

Государственное образовательное учреждение высшего профессионального
образования

«Московский государственный технический университет имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ
КАФЕДРА
технологии»

«Информатика и системы управления»
«Программное обеспечение ЭВМ и информационные

ОТЧЁТ ПО АНАЛИЗУ АЛГОРИТМОВ

к лабораторной работе №4 на тему:

Исследование сложности алгоритмов сортировки

Студент: Марабян К. ИУ7-55

Москва 2018

Содержание

Введение.....	2
1 Аналитическая часть.....	3
1.1 Постановка задачи.....	3
1.2 Описание алгоритмов.....	3
1.3 Область применения.....	5
2 Конструкторский раздел.....	6
2.1 Разработка алгоритмов.....	6
3 Технологический раздел.....	9
3.1 Минимальные требования к программному обеспечению.....	9
3.2 Средства реализации.....	10
3.3 Листинг кода.....	11
4 Экспериментальная часть.....	15
4.1 Расчет сложности алгоритмов.....	15
4.2 Постановка эксперимента.....	17
Заключение.....	21

Введение

Алгоритм сортировки — это алгоритм для упорядочивания элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

Целью данной работы является изучение алгоритмов сортировки по выбору.

Задачами данной лабораторной работы являются:

- 1) изучение трех алгоритмов сортировки по выбору;
- 2) получение практических навыков реализации выбранных алгоритмов;
- 3) сравнительный анализ алгоритмов (зависимость времени их выполнения от количества сортируемых элементов);
- 4) экспериментальное подтверждение различий во временной эффективности алгоритмов;
- 5) описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В данном разделе представлены теоретическое описание алгоритмов и указание области их применения.

1.1 Постановка задачи

Реализовать алгоритмы сортировки:

- а) вставками;
- б) слиянием;
- в) сортировкой Шелла.

Рассчитать сложность алгоритмов и провести временные эксперименты.

1.2 Описание алгоритмов

Алгоритм сортировки вставками.

1. Разделяем массив на две части: отсортированную и неотсортированную. На начальный момент времени отсортированная часть содержит первый элемент массива, неотсортированная – все остальные.
2. Берем первый элемент из неотсортированной части и вставляем его в правильную позицию в отсортированной части.
3. Повторяем второй шаг, пока размер неотсортированной части массива не станет равным нулю.

Алгоритм сортировки слиянием.

Сортировка слиянием подразумевает разбиение массива поровну до тех пор, пока из одного массива не получится несколько мелких — размером не более двух элементов. Два элемента легко сравнить между собой и упорядочить в зависимости от требования: по возрастанию или убыванию.

После разбиения следует обратное слияние, при котором в один момент времени (или за проход цикла) выбираются по одному элементу от каждого массива и сравниваются между собой. Наименьший (или наибольший) элемент отправляется в результирующий массив, оставшийся элемент остается актуальным для сравнения с элементом из другого массива на следующем шаге.

Алгоритм сортировки Шелла.

Является усовершенствованным вариантом сортировки вставками. Идея метода заключается в сравнение разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние равно d или $N/2$, где N — общее число элементов. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии $N/2$; они сравниваются между собой, и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние d сокращается на $d/2$, и количество групп, соответственно, уменьшается. Постепенно расстояние между элементами уменьшается, и на $d=1$ проход по массиву происходит в последний раз.

1.3 Область применения

Алгоритмы сортировки активно применяется:

- 1) во всех без исключения областях программирования, будь то базы данных или математические программы;
- 2) там, где речь идет об обработке и хранении больших объемов информации.

