{ ЛЕКЦИЯ 1-2 }

{ Элементы теории алгоритмов }

{ Алгебра логики }

Логика (др.-греч. λογική — «наука о правильном мышлении», «способность к рассуждению» от др.-греч. λόγος — «логос», «рассуждение», «мысль», «разум», «смысл») - наука о формах, методах и законах познавательной деятельности, формализуемых на *логическом* языке.

Логика является подразделом не только философии, но и математики, а *булева алгебра* — одной из основ информатики.

В реальной жизни мы соглашаемся с каким-либо утверждением или отрицаем его, например:

- 2 + 6 > 7 – истинное (правдивое) высказывание;
- 1 + 2 > 7 – ложное (ошибочное, неверное) высказывание.

Принять решение о правдивости *логического высказывания* можно только после того, как мы понимаем, что речь идёт о сложении чисел записанных арабскими цифрами.

Подобные фразы предполагают только два возможных ответа – либо "*да*", когда выражение оценивается как *правда*, *истина*, либо "*нет*", когда утверждение оценивается как *ошибочное*, *ложное*.

В программировании и математике если результатом вычисления выражения может быть лишь *истина* или *ложь*, то такое выражение называется *логическим*.

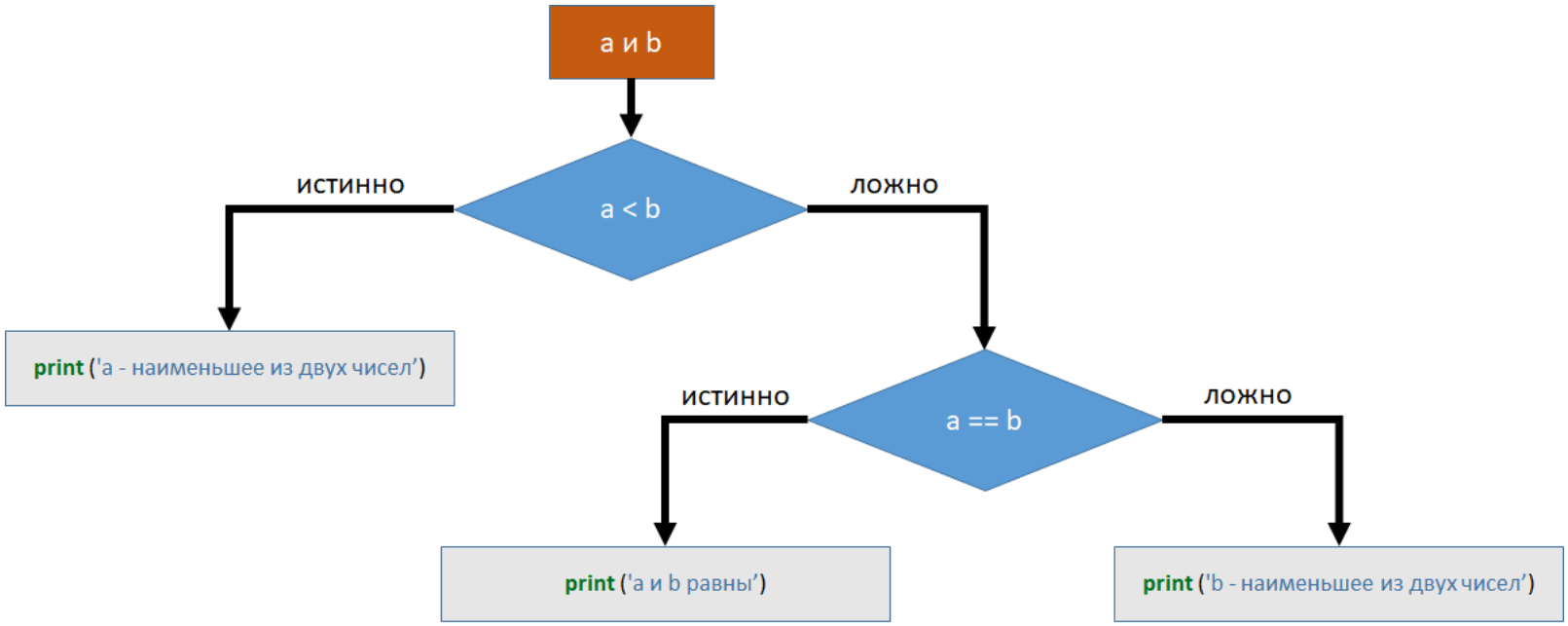
Логические высказывания используются в программировании в условных конструкциях (**if** – **elif** – **else**) для **принятия решения о дальнейшем ходе программы**.

«Носителем» принятого на основе логического высказывания решения является логическая переменная (*bool*).

(Обратите внимание, что логическая переменная не может быть введена с клавиатуры.)

{ Сравнение двух чисел }

Блок-схема алгоритма сравнение двух чисел



Программа на языке Python

```
a = input ('Введите целое число a: ')
b = input ('Введите целое число b: ')

if a < b:
    print ('a - наименьшее из двух чисел')
elif a == b:
    print ('a и b равны')
else:
    print ('b - наименьшее из двух чисел')
```

```
Введите целое число a: 1
Введите целое число b: 2
a - наименьшее из двух чисел
```

{ Сравнение двух чисел }

Изменим программу, явно обозначив логические переменные

```
a = input ('Введите целое число a: ')
b = input ('Введите целое число b: ')

aB = a < b     # эта переменная хранит информацию о результате логического выражения
ab = a == b   # эта переменная хранит информацию о результате логического выражения

print ('относительно утверждения, что  a < b  было принято решение, что оно %s' %(aB))
print ('относительно утверждения, что a == b было принято решение, что оно %s' %(ab))

if aB:
    print ('и значит: a - наименьшее из двух чисел')
elif ab:
    print ('и значит: a и b равны')
else:
    print ('и значит: b - наименьшее из двух чисел')
```

```
Введите целое число a: 5
Введите целое число b: 4
относительно утверждения, что  a < b  было принято решение, что оно False
относительно утверждения, что a == b было принято решение, что оно False
и значит: b - наименьшее из двух чисел
```

