

МГТУ им. Н.Э. БАУМАНА

ЛАБОРАТОРНАЯ РАБОТА №3

По курсу: "АНАЛИЗ АЛГОРИТМОВ"

Алгоритмы сортировки массивов

Работу выполнил: Луговой Дмитрий, ИУ7-51Б

Преподаватель: Волкова Л.Л.

Москва, 2019

Оглавление

Введение	3
1 Аналитическая часть	5
1.1 Сортировка вставками	5
1.2 Сортировка расческой	5
1.3 Быстрая сортировка	6
2 Конструкторская часть	7
2.1 Схемы алгоритмов	8
2.2 Расчет трудоемкости	11
3 Технологическая часть	13
3.1 Требования к ПО	13
3.2 Средства реализации	13
3.3 Листинги кода	13
4 Экспериментальная часть	15
4.1 Примеры работы	15
4.2 Функциональное тестирование	16
4.3 Сравнение алгоритмов по времени	16
Заключение	18

Введение

Целью данной лабораторной работы является исследование существующих алгоритмов сортировки массивов и их трудоемкости.

Примем следующую модель вычислений:

1. Трудоемкость базовых операций
Операции $+$, $-$, $*$, $/$, $\%$, $=$, $>$, $<$, \leq , \geq , $==$, \neq , $[]$, $+=$, $- =$ - имеют стоимость 1.
2. Трудоемкость условного перехода
Условный переход имеет стоимость 0, при этом оцениваем расчет условия:

```
if (n % 2 == 1)
{
    // Тело 1
}
else
{
    // Тело 2
}
```

$$f_{if} = f_{условия} + \begin{cases} f_{\text{тело 1}}, & \text{при нечетном } N \\ f_{\text{тело 2}}, & \text{при четном } N \end{cases}$$

3. Трудоемкость цикла *for*

$$f_{\text{цикла}} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}),$$

где N - число повторений цикла.

Задачи работы

Задачами данной лабораторной являются:

1. Реализовать следующие алгоритмы сортировки массивов:
 - Сортировка вставками
 - Сортировка расческой
 - Быстрая сортировка
2. Проанализировать трудоемкость данных алгоритмов
3. Провести эксперименты с замерами времени

1 | Аналитическая часть

В этом разделе содержатся описания алгоритмов сортировки массивов.

1.1 Сортировка вставками

Сортировка вставками (англ. Insertion sort) — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. Сортировка вставками делит массив на 2 части — отсортированную и неотсортированную. Из неотсортированной части извлекается текущий элемент. Поскольку другая часть массива отсортирована, то в ней достаточно быстро можно найти место для этого извлечённого элемента. Элемент вставляется куда нужно, в результате чего отсортированная часть массива увеличивается, а неотсортированная уменьшается.

1.2 Сортировка расческой

Сортировка расческой (англ. Comb sort) - это модификация пузырьковой сортировки. Производятся неоднократные прогоны по массиву, при которых сравниваются пары элементов. Если они неотсортированы друг относительно друга - то производится обмен. В результате крупные элементы мигрируют в конец массива, а небольшие по значению - в начало. В пузырьковой сортировке при каждом прогоне по массиву сравниваются соседние элементы. Здесь же сравниваются элементы, между которыми некоторое фиксированное расстояние. При каждом следующем прохождении расстояние уменьшается, пока не достигнет минимальной величины. Уменьшающееся расстояние между сравниваемыми элементами рассчитывается с помощью специальной величины, называемой фактором уменьшения. Длина массива делится на этот фактор, это и есть разрыв между индексами. После каждого прохода расстояние делится на фактор уменьшения и таким образом получается новое значение. В конце концов оно сужается до минимального значения - единицы, и массив просто досортировывается обычным "пузырьком". В результате практических тестов и теоретических исследований оптимальное значение для фактора уменьшения определено следующее: $\frac{1}{1-\frac{1}{e^{\frac{1}{2}}}}$

1.3 Быстрая сортировка

Быстрая сортировка (англ. Quick sort) — один из самых известных и широко используемых алгоритмов сортировки. Общая идея алгоритма состоит в следующем:

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

2 | Конструкторская часть

В этом разделе содержатся схемы алгоритмов сортировки массивов и подсчет трудоемкости.

