



**Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ДИСЦИПЛИНА «Анализ алгоритмов»

Лабораторная работа № 3

Тема Алгоритмы сортировки

Студент Воякин А. Я.

Группа ИУ7-54Б

Преподаватели Волкова Л. Л., Строганов Ю. В.

Москва.
2020 г.

Оглавление

Введение	4
1 Аналитическая часть	5
1.1 Сортировка пузырьком	5
1.2 Сортировка вставками	5
1.3 Быстрая сортировка	5
1.4 Вывод	6
2 Конструкторская часть	7
2.1 Схемы алгоритмов	7
2.2 Трудоемкость алгоритмов	11
2.2.1 Сортировка пузырьком	11
2.2.2 Сортировка вставками	11
2.2.3 Быстрая сортировка	12
2.3 Вывод	13
3 Технологическая часть	14
3.1 Выбор ЯП	14
3.2 Сведения о модулях программы	14
3.3 Листинг кода	14
3.4 Вывод	15
4 Исследовательская часть	16
4.1 Примеры работы программы	16
4.2 Технические характеристики	16
4.3 Время выполнения алгоритмов	17
4.4 Вывод	19

Заключение	20
Список литературы	21

Введение

На сегодняшний день существует не одна вариация алгоритмов сортировки. Все они различаются по скорости сортировки и по объему необходимой памяти.

Цель данной лабораторной работы - обучение расчету трудоемкости алгоритмов

Алгоритмы сортировки часто применяются в практике программирования. В том числе в областях связанных с математикой, физикой, компьютерной графикой и т.д.

В ходе лабораторной работы предстоит:

- изучить алгоритмы сортировок;
- дать теоретическую оценку алгоритмам сортировки;
- реализовать три алгоритма сортировки на одном из языков программирования;
- сравнить алгоритмы сортировок.

1 | Аналитическая часть

1.1 Сортировка пузырьком

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов.

1.2 Сортировка вставками

На каждом шаге выбирается один из элементов неотсортированной части массива (максимальный/минимальный) и помещается на нужную позицию в отсортированную часть массива.

1.3 Быстрая сортировка

Общая идея алгоритма состоит в следующем:

1. Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.
2. Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньше опорного», «равные» и «большие».

3. Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения

1.4 Вывод

В данном разделе были рассмотрены три алгоритма сортировки.

2 | Конструкторская часть

2.1 Схемы алгоритмов

На рисунке (2.1) приведена схема классического алгоритма сортировки пузырьком.

На рисунке (2.2) приведена схема алгоритма сортировки вставками.

На рисунке (2.3) приведена схема быстрой сортировки.

