Жадность как подход к решению задач (greedy algorithms)

Принцип, примеры, реализация, ограничения

Задачка – 1368A – «C+=»

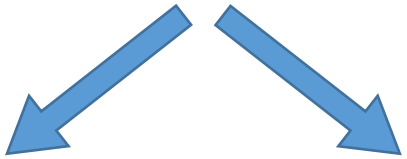
Лео создал новый язык программирования C+=. В C+= целочисленные переменные можно изменять только операцией «+=», которая прибавляет значение справа к переменной слева. Например, если выполнить «`a += b`», когда `a` = 2, `b` = 3, значение `a` станет равно 5 (значение `b` при этом не изменится).

Лео создал программу-прототип с двумя целочисленными переменными `a` и `b`, исходно содержащими некоторые положительные значения. Он может выполнить некоторое количество операций «`a += b`» или «`b += a`». Лео хочет протестировать обработку больших целых чисел, поэтому ему нужно сделать значение `a` либо `b` **строго больше**, чем некоторое данное число n. Какое наименьшее количество операций ему необходимо выполнить?

Перед нами типичная **задача оптимизации**, хоть и очень простая:

- Может быть много возможных решений
- Качество решения определяется некоторым параметром
- Требуется выбрать среди них одно оптимальное решение

Возможны всего 2 операции



a += b **b += a**

Какую из них нужно выбрать на каждом шаге, чтобы в итоге получить оптимальное решение?

Ответ интуитивно понятен: увеличивать наименьшее число в паре.

Такая тактика имеет формальное название:

«Локально-оптимальный выбор»

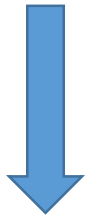
Итак, на каждом шаге жадного алгоритма из всех возможных вариантов выбирается самый оптимальный на данном конкретном шаге.

При этом:

- Прошлое можем помнить, но откатиться не можем
- О будущем не задумываемся

Принцип **жадного выбора** имеет место быть, когда:

Имеем последовательность локально-оптимальных выборов



Получаем глобально-оптимальное решение

Данная логика работает только при условии, что решаемая задача обладает **свойством локальности для подзадач**, т.е. оптимальное решение всей задачи обязательно содержит в себе оптимальные решения подзадач, из которых она состоит.

Получаем два условия для использования жадного алгоритма:

- **Принцип жадного выбора**
- **Оптимальность для подзадач**

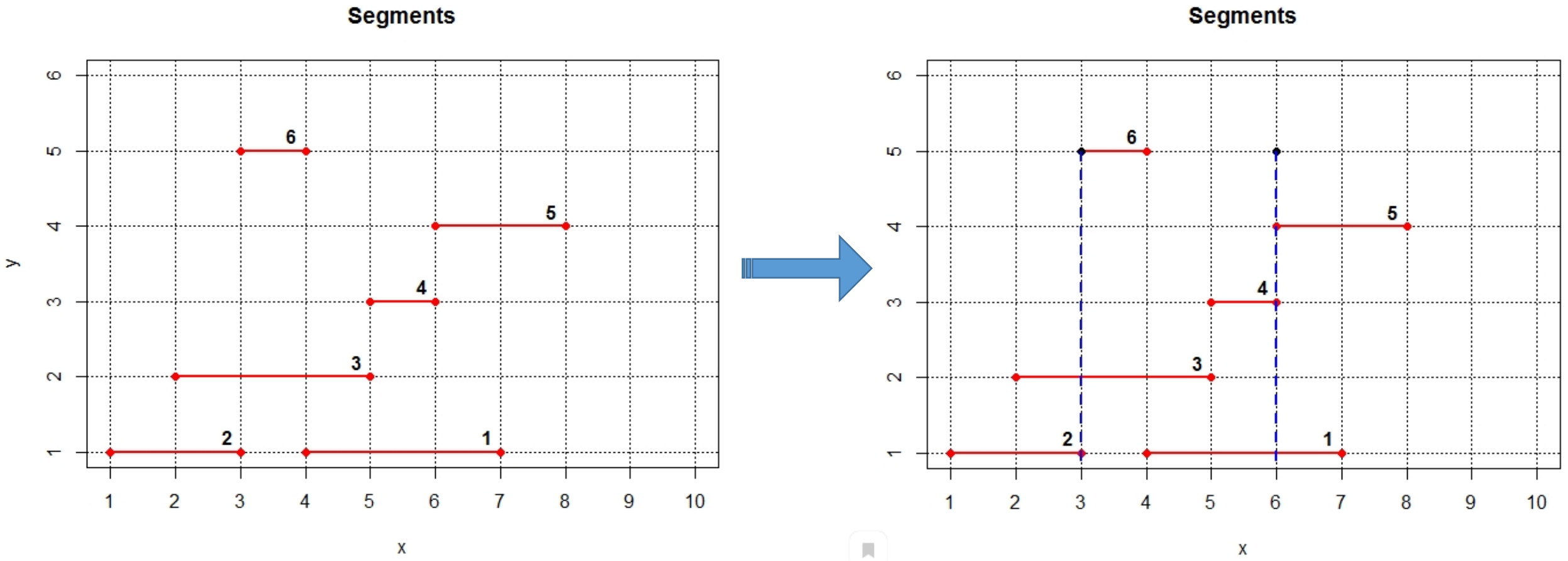
- Выполняются ли данные условия в рассматриваемой задаче?
- Правомерно ли решать эту задачу жадным алгоритмом?

