

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?

Задачка – 1368A – «C+=»

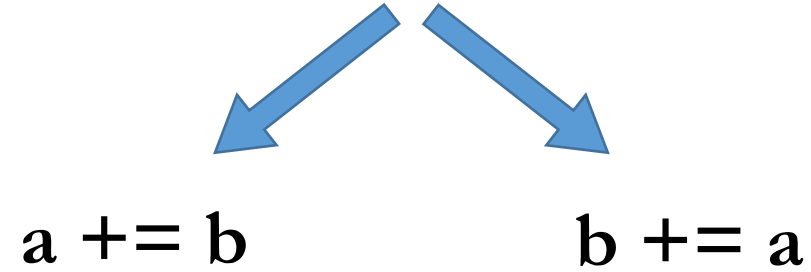
Лео создал новый язык программирования C+=. В C+= целочисленные переменные можно изменять только операцией «+=», которая прибавляет значение справа к переменной слева. Например, если выполнить «a += b», когда a = 2, b = 3, значение a станет равно 5 (значение b при этом не изменится).

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:

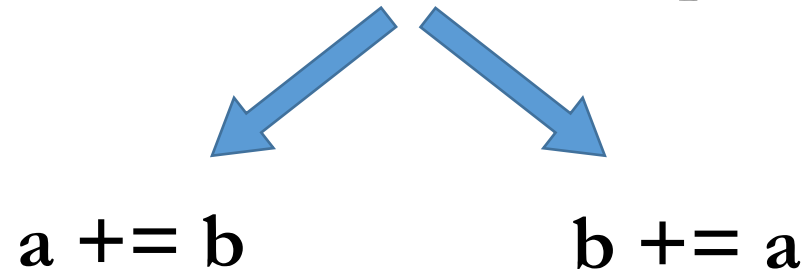
- Может быть много возможных решений
- Качество решения определяется некоторым параметром
- Требуется выбрать среди них одно оптимальное решение

Возможны всего 2 операции



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

Возможны всего 2 операции



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:

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

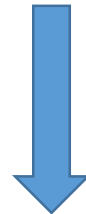
Итак, на каждом шаге жадного алгоритма из всех возможных вариантов выбирается самый оптимальный на данном конкретном шаге.

При этом:

- Прошрое 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 оптимальности для подзадач**, т.е. оптимальное решение всей задачи обязательно содержит в себе оптимальные решения подзадач, из которых она состоит.

Данная логика работает только при условии, что решаемая задача обладает **свойством оптимальности 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**

