Темы урока

Алгоритмы

- **Алгоритм** — это последовательность шагов, которая решает определенную задачу. Иными словами алгоритм — это способ решения этой задачи.
- **Пример** алгоритм заказа книги в интернет-магазине:
  - Открыть сайт интернет-магазина
  - Найти книгу по названию
  - Если книга есть в наличии, добавить ее в корзину
  - Оформить заказ
  - Оплатить
  - Получить номер заказа и с нетерпением ждать доставки
- В программировании алгоритм, как правило, имеет **входные данные**, над которыми **производятся вычисления**, и **выходной результат**. По сути задача алгоритма состоит в **преобразовании** входных значений в выходные.

Эффективность алгоритма

- Важным критерием алгоритма выступает **эффективность**. Алгоритм может прекрасно решать поставленную задачу, но при этом быть не эффективным. Как правило, под эффективностью алгоритма подразумевается **время работы**, т.е. время преобразований данных.
- Но время работы, например в секундах, всегда относительно – оно может быть разным на разных компьютерах, разных ОС, оно может зависеть от количества оперативной памяти, частоты и разрядности процессора.
- В связи с этим, **эффективность алгоритма часто измеряют функцией, зависящей от количества элементарных операций процессора**. В таком виде алгоритмы можно сравнивать даже не запуская их на компьютере.

Асимптотический анализ алгоритмов

Асимптотическое поведение — это производительность алгоритма **при росте размера задачи**. Часто размер задачи обозначается как **N**. Чтобы описать асимптотическое поведение, нужно ответить на вопрос — **что случится** с производительностью алгоритма, **если N сильно вырастет**?

Для представления временной сложности алгоритмов в основном используют три асимптотических нотации:
- **O** (нотация о большое) - представляет наихудший порядок сложности,
- **Ω** (нотация омега большое) - представляет наилучший порядок сложности,
- **Θ** (нотация тета большое) - описывает порядок сложности, когда наихудший и наилучший случаи пересекаются.

Обычно интересна только оценка сверху, т.е. “не хуже, чем”, поэтому нотацию O большое используют чаще остальных. Рассмотрим её на примерах.

Алгоритм с постоянной сложностью

Начало. Подсчет количества операций

Давайте рассмотрим простой код, где есть единственное ветвления и посчитаем количество элементарных операций процессора, необходимых для выполнения этого кода:

```
int x = 0;
x = x + 1;
```

В приведенном коде на первый взгляд 2 команды, но количество операций будет бо́льшим:

1. `int x = 0` : инициализация переменной состоит из **2 операций**:
   a. создать локальную переменную
   b. записать в нее значение 0
2. `x = x + 1` : присвоение значения переменной состоит также их **2 операций**:

a. вычислить значение по формуле x + 1
b. записать его в переменную x

Магия IL DASM: Сколько же операций на самом деле?

Если ребята интересующиеся, можно показать им немного магии:

- скомпилировать такой код
- открыть сборку в IL DASM: "c:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.7.1 Tools\ildasm.exe"
- Показать промежуточный IL-код, о котором мы говорили на первом уроке
- Рассказать, где можно получить немного информации о значении тех или иных команд IL: https://en.wikipedia.org/wiki/List_of_CIL_instructions
- Показать IL-код нашей программы из двух строк и рассказать, что происходит в этих 2 строчках:

```
// int x = 0;
IL_0001:  ldc.i4.0          // кладем 0 как int на вершину стека
IL_0002:  stloc.0           // кладем значение вершины стека
                            // в локальную переменную под номером 0

// x = x + 1;
IL_0003:  ldloc.0           // достаем значение локальной переменной
                            // под номером 0 и кладем его на вершину стека
IL_0004:  ldc.i4.1          // кладем 1 как int на вершину стека
IL_0005:  add               // сейчас в стеке лежит 2 значения 0, который
                            // мы достали из переменной x и 1, взятая из
                            // выражения x + 1; над ними выполняется
                            // операция сложения; результат кладётся на
                            // вершину стека.
IL_0006:  stloc.0           // берем с вершины стека результат сложения
                            // и записываем его в локальную переменную 0
```

Продолжение. Расчет сложности

Тогда сложность такого алгоритма, измеренная в **количестве операций** – это **4** (или **6**, если разобрали предыдущую подчасть; можно также упомянуть, что в нашем случае это совсем неважно).

