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»**

Отчет по лабораторной работе №3 по курсу «Анализ алгоритмов»

Тема Сравнение алгоритмов сортировки

Студент Саркисов А.С.

Группа ИУ7-53Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

На данный момент существует огромное количество вариаций сортировок. Эти алгоритмы необходимо уметь сравнивать, чтобы выбирать наилучше подходящие в конкретном случае.

Эти алгоримы оцениваются по:

- Времени быстродействия
- Затратам памяти

Целью данной лабораторной работы является изучение применений алгоритмов сортировки, обучение расчету трудоемкости алгоритмов.

1 | Аналитическая часть

1.1 Описание алгоритмов

Сортировка массива — одна из самых популярных операций над массивом. Алгоритмы реализуют упорядочивание элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. Область применения:

- физика,
- математика,
- экономика,
- итд.

1.1.1 Сортировка пузырьком

Алгоритм проходит по массиву n-1 раз или до тех пор, пока массив не будет полностью отсортирован. В каждом проходе элементы попарно сравниваются и, при необходимости, меняются местами. При каждом проходе алгоритма по внутреннему циклу, очереднй наибольший элемент ставится на своё место в конец неотсортированного массива. Таким образом наибольшие элементы "всплывают"как пузырек.

1.1.2 Сортировка вставками

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

1.1.3 Быстрая сортировка

Массив разбивается на два (возможно пустых) подмассива. Таких, что в одном подмассиве каждый элемент меньше либо равен опорному, и при этом не превышает любой элемент второго подмассива. Опорный элемент вычислыется в ходе процедуры разбиения. Подмассивы сортируются с помощью рекурсивного вызова процедуры быстрой сортировки. Поскольку подмассивы сортируются на месте, для их объединения не трубуются никакие действия.

2 | Конструкторская часть

2.1 Схемы алгоритмов

В данном разделе будут рассмотрены схемы алгоритмов пузырьком с флагом (2.1), сортировки вставками (2.2), быстрой сортировки (2.3).

Ниже представлен алгоритм сортировки пузырьком с флагом (2.1).

