Университет ИТМО

Факультет программной инженерии и компьютерной техники

Лабораторная работа №2

по «Алгоритмам и структурам данных»

Базовые задачи / Timus

Выполнил:

Студент группы P3211

Болорболд А.

Преподаватели:

Косяков М.С.

Тараканов Д.С.

Санкт-Петербург

2024

Задача №5 «Коровы»

1. **Пояснение к примененному алгоритму:**

Довольно известная задача, основанная на применение бинарного поиска. После ввода работает обычный бинарный поиск, в результате которого выводится максимальное расстояние между стойлами.

Исходный код:

```
#include<iostream>

int main() {
    int n, k, curDist = 1, maxDist;
    std::cin >> n >> k;
    int stall[n];
    for(int i = 0; i < n; i++){
        std::cin >> stall[i];
    }
    maxDist = stall[n - 1] - stall[0] + 1;
    while(maxDist - 1 > curDist){
        int mid = (curDist + maxDist) / 2;
        int last = stall[0];
        int cows = 1;
        for(int i = 0; i < n; i++) {
            if(mid <= stall[i] - last) {
                last = stall[i];
                cows++;
            }
        }
        (cows >= k) ? curDist = mid : maxDist = mid;
    }
    std::cout << curDist;
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

Здесь надо доказывать почему для решения задачи работает именно бинарный поиск. А в случае с бинарным поиском надо доказывать его корректность.

Данный алгоритм работает тогда и только тогда, когда набор исходных данных сортирована по умолчанию (иначе невозможно повторно определить середину). К счастью, это покрыто условием задачи, поэтому на этого обращать внимания нет смысла.

Теперь надо доказывать сам алгоритм.

Доказательство базировано на выражение алгоритма через монотонно возрастающей функции.

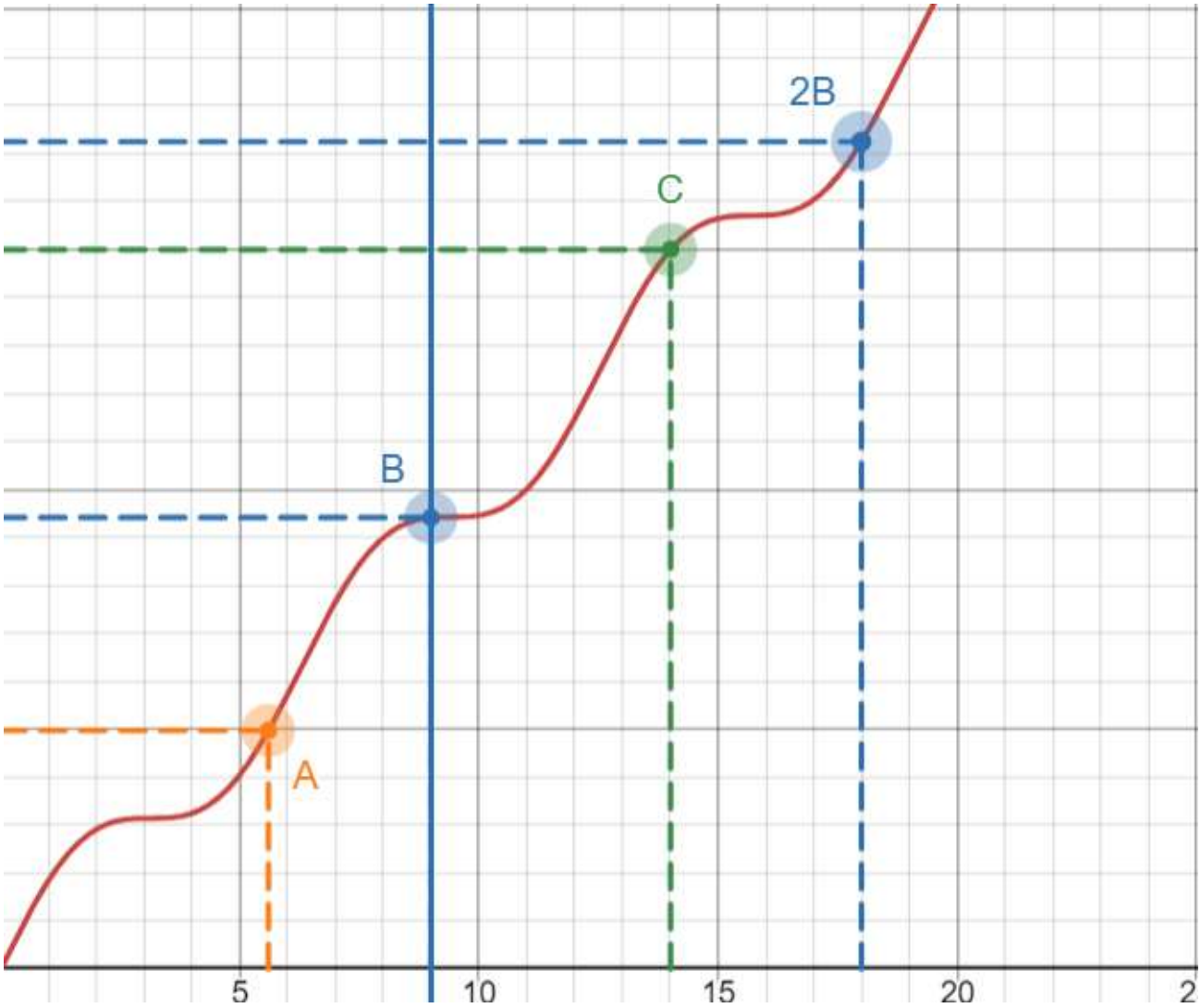
Представим монотонно возрастающую функцию ***f(x)***. Она задана таким образом, что при решения уравнении от этой функции невозможно найти корень аналитическим способом или через обратную функцию. Надо найти такую x_0, что $f(x_0) = c$.

Сначала берём такой ***a***, что $f(a) < c$. Потом берём границу b, такую, что $f(b) > c$. Очевидно, что этого делать сразу нелегко. Если $f(b) < c$, то находим $f(2b)$, так как это обратное логарифму. Если, например, попытаться найти $f(b + k)$, то вероятно, что ***c***

может быть настолько громадным, что до туда не дойти. Но если удвоить ***b*** до ***c***, то понятно, что мы обязательно находим такой b, который ***f(b) > c*** за максимум ~270 шагов при 10^{80} дискретных значений в функции. Оттуда мы **точно** можем найти x_0, согласно теореме Вейерштрасса, так как:

1. ***f(x)*** монотонна и непрерывна;
2. существуют максимум ***(b)*** и минимум ***(a)***.

Значит существует такой $f(x_0) = c$, что $f(a) <= f(x_0) = c <= f(b)$, что и требовалось доказать.



3. **Оценка сложности по времени:**

Несмотря на то, что здесь применяется алгоритм бинарного поиска (который обычно составляет O(logn)), сложность по времени должна составлять **O(n)**. Это происходит из-за того, что при поиске оптимального расстояния [0, max] для каждого возможного варианта алгоритм расставляет коровы в цикле по **n** стойлам.

4. **Оценка сложности по памяти:**

Для самого алгоритма — O(1). Но если учитывать сам массив, то в итоге получается **O(n)**.

Задача №6 «Число»

1. **Пояснение к примененному алгоритму:**

Здесь имплементирован собственный сравнительный метод. С помощью его логики получается так, что всегда получается большее значение сравниваемых строк.

Исходный код:

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

bool comparator(const std::string &s1, const std::string &s2) {
    return s1 + s2 > s2 + s1;
}

int main(){
    std::vector<std::string> number;
    std::string num;

    while(std::cin >> num){
        if(num.empty()) break;
        number.push_back(num);
    }

    std::sort(number.begin(), number.end(), comparator);
    for(const auto & i : number){
        std::cout << i;
    }
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

Здесь надо доказать, почему именно такая сортировка работает в данном случае. К самому сравнительному методу не должна быть вопросов, здесь всего лишь сравнивается значения строк, соединённые по возможным способам.