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

using integer = long long unsigned int;

bool isNumbersOverBorder(integer a, integer b, integer border) {
    if (a > border || b > border) {
        return true;
    }
    return false;
}

int main() {
    int tests;
    std::cin >> tests;
    for (auto i = 0; i < tests; i++) {
        integer a, b, border;
        std::cin >> a >> b >> border;

        integer summ = 0;
        int count = 0;
        while (!isNumbersOverBorder(a, b, border)) {
            if (a < b) {
                a += b;
            } else {
                b += a;
            }
            count++;
        }
        std::cout << count << "\n";
    }
}
```

Что можно улучшить?

Жадная задача: покрытие отрезка точками

Условие: Дано *n* отрезков на прямой. Необходимо найти множество точек минимального размера, для которого каждый из отрезков содержит хотя бы одну из точек.



Алгоритм в общем виде:

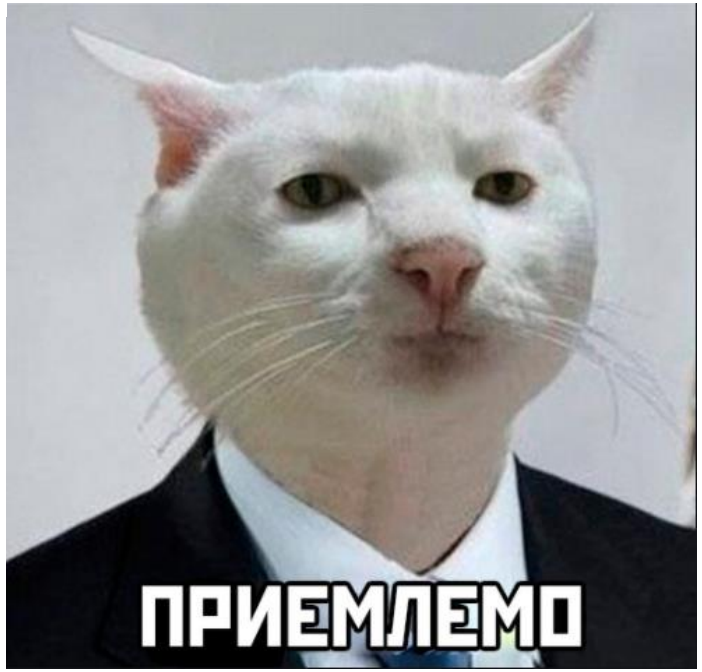
Пока не покрыты все отрезки:

находим минимальную правую границу

добавляем эту точку-границу в ответ

избавляемся от отрезков, в которые входит добавленная точка

Время работы: $O(n^2)$. **Как оценим?**



Но можно ли лучше?

Более эффективный алгоритм:

Сортируем отрезки по правому краю

Пока не покрыты все отрезки:

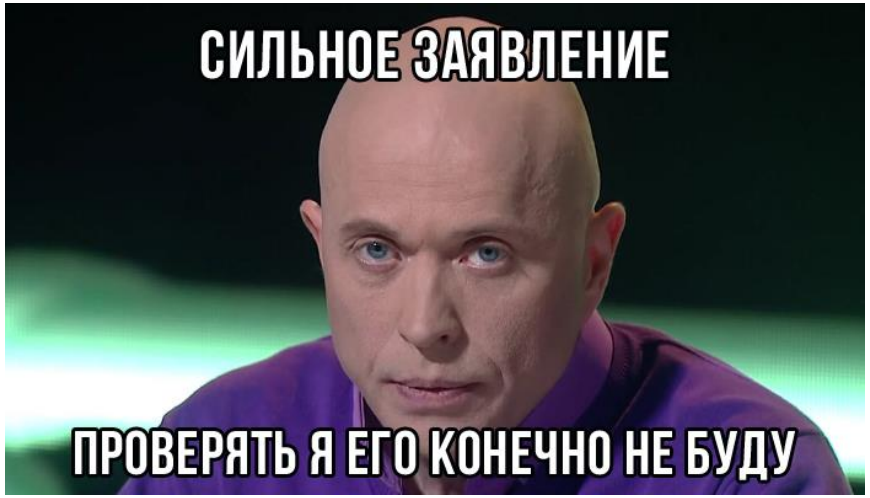
находим минимальную правую границу

