**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

**Отчёт**

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных».**

**Тема: Алгоритмы сортировки сравнением**.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3354 |  | Чикарёва М.Д. |
| Преподаватель |  | Пестерев Д. О. |

**Санкт-Петербург**

**202****4**

**Цель лабораторной работы**: реализация алгоритмов сортировки сравнением и исследование их временной сложности.

**Представленные в работе алгоритмы сортировки сравнением**: сортировка выбором, сортировка вставками, сортировка пузырьком, сортировка слиянием, сортировка Шелла (последовательность Шелла, Хиббарда, Пратта), быстрая сортировка, пирамидальная сортировка.

# Теоретическая часть.

**Сортировка выбором (selection sort).**

1. Асимптотическая временная и пространственная сложность

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| Θ(n2) | Θ(n2) | Θ(n2) | Θ(1) |

Алгоритм сортировки выбором имеет временную сложность θ(n2) и пространственную сложность θ(1), поскольку он не требует никакого дополнительного места в памяти, кроме временной переменной, используемой для перестановки.

1. Описание алгоритма

Есть отсортированная и неотсортированная части массива. Мы ищем минимальный элемент в неотсортированной части и добавляем его в конец отсортированной, после этого длина остортированной части увеличивается на единицу, а неотсортированной - уменьшается на 1. Цикл повторяется до тех пор, пока все элементы не займут нужную последовательность.

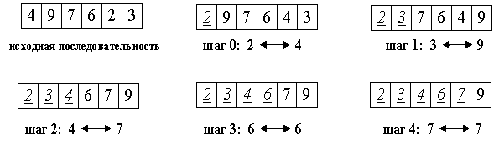


Рис.1

1. Анализ устойчивости алгоритма

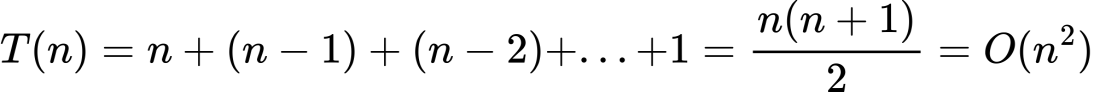
Сортировка, в результате которой относительная последовательность элементов не изменилась, называется устойчивой. При неустойчивой сортировке элементы в массиве меняются местами.

Сортировка выбором является неустойчивой и следующий пример это докажет: есть массив из элементов, каждый из которых имеет два поля, сортировка идет по первому полю.  
 Массив до сортировки: { (2, a), (2, b), (1, a) }. Уже после первой итерации внешнего цикла отсортированная последовательность будет следующей: { (1, a), (2, b), (2, a) }. Можно заметить, что взаимное расположение элементов (2, a) и (2, b) изменилось, таким образом, рассматриваемая реализация является неустойчивой.

1. Функции временной сложности для:

Пусть список содержит n элементов. Сначала нужно найти минимум среди n элементов списка, что потребует n операций. Потом нужно найти наименьший из n-1 элемента, на это нужно n-1 операция. Потом нужно n-2 операции и т. д. Тело внутреннего цикла выполняется за O(1), т.е. не зависит от размера сортируемого массива.

Таким образом, общее число операций равно:



Следовательно, сортировка выбором — квадратичный алгоритм, время его работы пропорционально квадрату от размера списка.

На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае O(n2), предполагая что сравнения делаются за постоянное время.

1. Асимптотическая оценка временной сложности:

- Наилучший случай возникает, когда массив уже отсортирован (где n - кол-во целых чисел в массиве) - O(n2);

- Средний случай возникает, когда элементы массива расположены в неупорядоченном или случайном порядке, без чёткого возрастания или убывания - O(n2);

- Худший случай возникает, когда нужно отсортировать массив в порядке возрастания, но изначально массив находится в порядке убывания - O(n2).

Временная сложность во всех случаях остаётся O(n2), так как на каждом шаге алгоритм определяет минимальный элемент и помещает его на правильное место, но этот минимальный элемент не может быть найден до тех пор, пока не будет пройден весь массив.

1. Функция пространственной сложности и её асимптотическая оценка.

При данном алгоритме память выделяется только для счётчиков цикла и переменной-буфера, позволяющей обменивать 2 элемента массива местами, поэтому S(n) = n+1 -> S’(n) = 1 = O(1).

1. График функции временной сложности для всех случаев будет одинаковым:

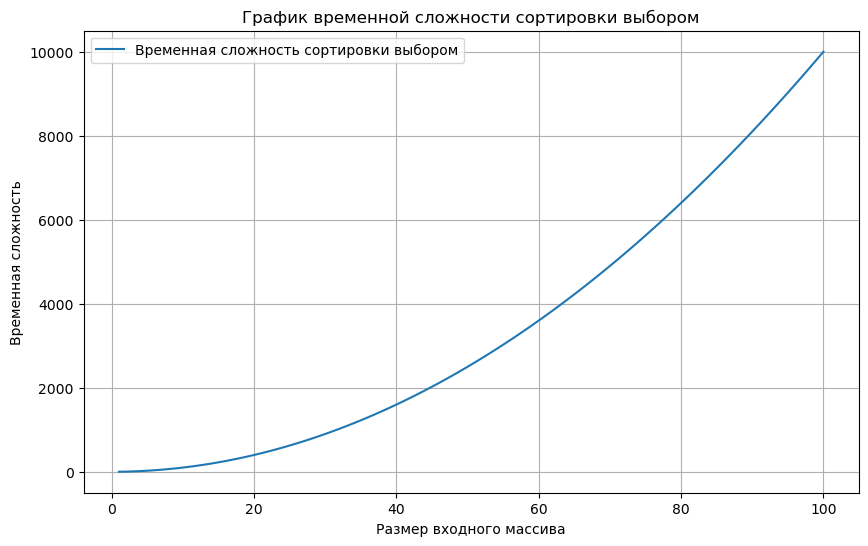
****

Рис.2

**Сортировка вставками (insertion sort).**

1. Асимптотическая временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| Θ(n) | Θ(n2) | Θ(n2) | Θ(1) |

1. Описание алгоритма.

Массив делится на 2 части - остортированную и неотсортированную. В неотсортированной части массива перебираем элементы, каждый извлекаемый элемент вставляется в отсортированную часть массива на то место, где он должен находиться, в результате чего отсортированная часть массива увеличивается, а неотсортированная уменьшается.

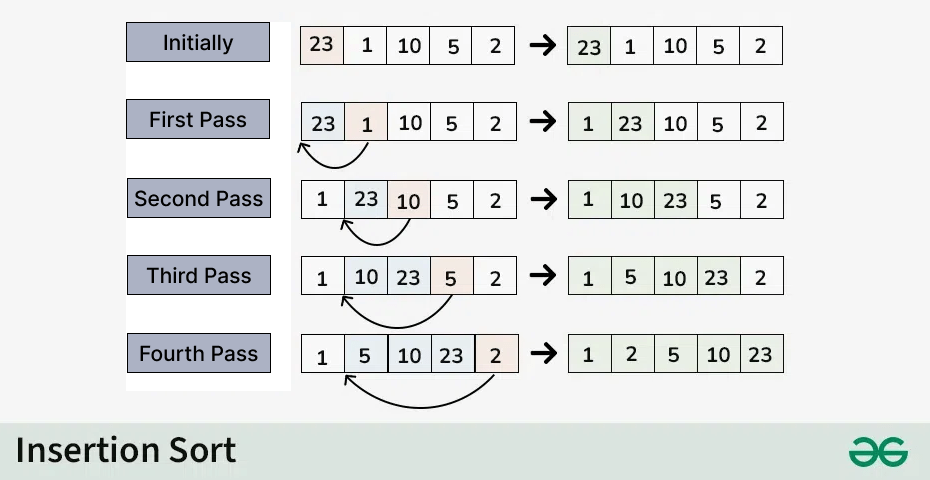


Рис.3

1. Анализ устойчивости алгоритма.

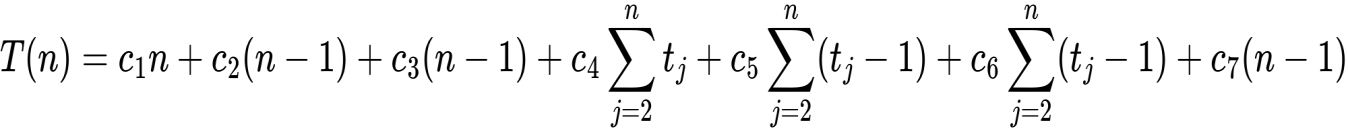
Алгоритм устойчивый, потому что в результате сортировки относительная последовательность элементов не изменяется.

1. Функции временной сложности.

Время сортировки вставками зависит от размера сортируемого массива: чем больше массив, тем больше может потребоваться времени. Также важен порядок элементов, так как если массив почти упорядочен, то времени потребуется меньше.

|  |  |  |
| --- | --- | --- |
| Код | Стоимость | Повторы |
| for j = 2 to A.length | c1 | n |
| key = A[j] | c2 | n-1 |
| i=j-1 | c3 | n-1 |
| while i > 0 and a[i]>key | c4 | wps |
| A[i+1]= A[i] | c5 | wps |
| i=i-1 | c6 | wps |
| A[i+1]=key | c7 | n-1 |

Время работы алгоритма сортировки вставками - сумма времён работы каждого шага:



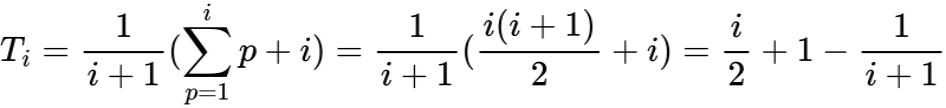
- Самым лучшим случаем является отсортированный массив, тогда все внутренние циклы состоят всего из одной итерации -> tj=1 для всех j.

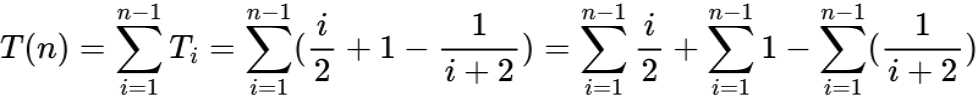
C:/Users/user/AppData/Local/Temp/wps.LGEDHWwps

Время работы зависит линейно от размера поступающих данных.

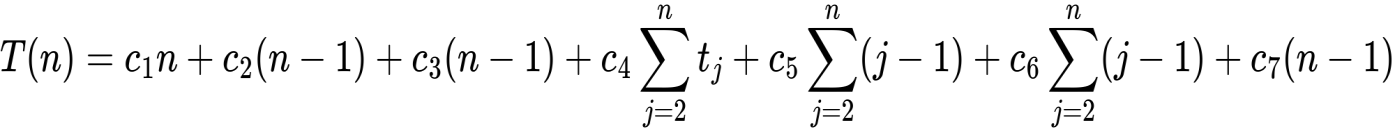
- Для получения функции сложности среднего случая нужно посчитать среднее число сравнений, требующихся для определения места в массиве очередного элемента.