{ Сравнение двух чисел }

Отметим, что выражение вида

```
a = input ('Введите целое число a: ')
b = input ('Введите целое число b: ')

aB = a < b     # эта переменная хранит информацию о результате логического выражения
ab = a == b   # эта переменная хранит информацию о результате логического выражения

print ('относительно утверждения, что  a < b  было принято решение, что оно %s' %(aB))
print ('относительно утверждения, что a == b было принято решение, что оно %s' %(ab))

if aB == True:
    print ('и значит: a - наименьшее из двух чисел')
elif ab == True:
    print ('и значит: a и b равны')
else:
    print ('и значит: b - наименьшее из двух чисел')
```

являются корректными, но лишены практического смысла, т.к.

```
aB = True
print (aB)
print (aB == True)
```

```
True
True
```

{ Алгебра логики }

Алгебра логики (*алгебра высказываний*) — раздел математической логики, изучающий логические операции над высказываниями (атомарными и составными).

Основные логические операции над логическими высказываниями **A** и **B**:

- **не A** – инверсия (отрицание), на *Python* **!A**;
- **A и B** – конъюнкция (логическое A * B), на *Python* **A and B**;
- **A или B** – дизъюнкция (логическое A + B), на *Python* **A or B**;
- **A → B** импликация (следствие);
- **A ↔ B** эквивалентность, на *Python* **A == B**;
- **A !↔ B** антиэквивалентность (XOR, исключающее или, ⊕);

x y	x * y	x + y	x ⊕ y	x → y	x == y	x != y
0 0	0	0	0	1	1	0
0 1	0	1	1	1	0	1
1 0	0	1	1	0	0	1
1 1	1	1	0	1	1	0

Любую логическую функцию F(x, y, z) можно представить в виде **дизъюнктивной**: F = F1 + F2 + F3, или **конъюнктивной**: F = F1 * F2 * F3 – нормальной формы.

{ Алгебра логики }

Законы алгебры логики.

Взаимодействие с константами:

$$0 + A = A, \quad 1 + A = 1, \quad 0 \cdot A = 0, \quad 1 \cdot A = A;$$

Закон повторения:

$$A \cdot A = A, \quad A + A = A;$$

«Третий лишний»:

$$A + \overline{A} = 1, \quad A \cdot \overline{A} = 0;$$

Закон поглощения:

$$A \cdot (A + B) = A, \quad A + A \cdot B = A;$$

Свойства **И** и **ИЛИ**:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C) = A \cdot B \cdot C, \quad (A + B) + C = A + (B + C) = A + B + C$$

$$A \cdot B = B \cdot A, \quad A + B = B + A, \quad A \cdot (B + C) = (A \cdot B) + (A \cdot C), \quad A + (B \cdot C) = (A + B) \cdot (A + C);$$

Законы Де Моргана (отрицания):

$$\overline{A + B} = \overline{A} \cdot \overline{B}, \quad \overline{A \cdot B} = \overline{A} + \overline{B}, \quad \overline{\overline{A}} = A.$$

{ Алгебра логики. Пример. }

Программа для проверки того, что хотя бы одно число из N введённых пользователем чётное:

```
N = 3
flag = False
for i in range (N):
    a = int( input ('Введите целое число: ') )
    flag = flag or (a%2 == 0)
print ('%sвсе введённые числа %sявляются чётными' %('не '*int(flag),'не '*(1-int(flag))))
```

```
Введите целое число: 3
Введите целое число: 4
Введите целое число: 5
не все введённые числа являются чётными
```

```
Введите целое число: 3
Введите целое число: 5
Введите целое число: 7
все введённые числа не являются чётными
```

Программа для проверки того, что все N чисел введённых пользователем чётные:

…?

{ Алгоритм }

Слово ”**алгоритм**” возникло довольно поздно, оно образовано от имени арабского математика *аль-Хорезми* (9 век н.э.), автора известного арабского учебника по математике (от его имени произошли также слова «алгебра» и «логарифм»).

Хотя, само понятие алгоритма является наиболее древним из базовых понятий математики. Вавилоняне, шумеры и египтяне знали, как вычислять различные геометрические характеристики объектов и решать простейшие уравнения. Эти рецепты (**алгоритмы**) измерений и вычислений не имели обоснования в современном смысле этого слова, доказательства (а вместе с ними и теоремы) появились только в древней Греции.

Одни из наиболее древних **алгоритмов**:

- Алгоритм Евклида (Euclidean algorithm) — 500-300 гг. д.н.э.
- Решето Эратосфена (Sieve of Eratosthenes) — 300-200 гг. д.н.э.

Алгоритм – это всякая система (последовательность) вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Способы записи **алгоритма**:

- Словесное описание или математическая формула;
- Блок-схема;
- Псевдокод или программа на одном из языков программирования.

{ Алгоритм }

Свойства **АЛГОРИТМОВ**.

Дискретность. На каждом шаге выполнения алгоритма происходит обновление (дополнение) текущих данных.

Детерминированность. Каждый шаг и переход от шага к шагу должны быть точно определены.

Элементарность. Выполняемые операции должны быть допустимы для **исполнителя**.

Массовость. Исходные данные допустимые для алгоритма должны составлять некоторое достаточно большое множество.

