# Алгоритм
# Понятие сложности алгоритма

## Предмет изучения теории алгоритмов

Попытки дать формальное определение алгоритма предпринимались многими выдающимися учёными: Тюрингом, Постом, Марковым, Гёделем, Чёрчем и др., но в рамках данного курса нам достаточно «бытового» определения.

Под алгоритмом мы будем понимать конечный набор правил, который определяет последовательность операций для решения конкретного множества задач. Любой алгоритм обладает следующими важными свойствами:

- Дискретность — процесс решения задачи должен быть представлен в виде последовательности элементарных шагов;
- Определённость – в каждый момент времени следующий шаг алгоритма полностью определяется текущим состоянием системы;
- Понятность — шаги алгоритма должны быть понятны и доступны исполнителю (ЭВМ, человеку и др.);
- Универсальность – алгоритм может быть применён к различным наборам исходных данных;
- Результативность и конечность — алгоритм должен завершаться определённым результатом (или сообщением о том, что результат не может быть получен) за конечное число шагов.

Теория алгоритмов изучает общие свойства алгоритмов. Следует различать качественную и количественную теорию алгоритмов. Основной задачей качественной теории алгоритмов является построение алгоритма, решающего поставленную задачу. Количественная теория алгоритмов исследует существующие алгоритмы с точки зрения времени их выполнения и ресурсоёмкости.

## Количественная теория алгоритмов

### *Понятие объёмно-временной сложности*

Существует несколько способов измерения сложности алгоритма. Обычно основное внимание уделяют скорости алгоритма, но не менее важны и другие показатели: требуемый объём ОЗУ или дискового пространства. Использование быстрого алгоритма не имеет смысла, если для его работы понадобится больше памяти, чем есть у компьютера.

Алгоритмы решения многих задач предполагают выбор между объёмом требуемой памяти и скоростью работы. Одну и ту же задачу можно решить быстро, используя большой объём памяти, или медленнее, занимая меньший объём.

Ярким примером различных подходов к решению задачи служит алгоритм поиска кратчайшего пути между двумя точками. Представив кару города в виде графа, можно написать алгоритм для

определения кратчайшего расстояния между двумя любыми точками этого графа. Чтобы не вычислять эти расстояния всякий раз, когда они необходимы, можно вычислить кратчайшие расстояния между всеми возможными точками заранее и сохранить результаты в таблице. Когда потребуется узнать кратчайшее расстояние между двумя заданными точками, можно просто взять готовое расстояние из таблицы. Результат будет получен мгновенно, но это потребует значительных расходов памяти. Карта большого города может содержать десятки тысяч точек. Тогда описанная выше таблица, должна содержать более ста миллионов ячеек. Т.е. для повышения быстродействие алгоритма, необходимо использовать дополнительные 100 Мб памяти.

Из примера очевидно, что алгоритмы необходимо оценивать, исходя не только из времени работы, но и из требований к ресурсам. Такой подход к изучению алгоритмов называется оценкой объёмно-временной сложности.

В данном курсе основное внимание будет уделено временной сложности алгоритмов, но в случаях, когда это важно, обязательно будет обговариваться и объём потребляемой памяти.

## *Классы сложности*

При сравнении различных алгоритмов между собой важно знать, как их сложность зависит от объёма входных данных. Допустим, при сортировке одним методом обработка тысячи чисел занимает одну секунду, а обработка миллиона чисел – десять секунд, при использовании другого алгоритма может потребоваться две и пять секунд соответственно. В таких условиях нельзя однозначно сказать, какой алгоритм лучше.

В общем случае сложность алгоритма принято оценивать по порядку величины. Общепринятой является методика определения класса скорости роста. Наиболее используемыми являются следующие три класса: $\Omega, \Theta, \mathrm{O}$, ниже даны их формальные определения.

Класс функций, растущих по крайней мере также быстро как функция $f$, принято называть классом $\Omega(f)$. Функция $g \in \Omega(f)$, если $\forall n > n_0\ g(n) > c * f(n), c > 0$.

Класс функций, растущих не быстрее функции $f$, принято называть классом $\mathrm{O}(f)$. Функция $g \in \mathrm{O}(f)$, если $\forall n > n_0\ g(n) \leq c * f(n), c > 0$.

И, наконец, функции, растущие с той же скоростью, что и функция $f$ обозначают как $\Theta(f)$. Формально класс $\Theta(f)$ проще всего определить как пересечение классов $\mathrm{O}(f)$ и $\Omega(f)$, т.е. $\Theta(f) = \mathrm{O}(f) \cap \Omega(f)$.