Даже если добавленный новый элемент оказался в правильной позиции, всё равно нужно как минимум одно сравнение. i-ый добавляемый элемент модет заниматься одно из i+1 мест. При случайных входных данных новый элемент может оказаться на любой позиции, поэтому среднее число сравнений для вставки i-го элемента:





- Худший случай - массив, отсортированный в обратном нужному порядке, каждый новый элемент сравнивается со всеми в отсортированной последовательности, что значит, что все внутренние циклы состоят из j итеарций -> tj=j для всех j.



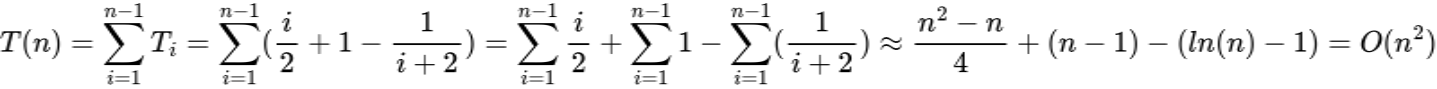
1. Асимптотическая сложность временной функции

Лучший случай имеет линейную сложность, оставшиеся - квадратичную.

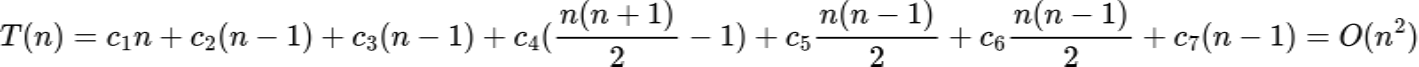
- Лучший случай

wps

-Средний случай.



- Худший случай



1. Функция пространственной сложности и её асимптотическая оценка.

Пространственная сложность сортировки вставками составляет O(1), она использует постоянное дополнительное пространство памяти независимо от размера входных данных.

Это связано с тем, что алгоритм выполняется сортирвоку на месте, то есть переставляет элементы внутри массива, не требуя дополнительных структур данных или выделения памяти.

1. Графики функций временной сложности.

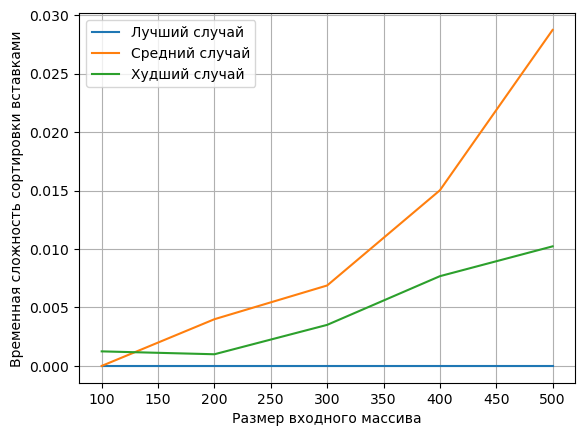
****

Рис.4

**Сортировка пузырьком (bubble sort)**

1. Асимптотическая временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(n) | O(n2) | O(n2) | O(1) |

Асимптотическая сложность алгоритма O(n2), пространственная сложность O(1).

1. Описание алгоритма.

Происходит сравнение рядом стоящих элементов: если элемент слева больше того, что справа, то они меняются местами и так далее до конца массива. Таким образом, после первого прохождения наибольшее число будет в конце. Отсюда и идёт название «пузырьком», наибольший элемент как бы всплыл наверх, как пузырёк в воде.

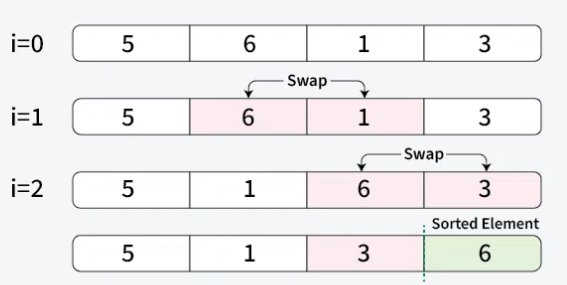


Рис.5

1. Анализ устойчивости алгоритма.

Сортировка пузырьком устойчива, так как в результате её выполнения относительная последовательность элементов не меняется. При выполнении алгоритма наибольший элемент «всплывает» наверх, а остальные остаются на своих местах относительно друг друга, смещаясь на единицу.

1. Функции временной сложности.

A[n]

For I = 0 to n-1

For j=0 to n-2

If a[j+1] < a[j]

Swap(a[j], a[j+1])

- В лучшем случае массив уже отсортирован, поэтому количество необходимых сравнений равно n-1, а количество необходимых перестановок = 0 -> T(n) = n-1.

-Средний случай. В сортировке пузырьком количество сравнений постоянно, это связано с тем, что независимо от расположения элементов количество сравнений C(n) одинаково. Однако если элемент находится в индексе i1, а должен быть в i2, то потребуется минимум i2-i1 переставновок, чтобы переместить элемент в нужное место.

Теперь предположим, что элемент находится на рассмотянии i3 от свого места в отсортированном массиве, тогда максимально значение i3 = (n-1) для крайних элементов и n/2 для тех, что в середине. Тогда

T(n) = (n-1)+(n-3)+(n-5)…+0+…+(n-3)+(n-1) = n × (n-2) × (1+3+5+…+n/2) = n2 - n2/2 = n2/2

-Худший случай - массив отсортирован в обратном порядке. Алгоритм (n-1) раз совершит (n-2), поэтому T(n) = (n-1)(n-2) = n2-2n-n+2 = n2 - 3n + 2 -> квадратичная функция.

1. Асимптотическая сложность временной функции.

- Лучший случай: T(n) = n-1 = O(n);

- Средний случай: T(n) = (n-1)+(n-3)+(n-5)…+0+…+(n-3)+(n-1) = n × (n-2) × (1+3+5+…+n/2) = n2 - n2/2 = n2/2 = O(n2);

- Худший случай: T(n) = (n-1)(n-2) = O(n2).

1. Функция пространственной сложности и её асимптотическая оценка.

Пространственная сложность пузырьковой сортировки равна O(1), так как объём дополнительного пространства (памяти), требуемый алгоритмом, остаётся постоянным независимо от размера сортируемого данного массива. Для сортировки пузырьком постоянный объём памяти нужен только для хранения временных переменных или индексов в процессе сортировки, поэтому пространственная сложность данной сортировки считается эффективной, ведь она не зависит от размера входных данных и не требует дополнительного пространства, пропроционального размеру этих входных данных.

1. Графики функций временной сложности.

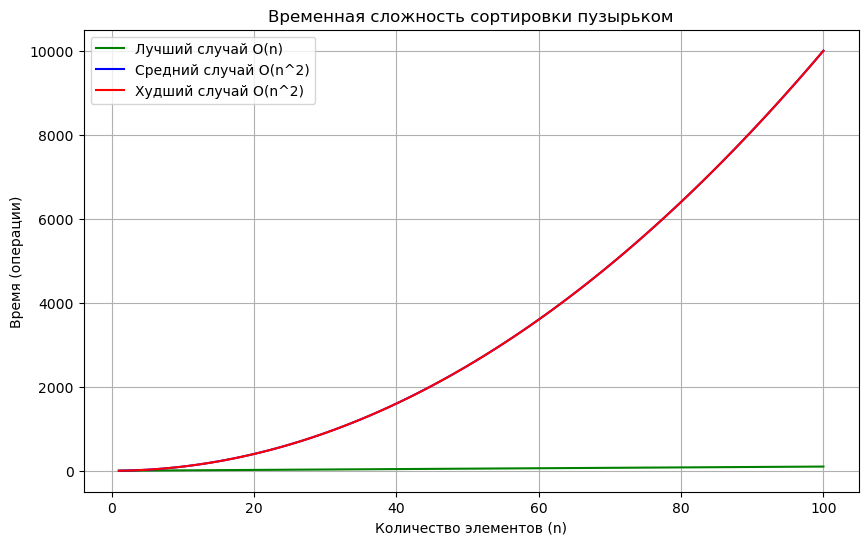
****

Рис.6

**Сортировка слиянием (mergesort)**

1. Асимптотическую временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(nlogn) | O(nlogn) | O(nlogn) | O(n) |

1. Алгоритм сортировки.

Алгоритм основан на подходе «разделяй и властвуй». Массив или список рекурсивно делится на две половины до тех пор, пока это возможно. Затем каждый подмассив сортируется по отдельности с помощью алгоритма сортировки слиянием и уже отсортированные подмассивы объединяются в отсортированном порядке. Данный процесс продолжается до тех пор, пока не будут объединены все элементы из обоих подмассивов.

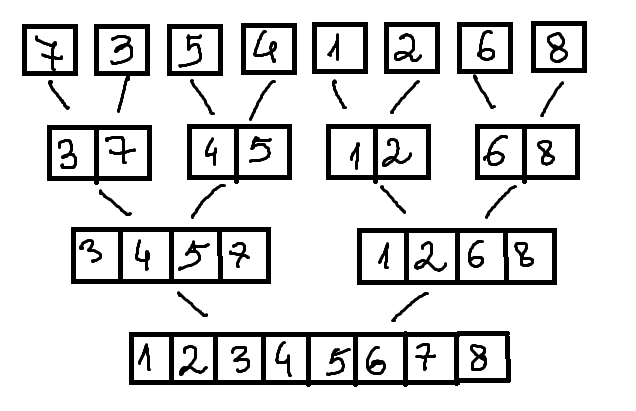


Рис.7

1. Анализ устойчивости алгоритма сортировки.

Данная сортировка устойчивая, она сохраняет порядок равных элементов (принадлежащих одному классу эквивалентности по сравнению).

Если ai==aj и i < j в исходном массиве, то g(i) < g(j) (g(i) - индекс 1ого элемента в отсортрованном массиве).

1. Функция временной сложности.

# C = output [length = N]

# A 1st sorted half [N/2]

# B 2nd sorted half [N/2]

i = j = 1

for k = 1 to n

if A[i] < B[j]

C[k] = A[i]

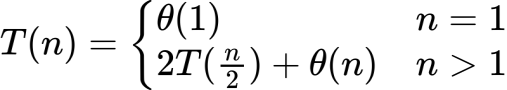
i++

else

C[k] = B[j]

j++

Рекурентное соотношение сортировки слиянием:



T(n) - общее время, которое было потрачено на сортировку массива размера n;

2T(n/2) - время, которое тратится на рекурсивную сортировку двух половин массива. Так как в каждой половине n/2 элементов, то мы имее два рекурсивных выхова с размером данных (n/2);

O(n) представляет время, затраченное на объединение двух отсортированных частей.

В лучшем случае, когда массив уже отсортирован или почти отсортирован, в среднем случае, когда массив упорядочен случайно и в худшем случае, когда массив отсортирован в обратном порядке, сложность останется одинаковой, в чём есть некоторое преимущество данной сортировки, ведь она будет работать с такой же скорость при большом количестве данных. Всё это благодаря рекурсивному делению массива и упорядочиванию уже отсортированных подмассивов.