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 = “14” и str2 = “88” у нас будет “8814” > “1488”, 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.

3. **Оценка сложности по времени:**

Здесь с первого взгляда — O(n). Но с учётом самой сортировки итоговая сложность по времени должна составлять **O(n*logn)**.

4. **Оценка сложности по памяти:**

С применением вектора получается **O(n)**, даже несмотря на саму сортировку, которая в теории должна составлять O(logn).

Задача №7 «Кошмар в замке»

1. **Пояснение к примененному алгоритму:**

Здесь пришлось наряду с собственным сравнительным методом имплементировать ещё и структуру, которая представляет из себя саму букву и заданный вес. После сортировки (в результате которого точно получается так, чтобы между буквами были максимальный вес) идёт конкатенация, соответствующая заданию.

Исходный код:

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

struct element {
    char letter;
    int weight;
};

bool comparator(const element a, const element b) {
    if(a.weight == b.weight) {
        return a.letter < b.letter;
    } else {
        return a.weight < b.weight;
    }
}

int main() {
    const std::string alphabet = "abcdefghijklmnopqrstuvwxyz";
    std::string str, answer;
    std::cin >> str;
    std::unordered_map<char, int> weights;
    for (int i = 0; i < 26; ++i) {
        int x;
        std::cin >> x;
        weights[alphabet[i]] = x;
    }
    std::map<char, int> taken, all;
    for (char &i : str) {
```

```
        all[i]++;
    }
    std::vector<element> word;
    for (char &i : str) {
        if (all[i] > 1 && taken[i] < 2) {
            word.push_back({i, weights[i]});
            taken[i]++;
        } else {
            answer += i;
        }
    }
    std::sort(word.begin(), word.end(), comparator);
    for (int j = 0; answer.length() != str.length(); j += 2) {
        answer = word[j].letter + answer;
        answer += word[j + 1].letter;
    }
    std::cout << answer;
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

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 – бóльшие 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.

3. **Оценка сложности по времени:**

Даже с учётом самой сортировки (O(27*log27) = O(1)) итоговая сложность по времени должна составлять **O(n)**.

4. **Оценка сложности по памяти:**

С применением вектора и мапов должна получаться **O(n)**.

Задача №8 «Магазин»

1. **Пояснение к примененному алгоритму:**

Опять же, используется собственный сравнительный метод, после которого используется дефолтная сортировка. Единственная особенность — здесь проверяется чётность товаров.

Исходный код:

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

bool comparator(int const a, int const b){
    return a > b;
}
int main(){
    int n, k, total = 0;
    std::cin >> n >> k;
    std::vector<int> products;
    for(int i = 0; i < n; i++) {
        std::cin >> products[i];
    }
    std::sort(products.begin(), products.end(), comparator);
    for(int i = 1; i <= n; i++) {
        if(i % k != 0) {
            total += products[i - 1];
        }
    }
    std::cout << total;
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

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 n_k при $n > 2$ не оптимально, так как в этом случае у нас будет n бесплатных товаров, но самых дешевых среди этих n_k. 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. Таким образом мы сэкономим больше всего.

3. **Оценка сложности по времени:**

С учётом самой сортировки итоговая сложность по времени должна составлять **O(nlogn)**.

4. **Оценка сложности по памяти:**

С применением вектора должна получаться **O(n)**.

Задача №1444 «Накормить элефпотама»

1. **Пояснение к примененному алгоритму:**

Здесь уже поинтереснее. Кроме выражения тыквы (или точки) здесь более интересный сравнительный метод: сравнивается конкретно тангенсы точек. Если точки равны, то сравнивается сумма квадратов координат. После сортировки находится начало элефпотама.

Исходный код:

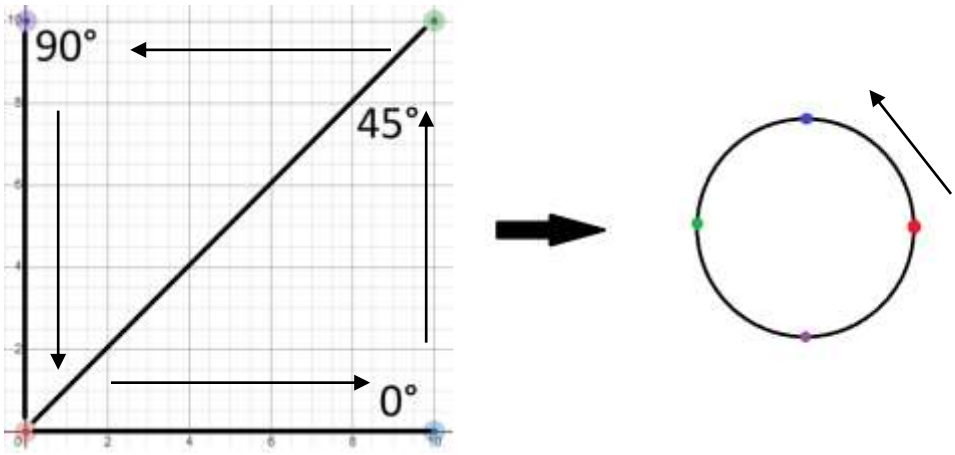
```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

