# Программы = алгоритмы + структуры данных

При построении программ важно:

1. Running time алгоритма (математические модели) для разного размера наборов данных
2. Memory usage для алгоритмов и структур данных (языки программирования, операционная система)

# Зачем изучать алгоритмы?

* Их влияние широко и far-reaching
* Старые корни, новые возможности
* Чтобы решать проблему, которая не могла быть иначе адресована
* Для интеллектуальной стимуляции (развития) :)
* Чтобы стать опытным (proficient) программистом.
* Они могут открыть секреты жизни и вселенной
* Ради забавы и прибыли :)

# Причины анализировать алгоритмы

1. Предсказывать эффективность
2. Сравнивать алгоритмы
3. Предоставлять гарантии
4. Понимать теоретический базис (теория алгоритмов)

Самая важная причина: избежать багов эффективности!!!

# Методы анализа алгоритмов

* Научный метод (эмпирический анализ нужен для валидации)
* Построение математической модели (мат анализ)
* Эмпирический анализ - запустили программу на разных по размеру инпутах и засекли время.

## Научный метод

Научный метод:

1. Наблюдение некоторых признаков естественного мира, как правило с точными измерениями.
2. Формулирование гипотезы - модели, которая согласуется с наблюдениями.
3. Прогнозирование - предсказывать события используя гипотезу.
4. Верификация - проверить предсказания сделав дальнейшие наблюдения.
5. Валидация - повторяем пока гипотеза и наблюдения не согласуются.

На алгоритмах:

1. Эмпирический анализ (наблюдение) Анализ данных (формулируется гипотеза) - используется регрессионный анализ и строится прямая вида T(N) = a N^b (power law), где b - slope. Гипотеза например: The running time is about 1.006 × 10 –10 × N^2.999 seconds. Причем порядок роста функции будет N^3.
2. Прогнозирование и валидация: подставляем в формулу N и получаем время.
3. Гипотеза удвоения: проще просто удвоить input в размере и получить b.

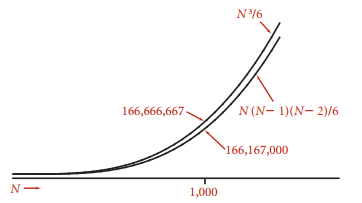
## Математический анализ

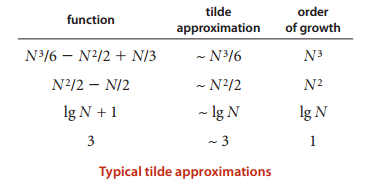
Total running time программы определяется двумя факторами:

* Стоимостью выполнения каждого оператора (statement)
* Частотой выполнения каждого оператора.

Тильда аппроксимация (асимптотический порядок роста - НЕТ!) или аппроксимация по лидирующему терму - откидываются термы низкого порядка, которые усложняют формулу и отображают пренебрежимо малый вклад в значения, представляющие интерес.

**Definition A**. Мы пишем ~ f(N) чтобы представить любую функцию, которая при делении на f(N) приближается к 1 по мере роста N.  
И мы записываем g(N) ~ f(N) чтобы показать, что g(N) / f(N) приближается к 1 по мере роста N.





Пример: .

Большую часть времени мы работаем с тильда аппроксимацией в форме  
константами, а f(N) порядок роста g(N) (order of growth).

Running time программ зависит только от маленького множества их инструкций.

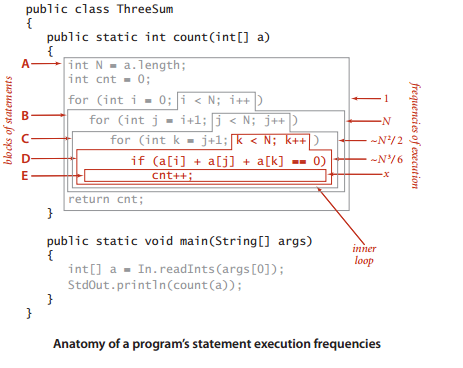
## Гипотеза порядка роста

**Property A**. Порядок роста времени выполнения ThreeSum (чтобы вычислить число троек, которые суммируют числа от 0 до N) есть N^3.