1. Асимптотическая оценка временной сложности.

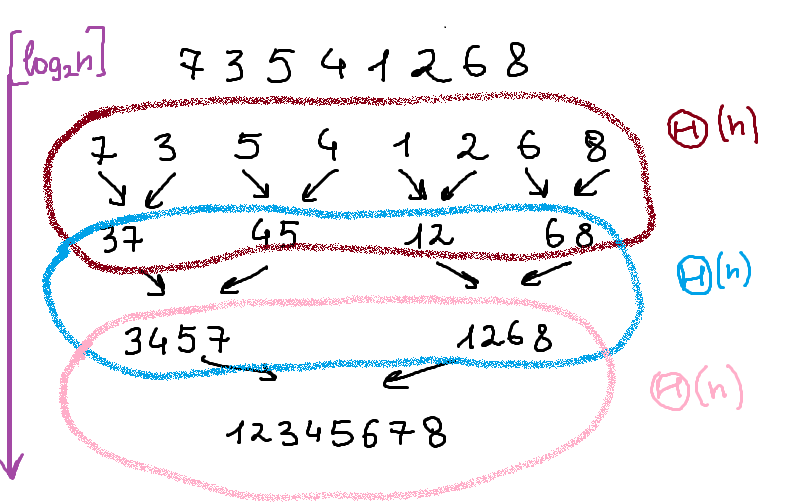


Рис.8

Для всех случаев сложность будет одинаковая. В худшем или среднем случае сортировка слиянием просто делить массив на две половины на каждом этапе, что даёт коэффициент logn, а другой зависит от сравнений, которые выполняются на каждом этапе. Для наихудшего случая на каждом этапе выполняется n сравнений для n входных даннных. На «почти остортированных» массивах сортировка работает так же долго, как и на хаотичных.

Итак, на каждом уровне дерева мы тратим θ(n) времени, количество уровней для массива длиной n - log2n. Тогда:

T(n) = 2T(n/2) + θ(n) = log2n \* θ(n) = θ(nlogn)

1. Функция пространственной сложности и её асимптотическая оценка.

Пространственная сложность Merge Sort составляет O(n), потому что необходима дополнительная память для хранения временных массивов, используемых на каждом этапе рекурсии.

1. Графики функции временной сложности для лучшего, худшего и среднего случая.

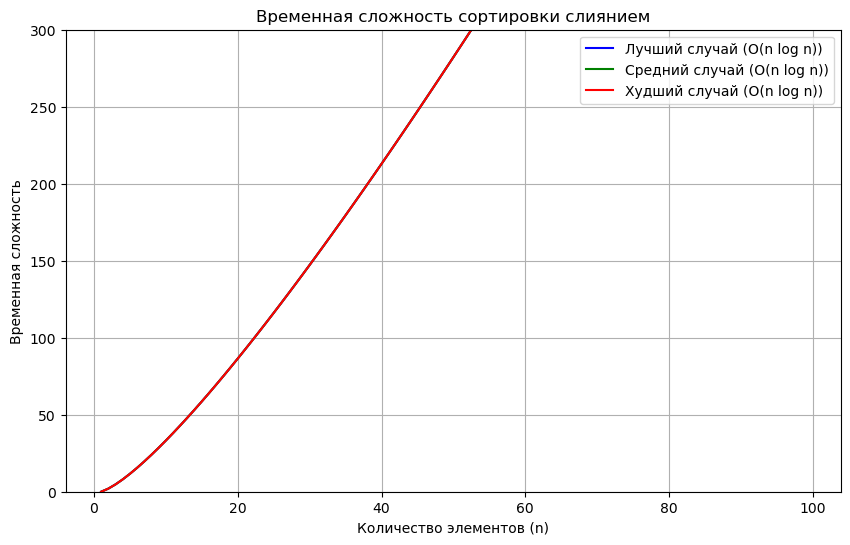


Рис.9

**Сортировка Шелла (shell sort)**

1. Асимптотическая временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(nlog2n) | O(nlogn) | O(n2) | O(n)+O(1) |

Асимптотическая сложность алгоритма O(n2), пространственная сложность O(n) - всего и O(1) дополнительно.

1. Описание алгоритма.

Данный алгоритм явялется усовершенствованным алгоритмом сортировки вставками.

Здесь используется свойство, что каждая подпоследовательность отсортированного массива также является отсортированной. Сортировка Шелла рассматривает подпоследовательности, в которых между элементами есть постоянный разрыв. Например:

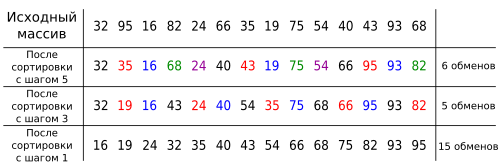


Рис.10

На каждом проходе Shellsort сортирует отдельные цепочки с помощью сортировки вставками, уменьшает разрыв и сортирует новые цепочки на следующем проходе. При сортироке цепочек с большими разрывами элементы могут «перепрыгивать» через большие участки массива, поэтому потребуется меньше работы. Помимо этого, по мере уменьшения разрывов цепочки становятся больше и включают в себя больше индексов, а значит элементы будут располагаться ближе к своим правильным позициям.

1. Анализ устойчивости алгоритма сортировки.

Данная сортировка устойчивая, она сохраняет порядок равных элементов (принадлежащих одному классу эквивалентности по сравнению).

Если ai==aj и i < j в исходном массиве, то g(i) < g(j) (g(i) - индекс 1ого элемента в отсортрованном массиве).

1. Функция временной сложности.

Среднее время работы алгоритма зависит от длин промежутков - d, на которых находятся сортируемые элементы исходного массива ёмкостью n на каждом шаге алгоритма.

Есть несколько последовательностей длин промежутков:

- Худший случай происходит при последовательности длин промежутков **Шелла**: d1=n/2, di=di-1/2, dk=1. На первом проходе разрыв равен n/2, на втором проходе он равен n/4, на третьем - n/8 и так продолжается до тех пор, пока на последнем проходе разрыв будет 1.

Пусть входные данные x содержат целые числа от 1 до n. Худший случай возникнет тогда, когда n будет являться степенью двойки, 1, 2, 3...,n/2 занимают нечётные позиции, а n/2+1, n/2+2,…,n-чётные. В таком случае подмассивы [x1, x3, … , xn-1] и [x2, x4, … , xn] сортируются независимо. После предпоследнего прохода два подмассива чередуются:

1, n/2+1, 2, n/2+2, 3, n/2+3 … n/2-1, n-1, n/2, n.

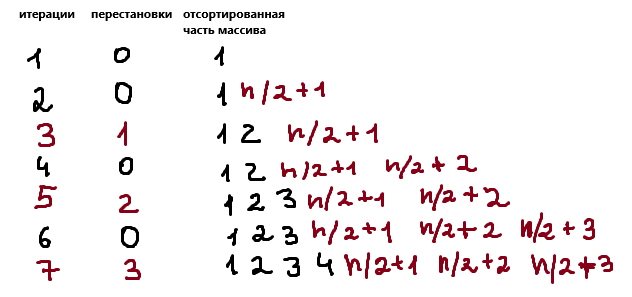
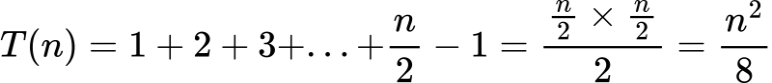


Рис.11

 .

Независимо от общего количества сравнений и количества перестановок во всех проходах, кроме пооследнего, сортировка Шелла при использовании этой последовательности разрывов в худшем случае имеет квадратичную временную сложность.

- Средний случай. Вторая последовательность, которая была предложена - последовательность Хиббарда. Общий термин: 2k-1 для k = log2n, log2n-1, …, 2, 1. (или 2i-1 ≤ N).

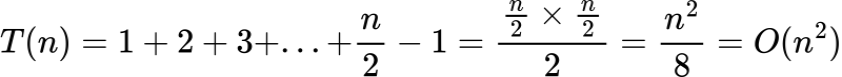
Например, если n=14, то последовательность разрывов будет 7, 3, 1. В худшем случае временная сложность будет равна O(n1.5).

- Лучший случай времени происходит при последовательности Пратта. Общий вид: 2i × 3j ≤ N/2, I, j - целые числа.

Так как в последовательности много пропусков, shellsort выполняет много проходов,что делает его медленным на практике.

1. Асимптотическая оценка временной сложности.

- Худший случай:



- Средний случай: T(n) = O(n1,5)≈O(nlogn)

- Последовательность Пратта обеспечивается наилучшее аисмптотическое поведение: T(n) = O(nlog2n)

1. Функция пространственной сложности и её асимптотическая оценка.

Во время работы аалгоритма требуется лишь небольшое количество дополнительной памяти для хранения временной переменной, используемой для обмена элементов, поэтому пространственная сложность сортировки Шелла составляет O(1). Сортировка Шелла явялется “in-place” сортировкой, поскольку она не требует дополнительного массива или структуры данных, размеры которых пропорциональны количеству сортируемых элементов.

1. Графики функции временной сложности для лучшего, худшего и среднего случая.

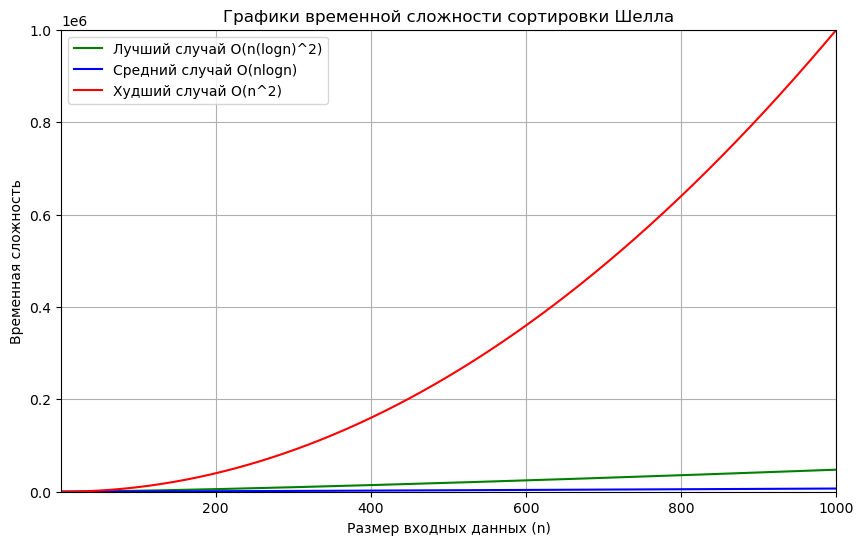


Рис.12

**Быстрая сортировка (quicksort)**

1. Асимптотическая временная и пространственная сложность.

|  |  |  |
| --- | --- | --- |
|  | T | S’ |
| Best | O(nlogn) | O(logn) |
| Average | O(nlogn) | O(logn) |
| Worst | O(n2) | O(n) |

1. Описание алгоритма.

