## Модуль 2 Жадные алгоритмы
## Лекция 2.2 Задача о расписании

Эта лекция посвящена известной задаче, которая решается жадным алгоритмом, — задаче о расписании.

**Задача о расписании**
У программиста есть $n$ заказов. Для каждого заказа известен дедлайн $d_i$ — срок, до которого его надо выполнить, и его стоимость $c_i$. На выполнение каждого заказа требуется один день. Программист начинает работать с 1-го дня, и если у заказа дедлайн $d_i$, то этот заказ нужно выполнить до конца $d_i$-го дня. Например, если дедлайн заказа равен 1, то программист успеет его выполнить в первый день. Нужно составить оптимальное расписание: определить, какие заказы программист может выполнять и в каком порядке, чтобы максимизировать их суммарную стоимость.

В этой задаче важны ограничения. При совсем маленьких ограничениях её можно решить перебором. У нас будут следующие ограничения:

$$1 \leqslant n \leqslant 5000, \quad 1 \leqslant d_i \leqslant 5000, \quad 1 \leqslant c_i \leqslant 10^9.$$

Как будет действовать программист, если он жадный? Ясно, что он будет сначала выполнять заказы наибольшей стоимости. Но эта стратегия не работает уже на простейшем примере с двумя заказами. Пусть у первого заказа дедлайн 1, у второго — 2, при этом стоимость второго заказа больше, чем первого. Тогда жадный программист в первый день выполнит второй заказ, а первый заказ вообще не успеет выполнить, потому что у него прошёл дедлайн. Если же выполнять сначала первый заказ, потом второй, то будут выполнены они оба.

**Упражнение 2.2.1**
У программиста пять заказов с дедлайнами $d = [1, 2, 2, 3, 5]$ и стоимостями $c = [2, 5, 4, 1, 3]$. Какую максимальную сумму сможет заработать программист?

Правильным решением задачи будет другой жадный алгоритм. Будем стараться выполнять заказы наибольшей стоимости, но ставить их на как можно более поздний день, который пока ещё не занят другими заказами.

Разберём код программы, которая это делает (см. рис. 1).

Пусть дедлайны и стоимости заказов содержатся в векторах $d$ и $c$.

```
vector<int> d;
```

```
vector<int> d;  // дедлайны
vector<int> c;  // стоимости
vector<bool> used(TMAX, false);

long long sum = 0;
for (int i = 0; i < n; i++)
{
    int k = d[i];
    while (k >= 1 && used[k])
        k--;
    if (k == 0)
        continue;
    used[k] = true;
    sum += c[i];
}
```

Рис. 1. Решение задачи о расписании

vector<**int**> c;

В векторе used мы будем отмечать, какие дни уже заняты, то есть used[i] = true, если в $i$-й день уже выполняется какой-то заказ, и used[i] = false, если $i$-й день свободен.

vector<**bool**> used(TMAX, **false**);

Вектор used содержит число элементов, равное числу дней TMAX. Константу TMAX нужно будет предварительно объявить. Вначале вектор used заполнен ложными значениями.

Объявим переменную sum, в которой будет ответ — наибольшая сумма стоимостей выполненных заказов.

**long long** sum = 0;

Эта переменная типа long long, потому что у нас стоимости до $10^9$ и их количество — до $5000$. Сумма может не поместиться в тип int.

Предположим, что заказы заданы в порядке убывания стоимости. Будем идти по заказам в цикле.

**for** (**int** i = 0; i < n; i++)

Рассмотрим $i$-й заказ. Сначала это будет заказ наибольшей стоимости. Нам нужно выбрать для него наибольший свободный день $k$. Мы начинаем со дня дедлайна d[i].

**int** k = d[i];

Уменьшаем $k$ до тех пор, пока день использован (отмечен в массиве used).

**while** (k >= 1 && used[k])

k−−;

Если мы прошли все дни до первого, и они все заняты, мы не можем выполнить $i$-й заказ и переходим к следующему заказу.

**if** (k == 0)

**continue**;

Иначе мы назначаем $i$-й заказ на $k$-й день, отмечаем это в массиве used:

used[k] = **true**;

И прибавляем к сумме стоимость заказа.

sum += c[i];

В результате в переменной sum получится заработанная сумма. При желании можно получить и само расписание. Для этого можно сделать массив used типа int и отмечать в нём не просто занят день или нет, а номер заказа, который мы на этот день поставили.

Приведённый алгоритм жадный. На каждом шаге он пытается добиться выполнения очередного заказа наибольшей стоимости, то есть оптимума для каждого заказа. На следующей лекции мы оценим время работы этого алгоритма и выясним, подойдёт ли он под ограничения задачи. Здесь же мы докажем корректность его работы.

Будем доказывать от противного. Предположим, что жадное решение на всегда дает наибольшую сумму, и есть «плохие» примеры, на которых это не так. Выберем из них пример с наименьшим количеством заказов. Мы покажем, что из этого «плохого» примера можно сделать другой «плохой» пример меньшего размера, на котором жадное решение тоже не оптимальное.

