**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В. И. Ульянова (Ленина)**

**Кафедра систем автоматизированного проектирования**

**Лабораторная работа №1**

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

**Тема: Алгоритмы сортировки сравнением**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3353 |  | Родачина А. А. |
| Преподаватель |  | Пестерев Д. О. |

Санкт-Петербург

2024

**Оглавление**

[Теоретическая часть 2](#_Toc180845316)

[Сортировка выбором (Selection Sort) 2](#_Toc180845317)

[Сортировка вставками (Insertion sort) 4](#_Toc180845318)

[Сортировка пузырьком (Bubble sort) 6](#_Toc180845319)

[Сортировка слиянием (Merge Sort) 8](#_Toc180845320)

[Сортировка Шелла (Shell Sort) 11](#_Toc180845321)

[Быстрая сортировка (Quick Sort) 13](#_Toc180845322)

[Пирамидальная сортировка (HeapSort) 15](#_Toc180845323)

[Предположение о наибыстрейшей сортировки 19](#_Toc180845324)

[Практическая часть 20](#_Toc180845325)

[Сортировка выбором (Selection Sort) 20](#_Toc180845326)

[Сортировка вставками (Insertion Sort) 23](#_Toc180845327)

[Сортировка пузырьком (Bubble Sort) 26](#_Toc180845328)

[Сортировка слиянием (Merge Sort) 29](#_Toc180845329)

[Сортировка Шелла (Shell Sort) 33](#_Toc180845330)

[Сортировка Шелла – Последовательность Хиббарда (Shell Sort) 36](#_Toc180845331)

[Сортировка Шелла – Последовательность Пратта (Shell Sort) 39](#_Toc180845332)

[Быстрая сортировка (Quick Sort) 43](#_Toc180845333)

[Пирамидальная сортировка (HeapSort) 46](#_Toc180845334)

[Сравнение теоретических и практических результатов 49](#_Toc180845335)

[Сравнение практических результатов сортировок 53](#_Toc180845336)

[Вывод 55](#_Toc180845337)

# Теоретическая часть

## Сортировка выбором (Selection Sort)

1. Асимптотическая временная и пространственная сложность:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Лучший случай | Худший случай | Средний случай |
| Временная сложность | O() | O() | O() |
| Пространственная сложность | O(1) | O(1) | O(1) |

1. Описание алгоритма сортировки :

Начинаем с первого элемента в массиве и считаем его текущим. Проходим по всему массиву и ищем элемент, который будет меньше текущего. Когда находим такой элемент, меняем его с текущим. Дальше переходим ко второму элементу и повторяем процесс. Так до конца, пока массив не будет отсортирован.

Код на Python:

def selection\_sort(a):

for i in range(len(a)):

min\_index = i

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

if a[j] < a[min\_index]:

min\_index = j

a[i], a[min\_index] = a[min\_index], a[i]

return a

Код на Java:

public static void selection\_sort(int[] a) {  
 int n = a.length;   
 for (int i = 0; i < n - 1; i++) {   
 int min\_index = i;  
 for (int j = i + 1; j < n; j++) {  
 if (a[j] < a[min\_index]) {  
 min\_index = j;  
 }  
 }   
 int temp = a[min\_index];  
 a[min\_index] = a[i];  
 a[i] = temp;  
 }  
}

Устойчивость алгоритма сортировки:

Сортировка выбором является не устойчивой.

Проанализируем: возьмем массив [7,7,8,3]. По алгоритму сортировки выбором после первой проходки 7 и 3 поменяются местами: [3,7,8,7]. Элементы с одинаковым значением (7) уже поменялись местами. Переходим ко второму элементу – массив не меняется. На третьей итерации элемент со значением 8 поменяется с последним элементом массива. Итог: [3,7,7,8] – 7-ки поменялись местами, значит сортировка выбором не устойчивая.

1. Функция временной сложности для всех случаев:

Лучший случай: если входной массив уже отсортирован, сначала будет проходиться внешний цикл по каждому элементу массива (n итераций). Внутренний цикл будет проходиться по оставшимся элементам (n-i), где i – порядок текущего элемента внешнего цикла.

Худший случай: если массив будет отсортирован в обратном порядке, алгоритм так же будет проходить во внешнем цикле по всем элементам, а во внутреннем – (n-i).

