \Phi(n, 2 \cdot cap) - \Phi(n, cap) = cap + 0 - 2 \cdot (cap - \dfrac{cap}{2}) = cap + 0 - cap = 0$

Как несложно заметить, в среднем операции работают за линейное время.

3

Идея решения такова — вместе с оригинальным (original) массивом инициализируем два массива такой же длины — iter и filled и будем поддерживать количество уже инициализированных элементов cnt. Когда мы делаем set какого-либо элемента original, который мы ещё не трогали, в ячейку с таким же номером в iter кладём cnt, а в ячейку filled с номером cnt кладём позицию самого элемента и увеличиваем cnt.

Теперь посмотрим, как мы будем проверять, инициализирован ли элемент на позиции i. Если да, то ячейка массива iter, на которую указывает filled[i], лежит в пределе уже заполненных и должна указывать на i. Почему такого не может случиться, если элемент неинициализирован, очевидно — первые cnt элементов массива filled указывают только на уже инициализированные элементы. Ниже приведён код, реализующий эту структуру данных.

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

struct setGetStruct{
    int* original;
    int* iter;
    int* filled;
    int cap;
    int cnt;
}; typedef setGetStruct sgs;

void sgsCreate(sgs* a, int n)
{
    a->original = (int*)malloc(n*sizeof(int));
    a->iter = (int*)malloc(n*sizeof(int));
    a->filled = (int*)malloc(n*sizeof(int));
    a->cap = n;
    a->cnt = 0;
}

int sgsGet(sgs* a, int i)
{
    if ((a->iter)[i] >= a->cnt || (a->iter)[i] < 0 || (a->filled)[(a->iter)[i]] != i) return
     0;
    return (a->original)[i];
}

void sgsSet(sgs* a, int i, int x)
{
    (a->original)[i] = x;
    if (a->cnt < a->cap && ((a->iter)[i] >= a->cnt || (a->iter)[i] < 0 || (a->filled)[(a->
    iter)[i]] != i)) {
        (a->iter)[i] = a->cnt;
        (a->filled)[a->cnt] = i;
        a->cnt++;
    }
}

int main() {
    int n;
    cin >>n;
```

```cpp
    sgs arr;
    sgsCreate(&arr, n);
    int choice = 0;
    while (choice == 0 || choice == 1) {
        cout <<"0␣to␣set(i,␣x);␣␣␣1␣to␣get(i);␣␣anything␣else␣to␣break\n";
        cin >>choice;
        if (choice == 0) {
            int i, x;
            cin >>i >>x;
            sgsSet(&arr, i, x);
        }
        if (choice == 1) {
            int i;
            cin >>i;
            cout <<sgsGet(&arr, i) <<endl;
        }
    }
    return 0;
}
```

3.cpp

4

Положим, что количество блоков, данных нам, равно N. Разобьём процесс запросов памяти на два этапа — до того, как мы выдали суммарно N блоков памяти и после. До того, как мы выдали N кусков памяти, будем просто выдавать их по порядку. После этого будем выдавать их из очереди, которая будет формироваться в процессе возвращения нам блоков памяти. Ниже приведен код, схематически показывающий этот процесс.

Так как операции добавления и изъятия из очереди выполняются за $\mathcal{O}(1)$, то и весь алгоритм будет выполняться за $\mathcal{O}(1)$.

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

int main() {
    queue<int> st;
    int n;
    cin >>n;
    int cnt = 0;
    int choice = 0;
    while (choice == 0 || choice == 1) {
        cout <<"0␣to␣malloc();␣␣␣1␣to␣free(x);␣␣anything␣else␣to␣break\n";
        cin >>choice;
        if (choice == 0) {
            if (cnt < n) {
                cout <<"You␣get␣block␣number␣" <<cnt <<"\n";
                cnt++;
            } else {
                if (st.size() > 0) {
                    cout <<"You␣get␣block␣number␣" <<st.front() <<"\n";
                    st.pop();
                } else {
                    cout <<"Not␣enough␣memory.\n";
                }
            }
        }
        if (choice == 1) {
            int x;
            cin >>x;
            st.push(x);
        }
    }
    return 0;
}
```

4.cpp

5

Опишем алгоритм решения. Для начала заметим, что всего ответов не может быть больше, чем $k-1$, так как в противном случае общее число их вхождений в массив превышает $k \cdot \frac{n}{k} = n$. Создадим массив для кандидатов размером $k-1$, каждый элемент которого содержит значение кандидата и встреченное нами его количество, изначально нулевое. Пойдем по массиву, считывая a$[i]$. Выполним следующее:

1. Если элемент с таким значением уже есть среди кандидатов, увеличим его счётчик на 1.
2. Если такого элемента нет, но есть элемент с нулевым счётчиком, то заменим его значение на значение a$[i]$ и выставим счётчик в 1.
3. Если ни один из элементов не равен a$[i]$ и все они имеют счётчик хотя бы 1, то мы нашли группу из k различных элементов. Вычтем из каждого счётчика по 1.