Алгоритм имеет сложность $\Theta(f(N))$, если при увеличении размерности входных данных $N$ время выполнения алгоритма возрастает с той же скоростью, что и функция $f(N)$. Рассмотрим код, который для матрицы $A[NxN]$ находит максимальный элемент в каждой строке.

**Листинг 1**

```
for i:=1 to N do
begin
  max:=A[i,1];
  for j:=1 to N do
  begin
    if A[i,j]>max then
      max:=A[i,j]
  end;
```

```
  writeln(max);
end;
```

В этом алгоритме переменная $i$ меняется от $1$ до $N$. При каждом изменении $i$ переменная $j$ тоже меняется от $1$ до $N$. Во время каждой из $N$ итераций внешнего цикла внутренний цикл тоже выполняется $N$ раз. Общее количество итераций внутреннего цикла равно $N * N$. Это определяет сложность алгоритма $\Theta(N^2)$.

Наибольший интерес представляет всё-таки класс $\mathrm{O}$, т.к. именно с его помощью можно легко сравнивать эффективность алгоритмов. Например, если сложность одного алгоритма принадлежит классу $\mathrm{O}$ от сложности второго, значит второй алгоритм решает исследуемую задачу не лучше первого. Легко показать, что сложность рассмотренного в Листинг 1 алгоритма можно оценить, как $\mathrm{O}(N^2)$.

Оценивая порядок сложности алгоритма, необходимо использовать только ту часть, которая возрастает быстрее всего. Предположим, что рабочий цикл описывается выражением $N^3 + N$. В таком случае его сложность будет равна $\mathrm{O}(N^3)$. Рассмотрение быстро растущей части функции позволяет оценить поведение алгоритма при увеличении $N$. Например, при $N = 100$, то разница между $N^3 + N = 1000100$ и $N = 1000000$ равна всего лишь $100$, что составляет $0{,}01\%$. При вычислении $\mathrm{O}$ можно также не учитывать постоянные множители в выражениях. Алгоритм с рабочим циклом $3N^3$ рассматривается, как $\mathrm{O}(N^3)$.

## *Определение сложности*

Наиболее сложными частями программы обычно является выполнение циклов и вызов процедур. В предыдущем примере весь алгоритм выполнен с помощью двух циклов.

Если одна процедура вызывает другую, то необходимо более тщательно оценить сложность последней. Если в ней выполняется определённое число инструкций (например, вывод на печать), то на оценку сложности это практически не влияет. Если же в вызываемой процедуре выполняется $\mathrm{O}(N)$ шагов, то это может значительно усложнить алгоритм. Если же процедура вызывается внутри цикла, то влияние может быть ещё больше.

В качестве примера рассмотрим две процедуры: `Slow` со сложностью $\mathrm{O}(N^3)$ и `Fast` со сложностью $\mathrm{O}(N^2)$.

Если основная программа вызывает процедуры по очереди, то их сложности складываются: $\mathrm{O}(N^2) + \mathrm{O}(N^3) = \mathrm{O}(N^3)$. Следующий фрагмент (функция `Both`) имеет именно такую сложность:

**Листинг 2**

```
procedure Slow;
var
i,j,k: integer;
begin
  for i:=1 to N do
    for j:=1 to N do
      for k:=1 to N do
        {какое-то действие}
end;
procedure Fast;
var
i,j: integer;
begin
```

```
  for i:=1 to N do
    for j:=1 to N do
      {какое-то действие}
end;
procedure Both;
begin
  Fast;
  Slow;
end;
```

Если же во внутренних циклах процедуры `Fast` происходит вызов процедуры `Slow`, то сложности процедур перемножаются.

**Листинг 3**

```
procedure Slow;
var
i,j,k: integer;
begin
  for i:=1 to N do
    for j:=1 to N do
      for k:=1 to N do
        {какое-то действие}
end;
procedure Fast;
var
i,j: integer;
begin
  for i:=1 to N do
    for j:=1 to N do
      Slow;
end;
procedure Both;
begin
  Fast;
end;
```

В данном случае сложность процедуры `Both` составляет $O(N^2) * O(N^3) = O(N^5)$.

### *Сложность рекурсивных алгоритмов*

Напомним, что рекурсивными процедурами называются процедуры, которые вызывают сами себя. Их сложность определить довольно тяжело. Сложность этих алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. Рекурсивная процедура может выглядеть достаточно простой, но она может серьёзно усложнить программу, многократно вызывая себя. Рассмотрим рекурсивную реализацию вычисления факториала:

**Листинг 4**

```
function Factorial(n: Word): integer;
begin
  if n > 1 then
    Factorial:=n*Factorial(n-1)
  else
    Factorial:=1;
end;
```

Эта процедура выполняется $N$ раз, таким образом, вычислительная сложность этого алгоритма равна $O(N)$.