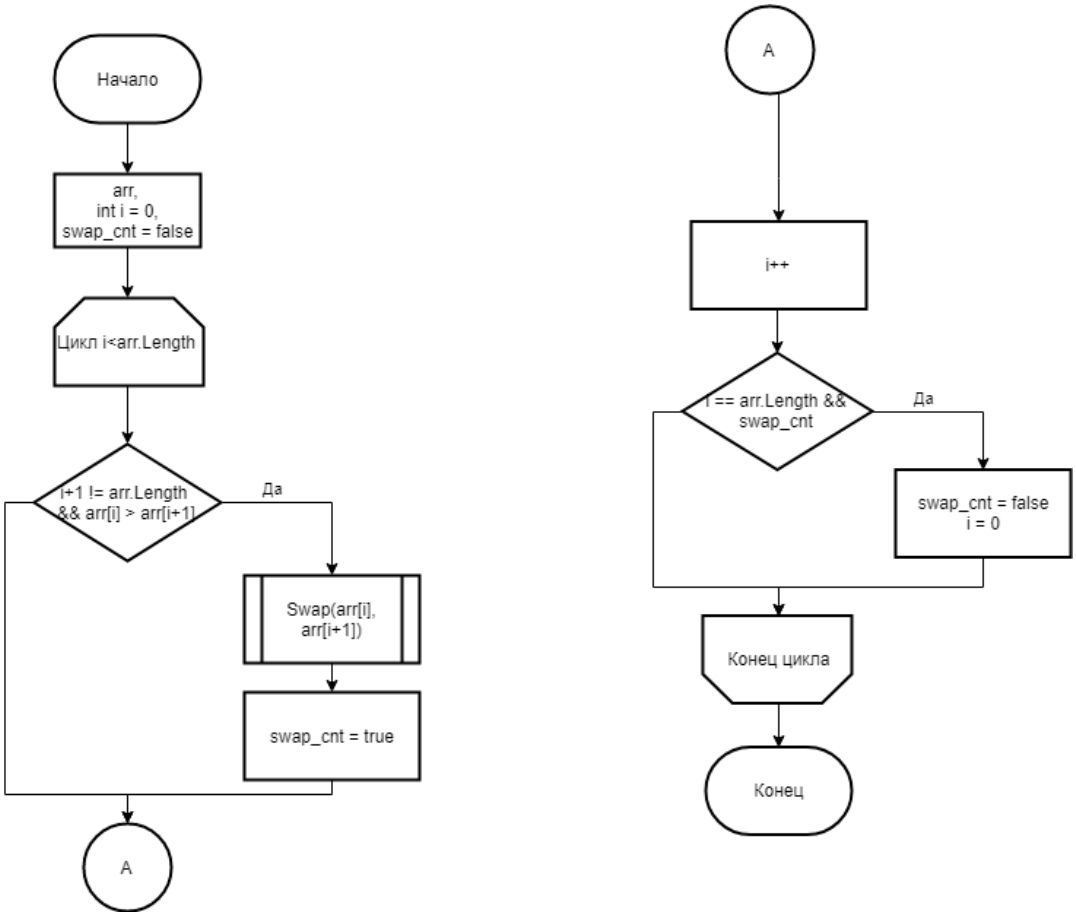

Рис. 2.1: Схема алгоритма сортировки пузырьком с флагом

Ниже представлен алгоритм сортировки вставкой (2.2).

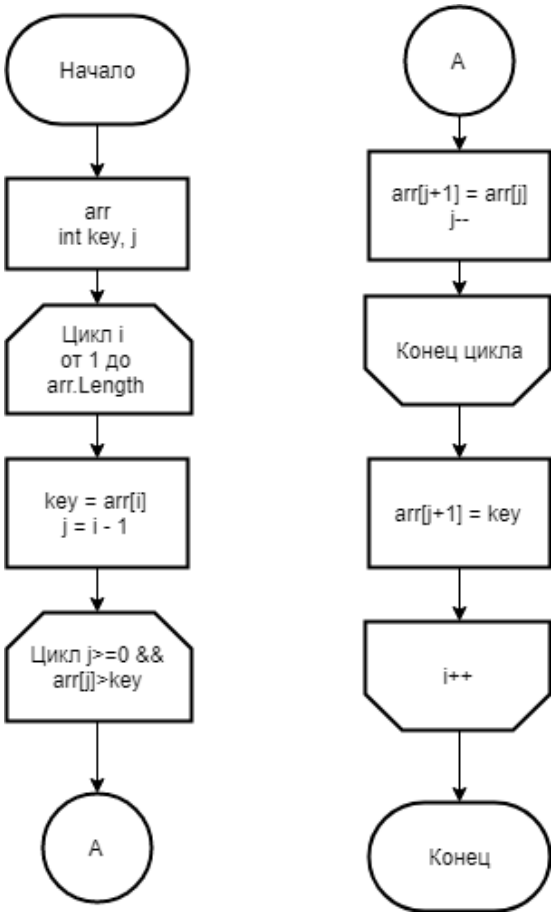

Рис. 2.2: Схема алгоритма сортировки вставками

Ниже представлен алгоритм быстрой сортировки (2.3).

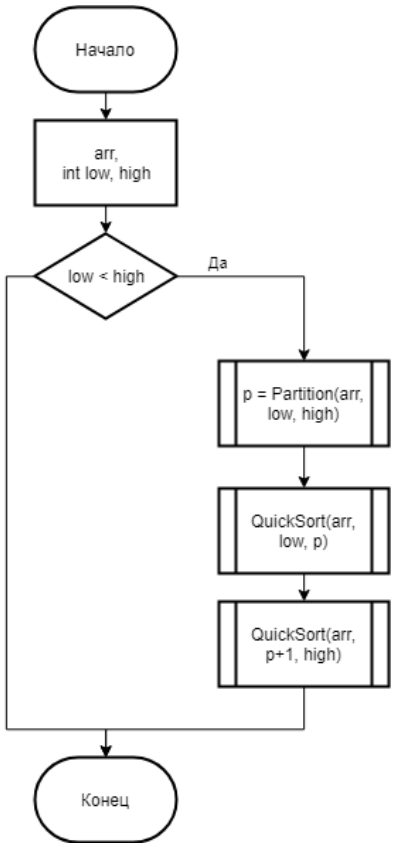

Рис. 2.3: Схема алгоритма быстрой сортировки

2.2 Трудоемкость алгоритмов

Введем модель трудоемкости для оценки алгоритмов:

1. базовые операции стоимостью 1 — +, -, *, /, =, ==, <=, >=, !=, +=, [], получение полей класса

2. оценка трудоемкости цикла: Fц = a + N*(a + Fтела), где a - условие цикла

3. стоимость условного перехода возьмем за 0, стоимость вычисления условия остаётся.