Направленность. Если способ обновления текущих данных не дает результата, то должно быть указано, что надо считать результатом алгоритма.

Конечность. Результат работы алгоритма должен быть получен за конечное врем.

Исполнитель - это человек, компьютер, автоматическое устройство и т.п. Исполнитель должен уметь выполнять все команды, составляющие алгоритм. Множество возможных команд конечно и изначально строго задано. Действия, которые выполняет исполнитель по этим командам называются элементарными.

Наиболее распространённые модели абстрактных исполнителей:

- **машины Тьюринга и Поста;**
- **конечные автоматы;**
- **регистровые машины.**

{ Машина Тьюринга }

Машина Тьюринга (МТ) состоит из двух частей – **ленты** и **автомата**.

Лента используется для хранения информации, она бесконечна в обе стороны и разбита на клетки, которые никак не нумеруются и не именуются.

Автомат – это активная часть **МТ**. В каждый момент он размещается под одной из клеток ленты и видит её содержимое (это видимая клетка, а символ в ней – видимый символ) содержимое других клеток автомат не видит. Кроме того, в каждый момент **автомат** находится в одном из **состояний**, которые будем обозначать буквой q.

Автомат может выполнять три элементарных **действия**: **1) записывать** в видимую клетку символ; 2) **сдвигаться** на одну клетку влево или вправо; 3) **переходить в новое состояние**.

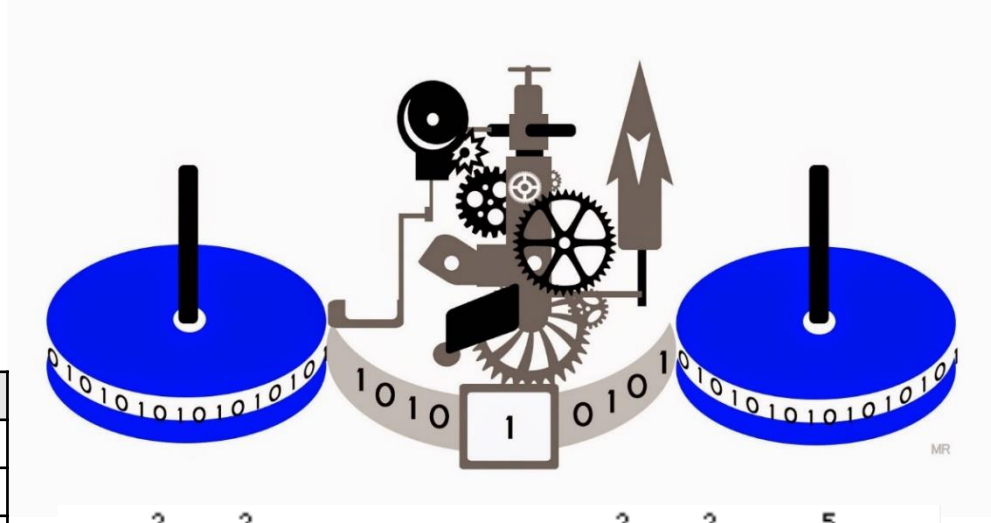
Пару из видимого **символа** (S) и текущего **состояния автомата** (q) будем называть **конфигурацией** и обозначать (S, q).

Такт работы машины Тьюринга: (S', [L,R,N], q')

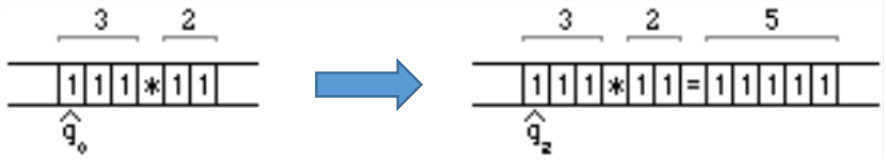
S' – символ для записи;

[L,R,N] – сдвиг автомата (влево, вправо, нет);

q' – новое состояние.



	q_0	q_1	q_2	q_3	q_4	$q_5 = q_z$
1	0 R q_1	1 R q_1	1 R q_2	1 L q_3		1 S q_z
0				0 R q_0	1 L q_4	
*	* R q_0	* R q_1		* L q_3	* L q_4	
=	= L q_4	= R q_2		= L q_3		
λ		= R q_2	1 L q_3		λ R q_5	



{ Вычислимость }

Вычислимые функции — это множество функций вида

$$f : N \to N$$

которые могут быть реализованы с помощью определённого исполнителя (например, машины Тьюринга).

Задачу вычисления такой функции называют алгоритмически разрешимой, если возможно написать алгоритм, вычисляющий эту функцию, иначе алгоритмически неразрешимой.

(т.к. машина Тьюринга это абстрактная модель ЭВМ, любая функция, которая может быть вычислена физическим устройством, может быть вычислена машиной Тьюринга. И все выводы касающиеся абстрактной машины Тьюринга, полностью применимы ЭВМ любой архитектуры.)

Проблема остановки (Тьюринг, 1936)

Проблему остановки можно сформулировать как: не существует общего алгоритма, который бы мог определить, остановится ли программа, по ее описанию и входным данным. Т.е. функция определения остановки **невычислима**.

На практике это значит, что сам компьютер никогда не сможет определить точно, зависнет ли его очередная программа в бесконечном потоке инструкций или нет.

Проблема остановки занимает центральное место в теории вычислимости, поскольку представляет собой первый пример задачи, которую невозможно решить алгоритмическим путём.

Подождите...

{ Теория сложности }

Ресурсные ограничения: время работы, память, объем коммуникаций.

Теория сложности для каждой задачи ищет верхние оценки затрат (эффективные алгоритмы), нижние оценки затрат (доказательства сложности).