добавляем эту точку-границу в ответ

избавляемся от отрезков, в которые входит добавленная точка

Время работы: $O(n^2)$. Но константы будут значительно лучше.

Также возможно добиться O(n log(n)).



Наивная реализация. Ключевые моменты.

Для удобства используем псевдонимы

```
using integer = long long unsigned int;
using Segment = std::pair<integer, integer>;
```

Чтение. Выделение буфера

```
std::vector<Segment> segments;
segments.resize(n);
for (auto i = 0; i < n; i++) {
    std::cin >> segments[i].first >> segments[i].second;
}
```

Удаление отрезков с точкой внутри

```
std::vector<integer> points;
while (!segments.empty()) {
    auto newPoint = segments.front().second;
    points.push_back(newPoint);

    auto it = std::find_if(std::cbegin(segments), std::cend(segments),
        [point=newPoint](auto segment) {
            if (segment.first > point)
                return true;
            return false;
    });
    if (it != std::cend(segments)) {
        segments.erase(std::cbegin(segments), it);
    } else {
        segments.clear();
    }
};
```

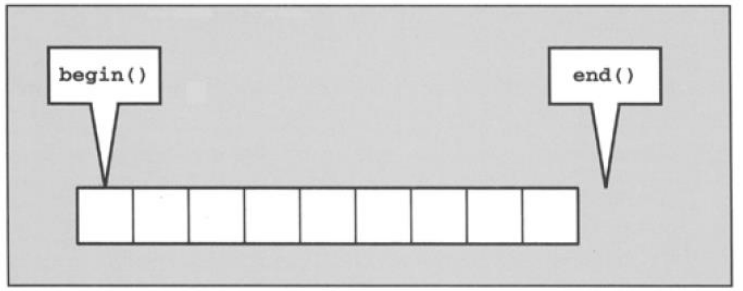
Сортировка отрезков по правому краю

```
std::sort(std::begin(segments), std::end(segments), [](auto lhs, auto rhs) {
    if (lhs.second < rhs.second)
        return true;
    return false;
});
```

Минутка C++

Рассмотрим типичного представителя алгоритма STL на примере std::find_if.

