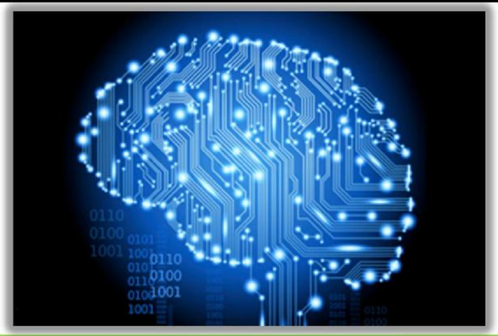
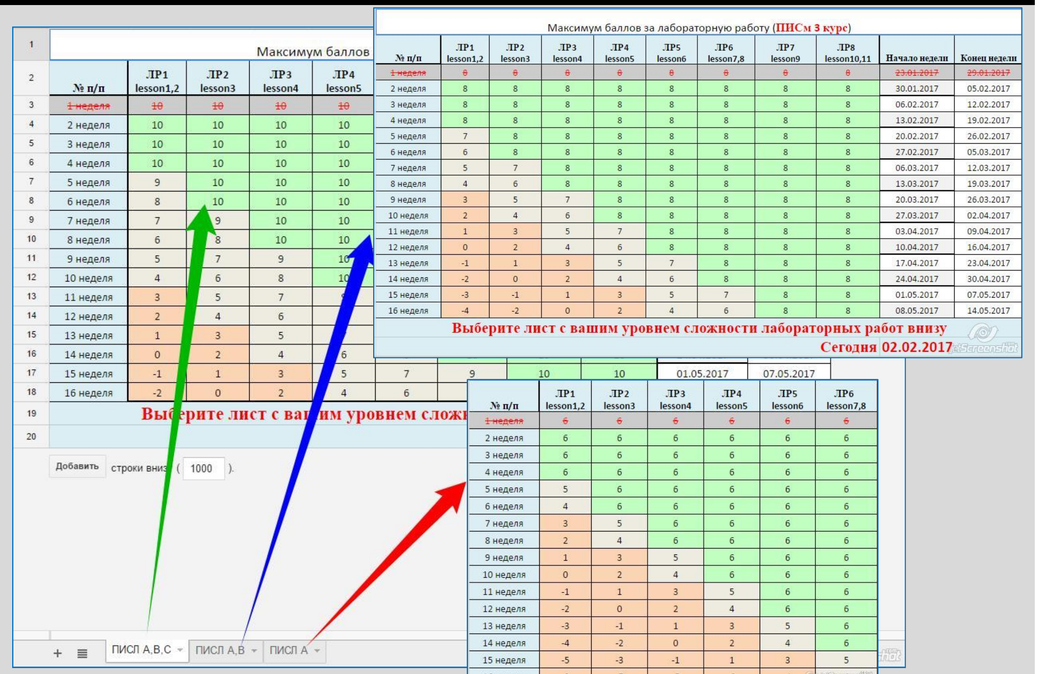
2017

PISL 02

Жадные алгоритмы. Введение.



Оценки за задания



Максимум баллов за лабораторную работу (ПИСм 3 курс)

№ п/п	ЛР1 lesson1,2	ЛР2 lesson3	ЛР3 lesson4	ЛР4 lesson5	ЛР5 lesson6	ЛР6 lesson7,8	ЛР7 lesson9	ЛР8 lesson10,11	Начало недели	Конец недели
~~1 неделя~~	~~8~~	~~8~~	~~8~~	~~8~~	~~8~~	~~8~~	~~8~~	~~8~~	~~23.01.2017~~	~~29.01.2017~~
2 неделя	8	8	8	8	8	8	8	8	30.01.2017	05.02.2017
3 неделя	8	8	8	8	8	8	8	8	06.02.2017	12.02.2017
4 неделя	8	8	8	8	8	8	8	8	13.02.2017	19.02.2017
5 неделя	7	8	8	8	8	8	8	8	20.02.2017	26.02.2017
6 неделя	6	8	8	8	8	8	8	8	27.02.2017	05.03.2017
7 неделя	5	7	8	8	8	8	8	8	06.03.2017	12.03.2017
8 неделя	4	6	8	8	8	8	8	8	13.03.2017	19.03.2017
9 неделя	3	5	7	8	8	8	8	8	20.03.2017	26.03.2017
10 неделя	2	4	6	8	8	8	8	8	27.03.2017	02.04.2017
11 неделя	1	3	5	7	8	8	8	8	03.04.2017	09.04.2017
12 неделя	0	2	4	6	8	8	8	8	10.04.2017	16.04.2017
13 неделя	-1	1	3	5	7	8	8	8	17.04.2017	23.04.2017
14 неделя	-2	0	2	4	6	8	8	8	24.04.2017	30.04.2017
15 неделя	-3	-1	1	3	5	7	8	8	01.05.2017	07.05.2017
16 неделя	-4	-2	0	2	4	6	8	8	08.05.2017	14.05.2017