2.1 Схемы алгоритмов

На Рис.2.1 представлена схема алгоритма сортировки массивов вставками.

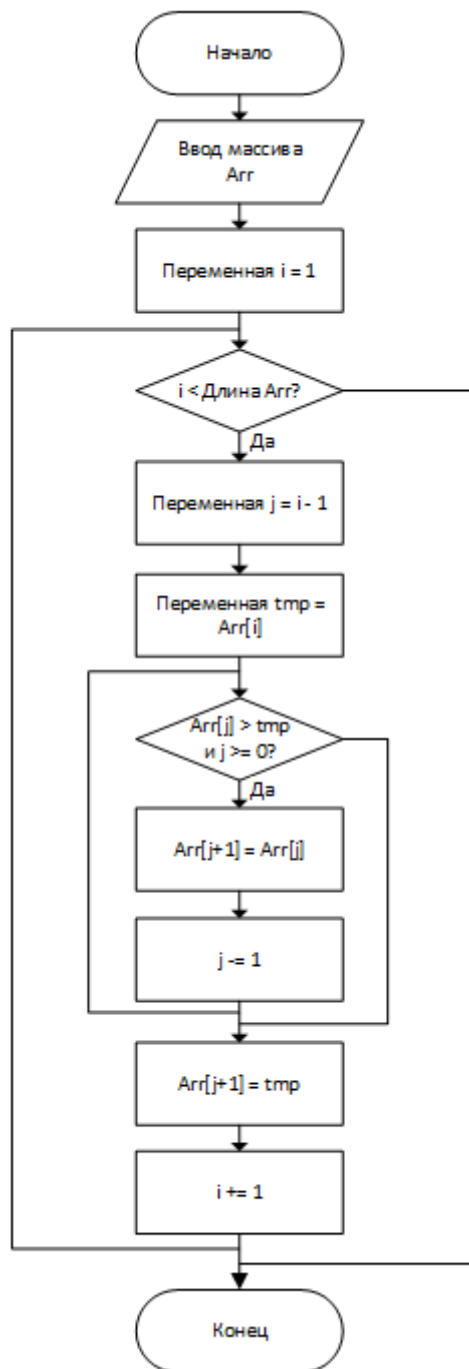


Рис. 2.1: Алгоритм сортировки вставками

На Рис.2.2 представлена схема алгоритма сортировки массивов расческой.

