ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«Национальный исследовательский университет ИТМО»

Факультет программной инженерии и компьютерной техники

«Алгоритмы и структуры данных»

отчет по блоку задач №1 (Яндекс.Контест)

Выполнил:

Студент группы P3215

Барсуков М.А.

Преподаватель:

Косяков М.С.

Санкт-Петербург, 2024г.

Задачи из 1-го блока Яндекс.Контеста

Задача №1 «A. Агроном-любитель»

Описание:

Лёша вырастил *n* цветов в длинной грядке. Нужно найти самый длинный участок, где нет трех одинаковых цветов подряд.

Доказательство:

Для этого рассматриваем текущий, предыдущий и предпредыдущий цветки, чтобы понять, когда встретятся 3 подряд одинаковых цветка (когда все три рассматриваемых цветка одинаковые). Изначально мы воспринимаем текущим началом самого длинного отрезка первый цветок. Когда встречаем три одинаковых – берем за текущее начало локального длинного участка предыдущий цветок (так, чтобы начало нового длинного участка составляли два одинаковых цветка – то есть в участке нет трех одинаковых цветков, но и длина благодаря этому максимальная). Если текущая длина участка больше длины предыдущего участка, считаем текущее начало участка началом глобально длиннейшего участка, конец глобального участка – текущим цветком, а максимальную длину – текущей длинной. Таким образом, обработав все цветки, мы отслеживаем максимальную длину участка и его границы на протяжении всего процесса, значит, алгоритм гарантирует, что в любом текущем отслеживаемом участке не будет трех одинаковых цветов подряд, а к концу выполнения будет найден глобальный максимальный участок, удовлетворяющий условиям.

Алгоритмическая сложность:

1. **По времени:** Алгоритм обрабатывает каждый элемент последовательности один раз и определяется количеством элементов (цветков) n, которые нужно обработать, что дает линейную временную сложность в среднем и худшем случаях – O(n).
2. **По памяти:** Программа использует константное количество дополнительной памяти для хранения переменных (n, d_start, d_end, max_length, cur_length, cur_start, cur_value, last_value, pre_last_value, i). Это не зависит от размера входных данных, поэтому сложность по памяти O(1).

Итого:

- По времени: O(n) (линейная сложность).
- По памяти: O(1) (константная сложность).

Код:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    uint64_t n;
    cin >> n;

    uint64_t  d_start = 1,
              d_end = 1,
              max_length = 0,
              cur_length,
              cur_start = 1;

    int64_t cur_value, last_value = -1, pre_last_value = -2;

    for (uint64_t i = 1; i <= n; ++i) {
        cin >> cur_value;

        if (cur_value == last_value && cur_value == pre_last_value && i > 2) {
            cur_start = i - 1;
        }

        cur_length = i - cur_start + 1;
        if (cur_length > max_length) {
            d_start = cur_start;
            d_end = i;

            max_length = cur_length;
        }

        pre_last_value = last_value;
        last_value = cur_value;
    }

    cout << d_start << " " << d_end << endl;