Какая бывает сложность:

- **комбинационная сложность** – минимальное число элементов для реализации алгоритма в виде вычислительного устройства;
- **описательная сложность** – длина описания алгоритма на формальном языке;
- **вычислительная сложность** – количество элементарных операций, исполняемых алгоритмом для допустимых входных данных.

При отсутствии в алгоритме циклов описательная и вычислительная сложность пропорциональны.

Асимптотические оценки.

Часто задачи и алгоритмы их решения могут иметь однотипные исходные данные n – штук.

Если затраты (время) на выполнение алгоритма $f(n)$ допускает оценки с помощью некоторой функции $g(n)$ при n > n', то говорят:

- $f(n)$ ограничена сверху – растёт как $O(g(n))$

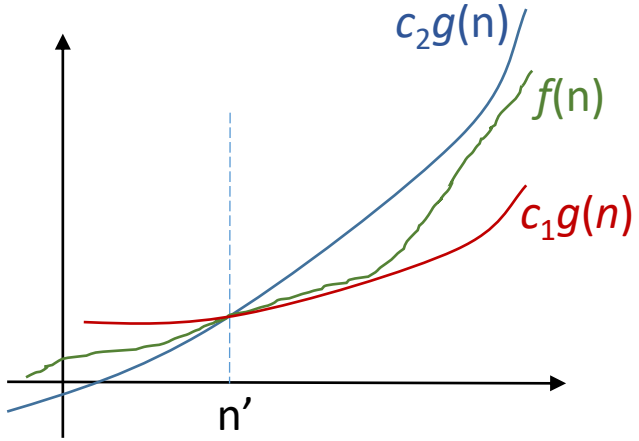
$$f(n) \le cg(n);$$

- $f(n)$ ограничена снизу – растёт как $\Omega(g(n))$

$$f(n) \ge cg(n);$$

- $f(n)$ ограничена – растёт как $\Theta(g(n))$

$$c_1 g(n) \le f(n) \le c_2 g(n).$$



Алгоритм называется **полиномиальным**, если время его работы ограничено сверху полиномом:

$$g(n) = P_k(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0.$$

Алгоритм называется **экспоненциальным**, если время его работы ограничено снизу показательной функцией:

$$g(n) = a^n.$$

n – например, размер массива, количество символов в строке …

{ Пример. Последовательный поиск. }

Постановка задачи. В списке *P* имеется *n* различных элементов. Список содержит некоторое значение *a*. Необходимо найти индекс (номер) элемента содержащий это значение *P*[*i*] = *a*.

Будем решать эту задачу методом перебора. Предполагается, что каждый элемент списка с равной вероятностью может быть искомым значением.

Сколько операций *K* потребуется для решения этой задачи?

$$K_{\min} = 1,$$
$$K_{\max} = n,$$
$$K_{avg} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n(n-1)}{2n} = \frac{n-1}{2}.$$

Код на языке *Python*

```
for i in range (n):
        if P[i] == a:
                print (i)
                break
```

{ Пример. Последовательный поиск. }

Рассмотрим алгоритм поиска с помощью последовательного сравнения.

```
def simple_search (a, key):
    n = len(a)
    for i in range(n):
        if a[i] == key:
            return i
    return n
```

```
ary = [7,8,1,2,3,4,13,5,1,2,44,5,1]
print(simple_search (ary, 13))
```

```
6
```

```
def clever_search (a, key):
    n = len(a)
    i=0
    while a[i]!=key:
        i=i+1
    return i
```

```
ary2 = [7,8,1,2,3,4,13,5,1,2,44,5,1, 13]
print(clever_search (ary2, 13))
```

```
6
```

```
int simple_search(int a[], int N, int key) {

    for (int i = 0; i < N; i++) {
        if (a[i] == key) {
            return i;
        }
    }
    return N;
}
```

```
int clever_search(int a[], int N, int key) {
    a[n] = key;
    int i;
    for (i = 0; a[i] != key; i++) {
        ;
    }
    return i;
}
```

{ Пример. Возведение в степень. }

Постановка задачи. Вычислить x^n.

Тривиальный алгоритм. (Код на языке *Python)*

```
print (x, n)
y = 1
for i in range (n):
        y = y * x
        print (y)
```

```
7 9
7
49
343
2401
16807
117649
823543
5764801
40353607
```

Количество умножений — *n.* Количество операций (сложность) — *O(n).*

{ Пример. Возведение в степень. }

Постановка задачи. Вычислить x^n.

Разумное вычисление. (Код на языке *Python)*

$$x^n = x^{\sum_{i=0}^{k} a_i 2^i} = \prod_{\substack{i=0 \\ a_i \in \{0,1\}}}^{k} x^{a_i 2^i} = \prod_{\substack{i=0 \\ a_i = 1}}^{k} x^{2^i}$$

```
print (x, n)
y = 1
X = x
N = n
while N > 0:
        if N % 2 == 1:
                y = y * X
        X = X * X
        N = N // 2
        print (y, N)
```

```
7 9
7 4
7 2
7 1
40353607 0
```

Количество арифметических операций — O(log(n)).

