

## Тема 8. Вычислимые функции и методы разработки алгоритмов

### План лекции

1. Понятие вычислимой функции.
2. Суперпозиция, примитивная рекурсия, минимизация. Примеры.
3. Связь с методами разработки алгоритмов.
4. Понятие об алгоритмической неразрешимости. Доказательство существования алгоритмически неразрешимых задач. Примеры.
5. Развитие понятия алгоритма: параллельное программирование и распределённые алгоритмы, объектно-ориентированный подход к разработке программ, методы искусственного интеллекта.
6. Понятие вычислительной сложности (по времени и памяти) алгоритма и его применение для анализа алгоритмов.
7. Основные методы и приёмы анализа сложности.
8. Сложность алгоритмов с ветвлениями, циклами.
9. Сложность рекурсивных алгоритмов.

### 8.1 Понятие вычислимой функции

Функция называется вычислимой, **если она вычисляется некоторым алгоритмом.**

Функция  $f$  называется вычислимой, если существует **алгоритм**, перерабатывающий всякий объект  $x$ , для которого определена функция  $f$ , в объект  $f(x)$  и не применимый ни к какому  $x$ , для которого  $f$  не определена.

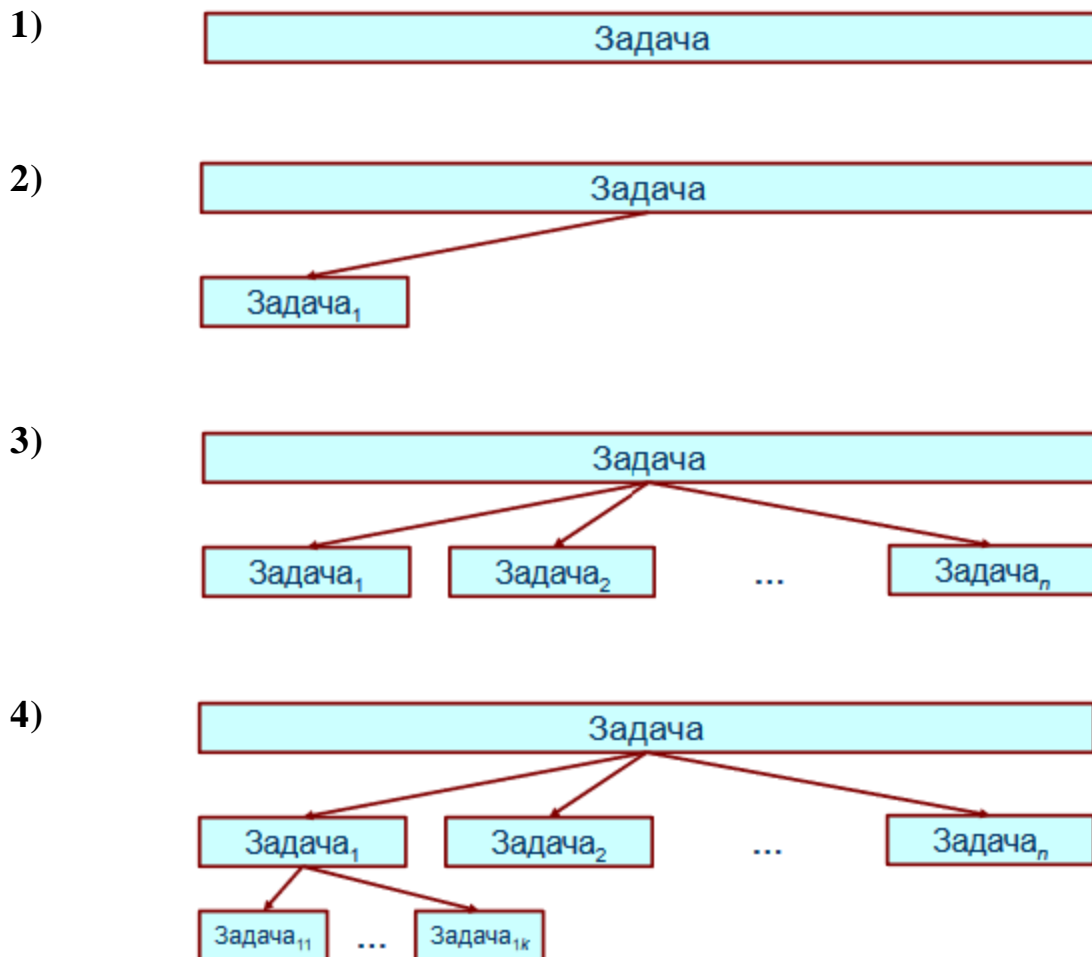
**Пример.**  $x$  — натуральное число,  $f(x) = x^2$

*Алгоритм задаёт процедуру отыскания значений вычислимой функции.*

**Можно ли определить общий вид вычислимых функций?**

**Вывести канонический метод написания алгоритмов, соответствующий построению вычислимых функций?**

Общий подход к разработке алгоритмов решения сложных задач – «разделяй и властвуй», т.е. метод пошаговой детализации, когда решение сложной задачи разбивается на решение более простых задач, декомпозиция выполняется до тех пор, пока решения не будут описаны в терминах операций исполнителя – элементарных операций



Рассмотрим схему:

1. возьмем набор простых базовых функций  $P$ , которые могут быть вычислены (вычислимость доказана разработкой алгоритма, например, программы машины Тьюринга);

2. введем операции над функциями  $O$ , где операнды – функции, и результат – **функции (а не значения функций)**;

3. выполним все возможные операции из (2) над базовыми функциями из (1), по-разному комбинируя их в качестве аргументов, и расширим набор функций  $P$  путем включения туда вновь полученных функций;

4. над новым набором  $P$  вновь произведем все возможные операции из набора  $O$ , получим новые функции, которые вновь включим в множество  $P$  и так далее.

Операции над функциями должны быть очевидны с математической точки зрения, т.е. не должно возникать сомнений в том, что из вычислимых функций с помощью введенных операций получаются вновь вычислимые функции.

Так как количество исполнений (3) не ограничено сверху, то таким образом будем получать все новые и новые функции.

Таким образом, шаги 1-3 описывают построение некоторого бесконечного множества функций, но каждая конкретная функция  $f$  из этого множества является результатом выполнения конечного числа операций, взятых из набора  $O$  над базовыми функциями.

Получили процесс построения функции  $f$ , который:

1. начинается с исходных данных – функций, выбираемых из базового набора  $P$ ;
2. выполняется пошагово (в дискретном времени);
3. на каждом шаге выполняется одна из элементарных операций (из набора  $O$ );
4. результат каждого шага строго определен;
5. процесс заканчивается через конечное число шагов.

Этот перечень позволяет говорить об алгоритме  $\alpha_f$  построения функции  $f$ .

**Все ли свойства алгоритма выполняются?**

*Свойство массовости алгоритмов* не выполняется: исходные данные всегда одни и те же – базовый набор функций.

Это позволяет говорить о том, что текст алгоритма (последовательность применения операций из  $O$  и места подстановок в эти операции операндов из базового набора) задают *единственную функцию, а не множество функций*.

Для различных функций  $f_1$  и  $f_2$  алгоритмы будут различны.

**Как добиться нужного результата?**

**Как построить алгоритм для «вычисления» именно той функции, которая даёт решение поставленной задачи?**

Следующие вопросы:

**1. Какие операции над функциями используются при построении алгоритма?**

**2. Как вычисляются значения построенных функций?**

Начнем с ответа на вопрос «**2. Как вычисляются значения построенных функций?**».

Алгоритм  $\alpha_f$  построения функции  $f$  легко преобразовать в алгоритм вычисления значений этой функции

$$f(x_1, x_2, \dots, x_n)$$

введением и использованием в алгоритме исходных данных – набора

$$x_1, x_2, \dots, x_n$$

Введем конкретные базовый набор функций  $P$  и систему операций  $O$  для формирования множества функций. В качестве области определения функций возьмем  $n$ -кратное декартово произведение множества неотрицательных целых чисел. Сами функции, как и их аргументы, также принимают значения из множества неотрицательных целых чисел.

Рассмотрим набор функций

$$P = \{Z(x), S(x), I_{m,n}(x_1, x_2, \dots, x_m, \dots, x_n)\}.$$

Эти функции вычислимы – для  $Z(x)$ ,  $S(x)$  в теме 6 были построены машины Тьюринга, вычисляющие их значения по значениям аргументов. Таким образом, этот набор удовлетворяет выдвинутому выше требованию.

## 8.2 Суперпозиция, примитивная рекурсия, минимизация. Примеры

Здесь мы дадим ответ вопрос «**1. Какие операции над функциями используются при построении алгоритма?**» в следующем разделе.

В систему операций  $O$  входят три операции:

1. суперпозиции  $\sigma$ ;
2. примитивной рекурсии  $\rho$ ;
3. минимизации  $\mu$ .

### 8.2.1 Суперпозиция

Операция **суперпозиции**  $\sigma$  получает набор из  $n+1$  операнда – функций  $f_0, f_1, \dots, f_n$  и производит результат – функцию

$$f = \sigma(f_0, f_1, \dots, f_n).$$

Каким образом вычисляется значение этой функции?

Будем считать, что  $f_1, \dots, f_n$  зависят от одного и того же набора аргументов  $x_1, x_2, \dots, x_k$ .

Формулу для вычисления значений вновь образованной функции  $f$  можно задать следующим образом:

$$f(x_1, x_2, \dots, x_k) = f_0(f_1(x_1, \dots, x_k), \dots, f_n(x_1, \dots, x_k)).$$

Здесь  $k$  – количество переменных в объединенном наборе переменных функций с индексами от 1 до  $n$ . Аргументы функции  $f_0$  не входят в перечень аргументов результата операции суперпозиции, но количество их должно быть равно  $n$ .

При вычислении функции

$$f(x_1, x_2, \dots, x_k) = f_0(f_1(x_1, \dots, x_k), \dots, f_n(x_1, \dots, x_k))$$

операции выполняются строго последовательно:

$$\begin{aligned} r_1 &= f_1(x_1, \dots, x_k), \\ &\dots, \\ r_n &= f_n(x_1, \dots, x_k), \\ f(x_1, x_2, \dots, x_k) &= f_0(r_1, \dots, r_n). \end{aligned}$$

Все рассматриваемые функции являются *частичными* (не всюду определенными), т.е. могут существовать такие комбинации значений аргументов, для которых значение функции не существует:

- 1)  $f(a_1, \dots, a_k)$  не существует, если не существует хотя бы одно из значений  $f_1(a_1, \dots, a_k), \dots, f_n(a_1, \dots, a_k)$  или,
- 2) если эти значения существуют и равны  $b_1, \dots, b_n$ , то не существует значение  $f_0(b_1, \dots, b_n)$ .

Таким образом, функция  $f$  также является *частичной*.

### 8.2.2 Операция примитивной рекурсии

Операция *примитивной рекурсии*  $\rho$  имеет два операнда:

$$f = \rho(g, h).$$

Первый операнд,  $g$ , зависит от  $n$  аргументов,  $n \geq 0$ , а второй операнд,  $h$ , имеет в общем случае два дополнительных аргумента, хотя в том и другом случае некоторые аргументы могут быть несущественными.

Функция-результат  $f$  определяется следующими уравнениями примитивной рекурсии:

$$\begin{cases} f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n), \\ f(x_1, \dots, x_n, y + 1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)). \end{cases}$$

**Пример 1.** Сложение неотрицательных целых чисел,  $Add(x, y) = x + y$  задается следующими уравнениями:

$$\begin{cases} Add(x, 0) = I_1^1(x), \\ Add(x, y + 1) = S(Add(x, y)). \end{cases}$$

#### Как получены эти формулы?

Можно ли операцию сложения выразить через функции базового набора?

Рассмотрим функции базового набора: в нем есть функции

1.  $S(x) = x + 1$  увеличения аргумента на 1;
2. выбора  $I_{1,1}(x)$

При  $y = 0$  получим:

$$Add(x, 0) = x = I_{1,1}(x)$$

Если взять аргумент больше нуля ( $y + 1 > 0$  для неотрицательных значений  $y$ ), то получим:

$$\begin{aligned} Add(x, y + 1) &= x + 1 + 1 + \dots + 1 + 1 = \\ &= (x + 1 + 1 + \dots + 1) + 1 = Add(x, y) + 1 = S(Add(x, y)) \end{aligned}$$

**Пример 2.** Умножение  $Mult(x, y) = x \times y$  задается уравнениями

$$\begin{cases} Mult(x, 0) = 0, \\ Mult(x, y + 1) = Add(Mult(x, y), x) \end{cases}$$

которые используют уже построенную функцию сложения чисел и известные соотношения  $x \times 0 = 0$ :

$$Mult(x, 0) = 0$$

и  $x \times (y + 1) = x \times y + x$ :

$$\begin{aligned} Mult(x, y + 1) &= x + x + \dots + x + x = (x + x + \dots + x) + x = \\ &= Mult(x, y) + x = Add(Mult(x, y), x) \end{aligned}$$

**Пример 3.** Возведение в степень  $Power(x, y) = x^y$  задается следующими уравнениями:

$$\begin{cases} Power(x, 0) = 1, \\ Power(x, y + 1) = Mult(x, Power(x, y)). \end{cases}$$

При  $y = 0$  получим:

$$Power(x, 0) = x^0 = 1$$

Если взять аргумент больше нуля ( $y + 1 > 0$  для неотрицательных значений  $y$ ), то получим:

$$\begin{aligned} Power(x, y + 1) &= x \times x \times \dots \times x \times x = \\ &= (x \times x \times \dots \times x) \times x = Power(x, y) \times x = Mult(Power(x, y), x) \end{aligned}$$

**Пример 4.** Вычисление функции факториала задается следующими уравнениями:

$$Fact(x) = \begin{cases} 1, & x = 0 \\ 1 \times 2 \times \dots \times x, & x > 0 \end{cases}$$

## Выведите формулу с использованием операций суперпозиции и примитивной рекурсии

### 8.2.3 Операция минимизации

Операция *минимизации*  $\mu$  имеет один операнд:

$$f = \mu(g).$$

Значения функции  $f$  на заданном наборе аргументов  $x_1, x_2, \dots, x_n$  получаются следующим образом:

1. Сначала с помощью функции  $g$  формируется уравнение

$$g(x_1, \dots, x_{n-1}, y) = x_n$$

2. Затем отыскивается его решение относительно переменной  $y$ . Если таких решений несколько, то берется минимальное из них (отсюда название операции); оно считается значением функции  $f$  на заданном наборе аргументов  $f(x_1, x_2, \dots, x_n)$ .