**Evidence**: Пусть T(N ) будет временем выполнения ThreeSum для N чисел. Только что описанная математическая модель предполагает, что T(N ) ~ aN^3 для некоторой машинно-зависимой константы **а**; эксперименты на многих компьютерах валидируют эту аппроксимацию.

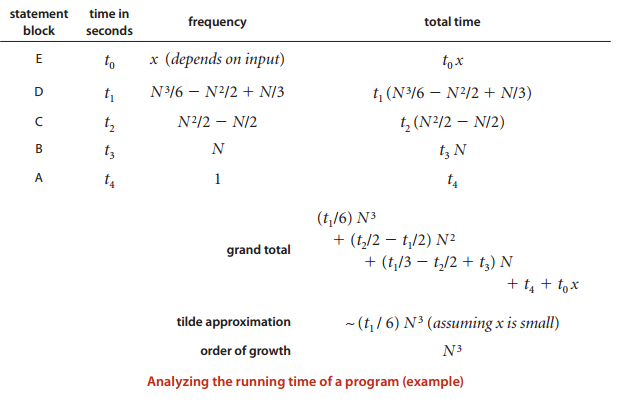
Property - это терм, ссылающийся к гипотезе, которую нужно проверить (валидировать) через эксперимент.

Рассмотрим на примере трех сум. Найдем частоты выполнения блоков операторов:



То есть if оператор выполняется точно N (N - 1) (N - 2) / 6 раз (число путей подобрать 3 разных числа из input array). 6 потому что 3! = 1 \* 2 \* 3

Анализ времени выполнения:



Порядок роста позволяет нам разделить программную реализацию от алгоритма.

Алгоритм определяет порядок роста. Разделение алгоритма от реализации позволяет нам разрабатывать знания о производительности (performance) алгоритма и затем применять это знание к любому компьютеру.

## Модель стоимости

Модель стоимости определяет стандартные операции, используемые алгоритмом, который мы изучаем. Для проблемы трех сумм модель стоимости будет: количество доступов к массиву. С помощью этой модели стоимости мы можем делать точные математические выражения о свойствах алгоритма, не только о конкретной реализации, как тут:

**Proposition B**. The brute-force 3-sum алгоритм использует ~ N^3 /2 доступов к массиву чтобы вычислить число триплетов, которые суммируют числа от 0 до N.

**Proof**: Алгоритм получает доступ к каждому из трех чисел для каждого из ~ N^3 /6 триплетов.

Proposition ссылается на математическую правду об алгоритмах в терминах **cost model**. Она поддерживает Property, которое мы валидируем с помощью экспериментов в соответствии с научным методом.

Наша цель - находить модели стоимости, такие что порядок роста времени выполнения для данной реализации такой же как и порядок роста стоимости нижележащего алгоритма (другими словами, **cost model**, должна включать операции в **inner loop**).

# Нахождение мат модели running time

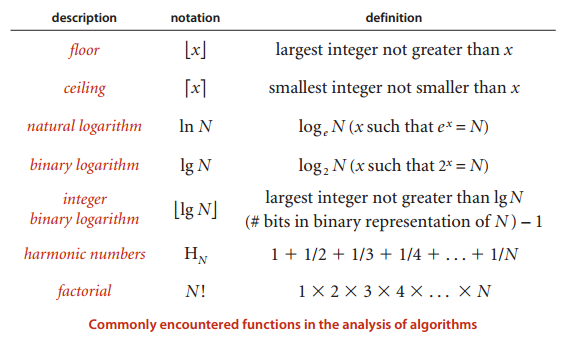
Для многих программ, разрабатывание мат модели running time разделяется на такие шаги:

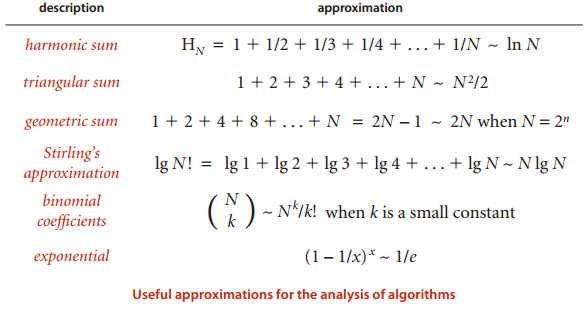
1. Разработать **input model**, включая определение проблемы размера
2. Найти **inner loop**
3. Определить **cost model**, которая включает операции в **inner loop**.
4. Определить частоту выполнения операций для данного **inputа**. Используя математический **анализ**, показать порядок роста.

Пример “Бинарный поиск”:

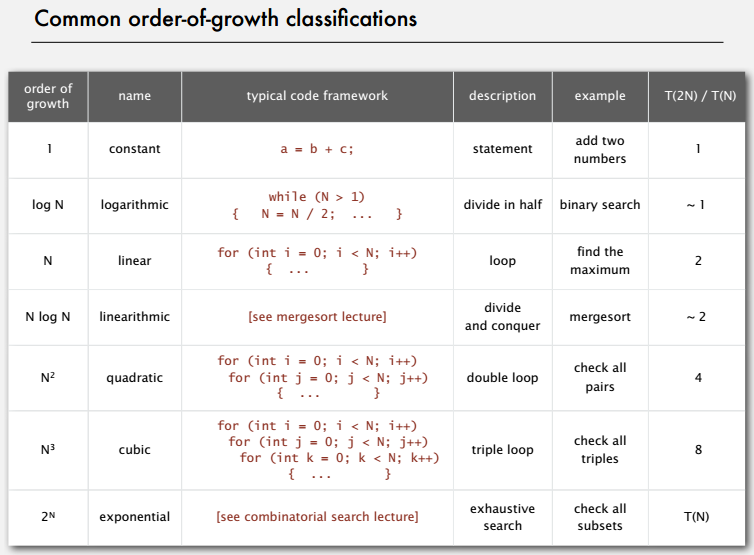
* **input model** это массив a[] размера N;
* **inner loop** это операторы в одиночном while цикле
* **cost model** - это операция сравнения (сравнение значений двух энтрисов массива)
* **analysis** показывает, что число сравнений не более чем lg N + 1

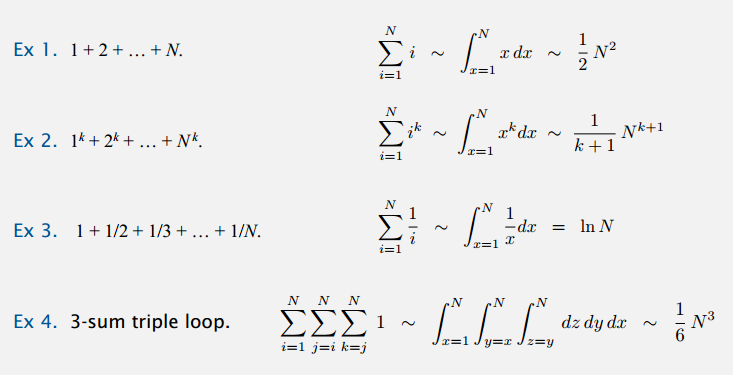
# Дополнительные формулы

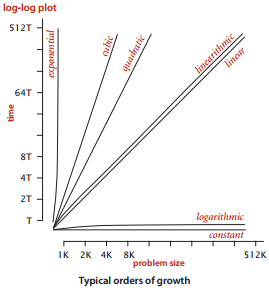
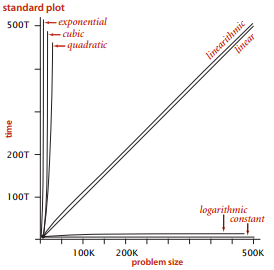




