Анализ сложности алгоритмов

Многие современные программисты, пишущие классные и широко распространённые программы, имеют крайне смутное представление о теоретической информатике. Это не мешает им оставаться прекрасными творческими специалистами, и мы благодарны за то, что они создают. Тем не менее, знание теории тоже имеет свои преимущества и может оказаться весьма полезным. Эта лекция посвящена анализу сложности алгоритмов. Как человек, работавший как в области академической науки, так и над созданием коммерческого ПО, я считаю этот инструмент по-настоящему полезными на практике. Надеюсь, что после сегодняшней лекции вы сможете применить его к собственному коду, чтобы сделать его ещё лучше.

Мы уже знаем, что существуют инструменты, измеряющие, насколько быстро работает код. Это программы, называемые *профайлерами* (profilers), которые определяют время выполнения в миллисекундах, помогая нам выявлять узкие места и оптимизировать их. Но, хотя это и полезный инструмент, он не имеет отношения к сложности алгоритмов. Сложность алгоритма — это то, что основывается на сравнении двух алгоритмов на идеальном уровне, игнорируя низкоуровневые детали вроде реализации языка программирования, «железа», на котором запущена программа, или набора команд в данном процессоре. Подсчёт миллисекунд тут мало поможет. Вполне может оказаться, что плохой алгоритм, написанный на низкоуровневом языке (например, ассемблере), будет намного быстрее, чем хороший алгоритм, написанный языке программирования высокого уровня (например, Python или C++). Так что пришло время определиться, что же такое «лучший алгоритм» на самом деле.

Алгоритм — это программа, которая представляет из себя исключительно вычисление, без других часто выполняемых компьютером вещей — сетевых задач или пользовательского ввода-вывода. Анализ сложности позволяет нам узнать, насколько быстра эта программа, когда она совершает вычисления. Примерами чисто *вычислительных* операций могут послужить операции над числами с плавающей запятой (сложение и умножение), поиск заданного значения из находящейся в ОЗУ базы данных, определение игровым искусственным интеллектом (ИИ) движения своего персонажа таким образом, чтобы он передвигался только на короткое расстояние внутри игрового мира, или запуск шаблона регулярного выражения на соответствие строке. Очевидно, что вычисления встречаются в компьютерных программах повсеместно.

Анализ сложности также позволяет нам объяснить, как будет вести себя алгоритм при возрастании входного потока данных. Если наш алгоритм выполняется одну секунду при 1000 элементах на входе, то как он себя поведёт, если мы удвоим это значение? Будет работать также быстро, в полтора раза быстрее или в четыре раза медленнее? В практике программирования такие предсказания крайне важны. Например, если мы создали алгоритм для web-приложения, работающего с тысячей пользователей, и измерили его время выполнения, то используя анализ сложности, мы получим весьма неплохое представление о том, что случится, когда число пользователей возрастёт до двух тысяч. Для соревнований по построению алгоритмов анализ сложности также даст нам понимание того, как долго будет выполняться наш код на наибольшем из тестов для проверки его правильности. Так что, если мы определим общее поведение нашей программы на небольшом объёме входных данных, то сможем получить хорошее представление и о том, что будет с ней при больших потоках данных. Давайте начнём с простого примера: поиска максимального элемента в массиве.

Подсчёт инструкций

Максимальный элемент массива можно найти с помощью простейшего отрывка кода.

```
int M = A[0];
for(int i = 0; i < n; ++i)
{
   if(A[i] >= M)
   {
      M = A[i];
   }
}
```

В процессе анализа данного кода, имеет смысл разбить его на простые инструкции — задания, которые могут быть выполнены процессором тотчас же или близко к этому. Предположим, что наш процессор способен выполнять как единые инструкции следующие

операции:

- Присваивать значение переменной
- Находить значение конкретного элемента в массиве
- Сравнивать два значения
- Инкрементировать значение
- Основные арифметические операции (например, сложение и умножение)

