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 №2 (Яндекс.Контест)

Выполнил:

Студент группы P3215

Барсуков М.А.

Преподаватель:

Косяков М.С.

Санкт-Петербург, 2024

Задачи из 2-го блока Яндекс.Контеста

Задача №5 «E. Коровы в стойла»

Описание:

K коров расставлены в *N* упорядоченных стойлах так, чтобы минимальное расстояние между ними было как можно большее. Нужно найти такое расстояние.

Доказательство:

Для решения задачи будем подбирать расстояние между стойлами так, чтобы уместить всех коров с максимальным расстоянием. Очевидно, что если расстояние между стойлами маленькое, мы всегда сумеем расставить всех коров (и, возможно, большее число коров, чем у нас есть), а если расстояние слишком большое, то расставить коров на такое расстояние не получится. Для подбора расстояния будем использовать алгоритм бинарного поиска, пока не придем к одному числу на нашем промежутке – решению. Начальный интервал поиска (мин. и макс. расстояние) будет между 0 и расстоянием между первой и последней координатой.

В процессе выполнения бинарного поиска мы получаем текущее проверяемое расстояние (как среднее). Первую корову мы всегда можем ставить в первое стойло, а остальных распределяем: пытаемся, перебирая координаты, ставить следующую корову в стойло, расстояние между которым и стойлом прошлой коровы больше текущего проверяемого расстояния, и сохраняем число расставленных коров. Если удалось расположить всех коров для данного расстояния, то мы меняем левую границу на текущее проверяемое расстояние + 1, иначе меняем правую на текущее проверяемое расстояние. Благодаря бинарному поиску каждый такой раз мы сокращаем наш интервал поиска в два раза. Когда он заканчивается, искомое максимальное расстояние будет в совпадающих границах.

Алгоритмическая сложность:

1. **По времени:** Алгоритм ищет оптимальную дистанцию бинарным поиском по промежутку [0, *C*], где *C* – это расстояние между координатами первого и последнего стойла, и для каждого такого варианта расставляет коров в цикле по *N* стойлам, что в среднем и худшем случаях временную сложность – O(N*log(C)).
2. **По памяти:** Программа хранит только массив из *N* стойл и некоторое постоянное число переменных (n, k, left, right, current_try_distance, cows_count, last_cow_coord), поэтому сложность по памяти O(N).

Итого:

- По времени: O(N*log(C)) (линейная логарифмическая сложность).
- По памяти: O(N) (линейная сложность).

Код:

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

using namespace std;

#define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)

int main() {
    int n, k;
    cin >> n >> k;

    int coords[n];
    repeat(n, i) cin >> coords[i];

    int left = 0;
    int right = coords[n - 1] - coords[0];

    if (k <= 2) {
        cout << right;
        return 0;
    }

    while (left != right) {
        int current_try_distance = (left + right) / 2;

        int cows_count = 1;
        int last_cow_coord = coords[0];

        for (int coord : coords) {
            if (coord - last_cow_coord > current_try_distance) {
                last_cow_coord = coord;
                cows_count++;

                if (cows_count >= k) {
                    left = current_try_distance + 1;
                    break;
                }
            }
        }