Далее будет приведены оценки трудоемкости алгоритмов. Построчная оценка трудоемкости сортировки пузырькм с флагом (Табл. 2.1).

2.2.1 Сортировка пузырьком с флагом

Построчная оценка веса сортировки пузырьком с флагом представлена в таблице 2.1.

Табл. 2.1 Построчная оценка веса

Код	Вес
int i = 0;	1
bool swapcnt = false;	1
while (i < arr.Length)	2
{	0
if(i + 1 != arr.Length && arr[i] > arr[i + 1])	7
{	0
Swap(ref arr[i], ref arr[i + 1]);	2+3
swapcnt = true;	1
}	0
i++;	1
if (i == arr.Length && swapcnt)	3
{	0
swapcnt = false;	1
i = 0;	1
}	0
}	0

Лучший случай: Массив отсортирован; не произошло ни одного обмена за 1 проход -> выходим из цикла
Трудоемкость: $1 + 1 + n * (2 + 7 + 1 + 3) + 2 = 13n + 4 = O(n)$

Худший случай: Массив отсортирован в обратном порядке; в каждом случае происходил обмен
Трудоемкость: $1+1+n*(n*(7+5+1+3)+1+1)+2 = n*(16n+2)+4 = 16n^2 + 2n + 4 = O(n^2)$

2.2.2 Сортировка вставками

Лучший случай: отсортированный массив. При этом все внутренние циклы состоят всего из одной итерации.
Трудоемкость: $T(n) = 3n + ((2 + 2 + 4 + 2) * (n - 1)) = 3n + 10(n - 1) = 13n - 10 = O(n)$

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.
Трудоемкость: $T(n) = 3n+(2+2)(n-1)+4\left(\frac{n(n+1)}{2} - 1\right)+5\frac{n(n-1)}{2}+3(n-1) = 3n+4n-4+2n^2+2n-4+2.5n^2-2.5n+3n-3 = 4.5n^2+9.5n-11 = O(n^2)$

2.2.3 Быстрая сортировка

Лучший случай: сбалансированное дерево вызовов $O(n * log(n))$ 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 ; $T(n) = O(nlgn)$. 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 $O(lgn)$ со стоимостью каждого уровня, равной $O(n)$. Следовательно, прилюбой постоянной пропорции разбиения полное время работы быстрой сортировки составляет $O(nlgn)$.

Худший случай: несбалансированное дерево $O(n^2)$ 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 - $O(nlgn)$, худший - $O(n^2)$

При этом сортировка вставками быстрее пузырька с флагом в худшем случае т.к. имеет меньший коэффициент. Вставки $4.5n^2$, пузырек $16n^2$.

3 | Технологическая часть

3.1 Выбор ЯП

В качестве языка программирования был выбран Golang, а средой разработки Visual Studio. Время работы алгоритмов было замерено с помощью встроенного модуля Benchmark.

3.2 Описание структуры ПО

Ниже представлена IDEF0 диаграмма (3.1), описывающая структуру программы.

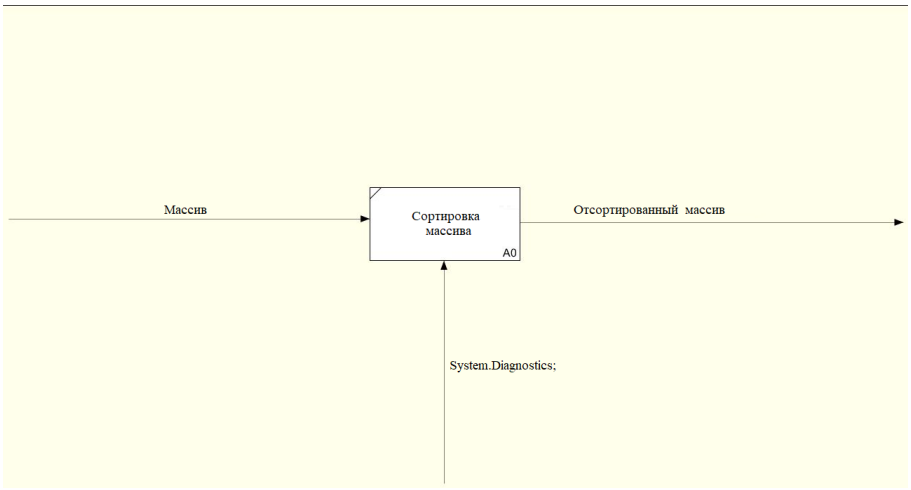

Рис. 3.1: Функциональная схема сортировки массива (IDEF0 диаграмма 1 уровня)

3.3 Сведения о модулях программы

Программа состоит из:

- main.go - главный файл программы, в котором располагается точка входа в программу и функция замера времени.
- sorts.go - файл sort, в котором находятся алгоримы сортировки

3.4 Листинг кода

Ниже представлен код алгоритма сортировки пузырьком в листинге (3.1).