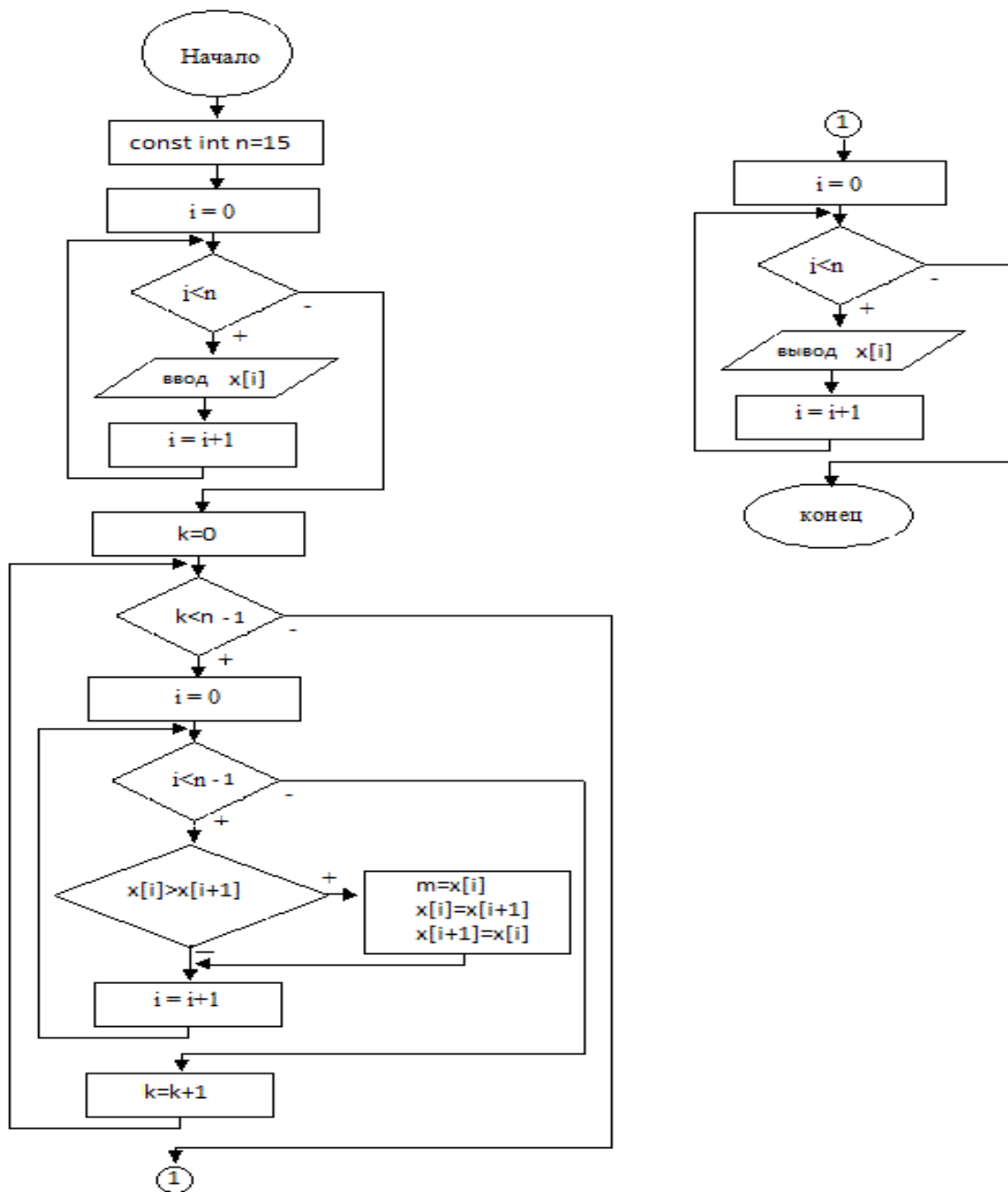
2 Конструкторский раздел

В этом разделе приведены схемы алгоритмов.

2.1 Разработка алгоритмов

Алгоритм сортировки Шелла:

Сортировка Шелла (*Shell sort*) — алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками.



```

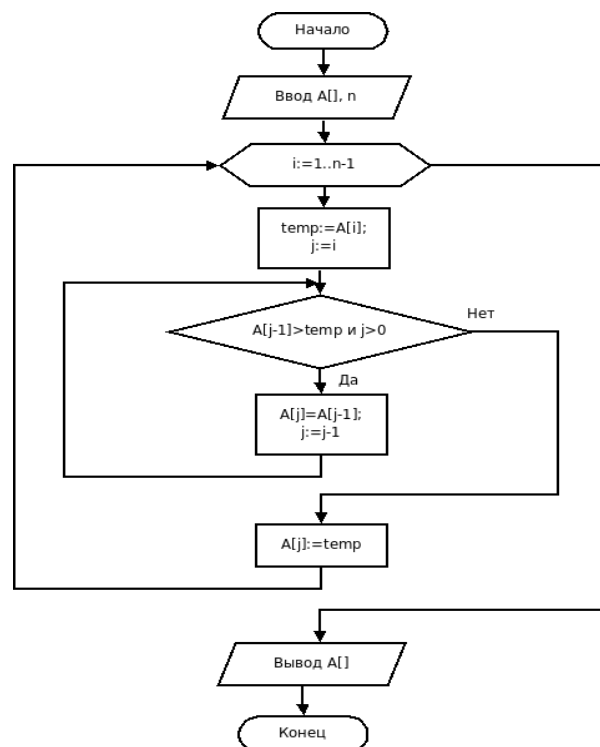
func insertionSort(_ list: inout [Int], start: Int, gap: Int) {
  for i in stride(from: (start + gap), to: list.count, by: gap) {
    let currentValue = list[i]
    var pos = i
    while pos >= gap && list[pos - gap] > currentValue {
      list[pos] = list[pos - gap]
      pos -= gap
    }
    list[pos] = currentValue
  }
}

func shellSort(_ list: inout [Int]) {
  var sublistCount = list.count / 2
  while sublistCount > 0 {
    for pos in 0..

```

Алгоритм сортировки вставками:

Здесь поступательно обрабатываются отрезки массива, начиная с первого элемента и затем расширяя подмассив, вставляем на своё место очередной неотсортированный элемент.



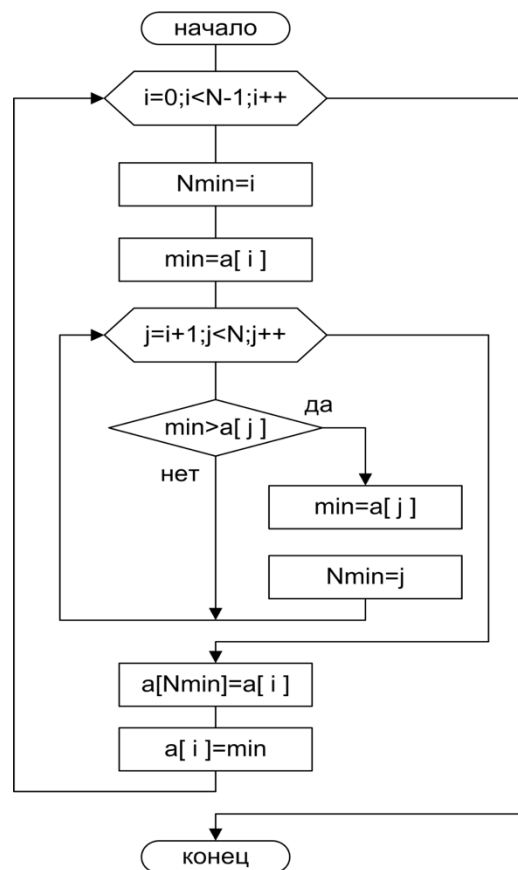

```

func insertionSort<T: Comparable>(_ array: [T]) -> [T] {
    var a = array
    for x in 1..

```

Алгоритм сортировки слиянием:

Сортировка слиянием подразумевает разбиение массива поровну до тех пор, пока из одного массива не получится несколько мелких — размером не более двух элементов. Два элемента легко сравнить между собой и упорядочить в зависимости от требования: по возрастанию или убыванию.



```

func mergeSort<T: Comparable>(_ array: [T]) -> [T] {
    guard array.count > 1 else { return array }
    let middleIndex = array.count / 2
    let leftArray = mergeSort(Array(array[0..

```

3 Технологический раздел

Данный раздел содержит указание использованного языка программирования и использованных средств замера процессорного времени, а также листинг кода.

3.1 Минимальные требования к программному обеспечению

Операционные системы: MacOS

Объем жесткого диска: 10 Мб.

Оперативная память: 1024 МВ ОЗУ.

3.2 Средства реализации

Swift – это новый язык программирования для разработки iOS и OS X приложений, который сочетает в себе все лучшее от C и Objective-C, но лишен ограничений, накладываемых в угоду совместимости с C. В Swift используются паттерны безопасного программирования и добавлены современные функции, превращающие создание приложения в простой, более гибкий и увлекательный процесс. Swift, созданный с чистого листа, – это возможность заново представить себе, как разрабатываются приложения.