Выберите лист с вашим уровнем сложности лабораторных работ внизу

Сегодня 02.02.2017

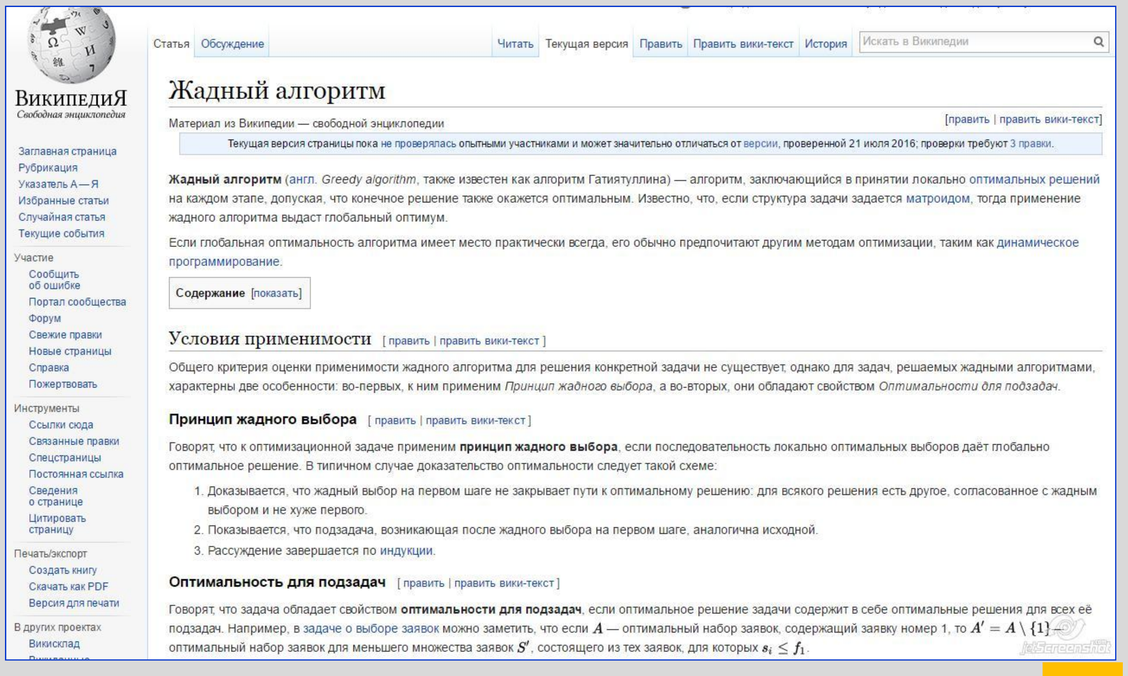
Добавить строки вниз (1000).

ПИСЛ А,В,С | ПИСЛ А,В | ПИСЛ А

№ п/п	ЛР1 lesson1,2	ЛР2 lesson3	ЛР3 lesson4	ЛР4 lesson5	ЛР5 lesson6	ЛР6 lesson7,8
~~1 неделя~~	~~6~~	~~6~~	~~6~~	~~6~~	~~6~~	~~6~~
2 неделя	6	6	6	6	6	6
3 неделя	6	6	6	6	6	6
4 неделя	6	6	6	6	6	6
5 неделя	5	6	6	6	6	6
6 неделя	4	6	6	6	6	6
7 неделя	3	5	6	6	6	6
8 неделя	2	4	6	6	6	6
9 неделя	1	3	5	6	6	6
10 неделя	0	2	4	6	6	6
11 неделя	-1	1	3	5	6	6
12 неделя	-2	0	2	4	6	6
13 неделя	-3	-1	1	3	5	6
14 неделя	-4	-2	0	2	4	6
15 неделя	-5	-3	-1	1	3	5
16 неделя	-9	-7	-5	-3	-1	3

- Определение.
- Примеры жадных алгоритмов.
- Программируем камеру наблюдения (покрытие точек отрезками) (задача A)
  - наивный алгорит
  - более быстрый алгорит
- Рассчитаем расписание аудитории (задача о выборе заявок) (задача B)
  - наивный алгорит
  - более быстрый алгорит
- Планируем новогодний корпоратив (задача о независимом множестве в деревьях)
- Задача о непрерывном рюкзаке (задача C)

- **Материалы:** http://tinyurl.com/ei-pisl
- **Github:** https://github.com/Khmelov/PISL2017-01-26



ВикипедиЯ
Свободная энциклопедия

Статья | Обсуждение | Читать | Текущая версия | Править | Править вики-текст | История | Искать в Википедии

Жадный алгоритм

Материал из Википедии — свободной энциклопедии [править | править вики-текст]

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 21 июля 2016; проверки требуют 3 правки.

Жадный алгоритм (англ. *Greedy algorithm*, также известен как алгоритм Гатиятуллина) — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным. Известно, что, если структура задачи задается матроидом, тогда применение жадного алгоритма выдаст глобальный оптимум.

Если глобальная оптимальность алгоритма имеет место практически всегда, его обычно предпочитают другим методам оптимизации, таким как динамическое программирование.

Содержание [показать]

Условия применимости [править | править вики-текст]

Общего критерия оценки применимости жадного алгоритма для решения конкретной задачи не существует; однако для задач, решаемых жадными алгоритмами, характерны две особенности: во-первых, к ним применим *Принцип жадного выбора*, а во-вторых, они обладают свойством *Оптимальности для подзадач*.

Принцип жадного выбора [править | править вики-текст]

Говорят, что к оптимизационной задаче применим **принцип жадного выбора**, если последовательность локально оптимальных выборов даёт глобально оптимальное решение. В типичном случае доказательство оптимальности следует такой схеме:

1. Доказывается, что жадный выбор на первом шаге не закрывает пути к оптимальному решению: для всякого решения есть другое, согласованное с жадным выбором и не хуже первого.
2. Показывается, что подзадача, возникающая после жадного выбора на первом шаге, аналогична исходной.
3. Рассуждение завершается по индукции.

Оптимальность для подзадач [править | править вики-текст]

Говорят, что задача обладает свойством **оптимальности для подзадач**, если оптимальное решение задачи содержит в себе оптимальные решения для всех её подзадач. Например, в задаче о выборе заявок можно заметить, что если A — оптимальный набор заявок, содержащий заявку номер 1, то $A' = A \setminus \{1\}$ — оптимальный набор заявок для меньшего множества заявок S', состоящего из тех заявок, для которых $s_i \leq f_1$.

Покрытие точек отрезками

Вход: множество n точек на прямой $x_1, \ldots, x_n \in \mathbb{R}$.