```
void remo  std::vector<Segment>::iterator it  Segment> &segments, integer point) {
    auto it = std::find_if(std::begin(segments), std::end(segments), [point](auto segment) {
        if (segment.first > point)
            return true;
        return false;
    });
```



- Первый аргумент – итератор на начало.
- Второй аргумент – итератор на элемент, за последним обрабатываемым.
- Третий аргумент – предикат – некий callable объект. В нашем случае используем шаблонную лямбда-функцию (*generic lambda C++14*).
- Возвращает итератор на первый найденный элемент, для которого предикат вернул true.

```
bool cmp(const Type1 &a, const Type2 &b);
```

```
if (it ≠ std::cend(segments)) {
    segments.erase(std::cbegin(segments), it);
} else {
    segments.clear();
}
```

Если ничего найдено не было – возвращается *end()*.

Приставка *c* означает *constant*, *r* – *reverse*.

Минутка философии

Если возникает интуитивно-стандартная задача поиска по критерию, удалению, изменению порядка, сортировки, модификации по правилу и тому подобного, то стараемся использовать **стандартное решение**.

- Уменьшается количество костылей (в чём мы более уверены: в алгоритме стандартной библиотеки или в самопальном велосипеде?)
- Вслед за этим уменьшается сложность и стоимость сопровождения (вообще само проектирование ПО и всякие методики программирования служат именно этой цели – уменьшение сложности)
- Заодно расширяем кругозор возможностей своих библиотек, не стесняемся гуглить (лучше по-английски)
- Есть варианты, когда библиотечной скорости не хватает, и нужно что-то побыстрее. Понимаем корень проблемы, локализуем её, обвешиваем тестами, делаем бенчмарки, одним словом – профилировка
- «Неважно, насколько код быстр, если он неправилен»

Форматированная реализация

Теперь функция main выглядит так:

```
19  int main() {
20      auto segments = readSegments(std::cin);
21      sortSegmentsByEnds(segments);
22
23      std::vector<integer> points;
24      while (!segments.empty()) {
25          auto newPoint = findGreedyNewPoint(segments);
26          points.push_back(newPoint);
27          removeSegmentWithPointInside(segments, newPoint);
28      };
29
30      printOptimalPointCovering(points);
31  }
```

- При необходимости внесения изменений в реализацию нужно тратить меньше умственных усилий. Сложность спрятана за ширмой.
- Легко тестировать отдельные функции
- Код читается как проза. Агоритм читается как псевдокод.

Пару слов о чтении данных

Поддерживайте модульность кода путем использования абстракций потока. Это позволит отвязать фрагменты исходного кода друг от друга и облегчит тестирование исходного кода, поскольку можно внедрить любой другой соответствующий тип потока.

std::cin и std::ifstream взаимозаменяемы. cin имеет тип std::istream, а std::ifstream наследует от std::istream.