Мы будем полагать, что ветвление (выбор между `if` и `else` частями кода после вычисления `if`-условия) происходит мгновенно, и не будем учитывать эту инструкцию при подсчёте. Для первой строки в коде выше: `int M = A[0];` требуются две инструкции: для поиска `A[0]` и для присвоения значения `M` (мы предполагаем, что `n` всегда как минимум 1). Эти две инструкции будут требоваться алгоритму, вне зависимости от величины `n`. Инициализация цикла `for` тоже будет происходить постоянно, что даёт нам ещё две команды: присвоение и сравнение.

```
i = 0; i < n;
```

Всё это происходит до первого запуска `for`. После каждой новой итерации мы будем иметь на две инструкции больше: инкремент `i` и сравнение для проверки, не пора ли нам останавливать цикл.

```
++i; i < n;
```

Таким образом, если мы проигнорируем содержимое тела цикла, то количество инструкций у этого алгоритма `4 + 2n` — четыре на начало цикла `for` и по две на каждую итерацию, которых мы имеем `n` штук. Теперь мы можем определить математическую функцию `f(n)` такую, что, зная `n`, мы будем знать и необходимое алгоритму количество инструкций. Для цикла `for` с пустым телом `f( n ) = 4 + 2n`.

Анализ наиболее неблагоприятного случая

В теле цикла мы имеем операции поиска в массиве и сравнения, которые происходят всегда:

```
if ( A[ i ] >= M ) { ...
```

Но тело `if` может запускаться, а может и нет, в зависимости от актуального значения из массива. Если произойдёт так, что `A[ i ] >= M`, то у нас запустятся две дополнительные команды: поиск в массиве и присваивание:

```
M = A[ i ]
```

Мы уже не можем определить `f(n)` так легко, потому что теперь количество инструкций зависит не только от `n`, но и от конкретных входных значений. Например, для `A = [1, 2, 3, 4]` программе потребуется больше команд, чем для A = [4, 3, 2, 1].

Когда мы анализируем алгоритмы, мы чаще всего рассматриваем наихудший сценарий. Каким он будет в нашем случае? Когда алгоритму потребуется больше всего

инструкций до завершения? Ответ: когда массив упорядочен по возрастанию, как, например, `A = [1, 2, 3, 4]`. Тогда `M` будет переприсваиваться каждый раз, что даст наибольшее количество команд. Теоретики имеют для этого причудливое название — *анализ наиболее неблагоприятного случая*, который является ничем иным, как просто рассмотрением максимально неудачного варианта. Таким образом, в наихудшем случае в теле цикла из нашего кода запускается четыре инструкции, и мы имеем `f(n) = 4 + 2n + 4n = 6n + 4`.

Асимптотическое поведение

С полученной выше функцией мы имеем весьма хорошее представление о том, насколько быстр наш алгоритм. Однако нам нет нужды постоянно заниматься таким утомительным занятием, как подсчёт команд в программе. Более того, количество инструкций у конкретного процессора, необходимое для реализации каждого положения из используемого языка программирования, зависит от компилятора этого языка и доступного процессору набора команд. Ранее же мы говорили, что собираемся игнорировать условия такого рода. Поэтому сейчас мы пропустим нашу функцию `f` через «фильтр» для очищения её от незначительных деталей, на которые теоретики предпочитают не обращать внимания. Наша функция `6n + 4` состоит из двух элементов: `6n` и `4`. При анализе сложности важность имеет только то, что происходит с функцией подсчёта инструкций при значительном возрастании `n`. Это совпадает с предыдущей идеей «наихудшего сценария»: нам интересно поведение алгоритма, находящегося в «плохих условиях», когда он вынужден выполнять что-то трудное. Заметьте, что именно это по-настоящему полезно при сравнении алгоритмов. Если один из них побивает другой при большом входном потоке данных, то велика вероятность, что он останется быстрее и на лёгких, маленьких потоках. Вот почему **мы отбрасываем те элементы функции, которые при росте `n` возрастают медленно, и оставляем только те, что растут сильно**. Очевидно, что 4 останется 4 вне зависимости от значения `n`, а `6n` наоборот будет расти. Поэтому первое, что мы сделаем, — это отбросим 4 и оставим только `f(n) = 6n`.