Листинг 3.1: Алгоритм сортировки пузырьком

```
1 func BubbleSort(intSlice []int) {
2   for i := len(intSlice); i > 0; i-- {
3     for j := 0; j < i-1; j++ {
4       if intSlice[j] > intSlice[j+1] {
5         intSlice[j], intSlice[j+1] = intSlice[j+1],
              intSlice[j]
6       }
7     }
8   }
9 }
```

Ниже представлен код алгоритма сортировки вставками в листинге (3.2).

Листинг 3.2: Алгоритм сортировки вставками

```
func InsertionSort(intSlice []int) {
  var n = len(intSlice)
  for i := 1; i < n; i++ {
    j := i
    for j > 0 {
      if intSlice[j-1] > intSlice[j] {
        intSlice[j-1], intSlice[j] = intSlice[j], intSlice[
            j-1]
      }
      j = j - 1
    }
  }
}
```

Ниже представлен код алгоритма быстрой сортировки в листинге (3.3).

Листинг 3.3: Алгоритм быстрой сортировки

```
func QuickSort(intSlice []int) []int {
  if len(intSlice) < 2 {
    return intSlice
  }
  left, right := 0, len(intSlice)-1
  pivot := rand.Int() % len(intSlice)
  intSlice[pivot], intSlice[right] = intSlice[right],
     intSlice[pivot]
  for i := range intSlice {
    if intSlice[i] < intSlice[right] {
      intSlice[left], intSlice[i] = intSlice[i], intSlice[
          left]
      left++
    }
  }
  intSlice[left], intSlice[right] = intSlice[right],
     intSlice[left]
  QuickSort(intSlice[:left])
  QuickSort(intSlice[left+1:])
```

```
17    return intSlice
18 }
```

3.5 Вывод

В данном разделе были представлены реализации алгоритмов сортировки пузырьком (рис. 3.1), вставками (рис. 3.2) и быстрой сортировки (рис. 3.3).

4 | Исследовательская часть

Был проведен замер времени работы каждого из алгоритмов.

4.1 Примеры работы программы

Ниже представлен пример работы программы сортировки массива, заполненного случайными числами на рисунке 4.1.

```
Enter a number of array's elements: 10
The random slice of integers: 177 -131 -655 -15 372 522 -256 915 358 -8

Sorted array: -655 -256 -131 -15 -8 177 358 372 522 915
```

Рис. 4.1: Сортировка массива, заполненного случайными числами

Ниже представлен пример работы программы сортировки массива, заполненного случайными числами на рисунке 4.2.

```
Enter a number of array's elements: 10
The random slice of integers: 203 598 524 245 -517 -225 271 86 426 440

Sorted array: -517 -225 86 203 245 271 426 440 524 598
```

Рис. 4.2: Сортировка массива, заполненного случайными числами

4.2 Эксперимент

В рамках данного эксперимента было произведено сравнение времени выполнения трех алгоритмов в лучшем/худшем/случайном случае заполения массива. При длине массивов от 100 до 1000 элементов с шагом 100. На предоставленных ниже графиках (Рис. 4.1), (Рис. 4.2), (Рис. 4.3) по оси l идет длина массива, а по оси t - время сортировки в тиках.

Ниже представлено сравнение работы алгоритмов в лучшем случае на графике 4.1.

