Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО»

Факультет Программной инженерии и компьютерной техники

**Отчёт по задачам блока №1**

**По дисциплине «Алгоритмы и структуры данных»**

Группа: P3212

Выполнил: Балин А. А.

Проверил: Косяков М. С.

# Задача A

## Решение

Решение сводится к поиску наибольшего подмножества идущих подряд чисел, в котором нет 3 идущих подряд одинаковых значений. Будем решать задачу с помощью двух численных указателей: один из них будет указывать на начало текущего подмножества, второй на его конец. Как только мы встретим недопустимую комбинацию, текущее подмножество становится плохим, поэтому мы сравниваем его предыдущее состояние, которые было хорошим. Сохраняем либо длину и указатели для этого множества, если его длина больше, чем длина какого-то другого, ранее найденного множества, либо оставляем всё как есть. А чтобы продолжить поиск дальше, сдвигаем первый указатель на место второго за минусом одного. Выделяя указатели жирным, рассмотрим пример: **a**bc…xx**x**. При сохранении учитываем, что нужно взять указатель справа на 1 меньше: **a**bc…x**x**x. Переход к поиску следующей последовательности: abc…x**xx** – таким образом, мы перебрали всевозможные последовательности без запрещённых комбинаций и нашли среди них последовательность с максимальной длиной.

Одна итерация по всему массиву длиной n, менее 10 вспомогательных действий с указателями/проверкой условий для каждого элемента, поэтому можно ограничить f(N) как N снизу и 10N сверху => f(N) = Θ(N).

## Код

#include <iostream>

#include <vector>

using namespace std;

int main() {

//считываем

int N;

cin >> N;

vector<int> row(N);

for (int i = 0; i < N; ++i) {

cin >> row[i];

}

//логика

int left = 0, right = 0;

int left\_out = 0, right\_out = 0;

int last = row[0];

int repeat = 1;

int max = 1;

for (int i = 1; i < N; i++) {

if (row[i] == last) {

repeat++;

}

else {

repeat = 1;

last = row[i];

}

if (repeat < 3) {

right++;

}

else {

int len = right - left + 1;

if (len > max) {

max = len;

left\_out = left;

right\_out = right;

}

left = i - 1;

right = i;

}

}

if (right - left + 1 > max) {

left\_out = left;

right\_out = right;

}

//вывод

cout << ++left\_out << ' ' << ++right\_out;

return 0;

}

# Задача B

## Решение

Ход решения отчасти основан на решении правильных скобочных последовательностей с помощью стека, однако, при добавлении и удалении элементов нам следует учитывать не только upper/lower-case букв, но и их значение. Также будет метод отличается тем, что при отсутствии в стеке нужной буквы, мы не будем сразу выдавать ответ «Impossible», потому что, в отличие от скобочных последовательностей, ловушка и само животное могут идти в любом порядке. А условие, что пути не пересекаются, однозначно можно сопоставить отсутствию на пути животных, еще не пойманных в клетку, что будет означать, что в конце стек должен оказаться пустым. Осталось определить порядок поимки животных в соответствии с номерами ловушек. Для этого мы создаём стек и счётчик для ловушек и для зверей: если мы находим животное/ловушку, кладем значение счётчика на стек и увеличиваем сам счётчик. Если мы видим, что животное ловится в ловушку на какой-то итерации, то есть последнее значение буквы в общем стеке соответствует ловушке для зверя/зверю для ловушки, это значит, что на предыдущей итерации мы положили в стек индекс животного/ловушки и оно соотносится (т. к. мы «ловим») с текущей буквой, поэтому мы сохраняем пару значений индексов, например, для комбинации “Aa” мы сохраняем индекс ловушки A, взятой из стека, а так же текущий счётчик животных, увеличенный на 1. Таким образом, у нас получаются правильные пары индексов Ловушка-Животное, которое удобно хранить в массиве.

Аналогично предыдущей задаче (с чуть более большим коэф. с2 >10) f(N) = Θ(N).

## Код

#include <iostream>

#include <stack>

#include <vector>

using namespace std;

int main() {

stack<char> current;

string animals;

cin >> animals;

vector<int> result(animals.size() / 2);

int s = 0, b = 0;

stack<int> small, big;

for (size\_t i = 0; i < animals.size(); i++) {

if (islower(animals[i])) {

small.push(s);

s++;

if (current.size() > 0) {

if (isupper(current.top()) &&

char(tolower(current.top()) == animals[i])) {

result[big.top()] = small.top() + 1;

big.pop();

small.pop();

current.pop();

}

else {

current.push(animals[i]);

}

}

else {

current.push(animals[i]);

}

}

else {

big.push(b);

b++;

if (current.size() > 0) {

if (islower(current.top()) &&

char(toupper(current.top())) == animals[i]) {

result[big.top()] = small.top() + 1;

big.pop();

small.pop();

current.pop();