Имеет смысл думать о 4 просто как о «константе инициализации». Разным языкам программирования для настройки может потребоваться разное время. Например, Java сначала необходимо инициализировать свою виртуальную машину. А поскольку мы договорились игнорировать различия языков программирования, то попросту отбросим это значение.

Второй вещью, на которую можно не обращать внимания, является множитель перед `n`. Так что наша функция превращается в `f(n) = n`. Как вы можете заметить, это многое упрощает. Ещё раз, константный множитель имеет смысл отбрасывать, если мы думаем о различиях во времени компиляции разных языков программирования (ЯП). Поиск в

массиве для одного ЯП может компилироваться совершенно иначе, чем для другого. Например, в Си выполнение `A[i]` не включает проверку того, что `i` не выходит за пределы объявленного размера массива, в то время как для Паскаля она существует. Таким образом, данный паскалевский код:

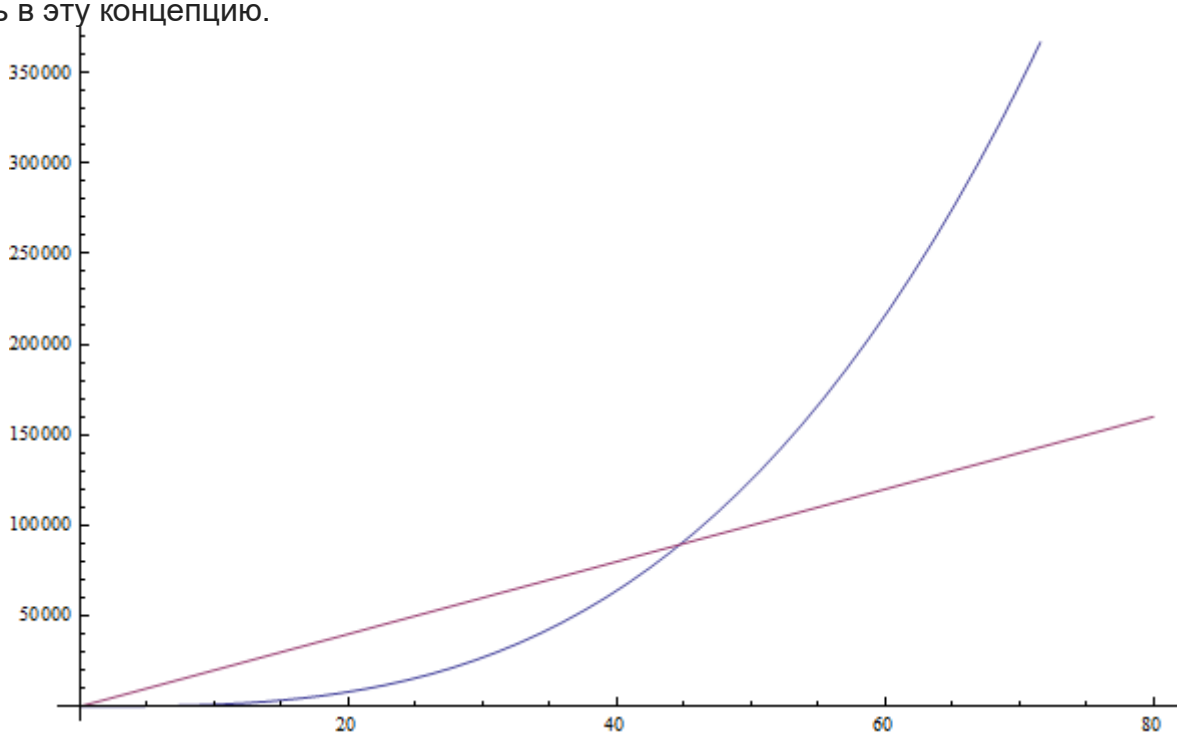
```
M := A[ i ]
```

эквивалентен следующему на Си:

```
if ( i >= 0 && i < n ) {    M = A[ i ];}
```

Так что имеет смысл ожидать, что различные языки программирования будут подвержены влиянию различных факторов, которые отразятся на подсчёте инструкций. В нашем примере, где мы используем «немой» паскалевский компилятор, игнорирующий возможности оптимизации, требуется по три инструкции на Паскале для каждого доступа к элементу массива вместо одной на Си. Пренебрежение этим фактором идёт в русле игнорирования различий между конкретными языками программирования с сосредоточением на анализе самой идеи алгоритма как таковой.
Описанные выше фильтры — «отбрось все факторы» и «оставляй только наибольший элемент» — в совокупности дают то, что мы называем *асимптотическим поведением*. Для `f(n) = 2n + 8` оно будет описываться функцией `f(n) = n`. Говоря языком математики, нас интересует предел функции `f` при `n`, стремящемся к бесконечности. Если вам не совсем понятно значение этой формальной фразы, то не переживайте — всё, что нужно, вы уже знаете. Строго говоря, в математической постановке мы не могли бы отбрасывать константы в пределе, но для целей теоретической информатики мы поступаем таким образом по причинам, описанным выше. Давайте проработаем пару задач, чтобы до конца вникнуть в эту концепцию.

Найдём асимптотики для следующих примеров, используя принципы отбрасывания константных факторов и оставления только максимально быстро растущего элемента:



1. `f(n) = 5n + 12` даст `f(n) = n`.
   Основания — те же, что были описаны выше
2. `f(n) = 109` даст `f( n ) = 1`.
   Мы отбрасываем множитель в `109*1` , но 1 по-прежнему нужен, чтобы показать, что функция не равна нулю.

3. `f(n) = n^2 + 3n + 112` даст `f(n) = n^2`
   Здесь `n^2` возрастает быстрее, чем `3n`, который, в свою очередь, растёт быстрее `112`
4. `f(n) = n^3 + 1999n + 1337` даст `f( n ) = n^3`
   Несмотря на большую величину множителя перед `n`, мы по прежнему полагаем, что можем найти ещё больший `n`, поэтому `f(n) = n^3` всё ещё больше `1999n` (см. рисунок выше)
5. `f(n) = n + √n` даст `f(n) = n,` потому что `n` при увеличении аргумента растёт быстрее, чем $\sqrt{n}$.

Упражнение 1

1. f(n) = n^6 + 3n
2. f(n) = 2^n + 12
3. f(n) = 3^n + 2^n
4. f(n) = n^n + n

Сложность

Из предыдущей части можно сделать вывод, что если мы сможем отбросить все эти декора тивные константы, то говорить об асимптотике функции подсчёта инструкций программы бу дет очень просто. Фактически, любая программа, не содержащая циклы, имеет `f(n) = 1`, потому что в этом случае требуется константное число инструкций (конечно, при отсутствии рекурсии — см. далее). Одиночный цикл от `1` до `n`, даёт асимптотику `f(n) = n`, поскольку до и после цикла выполняет неизменное число команд, а постоянное же количество инструкций внутри цикла выполняется `n` раз.
Руководствоваться подобными соображениями менее утомительно, чем каждый раз считать инструкции, так что давайте рассмотрим несколько примеров на закрепление этого материа ла. Следующая программа проверяет, содержится ли в массиве `A` размера `n` заданное значение:

```
ex = false;
for (i = 0; i < n; ++i )
        { if (A[i ] == v )
                { ex = true; break;
                }
        }
```

Такой метод поиска значения внутри массива называется *линейным поиском*. Это обоснованное название, пос кольку программа имеет `f(n) = n` (что означает «линейный» более точно, мы рассмотрим в следующем раз деле). Инструкция `break` позволяет программе завершиться раньше, даже после единственной итерации. Од нако, напоминаю, что нас интересует самый неблагоприятный сценарий, при котором массив `A` вообще не сод ержит заданное значение. Поэтому `f(n) = n` по-прежнему.
Давайте рассмотрим программу, которая складывает два значения из массива и записывает результат в новую переменную:

```
v = a[ 0 ] + a[ 1 ]
```