Операцию минимизации обычно записывают с помощью оператора минимизации  $\mu_y$  в виде

$$f(x_1, x_2, \dots, x_{n-1}, x_n) = \mu_y(g(x_1, \dots, x_{n-1}, y) = x_n)$$

Операция минимизации для функций одной переменной является средством отыскания *обратной функции*.

Примеры обратных функций:

- $Sub(x, y)$  – функция, обратная  $Add(x, y)$ ;
- $Div(x, y)$  – функция, обратная  $Mult(x, y)$ .

Вычисление обратной функции используется для решения многих задач

**Пример.** При разработке генераторов случайных чисел с заданной функцией распределения.



## 8.3 Связь с методами разработки алгоритмов

### 8.3.1 Методы разработки алгоритмов: постановка задачи

Заказчики (или «конечные пользователи», «end-users») обычно ставят перед программистом задачу следующим образом:

«Имеются исходные данные  $X$ . Компьютер должен выдать результаты  $Y$ , которые находятся в такой-то зависимости от исходных данных».

В лучшем случае заказчики дают описание существующей зависимости как некоторой функции или системы функций, возможно, с дополнительными условиями, налагаемыми на вид результата

**Примеры** постановки задачи:

1. Даны матрица и вектор. Найти вектор, равный произведению матрицы на исходный вектор.
2. Дана матрица. Определить, существует ли обратная матрица.
3. Дана строка символов. Исключить из нее все символы, не являющиеся буквами или цифрами.
4. Дана последовательность чисел. Найти минимальное из чисел.
5. Дана последовательность чисел. Переписать ее (найти новую последовательность) так, чтобы числа располагались по возрастанию.

Дают ли такие постановки задач алгоритм решения её решения?

Эти постановки задач достаточно просты, но их нельзя назвать описаниями алгоритмов: в этих формулировках *отсутствуют такие свойства алгоритмов*, как

- элементарность шагов;
- дискретность.

Эти постановки задач ориентированы на произвольные исходные данные, взятые из некоторых классов, т.е. свойство **массовости** выполняется. **Детерминированность** также имеет место. Относительно **конечности** есть некоторая неопределенность: даже единственный шаг может не завершиться.

Перед программистом (разработчиком) стоит задача – *превратить описание задачи пользователя в описание алгоритма, который будет реализован программой.*

Разработчик должен один неопределенный шаг превратить в последовательность элементарных шагов, выполнение которых компьютером, для которого разрабатывается программа, приводит к требуемому результату.

Первая задача программиста – сделать переход от «**что надо сделать**» к тому «**как это сделать**»

### Как это сделать?

#### 8.3.2 Методы разработки алгоритмов: сведение задачи к подзадачам

Ключевым подходом в алгоритмизации является **сведение задачи к подзадачам** – таким образом один шаг (описание задачи) превращается в *последовательность элементарных шагов*, где *каждый шаг – решение выделенной подзадачи.*

Последовательно выделяемые более **простые шаги** соответствуют **частным задачам (подзадачам)**, совокупное решение которых приводит к решению исходной задачи.

Иерархию задач в общем случае можно изобразить в виде **дерева**:



Различные методы разработки алгоритмов отличаются тем, на какие подзадачи производится разбиение и как эти подзадачи соотносятся между собой.

Общепринятой классификации методов разработки алгоритмов нет, но можно выделить *общие принципы*.

Любая задача может быть сформулирована как функция преобразования исходных данных в выходные данные,  $f(X) = Y$ . Как исходные данные, так и функция могут быть достаточно сложными.

Разбиение задачи на подзадачи может быть связано с попыткой «упрощения» **самой функции** (её сведением к более простым) или «упрощения» **данных** (сокращения их объема, уменьшения значений, упрощения структуры...).

### 8.3.3 Методы разработки алгоритмов: подходы к выделению подзадач

Разбивая задачу на подзадачи, можно:

1. *разбивать исходные и выходные данные на части или упрощать:*

– под **разбиением** понимается разделение структуры данных на части, **например**, разделение вектора из 10 компонентов на несколько векторов с меньшим числом компонентов; или разделение текста на предложения;

– под **упрощением** понимаются такие ситуации, когда, данные нельзя «разбить»: **например**,  $X$  – число и его нельзя разделить на части, которые будут обрабатываться отдельно но его можно разложить, скажем, в сумму  $X = X_1 + X_2$ , так, что результаты  $f(X_1)$  и  $f(X_2)$  отыскиваются проще, чем  $f(X)$ ;

2. *производить декомпозицию функции* – превращать ее в суперпозицию более простых:

$$f(X) = g(h(s(X))) \text{ или } f(X) = g(X, h(X), s(X)).$$

### 8.3.4 Методы разработки алгоритмов

Выделение подзадач можно выполнить следующим образом:

1. Разложение задачи в *последовательность разнородных подзадач*.
2. Разложение задачи в последовательность *однородных подзадач* (итерация).
3. *Метод последовательных приближений* (частный случай итерации).
4. Сведение задачи к *самой себе* (рекурсия).
5. Решение *обратной задачи*.

### 8.3.5 Методы разработки алгоритмов: разложение задачи в последовательность разнородных подзадач

Этот метод называют иногда методом «**разделяй и властвуй**»: обычно выделяется относительно небольшое число простых, легко решаемых подзадач.

Результат решения первой подзадачи становится исходными данными для второй подзадачи и так далее.

Таким образом, здесь использован второй подход «в чистом виде» — *декомпозиция функции*, задание ее суперпозицией более простых.

Такая суперпозиция может быть реализована последовательным соединением машин Тьюринга

### 8.3.6 Методы разработки алгоритмов: разложение задачи в последовательность однородных подзадач

**Итерация** — это важный частный случай предыдущего метода, где задача  $P$  сводится к  $n$  экземплярам более простой задачи  $R$  и к простой задаче  $Q$ , объединяющей  $n$  полученных решений.

**Пример.** Задача *P* – вычисление скалярного произведения двух векторов *A* и *B*:

```
S := 0; {задача Q - подготовка места для суммирования}  
for i := 1 to n do  
    {Задача R – перемножение элементов и суммирование:}  
    S := S + A[i] * B[i];
```

Этот алгоритм использует *сведение к более простым операциям и разбиение исходных данных на части* – отдельные компоненты векторов. Простые операции над отдельными элементами **повторяются** – выполняются в *цикле*.

### 8.3.7 Методы разработки алгоритмов: метод последовательных приближений

Это частный случай итерации. Задача *P* нахождения решения сводится к многократному решению задачи *R* улучшения решения.

Метод предполагает, что

- каким-то образом *может быть оценено «качество» решения* (обычно это точность);

- *на каждом шаге, в том числе на первом, решение уже существует*, но оно не является абсолютно точным.

Предполагается, что исходные данные **не разбиваются на части, не упрощаются, а используются на каждом шаге итерации**. Цель очередного шага – увеличить точность решения, полученного на предыдущем шаге

Чаше всего *абсолютная точность недостижима*, поэтому процесс потенциально бесконечен. Чтобы обеспечить *финитность алгоритма*, обычно требуют отыскать не точное решение *Y*, а любое решение, отличающееся от *Y* не более, чем на некоторую величину *E*, т.е. ищут «**приближенное**» решение с известной погрешностью.

### Пример:

- отыскание корня уравнения (численное решение);
- вычисления значения функции путём её разложения в ряд:

$$F(x) = 1 + \frac{x^2}{2 \times 3} + \frac{x^4}{2 \times 3 \times 4 \times 5} + \frac{x^6}{2 \times 3 \times 4 \times 5 \times 6 \times 7} + \dots + \frac{x^{2n}}{(2n+1)!} + \dots$$

```
program PR;  
var X,E: real;  
function F(X,E: real): real;  
var X,E,P,S: real; N: integer;  
begin P:=1; S:=1; N:=1;  
while abs(P)>E do  
begin P:=P*(X*X/(2*N*(2*N+1))); S:=S+P; N:=N+1 end;  
F:=S  
end;  
begin readln(X); readln(E); writeln(F(X,E)) end.
```

Каждая итерация – уточнение значения функции путём добавления к сумме очередного слагаемого. Выполнение цикла прекращается, когда очередное слагаемое становится меньше заданного значения E (по абсолютной величине)

1. Как выявить закономерность, позволяющую последовательно уточнять значения, начав с заданного значения?
2. Всегда ли алгоритм завершается с получением верного значения?

Основа решения в данном случае – математический анализ, теория функций (позволяет ответить на вопрос о сходимости функций...)

### 8.3.8 Методы разработки алгоритмов: сведение задачи к самой себе

Это рекурсия, «чистый» вариант 1 – упрощения исходных данных.

При реализации рекурсивных вычислений алгоритм (процедура, функция) либо обращается к себе прямо (*прямая рекурсия*), либо через процедуру, решающую другую задачу (*косвенная рекурсия*).

Рекурсия может быть *кратной* – в алгоритме имеется несколько рекурсивных обращений. Обязательное условие *финитности* – наличие *нерекурсивной ветви в алгоритме*.

**Пример.** Задача отыскания вещественного корня уравнения.

Пусть дано уравнение

$$f(x) = 0,$$

где функция  $f(x)$  определена на отрезке  $[a, b]$  и в единственной точке  $x'$  принимает нулевое значение,  $f(x') = 0$ .

Требуется с точностью до  $\varepsilon$  найти это значение  $x'$ , иначе говоря, найти такой интервал  $I$ , длина которого не превосходит  $\varepsilon$  и  $x' \in I$ .

Решение – рекурсивная функция, реализующая метод деления отрезка пополам (дихотомию) [1].

```
function OneHalf(u, v, eps: real; function f(x: real): real): real;  
  var c: real;  
  begin if (f(u) = 0.0) or ((v-u) <= eps) then OneHalf := u  
    else if f(v) = 0.0 then OneHalf := v  
    else  
      begin c := (u+v)/2;  
        if f(u)*f(c) >= 0.0  
          then OneHalf := OneHalf(c, v, eps, f)  
          else OneHalf := OneHalf(u, c, eps, f)  
        end  
    end;
```

Вычисление значения корня  $XRoot$  уравнения  $F(x) = 0$  с заданной точностью  $Epsilon$  производится обращением к функции:

$XRoot := OneHalf(a, b, Epsilon, F);$

**Пример.** Последовательность натуральных чисел

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, . . . ,

определяемая рекуррентными соотношениями

$$F(n) = F(n-1) + F(n-2), F(1) = 1, F(2) = 1,$$

называется числами Фибоначчи ( $F(0) = 0$ ).

**Задание:** разработать функцию, реализующую алгоритм вычисления заданного числа Фибоначчи (числа с заданным номером) в последовательности, основываясь на рекурсии и итерации.

### 8.3.9 Методы разработки алгоритмов: итоги

Если не существует математических оснований, показывающих, что задача «аддитивна» (ее решение может быть получено «объединением» решений частных задач), то *перечисленные методы неприменимы*.

**Остаётся применять:**

6. Метод полного перебора.

7. Эвристические методы.

### 8.4 Понятие об алгоритмической неразрешимости. Доказательство существования алгоритмически неразрешимых задач. Примеры

Алгоритмическая неразрешимость (АН) в математической логике – это свойство математической задачи, заключающееся в отсутствии алгоритма ее решения [2].

Алгоритмическая неразрешимость — важнейшее свойство некоторых классов корректно поставленных задач, допускающих применение алгоритмов. Оно состоит в том, что задачи каждого из этих классов в принципе не имеют какого-либо общего, универсального алгоритма решения, объединяющего этот класс. Несмотря на полную однотипность условий и требований, здесь, как ни парадоксально, принципиально невозможна однотипность метода решения. А. н. не означает неразрешимости тех или иных единичных проблем данного класса — часть из них может иметь свои решения [3].

Алгоритмически неразрешимыми являются, например, проблема распознавания того, можно ли из имеющихся конечных автоматов собрать



заданный автомат; а также множество других проблем, относящихся к топологии, к теории групп и к другим областям [3].

При доказательстве алгоритмической неразрешимости некоторой задачи принято сводить ее к классической задаче – «задаче останова» машины Тьюринга [4].

**Теорема 1.** Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных, при этом и алгоритм и данные заданы символами на ленте машины Тьюринга, определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Таким образом, фундаментально алгоритмическая неразрешимость связана с бесконечностью выполняемых алгоритмом действий.

Существуют ли какие-нибудь проблемы, для которых невозможно придумать алгоритмы их решения? Утверждение о существовании алгоритмически неразрешимых проблем является весьма сильным – мы констатируем, что мы не только сейчас не знаем соответствующего алгоритма, но мы не сможем никогда принципиально его найти [4].