{ Задача о рюкзаке }

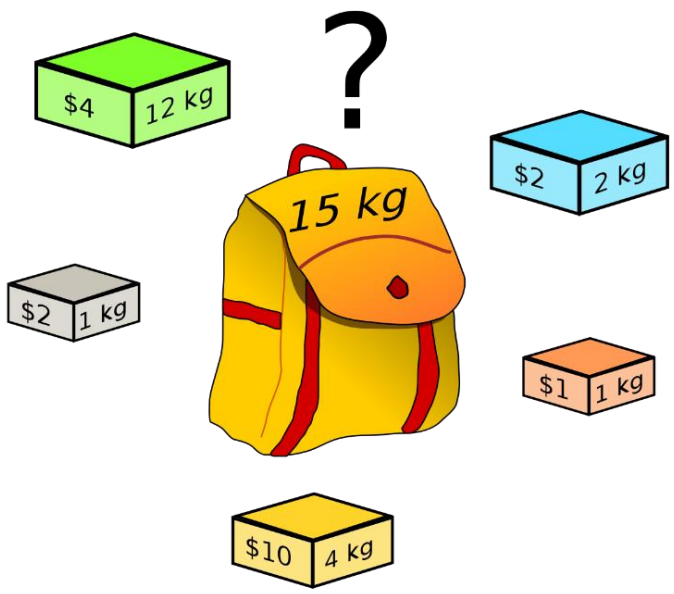
Постановка задачи. Есть некоторое количество вещей разной «ценности» c_i и разного веса (или размера) a_i. Вам необходимо собрать в рюкзак всё наиболее ценное, учитывая размер рюкзака B. Делить вещи нельзя. Т.е. нужно максимизировать функцию

$$\sum_{i=1}^{n} c_i x_i \to \max$$

с учётом ограничения

$$\sum_{i=1}^{n} a_i x_i \le B, \quad x_i \in \{0,1\}.$$

$$c = [10, 9, 5, 4, 3, 3, 1], \quad a = [5, 3, 6, 12, 2, 4, 1], \quad B = 15.$$



{ Задача о рюкзаке }

Постановка задачи. Есть некоторое количество вещей разной «ценности» c_i и разного веса (или размера) a_i. Вам необходимо собрать в рюкзак всё наиболее ценное, учитывая размер рюкзака B. Делить вещи нельзя. Т.е. нужно максимизировать функцию

$$\sum_{i=1}^{n} c_i x_i \to \max$$

с учётом ограничения

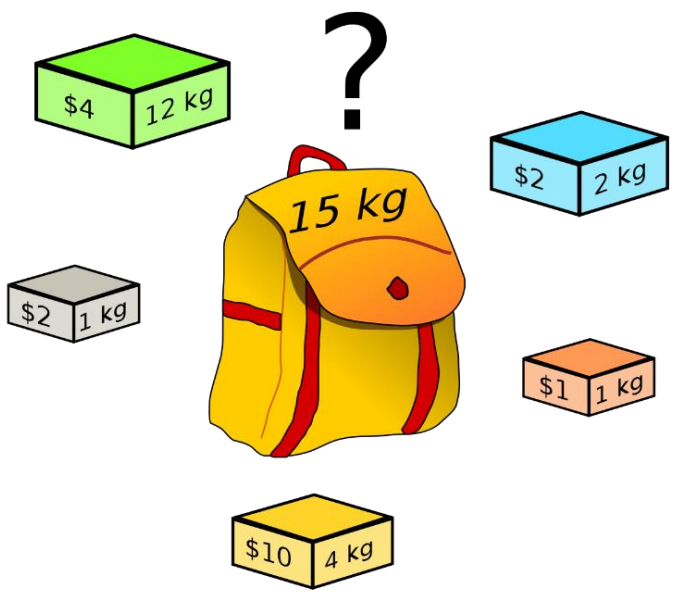
$$\sum_{i=1}^{n} a_i x_i \le B, \quad x_i \in \{0,1\}.$$

$$c = [10, 9, 5, 4, 3, 3, 1], \quad a = [5, 3, 6, 12, 2, 4, 1], \quad B = 15.$$

Алгоритм «жадина 1»:

$$i \in \hat{I}, \quad i : \max(c_j) = c_i, \quad j \in I \setminus \hat{I}$$

$$\hat{I} = [0,1,2,6], \quad \sum_{i \in \hat{I}} c_i = 10+9+5+1 = 25, \quad \sum_{i \in \hat{I}} a_i = 5+3+6+1 = 15$$



{ Задача о рюкзаке }

Постановка задачи. Есть некоторое количество вещей разной «ценности» c_i и разного веса (или размера) a_i. Вам необходимо собрать в рюкзак всё наиболее ценное, учитывая размер рюкзака B. Делить вещи нельзя. Т.е. нужно максимизировать функцию

$$\sum_{i=1}^{n} c_i x_i \to \max$$

с учётом ограничения

$$\sum_{i=1}^{n} a_i x_i \le B, \quad x_i \in \{0,1\}.$$

$$c = [10, 9, 5, 4, 3, 3, 1], \quad a = [5, 3, 6, 12, 2, 4, 1], \quad B = 15.$$

Алгоритм «жадина 1»:

$$i \in \hat{I}, \quad i : \max(c_j) = c_i, \quad j \in I \setminus \hat{I}$$

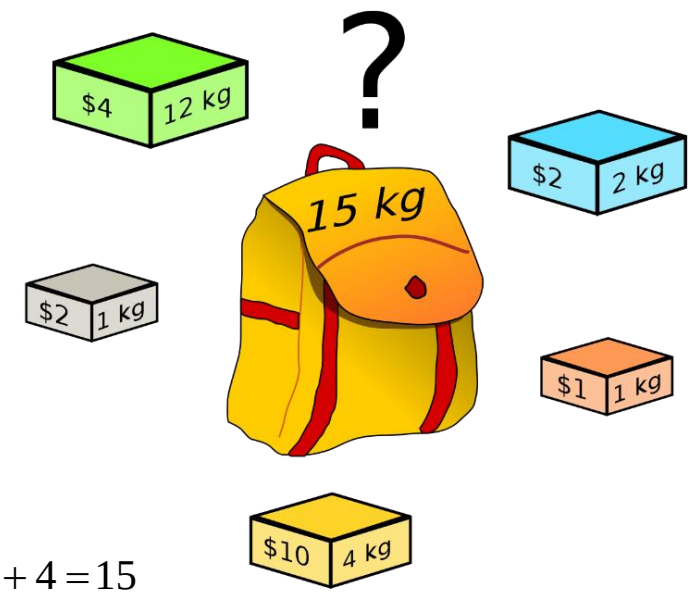
$$\hat{I} = [0,1,2,6], \quad \sum_{i \in \hat{I}} c_i = 10+9+5+1 = 25, \quad \sum_{i \in \hat{I}} a_i = 5+3+6+1 = 15$$