Здесь у нас постоянное количество инструкций, следовательно, `f(n) = 1`.
Следующая программа на C++ проверяет, содержит ли вектор (своеобразный массив) `A` размера `n` два одинаковых значения:

```
bool duplicate = false;
```

```
for ( int i = 0; i < n; ++i ) {
```

```
    for ( int j = 0; j < n; ++j ) {
```

```
        if ( (i != j) && (A[ i ] == A[ j ]) )
{duplicate = true;break;
            }
                                    }
    if ( duplicate ) {
        break;
    }
}
```

Два вложенных цикла дадут нам асимптотику вида $f(n) = n^2$.

Практическая рекомендация: простые программы можно анализировать с помощью подсчёта в них количества вложенных циклов. Одиночный цикл в n итераций даёт $f(n) = n$. Цикл внутри цикла — $f(n) = n^2$. Цикл внутри цикла внутри цикла — $f(n) = n^3$. И так далее.

Если в нашей программе в теле цикла вызывается функция, и мы знаем число выполняемых в ней инструкций, то легко определить общее количество команд для программы целиком. Рассмотрим в качестве примера следующий код на Си:

```
int i;
for ( i = 0; i < n; ++i ) {
    f( n );
}
```

Если нам известно, что $f(n)$ выполняет ровно n команд, то мы можем сказать, что количество инструкций во всей программе асимптотически приближается к n^2, поскольку $f(n)$ вызывается n раз.

Практическая рекомендация: если у нас имеется серия из последовательных for-циклов, то асимптотическое поведение программы определяет наиболее медленный из них. Два вложенных цикла, идущие за одиночным, асимптотически тоже самое, что и вложенные циклы сами по себе. Говорят, что вложенные циклы *доминируют* над одиночными.

Нотация асимптотического роста

Обозначение	Граница	Рост
(Тета) Θ	Нижняя и верхняя границы, точная оценка	Равно
(O - большое) O	Верхняя граница, точная оценка неизвестна	Меньше или равно
(o - малое) o	Верхняя граница, не точная оценка	Меньше
(Омега - большое) Ω	Нижняя граница, точная оценка неизвестна	Больше или равно
(Омега - малое) ω	Нижняя граница, не точная оценка	Больше

1. «O большое» — верхняя граница, в то время как «Омега большое» — нижняя граница. Тета требует как «O большого», так и «Омега большого», поэтому она является точной оценкой (она должна быть ограничена как сверху, так и снизу). К примеру, алгоритм, требующий Ω(n log n) требует не менее n log n времени, но верхняя граница не известна. Алгоритм, требующий Θ(n log n), предпочтительнее, потому, что он требует не менее n log n(Ω (n log n)) и не более чем n log n (O(n logn)).
2. f(x)=Θ(g(n)) означает, что f растет так же, как и g, когда n стремится к бесконечности. Другими словами, скорость роста f(x) асимптотически пропорциональна скорости роста g(n).
3. f(x)=O(g(n)). Здесь темпы роста не быстрее, чем g(n). «O большое» является наиболее полезной, поскольку представляет наихудший случай.

Если алгоритм имеет сложность, указанную в левом столбце, то его эффективность – в правом столбце:

Алгоритм	Эффективность
o(n)	< n
O(n)	≤ n
Θ(n)	= n
Ω(n)	≥ n
ω(n)	> n

«O» большое и **«o» малое** (*O* и *o*) — математические обозначения для сравнения асимптотического поведения (асимптотики) функций.

Под **асимптотикой** понимается характер изменения функции при её стремлении к определённой точке.

o(f), «*o* малое от f» обозначает «бесконечно малое относительно f», пренебрежимо малую величину при рассмотрении f. Смысл термина «O большое» зависит от его области применения, но всегда O(f) растёт не быстрее, чем f.