**Примеры АН задач [4]:**

1. Распределение девяток в записи числа Пифагора.
2. Вычисление совершенных чисел. Совершенные числа – это числа, которые равны сумме своих делителей, за исключением самого числа, например,  $28=1+2+4+7+14$ .
3. Десятая проблема Гильберта. Пусть задан многочлен  $n$ -ой степени с целыми коэффициентами  $P$ . Существует ли алгоритм, который определяет имеет ли уравнение  $P=0$  решение в целых числах? Советский математик Ю.В. Матиясевич показал, что такого алгоритма не существует.
4. Проблема останова машины Тьюринга (**Теорема 1**).
5. Проблема эквивалентности алгоритмов. По двум произвольным заданным алгоритмам, например, по двум машинам Тьюринга, определить,

будут ли они выдавать одинаковые выходные результаты для любых входных данных.

6. Проблема тотальности. По записи произвольного заданного алгоритма определить, будет ли он останавливаться на всех возможных наборах исходных данных. Другая формулировка этой задачи: является ли частичный алгоритм  $A$  всюду определенным алгоритмом?

В теории алгоритмов проблемы, для которых может быть предложен частичный алгоритм их решения, частичный в том смысле, что он возможно, но не обязательно, за конечное количество шагов находит решение проблемы, называются частично разрешимыми проблемами. Например, проблема останова является частично разрешимой проблемой, а проблема эквивалентности и тотальности не являются таковыми.

## **8.5 Развитие понятия алгоритма: параллельное программирование и распределённые алгоритмы, объектно-ориентированный подход к разработке программ, методы искусственного интеллекта**

### **8.5.1 Параллельное программирование**

Желание выполнять всё более сложные расчёты за минимальное время приводит к требованию повышения производительности вычислительных систем. Из методов повышения производительности в настоящее время отдаётся предпочтение параллелизации. Бесхитростному методу повышения производительности путём повышения тактовой частоты препятствует фундаментальный закон Рэлея (1871), постулирующий четвертую степень зависимости тепловыделения от тактовой частоты. При этом возможность отвода тепла затруднительна вследствие малой площади теплового контакта собственно процессора с внешним теплоотводом. При современной кремниевой технологии коэффициент  $k$  в формуле  $W \approx k \times F^4$  равен приблизительно единице Ватт/ГигаГерц<sup>4</sup>, что ограничивает (по возможностям

теплоотвода) тактовую частоту величиной нескольких гигагерц. Достижения технологии в виде реализации многоядерности позволяют заменить четвёртую степень линейной функцией от количества отдельных параллельных вычислителей, однако требуют серьёзных усилий по выявлению и реализации параллелизма [5].

**Поток** = последовательность элементов, команд или данных, обрабатываемая процессором [6].

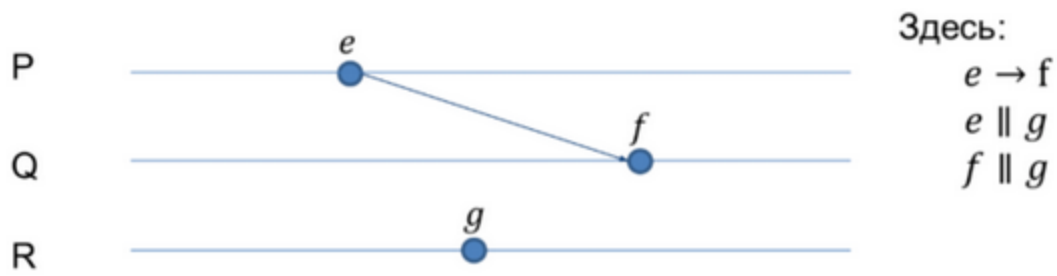
**Категории компьютерных систем.** Компьютерные системы можно разделить на четыре основных категории. Для этого нужно несколько сменить наше представление о том, как работает программа. С точки зрения центрального процессора программа представляет собой **поток инструкций (операций, команд)**, которые следует расшифровать и выполнить. Данные тоже можно считать поступающими в **виде потока**. Четыре категории компьютерных систем, о которых мы говорим, определяются тем, поступают ли программы и данные одним потоком или несколькими [4].

#### Модель параллельного программирования [7].

Рассмотрим:

- множество операций  $e, f, g \dots$  (чтение, запись ячеек памяти и т.п.), произошедших во время исполнения алгоритма
- множество процессов  $P, Q, R$

Операции  $e$  и  $f$  – **параллельные** (обозначается  $e \parallel f$ ):



**Исполнение системы** – это пара  $(H, \rightarrow_H)$ , где:

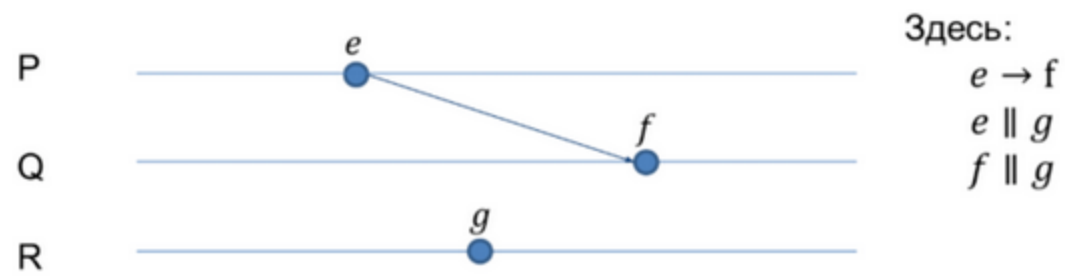
$H$  – множество операций  $e, f, g \dots$  (чтение, запись ячеек памяти и т.п.), произошедших во время исполнения;

$\rightarrow_H$  – это отношение частичного строгого порядка на множестве операций (транзитивное, антирефлексивное, асимметричное);

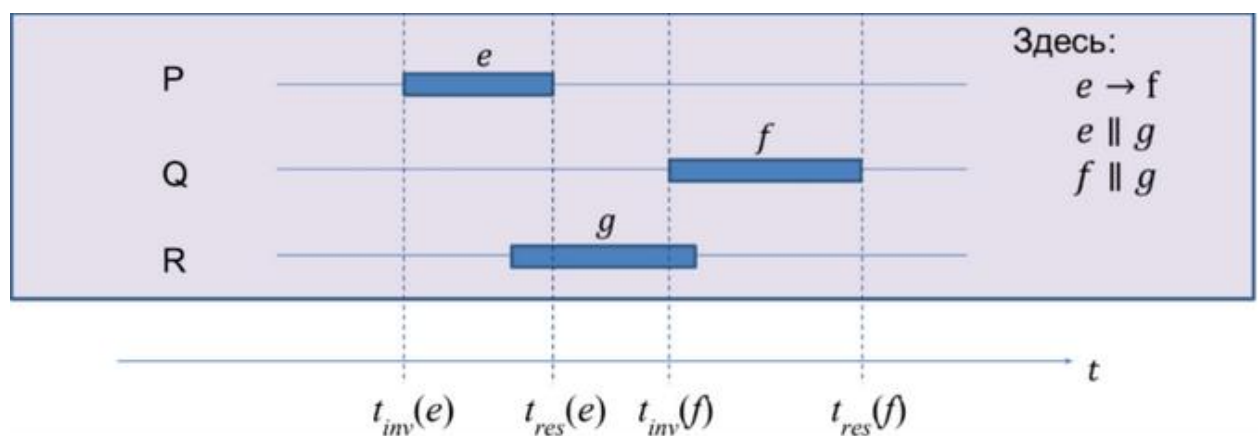
$E \rightarrow_H f$  означает, что операция  $e$  произошла до операции  $f$  в исполнении  $H$ .

Пусть  $e, f \in H$ . Тогда говорят, что  $e$  параллельна  $f$ , если  $e \not\rightarrow f \wedge f \not\rightarrow e$ .

Обозначение:  $e \parallel f$ :



**Модель глобального времени** определим так, что это модель, в которой в качестве **операции** используется временной интервал:  $e = [t_{inv}(e), t_{res}(e)]$ , причём  $t_{inv}(e), t_{res}(e) \in R$ . Зададим в этой модели отношение  $\rightarrow$  следующим образом:  $e \rightarrow f \Leftrightarrow t_{inv}(f) > t_{res}(e)$ .

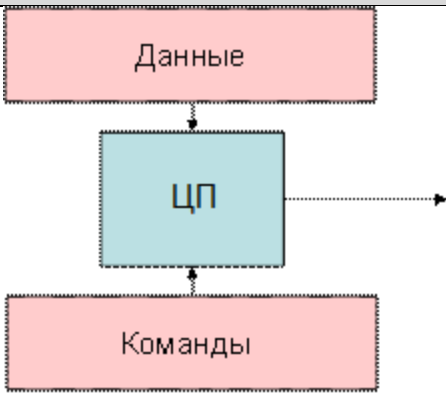


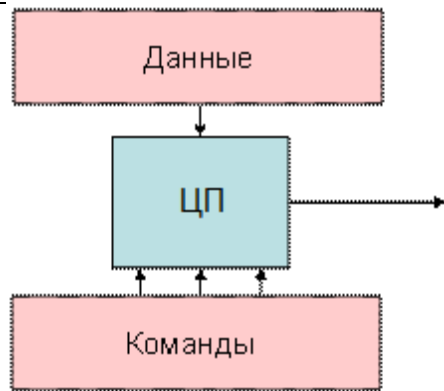
Неформально это означает, что вход в функцию, выполняющую операцию  $f$ , был осуществлён строго позже, чем был получен результат работы функции, выполняющей операцию  $e$ .

*Глобального времени не существует* из-за физических ограничений, поэтому в доказательствах такая модель не используется, но **помогает при визуализации различных исполнений**

В таксономии (классификации) Флинна (1966 г.) выделяется 4 класса архитектур [6]:

1. Один поток инструкций / Один поток данных (**ОКОД**, **SISD**, Single Instruction stream / Single Data stream));
2. Несколько потоков инструкций / Один поток данных (**МКОД**, **MISD**, Multiple Instruction stream / Single Data stream);
3. Один поток инструкций / Несколько потоков данных (**ОКМД**, **SIMD**, Single Instruction stream / Multiple Data stream);
4. Несколько потоков инструкций / Несколько потоков данных (**МКМД**, **MIMD**, Multiple Instruction stream / Multiple Data stream).

	Архитектура [8]	Описание [4]
1	 <p style="text-align: center;"><b>SISD</b></p>	<p>Представляет собой классическую модель с одним процессором. <b>К ней относятся как компьютеры предыдущих поколений, так и многие современные компьютеры.</b> Процессор такого компьютера способен выполнять в любой момент времени лишь одну инструкцию и работать лишь с одним набором данных. В таких последовательных системах никакого параллелизма нет в отличие от прочих категорий.</p>



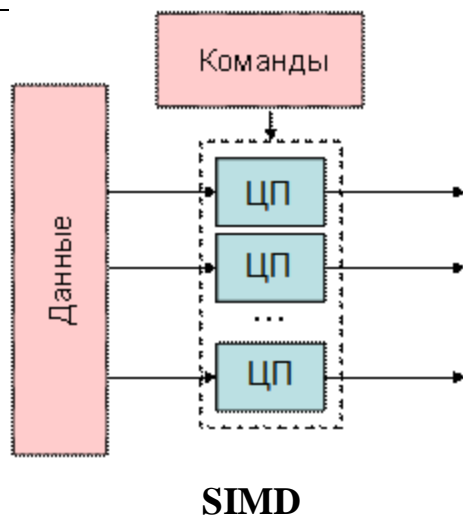
### MISD

Возможность одновременно выполнять различные операции над одними и теми же данными может поначалу показаться странной: не так уж часто встречаются программы, в которых нужно какое-то значение возвести в квадрат, умножить на 2, вычесть из него 10 и т.д. Однако, если посмотреть на эту ситуацию с другой точки зрения, то мы увидим, что на машинах такого типа можно усовершенствовать проверку числа на простоту. Если число процессоров равно  $N$ , то мы можем проверить простоту любого числа между 1 и  $N^2$  на MISD-машине за одну операцию: если число  $X$  составное, то у него должен быть делитель, не превосходящий  $X^{1/2}$ . Для проверки простоты числа  $X < N^2$  поручим первому процессору делить на 2, второму - делить на 3, третьему - на 4, и так до процессора с номером  $K - 1$ , который будет делить на  $K$ , где  $K = \lceil X^{1/2} \rceil$ . Если на одном из этих процессоров деление нацело проходит успешно, то число  $X$  составное. Поэтому мы достигаем результата за одну операцию. Как нетрудно видеть, на последовательной машине (SISD) выполнение этого алгоритма потребовало бы по меньшей мере  $\lceil X^{1/2} \rceil$  проходов, на каждом из которых производится одно деление.

*Напомним, что простое число - это такое натуральное число, большее 1, которое делится нацело только на само себя и на 1. Например, число 17 простое, поскольку у него нет делителей среди чисел от 2 до 16.*

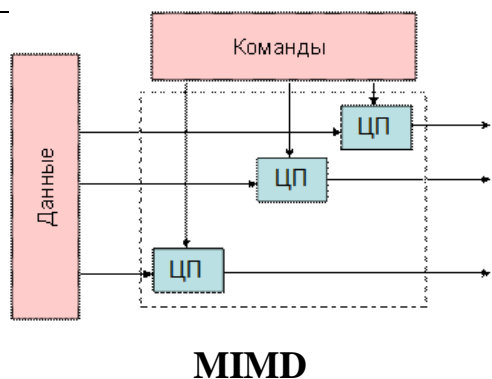
До сих пор не представлен убедительный пример реально существующей вычислительной системы MISD. Ряд исследователей относят конвейерные машины к данному классу [6].