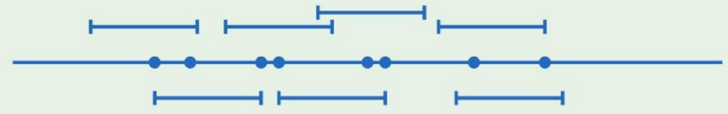
Выход: минимальное количество отрезков единичной длины, которыми можно покрыть все точки.

```
public static void main(String[] args) {
    VideoRegistrator instance=new VideoRegistrator();
    double[] events=new double[]{1, 1.1, 1.6, 2.2, 2.4, 2.7, 3.9, 8.1, 9.1, 5.5, 3.7};
    List<Double> starts=instance.calcStartTimes(events,1);
    System.out.println(starts);
}
```


Пример

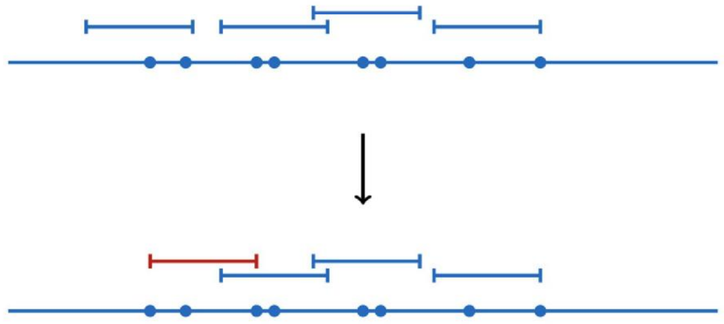
Пример

Пример



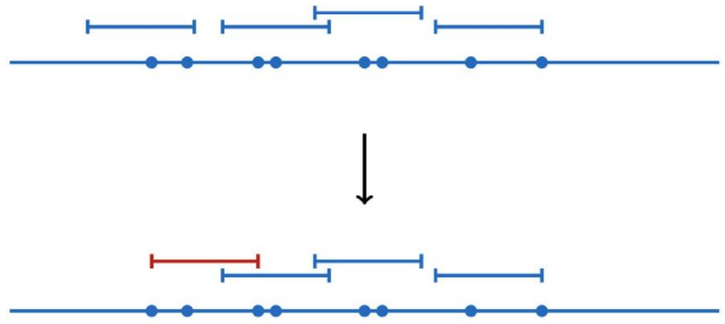
Надёжный шаг

Существует оптимальное покрытие, в котором самая левая точка покрыта левым концом отрезка.



Надёжный шаг

Существует оптимальное покрытие, в котором самая левая точка покрыта левым концом отрезка.



Поэтому можно сразу добавить в решение отрезок, левый конец которого совпадает с самой левой точкой.

Алгоритм

Функция POINTSCOVER$(x_1, \ldots, x_n)$

$S \leftarrow \{x_1, \ldots, x_n\}$
пока S не пусто:
  $x_m \leftarrow$ минимальная точка S
  добавить к решению отрезок $[\ell, r] = [x_m, x_m + 1]$
  выкинуть из S точки, покрытые отрезком $[\ell, r]$
вернуть построенное решение

Алгоритм

Функция POINTSCOVER$(x_1, \ldots, x_n)$

$S \leftarrow \{x_1, \ldots, x_n\}$
пока S не пусто:
  $x_m \leftarrow$ минимальная точка S
  добавить к решению отрезок $[\ell, r] = [x_m, x_m + 1]$
  выкинуть из S точки, покрытые отрезком $[\ell, r]$
вернуть построенное решение

Время работы: $O(n^2)$.

Улучшенный алгоритм

Функция POINTSCOVER$(x_1, \ldots, x_n)$

$x_1, \ldots, x_n \leftarrow$ SORT$(x_1, \ldots, x_n)$
$i \leftarrow 1$
пока $i \le n$:
  добавить к решению отрезок $[\ell, r] = [x_i, x_i + 1]$
  $i \leftarrow i + 1$
  пока $i \le n$ и $x_i \le r$:
    $i \leftarrow i + 1$
вернуть построенное решение

Улучшенный алгоритм

Функция POINTSCOVER$(x_1, \ldots, x_n)$

$x_1, \ldots, x_n \leftarrow$ SORT$(x_1, \ldots, x_n)$
$i \leftarrow 1$
пока $i \le n$:
  добавить к решению отрезок $[\ell, r] = [x_i, x_i + 1]$
  $i \leftarrow i + 1$
  пока $i \le n$ и $x_i \le r$:
    $i \leftarrow i + 1$
вернуть построенное решение

Время работы: $T(\text{SORT}) + O(n) = O(n \log n)$.

Пример

Пример

Пример

Пример


```
public class VideoRegistrator {

    public static void main(String[] args) {
        VideoRegistrator instance=new VideoRegistrator();
        double[] events=new double[]{1, 1.1, 1.6, 2.2, 2.4, 2.7, 3.9, 8.1, 9.1, 5.5, 3.7};
        List<Double> starts=instance.calcStartTimes(events,1);
        System.out.println(starts);
    }

    private List<Double> calcStartTimes(double[] events, double workDuration){
        //events - события которые нужно зарегистрировать
        //timeWorkDuration время работы видеокамеры после старта
        List<Double> result;
        result = new ArrayList<>();
        int i=0;                                //i - это индекс события events[i]
        [redacted]                              //подготовка к жадному поглощению массива событий
                                                //hint: сортировка Arrays.sort обеспечит скорость алгоритма
                                                //C*(n log n) + C1*n = O(n log n)

        [redacted]                              //пока есть незарегистрированные события
            [redacted]                            //получим одно событие по левому краю
            [redacted]                            //и запомним время старта видеокамеры
            [redacted]                            //вычислим момент окончания работы видеокамеры
            [redacted]                            //и теперь пропустим все покрываемые события
                [redacted]                        //за время до конца работы, увеличивая индекс
                [redacted]
            [redacted]
        [redacted]
        return result;
    }
}
```

