







# 4.1 Анализ алгоритмов

В этом разделе вы научитесь соблюдать принцип при программировании: *обратите внимание на стоимость.* Чтобы изучить стоимость их запуска, мы сами изучаем наши программы с помощью *научного метода* . Мы также применяем математический анализ для получения кратких моделей стоимости.

## Научный метод.

Следующий пятиступенчатый метод обобщает научный метод: следующий пятиступенчатый подход.

- *Соблюдайте* некоторые особенности мира природы.
- *Гипотезировать* модель, которая согласуется с наблюдениями.

- *Прогнозировать* события, используя гипотезу.
- *Проверьте* прогнозы, сделав дальнейшие наблюдения.
- *Подтвердите* , повторяя, пока гипотеза и наблюдения не согласуются.

Наши эксперименты должны быть *воспроизводимыми,* а сформулированные нами гипотезы должны быть *фальсифицируемыми* .

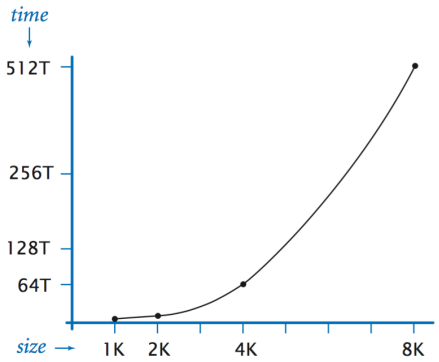
# Наблюдения.

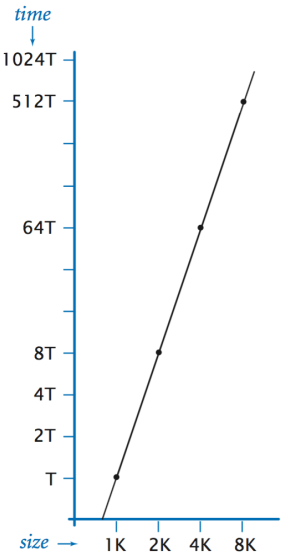
Измерить точное время выполнения нашей программы сложно, но есть ряд инструментов, которые могут помочь. В этой книге мы просто запускаем программу на различных входах и измеряем количество времени для обработки каждого ввода, используя тип данных Stopwatch.java . Наше первое качественное наблюдение о большинстве программ заключается в том, что существует *размер проблемы* , характеризующий сложность вычислительной задачи. Обычно размер проблемы - это либо размер ввода, либо значение аргумента командной строки. Интуитивно понятно, что время выполнения должно увеличиваться с увеличением размера проблемы, но вопрос о *том, насколько* она увеличивается, естественным образом возникает каждый раз, когда мы разрабатываем и запускаем программу.

# Конкретный пример.

Чтобы проиллюстрировать этот подход, мы начнем с ThreeSum.java, который подсчитывает количество троек в массиве из $N$целых чисел, сумма которых равна 0. Какова взаимосвязь между размером задачи $N$ и временем выполнения для `ThreeSum` ?

- *Удвоение гипотезы.* Для очень многих программ мы можем быстро сформулировать гипотезу для следующего вопроса: *Как влияет на время выполнения удвоение размера ввода?*
- *Эмпирический анализ.* Один простой способ разработать гипотезу удвоения состоит в том, чтобы удвоить размер ввода и наблюдать влияние на время выполнения. DoublingTest.java генерирует последовательность случайных входных массивов для `ThreeSum` , удваивая длину массива на каждом шаге, и печатает соотношение времени выполнения `ThreeSum.count ()` для каждого ввода по сравнению с предыдущим (которое было наполовину меньше размера). Если вы запустите эту программу, вы обнаружите, что истекшее время увеличивается примерно в 8 раз для печати каждой строки. Это сразу приводит к гипотезе, что время работы увеличивается в 8 раз, когда размер ввода удваивается.
- *Log-logplot.* Мы также можем построить график рабочего времени на стандартном графике (слева) или в журнале журнала (справа). Логарифмический график представляет собой прямую линию с наклоном 3, ясно предлагая гипотезу о том, что время работы удовлетворяет степенному закону вида с $п^3$.




- *Математический анализ.* Общее время работы определяется двумя основными факторами:
  - Стоимость выполнения каждого заявления.
  - Частота выполнения каждого оператора.

  Первое является свойством системы, а второе является свойством алгоритма. Если мы знаем оба для всех инструкций в программе, мы можем умножить их вместе и суммировать для всех инструкций в программе, чтобы получить время выполнения.

  Основная задача состоит в том, чтобы определить частоту выполнения операторов. Некоторые операторы легко анализируются: например, оператор, который устанавливает `счетчик` в `0` в `ThreeSum.count ()`, выполняется только один раз. Другие требуют рассуждений более высокого уровня: например, оператор `if` в `ThreeSum.count ()` выполняется точно $n ( n - 1 ) ( n - 2 ) / 6$ раз.