Алгоритм «жадина 2»:

$$i \in \hat{I}, \quad i : \max\left(\frac{c_j}{a_j}\right) = \frac{c_i}{a_i}, \quad j \in I \setminus \hat{I}$$

$$\frac{c_i}{a_i} = \left[2, 3, \frac{5}{6}, \frac{1}{3}, \frac{3}{2}, \frac{3}{4}, 1\right]$$

$$\hat{I} = [1,0,4,6,5], \quad \sum_{i \in \hat{I}} c_i = 9+10+3+1+3 = 26, \quad \sum_{i \in \hat{I}} a_i = 3+5+2+1+4 = 15$$



{Задача о рюкзаке. }

Единственный надёжный метод решения – **полный перебор.**

(Однако, если бы можно было резать предметы на куски задача имела бы более простое решение.)

Метод перебора это алгоритм, удовлетворяющий всем требованиям.

Определим его сложность как комбинацию *k* предметов из *n* возможных:

$$f(n) = 2^n.$$

Алгоритм полного перебора:

1. Перенумеруем все предметы.
2. Положим максимум стоимости равным 0.
3. Возьмём двоичное число с *n* разрядами для записи комбинации укладки вещей.
4. Рассмотрим все возможные варианты от 000…000 до 111…111.
5. Если объем не превосходит допустимый, вычисляем ценность комбинации и сравниваем с текущим максимумом (обновляем его значение, если оно меньше).

Оценим временные затраты на выполнение алгоритма в случае **n = 128.**
Исполнитель: пусть **1 процессор** вычисляет **1 комбинацию** за **1 нс (10^{-9} с),** пусть есть **10^{12} процессоров**, тогда:

?

{Задача о рюкзаке. }

Единственный надёжный метод решения – **полный перебор.** ***NP*** **– полная задача.**

(Однако, если бы можно было резать предметы на куски задача имела бы более простое решение.)

Метод перебора это алгоритм, удовлетворяющий всем требованиям.

Определим его сложность как комбинацию *k* предметов из *n* возможных:

$$f(n) = 2^n.$$

Алгоритм полного перебора:

1. Перенумеруем все предметы.
2. Положим максимум стоимости равным 0.
3. Возьмём двоичное число с *n* разрядами для записи комбинации укладки вещей.
4. Рассмотрим все возможные варианты от 000…000 до 111…111.
5. Если объем не превосходит допустимый, вычисляем ценность комбинации и сравниваем с текущим максимумом (обновляем его значение, если оно меньше).

Оценим временные затраты на выполнение алгоритма в случае **n = 128.**
Исполнитель: пусть **1 процессор** вычисляет **1 комбинацию** за **1 нс (10^{-9} с),** пусть есть **10^{12} процессоров**, тогда:

$$\frac{2^{128} \cdot 10^{-9}}{10^{12}} \approx 10.8 \cdot 10^{9} \left(лет \right).$$

{ Жадный алгоритм. }

Экстремальные задачи – задачи на нахождение оптимальных (минимальных или максимальных) значений.

Жадные алгоритмы имеют суть минимизации (максимизации) некоторого критерия качества исходя только из локальных свойств, и имеют итерационный характер.

Пример типичного жадного алгоритма:

Имеется непрерывная функция n переменных $f(x)$. Она задаёт поверхность в n – мерном пространстве.

Алгоритм минимизации:

- Выбирается начальная точка x_0
- Исследуется малая окрестность текущей точки
- если удаётся определить точку x_{i+1} в которой функция принимает меньшее значение, то делаем её текущей и переходим к предыдущему шагу
- конец

{ Жадный алгоритм. }

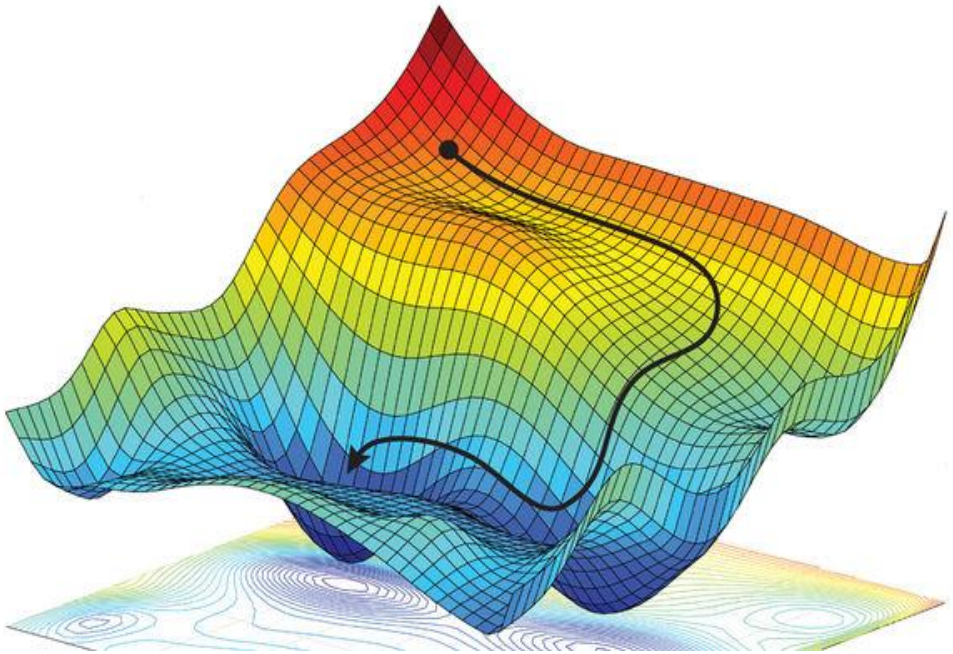
Прямой поиск минимума функции. Градиентный спуск.

