Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №2

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P3219

Билобрам Д. А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2023

Задача №E «Коровы в стойле»

Пояснение к примененному алгоритму:

Алгоритм заключается в поиске максимального минимального (между коровами) расстояния бинарным поиском. Левая граница будет 1 (наименьшее возможное расстояние) а правая будет разницей между первым и последним стойлом. Получая середину между левой и правой границей алгоритм пробует расставить коров так, чтобы расстояние между ними было не меньше этого значения. Если расставить коров удаётся, ты алгоритм пробует увеличить минимальное расстояние смещая левую границу поиска, если же расставить коров не удаётся, алгоритм смещает правую границу поиска.
Алгоритм будет работать корректно, так как он бинарным поиском находит такие минимальные расстояния k и k+1, что расставить коров на расстоянии k ещё можно, а на расстоянии k+1 уже невозможно, следовательно k является максимальным минимальным расстоянием между коровами.

Оценка сложности по времени: O(n * log n)

Бинарный поиск работает за log n, на каждой итерации мы перебираем n элементов.

Оценка сложности по памяти: O(n)

Вектор из n элементов.

Код алгоритма:

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

    vector<int> seats(n);

    for (int i = 0; i < n; i++)
    {
        cin >> seats[i];
    }

    int left = 1;
    int right = seats[n - 1] - seats[0];
    int mid;
    int min;

    while (left <= right)
    {

        mid = (left + right) / 2;

        int dist = 0;
        int c = 1;
        for (int i = 0; i < n - 1; i++)
```

```
        {

            if (dist == 0)
            {
                dist = seats[i + 1] - seats[i];
            }

            if (dist < mid)
            {
                dist += seats[i + 2] - seats[i + 1];
            }
            else
            {
                dist = 0;
                c += 1;
            }
        }
        if (c >= k)
        {
            left = mid + 1;
            min = mid;
        }
        else
        {
            right = mid - 1;
        }
    }

    cout << min;
}
```

Задача №F «Число»

Пояснение к примененному алгоритму:

Алгоритм заключается в сортировке всех строк по принципу A + B > B + A.

Для доказательства корректности алгоритма нужно доказать два тезиса:

1) Если A + B > B + A то строка A должна идти перед строкой B.
2) Выполнение 1) для всех пар строк максимизирует значение итогового числа.

Первый пункт является истиной, т.к если A + B > B + A то число полученное конкатенацией A перед B дает большее число чем конкатенация B перед A.

Второй пункт можно доказать через противоречие – предположим что есть последовательность в которой есть пара которая не соответствует пункту 1) и при этом производит наибольшее число, это означает что при A + B > B + A строка B идёт перед строкой A. Тогда переставляя местами B и A мы получил число большее чем было, что противоречит тому, что предыдущее число наибольшее.

Оценка сложности по времени: O(n * log n)

std::sort отрабатывает за n * log n в наихудшем случае.

Оценка сложности по памяти: O(n)

Вектор из n элементов.

Код алгоритма:

```
bool is_better(const string &a, const string &b)
{
    return a + b > b + a;
}

int main()
{
    vector<string> vec;
    string str;

    while (cin >> str)
    {
        vec.push_back(str);
    }

    sort(vec.begin(), vec.end(), is_better);