# Обозначение тильды.

Мы используем *нотацию тильды* для разработки более простых приближенных выражений. Во-первых, мы работаем с *основным термином* математических выражений, используя математическое устройство, известное как обозначение тильды. Мы пишем $\sim$ г( н ) для представления любой величины, которая при делении на e( н ) приближается к 1 ростом $N$. Мы также пишем грамм( $n$ ) $\sim$ f(н), чтобы указать, что грамм( н ) / f(н) приближается к 1 ростом $N$. С помощью этой записи мы можем игнорировать сложные части выражения, представляющие небольшие значения. Например, если утверждение `ThreeSum.count ()` выполняется $\sim$ н$^3$ / 6 раз потому что $n ( n - 1 ) ( n - 2 ) / 6 = n^3 / 6 -$ н$^2$ / 2 + н / 3, который, при делении на $N^3 / 6$, приближается к 1, как $N$ растет.

Мы ориентируемся на инструкции, которые выполняются наиболее часто, иногда называемые *внутренним циклом* программы. В этой программе разумно предположить, что время, уделяемое инструкциям вне внутреннего цикла, относительно незначительно.

# Порядок роста.

Ключевой момент в анализе времени выполнения программы заключается в следующем: для очень многих программ время выполнения удовлетворяет соотношению $T(n) \sim c\,f(\text{не})$ c - постоянная, а e ( н ) - функция, известная как *порядок роста* времени работает. Для типичных программ, e ( н ) является функцией , таких как журнал $n$ , $n \log_2$ н , н$^2$ или $N^3$ .

$N^3$ c $N^3$

$$\underset{n \to \infty}{\text{Ит}} \frac{T(2\text{ н})}{T(\text{ н})} \text{ знак равно } \frac{\text{c}\,(2\text{ н})^3}{\text{c}\,(\text{ н})^3} \text{ знак равно } 8$$

# Порядок роста классификаций.

Мы используем только несколько структурных примитивов (операторы, условные выражения, циклы и вызовы методов) для создания программ на Java, поэтому очень часто порядок роста наших программ является одной из немногих функций размера задачи, как показано в таблице ниже. ,

| *description* | *order of growth* | *example* | *framework* |
|---|---|---|---|
| *constant* | 1 | `count++;` | *statement (increment an integer)* |
| *logarithmic* | $\log n$ | `for (int i = n; i > 0; i /= 2)`<br>`  count++;` | *divide in half (bits in binary representation)* |
| *linear* | $n$ | `for (int i = 0; i < n; i++)`<br>`  if (a[i] == 0)`<br>`  count++;` | *single loop (check each element)* |
| *linearithmic* | $n \log n$ | [ *see mergesort* (PROGRAM 4.2.6) ] | *divide-and-conquer (mergesort)* |
| *quadratic* | $n^2$ | `for (int i = 0; i < n; i++)`<br>`  for (int j = i+1; j < n; j++)`<br>`  if (a[i] + a[j] == 0)`<br>`  count++;` | *double nested loop (check all pairs)* |
| *cubic* | $n^3$ | `for (int i = 0; i < n; i++)`<br>`  for (int j = i+1; j < n; j++)`<br>`  for (int k = j+1; k < n; k++)`<br>`  if (a[i] + a[j] + a[k] == 0)`<br>`  count++;` | *triple nested loop (check all triples)* |
| *exponential* | $2^n$ | [ *see Gray code* (PROGRAM 2.3.3) ] | *exhaustive search (check all subsets)* |

# Оценка использования памяти.

Чтобы обратить внимание на стоимость, вам необходимо знать об использовании памяти. Использование памяти четко определено для Java на вашем компьютере (каждое значение будет требовать одинакового объема памяти при каждом запуске вашей программы), но Java реализована на очень широком спектре вычислительных устройств, а потребление памяти - это реализация. зависимый.

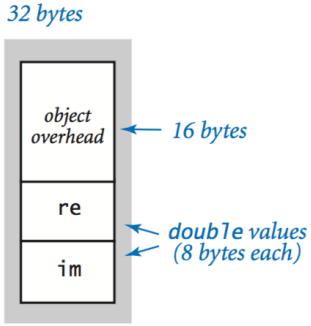
- *Примитивные типы.* Например, поскольку тип данных Java `int` представляет собой набор целочисленных значений от -2 147 483 648 до 2 147 483 647, то есть в общей сложности $2^{32}$ различных значения, разумно ожидать, что реализации будут использовать 32 бита для представления значений `int` .

| type | bytes |
|---|---|
| boolean | 1 |
| byte | 1 |
| char | 2 |
| int | 4 |
| float | 4 |
| long | 8 |
| double | 8 |

- *Объекты*. Чтобы определить потребление памяти объектом, мы добавляем объем памяти, используемый каждой переменной экземпляра, к издержкам, связанным с каждым объектом, обычно 8 байтов. Например, объект Complex.java использует 32 байта (16 байтов служебных данных, плюс 8 байтов для каждой из двух его переменных `двойного` экземпляра).