$$f\left(x_1, x_2\right) \to \min.$$

Градиентом называется направление наибольшего возрастания функции. Введём обозначение градиента

$$\text{grad}f = \nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \right).$$

Идея: для уменьшения значения функции надо двигаться в направлении обратном градиенту.



То есть итерации имеют вид:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \lambda \nabla f\left(\mathbf{x}_i\right), \quad \lambda > 0.$$

И продолжаются пока

$$\left|\mathbf{x}_{i+1} - \mathbf{x}_i\right| < \varepsilon.$$

{ Пример. Градиентный спуск. }

Рассмотрим минимизацию функции: $f(x,y)=(x-3)^2+(y+2)^2$.

Будем искать её минимум **дискретным покоординатным спуском:** (Код на языке *Python)*

```
def fun (x,y)
    return (x-3)*(x-3) + (y+2)*(y+2)
```

```
x0=0; y0=0; d=1.; dx = [d,-d,0,0]; dy = [0,0,d,-d];
bestfound = True
minF = fun (x0,y0)
while bestfound:
    bestfound = False
    for i in [0,1,2,3]:
        newF = fun (x0+dx[i], y0+dy[i])
        if newF<minF:
            minF=newF; newx = x0+dx[i]; newy = y0+dy[i]
            bestfound = True
    if bestfound:
        x0 = newx; y0 = newy
        print ('min(f(%13.6e, %13.6e)) = %13.6e' % (x0,y0,minF))
```

```
min(f( 0.000000e+00,  0.000000e+00)) =  1.300000e+01
min(f( 1.000000e+00,  0.000000e+00)) =  8.000000e+00
min(f( 2.000000e+00,  0.000000e+00)) =  5.000000e+00
min(f( 2.000000e+00, -1.000000e+00)) =  2.000000e+00
min(f( 3.000000e+00, -1.000000e+00)) =  1.000000e+00
min(f( 3.000000e+00, -2.000000e+00)) =  0.000000e+00
```

{ Разделяй и властвуй. }

Стратегия "*Разделяй и властвуй*":

Разделение задачи на несколько подзадач, которые представляют собой меньшие экземпляры той же задачи.

Властвование над подзадачами путем их рекурсивного решения. Если размеры

подзадач достаточно малы, такие подзадачи могут решаться непосредственно.

Комбинирование решений подзадач в решение исходной задачи.

Часто решение является *рекурсивным*, прерывающимся базовым случаем, который может быть решен тривиально.

Рекурсия — вызов функции (процедуры) из неё же самой, непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия). Количество вложенных вызовов функции или процедуры называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

{ Числа Фибоначчи. }

Последовательность чисел Фибоначчи задаётся рекуррентным выражением:

$$F_n = \begin{cases} 0, & если \quad n = 0, \\ 1, & если \quad n = 1, \\ F_{n-1} + F_{n-2}, & если \quad n > 1. \end{cases}$$

Что даёт следующую последовательность {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …}.

Рекуррентная формула → рекурсивный алгоритм:

```
def Fibonacci (n)
    if n == 0:
        return 0
    if n == 1:
        return 1
    return Fibonacci (n-1) + Fibonacci (n-2)
```