Данная логика работает только при условии, что решаемая задача обладает **свойством оптимальности 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?

```

1  #include <iostream>
2  #include <vector>
3
4  using integer = long long unsigned int;
5
6  bool isNumbersOverBorder(integer a, integer b, integer border) {
7      if (a > border || b > border) {
8          return true;
9      }
10     return false;
11 }
12
13 int main() {
14     int tests;
15     std::cin >> tests;
16     for (auto i = 0; i < tests; i++) {
17         integer a, b, border;
18         std::cin >> a >> b >> border;
19
20         integer summ = 0;
21         int count = 0;
22         while (!isNumbersOverBorder(a, b, border)) {
23             if (a < b) {
24                 a += b;
25             } else {
26                 b += a;
27             }
28             count++;
29         }
30         std::cout << count << "\n";
31     }
32 }

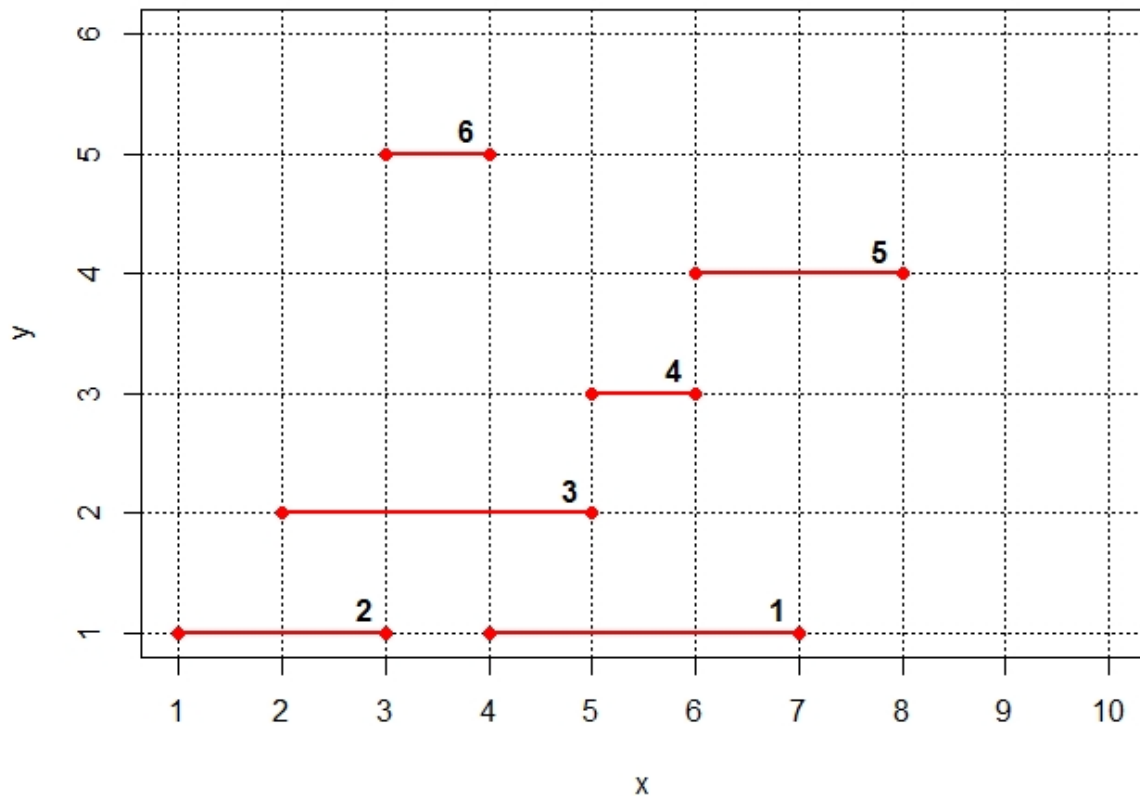
```