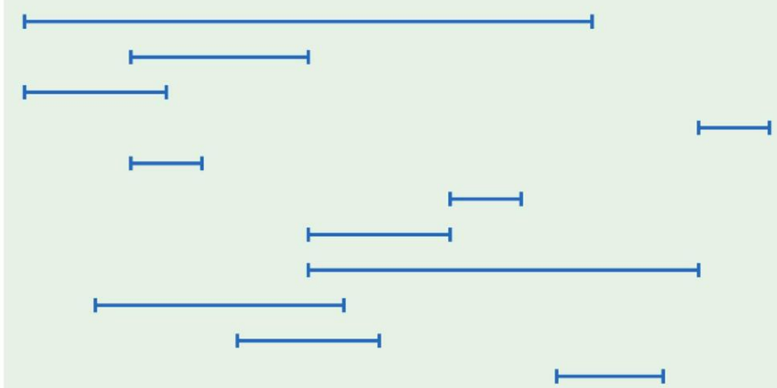
```
Run  VideoRegistrator
     "H:\Program Files\Java\jdk1.8.0_101\bin\java" ...
     [1.0, 2.2, 3.7, 5.5, 8.1]

     Process finished with exit code 0
```


Задача о выборе заявок

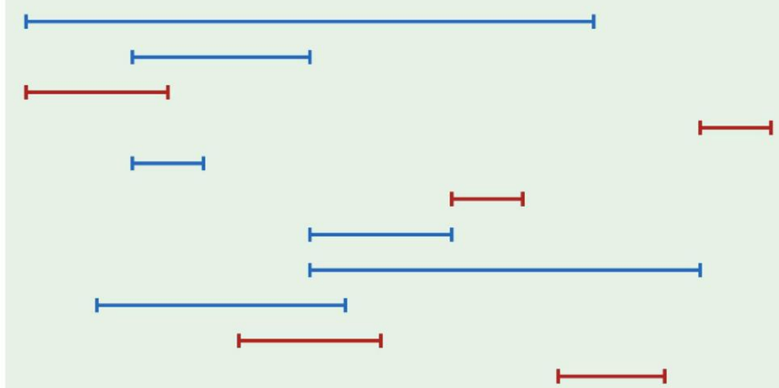
Вход: множество n отрезков на прямой.

Выход: максимальное количество попарно не пересекающихся отрезков.

Пример



Пример

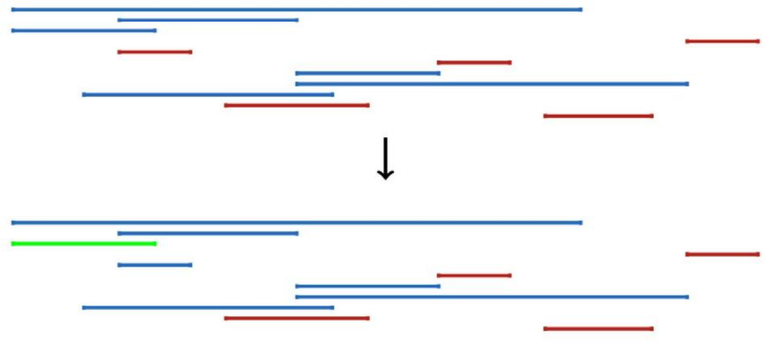


Замечание

Выбирая в первую очередь более короткие отрезки, можно получить неоптимальное решение.

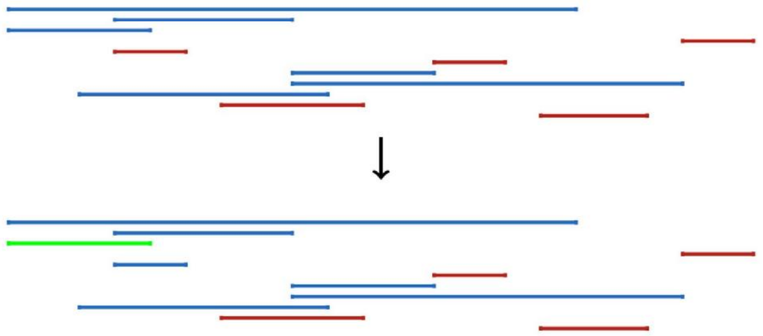
Надёжный шаг

Существует оптимальное решение, содержащее отрезок, правый конец которого минимален.



Надёжный шаг

Существует оптимальное решение, содержащее отрезок, правый конец которого минимален.



Можно сразу добавить в решение отрезок, правый конец которого минимален.

Алгоритм

Функция ACTSEL$(\ell_1, r_1, \ldots, \ell_n, r_n)$

$S \leftarrow \{[\ell_1, r_1], \ldots, [\ell_n, r_n]\}$
пока S не пусто:
  $[\ell_m, r_m] \leftarrow$ отрезок из S с мин. правым концом
  добавить $[\ell_m, r_m]$ к решению
  выкинуть из S отрезки, пересекающиеся с $[\ell_m, r_m]$
вернуть построенное решение

Алгоритм

Функция ACTSEL$(\ell_1, r_1, \ldots, \ell_n, r_n)$

$S \leftarrow \{[\ell_1, r_1], \ldots, [\ell_n, r_n]\}$
пока S не пусто:
  $[\ell_m, r_m] \leftarrow$ отрезок из S с мин. правым концом
  добавить $[\ell_m, r_m]$ к решению
  выкинуть из S отрезки, пересекающиеся с $[\ell_m, r_m]$
вернуть построенное решение

Время работы: $O(n^2)$.

Улучшенный алгоритм

Функция ACTSEL$(\ell_1, r_1, \ldots, \ell_n, r_n)$

отсортировать n отрезков по правым концам
для всех отрезков в полученном порядке:
  если текущий отрезок не пересекает
    последний добавленный:
    взять его в решение
вернуть построенное решение