struct pumpkin{
    int x;
    int y;
    int i;
};
bool comparator(const pumpkin a, const pumpkin b) {
    if(atan2(a.y, a.x) != atan2(b.y, b.x)) {
        return atan2(b.y, b.x) > atan2(a.y, a.x);
    }
    return pow(b.x, 2) + pow(b.y, 2) > pow(a.x, 2) + pow(a.y, 2);
}
int main() {
    int n, x_init, y_init, begin = 0;
    std::vector<pumpkin> lawn;
    std::cin >> n;
    std::cin >> x_init >> y_init;
    for(int i = 1; i < n; ++i) {
        int x, y;
        std::cin >> x >> y;
        lawn.push_back({x - x_init, y - y_init, i + 1});
    }
    sort(lawn.begin(), lawn.end(), comparator);
    for(int i = 0; i < n - 2; ++i) {
        if(lawn[i].x * lawn[i + 1].y <= lawn[i].y * lawn[i + 1].x
        && lawn[i].x * lawn[i + 1].x < -(lawn[i].y * lawn[i + 1].y)) {
            begin = i;
            break;
        }
    }
    std::cout << n << "\n" << 1 << "\n";
    for(int i = 0; i < n - 1; i++) {
        std::cout << (lawn[(begin + i) % (n - 1)].i) << std::endl;
    }
```