Средний случай: даже при случайном порядке элементов сортировка выбором все равно будет выполнять те же действия, что и в худшем или лучшем случаях.

1. Асимптотическая оценка временной сложности для лучшего, среднего и худшего случая:

Лучший случай:

Худший случай:

Средний случай:

1. Функция пространственной сложности и её асимптотическую оценка:

Для сортировки выбором требуется только небольшое количество дополнительных переменных (индексы i, j и временная переменная для обмена элементов min\_index). Это постоянная память, которая не зависит от n и ее можно обозначить как O(1).

Таким образом, асимптотическая пространственная сложность алгоритма сортировки выбором равна O(1).

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

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

## Сортировка вставками (Insertion sort)

1. Асимптотическая временная и пространственная сложность:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Лучший случай | Худший случай | Средний случай |
| Временная сложность | O(n) | O() | O() |
| Пространственная сложность | O(1) | O(1) | O(1) |

1. Описание алгоритма сортировки :

Алгоритм начинается с того, что первый элемент массива считается отсортированным. Для каждого следующего элемента массива находим его правильное место в отсортированной части (это делается путем сравнения текущего элемента с уже отсортированными элементами и сдвига этих элементов вправо, чтобы освободить место для вставки текущего элемента). Процесс продолжается до тех пор, пока все элементы не будут обработаны.

Код на Python:

def insertion\_sort(a):

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

cur = a[i]

j = i - 1

while j >= 0 and a[j] > cur:

a[j + 1] = a[j]

j -= 1

a[j + 1] = cur

return a

1. Устойчивость алгоритма сортировки:

Сортировка вставками является устойчивой.

Проанализируем: возьмем массив [7,3,8,7]. По алгоритму сортировки вставками начинаем с cur=3. Элемент меньше 7, следовательно они поменяются местами: [3,7,8,7]. На cur=8 массив не изменится. А на cur=7 (последний элемент) 8 поменяется с последним элементом. Заметим, что дальше cur элемент не будет меняться, так как по условию цикла a[j]>cur – строгое неравенство. Следовательно алгоритм закончит сортировку: [3,7,7,8], не меняя порядок элементов с одинаковыми значениями.

1. Функция временной сложности для всех случаев:

Лучший случай: если входной массив уже отсортирован, сортировка вставками проходит по массиву только один раз без перестановок.

Худший случай: если массив будет отсортирован в обратном порядке, алгоритм будет каждый i элемент массива сравнивать со всеми элементами до него и перемещать этот элемент в начало массива.

Средний случай: в среднем i-ый элемент будет сравниваться с ~ половиной элементов до него.

1. Асимптотическая оценка временной сложности для лучшего, среднего и худшего случая:

Лучший случай:

Худший случай:

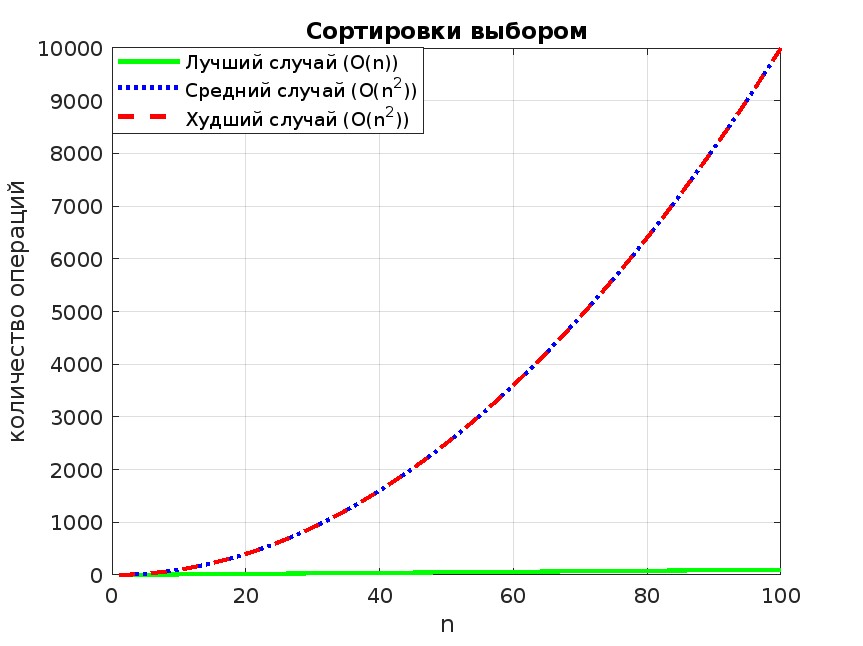
Средний случай:

1. Функция пространственной сложности и её асимптотическую оценка:

Для сортировки вставками требуется только небольшое количество дополнительных переменных (индексы i, j и временная переменная для обмена элементов cur). Это постоянная память, которая не зависит от n и ее можно обозначить как O(1).

Таким образом, асимптотическая пространственная сложность алгоритма сортировки вставками равна O(1).

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



## Сортировка пузырьком (Bubble sort)

1. Асимптотическая временная и пространственная сложность:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Лучший случай | Худший случай | Средний случай |
| Временная сложность | O(n) | O() | O() |
| Пространственная сложность | O(1) | O(1) | O(1) |

1. Описание алгоритма сортировки :

Алгоритм проходит по массиву, начиная с первого элемента. Каждый элемент сравнивается с его соседним элементом. Если текущий элемент больше, чем следующий, они меняются местами. После одного полного прохода самый большой элемент оказывается на своём месте в конце массива. Затем алгоритм повторяет этот процесс для оставшейся части массива. Алгоритм продолжает работу до тех пор, пока за один проход не произойдет ни одного обмена, что означает, что массив отсортирован.

Код на Python:

def bubble\_sort(a):

n = len(a)

for i in range(n):

swapped = False

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

if a[j] > a[j + 1]:

a[j], a[j + 1] = a[j + 1], a[j]

swapped = True

if not swapped:

break

return a

1. Устойчивость алгоритма сортировки:

Сортировка вставками является устойчивой.

Проанализируем: возьмем массив [7,3,8,7]. По алгоритму сортировки пузырьком начинаем с a[j]=7 и a[j+1]=3. 7>3, следовательно они поменяются местами: [3,7,8,7]. На a[j]=7 и a[j+1]=8: 7<8 массив не меняется. А на a[j]=8 и a[j+1]=7, элементы поменяются местами. Дальше обмен элементов закончится, следовательно алгоритм закончит сортировку: [3,7,7,8], не меняя порядок элементов с одинаковыми значениями.

1. Функция временной сложности для всех случаев:

Лучший случай: если входной массив уже отсортирован, сортировка пузырьком проходит по массиву только один раз без обменов.

Худший случай: если массив будет отсортирован в обратном порядке, алгоритм для каждого элемента должен выполнить n-i-1 сравнений, и потребуется выполнить максимальное количество обменов.

Средний случай: в среднем случае алгоритм ведет себя также, как и в худшем, так как для неупорядоченного массива все пары элементов должны быть проверены и, возможно, переставлены.

1. Асимптотическая оценка временной сложности для лучшего, среднего и худшего случая:

Лучший случай:

Худший случай:

Средний случай:

1. Функция пространственной сложности и её асимптотическую оценка:

Для сортировки пузырьком требуется только небольшое количество дополнительных переменных (индексы i, j и временная переменная для проверки наличия обмена swap). Это постоянная память, которая не зависит от n и ее можно обозначить как O(1).

Таким образом, асимптотическая пространственная сложность алгоритма сортировки вставками равна O(1).

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

Изображение выглядит как текст, График, снимок экрана, линия

Автоматически созданное описание

## Сортировка слиянием (Merge Sort)

1. Асимптотическая временная и пространственная сложность:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Лучший случай | Худший случай | Средний случай |
| Временная сложность | O() | O() | O() |
| Пространственная сложность | O(n) | O(n) | O(n) |

1. Описание алгоритма сортировки :

Алгоритм начинает с того, что рекурсивно делит массив пополам, до того, пока не достигнем подмассивов длиной 1. Процесс начинается с деления исходного массива на две равные части, пока каждая часть не станет содержать один элемент, что считается отсортированным. Затем, начиная с самых мелких подмассивов, происходит слияние: элементы сравниваются и объединяются в новый массив так, чтобы сохранялся порядок. Этот процесс продолжается до тех пор, пока все подмассивы не будут объединены в один отсортированный массив. Таким образом, на каждом этапе мы "сливаем" два отсортированных подмассива в один, что в итоге приводит к полной сортировке всего массива.