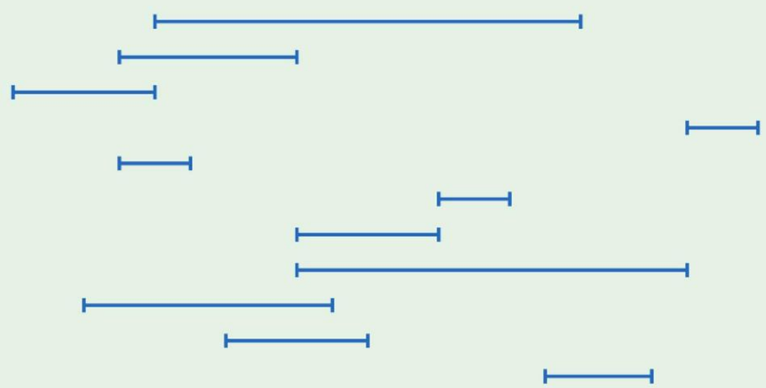
Улучшенный алгоритм

Функция ACTSEL$(\ell_1, r_1, \ldots, \ell_n, r_n)$

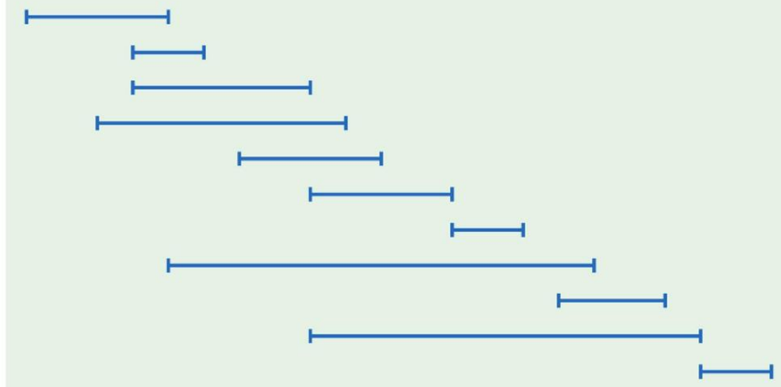
```
отсортировать n отрезков по правым концам
для всех отрезков в полученном порядке:
  если текущий отрезок не пересекает
    последний добавленный:
    взять его в решение
вернуть построенное решение
```

Время работы: $T(\text{SORT}) + O(n) = O(n \log n)$.