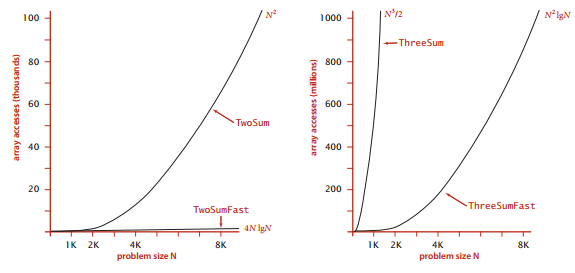
1 + 1 + 1 + … + 1 = lg N







Типичные порядки роста



Стоимость алгоритмов для решения проблемы “2 сумм” и “3 сумм”

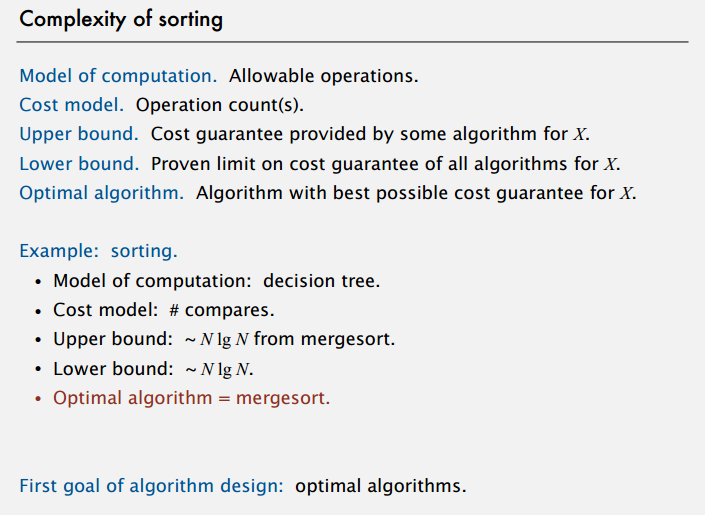
# 

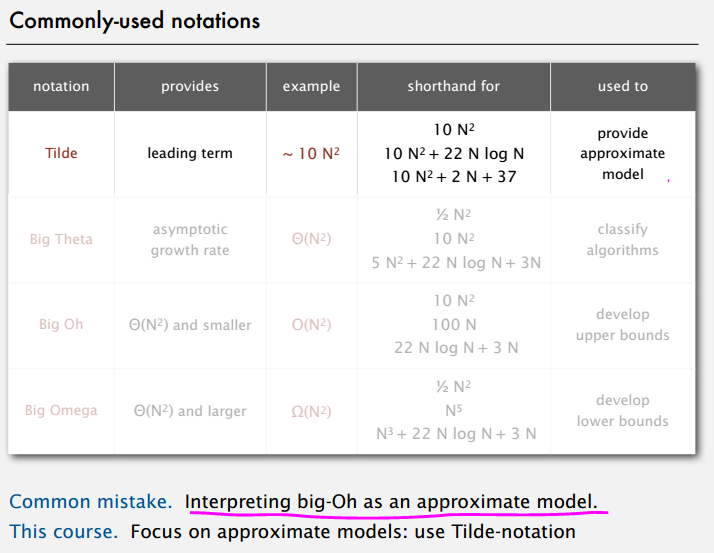
# 

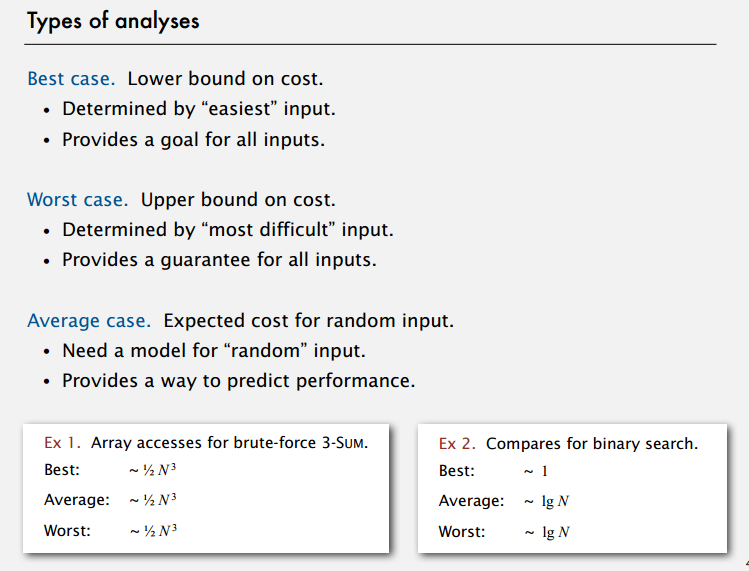
# Теория алгоритмов

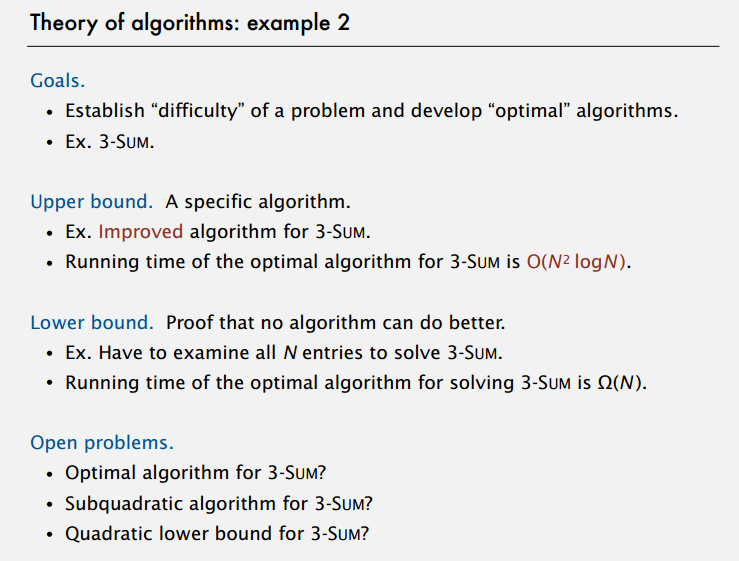
Коротко:

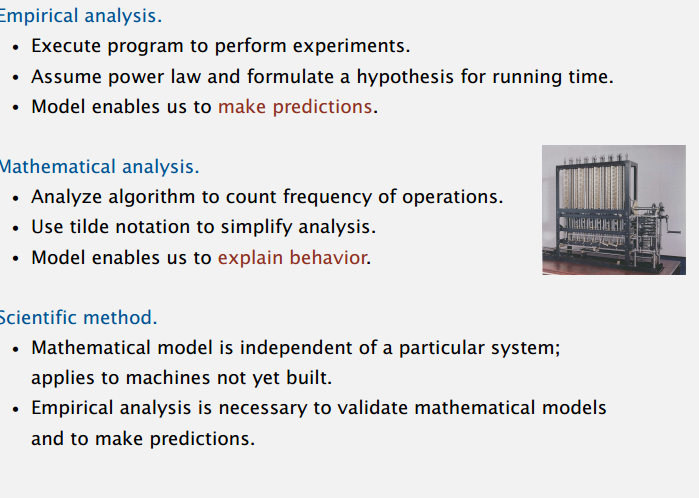
* Best case - **lower bound** по стоимости (чтобы разработать lower bound нужен
* Worst case - **upper bound** по стоимости (чтобы разработать upper bound нужен O(N))
* Average case - “ожидаемая” стоимость
* А асимптотический порядок роста нужен для классификации алгоритмов.
* Оптимальный алгоритм гарантирует, что **lower bound ~ upper bound**.
* Для анализа алгоритмов лучше использовать **“тильда нотацию” (~ !!!!!!)** для предоставления приблизительных моделей (approximate models). Зачем она? Ну она лучше описывает функцию. Предоставляет как **upper bound** так и **lower bound**











## Дополнительные вопросы и ответы

**Q:** Robert Sedgewick, at his **Algorithms - Part 1** course in Coursera, states that people usually misunderstand the **big-O** notation when using it to show the order of growth of algorithms. Instead, he advocates for the **tilde** notation.