        if (cows_count < k) right = current_try_distance;
    }

    cout << left;
    return 0;
}
```

Задача №6 «F. Число»

Описание:

Из нескольких строк (хотя бы в одной строке первая цифра отлична от нуля) нужно собрать одну строку, которая представляется максимальным числом из всех возможных способов сбора строк.

Доказательство:

Будем считать, что нам нужно отсортировать данные строки по убыванию их «значимости» для результата так, чтобы при объединении мы получили искомую максимальную числовую строку. Тогда очевидно, что для сортировки нам нужен критерий для сравнения любых двух строк, чтобы понять условие того, что «значимость» одной строки была меньше значимости другой. В C++ мы можем сравнивать строки по лексикографическому порядку, и так как в данных строках цифры — это символы, то сравнивать мы все так же можем строки по лексикографическому порядку (“1” < “2” < “3” < … < “9”). Алгоритмы сначала вызывают компаратор для пары элементов x и y. Если компаратор вернул *true*, значит, элемент x меньше y и он должен идти в коллекции перед элементом y, если *false*, то компаратор вызывается повторно для пары y и x. Если компаратор опять вернул *false*, значит, элементы равны, иначе порядок определен.

Тогда для сравнения строк `str1` и `str2` мы можем использовать `str1 + str2 > str2 + str1`. Например, при сравнении `str1`=“69” и `str2`=“420” у нас будет “69420” > “42069”, что верно, т.е. второе значение должно идти после первого, а значит эти две строки будет отсортированы как …`str1, str2`…, то есть как `str1 + str2`, то есть мы получили максимальную конкатенацию. Кроме того, в ходе сравнения строка, начинающаяся с нуля, будем меньше любой строки, начинающейся не с нуля, т.е. после сортировки на первом месте гарантированно будет стоять строка с ненулевым началом, значит в результате будет валидное число.

Для реализации этой идеи будем использовать структуру `multiset` из STL, которая при добавлении значения будет автоматически поддерживать набор в отсортированном порядке. После того, как мы закончим ввод данных нам строк, набор строк уже будет отсортирован, и мы сможем вывести его.

Алгоритмическая сложность:

1. **По времени:** Пусть нам дают *N* строк, значит вводить строки мы будем *N* раз. Так как каждый раз при вводе мы используем `insert` для `multiset`, работающий за O(log N), а потом *N* раз выводим строки (т.е. делаем ~ N*log(N) + N операций), то в среднем и худшем случаях временная сложность – O(N*log(N)).
2. **По памяти:** Программа хранит только `multiset` из *N* строк и некоторое постоянное число переменных, что требует O(N) памяти для *N* строк.

Итого:

- По времени: O(N*log(N)) (линейная логарифмическая сложность).
- По памяти: O(N) (линейная сложность).

Код:

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

using namespace std;

int main() {
    auto compare = [](const string& str1, const string& str2) { return str1 + str2 > str2 + str1; };

    multiset<string, decltype(compare)> data(compare);

    for (string input; cin >> input;) data.insert(input);
    for (const string& str: data) cout << str;

    return 0;
}
```

Задача №7 «G. Кошмар в замке»

Описание:

Нужно переставить символы в строке так, чтобы ее вес получился максимальным (*Вес строки, которую ты можешь получить из s многократным обменом любых двух букв, вычисляется так: для каждой буквы алфавита посчитай максимальное расстояние между позициями, в которых стоит эта буква и перемножь его с весом этой буквы*).

Доказательство:

Рассмотрим условие максимального веса. Очевидно, что если буква встречается 1 раз, то она не влияет на вес строки, так как нет расстояния до другой буквы (то есть = 0). Если это буква встречается 2 раза, то нужно расставить ее симметрично на краях строки в зависимости от веса – бОльшие по весу буквы будут ближе к краю строки (например, так: AB…BA при весе A > B).

Если буква встречается >2 раз, то для двух из них мы применяем правило выше для 2 штук, а остальные n-2 экземпляра не влияют на вес (т.к. максимальное расстояние для этой буквы уже обеспечили первые 2 экземпляра, то есть они не изменяют вес).

Таким образом, стратегия получения максимального веса такова. У нас есть «крайние» и «мусорные» символы. Если буква встречается 1 раз, она «мусорная», если >= 2 раз, то 2 экземпляра становятся крайними, чтобы обеспечить максимальное расстояние, а остальные из них «мусорными». «Крайние» символы мы будем располагать в зависимости от веса буквы по краям строки, «мусорные» будут между краями, по середине. Очевидно, что «крайние» символы мы сортируем так, чтобы **буквы, имеющие максимальный вес, получали при постановке в строке и максимальное расстояние**, чтобы максимизировать итоговый вес.

Реализуем это так: будем хранить 26 пар символ-вес, и отсортируем их по весу, после чего посчитаем, сколько раз каждая буква встречается в строке, а также количество букв, встречающихся >= 2 раз – это число пар символов, которые будут на краях, и это же число и есть сдвиг от начала (и от конца), с которого начинаются «мусорные» символы. После этого заводим указатели на начало и конец строки, в цикле для отсортированных пар: если буква данной пары встречается >= 2 раз, то мы ставим эту букву по указателям начала и конца и сдвигаем указатели ближе к центру, а также уменьшаем число появлений этих букв на 2. В том же цикле после установки парных символов, пока их число не закончится, мы будем устанавливать их в мусорные ячейки и сдвигать указатель мусорных ячеек.

Таким образом, после выполнения этого алгоритма мы получим перестановку строки с максимальным весом.