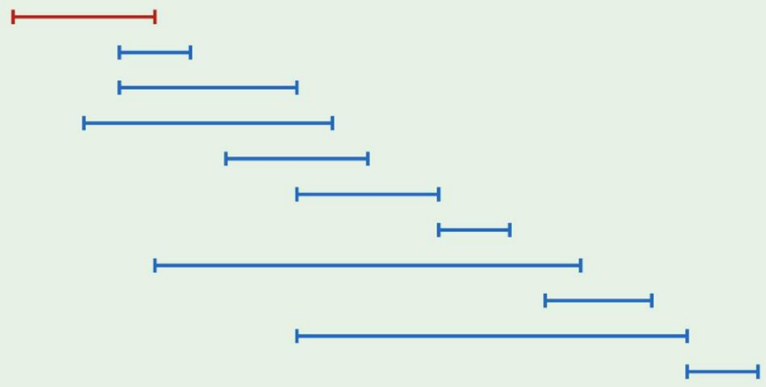
Пример



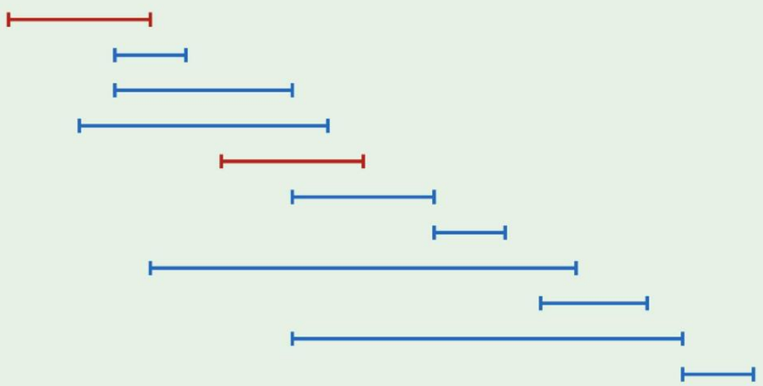
Пример



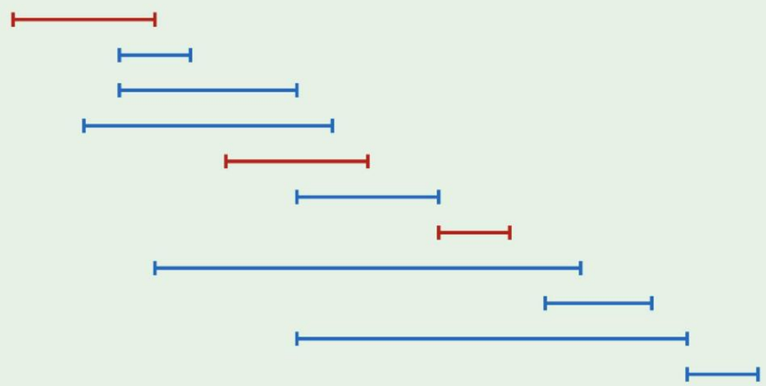
Пример



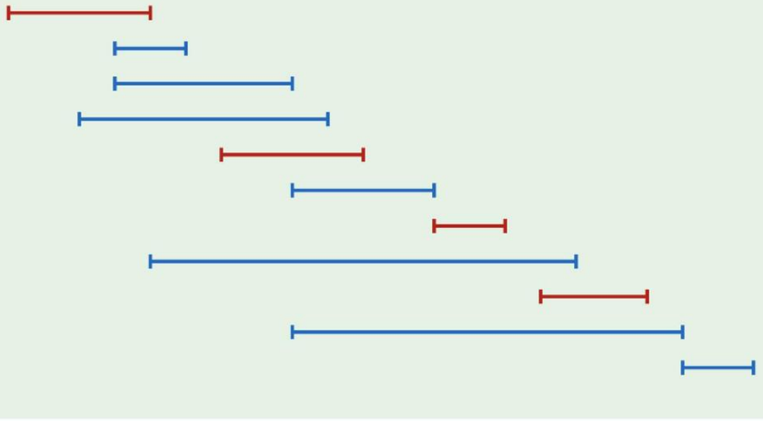
Пример



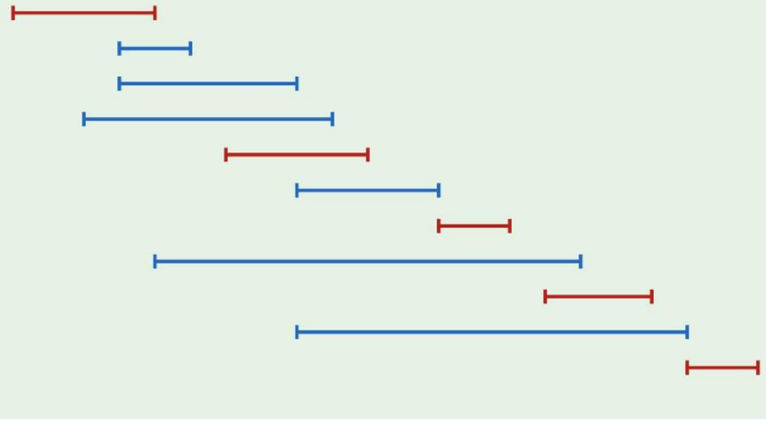
Пример



Пример



Пример

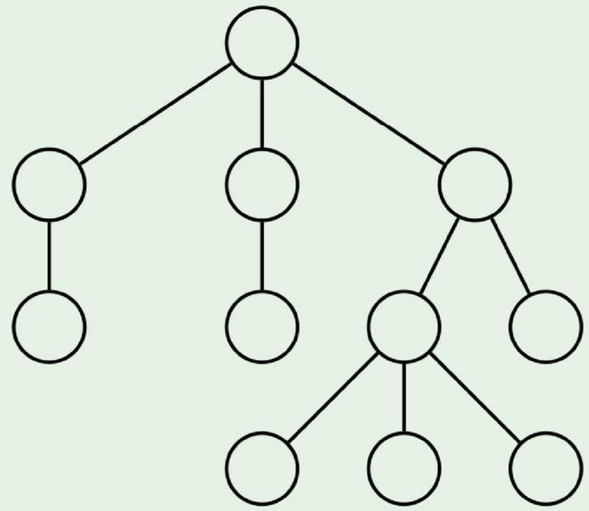


Планирование вечеринки в компании

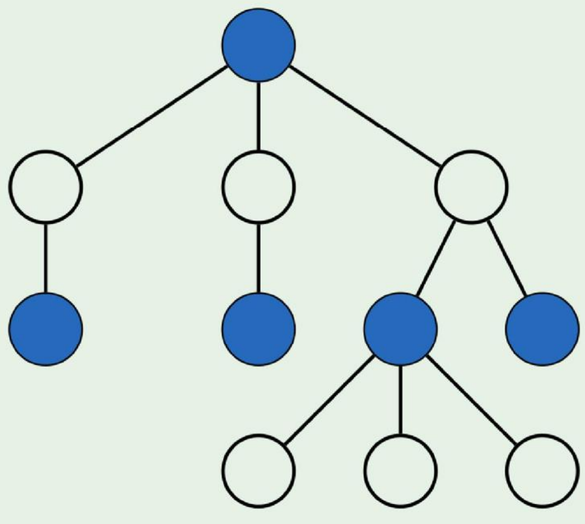
Вход: дерево.

Выход: независимое множество (множество не соединённых друг с другом вершин) максимального размера.

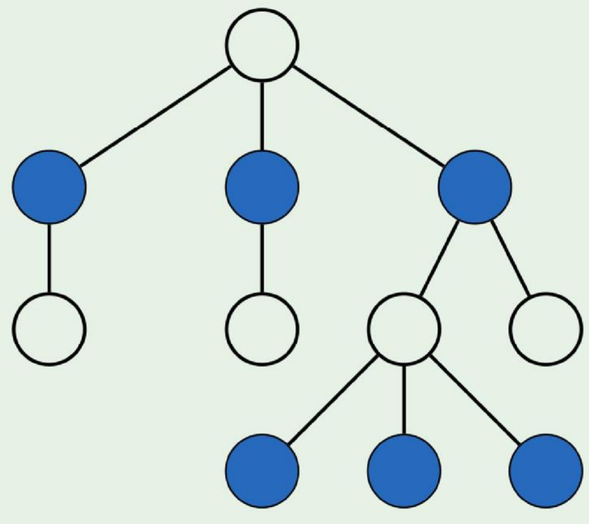
Пример



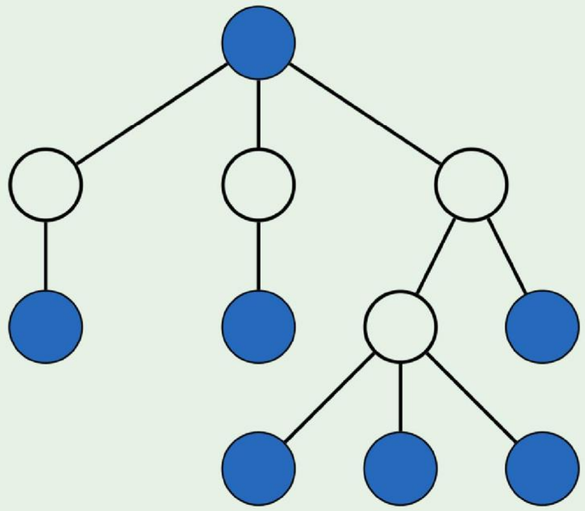
Пример



Пример

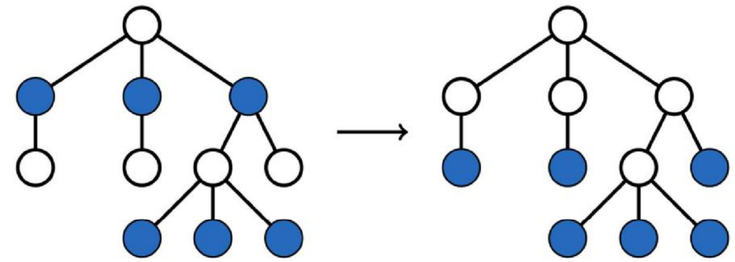


Пример



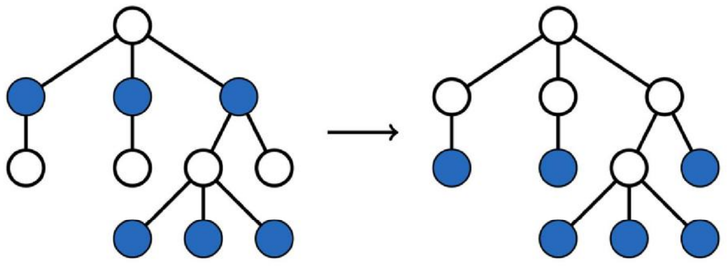
Надёжный шаг

Существует оптимальное решение, содержащее каждый лист дерева.