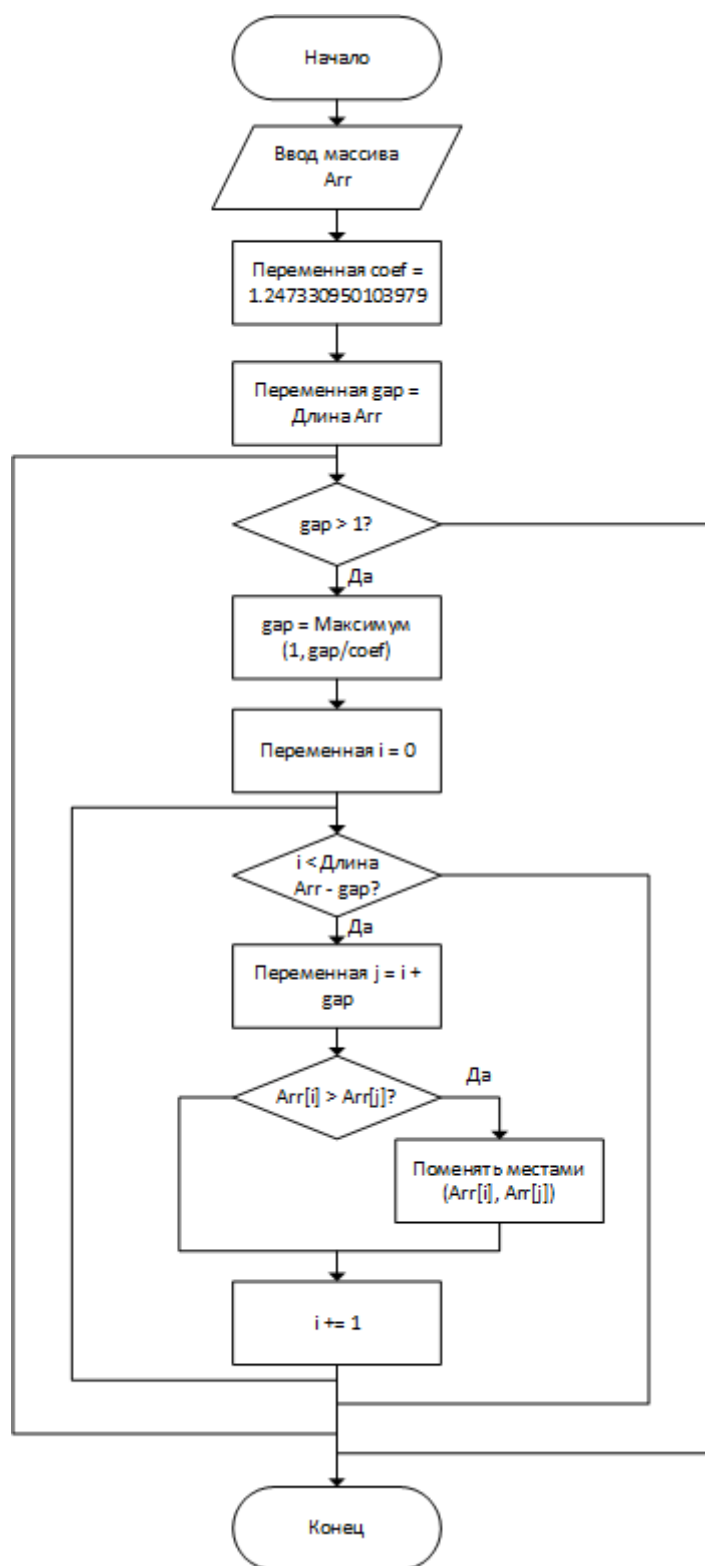


Рис. 2.2: Алгоритм сортировки расческой

На Рис.2.3 представлена схема алгоритма быстрой сортировки массивов.