3



В компьютерах с одним потоком инструкций и несколькими потоками данных (SIMD) имеется несколько процессоров, выполняющих одну и ту же операцию, но над различными данными. SIMD-машины иногда называются также векторными процессорами, поскольку они хорошо приспособлены для операций над векторами, в которых каждому процессору передается одна координата вектора и после выполнения операции весь вектор оказывается обработанным. Например, сложение векторов -- покоординатная операция. Первая координата суммы векторов -- сумма первых координат суммируемых векторов, вторая - сумма вторых координат, и т.д. В нашей SIMD-машине каждый процессор получит инструкцию сложить пару координат входных векторов. После выполнения этой единственной инструкции результат будет подсчитан полностью. Заметим, что на векторе из  $N$  элементов SIMD-машине потребуется выполнить  $N$  итераций цикла, а SIMD-машине с не менее, чем  $N$  процессорами, хватит **одной операции**. **Пример.** в мультипроцессоре графического процессора NVIDIA [6].

4



Это наиболее гибкая из категорий. В случае MIMD-систем мы имеем дело с несколькими процессорами, каждый из которых способен выполнять свою инструкцию. Кроме того, имеется несколько потоков данных, и каждый процессор может работать со своим набором данных. На практике это означает, что MIMD-система может выполнять на каждом процессоре свою программу или отдельные части одной и той же программы, или векторные операции так же, как и SIMD-конфигурация.

		<p><b>Пример.</b> Большинство современных подходов к параллелизму, включая <b>кластеры</b> компьютеров и <b>мультипроцессорные системы</b> - лежат в категории MIMD.</p>
--	--	--

## 8.5.2 Распределённые алгоритмы

**Модель** [9]: Имеется несколько независимых *процессов*, обычно обозначаются большими латинскими буквами:  $P, Q, R, \dots \in P$ . Происходящее внутри процесса нас не интересует: там может быть сложная многопоточная система или простой цикл; может быть детерминированным, а может кидать монетку.

Процессы могут посылать друг другу *сообщения*, обычно обозначаются  $m \in M$  (с индексами). Нас интересует взаимодействие между процессами, оно бывает ровно одного вида: посылки сообщений друг другу. С точки зрения внешнего мира процесс считается однопоточным: все отправки/получения сообщений одним процессом линейно упорядочены. Обычно каждый процесс может послать сообщение каждому, но иногда это ограничивается. В реальном мире может быть такое, что кому-то отослать сообщение быстрее, чем другому.

В каждом процессе могут происходить *события*, обычно обозначаются маленькими латинскими буквами:  $a, b, c, d, \dots \in E$ . Можно узнать процесс, в котором произошло событие:  $proc(e) \in P$  (такая нотация встречается редко). Если  $e$  и  $f$  произошли в одном процессе, то одно из двух произошло первым, обозначается  $e < f$ .

Каждому сообщению  $m$  соответствуют ровно два события: отправка сообщения  $snd(m) \in E$  и его получение  $rcv(m) \in E$ . Другие события нас не интересуют.

**Понятие** «Произошло-до (Happens-before)»:  $snd(m)$  произошло-до  $rcv(m)$



Транзитивное замыкание:

Если  $e < f$ , то  $e \rightarrow f$

Для любого сообщения  $m$ :  $snd(m) \rightarrow rcv(m)$

События  $a$  и  $c$ , не связанные отношением **произошло-до**, называются *параллельными*.

"Времени" или "глобального времени" в распределённых системах *не существует*. Следствие: слова "одновременно" и "текущий момент" запрещены, если только не подкрепляются определением на основе "произошло-до".

### 8.5.3 Объектно-ориентированный подход к разработке программ

ООП возникло в результате развития идеологии процедурного программирования, где данные и подпрограммы (процедуры, функции) их обработки формально не связаны. Для дальнейшего развития объектно-ориентированного программирования большое значение имели понятия события (в рамках событийно-ориентированного программирования) и компонента (в контексте компонентного программирования [10]).

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

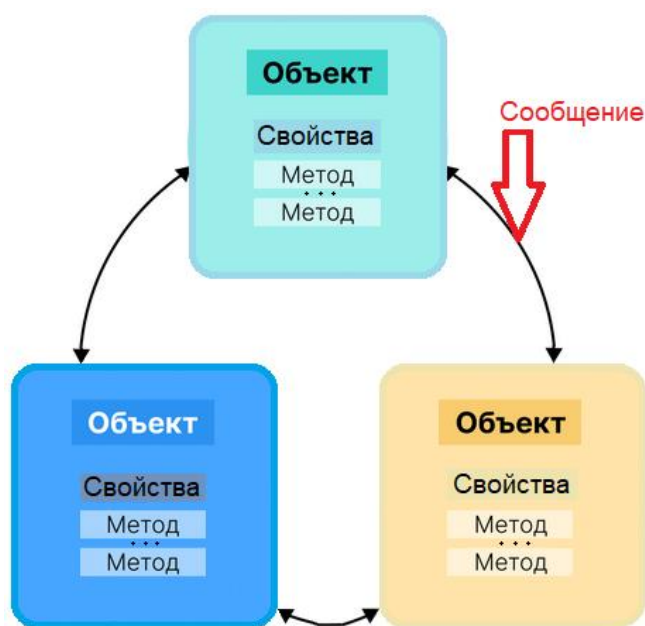
В языках программирования понятие объекта реализовано как совокупность **свойств** (структур данных, характерных для данного объекта), **методов** их обработки (подпрограмм изменения их свойств) и **событий**, на которые данный объект может реагировать и, которые приводят, как правило, к изменению свойств объекта.

При использовании данной методологии главными элементами программ являются объекты.

Понятие объекта в программе совпадает с обыденным смыслом этого слова: объект представляется как совокупность данных, характеризующих его состояние, и функций их обработки, моделирующих его поведение. Вызов функции на выполнение часто называют посылкой сообщения объекту.

**Пример.** Карточка товара в интернет-магазине, профиль пользователя, кнопка «купить» [11].

**Класс** – универсальный, комплексный **тип данных**, состоящий из тематически единого набора «полей /**свойств**» (переменных более элементарных типов) и «**методов**» (функций для работы с этими полями), то есть он является **моделью** информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей/свойств). Переменная-объект, относящаяся к заданному классом типу, называется экземпляром этого класса.



При создании объектно-ориентированной программы предметная область представляется в виде совокупности **объектов**. Выполнение программы состоит в том, что объекты обмениваются **сообщениями**.

Чтобы изменить программу, написанную с использованием ООП, меняют **свойства** или **методы** в **объекте**. При этом с другими объектами ничего не происходит — они продолжают работать как раньше [10].

#### 8.5.4 Методы искусственного интеллекта

Классификация методов искусственного интеллекта (ИИ) [12]:

1. Искусственные нейронные сети.
2. Нечеткая логика (нечеткие множества и мягкие вычисления).

3. Системы, основанные на знаниях (экспертные системы).
4. Эволюционное моделирование (генетические алгоритмы, многоагентные системы).
5. Machine Learning.

Охарактеризуем эти методы.

**1. Нейронная сеть (НС)** — математическая модель, прототипом которой служит центральная нервная система (ЦНС) человека или животного.

Данный метод ИИ применяется в задачах распознавания образов, прогнозирования, классификации, кластеризации и оптимизации.

**2. Нечеткая логика, теория нечетких множеств, нечеткие рассуждения, мягкие вычисления** — всё это близкие или тесно связанные между собой понятия, относящиеся к более высокому уровню работы ЦНС, нежели искусственные НС.

Нечеткая логика в большей степени связана с качественной оценкой анализируемых процессов и явлений и принятием решений на основе этой качественной оценки.

Методы нечеткой логики используются в экспертных системах, системах управления объектом.

### **3. Эволюционное или многоагентное моделирование.**

В рамках данной группы методов рассматривается концепция **не индивидуального, а коллективного интеллекта.**

Эволюционное моделирование целесообразно применять тогда, когда пространство поиска решения настолько большое и сложно устроенное, что традиционные и более простые методы просто неспособны выполнить глобальный поиск решения или способны, но на это потребуется неприемлемо много времени.

### **4. Системы, основанные на знаниях (экспертные системы, ЭС)**

ЭС — это искусственный аналог ЛПП или эксперта-консультанта предметной области. Структура и логико-математический аппарат ЭС определяются её назначением и предметной областью. Решения,

предлагаемые системой, могут вырабатываться с использованием различных механизмов вывода. Наиболее близкий аналог человеческому механизму вывода — это аппарат нечеткой логики и теории нечетких множеств

## 5. Machine Learning

**Machine Learning (машинное обучение)** — это целый класс методов искусственного интеллекта. Все они подразумевают решение задач не напрямую, а путем предварительного обучения как до, так и в процессе принятия решения.

**Data mining** (интеллектуальный анализ данных, поиск закономерностей в хранилищах данных)

Данный термин введен Григорием Пятецким-Шапиро в 1989 г.

Это собирательное название, которое применяется для обозначения целой группы методов обнаружения определенных закономерностей в общем объеме данных, которые могут получены в различных сферах человеческой деятельности.

**Пример.** Методы **Data Mining** могут быть использованы для больших данных (**Big Data**), накопленных в розничных продажах, для подтверждения каких-либо гипотез и принятия управленческих решений.

### Примеры российских систем ИИ [13]:

1. «**Роснефть**» исследовала керн: наукоемкий программный комплекс «РН – Цифровой керн» создает цифровой двойник горной породы – объемную копию породы, что помогает определить количество нефти и газа в пласте.

2. «**Газпром нефть**» нашла нефть в ХМАО и Томской области (НС обрабатывает информацию со скважин, обнаруживает закономерности и определяет месторасположение нефти).

3. «**МегаФон**» управляет транспортными потоками при строительстве крупных промышленных предприятий.

4. **SmartAGRO** прогнозирует урожай на протяжении всего сельскохозяйственного сезона.

5. «**ВкусВилл**» сгенерировал пейзажи для дизайна упаковок новой собственной линейки макаронных изделий.

6. «**Сбербанк**» урегулировал задолженности клиента.

7. «**Альфа-Банк**» (не) одобрил ипотеки.

8. **СДЭК** подключила к боту в Viber умный сервис «Мегамозг» для общения с клиентами в чатах.

9. «**Сбербанк**» создал SberMedAi (платформу в области диагностики заболеваний)

#### **Примеры российских систем ИИ [14]:**

1. Генератор пьес НейроСтаниславский (НИТУ «МИСиС»).
2. Голосовые помощники Алиса (Яндекс), Маруся (VK), Салют (Сбер).
3. Colorize (А.Кожевин) для раскрашивания чёрно-белых фото и видео
4. Инструмент для реставрации старых фотографий (VK).
5. Инструмент перефразирования текстов ReText.AI (ИП Шкряба О.С.)
6. Сервис развлечений для генерации текстов Балабоба (Яндекс).
7. Шедеврум (Яндекс) для генерации картинок по текстовому описанию
8. Генератор картинок по текстовому описанию Kandinsky (Сбер)
9. Сервис закадрового перевода видео (Яндекс).
10. Система синтеза речи Steos Voice (Mind Simulation).
11. Умная камера (Яндекс)

#### **Примеры систем ИИ [15]:**

1. Генераторы **AlphaGo, AlphaGo Zero** (Google DeepMind) – НС, смогла победить человека-чемпиона мира по игре Го

2. Виртуальные помощники **Siri (Apple), Alexa (Amazon)** используют НС для **распознавания речи**

3. **NVIDIA StyleGAN2** – генеративно-состязательная сеть (Generative adversarial network, GAN) используется для генерации реалистичных изображений

4. **Tesla Autopilot (Tesla), Drive AGX (NVIDIA)** – используют НС для управления беспилотными автомобилями

5. Системы распознавания лиц **DeepFace (Facebook), FaceNet (Google)**
5. **Google Translate и DeepL** – обработка естественного языка (NLP, Natural Language Processing)
6. **Predix (General Electric)** – система прогнозируемого технического обслуживания ветряных турбин
7. **PayPal** – использует систему обнаружения мошенничества
8. Рекомендательные системы – используются **Amazon, Netflix Spotify**

## **8.6 Понятие вычислительной сложности (по времени и памяти) алгоритма и его применение для анализа алгоритмов**

### **8.6.1 Понятие сложности алгоритма**

В общем случае сложность алгоритма можно оценить с разных точек зрения:

***Вычислительная сложность***, определяющая скорость выполнения.

***Ёмкостная сложность***, определяющая объём памяти, необходимый для выполнения алгоритма.

***Трудоёмкость разработки***, определяющая время, необходимое для разработки самого алгоритма решения задачи, его формального описания

**В каких единицах измерять сложность?**

### **8.6.2 Единицы измерения сложности алгоритма**

При выборе единицы измерения необходимо помнить о том, что оценивается *качество самого алгоритма*, а не *быстродействие процессора* или *эффективность реализации компиляторов*, которые по одному и тому же исходному коду на языке программирования высокого уровня могут сгенерировать различный код на машинном языке, в системе команд одного и того же процессора.

Следовательно:

– Нельзя оценивать *временную сложность* в **часах-минутах-секундах...**: эта характеристика будет зависеть от свойств исполнителя (архитектуры процессора, его быстродействия).