3.3 Листинг кода

```
//
// main.swift
// Lab04Sortings
//
// Created by Корюн Марабян on 18/11/2018.
// Copyright © 2018 Корюн Марабян. All rights reserved.
//

import Foundation
import Cocoa

//////////Слиянием//////////
func mergeSort<T: Comparable>(_ array: [T]) -> [T] {
    guard array.count > 1 else { return array }
    let middleIndex = array.count / 2
    let leftArray = mergeSort(Array(array[0..
```

```

//////////////////Шелл//////////////////

func insertionSort(_ list: inout [Int], start: Int, gap: Int) {
    for i in stride(from: (start + gap), to: list.count, by: gap) {
        let currentValue = list[i]
        var pos = i
        while pos >= gap && list[pos - gap] > currentValue {
            list[pos] = list[pos - gap]
            pos -= gap
        }
        list[pos] = currentValue
    }
}

```

```

func shellSort(_ list: inout [Int]) {
    var sublistCount = list.count / 2
    while sublistCount > 0 {
        for pos in 0..

```

```

//////////////////

```

```

//////////////////Вставка//////////////////

func insertionSort<T: Comparable>(_ array: [T]) -> [T] {
    var a = array
    for x in 1..

```

```

//////////////////

```

```

//Массив, отсортированный по возрастанию
for i in stride(from: 100, to: 1100, by: 100){

```

```

    var firstArr = [Int] (1...i)

```

```

    var totalTime: Double = 0
    var totalTime2: Double = 0
    var totalTime3: Double = 0

```

```

    var avgTime:Double = 0
    var avgTime2:Double = 0
    var avgTime3:Double = 0
    for _ in 1...100{

```

```

let begin = clock()
shellSort(&firstArr)
let diff = Double(clock() - begin) / Double(CLOCKS_PER_SEC) * 1000

let begin2 = clock()
insertionSort(firstArr)
let diff2 = Double(clock() - begin2) / Double(CLOCKS_PER_SEC) * 1000

let begin3 = clock()
mergeSort(firstArr)
let diff3 = Double(clock() - begin3) / Double(CLOCKS_PER_SEC) * 1000

totalTime = totalTime + diff
totalTime2 = totalTime2 + diff2
totalTime3 = totalTime3 + diff3
}
avgTime = totalTime/100
avgTime2 = totalTime2/100
avgTime3 = totalTime3/100

//print("For \i elements avgTime(Shell) is \String(format: "%.3f", avgTime))")
//print("For \i elements avgTime(Insertion) is \String(format: "%.3f", avgTime2))" )
//print("For \i elements avgTime(Merge) is \String(format: "%.3f", avgTime3))\n" )
}

//Массив, отсортированный по убыванию
for i in stride(from: 100, to: 1100, by: 100){

    var secondArr = [Int] ((1...i).reversed())

    var totalTimeMerge: Double = 0
    var totalTimeShell: Double = 0
    var totalTimeInsertion: Double = 0

    var avgTimeMerge:Double = 0
    var avgTimeShell:Double = 0
    var avgTimeInsertion:Double = 0

    for _ in 1...100{
        let beginMerge = clock()
        shellSort(&secondArr)
        let diffMerge = Double(clock() - beginMerge) / Double(CLOCKS_PER_SEC) * 1000

        let beginShell = clock()
        insertionSort(secondArr)
        let diffShell = Double(clock() - beginShell) / Double(CLOCKS_PER_SEC) * 1000

        let beginInsertion = clock()
        mergeSort(secondArr)
        let diffInsertion = Double(clock() - beginInsertion) / Double(CLOCKS_PER_SEC) * 1000

        totalTimeMerge = totalTimeMerge + diffMerge
        totalTimeShell = totalTimeShell + diffShell
    }
}