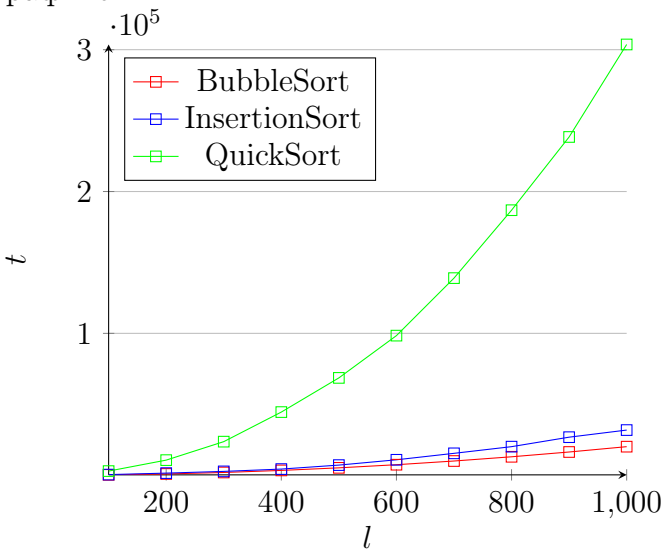

Рис. 4.1: Сравнение времени в лучшем случае

Ниже представлено сравнение работы алгоритмов в худшем случае на графике 4.2.

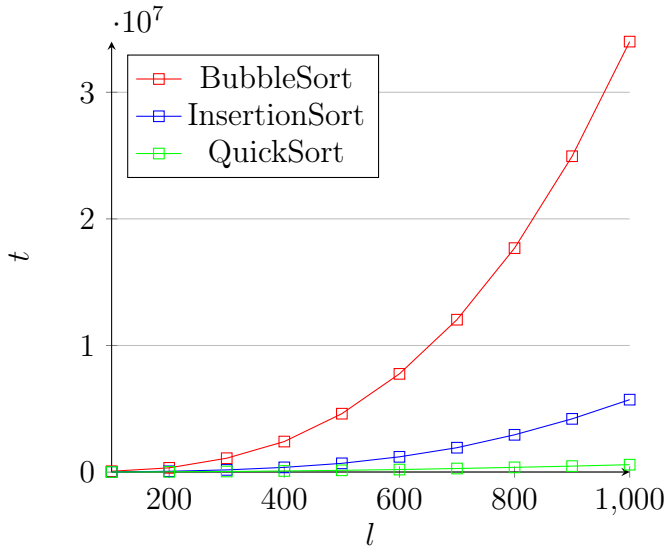

Рис. 4.2: Сравнение времени в худшем случае

Ниже представлено сравнение работы алгоритмов в худшем случае на графике 4.3.

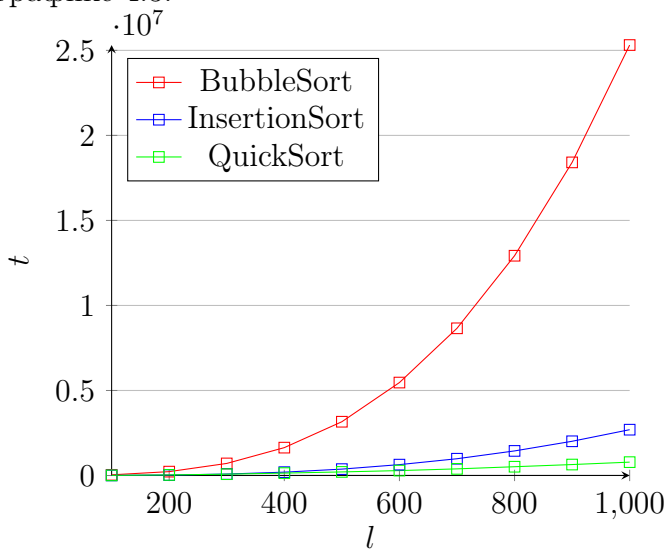

Рис. 4.3: Сравнение времени при случайном заполнении массива

4.3 Вывод

По результатам тестирования выявлено, что все рассматриваемые алгоритмы реализованы правильно. Самым быстрым алгоритмом, при использовании случайного заполнения, оказался алгоритм быстрой сортировки, а самым медленным — алгоритм сортировки пузырьком.

Заключение

В ходе работы были изучены алгоритмы сортировки массива: пузырьком с флагом, вставки, быстрая сортировка. Выполнено сравнение всех рассматриваемых алгоритмов. В ходе исследования был найден оптимальный алгоритм. Изучены зависимости выполнения алгоритмов от длины массива. Также реализован программный код продукта.

Список литературы

1. Кормен Т. Алгоритмы: построение и анализ [Текст] / Кормен Т. - Вильямс, 2014. - 198 с. - 219 с.

2. Руководство по языку Golang [Электронный ресурс], - режим доступа: https://golang/