Теперь давайте представим сложность в виде нотаций “о большое”, “омега большое” и “тета большое”.

Все эти нотации дают оценку сложности в динамике для переменных факторов. В связи с этим делается ряд **упрощений**:

1. Все операции, не зависящие от переменных факторов сводятся к 1
2. В расчет идут только функции высшего порядка

Что это значит на нашем примере?

- **O** (нотация о большое) – описывает наихудший порядок сложности нашего кода. Когда мы говорим “худший”, мы имеем в виду “худший при разных значениях *переменного фактора* или *переменных величин*”. Поскольку у нас

нет переменных величин (4 – это константа), то это просто 4. Теперь применяем правило асимптотических нотаций: “Все операции, не зависящие от переменных факторов сводятся к 1”. Таким образом, наша 4 превращается в 1, и запись сложности в нотации “о большое” будет выглядеть вот так: **O(1)**. Тогда можно сказать, что этот код в выполнится “за о 1”.
- **Ω** (нотация омега большое) – описывает наилучший порядок сложности нашего кода. Поскольку мы только что выяснили, что переменных факторов у нас нет, наихудший порядок также сводится к единице. И тогда будет справедливо утверждение, что этот код в выполнится “за омега 1”, т.е., **Ω(1)**.
- **Θ** (нотация тета большое). В случае, когда нотации O и Ω одинаковы, удобно использовать нотацию “тета большое”, которая требует предварительного расчета O и Ω.
  Θ-нотация дает больше информации о сложности алгоритма, чем просто O- или Ω-нотация, так как в ней сразу заключено знание о том, что это и O и Ω одновременно.
  Для нашего примера, действительно, O- и Ω-нотации посчитаны и равны, следовательно запись Θ-нотации будет выглядеть так: **Θ(1)**.
  Ещё раз про Θ: можно обойтись и без Θ-нотации, можно просто написать O- и Ω-нотации – при первом взгляде будет и так видно, что они равны, однако, использование Θ-нотации даёт более лаконичную и понятную запись этой информации.

Для нашего простейшего случая всё получилось легко – нотации O и Ω дают один и тот же ответ – наш код (или наш алгоритм) имеет постоянную сложность, так как не имеет переменных факторов, что в самой краткой форме можно записать так: **Θ(1)**. Это означает, что **сложность нашего алгоритма – постоянная**!

⚠ **Обратите внимание!** X в данном случае не является переменным фактором алгоритма (хотя и является переменной с точки зрения программиста), так как при при любых значениях X в нашем коде из двух строк будет одно и то же количество операций.

⚠ **Важно понимать!** То, что мы записали, не означает, что наш алгоритм работает за 1 операцию.
Если бы в нашем алгоритме было 100 операций или 1 000 000 операций – он занимал бы ощутимое время!
Асимптотическая запись Θ(1) говорит лишь о том, что сложность нашего алгоритма – это постоянная величина. Т.е. количество операций не меняется в зависимости от исходных данных.
С практической точки зрения, это означает, что если наш алгоритм и время его работы устраивает всех в текущем виде, можно не бояться, что мы в будущем получим проблемы производительности. Наш код всегда будет выполняться примерно за это время.

Чаще всего пользуются O-нотацией – она практически более полезна, так как отвечает на вопрос “Как в худшем случае поведет себя алгоритм?”

Ω-нотация отвечает на вопрос “Насколько сложен алгоритм при самом благоприятном стечении обстоятельств?” и может быть полезна, например, в случае, когда необходимо доказать, что алгоритм имеет слишком высокий порядок сложности и мог бы быть оптимизирован.

Далее мы рассмотрим примеры алгоритма сортировки с различной степенью сложности.

Алгоритмы сортировки массива чисел

Сортировка “пузырьком”

Описание алгоритма

Сортировка пузырьком или сортировка простыми обменами – один из простейших алгоритмов сортировки. Он может применяться для упорядочивания массивов небольших размеров.