```
    return 0;
}
```

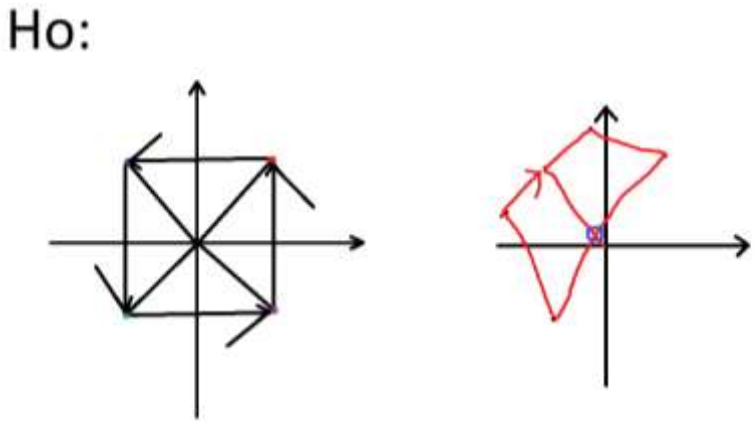
2. Корректность применяемого алгоритма:

По условию задачи элефпотам не должен пересекать свой путь. Чтобы такого не случилось, можно, чтобы элефпотам шёл по “окружности” (окружностью это было бы, если бы расстояния от первой точки до остальных были бы одинаковыми, но представим что это круг, но радиус он может менять). Чтобы выстроить круг, для каждой точки найдем угол между прямой соединяющей её и начало координат, и осью ***x***. Полученные углы отсортируем, и в таком порядке начнём движение.

Чтобы элефпотам не пересекал свой путь, надо чтобы он шел по кругу в одну сторону. Таким образом он не будет возвращаться, и не пересечет свои следы. Чтобы выстроить круг, для каждой точки найдем угол, между прямой, соединяющей центр координат и заданную точку и осью ***0x***. Полученные углы отсортируем, и будем идти по порядку. Таким образом элефпотам не пересекает свой след и сможет съесть все тыквы.



Казалось бы, на этом конец, но надо рассматривать направление обхода. Как мы будем его определять в общем? Здесь надо понимать, что раньше рассматривались примеры только с углом <180. Что если наоборот?



Как видно из иллюстрации, направление обхода не важно, если точки составляют между собой тупой, или в крайнем случае, развернутый угол, иначе происходит то

что творится в правом картинке: пересечение путей (из-за которого элефпотам будет заблудиться). Чтобы этого не произошло, нам надо найти правильную стартовую точку согласно двум вышеупомянутым абзацам.

В общем, элефпотаму, чтобы обходить все тыквы так, чтобы не пересек путь, надо выбрать ту точку с самым меньшим углом, откуда надо постепенно возрастать его до конца. Можно его выбрать слулчайным образом, если все углы тыкв составляют свыше 180 градусов, но это только частный случай.

3. **Оценка сложности по времени:**

С учётом дефолтной сортировки (quicksort) итоговая сложность по времени должна составлять **O(n*logn)**.

4. **Оценка сложности по памяти:**

С применением вектора должна получаться **O(n)**.

Задача №1604 «В стране дураков»

1. **Пояснение к примененному алгоритму:**

Здесь пришлось использовать помощь priority_queue. Это такая структура данных, который работает почти как стек (здесь FIFO), но ещё и автосортируемый (но требуется здесь класс). После этого здесь происходит поочередное вытаскивание знаков из вершины очереди до его опустошения.

Исходный код:

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

class comparable {
public:
    bool operator()(const std::pair<int, int> a, const std::pair<int, int> b) {
        if(a.first != b.first) {
            return a.first < b.first;
        }
        return a.second > b.second;
    }
};
int main() {
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>,
comparable> p_queue;
    int k;
    std::cin >> k;
    for(int i = 1; i <= k; ++i) {
        int x;
        std::cin >> x;
        p_queue.emplace(x, i);
    }
    while(!p_queue.empty()) {
        std::pair<int, int> top = p_queue.top();
        p_queue.pop();
        std::cout << top.second << " ";
        if(!p_queue.empty()) {
            std::pair<int, int> second_top = p_queue.top();
```

```
            p_queue.pop();
            std::cout << second_top.second << " ";
            if(second_top.first > 1) {
                p_queue.emplace(second_top.first - 1, second_top.second);
            }
        }
        if(top.first > 1) {
            p_queue.emplace(top.first - 1, top.second);
        }
    }
    return 0;
}
```