```

```

        totalTimeInsertion = totalTimeInsertion + diffInsertion
    }
    avgTimeMerge = totalTimeMerge/100
    avgTimeShell = totalTimeShell/100
    avgTimeInsertion = totalTimeInsertion/100

    //print("For \i elements avgTime(Merge) is \String(format: "%.3f", avgTimeMerge))"
    //print("For \i elements avgTime(Shell) is \String(format: "%.3f", avgTimeShell)" )
    //print("For \i elements avgTime(Insertion) is \String(format: "%.3f", avgTimeInsertion))\n" )
}

//Массив, заполненный рандомными целыми числами

for i in stride(from: 100, to: 1100, by: 100){
    var thirdArr = [Int] (1...i)
    thirdArr.append(Int.random(in: 1...100))

    var totalTimeMerge2: Double = 0
    var totalTimeShell2: Double = 0
    var totalTimeInsertion2: Double = 0

    var avgTimeMerge2:Double = 0
    var avgTimeShell2:Double = 0
    var avgTimeInsertion2:Double = 0

    for _ in 1...100{
        let beginMerge2 = clock()
        shellSort(&thirdArr)
        let diffMerge2 = Double(clock() - beginMerge2) / Double(CLOCKS_PER_SEC) * 1000

        let beginShell2 = clock()
        insertionSort(thirdArr)
        let diffShell2 = Double(clock() - beginShell2) / Double(CLOCKS_PER_SEC) * 1000

        let beginInsertion2 = clock()
        mergeSort(thirdArr)
        let diffInsertion2 = Double(clock() - beginInsertion2) / Double(CLOCKS_PER_SEC) * 1000

        totalTimeMerge2 = totalTimeMerge2 + diffMerge2
        totalTimeShell2 = totalTimeShell2 + diffShell2
        totalTimeInsertion2 = totalTimeInsertion2 + diffInsertion2
    }
    avgTimeMerge2 = totalTimeMerge2/100
    avgTimeShell2 = totalTimeShell2/100
    avgTimeInsertion2 = totalTimeInsertion2/100

    //print("For \i elements avgTime(Merge) is \String(format: "%.3f", avgTimeMerge2))"
    //print("For \i elements avgTime(Shell) is \String(format: "%.3f", avgTimeShell2)" )
    //print("For \i elements avgTime(Insertion) is \String(format: "%.3f", avgTimeInsertion2))\n" )
}

```

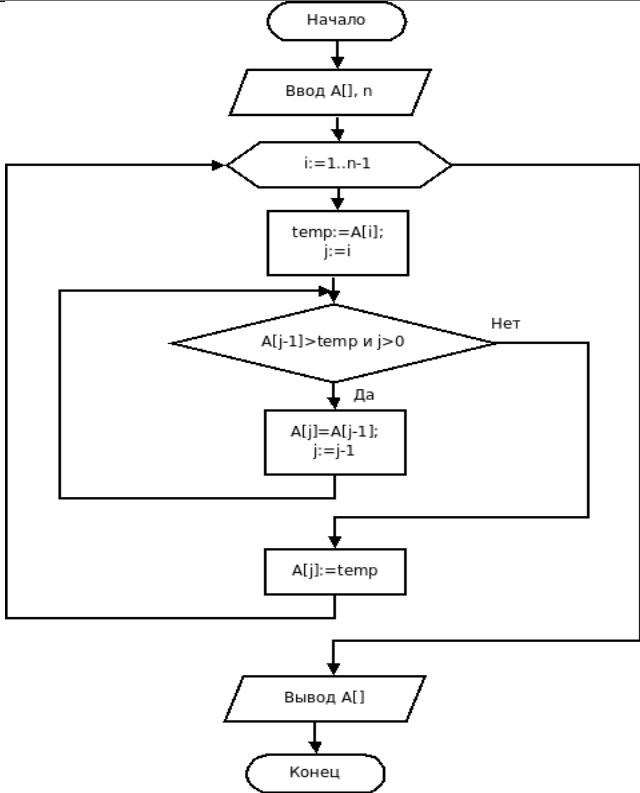
4 Экспериментальная часть

В данном разделе представлены примеры работы для нескольких случаев с указанием матричных значений для нерекурсивных реализаций и результирующих длин для всех реализаций

4.1 Расчет сложности алгоритмов

Таблица 1

Расчет сложности для сортировки вставками (M – размер массива)