– Нельзя оценивать *ёмкостную сложность* в **байтах-килобайтах-мегабайтах...**: данные в памяти компьютера могут быть представлены в разных форматах, которые определяются особенностями системы программирования, реализации.

Абстрагируясь от особенностей представления алгоритма, характеристик исполнителя, выберем в качестве единиц измерения следующие:

– Будем оценивать *временную сложность* **количеством операций**, выполняемых при исполнении алгоритма (хотя сложность различных команд в системе команд исполнителя будет разной, и для их выполнения требуется в общем случае разное время).

– Будем оценивать *ёмкостную сложность* в **количестве единиц данных** (хотя разные данные, для представления которых при выполнении алгоритма будут использоваться разные форматы, потребуют разного объема памяти при выполнении программы, реализующей алгоритм).

Чтобы более точно оценить время, которое потребуется для выполнения алгоритма, часто *все операции разбивают на группы в соответствии с их сложностью*:

- 1 – команды пересылки;
- 2 – команды сравнения;
- 3 – операции целочисленной арифметики;
- 4 – операции над данными в формате с плавающей точкой...

Для каждой группы команд получают оценку: **сколько команд из данной группы будет выполняться при решении задачи с использованием разработанного алгоритма.**

Обычно оценивают только количество самых «сложных» команд (например, операций ввода и вывода).

## 8.7 Основные методы и приёмы анализа сложности

### 8.7.1 Оценка исходных данных

В общем случае количество операций и требуемая память зависят от исходных данных, т.е. являются функциями вектора исходных данных

$$X = (x_1, x_2, \dots, x_n)$$

#### Как оценить сложность самих исходных данных?

Рассмотрим два фрагмента кода:

```
...  
Read (N);  
F:=1;  
for l:=1 to N do F:=F*l;  
Write (F);  
...
```

```
...  
S:=1;  
for l:=1 to N do  
begin  
  Read(A[l],B[l]); S:=S+A[l]*B[l];  
end;  
Write (S);  
...
```

Сложность алгоритма определяется значением переменной N

```
...  
Read (N);  
F:=1;  
for l:=1 to N do F  
Write (F);  
...
```

**Какая оценка исходных данных (значение или размерность) является более общей?**

Сложность алгоритма определяется размерностью данных (числом элементов в векторах A и B)

```
...  
to N do  
[l],B[l]); S:=S+A[l]*B[l];  
end;  
Write (S);  
...
```



### 8.7.2 Оценка исходных данных и алгоритма

Исходные данные могут быть *нечисловыми* (графы, географические карты, строки символов, звуки и т.д.), поэтому сложность алгоритма  $\alpha$  рассматривается как *функция от некоторого интегрированного числового параметра  $V$ , характеризующего исходные данные*.

Обозначим:

–  $T_\alpha(V)$  – *временная сложность* алгоритма  $\alpha$ , которая определяет время, необходимое для решения задачи с использованием алгоритма  $\alpha$  в зависимости от  $V$ ;

–  $S_\alpha(V)$  – *ёмкостная сложность*, которая определяет необходимый для решения задачи объем памяти  $\alpha$  в зависимости от  $V$ .

Параметр  $V$ , характеризующий данные, называют иногда *сложностью данных*.

Выбор параметра  $V$  (интегральной характеристики входных данных) зависит не только от объёма и даже вида данных, но и от вида алгоритма или от задачи, которую этот алгоритм решает.

### 8.7.3 Оценка сложности алгоритма с использованием управляющего графа

*Управляющий граф* строится следующим образом:

1. Имеется *стартовая вершина* (вход в алгоритм).
2. Каждому оператору, включённому в алгоритм (оператору присваивания, вызову процедуры и пр.), ставится в соответствие *вершина графа* (точка), имеющая *метку* (вес, равный сложности этого оператора).
3. Два последовательно записанных оператора (последовательно исполняемых) изображаются вершинами, соединёнными стрелкой, указывающей порядок их исполнения.

4. Условный оператор задаёт **разветвление** в управляющем графе: начинается с вершины, соответствующей *вычислению условия*, с двумя выходящими дугами с приписанными им вариантами *then, else* (т.е. *True* и *False*), которые указывают вершины, представляющие операторы, исполняющиеся при выполнении и невыполнении условия; заканчивается *пустой вершиной* (без операций).

5. Циклы реализуются через **ветвления**.

6. Имеется вершина, отмечающая завершение алгоритма (**конечная**, финишная)

**Максимальная сложность алгоритма** с помощью управляющего графа оценивается как **длина самой «длинной» ветви** – считаются суммы весов всех вершин, входящих в каждую ветвь, соединяющую стартовую (начальную) и конечную (финишную) вершины, и среди них выбирается максимум.

**Минимальная сложность алгоритма** с помощью управляющего графа оценивается как **длина самой «короткой» ветви** – считаются суммы весов всех вершин, входящих в каждую ветвь, соединяющую стартовую (начальную) и конечную (финишную) вершины, и среди них выбирается минимум

Если не оговорено отдельно, то за одну условную операцию будем считать каждую арифметическую или логическую операцию, а также операции пересылки данных.

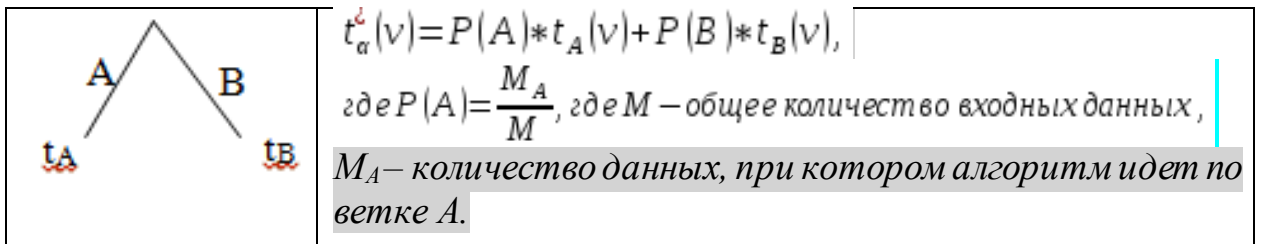
Линейные алгоритмы [16]:

$$t_{\alpha}(v) = t_{\alpha}^{\downarrow}(v) = \dot{t}_{\alpha}(v) = \sum \text{составных частей}$$

Ветвление [16]:

$$t_{\alpha}(v) = \min(t_A(v), t_B(v))$$

$$\dot{t}_{\alpha}(v) = \max(t_A(v), t_B(v))$$



Циклы [16]:

1) заранее известно количество повторений:

$$t_{\alpha}(v) = t_{\alpha}'(v) = t_{\alpha}' = \sum_{i=1}^n t_c(i), \text{ где } t_c(i) - \text{сложность тела цикла}$$

2) заранее неизвестно количество повторений: универсальных методов нет.

## 8.8 Сложность алгоритмов с ветвлениями, циклами

### 8.8.1 Пример. Алгоритм с ветвлением

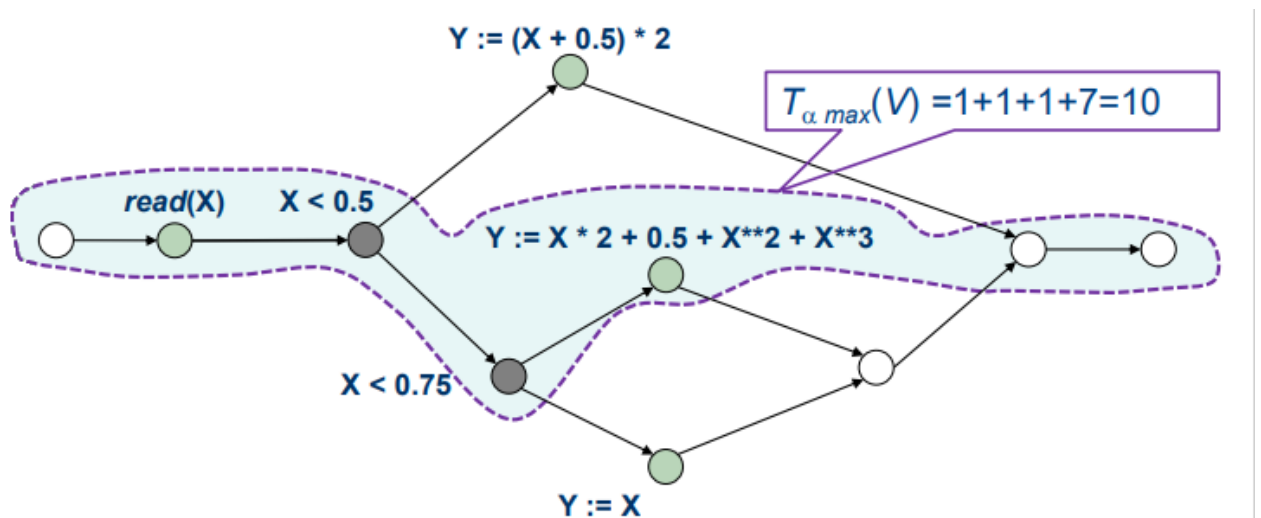
Пусть алгоритм имеет вид

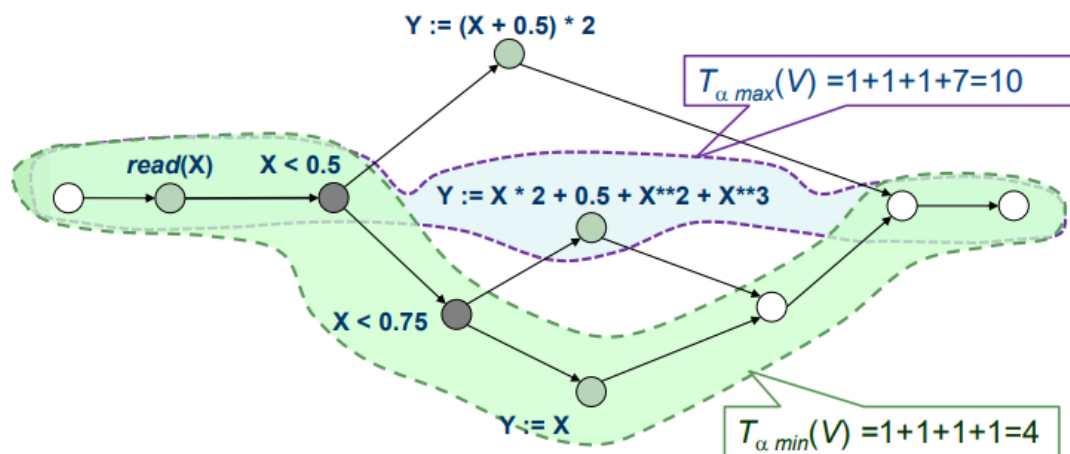
```

read(X);
if X < 0.5 then Y := (X + 0.5) * 2
else if X < 0.75 then Y := X * 2 + 0.5 + X**2 + X**3
else Y := X

```

Максимальная сложность алгоритма:





**Как найти среднюю сложность?**

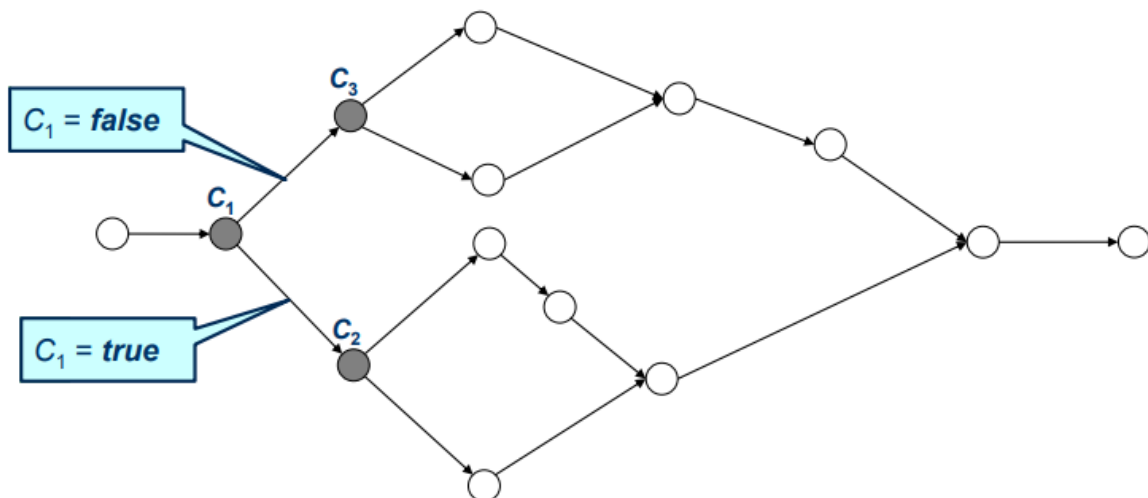
### 8.8.2 Алгоритм с ветвлением. Оценка сложности на основе оценки входных данных