Алгоритм быстрой сортировки является рекурсивным, поэтому процедура на вход будет принимать границы участка массива от 0 до n.

В основе алгоритма лежит процедура partition. Она выбирает некоторый элемент массива и переставляет элементы участка массива таким образом, чтобы массив разбился на 2 части: левая часть содержит элементы, которые меньше выбранного элемента, а правая - меньше. Подобный разделяющий элемент называется пивотом (pivot):

*partition(l, r):*

*pivot = a[random(l ... r - 1)]*

*m = l*

*for i = l ... r - 1:*

*if a[i] < pivot:*

*swap(a[i], a[m])*

*m++*

*return m*

Пивот в нашем случае выбирается случайным образом, поэтому такой алгоритм называется рандомизированным. Сложность процедуры partition - O(n), где n = r-l-длина участка.

Выбор подобного элемента на рисунке ниже был сделан как «последнего» в массиве и от него и начинаем сортировку.

Крайний случай - массив из одного элемента является упорядоченным, если же массив длинный, то вновь применяем partition и вызываем данную процедуру рекурсивно для двух половин массива.

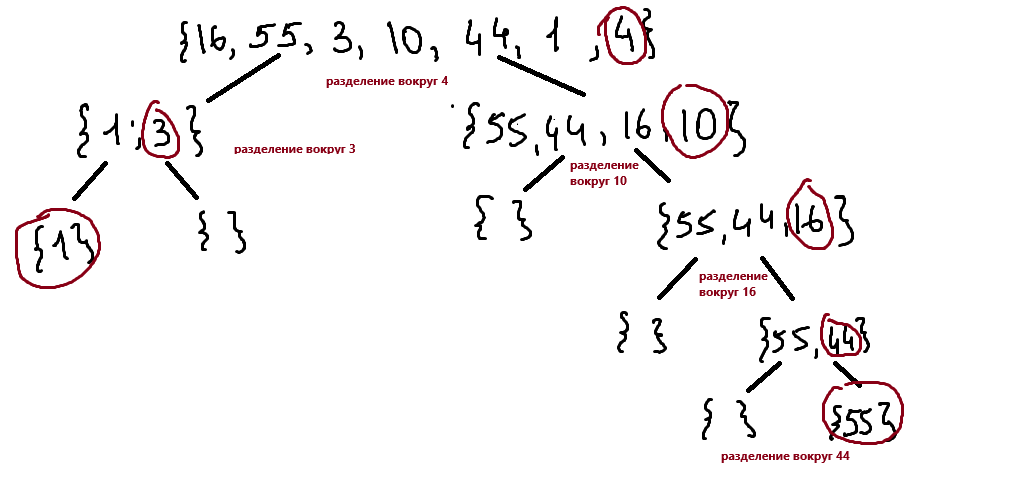


Рис. 13

1. Анализ устойчивости.

Быстрая сортировка — неустойчивый алгоритм, то есть не сохраняет порядок одинаковых элементов. Это связано с тем, что элементы меняются местами в соответствии с положением точки поворота без учёта их исходного положения.

Её можно сделать стабильной, но для этого потребуется либо дополнительное пространство, либо дополнительная логика. Например, можно составить два отдельных списка: первый — элементы меньшего размера, чем сводный, второй — элементы, размер которых превышает сводный.

Также устойчивость сортировки можно достичь путём удлинения исходных ключей, если включить в них информацию об первоначальном порядке значений или учитывать положение элементов изначально, но для этого потребуется дополнительное пространство.

1. Функции временной сложности

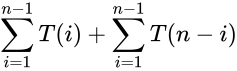
T(n) = n + T(a\*n) + T((1-a)\*n). Таким образом, при вызове сортировки массива из n элементов тратится порядка n операций на выполнение разделения и на выполнения себя же 2 раза с параметрами a\*n и (1-a)\*n, потому что пивот разделил элемент на доли.

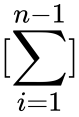
- Лучший случай. В наиболее сбалансированным варианте при каждой операции разделение массив делится на две одинаковые части, т.е. a = 1/2.

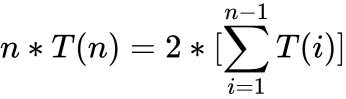
Тогда: T(n) = n + 2\*T(n/2) = n + 2\*(n/2 + 2\*T(n/4)) = n + n + 4\*T(n/4) = n + n + 4\*(n/4 + 2\*T(n/8)) = n + n + n + 8\*T(n/8) = … И так далее, максимальная глубина рекурсии, при которой размеры обрабатываемых подмассивов достигнут 1, составит log2n.

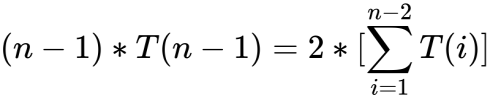
В результате количество сравнений, совершаемых быстрой сортировкой, было бы равно значению рекурсивного выражения Cn=2\*Cn/2+n.

- Средний случай происходит при случайном распределении входных данных можно оценить лишь вероятно. В таком случае наш массив делится на две части k и (n-k). Тогда:

T(n) = T(n-k) + T(k) = 1/N\*[]. Так как эти функции равновероятны, то можно получить следующее:

T(n) = 2/n \*





Если вычесть эти два уравнения, то:

n\*T(n) - (N-1)\*T(n-1) = 2\*T(n-1) + n2 \* const - (n-1)2 \* const ->

n\*T(n) = T(n-1)\*(2+n-1) + const + 2\*n\*const-const = (n+1)\*T(n-1)+2\*n\*const. ->

T(n)/(n+1) = T(n-1)/n + 2\*const/(n+1). Пусть n = n-1 ->

T(n-1)/n = T(n-2)(n-1) + 2\*const/n ->

T(n) / (n+1) = T(n-2)/(n-1) +2\*const/(n+1)+2\*const/n

Заменим n на n-2 ->

T(n)/(n+1) = T(1)/2 + 2\*const\*[1/2 + 1/3 + … + 1/(n-1) + 1/n + 1/(n+1)] -> T(n) = 2 \* const + log2n \* (n+1).

Если убрать константу, то: T(n) = log2n \* (n+1).

- Худший случай возникает, когда пивот отсекает ровно один элемент, то есть a = 1/n. В первой части массива находится 1 элемент, во второй - n - 1. Тогда:

T(n) = n + T(1) + T(n-1) = n + O(1) + T(n-1) = n + O(1) + (n-1+O(1)+T(n-2)).

Сортировка называется быстрой, потому что константа, которая скрывается по знаком O на практике оказывается достаточно небольшой.

1. Асимптотическая оценка временной сложности

- Лучший случай: T(n) = 2\*T(n/2) + N \* const = 2\*(2\*T(n/4) + n/2 \* const) + n\*const = 4\*T(n/4) + 2\*const\*n ->

T(n) = 2k \* T(n/2k) + k \* const \* n, тогда 2k = n, k = log2n ->

T(n) = n\*T(1) + n\*log2n = O(nlogn)

- Средний случай выведен выше, поэтому T(n) = log2n \* (n+1) = O(nlogn).

- Худший случай: T(n) = n + T(1) + T(n-1) = n + O(1) + T(n-1) = n + O(1) + (n-1+O(1)+T(n-2)) = O(n2).

1. Функция пространственной сложности и её асимптотическая оценка.

Пространственная сложность при наихудшем сценарии будет O(n) из-за несбалансированного разбиения на разделы, что приводит к перекошенному дереву рекурсии, требующему стек вызовов размером O(n).

При наилучшем сценарии в результате сбалансированного разбиения на разделы мы получим сбалансированное дерево рекурсии со стеком вызовов размером O(logn).

1. Графики функций.

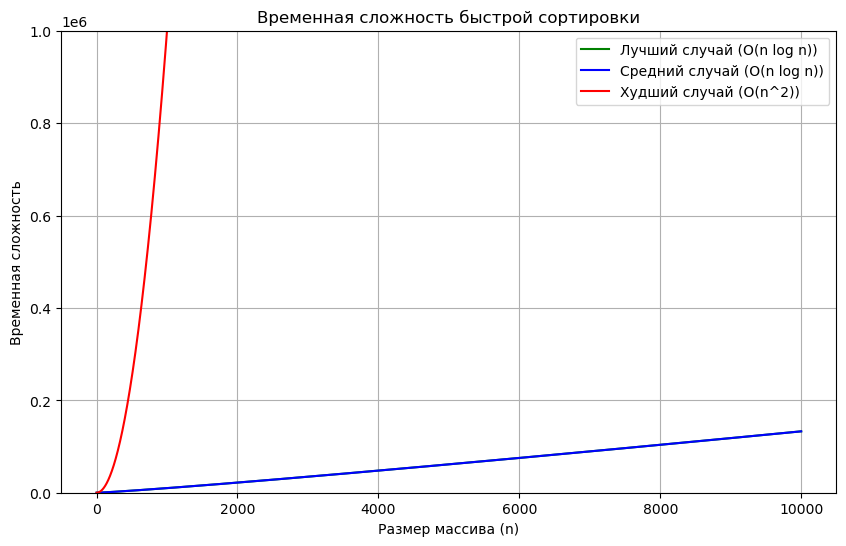


Рис. 14

**Пирамидальная сортировка (heap sort).**

1. Асимптотическую временная и пространственная сложность.

|  |  |  |  |
| --- | --- | --- | --- |
| Tbest | Taverage | Tworst | S’(n) |
| O(nlogn) | O(nlogn) | O(nlogn) | O(1) |

1. Алгоритм сортировки.

Пирамидальная сортировка использует бинарное сортирующее дерево - такое дерево, у которого выполнены следующие условия: каждый лист имеет глубину либо d, либо d-1, d - максимальная глубина дерева, а также значение в любой вершине не меньше значения её потомков.

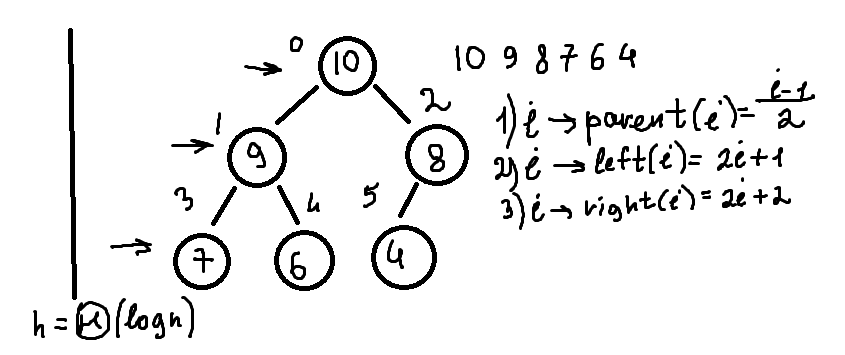


Рис. 15

Алгоритм будет состоять из двух основных шагов:

- Сначала выстраиваем элементы массива в виде сортирующего дерева: a[I] >= a[2i+1], a[I] >= a[2i+2] при 0 <= I <= n/2. Элементы массива от n/2 до n являются листьями дерева, а следовательно, правильными пирамидами одного элемента. Для остальных в порядке уменьшения индекса просеиваем их через правую часть массива.