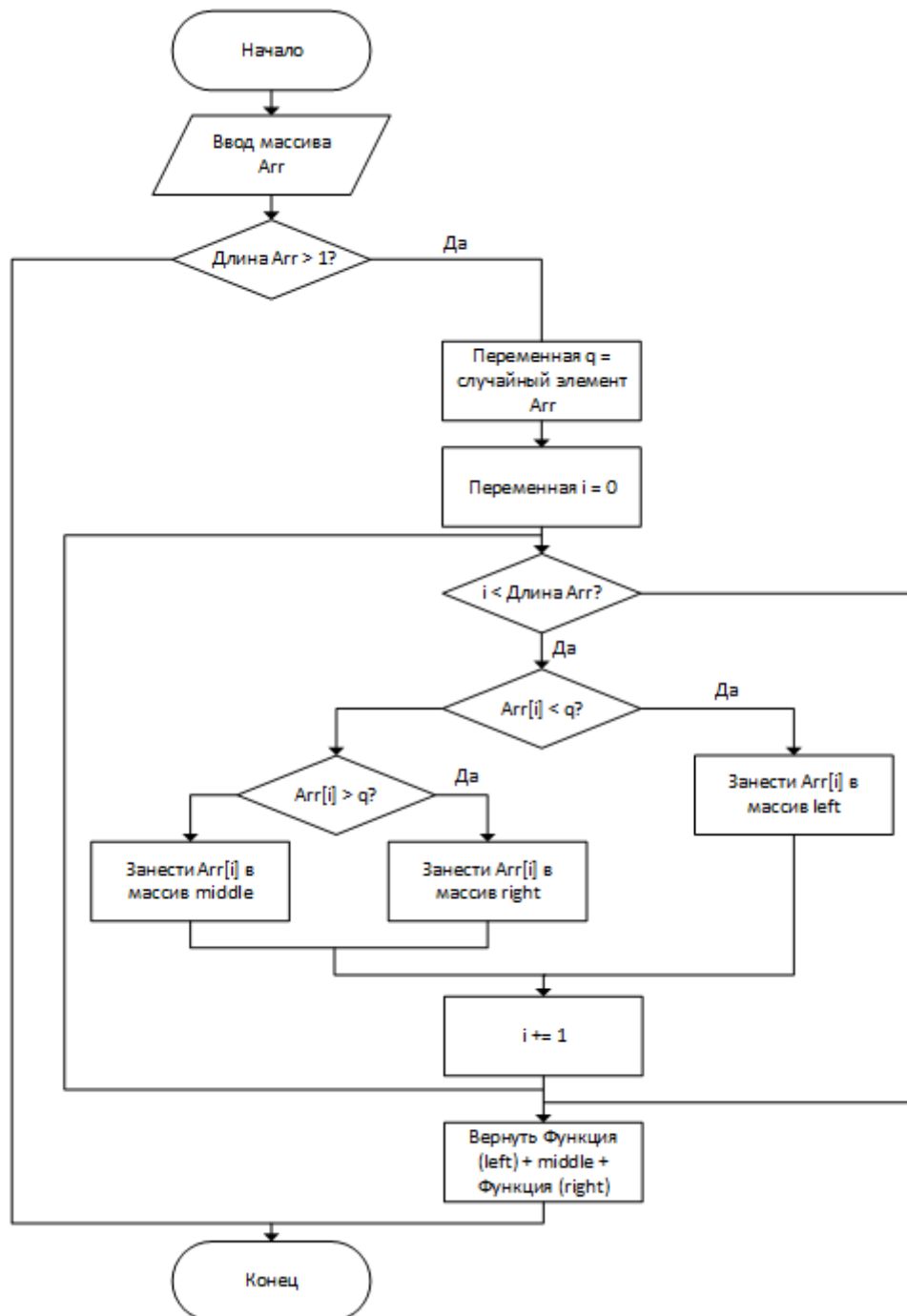


Рис. 2.3: Алгоритм быстрой сортировки

2.2 Расчет трудоемкости

Пусть задан массив размерности N . Используя модель вычислений, заданную ранее, произведем подсчет трудоемкости алгоритмов сортировки:

1. Сортировка вставками

- **Лучший случай** - отсортированный массив. В этом случае внутренний цикл выполняться не будет:

$$f_{\text{вставками}} = 2 + (N - 1)(2 + 2 + 2 + 2 + 3) = 11N - 9 \approx 11N$$

- **Худший случай** - обратно отсортированный массив. В этом случае внутренний цикл будет выполняться максимально большое количество раз.

$$\begin{aligned} f_{\text{вставками}} &= 2 + (N - 1)(2 + 2 + 2 + 3) + \frac{4 + 4 + 1 + (4 + 4 + 1)(N - 1)}{2}(N - 1) = \\ &= 4.5N^2 + 4.5N - 7 \approx N^2 \end{aligned}$$

2. Сортировка расческой

- **Лучший случай** - полностью отсортированный массив, как и для сортировки пузырьком, модификацией которой она является. В отличие от сортировки пузырьком, данная сортировка отлично справляется с "черепашками" - элементами которые нужно протащить через весь массив, поэтому имеет трудоемкость: $O(n \cdot \log n)$.
- **Худший случай** - обратно отсортированный массив, как и для сортировки пузырьком. Трудоемкость: $O(n^2)$, однако является самой быстрой из квадратичных сортировок.

3. Быстрая сортировка

- **Лучший случай** - в наиболее сбалансированном варианте при каждой операции деления массив делится на две одинаковые (плюс-минус один элемент) части, следовательно, максимальная глубина рекурсии, при которой размеры обрабатываемых подмассивов достигнут 1, составит $\log_2 n$. В результате количество сравнений, совершаемых быстрой сортировкой, было бы равно значению рекурсивного выражения $C_n = 2 \cdot C_{n/2} + n$, что даёт общую сложность алгоритма $O(n \cdot \log n)$.

- **Худший случай** - в самом несбалансированном варианте каждое разделение даёт два подмассива размерами 1 и $n - 1$, то есть при каждом рекурсивном вызове больший массив будет на 1 короче, чем в предыдущий раз. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых. При простейшем выборе опорного элемента — первого или последнего в массиве, — такой эффект даст уже отсортированный (в прямом или обратном порядке) массив, для среднего или любого другого фиксированного элемента «массив худшего случая» также может быть специально подобран. В этом случае потребуется $n - 1$ операций разделения, а общее время работы составит $\sum_{i=0}^n (n - i) = O(n^2)$ операций.

3 | Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода

3.1 Требования к ПО

На вход поступает массив чисел, на выходе должен возвращаться результат его сортировки.