Надёжный шаг

Существует оптимальное решение, содержащее каждый лист дерева.



Можно взять в решение все листья.

Алгоритм

Функция MAXINDEPENDENTSET(T)

```
пока T не пусто:
  взять в решение все листья
  выкинуть их и их родителей из T
вернуть построенное решение
```

Алгоритм

Функция MAXINDEPENDENTSET(T)

```
пока T не пусто:
  взять в решение все листья
  выкинуть их и их родителей из T
вернуть построенное решение
```

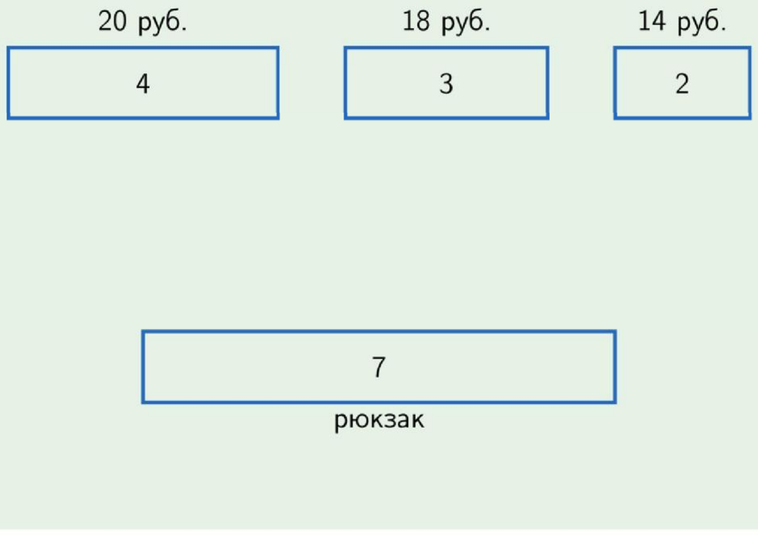
Время работы: $O(|T|)$.

Непрерывный рюкзак

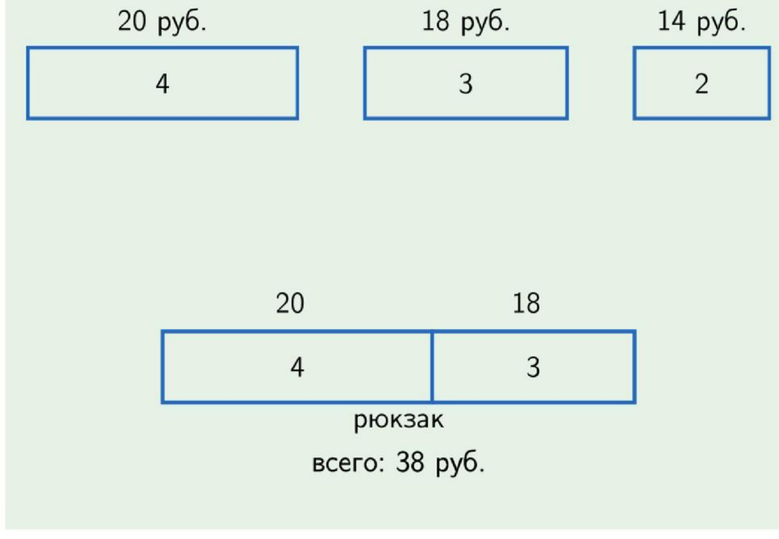
Вход: веса $w_1, \ldots, w_n$ и стоимости $c_1, \ldots, c_n$ данных n предметов; вместимость рюкзака W.

Выход: максимальная стоимость частей предметов суммарного веса не более W.