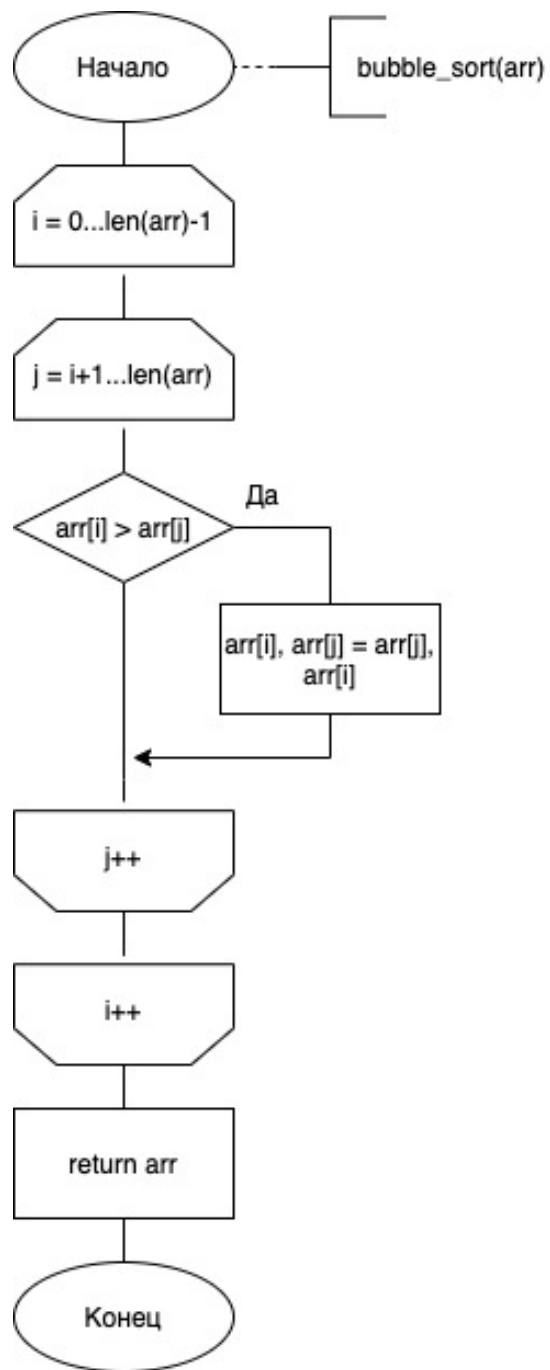


Рис. 2.1: Схема алгоритма сортировки пузырьком

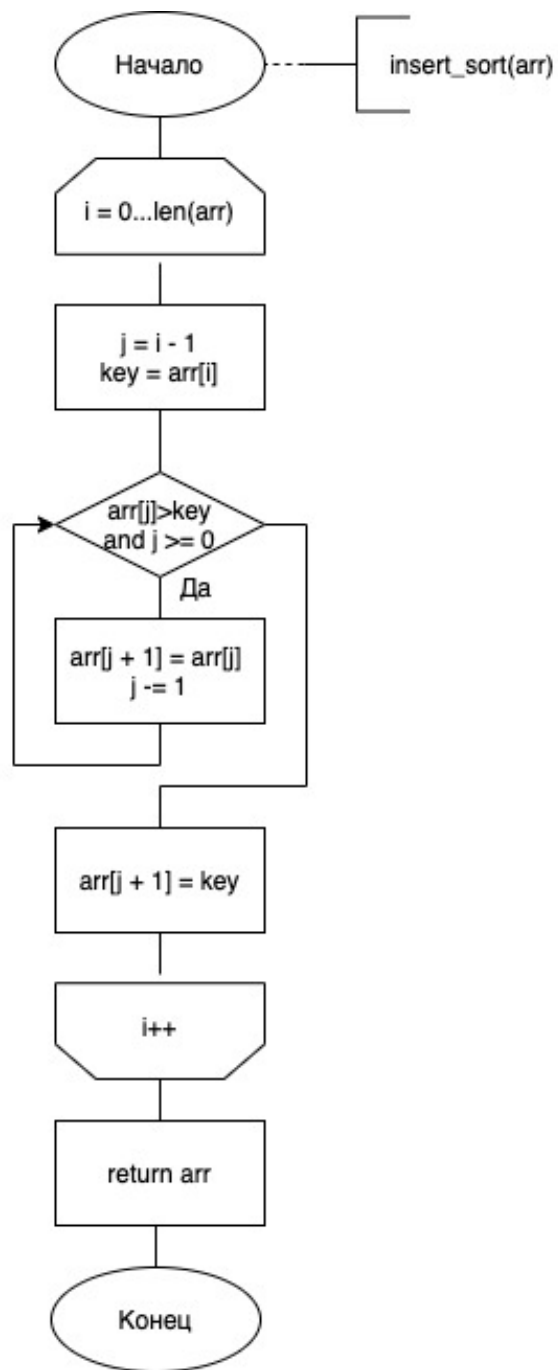


Рис. 2.2: Схема алгоритма сортировки вставками

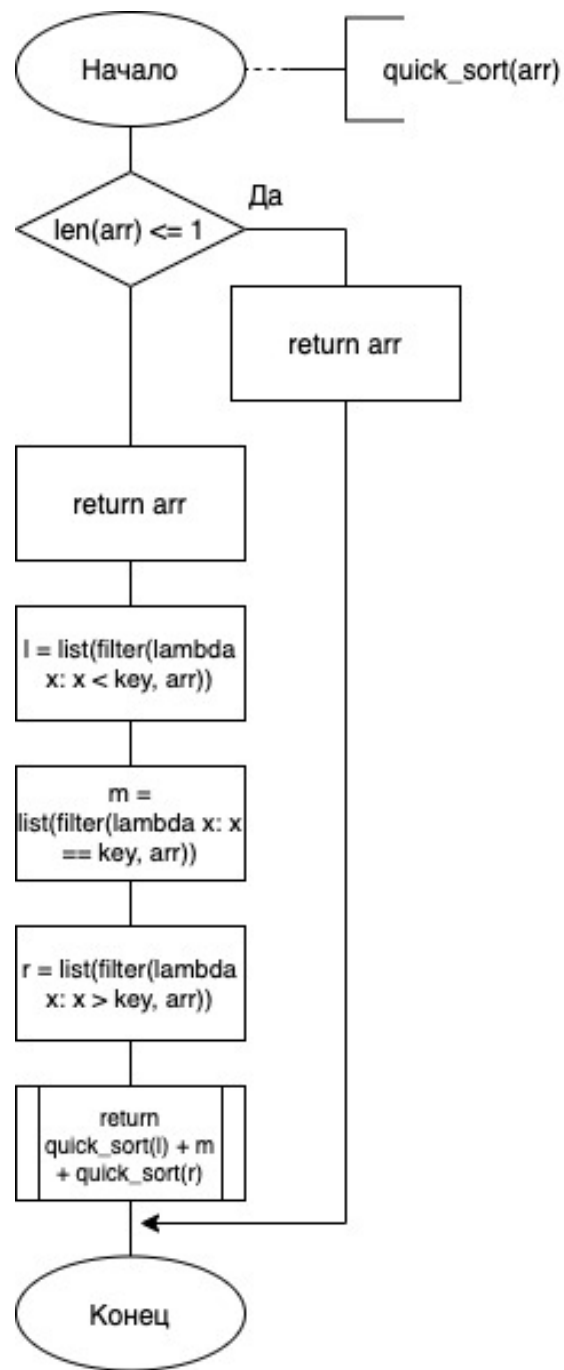


Рис. 2.3: Схема алгоритма быстрой сортировки

2.2 Трудоемкость алгоритмов

Введем модель трудоемкости для оценки алгоритмов:

1. базовые операции стоимостью 1 — +, -, *, /, =, ==, <=, >=, !=, +=, [], ++, — получение полей класса
2. оценка трудоемкости цикла: $F_{\text{ц}} = a + N^*(a + F_{\text{тела}})$, где a - условие цикла
3. стоимость условного перехода возьмем за 0, стоимость вычисления условия остаётся.

Далее будут приведены оценки трудоемкости алгоритмов. Построчная оценка трудоемкости сортировки пузырьком с флагом (Табл. 2.1).

2.2.1 Сортировка пузырьком

Лучший случай: Массив отсортирован; не произошло ни одного обмена за 1 проход -> выходим из цикла

Трудоемкость: $1 + 2n * (1 + 2n * 4) = 1 + 2n + 16n * n = O(n^2)$

Худший случай: Массив отсортирован в обратном порядке; в каждом случае происходил обмен

Трудоемкость: $1 + 2n * (1 + 2n * (4 + 5)) = O(n^2)$

2.2.2 Сортировка вставками

Табл. 2.1 Построчная оценка веса

Код	Вес
for i in range(1, len(a)):	$1+n*2$
key = a[i]	2
j = i-1	2
while (j >= 0 and a[j] > key):	$n*4$
a[j+1] = a[j]	4
j-= 1	1
a[j+1] = key	3

Лучший случай: отсортированный массив. При этом все внутренние циклы состоят всего из одной итерации.