- Теперь будем удалять элементы из корня п одному и перестраивать дерево. Первый элемент массива максимален (корень пирамиды), поменяем его с последним ( таким образом, последний элемент отсортирован). Теперь для первого элемента свойство кучи нарушено: повторим просеивание первого элемента в пирамиде от первого до предпоследнего, снова поменяем первый и предпоследний и т.д до тех пор, пока в сортирующем дереве не останется один элемент.

1. Анализ устойчивости.

Сортировка кучей нестабильна, так как может изменить относительный порядок элементов в массиве.

Рассмотрим массив 21 20a 20b 12 11 8 7 (уже в формате max-heap), здесь 20a = 20b просто чтобы различать порядок, представим их как 20a и 20b.

В то время как heapsort сначала удаляет 21 и помещает в последний индекс, затем 20a удаляется и помещается в предпоследний индекс и 20b перемещается в предпоследний для предыдущего, то после сортировки кучей массив выглядит так:

7 8 11 12 20b 20a 21.

Видно, что он порядок элементов не созхранился, поэтому данная сортировка не устойчива.

1. Функции временной сложности.

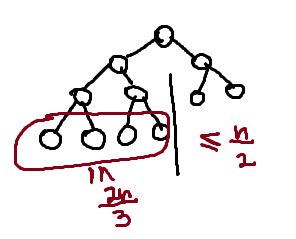
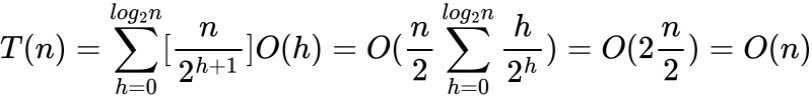


Рис. 16

Оценим время работы построения пирамиды (1 шаг алгоритма). Высота в пирамиде - число рёбер в самом длинном простом нисходящем пути от узла к какому-нибудь из листьев: h = log2n. На любом уровне на высоте h содержится не более [n/2h+1] вершин. Тогда:



Сложность самой сортировки (2 шаг в алгоритме): для каждого извлечения максимального элемента требуется O(logn) времени, так как необходимо поддерживать структуру кучи после удаления. Посколько мы извлекаем элемент n раз, то это приводит к временной сложности O(nlogn).

Для всех случаев сортировки сложность будет одинаковая. Это происходит по нескольким причинам:

Во-первых, процесс построения кучи, который выполняется за O(n), не зависит от порядка элементов в исходном массиве. Во-вторых, после того как куча сформирована, каждый элемент извлекается и перемещается на его правильную позицию. Этот процесс также выполняется за O(logn) на каждую итерацию, поскольку требуется восстановить свойства кучи после удаления корня. Поскольку в общем случаев нужно выполнить n извлечений, общая сложность остается O(nlogn).

Таким образом, даже если входные данные упорядочены или полностью неупорядочены, этапы алгоритма не изменяются, и это приводит к постоянной временной сложности.

1. Асимптотическая оценка функций временной сложности.

O(n) + O(nlogn) = O(nlogn)

Таким образом, эффективность пирамидальной сортировки остаётся стабильной вне зависимости от исходного порядка данных.

1. Функция и асимптотическая оценка пространственной сложности.

Функция пространственной сложности пирамидальной сортировки (heapsort) составляет O(1) для итеративной реализации. Это связано с тем, что алгоритм выполняется на месте, используя только постоянное количество дополнительной памяти для переменных, таких как индексы и временные переменные. Строительство кучи также может быть выполнено на месте, то есть без необходимости выделять дополнительное пространство для хранения элементов в промежуточных структурах данных, что дополнительно подтверждает низкую пространственную сложность. Однако стоит отметить, что для реализации, использующей рекурсию, пространственная сложность может быть O(logn) из-за хранения контекста вызовов стека.

Таким образом, асимптотическая оценка пространственной сложности пирамидальной сортировки для итеративного подхода — это O(1).

1. Графики функции временной сложности.

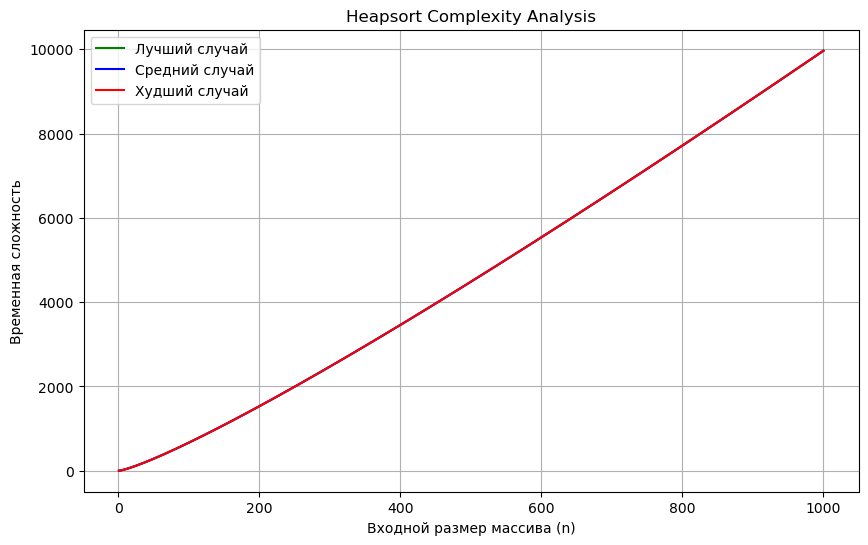


Рис. 17

# Экспериментальная часть.

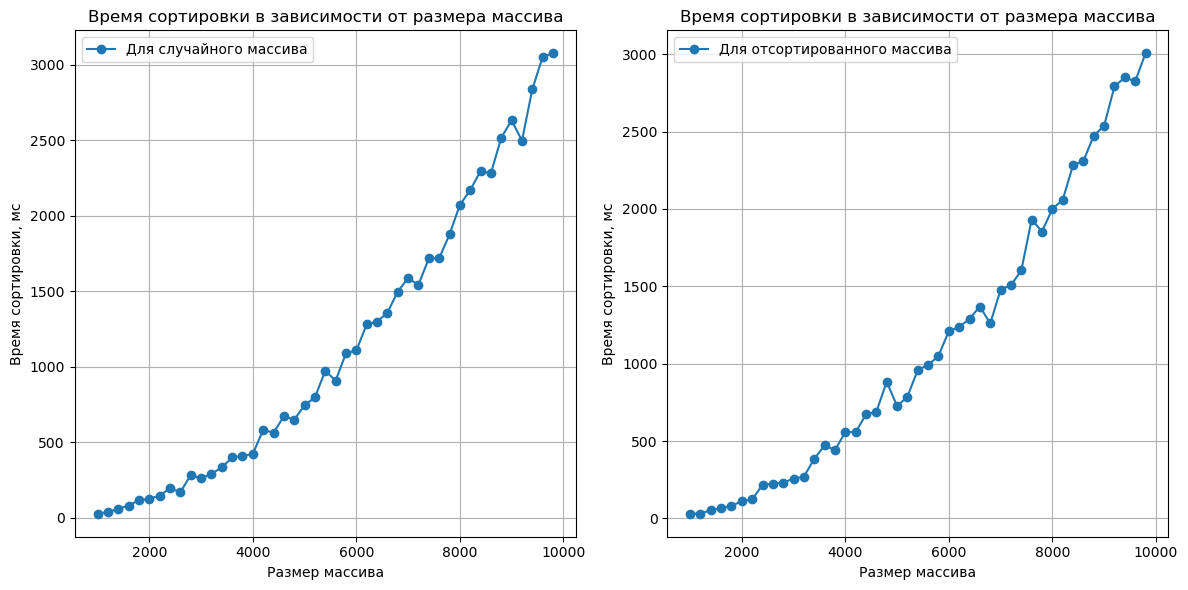
В данном разделе реализованные функции будут воспроизведены на массивах длиной от 1000 до 10000 элементов с шагом 200 (там где несколько на одном графике - от 500 до 5000 с шагом 50), и для каждого размера было посчитано время выполнения.

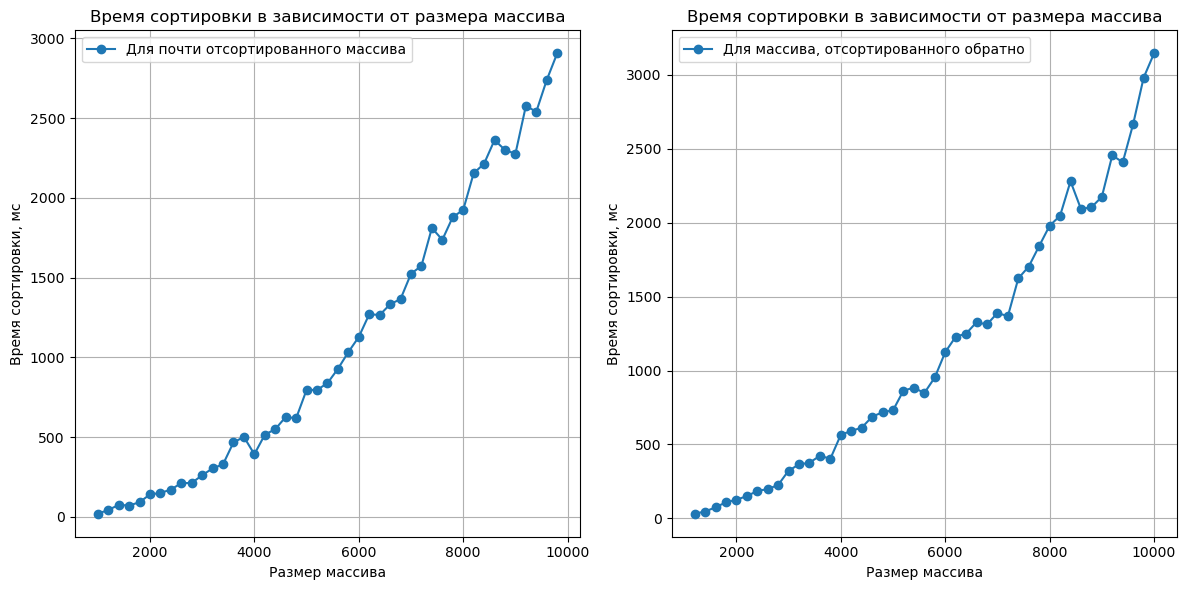
Представлены четыре набора массивов: произвольные, отсортированные, почти отсортированные и обратно отсортированные, для каждого набора были составлены аппроксимированные графики зависимости времени от размера.

Оси абсцисс соответствуют размеры массивов, оси ординат — время выполнения для них в миллисекундах.