}

else {

current.push(animals[i]);

}

}

else {

current.push(animals[i]);

}

}

}

if (current.size() != 0) {

cout << "Impossible";

}

else {

cout << "Possible\n";

for (int i = 0; i < result.size(); i++) {

cout << result[i] << " ";

}

}

return 0;

}

# Задача C

## Решение

Решение заключается в том, что для каждого блока у нас есть список элементов и их значений, а если требуется значение из блока уровня выше, мы ищем самый первый, поднимаясь по уровням вверх (а не самый верхний). Идейно задача довольно простая, осталось понять, в какой комбинации структур это будет работать достаточно быстро, чтобы пройти тесты. Моя первая реализация была в одном стеке, при окончании блока я делал pop() стека, пока не встречу “{”, что провалилось на ~40 тесте, поэтому я решил сделать вектор мап, чтобы удалять блок при закрытии одним удалением мапы из вектора, что тоже работало не сильно быстрее. Был еще один вариант с двумя векторами, как в первом варианте, только 1 вектор для имён переменных, а другой для их значений, но закрытие блока, видимо, вызывало слишком долгий процесс удаления элементов до первого «{». И последнее решение заключалось в том, чтобы хранить в мапе имя переменной и стек с её значениями на разных уровнях блока (первый элемент – самый высокий уровень), а вектор с переменными в текущем блоке: таким образом, пройдя по нему, можно удалить по 1 элементу из стека в тех переменных, значения которых сохранялись в текущем блоке. Этот вариант обладает довольно быстрым удалением элементов, а также очень быстрым поиском, в отличие от моих предыдущих реализаций: взять стек по имени переменной и его значение на вершине.

## Код

#include <iostream>

#include <map>

#include <stack>

#include <vector>

using namespace std;

int main() {

map<string, stack<string>> values;

stack<vector<string>> cur\_old;

vector<string> current;

string buf;

while (cin >> buf) {

if (buf == "{") {

cur\_old.push(current);

current.clear();

}

else if (buf == "}") {

for (int i = 0; i < current.size(); i++) {

auto curr\_stack = values.find(current[i]);

curr\_stack->second.pop();

}

if (cur\_old.size() != 0) {

current = cur\_old.top();

cur\_old.pop();

}

}

else {

size\_t position = buf.find\_first\_of("=");

string p = buf.substr(0, position);

string n = buf.substr(position + 1);

current.push\_back(p);

if (values.find(p) == values.end()) {

stack<string> temp;

values[p] = temp;

}

if (isdigit(n[0]) || n[0] == '-') {

values[p].push(n);

}

else {

if (values.find(n) != values.end()) {

if (values[n].size() > 0) {

values[p].push(values[n].top());

cout << values[n].top() << '\n';

}

else {

values[p].push("0");

cout << "0\n";

}

}

else {

values[p].push("0");

cout << "0\n";

}

}

}

}

return 0;

}

# Задача D

## Решение

Идея состоит в том, чтобы предугадать ответ, не вычисляя все итерации. Первая идея состоит в том, что так как максимум в конце дня остаётся d бактерий, при условии d\*b <c мы имеем, что эксперимент закончится (хотя в таком случае перебор бы был по 2 дням, он остановился бы на 2 дне). Вторая идея заключается в том, что, если несколько дней подряд у нас остаётся одинаковое количество бактерий <= d, с учётом того, что день ото дня алгоритм не меняется => число к концу дня тоже, можем сделать вывод, что оно останется так до последнего дня, не выполняя итерации все k раз. Остаётся вариант, что количество бактерий в конце дня чередуется с какой-то периодичностью t, тогда есть возможность сохранять эти значения в массив и искать повторяющиеся подпоследовательности в нём, что при довольном большом t может занять много времени, собственно, поэтому этот вариант оптимизации не был рассмотрен. Довольно очевидно, что f(N) = o(1), так как при ряде условий алгоритм выполняется за константу, а также f(N) = O(N), что верно по той причине, что при несрабатывании оптимизаций, алгоритм пройдёт все N = k раз.

## Код

#include <iostream>

using namespace std;

#define long\_t unsigned long long

int main() {

int a, b, c, d;

cin >> a;

cin >> b, cin >> c;

cin >> d;

long\_t k;

cin >> k;

if (d \* b < c) {

cout << '0';

return 0;

}

short x = 0;

int last = -1;

for (long\_t i = 0; i < k; i++) {

if (a == last) {

x++;

if (x == 2) {

cout << a;

return 0;

}

}

else {

last = a;

x = 0;

}

a \*= b;

if (a < c) {

a = 0;

break;

}

a = min(a - c, d);

}

cout << a;

return 0;

}