В частности:

- фраза «сложность алгоритма есть O(f(n))» означает, что с увеличением параметра n, характеризующего количество входной информации алгоритма, время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на f(n);
  - фраза «функция f(x) является „o“ малым от функции g(x) в окрестности точки p» означает, что с приближением x к p f(x) уменьшается быстрее, чем g(x) (отношение |f(x)|/|g(x)| стремится к нулю).

Обычно выражение «f является O большим (o малым) от g» записывается с помощью равенства $f(x) = O(g(x))$ (соответственно, $f(x) = o(g(x))$).

Это обозначение очень удобно, но требует некоторой осторожности при использовании (а потому в наиболее элементарных учебниках его могут избегать). Дело в том, что это не равенство в обычном смысле, а несимметричное отношение.

В частности, можно писать

$$f(x) = O(g(x)) \text{ (или } f(x) = o(g(x))),$$

но выражения

$$O(g(x)) = f(x) \text{ (или } o(g(x)) = f(x))$$

бессмысленны.

Другой пример: при $x \to 0$ верно, что

$$O(x^2) = o(x)$$

но неверно, что

$$o(x) = O(x^2).$$

При любом x верно

$$o(x) = O(x),$$

то есть бесконечно малая величина является ограниченной, но неверно, что ограниченная величина является бесконечно малой:

$$O(x) = o(x).$$

Вместо знака равенства методологически правильнее было бы употреблять знаки принадлежности и включения, понимая $O(\,)$ и $o(\,)$ как обозначения для множеств функций, то есть, используя запись в форме

$$x^3 + x^2 \in O(x^3)$$

или

$$O(x^2) \subset o(x)$$

вместо, соответственно,

$$x^3 + x^2 = O(x^3)$$

и

$$O(x^2) = o(x)$$

Однако на практике такая запись встречается крайне редко, в основном, в простейших случаях.

При использовании данных обозначений должно быть явно оговорено (или очевидно из контекста), о каких окрестностях (одно- или двусторонних; содержащих целые, вещественные, комплексные или другие числа) и о каких допустимых множествах функций идет речь (поскольку такие же обозначения употребляются и применительно к функциям многих переменных, к функциям комплексной переменной, к матрицам и др.).

Другие подобные обозначения

Для функций $f(n)$ и $g(n)$ при $n \to n_0$ используются следующие обозначения:

Обозначение	Интуитивное объяснение	Определение
$f(n) \in O(g(n))$	f ограничена сверху функцией g (с точностью до постоянного множителя) асимптотически	$\exists (C > 0), U : \forall (n \in U)\ \lvert f(n)\rvert \leq C\lvert g(n)\rvert$
$f(n) \in \Omega(g(n))$	f ограничена снизу функцией g (с точностью до постоянного множителя) асимптотически	$\exists (C > 0), U : \forall (n \in U)\ C\lvert g(n)\rvert \leq \lvert f(n)\rvert$

$f(n) \in \Theta(g(n))$	f ограничена снизу и сверху функцией g асимптотически	$\exists(C>0),(C'>0),U:\forall(n\in U)\ C\lvert g(n)\rvert \le \lvert f(n)\rvert \le C'\lvert g(n)\rvert$
$f(n) \in o(g(n))$	g доминирует над f асимптотически	$\forall(C>0),\exists U:\forall(n\in U)\ \lvert f(n)\rvert < C\lvert g(n)\rvert$
$f(n) \in \omega(g(n))$	f доминирует над g асимптотически	$\forall(C>0),\exists U:\forall(n\in U)\ C\lvert g(n)\rvert < \lvert f(n)\rvert$
$f(n) \sim g(n)$	f эквивалентна g асимптотически	$\lim\limits_{n\to n_0}\dfrac{f(n)}{g(n)}=1$

где U — проколотая окрестность точки n_0.

Обозначение «„O“ большое» введено немецким математиком Паулем Бахманом во втором томе его книги «Analytische Zahlentheorie» (Аналитическая теория чисел), вышедшем в 1894 году. Обозначение «„o“ малое» впервые использовано другим немецким математиком, Эдмундом Ландау в 1909 году; с работами последнего связана и популяризация обоих обозначений, в связи с чем их также называют **символами Ландау**. Обозначение пошло от немецкого слова «Ordnung» (порядок).