2. **Корректность применяемого алгоритма:**

Очевидно, что максимальное количество смен скоростей в последовательности ограничено сверху ***n***. Пусть таким алгоритмом количество смен скоростей по итогу получилось меньше, чем ***n***. Значит, в полученной строке идут неодинаковые пары, а в конце плато (выбираем максимумы, пока можем чередовать, а после подряд ставим оставшиеся знаки одного вида):

n0,ni , ... , nj ,n0, ... , n0.

Рассмотрим другую возможность расположить знаки в последовательности так, чтобы количество смен скоростей было максимально, и предположим, что количество смен знаков в таком случае будет больше. Тогда возьмём эту последовательность, и будем перемещать все n_i, соседствующие друг с другом, в конец последовательности так, чтобы плато было только в конце. Очевидно, что оно будет одно, так как знаки из двух наборов, идущих подряд, мы можем просто чередовать, пока не закончится один из них. Заметим, что в таком случае мы получим нашу исходную последовательность. Значит, количество смен знаков в рассматриваемой последовательности такое же, как в исходной. Противоречие.

3. **Оценка сложности по времени:**

С учётом дефолтной сортировки (heapsort), которая используется для сортировки очереди, итоговая сложность по времени должна составлять **O(n*logn)**.

4. **Оценка сложности по памяти:**

С применением очереди должна получаться **O(n)**.

Дополнительное задание:

Написать функцию heapify(), который преобразует массив в кучу (max-heap) с подходом снизу-вверх (bottom-up) за линейное время.

1. **Пояснение к примененному алгоритму:**

Функция принимает итераторы в качестве аргументов, в результате которого исходный вектор преобразуется в кучу (с использованием подхода просеивания вниз).

Исходный код:

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

int n;
void heapify(int *arr, int i){
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    int largest;
    if(l <= n && arr[l] > arr[i]){
        largest = l;
    } else {
        largest = i;
    }
    if(r <= n && arr[r] > arr[largest]){
        largest = r;
    }
    if(largest != i){
        int tmp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = tmp;
        heapify(arr, largest);
    }
}

void heapify(std::vector<int>::iterator begin, int i) {
    int largest;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if(l <= n && begin[l] > begin[i]) {
        largest = l;
    } else {
        largest = i;
    }
    if(r <= n && begin[r] > begin[largest]) {
        largest = r;
    }
    if(largest != i) {
        int tmp = begin[i];
        begin[i] = begin[largest];
        begin[largest] = tmp;
        heapify(begin, largest);
    }
}
void buildMaxHeap(int arr[], int sz) {
    for(int i = sz / 2 - 1; i >= 0; i--) {
        heapify(arr, i);
    }
}
void buildMaxHeap(std::vector<int>::iterator begin, std::vector<int>::iterator
end){
    for(int i = (int)(end - begin) / 2 - 1; i >= 0; i--) {
        heapify(begin, i);
    }
}

int main() {
    int arr[] = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17};
    n = sizeof(arr) / sizeof(arr[0]);
    std::cout << "Array to heap:\n";
    buildMaxHeap(arr, n);
    for(int i = 0; i < n; i++) {
```