    return 0;
}
```

Задача №2 «B. Зоопарк Глеба»

Описание:

Животные и их ловушки соответствуют парным символам разного регистра, нужно определить, можно ли провести непересекающиеся дуги внутри круга и восстановить ответ.

Доказательство:

Будем делать это через стеки (для индексов животных, для индексов ловушек и для всех символов). Индексы животных для ответа (т.е. соответствующие ловушкам в порядке обхода) будем хранить в словаре ловушка-животное. Стек используется для отслеживания текущего состояния последовательности, обеспечивая правильное сопоставление животных и ловушек.

Встречая новый символ и определяя, ловушка это или животное, мы кладем его индекс по порядку обхода (ловушка или животного) в соответствующий стек. Если текущий стек символов пуст (= все пары до этого установлены и их нету в стеках), то добавляем туда секущий символ. Иначе, если стек символов не пустой:

- Если текущий символ — это пара для верхнего символа со стека символов, то мы находим соответствие, а значит добавляем пару ловушка-животное, которые мы берем из стека для животных и стека для ловушек, так как очевидно, что текущий символ и верхний на стеке символов соответствуют верхним на стеке животного или ловушки в словарь ответов (если текущий символ нижнего регистра – то он, очевидно верхний на стеке животных, если верхний регистр – то он верхний на стеке ловушек, для верхнего на стеке символов соответственно – главное они составляют пару), а также удаляем верхние элементы стеков животных, ловушек и символов.
- Если текущий символ — не пара для верхнего символа со стека символов, то просто кладем его в стек символов, для него пара должна найтись позже или не найтись, если не пересекающую хорду найти невозможно.

В конце все пары, для которых можно провести хорды, исчезнут из стека всех символов. Если в этом стеке что-то осталось, то значит, что провести хорды в круге нельзя и решения нет. Если стек пуст, то есть решение и нужно вывести индексы животных из словаря, то есть решение восстанавливается тоже при помощи стеков и словаря ответов.

После обработки каждого символа стек либо пуст, либо содержит последовательность животных, для которых ещё не найдены соответствующие ловушки. Каждая ловушка проверяется на соответствие с верхним элементом стека. Это обеспечивает, что животное будет помещено в правильную ловушку. Таким образом, алгоритм сопоставляет животное и ловушку в независимости от их расположения и обеспечивает добавление пары соответствий так, что пути от животных к ловушкам не будут пересекаться – иначе для него просто не будет найдено такой пары и в стеке символов останутся символы таких пар, а значит, алгоритм выдаст Impossible.

Алгоритмическая сложность:

1. **По времени:** Алгоритм проходит по входной строке один раз, что составляет $O(n)$, где n — длина строки. Операции с каждым символом – добавление в стек или удаление из него – занимают константное время.
2. **По памяти:** Используются стеки и словарь для хранения индексов и символов, что в худшем случае требует $O(n)$ памяти для n символов.

Итого:

- По времени: $O(n)$ (линейная сложность).
- По памяти: $O(n)$ (линейная сложность).

Код:

```cpp
#include <bits/stdc++.h>

using namespace std;

bool different_cases(char a, char b) {
    return toupper(a) == toupper(b) && a != b;
}

int main() {
    stack<char> s;
    stack<int> animals;
    stack<int> traps;
    map<int, int> animal_indexes;
    int animal_count = 0,
        trap_count = 0;

    char cur_char;
    while(cin.get(cur_char)) {
        if (cur_char == '\n') break;

        if (isupper(cur_char)) {
            trap_count++;
            traps.push(trap_count);
        } else {
            animal_count++;
            animals.push(animal_count);
        }

        if (s.empty() || !different_cases(cur_char, s.top())) {
            s.push(cur_char);
        } else {
            animal_indexes[traps.top()] = animals.top();

            animals.pop();
            traps.pop();
            s.pop();
        }
    }

    if (!s.empty()) {
        cout << "Impossible\n";
        return 0;
    }
    cout << "Possible\n";
    for (const auto& [_trap, animal_index] : animal_indexes) {
        cout << animal_index << " ";
    }

    return 0;
}
```

Задача №3 «C. Конфигурационный файл»

Описание:

Нужно уметь определять значение переменной в разных блоках (на уровнях вложенности). Чтобы удобно хранить информацию, будем использовать словарь, где ключ – это имя переменной, а значение – стек значений (чисел), которые когда-либо присваивались этой переменной, а также стек из множеств (для этого здесь оптимально использовать queue) назначений переменных (состоящий из имен) чтобы знать переменные на разных уровнях блоков. При встрече выражения, где значение одной переменной присваивается другой, нужно вывести это значение.

Доказательство:

Будем последователь обрабатывать (интерпретировать) конфигурацию строку за строкой.

При открытии блока ({) мы инициализируем новое множество назначений переменных в стеке.

Когда мы встречаем присваивание и это число, мы устанавливаем новое значение на это число, если это другая переменная – мы берем верхнее значение (текущая область, то есть последнее назначенное) со стека значений словаря конфига и выводим его. После этого мы добавляем имя переменной в верхнее множество (то есть для текущего блока) стека назначений переменных, а также в конфиге по имени этой переменной мы пушим новое значение переменной.

При закрытии блока нужно очистить переменные текущего уровня. Для этого для всех значений верхнего (то есть последнего открытого) множества стека переменных мы убираем верхнее значение стека значений переменных в словаре конфига для этой переменной и убираем со стека переменных это множество. То есть соответственно всем назначениям этого блока (которые хранятся во множестве на вершине стека) мы удаляем такое же количество назначений из конфига, оставляя наверху стека конфига значение, назначенное до входа в блок.

Для лучшего понимания структуры данных в данном алгоритме, рассмотрим пример:

```
1. a=1
2. a=2
3. b=a
4. {
5. b=3
6. a=b
7. }
8. a=b
9. b=4
```

В строчках 1-2 при назначении мы пушим на стек конфига для *a* значения 1 и 2. В строке 3 для *b* пушим верхнее значение со стека для *a*, то есть 2. Сейчас конфиг содержит { a: [1,2], b: [2] }. Верхнее множество стека переменных: [a, a, b]. Открываем новый блок, теперь стек: [[a,a,b], **[]**]. После строк 5 и 6 структуры данных имеют вид: { a: [1,2,3], b: [2,3] } и [[a,a,b], **[b,a]**]. Закрываем блок и удаляем значения, присвоенные за время существования блока следующим образом: пока верхнее множество стека (выделено жирным) не содержит пустое множество, мы удаляем последнее значение стека из конфига, основываясь на имени переменной в стеке назначений переменных (сначала для a удаляем 3, потом для b удаляем 3), после чего удаляем верхнее значение (имя переменной) со стека назначений переменных – той, чье значение мы удалили перед этим. Когда мы очистим все назначения из этого блока, мы удаляем верхнее множество со стека назначений переменных, соответствующее данной области видимости. После этого значения словаря конфига снова будут { a: [1,2], b: [2] }. Поэтому в строках 8-9 мы используем

значения, установленные для блока. К концу выполнения структуры данных будут содержать { a: [1,2,2], b: [2,4] } и [[a,a,b,a,b]].

Алгоритмическая сложность:

1. **По времени:** Количество выполняемых циклов равно количеству строк конфига *N*. Если это операция закрытия блока, то она выполняется за O(K), где K – это количество назначений в закрытом блоке. Открытие блока осуществляется за O(1), назначение переменной – за O(log(n)) – при добавлении значения в словарь.
2. **По памяти:** В худшем случае в стеке словаря будет храниться L (все назначения за время работы программы, если не было блоков кроме внешнего) значений, а в стеке назначений в таком случае L значений будет только если все эти назначения – разные (то есть [[a,b,c,d,e,…]] в стеке и {a: [1], b: [1], …} в словаре). В таком случае сложность по памяти будет O(L).

Итого:

- По времени: O(K)
- По памяти: O(L)

Код:

```cpp
#include <bits/stdc++.h>
using namespace std;
pair<string, string> parse_assignment(string s) {
    size_t pos = s.find('=');
    pair<string, string> result(s.substr(0, pos), s.substr(pos + 1, s.size() - pos));
    return result;
}
inline bool is_number(const string& s) {
    return isdigit(s[0]) || s[0] == '-';
}
int get_val_by_var(map<string, stack<int>> &config, const string& variable) {
    if (!config.count(variable) || config[variable].empty()) {
        return 0;
    }
    return config[variable].top();
}
inline void clean_current_level(map<string, stack<int>> &config, stack<queue<string>> &context_vars) {
    while (!context_vars.top().empty()) {
        config[context_vars.top().front()].pop();
        context_vars.top().pop();
    }
    context_vars.pop();
}
int main() {
    cin.tie(0);
    cout.tie(0);
    map<string, stack<int>> config;
    stack<queue<string>> context_vars;
    context_vars.emplace();
    string current_line;
    while (getline(cin, current_line)) {
        if (current_line == "{") {
            context_vars.emplace();
        } else if (current_line == "}") {
            clean_current_level(config, context_vars);
        } else {
            const auto& [variable, value] = parse_assignment(current_line);
            int assigned_value;
            if (is_number(value)) {
                assigned_value = stoi(value);
            } else {
                assigned_value = get_val_by_var(config, value);
                cout << assigned_value << endl;
            }
            context_vars.top().push(variable);
            config[variable].push(assigned_value);
        }
    }
    return 0;
}
```

Задача №4 «D. Профессор Хаос»

Описание:

Каждый день количество бактерий умножается на *b* и уменьшается на *c*. Если после этого остаётся меньше *c* бактерий, эксперимент завершается. Если количество бактерий превышает *d*, оно ограничивается этим значением.

Доказательство:

Наивный алгоритм с некоторой оптимизацией: алгоритм воспроизводит вычисления, описанные выше до того момента, когда количество бактерий на начнёт повторяться.

Очевидно, что наивный алгоритм с повторением описанных в условиях вычислений работает корректно, учитывая простоту алгоритма и заданные границы для *a, b, c, d, k*. Кроме того, очевидно, что он точно завершит работу после *k* повторений тела цикла (так как требуется найти количество бактерий на *k*-тый день, то есть повторить вычисления *k* раз).

Докажем корректность завершения цикла вычислений в случае, если текущее число бактерий на конец дня совпадает с числом бактерий в предыдущий день.

```
1. a = a * b - c;
2. if (a > d) a = d;
```

Пусть *a'* – количество бактерий, которое получилось после применения этих вычислений на очередной день, а *a* – число бактерий до вычислений (то есть на предыдущий день). Очевидно, что тогда *a'* это или *d*, или *a*b – c*. Если это *d*, это значит, что с предыдущим значением *a* результат вычислений превысил *d*, а если учесть, что *a' = a*, то и на следующую итерацию он превысит *d* и снова *a'* будет равно *d*, то есть результат вычислений не изменится в будущих итерациях и цикл можно прервать. Во втором же случае, если *a'* это *a*b – c*, то очевидно, что при *a' = a*b – c* и *a' = a*, результат опять-таки не изменится, сколько раз не повторяй эти вычисления, так как они сводятся к *a = a*b – c*. Значит, *a'* и будет значением числа бактерий в любой последующий день (то есть и на *k*-тый день), следовательно, *a'* и есть ответ, и выполнение цикла можно прервать.

Алгоритмическая сложность:

1. **По времени:**
   - **Средний и худший случай (Θ):** Обычно средняя сложность будет близка к худшему случаю, так как в данном алгоритме мы не можем уменьшить количество итераций без информации о том, когда значение a начнет совпадать с предыдущим. Поэтому, так как число итераций зависит от дня *k*, на который мы должны узнать количество бактерий, а узнать, когда число бактерий начнет повторяться без выполнения хотя бы нескольких итераций мы не можем, итерации зависят от k, и временная сложность будет O(k).
2. **По памяти:**
   - Программа использует фиксированное количество переменных (a, b, c, d, k, last_value). Оно не зависит от размера входных данных, поэтому сложность по памяти составляет O(1).

Итого:

- По времени: O(k).
- По памяти: O(1).

Код:

```
3. #include <bits/stdc++.h>
4.
5. #define repeat(x) for (int64_t i = (x); i--;)
6.
7. using namespace std;
8.
9. int main() {
10.     int64_t a, k, last_value = -1;
11.     uint16_t b, c, d;
12.
13.     cin >> a >> b >> c >> d >> k;
14.
15.     repeat(k) {
16.         a = a * b - c;
17.
18.         if (a > d) {
19.             a = d;
20.         }
21.
22.         if (a <= 0) {
23.             a = 0;
24.             break;
25.         }
26.
27.         if (a == last_value) break;
28.         last_value = a;
29.     }
30.
31.     cout << a << endl;
32.
33.     return 0;
34. }
```