Трудоемкость: $T(n) = 1 + 2n * (2 + 2 + 3) = 2n * 7 = 14n + 1 = O(n)$

Худший случай: массив отсортирован в обратном нужному порядке. Каждый новый элемент сравнивается со всеми в отсортированной последовательности. Все внутренние циклы будут состоять из j итераций.

Трудоёмкость: $T(n) = 1 + n * (2 + 2 + 4n * (4 + 1) + 3) = 2n * n + 7n + 1 = O(n^2)$

2.2.3 Быстрая сортировка

Лучший случай: сбалансированное дерево вызовов^[2] $O(n * \log(n))$ В наиболее благоприятном случае процедура PARTITION приводит к двум подзадачам, размер каждой из которых не превышает $\frac{n}{2}$, поскольку размер одной из них равен $\frac{n}{2}$, а второй $\frac{n}{2} - 1$. В такой ситуации быстрая сортировка работает намного производительнее, и время ее работы описывается следующим рекуррентным соотношением: $T(n) = 2T(\frac{n}{2}) + O(n)$, где мы не обращаем внимания на неточность, связанную с игнорированием функций “пол” и “потолок”, и вычитанием 1. Это рекуррентное соотношение имеет решение ; $T(n) = O(n \lg n)$. При сбалансированности двух частей разбиения на каждом уровне рекурсии мы получаем асимптотически более быстрый алгоритм.

Фактически любое разбиение, характеризующееся конечной константой пропорциональности, приводит к образованию дерева рекурсии высотой $O(\lg n)$ со стоимостью каждого уровня, равной $O(n)$. Следовательно, при любой постоянной пропорции разбиения полное время работы быстрой сортировки составляет $O(n \lg n)$.

Худший случай: несбалансированное дерево^[2] $O(n^2)$ Поскольку рекурсивный вызов процедуры разбиения, на вход которой подается массив размером 0, приводит к немедленному возврату из этой процедуры без выполнения каких-ли-бо операций, $T(0) = O(1)$. Таким образом, рекуррентное соотношение, описывающее время работы процедуры в указанном случае, записывается следующим образом: $T(n) = T(n - 1) + T(0) + O(n) = T(n - 1) + O(n)$. Интуитивно понятно, что при суммировании промежутков времени, затрачиваемых на каждый уровень рекурсии, получается арифметическая прогрессия, что приводит к результату $O(n^2)$.

2.3 Вывод

Сортировка пузырьком: лучший - $O(n)$, худший - $O(n^2)$

Сортировка вставками: лучший - $O(n)$, худший - $O(n^2)$

Быстрая сортировка: лучший - $O(n \lg n)$, худший - $O(n^2)$

3 | Технологическая часть

3.1 Выбор ЯП

Был выбран Python в качестве языка программирования, потому как он достаточно удобен и гибок. Среда разработки PyCharm.

Время работы алгоритмов было замерено с помощью функции `process_time_ns()` из библиотеки `time`.

3.2 Сведения о модулях программы

Программа состоит из:

- `lab03.py` - файл, в котором находятся функции сортировки
- `time_test.py` - файл, в котором реализовано тестирование алгоритмов по времени

3.3 Листинг кода

В листингах 3.1 - 3.3 приведены реализации алгоритмов сортировки массивов.

В листинге 3.4 приведена реализация функции замера процессорного времени.

Листинг 3.1: Сортировка пузырьком

```
1 def bubble_sort(arr):  
2     for i in range(len(arr) - 1):  
3         for j in range(i + 1, len(arr)):  
4             if arr[i] > arr[j]:
```

```

5         arr[i], arr[j] = arr[j], arr[i]
6     return arr

```

Листинг 3.2: Сортировка вставками

```

1 def insert_sort(arr):
2     for i in range(len(arr)):
3         j = i - 1
4         key = arr[i]
5         while arr[j] > key and j >= 0:
6             arr[j + 1] = arr[j]
7             j -= 1
8         arr[j + 1] = key
9     return arr