Идея данной сортировки заключается в попарном сравнении соседних элементов, начиная с нулевого в массиве. Больший элемент при этом в конце первой итерации оказывается на месте последнего элемента массива, и в следующих итерациях мы его уже не сравниваем его с остальными элементами.

- Если принять n за длину массива, в первой итерации у нас будет n-1 сравнение.
- Затем таким же образом мы находим второй по максимальности элемент и ставим его на предпоследнее место, и т. д.
- После всех итераций получится, что на месте нулевого элемента окажется элемент с наименьшим числовым значением, а на месте последнего – с наибольшим числовым значением. Таким образом, большие элементы у нас как бы “всплывают” словно пузырьки, вытесняя меньшие. Отсюда и название метода.

Реализация на C#

Код C# для такого алгоритма может выглядеть так:

⚠ *Не стоит забывать, что они пока не знают, что бывают методы класса, объяснить, что непосредственно работа по сортировке “пузырьком” заключен внутри фигурных скобок. Пусть воспринимают отдельные методы как способ изоляции кода и структурирования кода, который ещё и переиспользовать можно!*

Начинаем с первого прохода по массиву. При этом проходе самый элемент, имеющий наибольшее значение, будет вытеснен в самый конец массива:

```
private static int[] BubbleSort(int[] arr)
{
	// перебираем массив по j, не доходя до последнего элемента
	// до него мы доберемся через выражение j + 1
	int limit = arr.Length - 1;

	for (int j = 0; j < limit; j++)
	{
		// сравниваем текущий и последующий элементы
		// если текущий больше последующего, меняем их местами
		if (arr[j] > arr[j + 1])
		{
			int temp = arr[j + 1];     // обмен значений
			arr[j + 1] = arr[j];            // двух переменных
			arr[j] = temp;             // через третью

		}
	}
}
```

Показать в debug-режиме, что происходит с массивом в каждую итерацию цикла.

Затем усложняем этот код, накручивая сверху второй массив, чтобы все элементы заняли нужные места. При каждой итерации лимит будет уменьшаться на 1:

```
private static int[] BubbleSort(int[] arr)
{
	// i нам нужна уже не для доступа к массиву, а всего лишь
	// для уменьшения лимита внутреннего цикла
	for (int i = 0; i < arr.Length - 1; i++)
	{
		// перебираем массив по j, не доходя до последнего элемента
		// до него мы доберемся через выражение j + 1
		int limit = arr.Length - 1 - i;
		for (int j = 0; j < limit; j++)
		{
			// сравниваем текущий и последующий элементы
			// если текущий больше последующего, меняем их местами
			if (arr[j] > arr[j + 1])
			{
				int temp = arr[j + 1];  // обмен значений
				arr[j + 1] = arr[j];    // двух переменных
				arr[j] = temp;          // через третью
			}
		}
	}}
```

- При первой итерации по i значение i, равное 0, никак не влияет на наш расчет. При этом после завершения первой итерации по i (а внутренний цикл по j отработает полностью), в самом последнем элементе массива уже будет находиться наибольшее значение.
- В следующей итерации по i (когда i станет равным 1), последний элемент сравнивать было бы избыточно, он и так на своем месте. Поэтому мы будем “не доходить” до него вычитая из лимита нашу единицу, хранящуюся в i.

- В третьей итерации по i (i = 2), уже 2 последних элемента будут на своих местах и, вычитая из лимита значение переменной i, мы будем будем опускать проверку двух последних элементов.
- И так далее, пока не останется сравнить только нулевой и первый элементы массива. Чтобы закончить именно так, максимальное значение j во внутреннем цикле должно быть 0. Оно ограничивается переменной limit условием j < limit, минимальное значение limit для сохранения истинности этого выражения – это 1. Давайте посмотрим на формулу определения переменной limit:

  ```
  limit = arr.Length - 1 - i
  ```

  Чему должно равняться i, чтобы максимальное значение limit было 2?

  ```
  1 = arr.Length - 1 - i
  i = arr.Length - 1 - 1
  i = arr.Length - 2
  ```

