Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №2

по «Алгоритмам и структурам данных»

Базовые задачи

Выполнил:

Студент группы P3233

Гуменник П. О.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2023

Оглавление

Задача E. Коровы в стойла

Код:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool canPlaceCows(const vector<int>& stalls, int distance, int numCows) {
    int lastCowPos = stalls[0];
    int cowsPlaced = 1;

    for (int i = 1; i < stalls.size(); ++i) {
        if (stalls[i] - lastCowPos >= distance) {
            cowsPlaced++;
            lastCowPos = stalls[i];
            if (cowsPlaced == numCows) {
                return true;
            }
        }
    }

    return false;
}

int findOptimalDistance(const vector<int>& stalls, int numCows) {
    int low = 1;
    int high = stalls.back() - stalls.front(); // Maximum possible distance

    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (canPlaceCows(stalls, mid, numCows)) {
            low = mid + 1; // Try larger distances
        } else {
            high = mid - 1; // Try smaller distances
        }
    }

    return high; // Optimal distance is the last valid 'high'
}

int main() {
    int numStalls, numCows;
    cin >> numStalls >> numCows;

    vector<int> stalls(numStalls);
    for (int i = 0; i < numStalls; ++i) {
        cin >> stalls[i];
    }

    sort(stalls.begin(), stalls.end()); // Sort stalls for efficient binary search

    int optimalDistance = findOptimalDistance(stalls, numCows);
    cout << optimalDistance << endl;

    return 0;
}
```

Пояснение к примененному алгоритму:

Постановка задачи:

Нам дано N стойл и K коров. Задача состоит в том, чтобы разместить коров в стойлах так, чтобы минимальное расстояние между любыми двумя коровами было максимальным.

Алгоритм (бинарный поиск):

Инициализация:

Определить функцию isPossible(distance), которая проверяет, можно ли разместить все K коров в стойлах с минимальным расстоянием между ними.

Установить low = 1 (минимально возможное расстояние) и high = max_stall_position (максимально возможное расстояние).

Бинарный поиск:

Пока low <= high:

Вычислять mid = (low + high) / 2.

Если isPossible(mid) - true, это значит, что мы можем разместить коров на расстоянии не менее середины. Поэтому пробуем увеличить расстояние: low = mid + 1.

В противном случае нужно уменьшить расстояние: high = mid - 1.

Результат:

После цикла high будет содержать наибольшее расстояние, для которого isPossible было true, что является максимально возможным минимальным расстоянием между коровами.

Доказательство корректности:

Инвариант: На протяжении всего бинарного поиска выполняется следующий инвариант:

isPossible(distance) истинно для всех расстояний в диапазоне [low, mid].

isPossible(distance) ложно для всех расстояний в диапазоне (mid, high].

Базовый случай: Изначально low = 1 и high = max_stall_position.

isPossible(1) всегда истинно (мы можем поместить коров в любое стойло).

isPossible(max_stall_position) истинно только в том случае, если существует не менее K стойл с таким расстоянием между ними.

Индуктивный шаг: Предположим, что инвариант выполняется до итерации цикла.

Если isPossible(mid) истинно:

Согласно инварианту, все расстояния в [low, mid] возможны.

Мы обновляем low = mid + 1, сохраняя инвариант для нового диапазона [low, high].

Если isPossible(mid) ложно:

Согласно инварианту, все расстояния в (mid, high) невозможны.

Мы обновляем high = mid - 1, сохраняя инвариант для нового диапазона [low, high].

Завершение: Когда low > high, цикл завершается.

Согласно инварианту, high - это наибольшее расстояние, для которого isPossible истинно.

Все расстояния, превышающие high, невозможны (иначе цикл продолжался бы).

Таким образом, бинарный поиск правильно находит максимально возможное минимальное расстояние между коровами.

Задача F. Число

Код:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

bool compare(const string& a, const string& b) {
    return a + b > b + a; // Concatenate and compa
}

int main() {
    vector<string> numbers;
    string num;

    while (cin >> num) {
        numbers.push_back(num);
    }

    sort(numbers.begin(), numbers.end(), compare);

    string result;
    for (const string& num : numbers) {
        result += num;
    }
    cout << result << endl;

    return 0;
}
```

Пояснение к примененному алгоритму:

Постановка задачи:

Нам дан набор строк, представляющих числа. Задача состоит в том, чтобы расположить эти строки таким образом, чтобы при объединении они образовывали наибольшее возможное число.

Алгоритм (жадный с пользовательским сравнением):

Вход: Код считывает входные строки из стандартного потока ввода (cin) и сохраняет их в векторе с именем numbers.

Пользовательская функция сравнения:

Функция сравнения определяется для лексикографического сравнения двух строк определенным образом. Она объединяет строки в обоих возможных порядках (a + b и b + a) и сравнивает результаты. Это сравнение гарантирует, что строки будут расположены в порядке убывания, исходя из их потенциальной способности образовать большее число при объединении.

Сортировка:

Функция sort используется для сортировки строк в векторе чисел с помощью пользовательской функции сравнения. На этом этапе сортировки строки располагаются в том порядке, который позволит получить наибольшее возможное конкатенированное число.

Конкатенация и вывод:

Код выполняет итерации по отсортированному вектору чисел и конкатенирует каждую строку в строку-результат.

Наконец, строка результата, которая теперь содержит наибольшее возможное число, выводится в стандартный поток вывода (cout).

Доказательство корректности:

Ключ к корректности этого алгоритма лежит в пользовательской функции сравнения. Она гарантирует, что строки будут отсортированы таким образом, что при их объединении приоритет будет отдан формированию большего числа.