    for (const string &el : vec)
    {
        cout << el;
    }
}
```

Задача №G «Кошмар в замке»

Пояснение к примененному алгоритму:

Алгоритм сортирует буквы по их весу а затем проходится по буквам в порядке убывания их веса и если количество этой буквы в строке > 1, то алгоритм вычитает 2 из количества этой буквы в строке и расставляет эту букву в конец правой части итоговой строки и в начало левой части итоговой строки, затем он добавляет эту букву в центр строки оставшееся количество раз.

Для доказательства корректности алгоритма рассмотрим букву c0 с весом w и букву c1 с весом w+1. Расстановка …c1…c0…c0…c1…(c1 – позиция k, c0 – позиция k+1, c0 – позиция n, c1 – позиция n+1) всегда будет выгоднее чем расстановка …c0…c1…c1…c0… …(c0 – позиция k, c1 – позиция k+1, c1 – позиция n, c0 – позиция n+1) так как при первой расстановке мы получим вес строки (1) (w+1)*(n+1-k) + w*(k+1-n) = -k + n + 2*w + 1, при второй расстановке получим вес строки (2) w*(n+1-k)+(w+1)*(k+1-n) = k – n + 2*w + 1; (2) – (1) = 2*k – 2*n а так как n > k тогда (2) - (1) < 0, следовательно вес строки (1) больше веса строки (2) а следовательно выгоднее ставить буквы с большим весом дальше от центра.

Оценка сложности по времени: O(n * log n)

Доступ по ключу к unordered_map в худшем случае имеет сложность O(n),

std::sort отрабатывает за n * log n в наихудшем случае.

Оценка сложности по памяти: O(n)

Вектор из n элементов.

Код алгоритма:

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

    vector<pair<int, char>> weights;

    unordered_map<char, int> char_count;

    for (int i = 0; i < 26; i++)
    {
        weights.push_back(make_pair(0, 'a' + i));
        cin >> weights[i].first;
    }

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

    for (int i = 0; i < s.size(); i++)
    {
        char_count[s[i]] += 1;
    }

    string border_left = "";
    string border_right = "";
    string center = "";

    for (auto inst : weights)
    {
        if (char_count[inst.second] > 1)
        {
            border_left += inst.second;
            char_count[inst.second] -= 2;
        }

        center += string(char_count[inst.second], inst.second);

    }

    for (int i = border_left.size() - 1; i >= 0; i--)
    {
        border_right += border_left[i];
    }

    string result = border_left + center + border_right;
```

```
    cout << result;
}
```

Задача №H «Магазин»

Пояснение к примененному алгоритму:

Идея алгоритма в том, чтобы отсортировать товары по цене а затем разбить чек на несколько чеков поменьше в каждом из которых меньше или равно k товаров идущих в порядке убывания изначального чека. Затем в каждом чеке последний товар будет бесплатным. В итоге вместо того чтобы получать бесплатно k самых дешевых товаров из большого чека мы получаем бесплатно k товаров которые самые дешевые в маленьких чеках (где самый дешевый товар стоит дороже чем в общем чеке).

Алгоритм будет работать корректно так как он стремится получить как можно больше бесплатных товаров с максимальной стоимостью. Это достигается формированием чеков по k товарам, так как если взять чеки по > k товаров, то суммарно чеков получится меньше => и бесплатных товаров тоже а если взять чеки по < k то бесплатных товаров вообще не будет. Максимизация стоимости бесплатных товаров происходит потому, что в большом чеке бесплатными бы стали самые дешёвые товары, в то время как при разделении на маленькие чеки бесплатными становятся товары с более высокой индивидуальной стоимостью.

Оценка сложности по времени: O(n * log n)

std::sort отрабатывает за n * log n в наихудшем случае.

Оценка сложности по памяти: O(n)

Вектор из n элементов.

Код алгоритма:

```
int main() {
    int n, k;
    cin >> n >> k;
    vector<int> prices(n);

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

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

    int sum = 0;
    for (int el : prices) {
        sum += el;
    }
```

```
    for (int i = k-1; i < n; i += k) {
        sum -= prices[i];
    }

    cout << sum;
}
```

Задача №1444 «Накормить элефопотама»

Пояснение к примененному алгоритму:

Основная идея в том, чтобы отсортировать все точки относительно начальной по величине полярного угла (если встречаются несколько точек с одинаковым углом алгоритм сортирует их в порядке отдаления от начальной точки). В таком случае если правильно выбрать первую точку обхода и идти по точкам в порядке не убывания полярного угла, то всегда можно пройти все точки. Однако если начинать обход с любой точки, то могут встретиться две точки, угол между которыми > 180 градусов, в таком случае если соединить их то есть вероятность пересечь путь от начальной точки к точке которая была первая по обходу. В этом случае нужно начать обход с той точки которая находится под углом > 180 градусов, потому что если начать с неё то все остальные точки для неё будут находиться под углом < 180 градусов и мы сможем обойти все точки.