*Complex object (PROGRAM 3.2.6)*

```
public class Complex
{
   private double re;
   private double im;
...
}
```




Ссылка на объект обычно использует 8 байтов памяти. Когда тип данных содержит ссылку на объект, мы должны отдельно учитывать 8 байтов для ссылки и 16 байтов для каждого объекта *плюс* память, необходимую для переменных экземпляра объекта.

- *Массивы и строки.* Массивы в Java реализованы как объекты, как правило, с двумя переменными экземпляра (указатель на место в памяти первого элемента массива и длина). Для примитивных типов массив из $n$ элементов использует 24 байта информации заголовка, плюс $n$ раз количество байтов, необходимых для хранения элемента. Двумерный массив в Java - это массив массивов. Так , например, $n$ - матрицы с размерностью $n$ массив целых чисел использований ~ $4\,n^2$ байта памяти. Строка длиной $N$использует $56 + 2\,N$байтов памяти.

| type | bytes |
|---|---|
| boolean[] | $n + 24 \sim n$ |
| int[] | $4n + 24 \sim 4n$ |
| double[] | $8n + 24 \sim 8n$ |
| Charge[] | $40n + 24 \sim 40n$ |
| Vector | $8n + 48 \sim 8n$ |
| String | $2n + 56 \sim 2n$ |
| boolean[][] | $n^2 + 32n + 24 \sim n^2$ |
| int[][] | $4n^2 + 32n + 24 \sim 4n^2$ |
| double[][] | $8n^2 + 32n + 24 \sim 8n^2$ |

Смотрите учебник для более подробной информации.

**упражнения**

1. Реализуйте метод `printAll ()` для ThreeSum.java , который печатает все тройки, которые суммируются до нуля.
3. Напишите программу FourSum.java, которая принимает целое число, читает длинные целые числа из стандартного ввода и подсчитывает количество 4-кортежей, которые суммируются до нуля. Используйте четверной вложенный цикл. Каков порядок роста времени работы вашей программы? Оцените наибольший размер ввода, который ваша программа может обработать за час. Затем запустите вашу программу, чтобы проверить свою гипотезу.
7. $n$

```
int count = 0;
для (int i = 0; i <n; i ++)
    для (int j = i + 1; j <n; j ++)
        для (int k = j + 1; k <n; k ++)
            подсчитывать ++;
```

$$\binom{n}{3} = n(n - 1)(n - 2)/6$$

9. Определите порядок роста времени выполнения этого оператора в `ThreeSum` как функцию от числа целых чисел $n$ на стандартном вводе:

```
int [] a = StdIn.readAllInts ();
```

*Решение* : линейное. Узкими местами являются инициализация массива и цикл ввода. Однако, в зависимости от вашей системы, стоимость такого цикла ввода может доминировать в программе с линейным временем или даже в программе с квадратичным временем, если размер ввода не достаточно велик.

12. Что бы вы предпочли: квадратичный, линейный или линейный алгоритм?

$N^2 100$ н $\log_2 N 0000$ н $N 1000 N 2^{100}$ слишком большой, чтобы беспокоиться об этом).

16. $N$

```
Строка s = "";
for (int i = 0; i <n; i ++) {
    if (StdRandom.bernoulli (0.5)) s + = "0";
    иначе s + = "1";
}

StringBuilder sb = new StringBuilder ();
for (int i = 0; i <n; i ++) {
    if (StdRandom.bernoulli (0.5)) sb.append ("0");
    иначе sb.append ("1");
}
String s = sb.toString ();
```

*Решение* : первое является квадратичным; вторая линейная.

20. Дайте алгоритм линейного времени для обращения строки.

*Решение* :

```
public static String reverse (String s) {
    int n = s.length ();
    char [] a = новый символ [n];
    для (int i = 0; i <n; i ++)
       a [i] = s.charAt (ni-1);
    String reverse = новая строка (a);
    вернуться назад;
}
```

**Творческие Упражнения**

31. **Сумма подмножества.** Напишите программу SubsetSum.java, которая читает длинные целые числа из стандартного ввода и считает количество подмножеств тех целых чисел, которые в сумме равны нулю. Дайте порядок роста вашего алгоритма.

39. **Субэкспоненциальная функция.** Найдите функцию, порядок роста которой больше, чем любая полиномиальная функция, но меньше, чем любая экспоненциальная функция. *Дополнительный кредит* : найдите программу, время выполнения которой имеет такой порядок роста.

$N^{\text{пер } N}$