Алгоритмическая сложность:

1. **По времени:** Так как букв у нас всегда 26, худшая сложность сортировка весов не зависит от введённых весов. Пусть в строке *s* есть *N* символов. Число встреченных букв считаем за *N* выполнений цикла, потом для каждого из 26 букв-весов мы заполняем новую строку этой буквой, причем для каждого веса выполняем цикл из стольких повторений, сколько встречалась эта буква, значит для каждой буквы мы прохожим по количеству ее экземпляром, а значит это количественно эквивалентно циклу по всем символам строки. Таким образом, в среднем и худшем случаях временная сложность – O(N).

2. **По памяти:** Мы храним 26 пар символ-вес и 26 счетчиков символов, что является постоянным числом занятой памяти, а также введённую строку из *N* символов. В таком случае сложность по памяти будет O(N).

Итого:

- По времени: O(N) (линейная сложность).
- По памяти: O(N) (линейная сложность).

Код:

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

vector<pair<char, int>> weights;
map<char, int> char_count;

int main() {
    string s;
    cin >> s;
    for (int cur_weight, i = 0; i < 26; i++) {
        cin >> cur_weight;
        weights.push_back(make_pair('a' + i, cur_weight));
    }

    sort(weights.begin(), weights.end(),
        [](const pair<char, int> &a, const pair<char, int> &b) {
            return a.second > b.second;});

    for (char& c : s) char_count[c]++;
    int ptr_after_duos = count_if(char_count.begin(), char_count.end(),
        [](pair<char, int> p) { return p.second >= 2; }
    );

    int start = 0;
    int end = s.length() - 1;
    char result[s.length()];
    result[s.length()] = 0;
    for (auto it : weights) {
        char c = it.first;

        if (char_count[c] >= 2) {
            result[start] = c;
            result[end] = c;
            char_count[c] -= 2;
            start++;
            end--;
        }

        for (; char_count[c] > 0; char_count[c]--) {
            result[ptr_after_duos] = c;
            ptr_after_duos++;
        }
    }
    printf("%s", result);
    return 0;
}
```

Задача №8 «H. Магазин»

Описание:

Пусть в чеке n товаров, тогда n/k округленное вниз самых дешевых из них достаются бесплатно. Однако товары можно разбить на несколько чеков, и благодаря этому потратить меньше денег. Какая минимальную сумму можно заплатить за выбранные товары, возможно разбив их на несколько чеков?

Доказательство:

Очевидно, что мы заплатим минимально, если сумма цен бесплатных товаров максимальна. Брать в чеке меньше k товаров не оптимально, так как тогда бесплатных товаров не будет вообще, а брать товаров nk, n > 2 не оптимально, так как в этом случае у нас будет n бесплатных товаров, но самых дешевых среди этих nk. Таким образом, оптимально будет брать товары так, чтобы на один чек был один бесплатный товар (т.е. берем k товаров), причем цена k-тых товаров должна быть максимальна.

Чтобы сделать это, отсортируем товары по убыванию цены. Для того, чтобы больше сэкономить, мы должны будем выбирать каждый k-тый товар – он будет бесплатным. Мы забираем бесплатно самый дешевый товар из группы k самых дорогих товаров, потом из группы k вторых по дороговизне товаров и так далее. Очевидно, что так мы сэкономим больше всего.

Алгоритмическая сложность:

1. **По времени:** Мы вводим и сортируем n товаров и проходим по ~ n/k товаров для экономии и считаем итоговую сумму цен (n + n*log(n) + n/k + n). Таким образом, в среднем и худшем случаях временная сложность – O(n*log(n)).
2. **По памяти:** Программа использует фиксированное количество переменных (n, k, i), а также массив из n введенных товаров. Оно зависит от размера входных данных, поэтому сложность по памяти составляет O(n).

Итого:

- По времени: O(n*log(n)) (линейная логарифмическая сложность).
- По памяти: O(n) (линейная сложность).

Код:

```
#include <bits/stdc++.h>
using namespace std;
#define repeat(times, i) for (int (i) = 0; (i) < (times); (i)++)

int main() {
    int n, k;
    cin >> n >> k;
    int elements[n];
    repeat(n, i) cin >> elements[i];
    sort(elements, elements + n, greater<int>());
    for (int i = k - 1; i <= n; i += k) elements[i] = 0;

    cout << accumulate(elements, elements + n, 0, plus<int>());
    return 0;
}
```