Уравнение регрессионной кривой — .

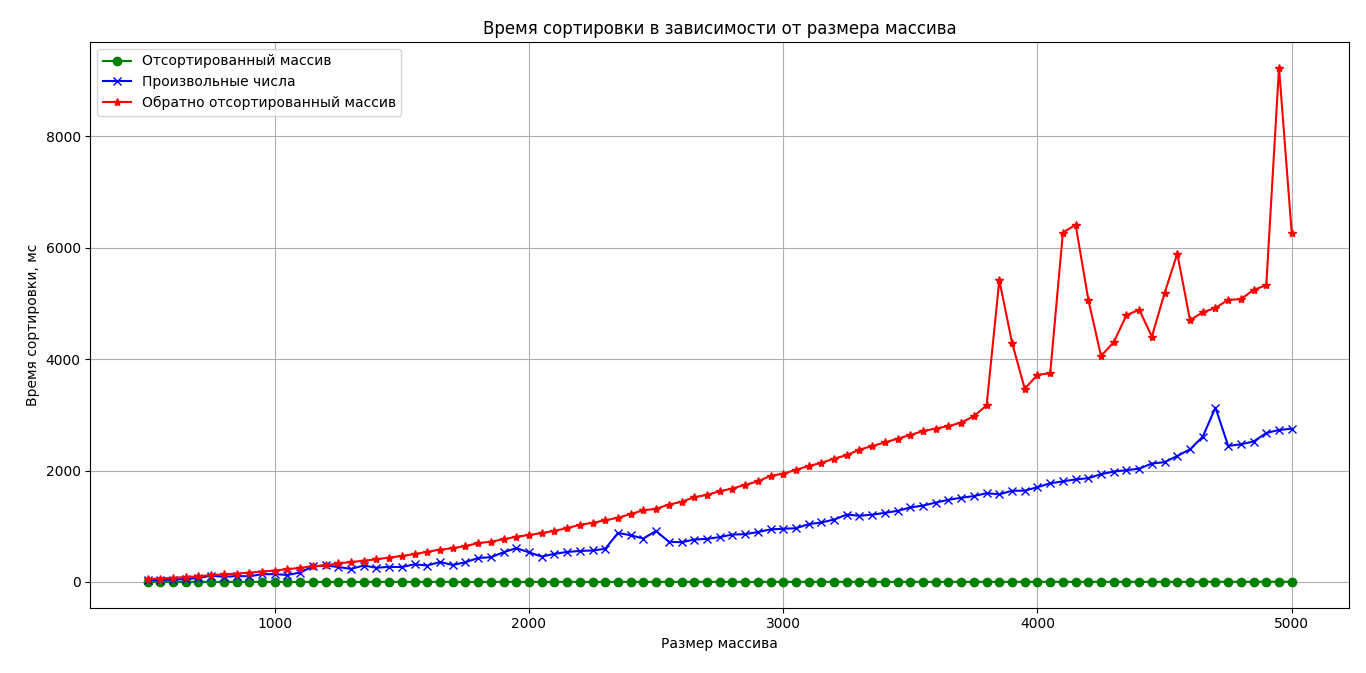
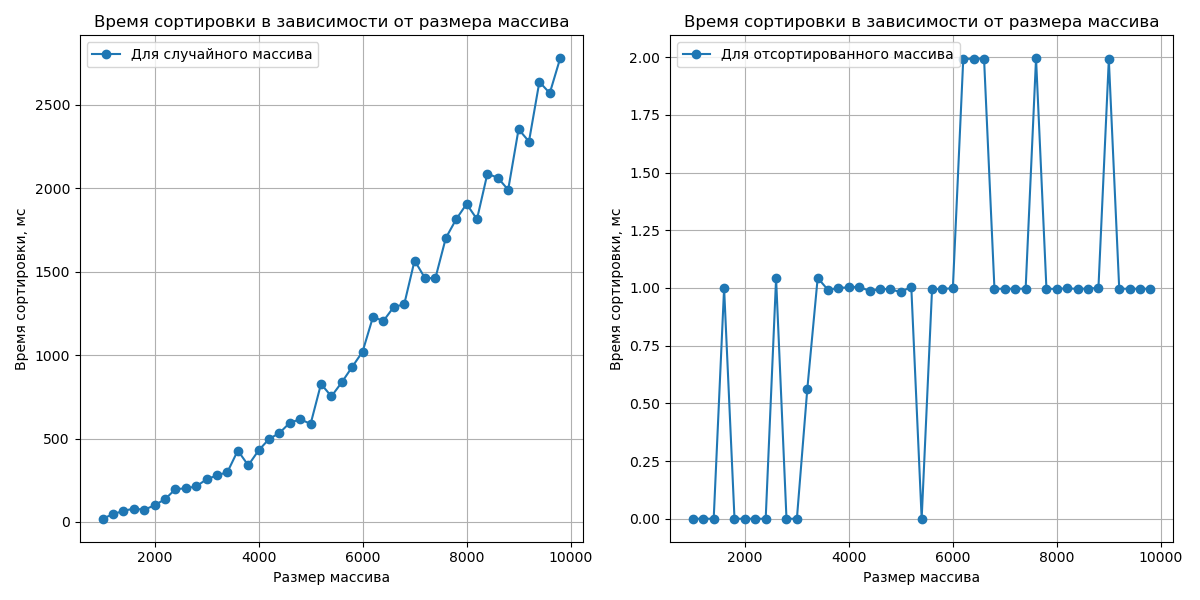
1. Сортировка выбором (selection sort).





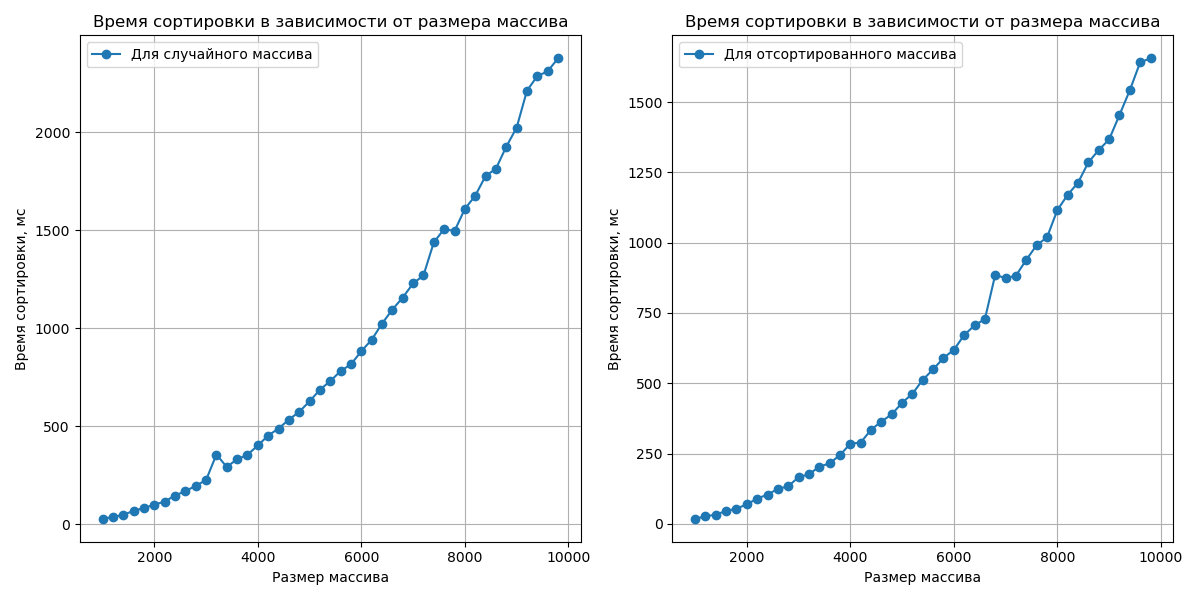
Уравнения регрессионных кривых:

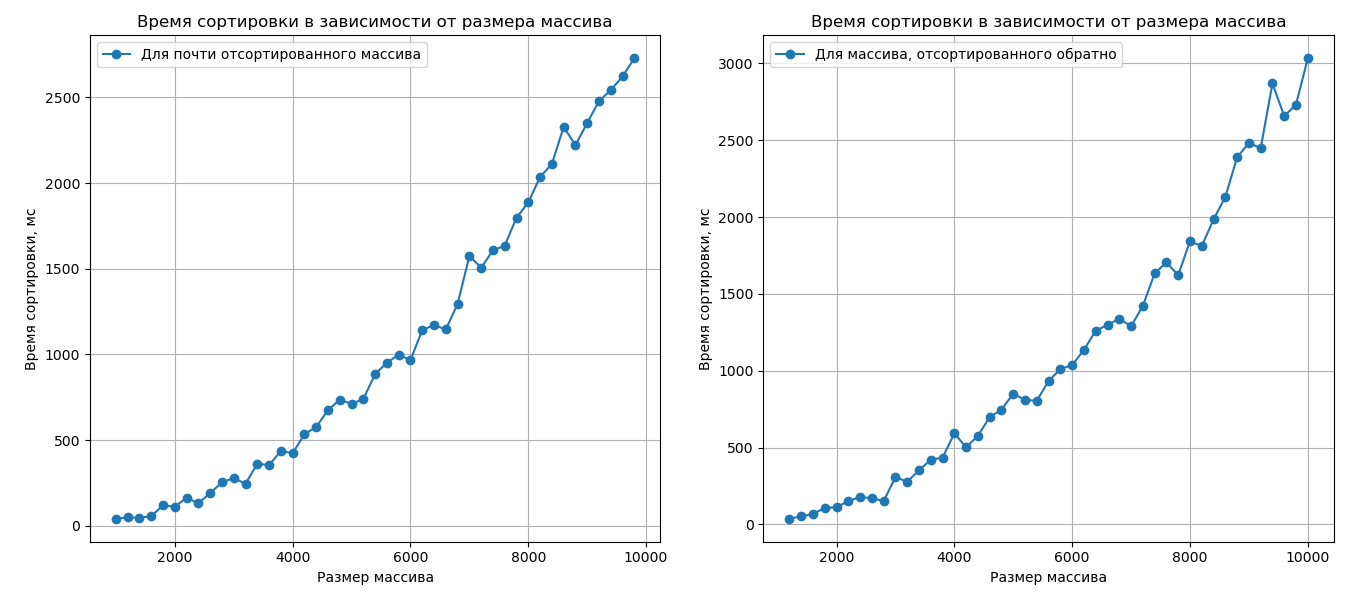
1. Сортировка вставками (insertion sort).