```
30  std::vector<Segment> readSegments(std::istream &in) {
31      int n;
32      in >> n;
33      std::vector<Segment> segments;
34      segments.resize(n);
35
36      // псеводокод! Необходимо создать отдельный тип вместо п
37      std::istream_iterator<Segment> it{std::cin};
38      std::istream_iterator<Segment> end;
39      std::copy(it, end, std::back_inserter(segments));
40      return segments;
41  }
```

Также не забываем про возможность перенаправления потока в консоли:

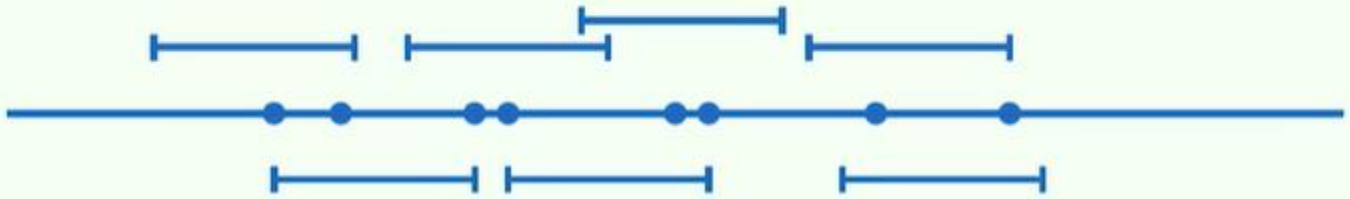
```
C:\Users\Максим\Desktop\pia
λ .\a.exe < test.txt
```

Есть ещё способ работать с потоками круче – но о нем попозже.

Другие типичные жадные задачи

Вообще задача о покрытии отрезков точками имеет более популярную вариацию – **задачу о покрытии точек отрезками**. Естественно, также минимальным их количеством.

Пример



Ключ к решению – в оптимальном покрытии (которое мы и ищем) самая левая точка покрыта левым концом отрезка.

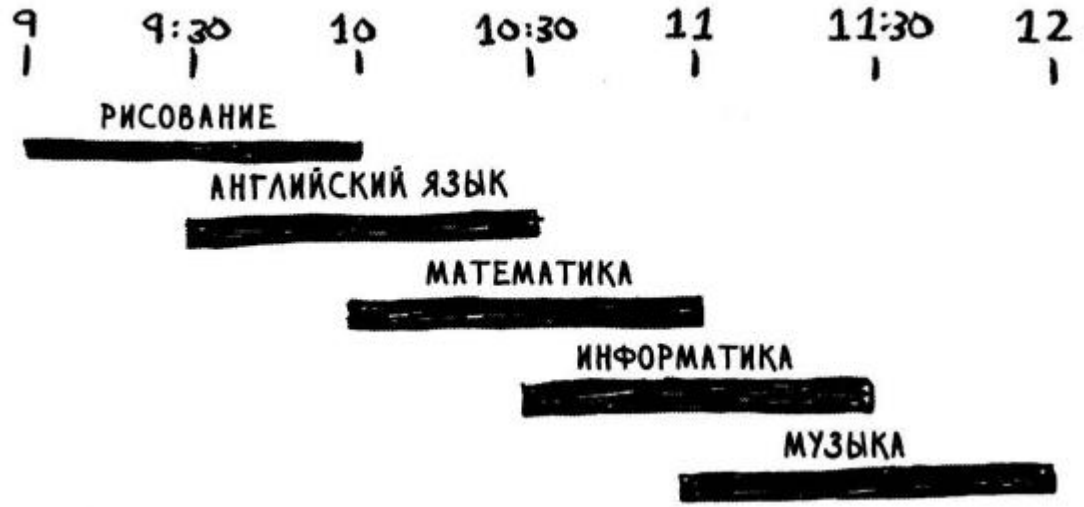
Более подробно здесь: Алгоритмы: теория и практика. Методы
Computer Science Center (CS центр)
Полное объяснение там же.

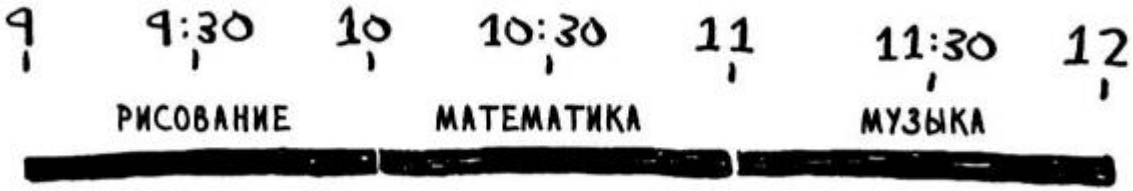
Другие типичные жадные задачи

Другая классика имеет название **«Задача о составления расписания».**

Суть: есть кабинет и несколько заявок на проведение занятий в этом кабинете, которые пересекаются. В один момент времени можно проводить одно занятие. Требуется максимизировать количество занятий.

Задача эта также решается жадным алгоритмом подобным же образом. И также мы получаем глобально-оптимальное решение. Какой принцип?





Другие типичные жадные задачи

Алгоритм Борувки – один аз алгоритмов построения MST (минимальное остовное дерево).

Является ли этот алгоритм жадным?