```

Листинг 3.3: Быстрая сортировка

```

1 def quick_sort(arr):
2     if len(arr) <= 1:
3         return arr
4     key = choice(arr)
5     l = list(filter(lambda x: x < key, arr))
6     m = list(filter(lambda x: x == key, arr))
7     r = list(filter(lambda x: x > key, arr))
8     return quick_sort(l) + m + quick_sort(r)

```

Листинг 3.4: Функция замера процессорного времени

```

1 def count_time(func, arr):
2     start = process_time_ns()
3     func(arr)
4     end = process_time_ns()
5     return end - start

```

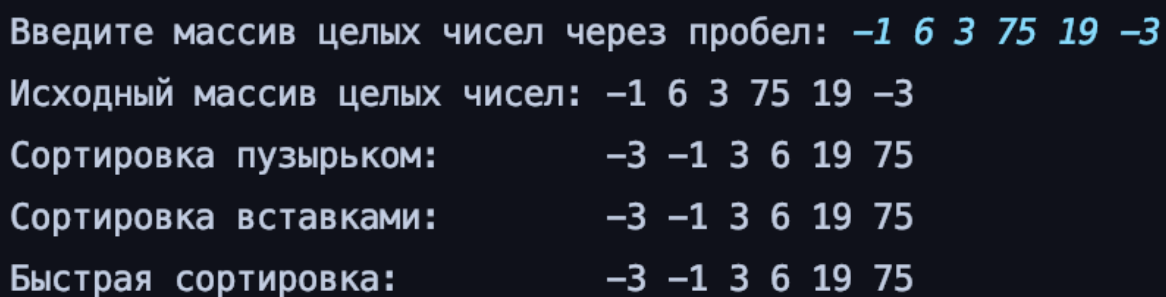
3.4 Вывод

В данной части был описан выбор языка программирования, приведена информация о модулях программы и отображены листинги реализаций трёх алгоритмов сортировки массивов, функции замера процессорного времени.

4 | Исследовательская часть

4.1 Примеры работы программы

На рисунке (4.1) приведена демонстрация работы программы.



```
Введите массив целых чисел через пробел: -1 6 3 75 19 -3
Исходный массив целых чисел: -1 6 3 75 19 -3
Сортировка пузырьком:          -3 -1 3 6 19 75
Сортировка вставками:          -3 -1 3 6 19 75
Быстрая сортировка:            -3 -1 3 6 19 75
```

Рис. 4.1: Пример работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: MacOS Big Sur версия 11.0.1;
- Оперативная память: 8 GB;
- Процессор: Intel(R) Core(TM) i5-8210Y CPU @ 1.60GHz.

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.3 Время выполнения алгоритмов

Произведено измерение времени работы алгоритмов на случайно сгенерированных, отсортированных по возрастанию и убыванию массивах. Для замера времени была использована функция `process_time_ns`.

Измерение было произведено 50 раз и результат был усреднён.

На рисунках 4.2 - 4.4 приведены графики отображающие время работы алгоритмов в nano секундах от длины массивов для неупорядоченных, отсортированных по возрастанию, отсортированных по убыванию массивов соответственно.