Рекурсивный алгоритм, который вызывает себя несколько раз, называется многократной рекурсией. Такие процедуры гораздо сложнее анализировать, кроме того, они могут сделать алгоритм гораздо сложнее. Рассмотрим следующую процедуру:

**Листинг 5**

```
procedure DoubleRecursive(N: integer);
begin
  if N>0 then
  begin
    DoubleRecursive(N-1);
    DoubleRecursive(N-1);
  end;
end;
```

Поскольку процедура вызывается дважды, можно было бы предположить, что её рабочий цикл будет равен $O(2N) = O(N)$. Но на самом деле ситуация гораздо сложнее. Если внимательно исследовать этот алгоритм, то станет очевидно, что его сложность равна $O(2^N)$.

Для всех рекурсивных алгоритмов также очень важно понятие объёмной сложности. При каждом вызове процедура запрашивает небольшой объём памяти в стеке для сохранения значений локальных переменных, но в сумме этот объём может занимать значительную часть доступной памяти. По этой причине всегда необходимо проводить хотя бы поверхностный анализ объёмной сложности рекурсивных процедур.

### *Средний и наихудший случай*

Оценка сложности алгоритма до порядка является верхней границей сложности алгоритмов. Если программа имеет большой порядок сложности, это вовсе не означает, что алгоритм будет выполняться действительно долго. На некоторых наборах данных выполнение алгоритма занимает намного меньше времени, чем можно предположить на основе их сложности. Например, рассмотрим код, который ищет заданный элемент в одномерном массиве $A$.

**Листинг 6**

```
function Locate(data: integer): word;
var
  i: integer;
  fl: boolean;
begin
  fl := false;
  i := 1;
  while (not fl) and (i <= N) do
  begin
    if A[i] = data then
      fl := true
    else
      i := i + 1;
  end;
  if not fl then
    i := 0;
  Locate := i;
end;
```

Если искомый элемент находится в конце списка, то программе придётся выполнить все $N$ шагов. В таком случае сложность алгоритма составит $O(N)$. В этом наихудшем случае время работы алгоритма будем максимальным.

С другой стороны, искомый элемент может находится в списке на первой позиции. Алгоритму придётся сделать всего один шаг. Такой случай называется наилучшим и его сложность можно оценить, как $\mathrm{O}(1)$.

Оба эти случая маловероятны. Нас больше всего интересует ожидаемый вариант. Если элементы списка изначально беспорядочно перемешаны, то искомый элемент может оказаться в любом месте списка. В среднем потребуется сделать $N/2$ сравнений, чтобы найти требуемый элемент. Значит, сложность этого алгоритма в среднем составляет $\mathrm{O}(N/2) = \mathrm{O}(N)$.

В данном случае средняя и ожидаемая сложность совпадают, но для многих алгоритмов сложность в наихудшем случае сильно отличается от сложности в среднем случае. Например, алгоритм быстрой сортировки в наихудшем случае имеет сложность порядка $\mathrm{O}(N^2)$, в то время как ожидаемое поведение описывается оценкой $\mathrm{O}(N * log(N))$, что намного быстрее.

### *Общие функции оценки сложности*

Сейчас мы перечислим некоторые функции, которые чаще всего используются для оценки сложности. Функции перечислены в порядке возрастания сложности. Чем выше в этом списке находится функция, тем быстрее будет выполняться алгоритм с таким классом сложности.

**Таблица 1**

| Функция | Примечание |
|---|---|
| $C$ | |
| $\log(\log(N))$ | |
| $\log(N)$ | |
| $N^C$ | $0 < 1 < C$ |
| $N$ | |
| $N * log(N)$ | |
| $N^C$ | $C > 1$ |
| $C^N$ | $C > 1$ |
| $N!$ | |

Если необходимо оценить сложность алгоритма, выражение сложности которого содержит сумму нескольких функций, то выражение можно сократить до функции, расположенной ниже в таблице. Например, $\mathrm{O}(log(N) + N!) = \mathrm{O}(N!)$.

Если алгоритм вызывается редко и для небольших объёмов данных, то приемлемой можно считать сложность $\mathrm{O}(N^2)$. Алгоритмы со сложностью $O(N * \log(N))$ подходят для редкой работы с большим массивом данных. Если же требуется работа в реальном времени, то не всегда достаточно производительности $\mathrm{O}(N)$.

Алгоритмы со сложностью $N^C$ можно использовать только при небольших значениях $C$. Вычислительная сложность алгоритмов, порядок которых определяется функциями $C^N$ и $N!$ очень велика, поэтому такие алгоритмы могут использоваться только для обработки сильно ограниченного объёма данных.