Участок кода	Лучший случай	Худший случай
 <pre> graph TD Start([Начало]) --> Input[/Ввод A[], n/] Input --> LoopStart{i:=1..n-1} LoopStart --> AssignTemp[temp:=A[i]; j:=i] AssignTemp --> Decision{A[j-1]>temp и j>0} Decision -- Да --> Shift[A[j]=A[j-1]; j:=j-1] Shift --> Decision Decision -- Нет --> AssignA[j]:=temp[A[j]:=temp] AssignA[j]:=temp --> LoopStart AssignA[j]:=temp --> Output[/Вывод A[]/] Output --> End([Конец]) </pre>	"=" + M * "<" + M * "++"	"=" + M * "<" + M * "++"
	M * "=" + M * "[]"	M * "=" + M * "[]"
	M * "="	M * "="
	M * "=" + M * "<" + M * " -- "	M * "=" + M ² / 2 * "<" + M ² / 2 * "--"
	M * ">=" + M * "[]" + M * "-"	M ² /2 * ">=" + M ² /2 * "[]" + M ² /2 * "-"
	-----	3M * "+" + M * "- " + M * "*" + M * "memmove"
	-----	M * "[]" + M * "="
	-----	M * "="

	$M * "="$	$M * "="$
	$M * "+" + M *$ $"*" + M *$ $"memmove" +$ $M * "[" + M *$ $"_"$	<p>-----</p> <p>-</p>
Итог	Сравнений: $3M$ Сложений: $4M$ Умножений: M Присвоений: $3M + 1$ Обращений по индексу: $3M$ Memmove: M	Сравнений: $M^2 + 2M$ Сложений: $M^2 + 5M$ Умножений: M Присвоений: $5M + 1$ Обращений по индексу: $M^2/2 + 2M$ Memmove: M

Сортировка вставками:

Лучший случай: $O(M)$

Худший случай: $O(M^2)$

Сортировка расческой:

Лучший случай: $O(M \log M)$

Худший случай: $O(M^2)$

Сортировка с помощью двоичного дерева:

Лучший случай: $O(\log M)$

Худший случай: $O(M^2)$

4.2 Постановка эксперимента

Алгоритмы будут протестированы по скорости работы.

Для замера времени выполнения (скорости работы) были выбраны:

1. Массив целых чисел, отсортированных по возрастанию.
2. Массив целых чисел, отсортированных в обратном порядке.
3. Массив случайных целых чисел.

Замеры проводились 100 раз для каждой сортировки с усреднением результата.