Разберём, как перейти от большего примера к меньшему на конкретных числах. Пусть для какого-то плохого примера имеется два расписания. Первое составлено жадным алгоритмом, второе — оптимальное решение, причём заработанная сумма для этих двух решений не совпадает (см. рис. 2). На рисунке клетки — это дни, цифры в них — стоимости заказов, выполняемых в эти дни. Для наглядности все стоимости заказов разные. В клетке нет цифры, если в этот день не выполняется заказов. Дедлайны здесь не показаны, но мы предполагаем, что в обоих расписаниях заказы стоит не позже своих дедлайнов.

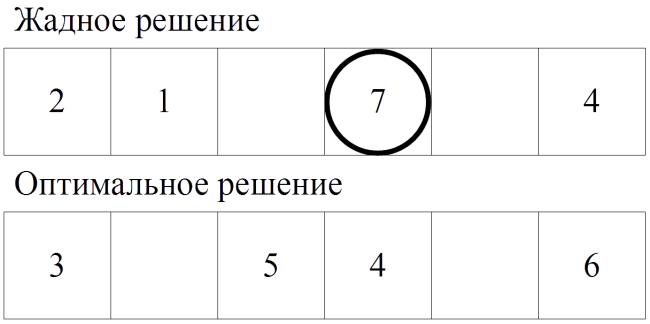


Рис. 2

Рассмотрим самый большой по стоимости заказ в жадном решении. В данном случае это 7. Ясно, что в оптимальном решении не можем быть заказов стоимости больше 7, потому что, если жадное решение не смогло им выбрать дни, это вообще невозможно сделать. Если в оптимальном решении нет заказа стоимости 7, то поставим её в то же место, как и в жадном решении (см. рис. 3). Это можно сделать, не нарушив дедлайны. Тогда суммарная стоимость заказов в оптимальном решении увеличится, а значит, оно было не оптимальным. Мы пришли к противоречию. Значит, в оптимальном решении тоже есть заказ стоимости 7.


Рис. 3

Пусть заказ стоимости 7 есть, то в другой позиции. Эта позиция не может быть больше, чем в жадном решении, потому что жадное решение пытается поставить заказ наибольшей стоимости в наиболее поздний день. Значит, заказ стоимости 7 в оптимальном решении стоит раньше (см. рис. 4). Переставим его на тот же день, что и в жадном решении. Поменяем его с другим заказом, стоящим в этот день (см. рис. 5). Этот другой заказ попадет на более ранний день, что не испортит расписание. Мы преобразовали оптимальное решение так, чтобы наибольший заказ в нем стоял на том же месте, что и в жадном.


Рис. 4

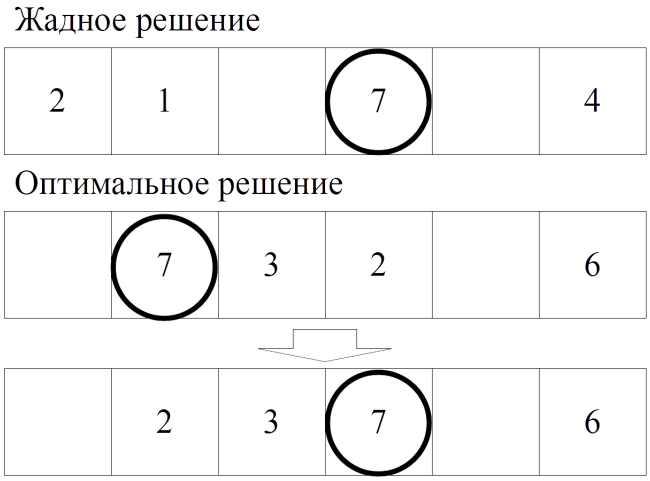


Рис. 5

Теперь можно исключить день с наибольшим заказом из обоих расписаний (см. рис. 6). Получится новый пример, на котором жадное и оптимальное решения дают разные ответы, и этот пример меньше предыдущего. Мы разобрали алгоритм перехода к меньшему примеру на конкретных числах, но ясно, что его можно применить к любому «плохому» примеру. Если мы изначально выберем «плохой» пример с наименьшим количеством заказов, то придем к противоречию. Значит, жадное решение всегда дает оптимальный ответ.


Рис. 6

Этот метод доказательства называется *методом минимального контрпримера*. Мы выбрали минимальный контрпример и показали, что он не минимальный, что от него можно перейти к меньшему примеру. Можно использовать для доказательства другие методы. Например, метод математической индукции. Метод минимального контрпримера удобен тем, что не нужно анализировать все шаги жадного решения. Достаточно рассмотреть только первый шаг, который работает с заказом наибольшей стоимости.

Подведём итоги. Мы разобрали жадное решение задачи о расписании и доказали корректность его работы. На следующей лекции мы оценим время его работы при ограничениях задачи.