Код на Python:

def merge\_sort(arr):

if len(arr) <= 1:

return arr

mid = len(arr) // 2

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

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

return merge(left\_half, right\_half)

def merge(left, right):

result = []

i = j = 0

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

if left[i] <= right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

result.extend(left[i:])

result.extend(right[j:])

return result

1. Устойчивость алгоритма сортировки:

Сортировка слиянием является устойчивой.

Проанализируем: если элементы из левой и правой половины равны, алгоритм сначала выбирает элемент из левой половины, сохраняя порядок относительно элементов в правой половине.

1. Функция временной сложности для всех случаев:

Алгоритм сортировки слиянием делит массив на две части (каждая из которых имеет размер примерно n/2) и рекурсивно применяет сортировку к каждой половине. После этого сливаются две отсортированные части, что требует O(n) операций:

. Применяем мастер-теорему, после которой временная сложность алгоритма сортировки слиянием будем равна

Лучший случай: если входной массив уже отсортирован, сортировка слиянием всё равно разделяет его и выполняет слияние на всех уровнях рекурсии.

Худший случай: если массив будет отсортирован в обратном порядке, алгоритм выполняет такое же количество операций деления и слияния, как и в других случаях.

Средний случай: когда элементы массива расположены случайным образом, алгоритм так же, как и в остальных случаях выполняет слияние на всех уровнях рекурсии.

1. Асимптотическая оценка временной сложности для лучшего, среднего и худшего случая:

Лучший случай:

Худший случай:

Средний случай:

1. Функция пространственной сложности и её асимптотическую оценка:

Для сортировки слиянием требуется дополнительная память для временных массивов, в которые сливаются подмассивы.

В результате, на каждом уровне рекурсии создаются новые массивы для хранения элементов, что требует O(n) дополнительной памяти.

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

Изображение выглядит как текст, снимок экрана, число, линия

Автоматически созданное описание

## Сортировка Шелла (Shell Sort)

1. Асимптотическая временная и пространственная сложность:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Лучший случай | Худший случай | Средний случай |
| Временная сложность | O | O( |  |
| Пространственная сложность | O(1) | O(1) | O(1) |

1. Описание алгоритма сортировки :

На каждом шаге массив делится на несколько подмассивов с элементами, расположенными на фиксированном расстоянии друг от друга (шаг). Сначала шаг большой, а затем он постепенно уменьшается.

На каждом шаге элементы сортируются с использованием алгоритма вставок, но не для соседних элементов, а для элементов, которые находятся на расстоянии шага друг от друга.

После сортировки для текущего шага он уменьшается, и процесс повторяется для меньших значений шага, пока шаг не станет равным 1. Дальше алгоритм превращается в обычную сортировку вставками.

Алгоритм заканчивает работу, когда шаг становится равен 1 и после завершения последней сортировки вставками.

Код на Python:

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

1. Устойчивость алгоритма сортировки:

Сортировка Шелла является не устойчивой.

Проанализируем: на больших шагах элементы могут перемещаться далеко друг от друга, нарушая порядок одинаковых элементов. Пример: возьмем массив [7,3,7,2,7] с шагом 3. В начале поменяются элементы со значением 7 и 2: [2,3,7,7,7] (порядок одинаковых элементов изменен). Следовательно данная сортировка не является устойчивой.

1. Функция временной сложности для всех случаев:

Для классической последовательности с шагом (n/2, n/4, …,1):

Лучший случай: массив уже отсортирован, алгоритм ведет себя как сортировка вставками с оптимальной последовательностью шагов:

Худший случай: если массив будет отсортирован в обратном порядке, шаги будут уменьшаться слишком быстро, и элементы, которые находятся близко друг к другу, могут быть неэффективно отсортированы.

Средний случай: когда элементы массива расположены случайным образом, сложность варьируется в зависимости от распределения данных, но последовательность Шелла не всегда эффективна.