Алгоритм состоит из нескольких шагов:

1. Изначально каждая вершина графа G— тривиальное дерево, а ребра не принадлежат никакому дереву.
2. Для каждого дерева T найдем минимальное инцидентное ему ребро. Добавим все такие ребра.
3. Повторяем шаг 2 пока в графе не останется только одно дерево T.

Да!

Другие типичные жадные задачи

С ещё одной жадной задачей мы уже встречались…

Построение кода Хаффмана!

Задача: построить оптимальный беспрефиксный код (переменной длины).

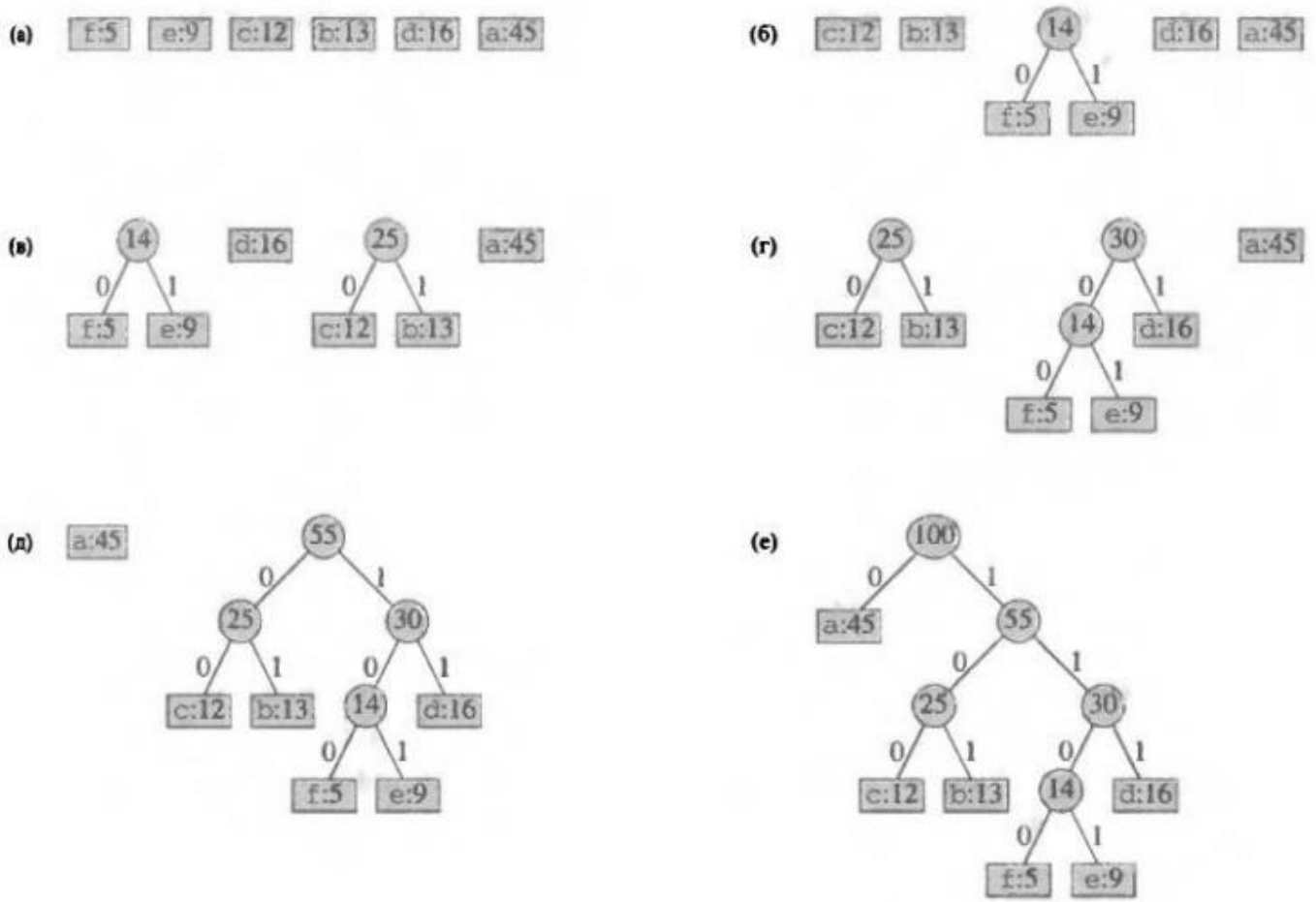
Вопрос: Где здесь жадность? Почему она работает?

Интуитивный ответ: Оптимальный код представлен полным бинарным деревом. Строим такое двоичное дерево снизу вверх, на каждом шаге «сливая» вместе наименее встречающиеся соседние отсортированное по весу встречаемости символы (вот она, однозначность выбора на каждом шаге) в один «символ». Переупорядочиваем и идём дальше, пока не получим один большой «символ», вес которого будет равен общему количеству символов в тексте.

Строгий ответ: в книге у Кормена с ~~блекджеком~~ теоремами и док-вами.

ЖАДНЫЙ КРИТЕРИЙ ХАФФМАНА

Слияние пары деревьев приводит к минимально возможному увеличению средней глубины листа.



- Жадный алгоритм Хаффмана поддерживает лес, где листья находятся в соответствии с символами алфавита, и на каждой итерации жадно выполняет слияние пары деревьев, вызывая минимально возможное увеличение средней глубины листа.
- Алгоритм Хаффмана гарантированно вычисляет беспрефиксный код с минимально возможной средней длиной кодирования.
- Алгоритм Хаффмана может быть реализован с работой за время $O(n \log n)$, где n — это число символов.

Другие типичные жадные задачи

Есть ещё:

- Задача о рюкзаке (непрерывном)
- Нахождение кратчайшего пути (вариации и размышления)