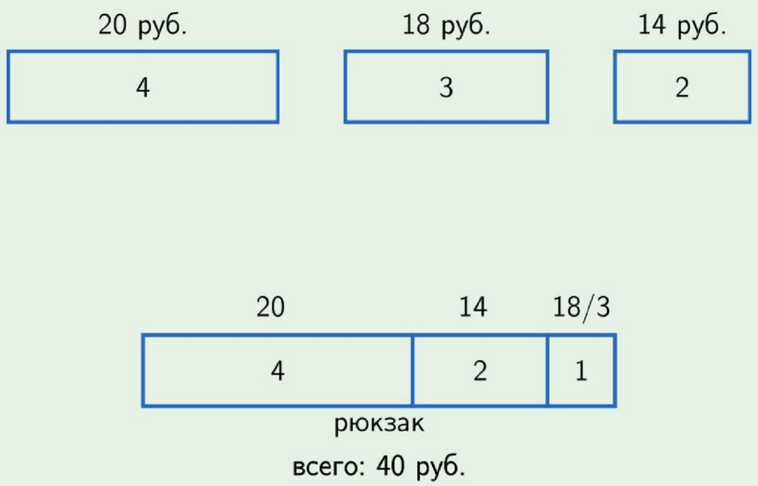
Пример



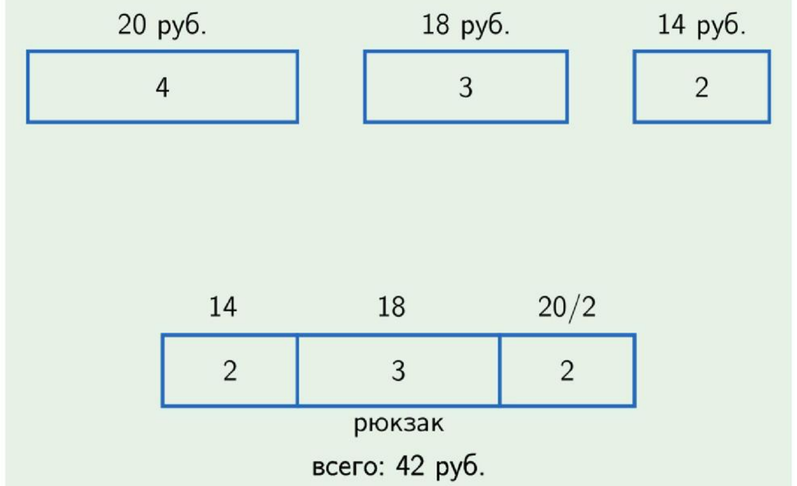
Пример



Пример

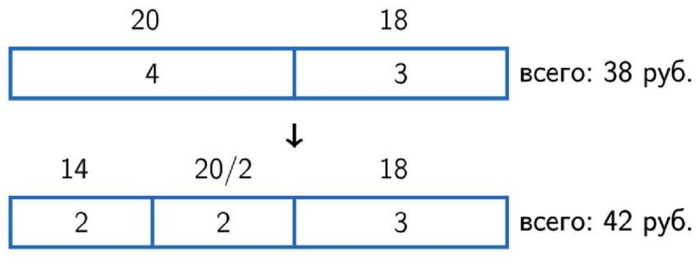


Пример



Надёжный шаг

Существует оптимальное решение, содержащее максимально возможную часть предмета, стоимость которого за килограмм максимальна.



Алгоритм

Функция KNAPSACK$(w_1, c_1, \ldots, w_n, c_n)$

```
отсортировать предметы по убыванию c/w
для всех предметов в полученном порядке:
  взять по максимуму текущего предмета
вернуть построенное решение
```

Алгоритм

Функция KNAPSACK$(w_1, c_1, \ldots, w_n, c_n)$

```
отсортировать предметы по убыванию c/w
для всех предметов в полученном порядке:
  взять по максимуму текущего предмета
вернуть построенное решение
```

Время работы: $T(\text{SORT}) + O(n) = O(n \log n)$.

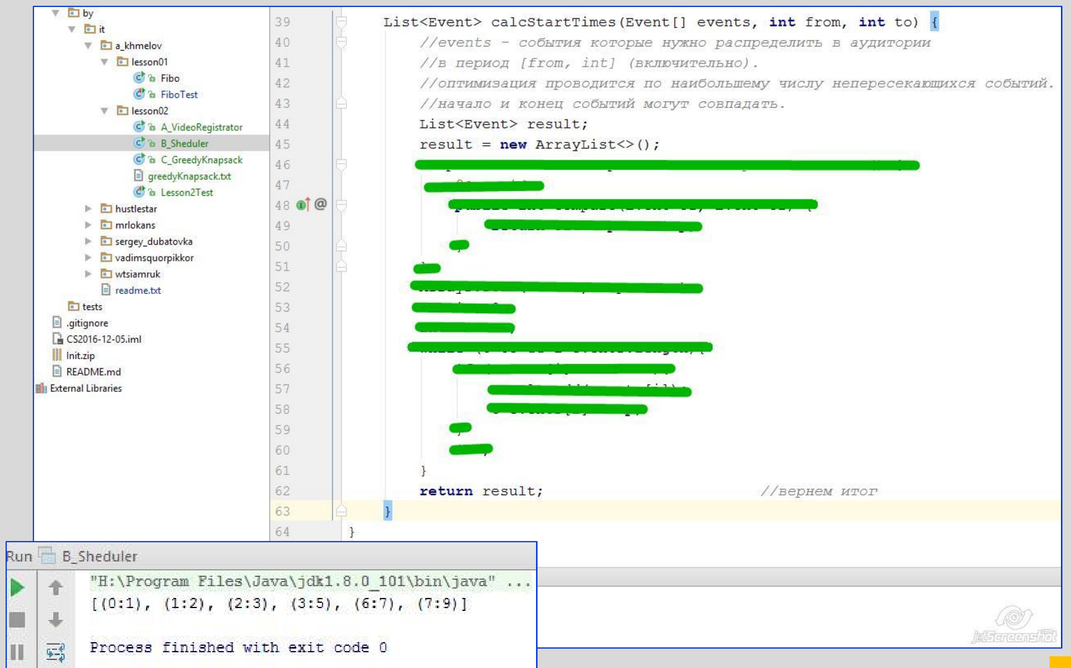
Основные идеи

Надёжный шаг. Существует оптимальное решение, согласованное с локальным жадным шагом.

Оптимальность подзадач. Задача, остающаяся после жадного шага, имеет тот же тип.


```java
public class VideoRegistrator {

    public static void main(String[] args) {
        VideoRegistrator instance=new VideoRegistrator();
        double[] events=new double[]{1, 1.1, 1.6, 2.2, 2.4, 2.7, 3.9, 8.1, 9.1, 5.5, 3.7};
        List<Double> starts=instance.calcStartTimes(events,1);
        System.out.println(starts);
    }

    private List<Double> calcStartTimes(double[] events, double workDuration){
        //events - события которые нужно зарегистрировать
        //timeWorkDuration время работы видеокамеры после старта
        List<Double> result;
        result = new ArrayList<>();
        int i=0;                                    //i - это индекс события events[i]
                                                    //подготовка к жадному поглощению массива событий
                                                    //hint: сортировка Arrays.sort обеспечит скорость алгоритма
                                                    //C*(n log n) + C1*n = O(n log n)


                                                    //пока есть незарегистрированные события
                                                      //получим одно событие по левому краю
                                                      //и запомним время старта видеокамеры
                                                      //вычислим момент окончания работы видеокамеры
                                                      //и теперь пропустим все покрываемые события
                                                      //за время до конца работы, увеличивая индекс
        return result;
    }
}
```

```
Run  VideoRegistrator
"H:\Program Files\Java\jdk1.8.0_101\bin\java" ...
[1.0, 2.2, 3.7, 5.5, 8.1]

Process finished with exit code 0
```

Задание C. Непрерывный рюкзак