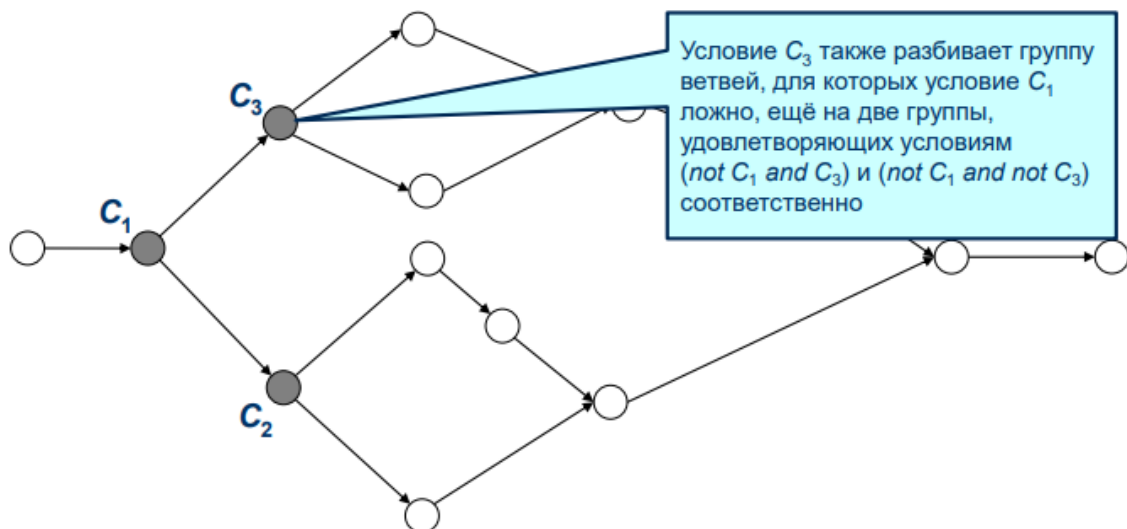
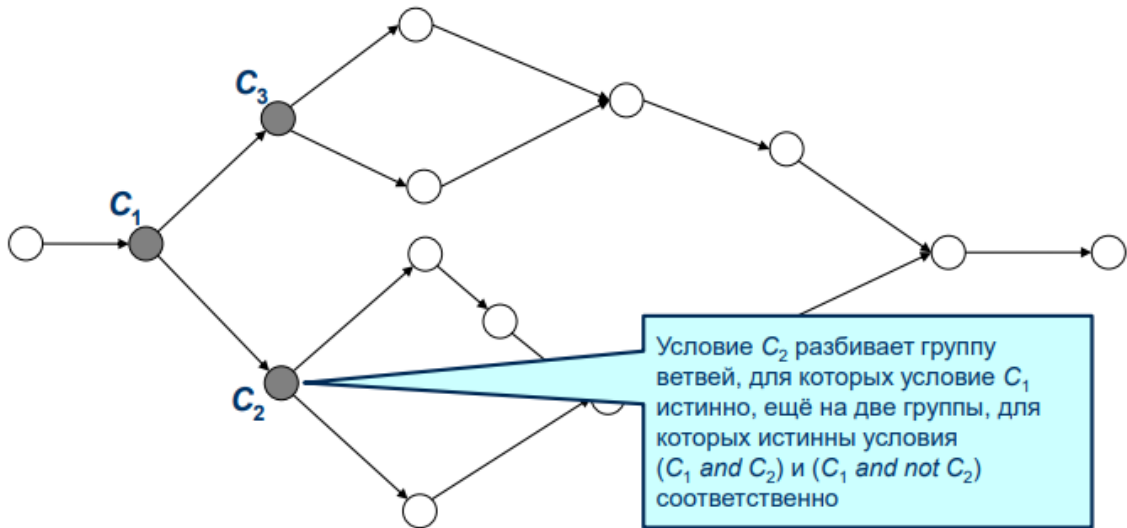
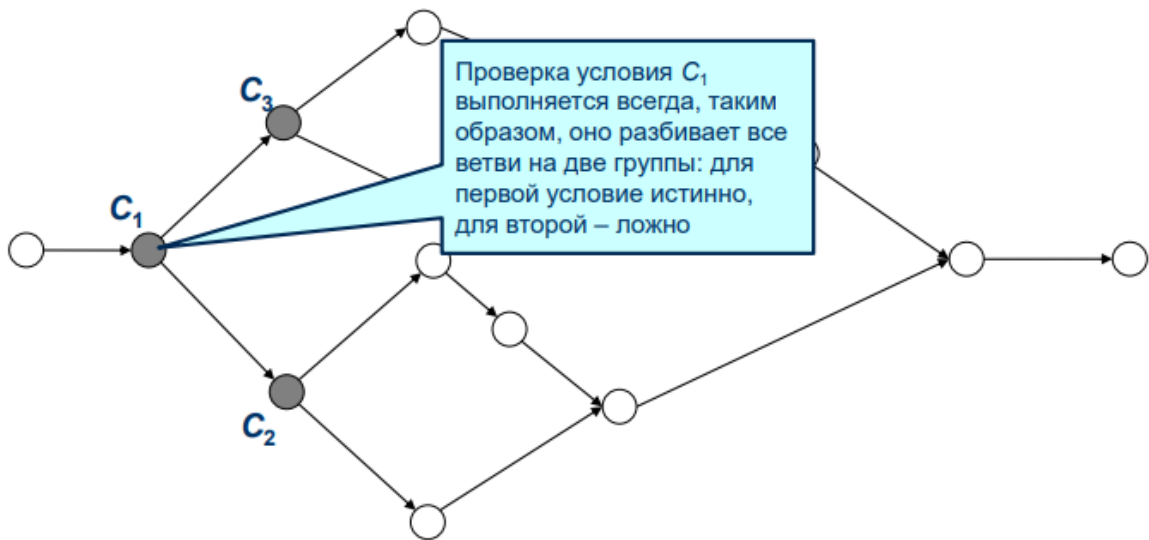
Пусть программа содержит ветвления, где выбор ветвей определяется условиями  $C_1, C_2, C_3, \dots, C_n$ :

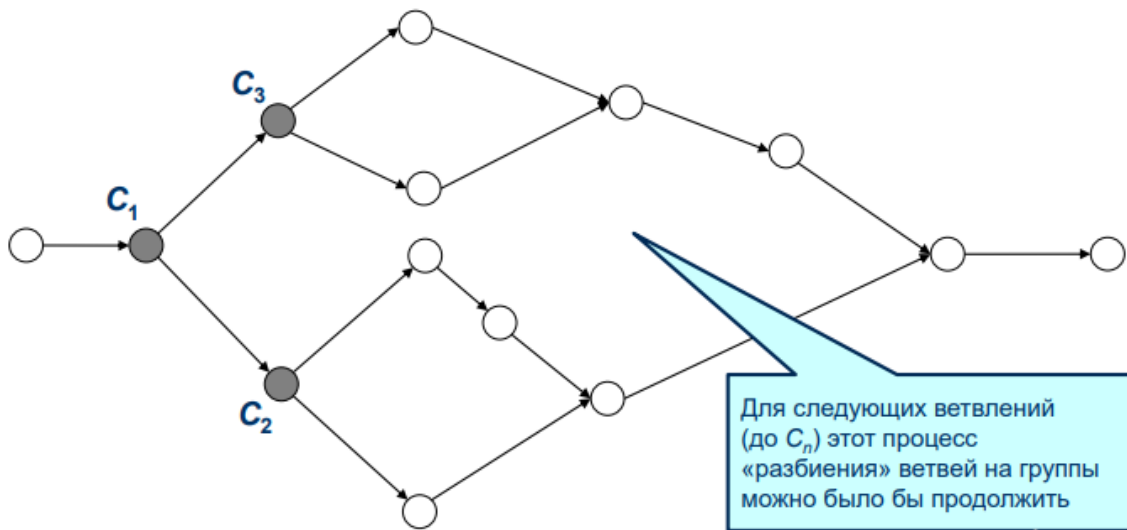
```

if C1 then
  if C2 then ...
  else ...
else if C3 then
  else ...
  
```

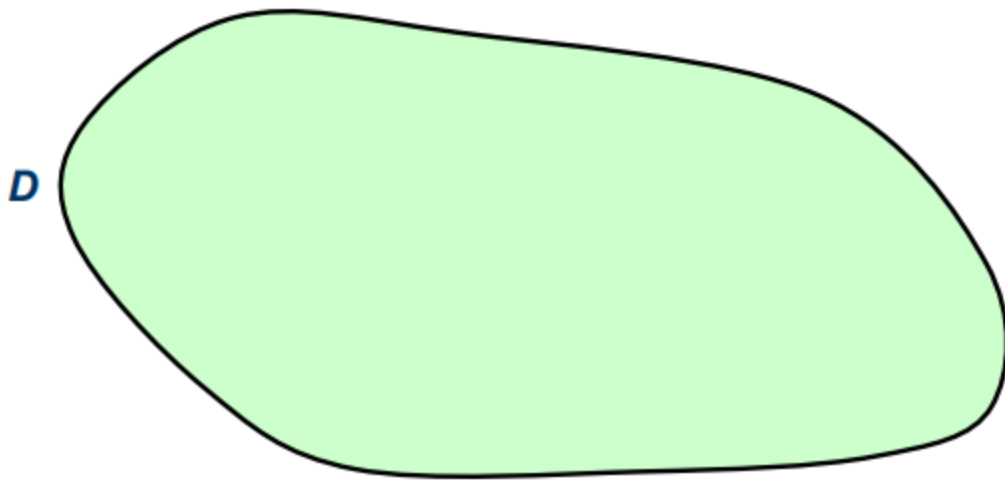
Пусть управляющий граф программы содержит ветвления, где выбор ветвей определяется условиями  $C_1, C_2, C_3, \dots, C_n$ :

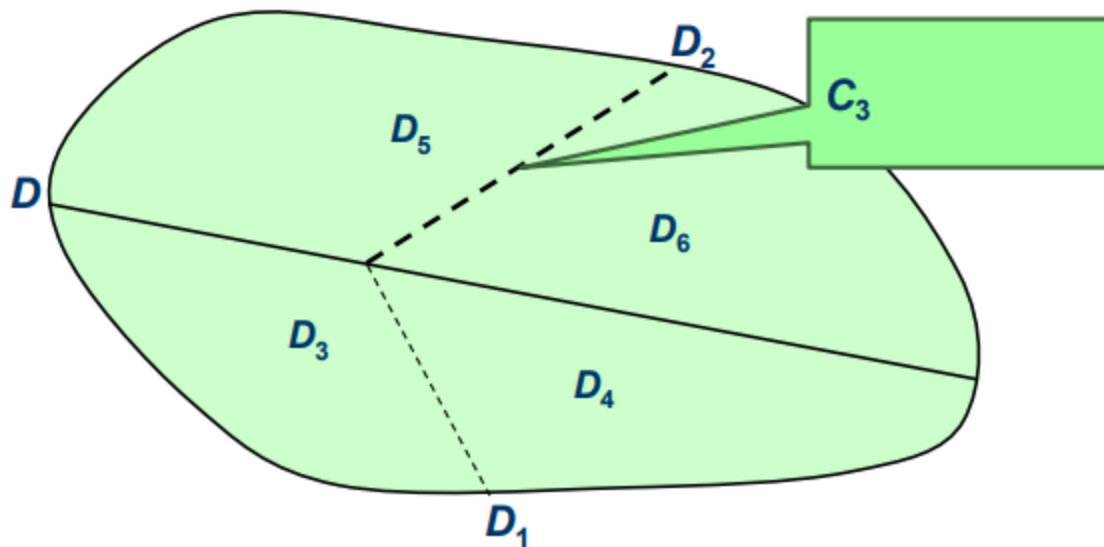
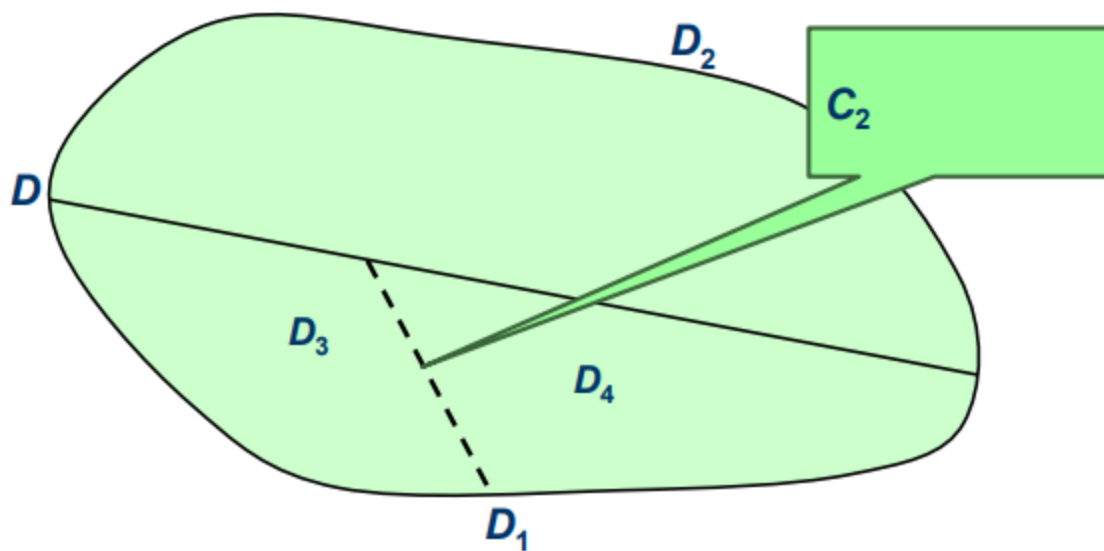
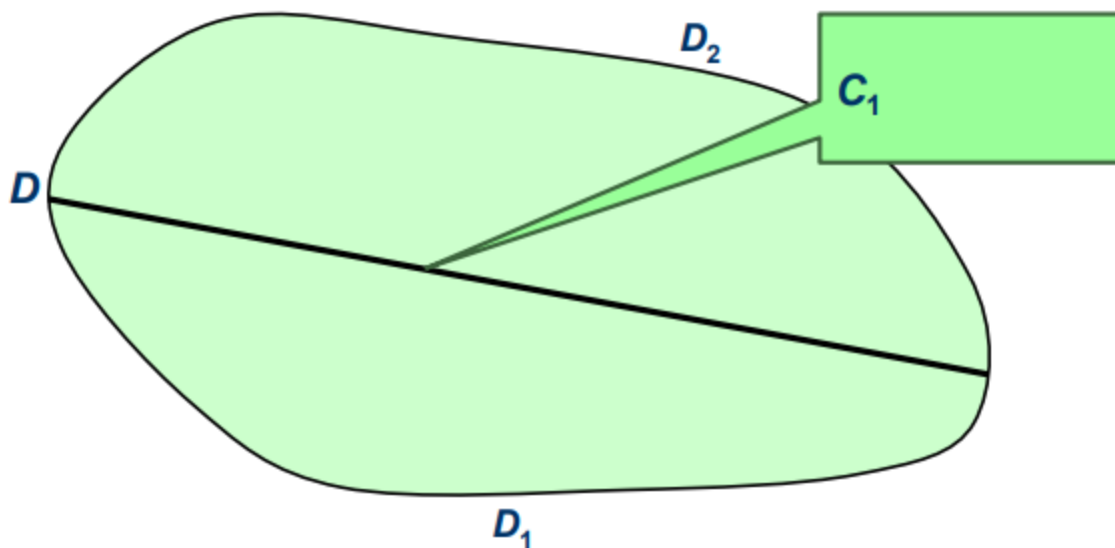






Выбор ветвей определяется *комбинацией входных данных  $D$* . Множество комбинаций исходных данных  $D$  также можно разбить на две части  $D_1$  и  $D_2$  (два подмножества), порождаемые условием  $C_1$ . В свою очередь  $D_1$  делится на  $D_3$  и  $D_5$ , а  $D_2$  разбивается на  $D_5$  и  $D_6$  (по соответствующим условиям).