Уравнения регрессионных кривых:

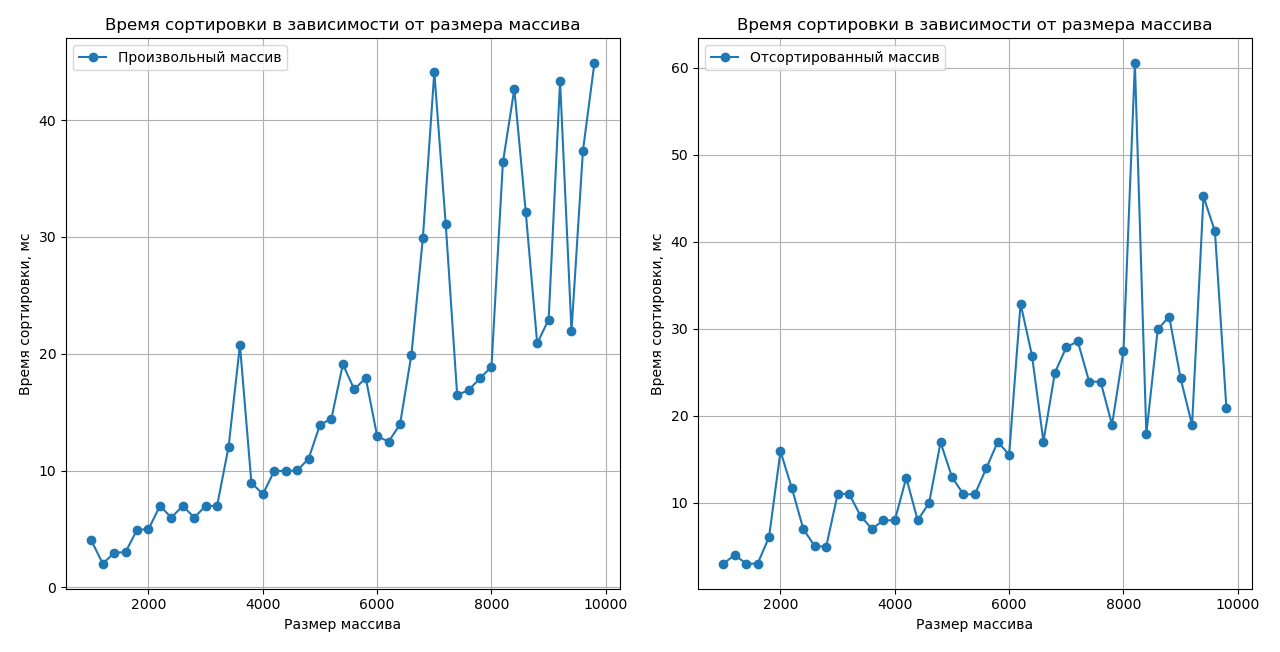
2. Сортировка пузырьком (bubble\_sort)

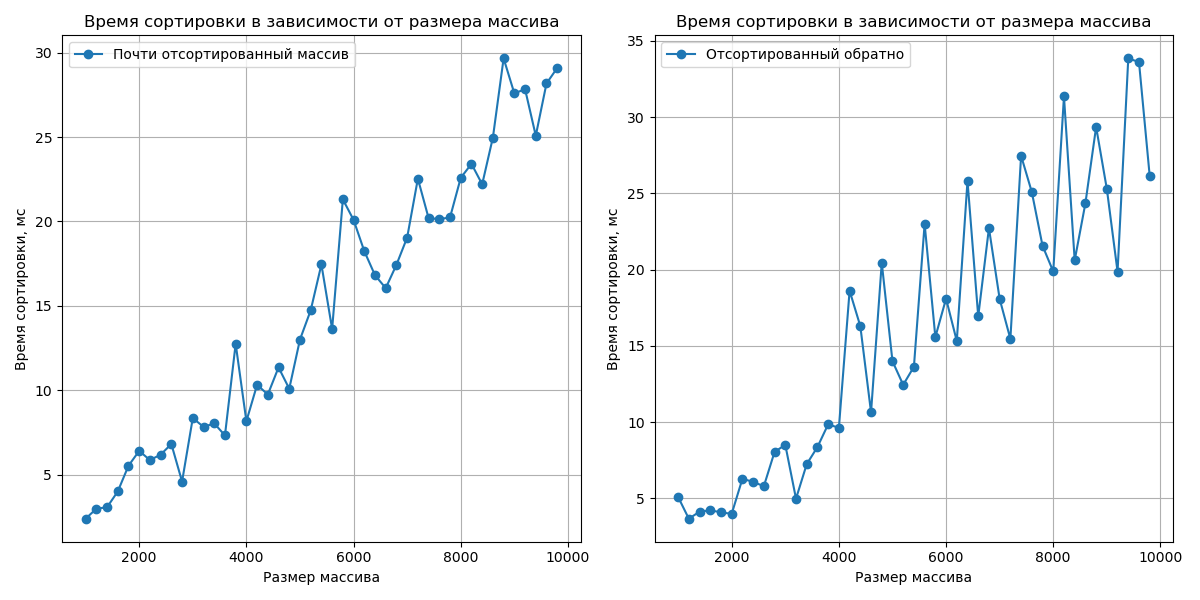




Уравнения регрессионных кривых:

4. Сортировка слиянием (merge sort).





Уравнения регрессионных кривых:

3.

4.

1. Сортировка Шелла (shell sort)

# shell_sort 2shell_sort 1

Уравнения регрессионных кривых:

3.

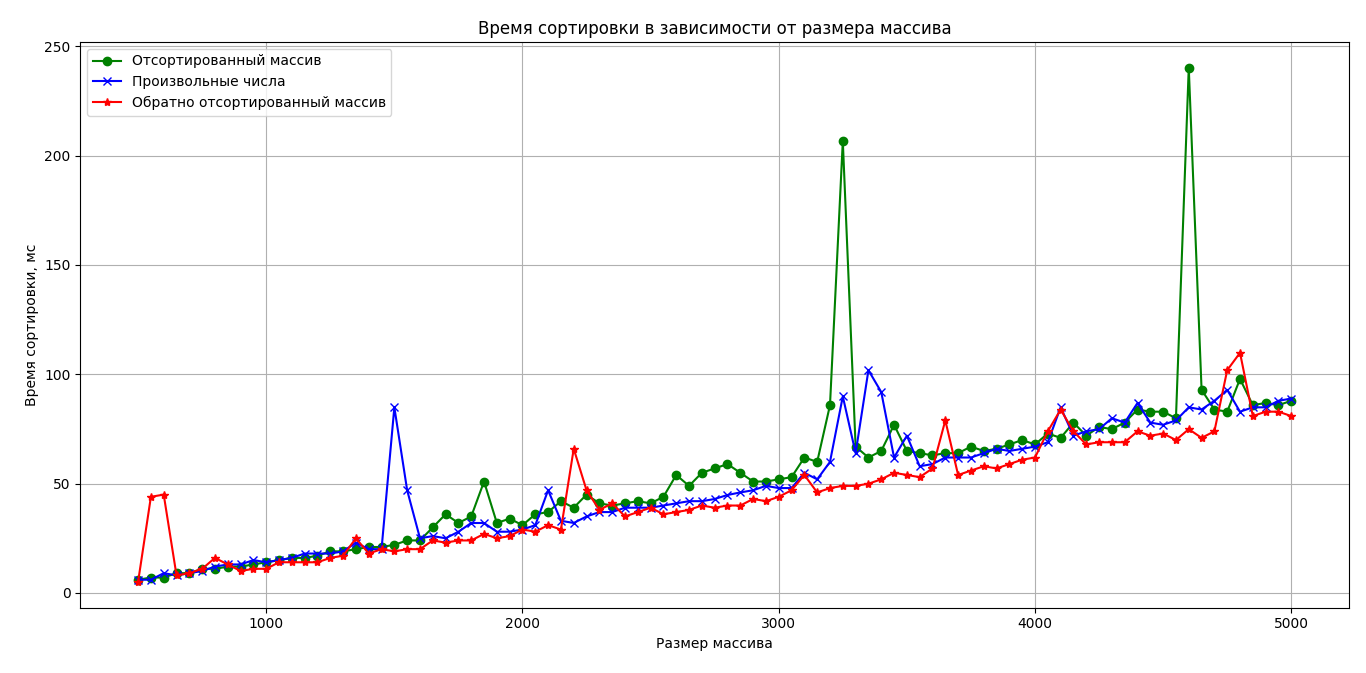
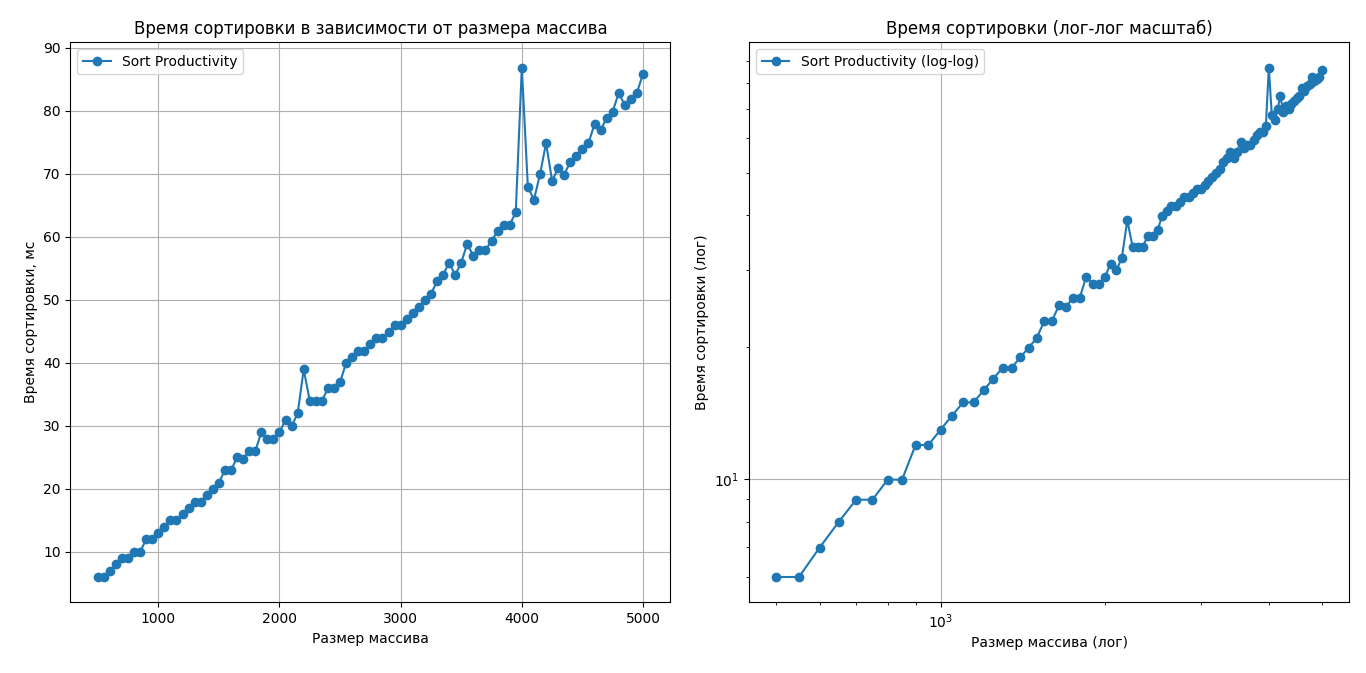
4.