Здесь ест нюансы, поэтому мы разберём их на другой теме.

Выводы

- Жадные алгоритмы конструируют решения итеративно, посредством последовательности близоруких решений в надежде, что в конце полученные решение будет оптимальным
- Жадные алгоритмы не всегда являются правильными
- Важно проверять выполнение обоих условий, но это не всегда бывает тривиально

Ссылки и источники

https://stepik.org/course/217/

Томас Кормен
Чарльз Лейзерсон
Рональд Ривест
Клиффорд Штайн

АЛГОРИТМЫ
ПОСТРОЕНИЕ И АНАЛИЗ

Упражнения

Дописать фрагменты и собрать воедино решённую задачу о покрытии отрезков точками. Проверить своё решение можно в курсе на степике (4.1.9).

Также в исходнике на гитхабе есть некоторые пояснения. Как её можно улучшить?

Прочитать про erase-remove idiom.

Greedy Florist ★

A group of friends want to buy a bouquet of flowers. The florist wants to maximize his number of new customers and the money he makes. To do this, he decides he'll multiply the price of each flower by the number of that customer's previously purchased flowers plus 1. The first flower will be original price, $(0 + 1) \times \textbf{original price}$, the next will be $(1 + 1) \times \textbf{original price}$ and so on.

Given the size of the group of friends, the number of flowers they want to purchase and the original prices of the flowers, determine the minimum cost to purchase all of the flowers.

For example, if there are $k = 3$ friends that want to buy $n = 4$ flowers that cost $c = [1, 2, 3, 4]$ each will buy one of the flowers priced $[2, 3, 4]$ at the original price. Having each purchased $x = 1$ flower, the first flower in the list, $c[0]$, will now cost $(current\ purchase + previous\ purchases) \times c[0] = (1 + 1) \times 1 = 2$. The total cost will be $2 + 3 + 4 + 2 = 11$.