В конце исполнения данного алгоритма в массиве кандидатов останутся только ответы, и, быть может, какие-то лишние элементы. Для каждого элемента массива кандидатов проверим, является ли он ответом, пройдясь по a[].

Покажем, что в результате мы не могли потерять ответ. Так как каждый раз мы удаляли из массива кандидатов ровно k различных элементов, то всего удалений не могло быть больше, чем $\frac{n}{k}$. Таким образом, если элемент встречается больше, чем $\frac{n}{k}$ раз, то он остался в массиве кандидатов с ненулевым счётчиком. Лишний ответ мы дать не могли, так как проверили их все.

Ниже приведён код, реализующий указанный алгоритм за $\mathcal{O}(nk)$ времени и $\mathcal{O}(n+k)$ памяти.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, k;
    cin >>n >>k;
    int arr[n];
    for (int i = 0; i < n; i++) cin >>arr[i];
    pair<int, int> candidates[k-1];
    for (int j = 0; j < k - 1; j++) candidates[j] = make_pair(0, 0);
    for (int i = 0; i < n; i++) {
        bool found_identical = false;
        for (int j = 0; j < k - 1; j++) {
            if (candidates[j].first == arr[i]) {
                found_identical = true;
                candidates[j].second++;
                break;
            }
        }
        bool found_zero = false;
        if (!found_identical) {
            for (int j = 0; j < k - 1; j++) {
                if (candidates[j].second == 0) {
                    found_zero = true;
                    candidates[j].first = arr[i];
                    candidates[j].second = 1;
                    break;
                }
            }
        }
        if (!found_identical && !found_zero) {
            for (int j = 0; j < k - 1; j++) {
                candidates[j].second--;
            }
        }
    }
    for (int j = 0; j < k - 1; j++) {
        if (candidates[j].second > 0) {
            int cnt = 0;
            for (int i = 0; i < n; i++) {
                if (candidates[j].first == arr[i]) {
                    cnt++;
                }
            }
            if (cnt > n/k) {
                cout <<candidates[j].first <<' ';
            }
        }
    }
    return 0;
}
```

5.cpp

6

Для начала покажем, что среднее время работы операции contains равняется $\mathcal{O}(k^2)$. Это очевидно следует из того, что применение этой операции не меняет потенциал, а сама она проходит по не больше чем k массивам, на i-ом тратя $\mathcal{O}(i)$ времени. Просуммировав по всем i, получаем, что $T = \sum\limits_{j=0}^{k} \mathcal{O}(j)$, то есть $\exists A:\ T \le A \cdot \sum\limits_{j=0}^{k} j = A \cdot \dfrac{k(k+1)}{2} = \mathcal{O}(k^2)$.

Теперь разберёмся с операцией add. Посчитаем её среднее время работы руками. Представим число n в двоичной форме. Заметим, что каждой единице в его записи соотвтетствует полный массив, а каждому нулю — пустой. Таким образом, функция add проведет операцию merge для каждого из полных массивов до первого пустого, то есть для каждой единицы до первого нуля. Таким образом, если в числе первые m чисел — единицы, то add проведет $\sum\limits_{j=0}^{m} \mathcal{O}(2^j)$ операций, то есть $\mathcal{O}(2^{m+1})$ (там появляется операция, занимающая 2^0 времени — это присвоение пустому массиву ссылки на смёрженный новый массив).

Посчитаем, сколько всего есть возможных входных данных, на которых в начале стоит ровно m единиц, а после — ноль. Поймём, что для $m < k$ таких будет ровно 2^{k-m-1}, а для $k = m$ — одна. Таким образом, общее время работы алгоритма на всех входах будет равно:

$$\sum_{j=0}^{k-1} \left(2^{k-j-1} \cdot \mathcal{O}(2^{j+1})\right) + 1 \cdot \mathcal{O}(2^{k+1}) \ =$$

$$= \ \sum_{j=0}^{k} \mathcal{O}(2^k) \ = \ \mathcal{O}(k \cdot 2^k)$$

Отсюда имеем, так как всего вариантов входных данных в точности 2^k, что среднее время работы add будет равно $\dfrac{\mathcal{O}(k \cdot 2^k)}{2^k} = \mathcal{O}(k) = \mathcal{O}(\log n)$.