6. Быстрая сортировка (quick sort)

# йгшслскн

Уравнения регрессионных кривых:

7. Пирамидальная сортировка (heap sort)

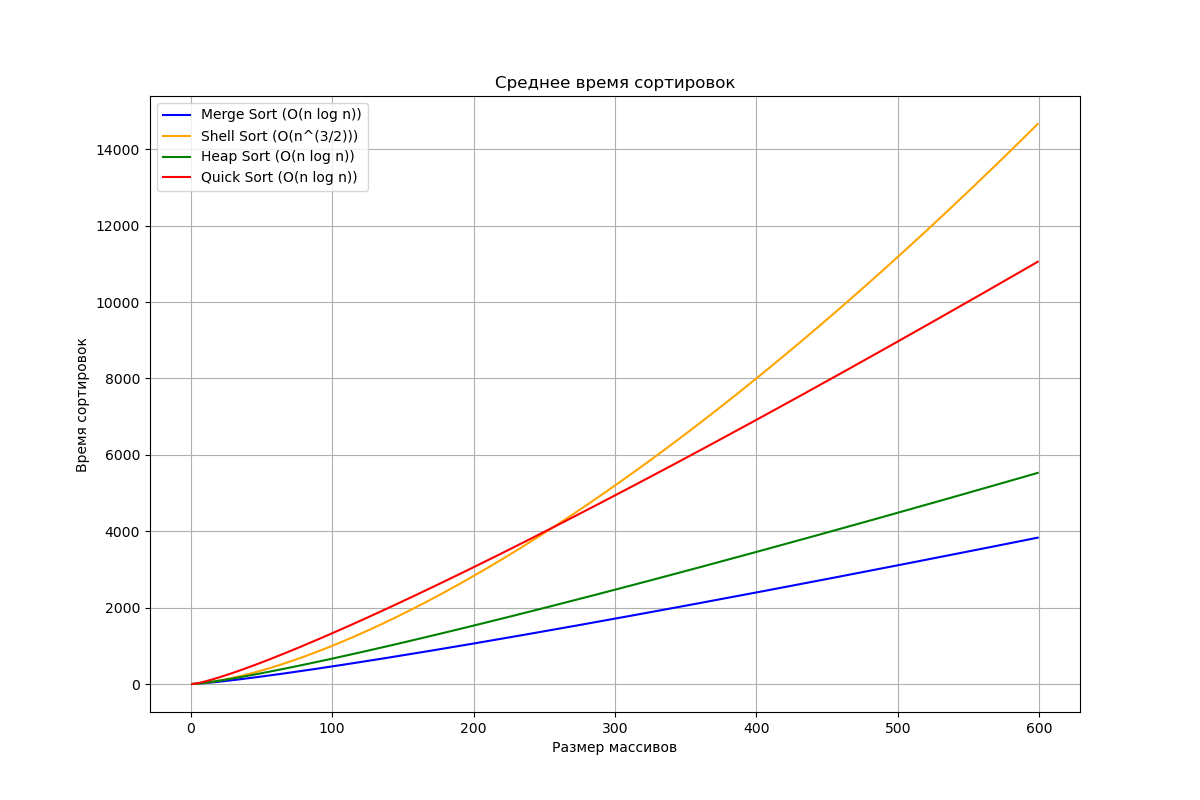


Уравнения регрессионных кривых:

# Сравнение всех сортировок.

# all

Сравнение всех сортировок без квадратичных.



**Итоговая оценка эффективности использования алгоритмов.**

Самые лёгкие и понятные для реализации сортировки из разобранных - сортировки выбором и пузырьком, однако на практике они являются самыми медленными и неэффективными, поэтому их практическая польза только в качестве учебных примеров.

Сортировка вставками тоже неэффективная, но у неё есть преимущество на небольших данных, поэтому она часто применяется в сочетании с другими алгоритмами (на её основе создаются более эффективные).

В общем случае, алгоритмы, такие как QuickSort и MergeSort, являются одними из самых эффективных для сортировки больших объемов данных. QuickSort часто демонстрирует хорошую производительность благодаря своей средней временной сложности O(n log n), тогда как MergeSort гарантирует эту сложность даже в худших случаях, но требует дополнительной памяти.

Сортировка Шелла хоть и не такая быстрая, но относительно устойчивая по асимптотике и не затратная по памяти, поэтому она нашла себе практическое применение( например, Ядро Linux - в нём используется сортировка Шелла, потому что она не требует стека вызовов, ещё библиотека uClibc).

Пирамидальная сортировка тоже является очень быстрым и не затратным по памяти алгоритмом, но теряет свою эффективность на очень больших размерах данных или на почти отсортированных наборах данных.

Однако всё же для маленьких массивов или почти отсортированных данных может быть эффективнее использовать такие простые алгоритмы, как Insertion Sort.

# Репозиторий Github c исходным кодом.

# Раздел со всем кодом

import random

import time

import matplotlib

import matplotlib.pyplot as plt

from collections import defaultdict

# Сортировка выбором

def selection\_sort(arr):

for i in range(0, len(arr) - 1):

mini = i

for j in range(i + 1, len(arr)):

if arr[j] < arr[mini]:

mini = j

arr[i], arr[mini] = arr[mini], arr[i]

# Сортировка вставками

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

while j >= 0 and key < arr[j]:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

# Пузырьковая сортировка

def bubble\_sort(arr):

n = len(arr)

for bypass in range(1, n):

for i in range(n-bypass):

if arr[i] > arr[i+1]:

arr[i], arr[i+1] = arr[i+1], arr[i]

# Сортировка слиянием

def merge\_sort(arr):

if len(arr) <= 1:

return arr

mid = len(arr) // 2

left\_part = merge\_sort(arr[:mid])

right\_part = merge\_sort(arr[mid:])

return merge(left\_part, right\_part)

def merge(left, right):

sorted\_array = []

i = j = 0

while i < len(left) and j < len(right):

if left[i] < right[j]:

sorted\_array.append(left[i])

i += 1

else:

sorted\_array.append(right[j])

j += 1

sorted\_array.extend(left[i:])

sorted\_array.extend(right[j:])

return sorted\_array

# Сортировка Шелла

def shell\_sort(arr):

n = len(arr)

gap = n // 2

while gap > 0:

for i in range(gap, n):

temp = arr[i]

j = i

while j >= gap and arr[j - gap] > temp:

arr[j] = arr[j - gap]

j -= gap

arr[j] = temp

gap //= 2

# Быстрая сортировка

def quick\_sort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2]

left = [x for x in arr if x < pivot]

middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quick\_sort(left) + middle + quick\_sort(right)

# Пирамидальная сортировка

def heapify(arr, n, i):

main = i

left\_child = (2 \* i) + 1

right\_child = (2 \* -) + 2

if left\_child < n and arr[left\_child] > arr[largest]:

main = left\_child

if right\_child < n and arr[right\_child] > arr[largest]:

main = right\_child

if largest != i:

arr[i], arr[main] = arr[main], arr[i]

heapify(arr, n, main)

def heap\_sort(arr):

n = len(arr)

for i in range(n, -1, -1):

heapify(arr, n, i)

for i in range(n - 1, 0, -1):

arr[i], arr[0] = arr[0], arr[i]

heapify(arr, i, 0)

def fill\_array(arr):

for i in range(len(arr)):

arr[i] = random.randint(-len(arr), len(arr))

def almost\_sorted(arr):

sorted\_array = list(range(len(arr)))

num\_unsorted\_elements = len(arr) // 10

random\_idx = random.sample(range(len(arr)), num\_unsorted\_elements)

for idx in random\_idx:

swap\_with = random.randint(0, len(arr)-1)

sorted\_array[idx], sorted\_array[swap\_with] = sorted\_array[swap\_with], sorted\_array[idx]

return sorted\_array

def beg\_sort(arr):

current\_value = random.randint(-len(arr), len(arr))

for i in range(len(arr)):

arr[i] = current\_value

step = random.randint(-len(arr), len(arr))

current\_value += step

def back\_sorted(arr):

for i in range(len(arr)):

arr[i] = random.randint(-len(arr), len(arr))

for i in range(len(arr)):

for j in range(0, len(arr) - i - 1):

if arr[j] < arr[j + 1]:

arr[j], arr[j + 1] = arr[j + 1], arr[j]

return arr

def running\_time(action, arr):

start\_time = time.time()

action(arr)

return (time.time() - start\_time) \* 1000

def display\_statistics():

results = defaultdict(int)

results1 = defaultdict(int)

results2 = defaultdict(int)

results3 = defaultdict(int)

# Сбор результатов - в running\_time поменять на функцию нужной сортировки

for i in range(1000, 10000, 200):

ar = [0] \* i

fill\_array(ar)

results[i] = running\_time(selection\_sort, ar)

for i in range(1000, 10000, 200):

ar1 = [0] \* i

beg\_sort(ar1)

results1[i] = running\_time(selection\_sort, ar1)

for i in range(1000, 10000, 200):

ar2 = [0] \* i

almost\_sorted(ar2)

results2[i] = running\_time(selection\_sort, ar2)

for i in range(10000, 1000, -200):

ar3 = [0] \* i

back\_sorted(ar3)

results3[i] = running\_time(selection\_sort, ar3)

# Обработка данных для графика

sizes = list(results.keys())

times = list(results.values())

sizes1 = list(results1.keys())

times1 = list(results1.values())

sizes2 = list(results2.keys())

times2 = list(results2.values())

sizes3 = list(results3.keys())

times3 = list(results3.values())

plt.figure(figsize=(12, 6))

plt.subplot(1, 4, 1)

plt.plot(sizes, times, label='Для произвольного массива', marker='o')

plt.title('Время сортировки в зависимости от размера массива')

plt.xlabel('Размер массива')

plt.ylabel('Время сортировки, мс')

plt.legend()

plt.grid()

plt.subplot(1, 4, 2)

plt.plot(sizes1, times1, label='Для отсортированного массива', marker='o')

plt.title('Время сортировки в зависимости от размера массива')

plt.xlabel('Размер массива')

plt.ylabel('Время сортировки, мс')

plt.legend()

plt.grid()

plt.subplot(1, 4, 3)

plt.plot(sizes2, times2, label='Для почти отсортированного массива', marker='o')

plt.title('Время сортировки в зависимости от размера массива')

plt.xlabel('Размер массива')

plt.ylabel('Время сортировки, мс')

plt.legend()

plt.grid()

plt.subplot(1, 4, 4)

plt.plot(sizes3, times3, label='Для массива, отсортированного обратно', marker='o')

plt.title('Время сортировки в зависимости от размера массива')

plt.xlabel('Размер массива')

plt.ylabel('Время сортировки, мс')

plt.legend()

plt.grid()

plt.tight\_layout() # Компактное размещение графиков

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

display\_statistics()