Алгоритм будет работать корректно, так как при обходе точек в порядке не убывания полярного угла (относительно начальной точки) всегда можно соединить точки по прямой при этом не пересекая предыдущий путь. Исключением является ситуация в которой угол между соседними точками по возрастанию угла будет > 180 градусов, но если такой угол существует, то все остальные точки точно будут на других 180 градусах, а именно на них мы обход и начинаем.

Оценка сложности по времени: O(n * log n)

std::sort отрабатывает за n * log n в наихудшем случае.

Оценка сложности по памяти: O(n)

Вектор из n элементов.

Код алгоритма:

```
int main() {

    int n;
    cin >> n;
    vector<pair<int, int>> points(n); // x, y
    vector<pair<double, int>> angles(n-1); // angle, index

    cin >> points[0].first >> points[0].second;
```

```
    for (int i = 1; i < n; i++) {
        cin >> points[i].first >> points[i].second;
        angles[i-1].second = i;
        angles[i-1].first = atan2(points[i].second - points[0].second,
points[i].first - points[0].first) * (180.0 / M_PI);

    }

    sort(angles.begin(), angles.end());


    for (int i = 0; i < n-1; i++) {
        int c = i;
        while (angles[i].first == angles[i+1].first) {
            i += 1;
        }
        if (i != c) {

            vector<pair<double, int>> dist; // dist, index

            for (int j = c; j <= i; ++j) {
                int index = angles[j].second;
                int dx = points[index].first - points[0].first;
                int dy = points[index].second - points[0].second;
                double distance = sqrt(dx*dx + dy*dy);
                dist.push_back(make_pair(distance, index));
            }

            sort(dist.begin(), dist.end());

            for (int j = c; j <= i; ++j) {
                angles[j].second = dist[j - c].second;
            }

        }
    }

    int start = 0;

    for (int i = 0; i < n-1; i++) {

        if (abs(angles[i].first - angles[i+1].first) >= 180) {
            start = i+1;
            break;
        }

    }

    cout << n << "\n1\n";
    for (int i = 0; i < n-1; i++) {
        int index = (start + i) % (n-1);
        cout << angles[index].second + 1 << "\n";
    }

}
```

Задача №1207 «Медиана на плоскости»

Пояснение к примененному алгоритму:

Идея алгоритма заключается в том, чтобы выбрать сначала выбрать точку с минимальным y (чтобы полярный угол к остальным точкам был > 0 и <= 180), а затем отсортировать все остальные точки по полярному углу относительно выбранной точки. Тогда если взять серединную точку из вектора отсортированных точек, то мы получим n-2 точек с углом больше чем у серединной точки и n-2 точек с углом меньше чем у серединной точки. Эти две точки и будут ответом.

Алгоритм будет работать корректно, поскольку N чётно и точек всего N, отсортированный список точек (кроме начальной) будет содержать N−1 элементов. Выбрав серединную точку в этом списке, мы делим список на две части: одна часть содержит точки с углом меньше серединной точки, а другая — с углом больше. Добавив начальную точку к одной из этих частей, получаем две группы по (N−2)/2+1=N/2 точек.

Оценка сложности по времени: O(n * log n)

std::sort отрабатывает за n * log n в наихудшем случае.

Оценка сложности по памяти: O(n)

Вектор из n элементов.

Код алгоритма:

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

    vector<pair<long long, long long>> points(n);
    vector<pair<long double, int>> angles(n); // angle, index

    for (int i = 0; i < n; i++) {
        cin >> points[i].first >> points[i].second;
    }

    pair<int, int> base_one = points[0];
    int base_one_index = 0;
    for (int i = 0; i < n; i++) {
        if (points[i].second < base_one.second || (points[i].second ==
base_one.second && points[i].first < base_one.first)) {
            base_one = points[i];
            base_one_index = i;
        }
    }

    for (int i = 0; i < n; i++) {
        angles[i].second = i;
```

```
        angles[i].first = atan2(points[i].second - base_one.second,
points[i].first - base_one.first) * (180.0 / M_PI);

    }

    angles.erase(angles.begin()+base_one_index);

    sort(angles.begin(), angles.end());

    cout << base_one_index + 1 << " " << angles[(n-1)/2].second + 1;

}
```