Рассмотрим таблицу, которая показывает, как долго компьютер, осуществляющий миллион операций в секунду, будет выполнять некоторые медленные алгоритмы, и график, наглядно показывающий скорость роста соответствующих функций.

**Таблица 2**

| | N=10 | N=20 | N=30 | N=40 | N=50 |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| $N^3$ | 0.001 с | 0.008 с | 0.027 с | 0.064 с | 0.125 с |
| $2^N$ | 0.001 с | 1.05 с | 17.9 мин | 1. 29 дней | 35.7 лет |
| $3^N$ | 0.059 с | 58.1 мин | 6.53 лет | $3.86 * 10^5$ лет | $2.28 * 10^{10}$ лет |
| $N!$ | 3.63 с | $7.71 * 10^4$ лет | $8.41 * 10^{18}$ лет | $2.59 * 10^{34}$ лет | $9.64 * 10^{50}$ лет |

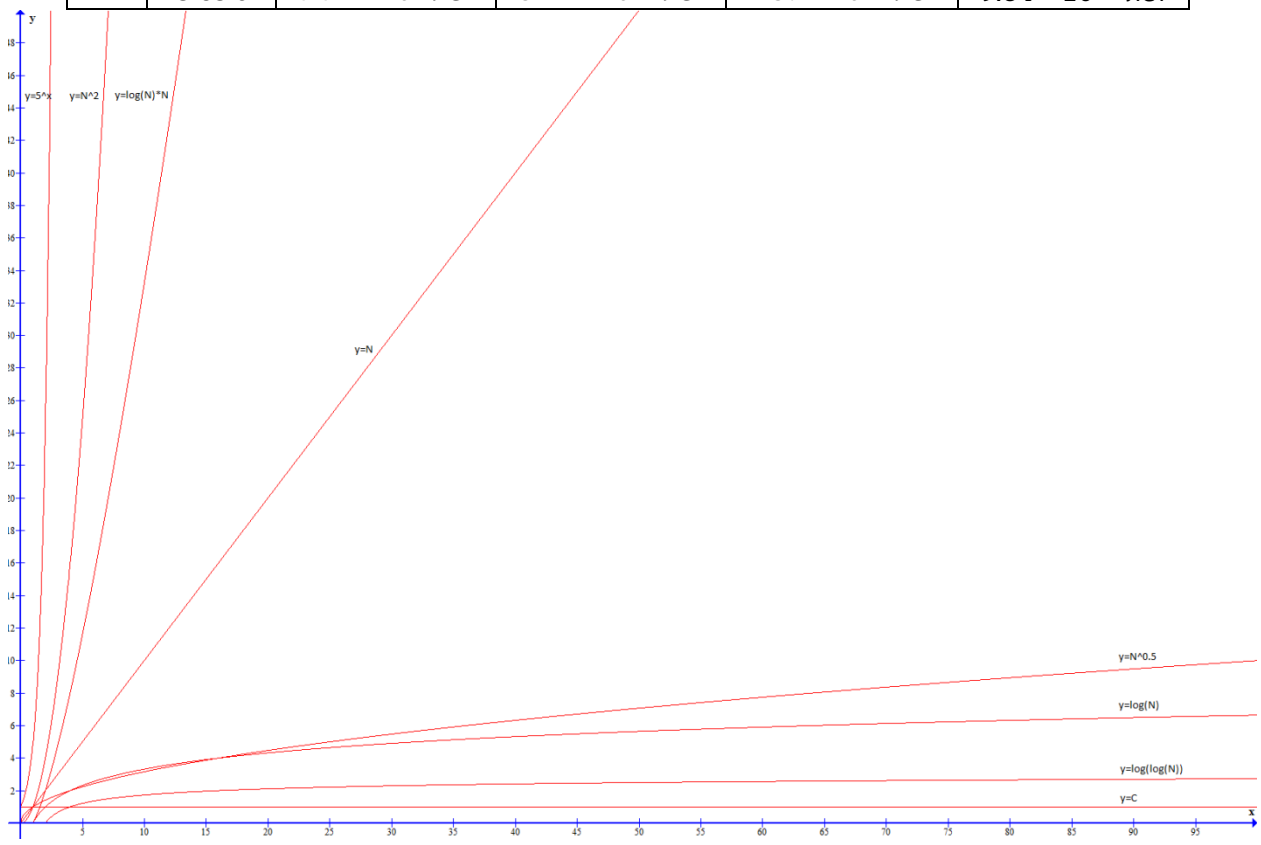


**Рисунок 1**