3.2 Средства реализации

Для реализации представленных алгоритмов был выбран язык Python. Время работы алгоритмов было замерено с помощью функции `perf_counter()` из библиотеки `time`.

3.3 Листинги кода

В Листинге 3.1 показана реализация алгоритма сортировки вставками.

Листинг 3.1: Функция сортировки вставками

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

В Листинге 3.2 показана реализация алгоритма сортировки расческой.

Листинг 3.2: Функция сортировки расческой

```
1 COMB_COEF = 1.247330950103979
2
3 def comb(arr):
4     result = list(arr)
5     gap = len(result)
6     while gap > 1:
7         gap = max(1, int(gap / COMB_COEF))
8         for i in range(len(result) - gap):
9             j = i + gap
10            if result[i] > result[j]:
11                result[i], result[j] = result[j], result[i]
12    return result
```

В Листинге 3.3 показана реализация алгоритма быстрой сортировки.

Листинг 3.3: Функция быстрой сортировки

```
1 def quick(arr):
2     if len(arr) <= 1:
3         return arr
4     else:
5         q = random.choice(arr)
6         left = []
7         middle = []
8         right = []
9         for elem in arr:
10            if elem < q:
11                left.append(elem)
12            elif elem > q:
13                right.append(elem)
14            else:
15                middle.append(elem)
16    return quick(left) + middle + quick(right)
```

4 | Экспериментальная часть

В данном разделе приведены примеры работы программы, постановка эксперимента и сравнительный анализ алгоритмов на основе экспериментальных данных.

4.1 Примеры работы

Пример 1

Матрица A:

1 2 3

4 5 6

Матрица B:

1

2

3

Результирующая матрица:

14

32

Пример 2

Матрица A:

5 2

1 4

Матрица B:

0 3

-6 1

Результирующая матрица:

-12 17

-24 7

Пример 3

Матрица A:

2 7

1 3

Матрица B:

-3 7

1 -2

Результирующая матрица:

1 0

0 1

4.2 Функциональное тестирование

Было проведено функциональное тестирование программы, результаты которого занесены в Таблицу 4.1, 1 столбец которой - номер тестового случая, 2 и 3 столбцы - виды матриц, поступающих на вход, 4 столбец - ожидаемый результат, где *None* означает, что матрицы несовместимы по размеру, 5 столбец - полученный результат.

№	A	B	Ожидаемый результат	Полученный результат
1	Случайная	Пустая	None	None
2	Пустая	Случайная	None	None
3	Пустая	Пустая	None	None
4	Случайная	Нулевая	Нулевая	Нулевая
5	Нулевая	Случайная	Нулевая	Нулевая
6	Единичная	Квадратная	B	B
7	Квадратная	Единичная	A	A
8	Размера $M \times N$	Размера $M \times N$	None	None

Таблица 4.1: Тестовые случаи

Программа успешно прошла все тестовые случаи, все полученные результаты совпали с ожидаемыми.

4.3 Сравнение алгоритмов по времени

Для экспериментов использовались матрицы, размер которых варьируется от 100×100 до 1000×1000 с шагом 100 для матриц четных размеров и от 101×101 до 1001×1001 с шагом 100 для матриц нечетных размеров. Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными.

На Рис. 4.1 видно, что оптимизированный алгоритм Винограда превосходит стандартный алгоритм умножения на $\approx 30\%$, а алгоритм Винограда на $\approx 45\%$.

|

На Рис. 4.2 видно, что оптимизированный алгоритм Винограда сохраняет свое превосходство и при нечетных размерах матриц, стандартный алгоритм не изменил своего времени работы, алгоритм Винограда стал работать на пренебрежимо малое количество времени дольше.

Заключение

В ходе работы были изучены и реализованы алгоритмы стандартного умножения матриц, алгоритма Винограда и оптимизированного алгоритма Винограда. Был проведен сравнительный анализ перечисленных алгоритмов по трудоемкости и экспериментально подтверждено различие во временной эффективности. Классический алгоритм в неоптимизированном виде является более эффективным, чем алгоритм Винограда, однако при ряде оптимизаций алгоритм Винограда становится значительно быстрее классического. При этом разница лучшего и худшего случая алгоритма Винограда пренебрежимо мала.