Каждая область (подмножество  $D$ ), которая не разбивается больше ни на какие другие более мелкие части, соответствует ветви в управляющем графе

(ветви пересекаются в начальной части графа – для общих условий, определяющих их выбор).

Все такие подмножества комбинаций входных данных (областей на рисунке) обозначены  $d_i$ . В том случае, когда все комбинации исходных данных являются равновероятными, частоту исполнения  $i$ -й ветви можно оценить как отношение мощности подмножества  $d_i$  к мощности всего множества  $D$ :  $p_i = |d_i| / |D|$ .

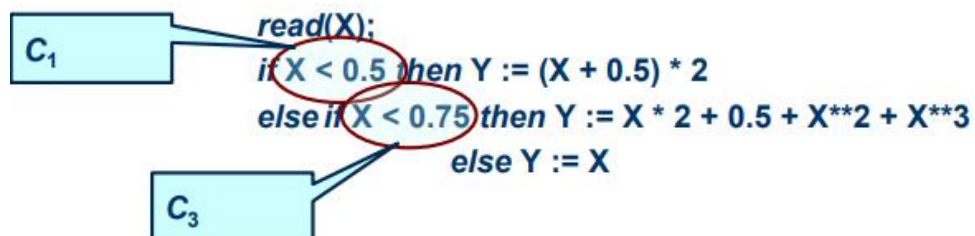
Если имеются оценки  $l_i$  веса (сложности)  $i$ -й ветви, то сложность такого алгоритма  $\alpha$  с ветвлениями можно оценить величиной

$$T_{\alpha_{Avg}} = \sum_{i=1}^k \frac{|d_i|}{|D|} l_i$$

где  $k$  – количество всех ветвей.

### 8.8.3 Пример оценки сложности алгоритма с ветвлениями

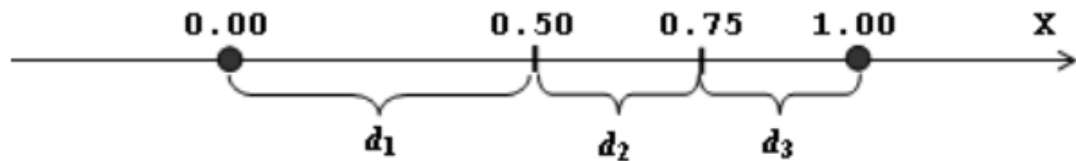
Пусть алгоритм с ветвлением имеет вид



Допустим, что значение переменной  $X$  может принадлежать диапазону  $[0.00, 1.00]$ , причём ввод любого значения из указанного диапазона *равновероятен*. Тогда вероятность того, что значение  $X \in [0.00, 0.50)$ , примерно равна вероятности того, что  $X \in [0.50, 1.00]$ . В приведённом примере вероятность выбора ветви  $Y := (X + 0.5) * 2$  равна 0.5. При невыполнении условия  $X < 0.5$  выбор ветви определяется проверкой условия  $X < 0.75$ .



Разбиение всего множества комбинаций (в данном случае каждая комбинация определяется только значением переменной  $X$ ) показано на рисунке:



*Геометрическая вероятность ввода значений переменной, определяемая длинами отрезков числовой прямой (значения равновероятны)*

Таким образом, при выполнении программы возможен выбор *трёх* ветвей:

1) ветви  $Y := (X + 0.5) * 2$ , соответствующей условию  $X < 0.5$ , вес (сложность) которой составляет **5** (ввод, проверка условия, вычисление выражения (2 операции) и присваивание), а вероятность выбора равна **0.5**;

2) ветви  $Y := X * 2 + 0.5 + X * 3$ , соответствующей условию **not**( $X < 0.5$ ) **and** ( $X < 0.75$ ), вес (сложность) которой составляет **10** (ввод, проверка условия  $X < 0.5$ , проверка условия  $X < 0.75$ , вычисление выражения (6 операций) и присваивание), а вероятность выбора равна **0.25**;

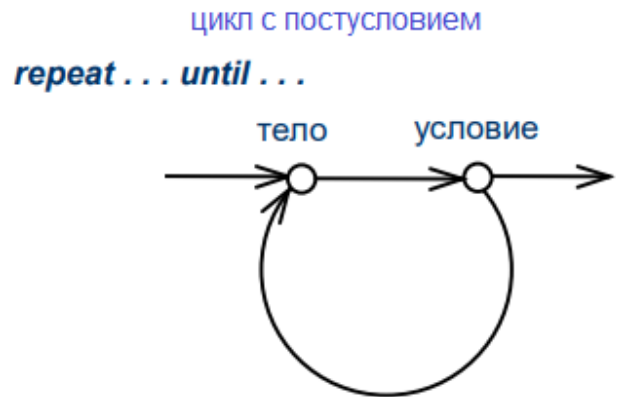
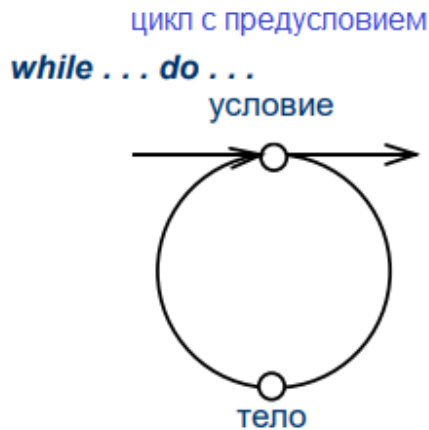
3) ветви  $Y := X$ , соответствующей условию **not**( $X < 0.5$ ) **and** **not**( $X < 0.75$ ), вес (сложность) которой составляет **4** (ввод, проверка условия  $X < 0.5$ , проверка условия  $X < 0.75$ , и присваивание), а вероятность её выбора – примерно **0.25**. Средняя сложность алгоритма составляет

$$T_{Avg} = 0,5 \times 5 + 0,25 \times 10 + 0,25 \times 4 = 2,5 + 2,5 + 1 = 6$$

#### 8.8.4 Алгоритмы, содержащие циклы с предусловием и постусловием

*Операторы цикла* изображаются в управляющем графе замкнутой последовательностью вершин – **циклом**. Если выражения, определяющие начальное и конечные значения параметра цикла – **константы**, то можно

посчитать, сколько раз будет выполняться цикл, и умножить это число на вес (сложность) тела цикла. Если число повторений **зависит от исходных данных**, то необходимо оценить значение разности между начальным и конечным значениями в худшем, лучшем и среднем случаях.



### 8.8.5 Оценка алгоритмов, содержащих циклы со счётчиком

**Пример 1.** Процедура содержит цикл вида

*for I := 1 to X do Тело цикла*

Для оценки сложности данного фрагмента кода нужно *оценить сложность тела цикла* и *умножить на число его выполнений* в цикле.

Допустим, что переменная  $X$  может принимать значения от 0 до 100 с равной вероятностью (1/101).

Тело цикла выполняется  $X$  раз, причём худший случай реализуется при  $X = 100$ , а наилучший – при  $X = 0$ .

*Среднее число* выполнений тела цикла равно

$$t_{Avg} = (1/101 \times \underline{0} + 1/101 \times \underline{1} + 1/101 \times \underline{2} + \dots + 1/101 \times \underline{100}) \times 101 =$$

$$= (1/101 \times (\underline{0+1+2+\dots+100})) \times 101 = 1/101 \times (0+100)/2 \times 101 = 50.$$

Для более точной оценки оценивается и сложность операций, записанных в заголовке цикла.

**Пример 2.** Процедура содержит вложенные циклы вида

*for i := 1 to X do*

*for j := i to X do Тело цикла*

Допустим, что переменная  $X$  может принимать значения от 1 до 5 с равной вероятностью (1/5).

Внешний цикл выполняется  $X$  раз. Тело цикла выполняется

$$X \times (X + (X - 1) + (X - 2) + \dots + 1) = X \times (X + 1)/2$$

раз, причём худший случай реализуется при  $X = 5$ , а наилучший – при  $X = 1$ .

Таким образом, верхняя граница сложности равна

$$X \times (X+1)/2 = 5 \times (5+1)/2 = 15 .$$

Минимальное число выполнений тела цикла равно

$$X \times (X+1)/2 = 1 \times (1+1)/2 = 1 .$$

Среднее число выполнений тела цикла равно

$$\begin{aligned} \sum_{X=1}^5 \left( \frac{1}{5} \times X \frac{X+1}{2} \right) &= \frac{1}{10} \sum_{X=1}^5 (X \times (X+1)) = \frac{1}{10} (1 \times 2 + 2 \times 3 + 3 \times 4 + 4 \times 5 + 5 \times 6) = \\ &= \frac{1}{10} (2 + 6 + 12 + 20 + 30) = \frac{70}{10} = 7. \end{aligned}$$

**Пример 3.** Оценим сложность алгоритма умножения двух квадратных матриц  $A$  и  $B$  размерностью  $N \times N$  и получения в качестве результата третьей матрицы  $C$ :

```

for i := 1 to N do {Цикл 1}
  for j := 1 to N do {Цикл 2}
    begin
      C[i,j] := 0;
      for k := 1 to N do {Цикл 3}
        C[i,j] := C[i,j] + A[i,k] * A[k,j]
      end
    end
  end
end

```

В качестве параметра, определяющего сложность данных, в этом случае можно взять  $N$ .

Сложность алгоритма определяется величиной

$$T_{\alpha}(N) = N \times (T_1) ,$$

где  $T_1$  – сложность тела цикла 1 по  $i$ , которая равна

$$T_1 = N \times (T_2) ,$$

где  $T_2$  – сложность тела цикла 2 по  $j$  равна

$$T_2 = 1 + N \times (T_3) .$$

Здесь сложность тела цикла 3 по  $k$  ( $T_3$ ) равна 3 (умножение, сложение и присваивание), а 1 – это операция присваивания начального значения элементу матрицы  $C$ .

Выполняем подстановку и получаем выражение сложности алгоритма через сложность исходных данных:

$$T_{\alpha}(N) = N \times (N \times (1 + N \times 3)) = N \times (N + N^2 \times 3) = N^2 + N^3 \times 3 = 3N^3 + N^2.$$

## 8.9 Сложность рекурсивных алгоритмов

### 8.9.1 Сложность простой рекурсии

Условие, заданное в рекурсивном алгоритме, делит все комбинации входных данных на **два подмножества**: для первого выполняется *условие завершения рекурсивных вызовов* – присваивание начального значения и возврата из рекурсии (обратного хода), для второго *рекурсивные вызовы продолжаются*.

Необходимо установить зависимость числа рекурсивных вызовов от того, как меняются данные. Эти оценки в каждом случае выполняются «индивидуально», но все операции в алгоритме можно разбить на группы:

1. операции, которые выполняются при *проверке условия* (они выполняются всегда);
2. операции, выполняемые по *нерекурсивной ветви* (выполняются **один раз** – перед выходом из рекурсии, это начало обратного хода рекурсии);
3. операции, которые выполняются при выборе *рекурсивной ветви* (каждый раз, когда проверка условия приводит к рекурсивному вызову). Эти операции также разбиваются на группы:

3.1. операции, понижающие сложность исходных данных при рекурсивном вызове;

3.2. операции, выполняемые при рекурсивном вызове с новыми данными, сложность которых стала ниже (*это сложность самого рекурсивного алгоритма*);

3.3. операции, выполняемые каждый раз при выборе этой рекурсивной ветви, но не связанные с рекурсивным вызовом или понижением сложности данных.

В общем случае для простой рекурсии (алгоритм  $\alpha$ ) можно построить следующее рекуррентное соотношение:

$$T_{\alpha}(V_{in}) = T_{Cond}(V_{in}, S) + (T_H(V_{in}) + T_{\alpha}(f(V_{in})) + T_f(V_{in})),$$

где  $T_{Cond}(V_{in}, S)$  – сложность проверки условия;

–  $T_f(V_{in})$  – сложность вычислений, понижающих сложность исходных данных;

–  $T_{\alpha}(f(V_{in}))$  – сложность операций, выполняемых при рекурсивном вызове с новыми данными («упрощенными») данными;

–  $T_H(V_{in})$  – сложность вычислений, выполняемых каждый раз при выборе рекурсивной ветви, но не связанных с рекурсивным вызовом или понижением сложности данных.