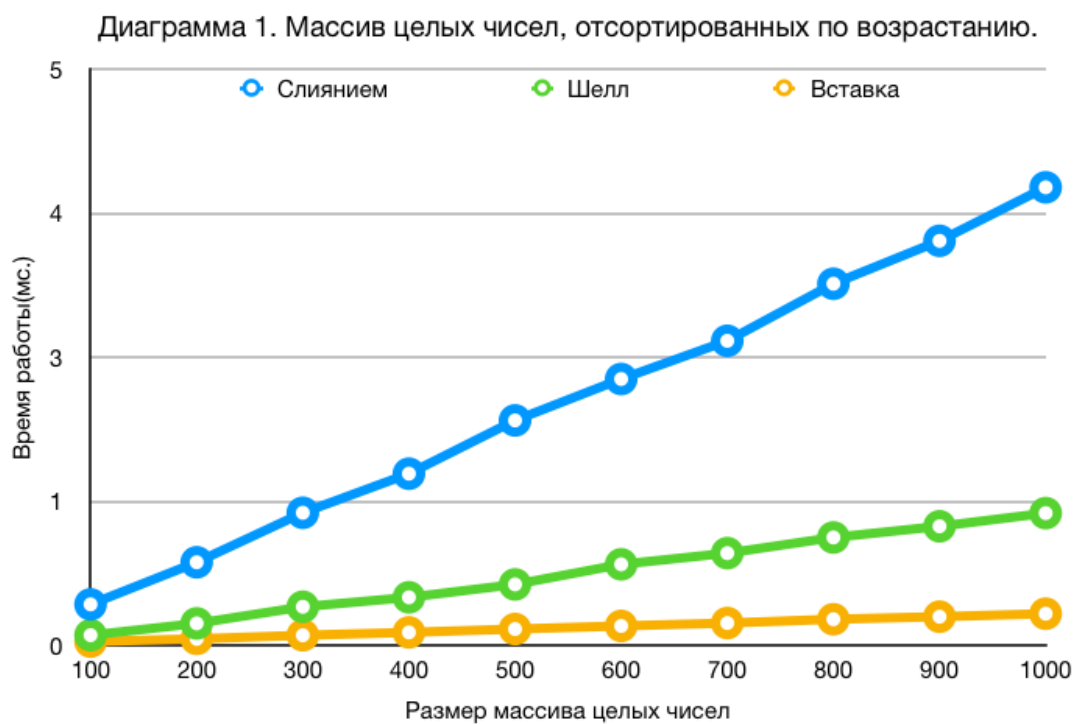


Рис. 1 - Графики временных замеров для сортированного массива.



Рис. 2 - Графики временных замеров для отсортированного в обратном порядке массива.

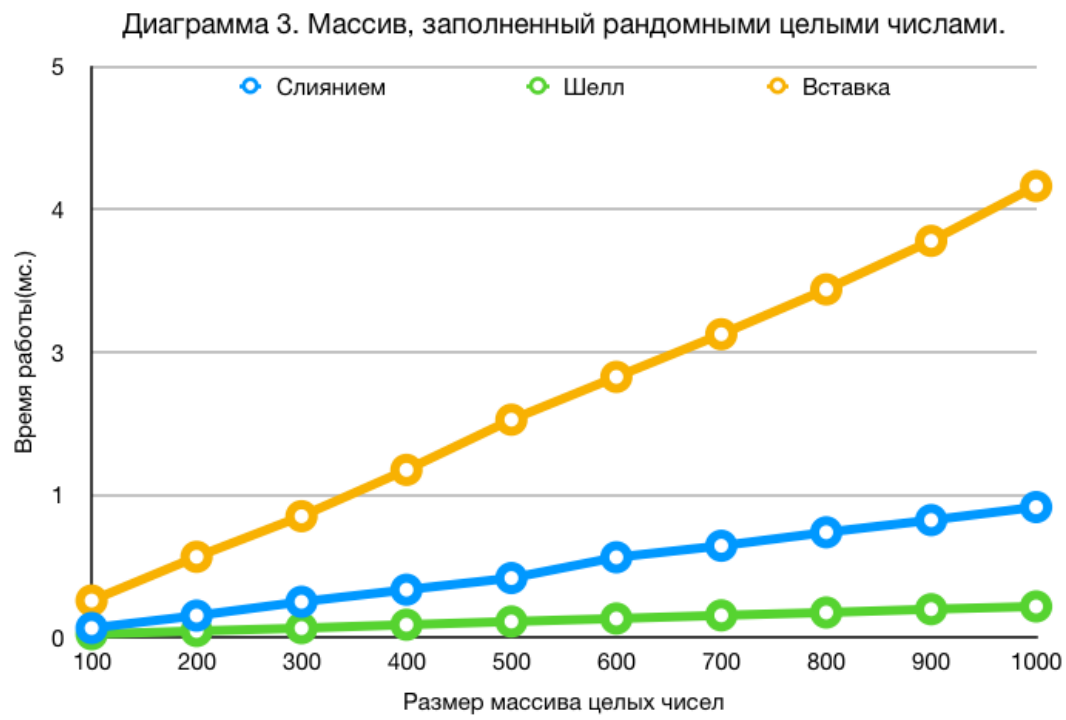


Рис. 3 - Графики временных замеров для массива, заполненного случайными числами.

Вывод

В результате проведенного исследования и полученных данных, для сортировки неотсортированного массива, наиболее оптимальным из представленных алгоритмов для сортировки массива является сортировки Шелла. При массивах, заполненных случайно и по убыванию, алгоритмы показывают приблизительно одинаковое время.

Заключение

Изучены три алгоритма сортировки по выбору. Получены практические навыки реализации выбранных алгоритмов. Был сделан сравнительный анализ алгоритмов (зависимость времени их выполнения от количества сортируемых элементов). Проведено экспериментальное подтверждение различий во временной эффективности алгоритмов. Сделано описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.