I understand the **big-O** is an upper bound for certain problem at certain condition.

What is the difference between the **tilde** and **big-O** notations?

**A**:

The ∼ notation is similar to the more conventional Θ notation. There are two main differences between ∼ and O:

1. O only provides an *upper bounds*, while ∼ is simultaneously an upper bound and a lower bound. When we say that the running time of an algorithm is O(n^2), this doesn't preclude the possibility that it is O(n). In contrast, if the running time is ∼n^2 then it cannot be ∼n. Another notation with these properties is Θ.
2. O only holds up to a constant: f=O(g) if f(n)≤Cg(n) for some C>0 (and large enough n). In contrast, for ∼ the implied constant is always 1: if f∼g then f/g→1. This contrasts with Θ in which the implied constant is arbitrary, and indeed there could be different constants for the lower and upper bounds.

Exact constants are impractical in general, for many reasons: they are machine dependent, hard to compute, and could fluctuate depending on n. The first problem can be mitigating by measuring some proxy for the actual time complexity, such as the number of comparisons in a sorting algorithm.

Sampling the course, it seems they are using Θ, but call it ***order of growth***.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

In the following video is described some asymptotic analyzes issues: <https://class.coursera.org/algo-004/lecture/169>

But I can't understand what is "low-order term" and "constant factor" itself? (it is at the 4th minute of the video).

The merge sort is 6n\*log2(n)+6n. Why 6n is low-order term in the case mentioned in the video and 6 is constant factor? Do these terms have concrete definition?

17down voteaccepted

**Lower-order term:**

"Order" here refers to the [order of magnitude](http://en.wikipedia.org/wiki/Order_of_magnitude).

The concept is easy to understand and explain when dealing with very simple terms such as x or x2. x has order of magnitude 1, since it can be written as x1, and x2 has order 2 - the order of magnitude is equal to the power of the variable in the term. But things get a little more hazy (at least for me) when you complicate things by, for example, adding log. [1]

In somewhat informal terms, f(x) is a lower order than g(x) if f(x) < g(x) as x tends to infinity.

It's easy to see that f(n) = 6n is a lower order than g(n) = 6n\*log2(n) by just substituting some really large value for n (the correct approach is to mathematically prove it, but substituting a large value tends to work for simple terms).

The [terms](http://en.wikipedia.org/wiki/Term_%28mathematics%29#Elementary_mathematics) are essentially the things separated by plus / minus symbols.

So a lower-order term is just any term which is of a lower order than some other term.

Presumably this is the opposite of [the leading-order term](http://en.wikipedia.org/wiki/Leading-order_term), which is the term with the largest order of magnitude.

**Constant factor:**

"Factor" is a term in multiplication. For 6n, 6 and n are factors.

A constant factor is simply anything that doesn't depend on the input parameter(s) (which is n in this case).

Here, regardless of what we make n, 6 will always stay 6, so it's **constant factor**.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Q**: In a text I am reading (Algorithms 4th Edition by Robert Sedgewick and Kevin Wayne) there is the [following passage](https://books.google.co.uk/books?id=idUdqdDXqnAC&pg=PA262#v=onepage&q&f=false):

Increment sequences have been devised [for shellsort] that drive the asymptotic growth of the worst-case number of compares down to N^4/3, N^5/4, N^6/5, ... , but many of these results are primarily of academic interest because these functions are hard to distinguish from one another (**and from a constant factor of N** ) for practical values of N.

In this context what is the meaning of "constant factor of N"?

**A**: The sequence N^4/3, N^5/4, N^6/5, ... approaches N because the exponent gets closer and closer to 1.

This means that the terms in "the asymptotic growth of the worst-case number of compares" get closer and closer to each other as N is approached. In practice, N can be treated as the constant factor.

(The author adds the caveat "for practical values of N" as, for huge values, the terms in the sequence will be distinguishable for longer.)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* bonus: <https://cs.stackexchange.com/questions/57/how-does-one-know-which-notation-of-time-complexity-analysis-to-use?noredirect=1&lq=1>