**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.

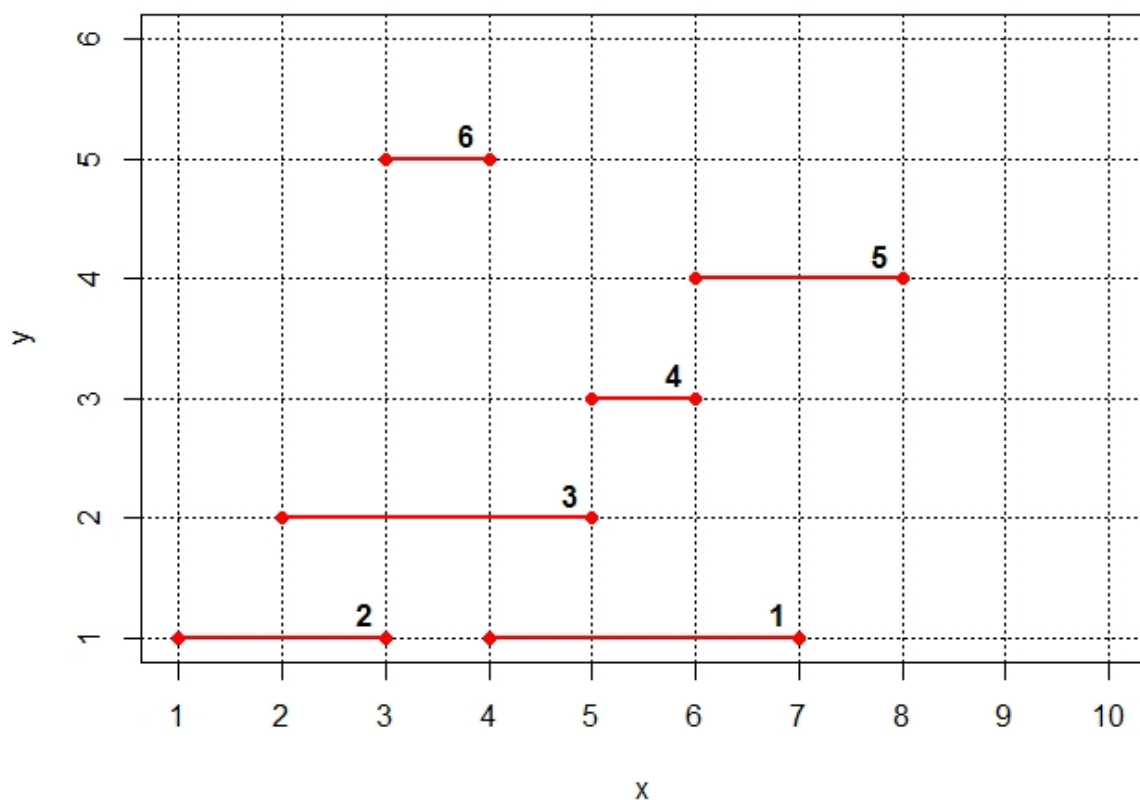
Segments



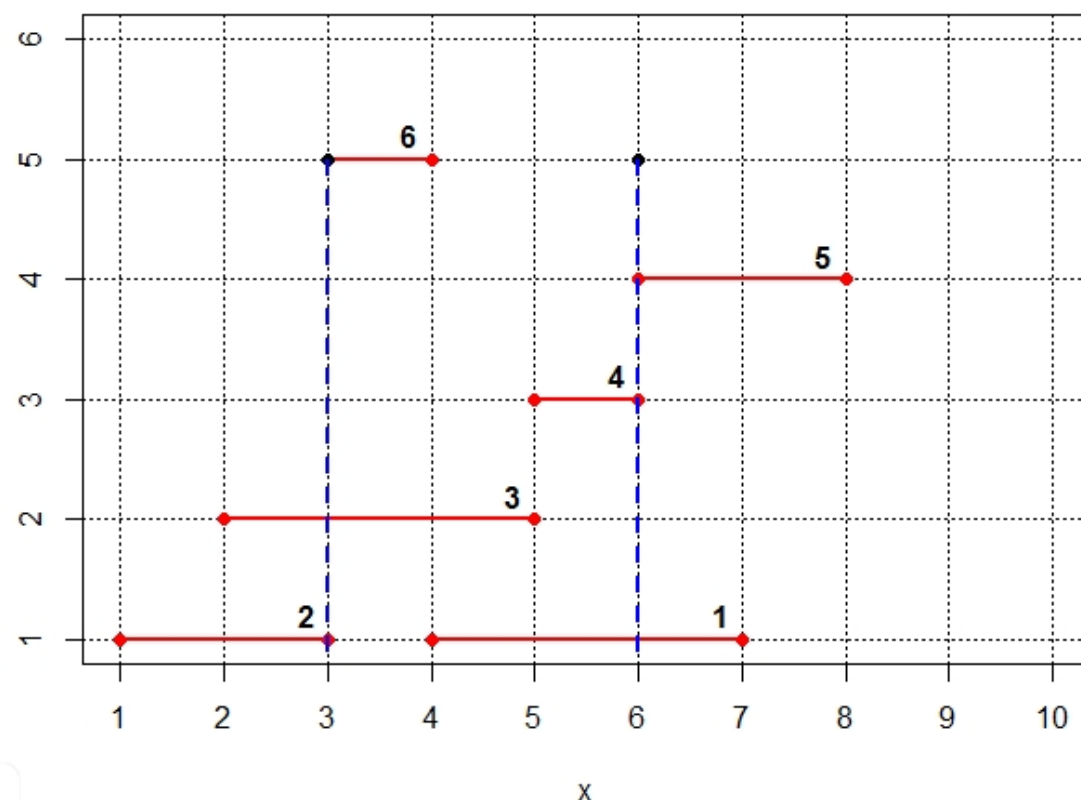
Жадная задача: покрытие отрезка точками

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

Segments



Segments



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

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?