Как сравнивают быстродействие алгоритмов (асимптотические нотации: о, омега и тета).

Целые книги написаны по теории алгоритмов, но нам, простым смертным, которым не хочется влезать в дебри математики, но всё же любопытно узнать, как сравнивают алгоритмы по эффективности между собой, достаточно знать, что такое асимптотические нотации в общих чертах.

Допустим, у нас есть простейший алгоритм линейного поиска (то есть алгоритм, который перебирает последовательно все элементы массива друг за другом, в ходе этого запоминает индекс того элемента, значение которого равно искомому значению, а завершает работу, когда все элементы массива пройдены. То есть для искомого значения x, массива A с количеством элементов n:

1. Присваиваем значение «не найдено» переменной «Ответ».
2. Для каждого индекса i (от 1 до n):
3. Если A[i] = x, то присваиваем переменной «Ответ» значение i.
4. Возвращаем значение переменной «Ответ».

Попробуем подсчитать примерное время выполнения алгоритма в зависимости от n.

Допустим, t1 — время на выполнение шага 1 (выполняется 1 раз), t3 — время на выполнение шага 3 (один раз), t2 — время на проверку условия в шаге 2 (n+1 раз), t2′ — время на инкремент в шаге 2 (выполняется n раз), t3 — время на проверку условия A[i] = x (выполняется n раз) и t3’ — время на присваивание переменной «Ответ» = i (выполняется от 0 до n раз в зависимости от содержимого массива, в котором ищем значение).

Итак, в худшем случае имеем: t1 + t2*(n+1) + t2′*n + t3*n + t3’ * n + t3,

а в лучшем: t1 + t2 * (n+1) + t2′ * n + t2A *n + t2A’ * 0 + t3

Сгруппируем относительно переменного фактора n (все остальные величины — константы):

(t2 + t2′ + t3) * n + (t1 + t2 + t3) в лучшем случае и

(t2 + t2′ + t3 + t3’) * n + (t1 + t2 + t3) в худшем

В обоих случаях у нас многочлен типа a*n + b, где a и b — константы.

Мы видим, что при увеличении n все меньше значения будет играть b.

Не вдаваясь в определение асимптотической нотации, просто покажем, что это такое. У функции, зависящей от количества шагов алгоритма, приведенной к виду многочлена, убираем все младшие члены и коэффициент при старшем члене. То есть a*n + b превращается просто в n. Это называется тета (θ) функцией или нотацией. В нашем случае и в лучшем, и в худшем случае тета будет θ(n). Вообще, асимптотическая нотация для «худшего» случая называется O-нотацией, а для «лучшего» случая — омега (Ω)-нотацией. В нашем случае они совпадают. То есть при O-нотации O(n) мы показываем, что время выполнения алгоритма не может быть больше, чем n, помноженный на коэффициент. Младшими членами, как уже показано, при увеличении количества шагов алгоритма мы можем пренебречь. Например, если у нас функция, описывающая быстродействие алгоритма, вроде такой: $t1 * n^2 + t2*n + t3$, то тета-нотация будет такой: $\theta(n^2)$.

В целом, алгоритм с O(1) лучше, чем O(n), который лучше, чем алгоритм с O(n*log n) и т.д.

При сравнении конкретных алгоритмов надо помнить, что при малых значениях n нет смысла использовать эти нотации для сравнения, потому что в этом случае младшие члены многочленов могут быть даже больше, чем старшие. Однако, обычно алгоритмы все же используются для обработки больших объемов данных, и именно тогда имеет значение их эффективность.

Пример

n (размер задачи)	O(n)	$O(2^n)$
50	1 сек	1 сек
51	1,02 сек	2 сек
60	1,2 сек	17 мин
70	1,4 сек	12 суток
80	1,6 сек	34 года
90	1,8 сек	~35 тыс. лет