```
        std::cout << arr[i] << " ";
    }
    std::vector<int> vec(arr, arr + n);
    std::cout << "\nVector to heap (custom):\n";
    n = (int) (vec.end() - vec.begin());
    buildMaxHeap(vec.begin(), vec.end());
    for(int i = 0; i < n; i++) {
        std::cout << vec[i] << " ";
    }
    std::cout << "\nVector to heap (make_heap):\n";
      std::make_heap(vec.begin(), vec.end());
    for (int i : vec) {
        std::cout << i << " ";
    }
    return 0;
}
```

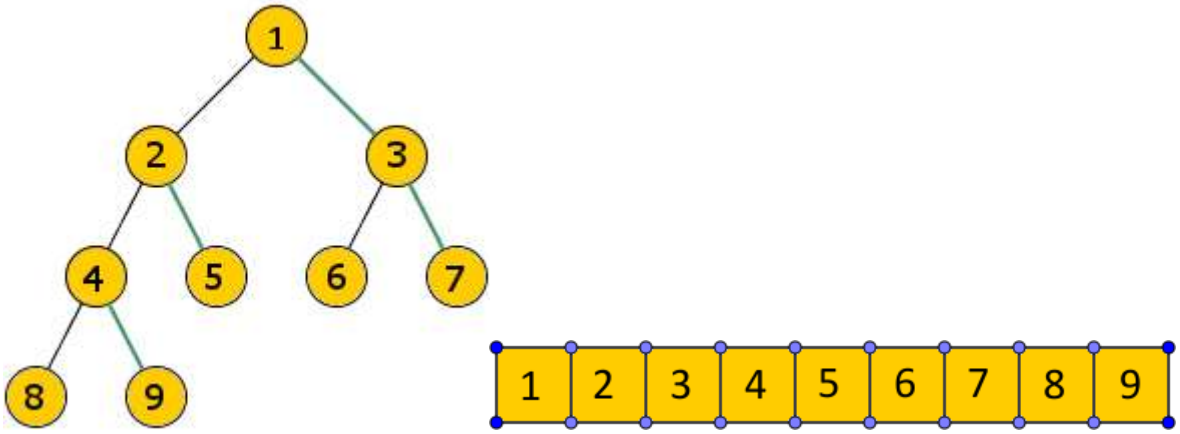
Результат (сравнение с библиотечным методом make_heap()):

```
D:\ITMOrbius\A&DS_XVIIstarPt_\Sorting\cmake-build-debug\Sorting.exe
Array to heap:
17 15 13 9 6 5 10 4 8 3 1
Vector to heap (custom):
17 15 13 9 6 5 10 4 8 3 1
Vector to heap (make_heap):
17 15 13 9 6 5 10 4 8 3 1
Process finished with exit code 0
```

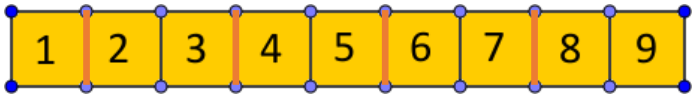
2. **Корректность применяемого алгоритма:**

Доказательство корректности будет состоять из 2 частей: доказательство индексов дочерних узел (почему они именно 2 * i + 1 и 2 * i + 2) и доказательство того, почему результирующая структура является двоичной max-кучей.

Первое доказательство можно привести с помощью индукции, но здесь будет применён другой подход (док-во с индукцией приведено ниже этого). Сначала выравниваем дерево в массив следующим образом:



Потом отделяем массив в пары так, чтобы они являлись дочерними какого-то общего узла:



Теперь начнём продумать с обратной стороны. Что если у нас есть массив пар [{2, 3}, {4, 5}, {6, 7}, {8, 9}], и мы хотим преобразовать их в один массив? Допустим, что мы уже поставили первые 3 пары:

[2 3] [4 5] [6 7]

Чему будет равен индекс числа 8 в итоговом массиве? Мы уже располагали первые 3 пары и они занимают 3 * 2 = 6 мест в начале, поэтому оно будет на 7 месте.

По коду:

```
std::pair<int, int> pairs[4] = { {2, 3}, {4, 5}, {6, 7}, {8, 9} };
int aggregate[2 * 4];
for (int i = 0; i < 4; i++) {
    aggregate[2 * i + 1] = pairs[i].first;
    aggregate[2 * i + 2] = pairs[i].second;
    std::cout << aggregate[2 * i + 1] << " " << aggregate[2 * i + 2] << "\n";
}
```

```
D:\ITMOrbius\A&DS_XVIIstarPt_\Sorting\
2 3
4 5
6 7
8 9
```

Ключевая часть кода: aggregate[2 * i + 1]. Можно очевидно понять, почему умножаем на 2: это для того, чтобы пропустить предыдущие i пары.

Сливая всё это (то есть, соотносив выравнивание дерева с преобразованием пар в один массив), получим следующее: у каждой пары есть родительский узел. Если [2 3] и [4 5] расположены один за другим, то их родители тоже расположены один за другим. У пары [2 3] родителем является 1 (корневой узел), у пары [6 7] родителем является 3. Поэтому родительский узел является индексом в этом массиве пар pairs[4] = { {2, 3}, {4, 5}, {6, 7}, {8, 9} }, и когда мы используем 2 * i + 1 для того, чтобы получить доступ к его левым дочерним узлом, мы можем это продумать как пропуск i пар дочерних узлов, образованные предыдущими узлами.

Не знаю, настолько этого можно принимать как доказательство, поэтому здесь приведено второе доказательство через индукцию.

Примечание: в англосфере обычно используется 2 * i и 2 * i + 1 для обозначения индекса в «массиве», так как в таких доказательствах нумерация «массива» начинается с 1.

1. Дочерние узлы корня (root(0)): child_1 = 2 * 0 + 1 = 1, child_2 = 2 * 0 + 2 = 2
2. Предполагая, что индексами дочерей k-ого узла являются child_1 = 2 * i + 1 и child_2 = 2 * i + 2,

3. Надо доказывать, что индексами дочерей (k+1)-ого узла являются: child_1 = 2 * (k + 1) + 1 и child_2 = 2 * (k + 1) + 2.

   Доказательство: так как дочери k-ого элемента расположены на 2 * k + 1 и 2 * k + 2 (исходя из допущения), то следующие 2 элементы равны 2 * k + 3 и 2 * k + 4. А так как:

   2 * k + 3 = 2 * (k + 1) + 1

   2 * k + 4 = 2 * (k + 1) + 2

   то очевидно, что третье утверждение является истинной.

   Значит, дочерние элементы расположены на индексах 2 * i + 1 и 2 * i + 2, ч.т.д.

   Второе доказательство отчасти исходит из первого, но здесь есть ещё одно важное составляющее: свойство кучи: если B является узлом-потомком узла A, то ключ(A) ≥ ключ(B).

   Из кода можно видеть, что максимальное значение сохраняется в течении работы этой функции. И можно ещё обнаружить, что если новое значение больше по значению текущей кучи, то в таком случае происходит обмен узлами. Это повторяется во всех уровнях из-за того, что вызов происходит рекурсивно. С помощью этого сохраняется свойство кучи, исходя из которого можно выручать за корректность работы алгоритма.

3. Оценка сложности по времени:

На самом нижнем уровне находятся не более 2^h узлов (их не надо просеивать вниз), на преднижнем уровне 2^{h-1} (то есть они просеиятся на один уровень вниз), на препреднижнем уровне 2^{h-2} узлов (на 2 уровень вниз) и так далее. Если просуммировать их всё, то получится:

$$T(N) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j} = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} = 2^h \cdot 2 = 2^{h+1}$$

Исходя из этого, можем подытожить, что сложность алгоритма по времени должна составлять **O(n)**, так как:

$$\sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2} \text{ для } x < 1$$

4. Оценка сложности по памяти:

Объём вспомогательной памяти зависит от стека вызова рекурсивной функции, поэтому сложность алгоритма по памяти должна составлять **O(logn)**.