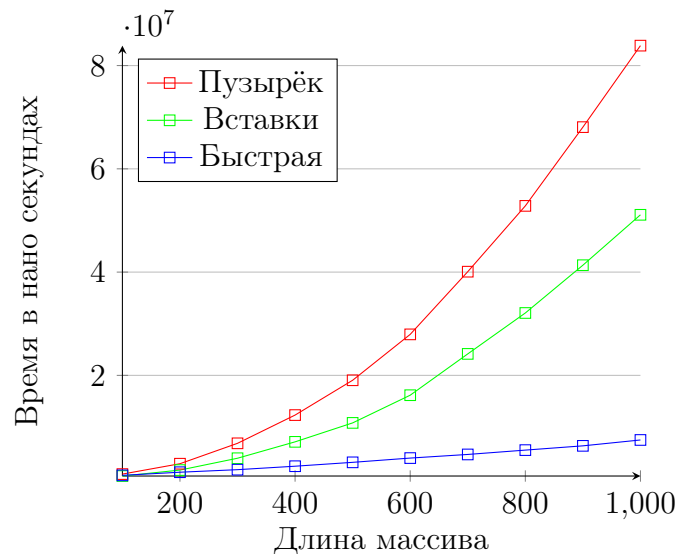


Рис. 4.2: Сравнение времени сортировок неупорядоченных массивов

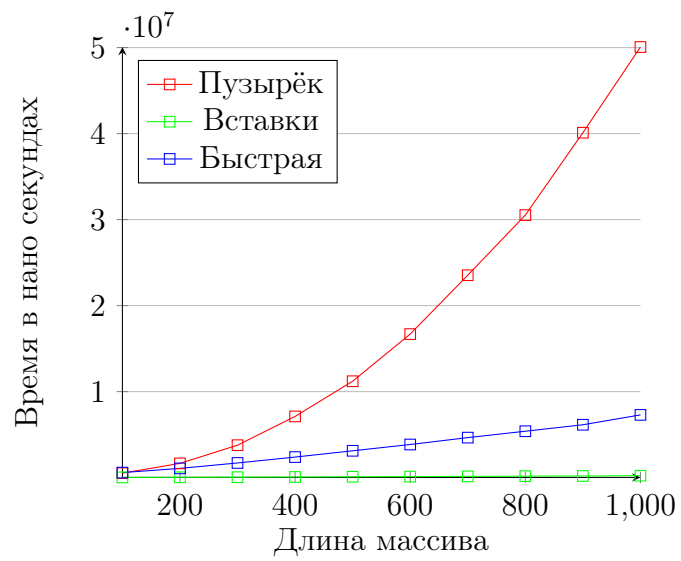


Рис. 4.3: Сравнение времени сортировок отсортированных по возрастанию массивов

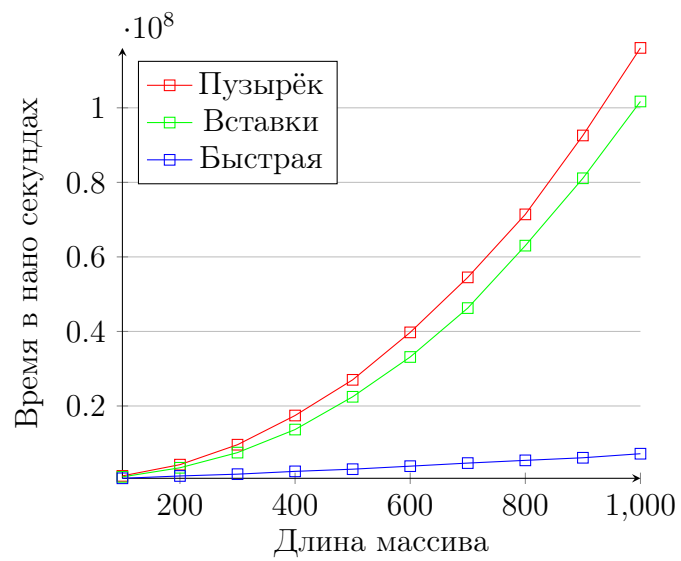


Рис. 4.4: Сравнение времени сортировок отсортированных по убыванию массивов

4.4 Вывод

Были протестированы алгоритмы сортировки на массивах размерами 100...1000 с шагом 100. Рассмотрены нупорядоченные, отсортированные по возрастанию, отсортированные по убыванию массивы.

Экспериментально было подтверждено, что при сортировке отсортированных по возрастанию массивов сортировка вставками показывает наилучший результат.

Исходя из графиков наглядно видно, что сортировка пузырьком является самой медленной.

Заключение

В ходе выполнения данной лабораторной работы были реализованы три алгоритма сортировки: сортировка пузырьком, сортировка вставками и быстрая сортировка. Был проведён анализ каждого алгоритма и измерено время работы алгоритмов для массивов разных размеров. Была оценена трудоёмкость алгоритмов.

Список литературы

1. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Глава 7. Быстрая сортировка // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005.
2. Левитин А. В. Глава 4. Метод декомпозиции: Быстрая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.