Имеем уравнение для  $T_{\alpha}(V_{in})$ .

Его можно решить, используя известное начальное условие, которое описывает сложность выбора и выполнения нерекурсивной ветви:

$$T_{\alpha}(S) = T_{Cond}(S, S) + T_D(S)$$

В общем случае получаем систему:

$$T_{\alpha}(V_{in}) = T_{Cond}(V_{in}, S) + (T_H(V_{in}) + T_{\alpha}(f(V_{in})) + T_f(V_{in}))$$

$$T_{\alpha}(S) = T_{Cond}(S, S) + T_D(S).$$

В это соотношение могут быть также дополнительно включены оценки сложности операций, которые выполняются каждый раз при вызове перед проверкой условия и после условного оператора – перед возвратом из алгоритма.

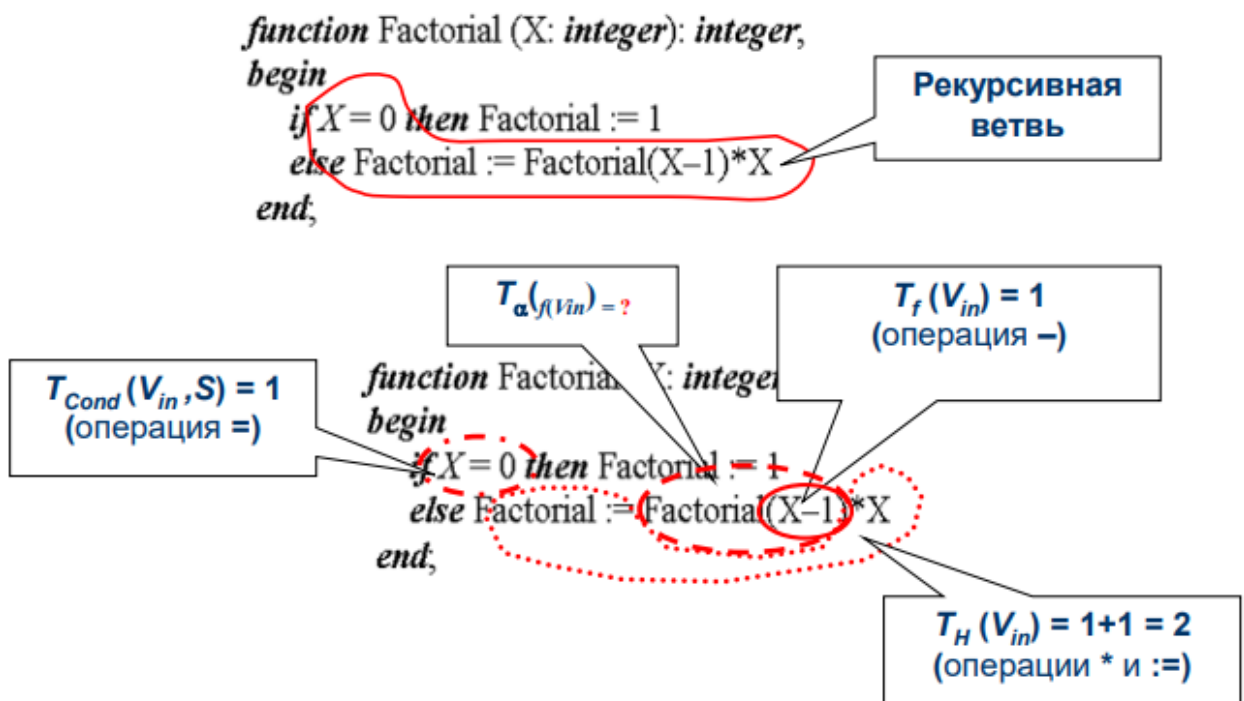
## 8.9.2 Пример оценки сложности простой рекурсии

### Пример. Вычисление факториала

Оценка сложности функции вычисления факториала ( $Factorial(X)$  при  $X \geq 0$ ):

```
function Factorial (X: integer): integer;  
begin  
  if X = 0 then Factorial := 1  
  else Factorial := Factorial(X-1)*X  
end;
```

Выделим в данном коде перечисленные выше фрагменты, определяющие сложность вычислений.



```

function Factorial (X: integer): integer,
begin
  if X = 0 then Factorial := 1
  else Factorial := Factorial(X-1)*X
end,

```

Нерекурсивная  
ветвь

$T_{Cond}(S, S) = 1$   
(операция =)

```

function Factorial (X: integer): integer,
begin
  if X = 0 then Factorial := 1
  else Factorial := Factorial(X-1)*X
end,

```

$T_D(S) = 1$   
(операция :=)

Получаем систему:

$$T_{\alpha}(X) = T_{Cond}(0, 0) + (T_H(V_n) + T_{\alpha}(X-1) + T_f(X-1))$$

$$T_{\alpha}(0) = T_{Cond}(0, 0) + T_D(0) .$$

Делаем подстановку сложности выделенных фрагментов алгоритма:

$$T_{\alpha}(X) = 1 + (2 + T_{\alpha}(X-1) + 1)$$

$$T_{\alpha}(0) = 1 + 1$$

Получаем:

$$T_{\alpha}(X) = 4 + T_{\alpha}(X-1)$$

$$T_{\alpha}(0) = 2$$

Рассмотрим первое уравнение (рекуррентное) для данного случая:

$$\begin{aligned}
 T_{\alpha}(X) &= 4 + T_{\alpha}(X-1) = \\
 &= 4 + (4 + T_{\alpha}(X-2)) = \\
 &= 4 + (4 + (4 + T_{\alpha}(X-3))) = \dots = \\
 &= 4 + (4 + (4 + \dots + (4 + T_{\alpha}(1)) \dots)) = \\
 &= 4 + (4 + (4 + \dots + (4 + (4 + T_{\alpha}(0)) \dots)) = \\
 &= 4 + (4 + (4 + \dots + (4 + (4 + 2)) \dots)) = 4 \times X + 2
 \end{aligned}$$

Получили итоговую оценку сложности алгоритма:

$$T_{\alpha}(X) = 4 \times X + 2$$

### 8.9.3 Сравнение оценки сложности простой рекурсии и итерации

**Пример.** Оценим сложность алгоритма вычисления факториала с помощью итерации. Оценка сложности итерационного алгоритма

```
function Factorial (X: integer): integer;  
    var M, i: integer;  
begin  
    M := 1;  
    for i := 2 to X do M := M*i;  
    Factorial := M  
end;
```

будет следующей:

$$T_{\alpha}(X) = 1 + (X - 1) \times (1 + 1) + 1 = (X - 1) \times 2 + 2 = X \times 2$$

Сравнение рекурсивного и итерационного алгоритмов показывает, что сложность последнего **ниже**.





## Отличия параллельных и распределенных алгоритмов

**Принципы анализа параллельных алгоритмов** [4]. При работе с параллельными алгоритмами нас будут интересовать два новых понятия: **коэффициент ускорения** и **стоимость**. Коэффициент ускорения параллельного алгоритма показывает, насколько он работает быстрее оптимального последовательного алгоритма. Так, мы видели, что оптимальный алгоритм сортировки требует  $O(N \log N)$  операций. У параллельного алгоритма сортировки со сложностью  $O(N)$  коэффициент ускорения составил бы  $O(\log N)$ .

Второе интересующее нас понятие – стоимость параллельного алгоритма, которую мы определяем как **произведение сложности алгоритма на число используемых процессоров**. Если в нашей ситуации алгоритм параллельной сортировки за  $O(N)$  операций требует столько же процессоров, каково число входных записей, то его стоимость равна  $O(N^2)$ . Это означает, что алгоритм параллельной сортировки обходится дороже, поскольку стоимость алгоритма последовательной сортировки на одном процессоре совпадает с его сложностью и равна поэтому  $O(N \log N)$ .

Близким понятием является **расщепляемость** задачи. Если единственная возможность для нашего алгоритма параллельной сортировки состоит в том, что число процессоров должно равняться числу входных записей, то такой алгоритм становится бесполезным как только число входных записей оказывается достаточно большим. Никаких похожих ограничений в алгоритме последовательной сортировки нет. По большей части нас будут интересовать такие алгоритмы параллельной сортировки, в которых число процессоров значительно меньше потенциального объема входных данных и это число не требует увеличения при росте длины входа.

**Отличия распределенных вычислений от параллельных систем** [9]. Самая дорогая и длительная операция в распределённых системах обычно **не вычисления**, а **посылка сообщения**, потому что это включает в себя сеть (задержки между континентами порядка сотен миллисекунд). Так

что в алгоритмах нас интересует не время вычислений, а количество посланных сообщений, причём точное, а не просто асимптотика.

Отказ узлов или связи в распределённых системах — обычное дело, это *основная проблема разработки распределённых алгоритмов* (в отличие от просто параллельных).

Из-за сети сообщения могут идти непонятно сколько времени (в том числе на практике). В *синхронных* системах гарантируется, что каждое сообщение либо доходит за некоторое константное время  $C$ , либо теряется. В *асинхронных* системах сообщения могут идти сколь угодно долго. На практике же везде ставят таймаут, превращая систему в синхронную.

Дополнительные сложности возникают с *порядком сообщений*: как между процессами ..., так и глобально во всей системе.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Метод дихотомии или метод половинного деления. URL: <http://bpascal.ru/download/desc/319.php>
2. Алгоритмическая неразрешимость: основные понятия и термины. URL: <https://www.finam.ru/publications/item/algoritmicheskaya-nerazreshimost-20230629-0719/>
3. Энциклопедия эпистемологии и философии науки. Алгоритмическая неразрешимость. URL: [https://epistemology\\_of\\_science.academic.ru/25/%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F\\_%D0%BD%D0%B5%D1%80%D0%B0%D0%B7%D1%80%D0%B5%D1%88%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D1%8C](https://epistemology_of_science.academic.ru/25/%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BD%D0%B5%D1%80%D0%B0%D0%B7%D1%80%D0%B5%D1%88%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D1%8C)
4. Макконнелл Дж. Основы современных алгоритмов. 2-е дополненное издание Москва: Техносфера, 2004. 368 с. URL: [lib.yusu.am/disciplines\\_bk/64e526e10ece3cc6ce6472cddc7712cd.pdf](lib.yusu.am/disciplines_bk/64e526e10ece3cc6ce6472cddc7712cd.pdf)
5. Валерий Баканов. Параллелизм в алгоритмах — выявление и рациональное его использование. Возможности компьютерного моделирования. URL: <https://habr.com/ru/articles/688196/>
6. Классификация Флинна. Большая российская энциклопедия. URL: <https://bigenc.ru/c/klassifikatsiia-flinna-d0c755>
7. ИТМО. Параллельное программирование. Базовые определения и формализм. URL: [https://neerc.ifmo.ru/wiki/index.php?title=%D0%91%D0%B0%D0%B7%D0%BE%D0%B2%D1%8B%D0%B5\\_%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F\\_%D0%B8\\_%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D0%B8%D0%B7%D0%BC](https://neerc.ifmo.ru/wiki/index.php?title=%D0%91%D0%B0%D0%B7%D0%BE%D0%B2%D1%8B%D0%B5_%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B8_%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D0%B8%D0%B7%D0%BC)
8. 5.1. Классификация архитектур вычислительных систем с параллельной обработкой данных. URL:

<https://siblec.ru/telekommunikatsii/vychislitelnye-sistemy-seti-i-telekommunikatsii/5-arkhitektury-vysokoproizvoditelnykh-vychislitelnykh-sistem/5-1-klassifikatsiya-arkhitektur-vychislitelnykh-sistem-s-parallelnoj-obrabotkoj-dannykh>

9. ИТМО. Формализм распределённых систем. URL:

[https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D0%B8%D0%B7%D0%BC\\_%D1%80%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D1%85\\_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BCs](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D0%B8%D0%B7%D0%BC_%D1%80%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D1%85_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BCs)

10. 6.1 История развития ООП. Базовые понятия ООП: объект, его свойства. URL: <https://stepik.org/lesson/327921/step/1>

11. ООП, или объектно-ориентированное программирование: что это такое и как работает. URL: <https://orbitsoft.com/ru/blog/objektno-orijentirovannoje-programmirovanije/>

12. Курс по ИИ для начинающих // Урок 1.4. Методы искусственного интеллекта. <https://aisimple.ru/6-methody-ai.html>

13. Как российские компании используют нейросети и какие преимущества они получают. URL: <https://42clouds.com/ru-ru/blog/business/kak-rossijskie-kompanii-ispolzuyut-nejroseti-i-kakie-preimushhestva-oni-poluchayut/>

14. Обзор российских AI-сервисов и приложений: знай наших! URL: <https://3dnews.ru/1085541/obzor-rossiyskih-ai-servisov-znay-nashih>

15. 10 примеров использования нейросетей, которые поражают воображение. URL: [https://dzen.ru/a/Y\\_MicqNEdzu6R48y](https://dzen.ru/a/Y_MicqNEdzu6R48y)

16. 29. Понятие сложности алгоритма и сложности (объема) входных данных. Основные правила вычисления сложности алгоритма (сложность линейного алгоритма, ветвления, цикла). Пермский государственный национальный исследовательский университет. URL: <https://studfile.net/preview/2674691/page:21/>