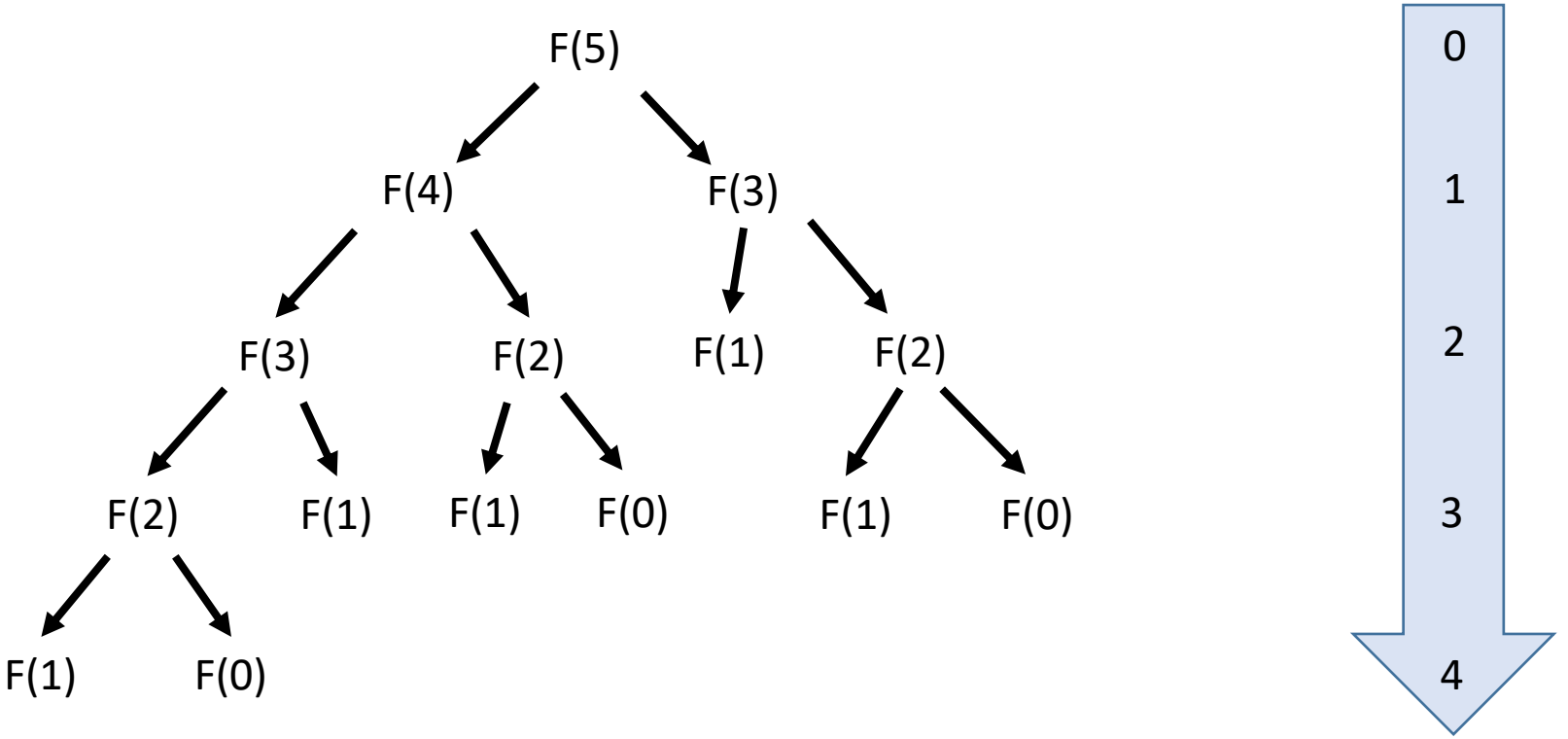
Fibonacci (*0*)

0

Fibonacci (*9*)

34

{ Числа Фибоначчи. }



F(0) – 3 раза, F(1) – 5 раз, F(2) – 3 раза, F(3) – 2 раза.

{ Числа Фибоначчи. }

Ускорим вычисления сохраняя все вычисляемые числа Фибоначи:

```
c = [0 for i in range (100)]

def Fibonacci (n)
        if n == 0:
                return 0
        if n == 1:
                return 1
        if c[n]  > 0:
                return c[n]
        c[n] = Fibonacci (n-1) + Fibonacci (n-2)
        return c[n]
```

```
Fibonacci (0)
```

```
0
```

```
Fibonacci (9)
```

```
34
```

{ Числа Фибоначчи. }

Введём вектор из двух последовательных (например, первых) чисел Фибоначчи:

$$\begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

И умножим его на матрицу

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}.$$

Повторяя эту процедуру *n* раз имеем:

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}.$$

{ НОД. }

Постановка задачи. Для заданных натуральных *a* и *b* найти наибольший общий делитель *q.*

Идея. $$пусть \quad a = bq + r \quad \Rightarrow \quad НОД(a,b) = НОД(b,r).$$

Т.к. если *k* — любой общий делитель чисел *a* и *b*, не обязательно наибольший, тогда $a = t_1 \cdot k$ и $b = t_2 \cdot k$, где t_1 и t_2 — целые числа из определения. Тогда *k* является также общим делителем чисел *b* и *r*, так как *b* делится на *k* по определению, а

$$r = a - bq = (t_1 - t_1 q)k$$

(выражение в скобках есть целое число, следовательно, k делит r без остатка).

Алгоритм. Рассмотрим рекуррентные соотношения

$$a = bq_0 + r_1; \quad b = r_1 q_1 + r_2; \quad r_1 = r_2 q_2 + r_3 \quad \ldots \quad r_{n-1} = r_n q_n.$$

Тогда НОД(*a,b*) равен r_n, последнему ненулевому члену этой последовательности.

Сложность вычисления НОД алгоритмом Евклида можно оценить как **O(log(a)+log(b))** арифметических операций над натуральными числами.

Доказательство.

$$a > b \quad \Rightarrow \quad a \geq b + (a \bmod b) \geq 2(a \bmod b) = 2r$$

$$a \geq 2r \quad \Rightarrow \quad r \leq \frac{a}{2} \quad \Rightarrow \quad br \leq \frac{ab}{2}$$

т. е. произведение *ab* уменьшается на каждой итерации вдвое, и после log(*ab*) итераций станет меньше 1, т. е. равно нулю. А это означает, что *b* = 0 (т. е. НОД уже найден).

{ НОД. }

(Код на языке Python)

```
def gcd (a, b)
        print (a, b)
        if a == 0:
                return b
        return gcd (b % a, b)
```

```
gcd (125, 35)
```

```
125 35
35 125
20 35
15 20
5 15
0 5
```

{ Проблема работы с данными. }

Затраты на элементарные операции:

X86, X64 → double 64 бита.

X86: сложение 32-битных ~ 1 такт, сложение 64-битных ~ 3 такта.

X64: сложение 32-битных ~ 1 такт, сложение 64-битных ~ 1 такта.

X86: умножение 32-битных ~ 3-4 такта, умножение 64-битных ~ 15-50 такта.

X64: умножение 32-битных ~ 3-4 такта, умножение 64-битных ~ 4-5 такта.

X86, X64: деление ~ 12-44 тактов.

{ ВСЁ }