Алгоритм в общем виде:

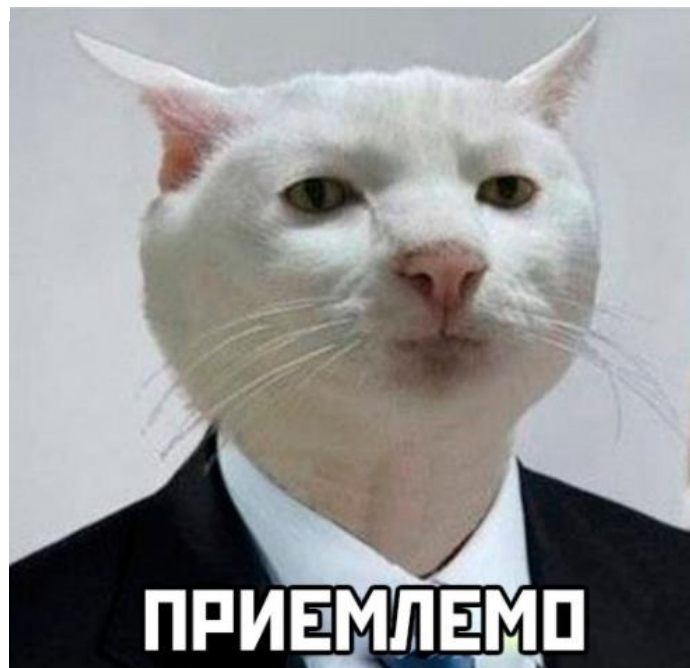
Пока не покрыты все отрезки:

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?



Алгоритм в общем виде:

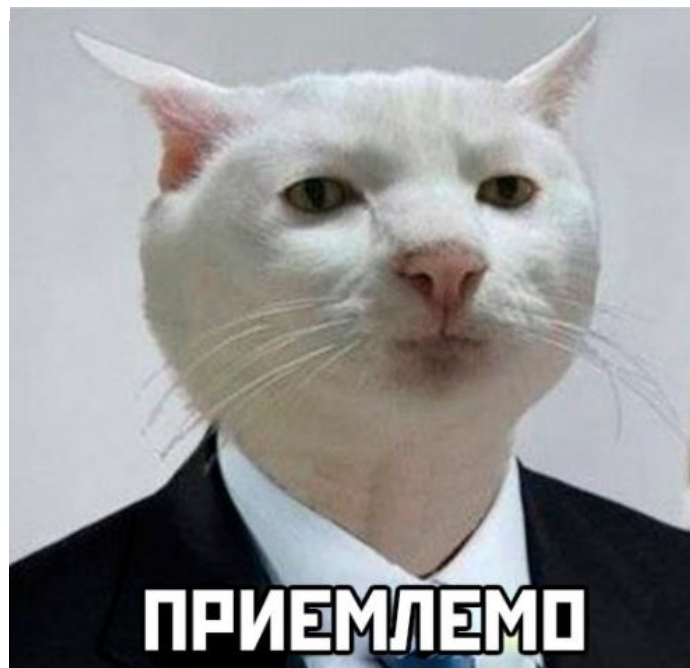
Пока не покрыты все отрезки:

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:

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: $O(n^2)$.

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.

Более эффективный алгоритм:

Сортируем отрезки по правому краю

Пока не покрыты все отрезки:

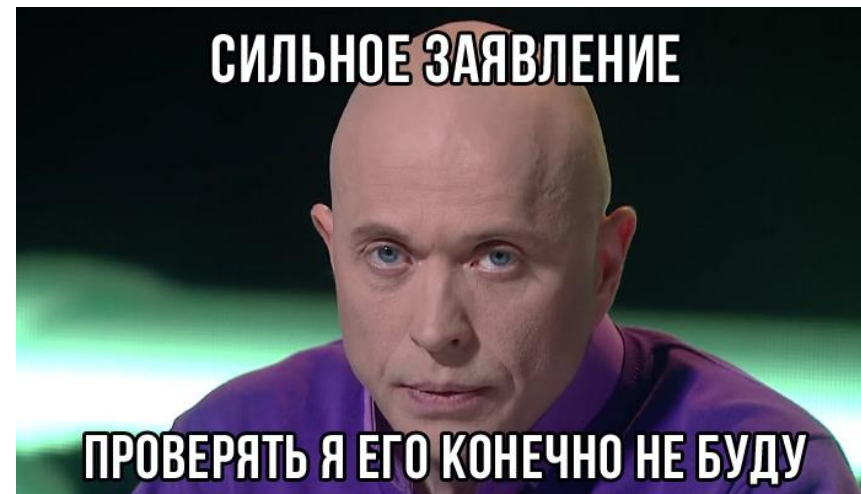
находим минимальную правую границу

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>;
```

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;  
}
```

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();
    }
};
```


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();
    }
};
```

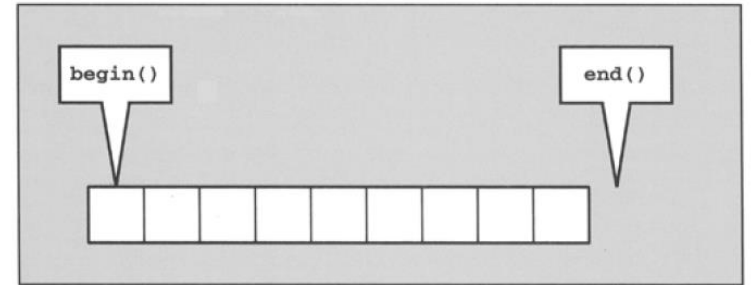
Сортировка отрезков по правому краю

```
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`.

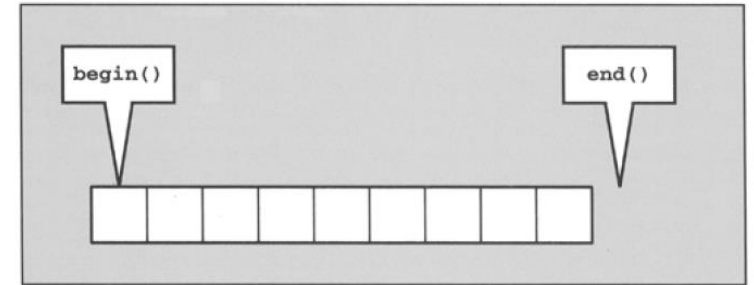
```
void remove_if(std::vector<Segment>::iterator it, Segment &segments, integer point) {  
    auto it = std::find_if(std::begin(segments), std::end(segments), [point](auto segment) {  
        if (segment.first > point)  
            return true;  
        return false;  
    });  
}
```



Минутка C++

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

```
void remove_if(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;  
    });  
}
```



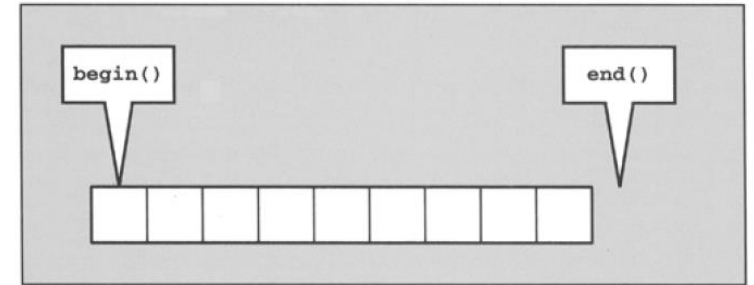
- Первый аргумент – итератор на начало.
- Второй аргумент – итератор на элемент, за последним обрабатываемым.
- Третий аргумент – предикат – некий callable объект. В нашем случае используем шаблонную лямбда-функцию (*generic lambda C++14*).
- Возвращает итератор на первый найденный элемент, для которого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

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

Минутка C++

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

```
void remove_if(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;  
    });  
    segments.erase(it);  
}
```



- Первый аргумент — итератор на начало.
- Вторым аргументом — 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.

```
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 }
```

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. Алгоритм 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.



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

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

Задача о рюкзаке (непрерывном)

Нахождение кратчайшего пути (вариации и размышления)

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