Рассмотрим две строки, a и b. Если a + b > b + a, это означает, что размещение a перед b приведет к большему конкатенированному числу. Функция сравнения улавливает эту логику и соответствующим образом направляет процесс сортировки.

Сортируя строки с помощью сравнения, алгоритм гарантирует, что результирующая конкатенация будет наибольшим возможным числом, которое может быть образовано из заданных входных строк.

Задача G. Кошмар в замке

Код:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <string>

using namespace std;

int main() {
    string s;
    cin >> s;

    vector<int> weights(26);
    for (int& weight : weights) {
        cin >> weight;
    }

    map<char, int> frequency;
    for (char c : s) {
        frequency[c]++;
    }

    vector<pair<int, char>> freqChars;

    for (const auto& [charact, freq] : frequency) {
        if (freq > 1) {
            freqChars.push_back(make_pair(weights[charact - 'a'], charact));
        }
    }

    sort(freqChars.rbegin(), freqChars.rend());

    string result(s.length(), ' ');

    int start = 0;
    int end = s.length() - 1;
    for (const auto& p : freqChars) {
        char ch = p.second;
        if (frequency[ch] > 1) {
            result[start++] = ch;
            result[end--] = ch;
            frequency[ch] -= 2;
        }
    }

    for (const auto& [charact, freq] : frequency) {
        while (frequency[charact] > 0) {
            if (start > end) break;
            if (result[start] == ' ') {
                result[start++] = charact;
            } else if (result[end] == ' ') {
                result[end--] = charact;
            }
            frequency[charact]--;
        }
    }

    cout << result << endl;

    return 0;
}
```

Пояснение к примененному алгоритму:

Постановка задачи:

Задача состоит в перестановке букв в строке таким образом, чтобы максимизировать вес строки. Вес строки определяется как сумма произведений максимальных расстояний между позициями одинаковых букв на вес каждой буквы. У нас есть строка, состоящая из строчных латинских букв, и заданы веса для каждой буквы алфавита.

Алгоритм:

Вход: Программа считывает строку s из стандартного потока ввода и массив весов weights для каждой буквы алфавита.

Подготовка: Создается карта frequency, где для каждой буквы строки s подсчитывается ее количество в строке.

Обработка: Для каждой буквы, встречающейся более одного раза, создается пара, содержащая ее вес (получаемый из массива weights) и сам символ. Эти пары помещаются в вектор freqChars.

Сортировка: Вектор freqChars сортируется в убывающем порядке по весу букв.

Формирование результата: Создается строка result той же длины, что и исходная, заполненная пробелами. По сортированному вектору freqChars буквы расставляются в result так, чтобы максимально расширить расстояния между одинаковыми буквами, ставя их на начальные и конечные позиции строки result.

Заполнение остатков: Оставшиеся буквы (те, что встречаются один раз или не были расставлены) добавляются в промежутки.

Вывод: Выводится измененная строка result, представляющая собой искомую строку максимально возможного веса.

Доказательство корректности:

Алгоритм корректен, поскольку он увеличивает расстояние между одинаковыми буквами, у которых самый большой вес, тем самым максимизируя общий вес строки. Поскольку вес строки зависит от произведения веса букв на их максимальное расстояние в строке, распределение букв с наибольшим весом на максимально далекие позиции увеличивает общий вес строки. При этом, если существуют различные способы перестановки, которые приводят к одинаковому максимальному весу, любая из таких перестановок будет являться корректным решением задачи.

Задача H. Магазин

Код:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int n, k;
    cin >> n >> k;
    vector<int> prices(n);
    for (int i = 0; i < n; i++) {
        cin >> prices[i];
    }

    sort(prices.begin(), prices.end(), greater<int>());

    long long total_cost = 0;
    for (int i = 0; i < n; i++) {
        if (i % k != k - 1) {
            total_cost += prices[i];
        }
    }

    cout << total_cost << endl;

    return 0;
}
```

Пояснение к примененному алгоритму:

Постановка задачи:

Билл хочет максимально сэкономить при покупке товаров в магазине, где действует акция "каждый k-й товар бесплатно". Нужно определить, какую минимальную сумму денег Билл сможет заплатить, возможно разбив свои покупки на несколько чеков.

Алгоритм:

Вход: Программа считывает два числа n (количество товаров) и k (через сколько товаров один можно получить бесплатно), а затем считывает n целых чисел, представляющих цены товаров.

Сортировка: Список цен на товары сортируется в убывающем порядке, чтобы сначала обрабатывать самые дорогие товары.

Вычисление стоимости: Программа проходит по списку отсортированных цен и суммирует стоимость всех товаров, кроме тех, которые могут быть получены бесплатно в рамках акции (каждый k-й товар). Товары, подходящие под условия акции (то есть каждый k-й товар), не добавляются в общую сумму.

Вывод: Программа выводит общую стоимость покупок Билла после применения акции.

Доказательство корректности:

При оптимальном разделении товары группируются так, что каждая группа из k товаров содержит товары максимально близкие по стоимости. Для любого чека, содержащего k товаров, минимальная возможная стоимость бесплатного товара будет максимальна при условии, что все k товаров одинаковы по цене.

Тогда разбиение товаров на чеки таким образом, что каждый чек содержит товары, максимально приближенные друг к другу по стоимости, и так, что самые дорогие товары группируются вместе, а самые дешевые — также вместе, гарантирует, что бесплатные товары будут иметь максимально возможную стоимость для каждой группы из k, минимизируя таким образом общую стоимость покупки.