  Итак, максимальное значение i – это arr.Length - 2. Но в цикле указывается условие выполнения цикла

  ```
  i < X
  ```

  Каково должно быть минимальное значение X, i было равно arr.Length - 2?

  ```
  arr.Length - 2 < X
  X > arr.Length - 2
  ```

  Минимальное значение X, при котором это условие будет всё ещё верным:

  ```
  X = arr.Length - 1
  ```

  Итого: в условие цикла по i пишем i < arr.Length - 1

Расчет сложности (дать посчитать самостоятельно)

```
// представим массив длиной N элементов
private static void BubbleSort(int[] arr)
{
  // две операции на int i = 0;                 // 2 +
  for (int i = 0; i < arr.Length - 1; i++)
  // все остальные операции будут умножаться    // (N - 1) × (
  // на количество итераций цикла по i

  // одна операция на проверку i < arr.Length     // 1 +
  // одна операция на инкремент i++               // 1 +
  {
    // создание переменной limit                  // 1 +
    // присвоение значения переменной limit       // 1 +
    // и вычисление arr.Length - 1 - i            // 1 +
    int limit = arr.Length - 1 - i;

    // две операции на int j = 0;                 // 2 +
    // остальные операции будут умножаться
    // на количество итераций цикла по j          // ((N - 1) / 2) × (

    // одна операция на проверку j < limit          // 1 +
    // одна операция на инкремент j++               // 1 +
    for (int j = 0; j < limit; j++)
    {
      // одна операция на сравнение                 // 1 +
      // + одна на вычисление j + 1                 // 1 +
      if (arr[j] > arr[j + 1])
      {
        // этот код может не выполниться ни разу
        // для уже отсортированного
        // по возрастанию массива,
        // а может выполняться всегда, если
        // массив отсортирован в обратном порядке

        // одна операция на присвоение              // 1 +
        // + одна на вычисление j + 1               // 1 +
        int temp = arr[j + 1];

        // одна операция на присвоение              // 1 +
        // + одна на вычисление j + 1               // 1 +
        arr[j + 1] = arr[j];

        // одна операция на присвоение              // 1
        arr[j] = temp;
                                                  // )
      }
    }
  }                                             // )
}
```

Подсчет O большого	**Подсчет Омега большого**
2 + (N - 1) × (7 + ((N - 1) / 2) × 9) 2 + (N - 1) × (4.5N + 3.5) 2 + 4.5N^2 + 3.5N - 4.5N - 3.5 4.5**N^2** - N - 1.5 **O(N^2)**	2 + (N - 1) × (7 + ((N - 1) / 2) × 4) 2 + (N - 1) × (2N + 5) 2 + 2N^2 + 5N - 2N - 5 2**N^2** + 3N - 3 **Ω(N^2)**

Квадратичное увеличение сложности – это весьма нехороший показатель и мы сейчас увидим это на примере. Давайте посчитаем время, требующееся сортировки пузырьком на входных данных разного порядка.

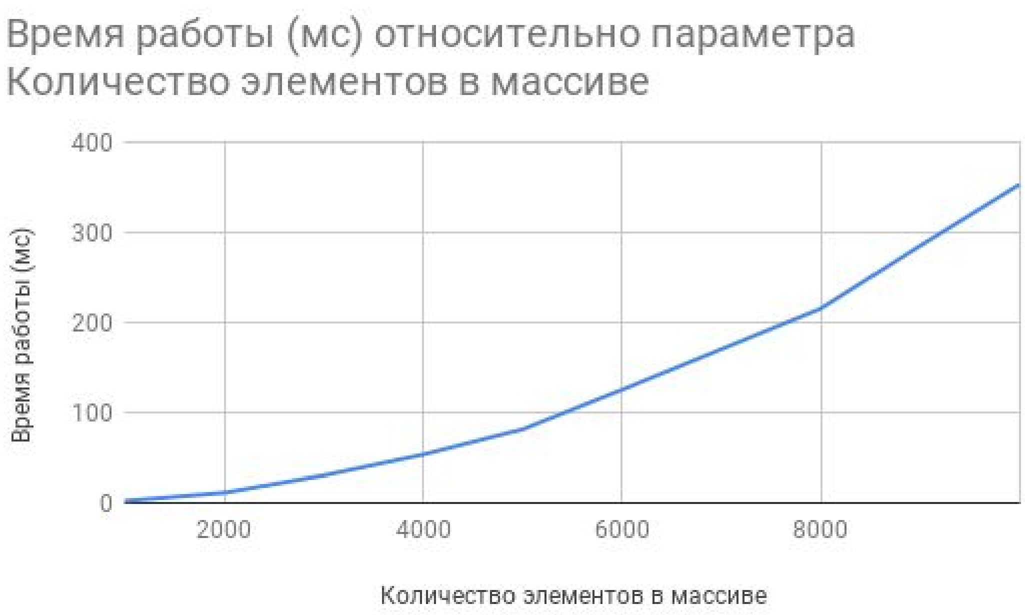
Подсчет времени работы

Для подсчета времени работы мы будем пользоваться классом **Stopwatch** (секундомер), расположенный в неймспейсе System.Diagnostics.

- Методы класса:
    - Start() : запустить секундомер
    - Stop() : остановить секундомер
    - Restart() : сбросить предыдущий замер и запустить секундомер
- Свойства класса:
    - ElapsedMilliseconds : количество миллисекунд между вызовами Start и Stop.

Оборачиваем замерами вызов функции BubbleSort

- 1000 элементов сортируется за ~2 мс
- 2000 элементов сортируется за ~11 мс
- 3000 элементов сортируется за ~30 мс
- 4000 элементов сортируется за ~53 мс
- 5000 элементов сортируется за ~81 мс
- 6000 элементов сортируется за ~125 мс
- 7000 элементов сортируется за ~170 мс
- 8000 элементов сортируется за ~215 мс
- 9000 элементов сортируется за ~285 мс
- 10000 элементов сортируется за ~353 мс



Как видно, увеличение времени работы растет по экспоненте относительно количества элементов массива. Это значит, что наш алгоритм скорее всего окажется нежизнеспособным на больших данных.

Типичные функции, к которым сводится расчёт сложности

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

- O(**1**) – константная сложность;
- O(**log(N)**) – логарифмическая сложность;
- O(**N**) – линейная сложность;
- O(**N×log(N)**) – квазилинейная сложность;
- O(**N^C**), где C > 1 – экспоненциальная сложность;
- O(**C^N**), где C > 1 – “гладкая” функция, она растет ещё быстрее, чем N^C
- O(**N!**) – факториальная сложность.

Встроенная сортировка .NET: Array.Sort

.NET имеет встроенные алгоритмы работы с данными и для сортировки, конечно же, имеется своя реализация.

Если заглянуть “под капот”, можно узнать, Array.Sort() динамически выбирает один из трех алгоритмов сортировки в зависимости от размера массива:
- Если размер меньше 16 элементов, используется "**сортировка вставками**" (Insertion Sort algorithm),
  Ω(N), **O(N^2)**
- Если размер превышает 2×log^N, где N - диапазон значений входного массива, используется алгоритм пирамидальной сортировки (Heap Sort algorithm).
  O(N×log(N)).
- В остальных случаях используется "быстрая сортировка" (Quicksort algorithm).
  Ω(N×log(N)), **O(N^2)**

На частично отсортированных данных все эти методы будут работать лучше, так как наша “пузырьковая” реализация даже в лучшем случае – это **Ω(N^2)**.

Самостоятельная работа

Дописываем программу таким образом, чтобы можно было сравнить наш алгоритм сортировки “пузырьком” с встроенной сортировкой .NET по времени (см. **L09_C04_bubble_vs_dotnet_sort.cs**).

Оценка алгоритмов относительно памяти

Аналогично проводят оценку и по памяти, когда это важно. Одни алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.

Домашнее задание

Посчитать асимптотическую сложность алгоритма вашего решения задачи прошлого урока (на валидацию скобок):

- наилучший случай – Ω,
- наихудший случай – O