Для последовательности Хиббарда с шагом ((2k – 1): 1, 3, 7, 15, 31…):

Для последовательности Пратта с шагом ((,где i,j≥0): 1, 2, 3, 4, …):

1. Асимптотическая оценка временной сложности для лучшего, среднего и худшего случая при классической последовательности:

Лучший случай: O

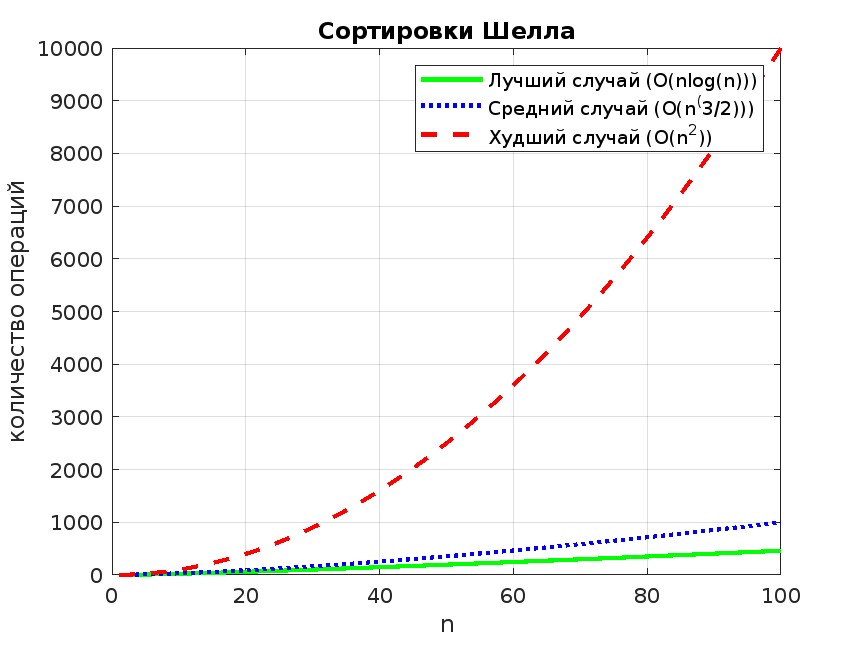
Худший случай:O

Средний случай

1. Функция пространственной сложности и её асимптотическую оценка:

Для сортировки Шелла не требуется дополнительная память, следовательно пространственная сложность - O(1).

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



## Быстрая сортировка (Quick Sort)

1. Асимптотическая временная и пространственная сложность:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Лучший случай | Худший случай | Средний случай |
| Временная сложность | O | O( |  |
| Пространственная сложность | O() | O() | O() |

1. Описание алгоритма сортировки :

Выбирается элемент из массива, который будет использоваться в качестве опорного для разделения массива на две части. Массив перестраивается таким образом, что все элементы меньше опорного элемента оказываются слева от него, а все элементы больше — справа. Опорный элемент, таким образом, оказывается на правильной позиции. После разделения массив фактически делится на две части (левая часть с элементами меньше опорного и правая часть с элементами больше опорного). Затем быстрая сортировка рекурсивно применяется к каждой из этих частей. Рекурсия завершается, когда подмассивы содержат один или ноль элементов, так как такие массивы уже отсортированы.

Код на Python:

def quicksort(a):

if len(a) <= 1:

return a

else:

pivot = a[len(a) - 1]

left = [x for x in a[:-1] if x <= pivot]

right = [x for x in a[:-1] if x > pivot]

return quicksort(left) + [pivot] + quicksort(right)

1. Устойчивость алгоритма сортировки:

Быстрая сортировка является не устойчивой.

Проанализируем: алгоритм использует операции обмена элементов в массиве для разделения его на две части. При этом могут быть перемещены одинаковые элементы, что нарушает их первоначальный порядок в массиве. Из-за того, что быстрая сортировка перемещает элементы в зависимости от опорного элемента и не заботится о сохранении порядка равных элементов, она неустойчива.

1. Функция временной сложности для всех случаев:

Лучший случай: если входной массив уже отсортирован, быстрая сортировка всё равно разделяет его и выполняет слияние на всех уровнях рекурсии.

Худший случай: возникает, когда каждый раз опорный элемент делит массив на очень неравные части. Например, одна из частей всегда оказывается пустой или содержит всего один элемент. В этом случае массив делится на части, где одна часть содержит n−1 элемент, а другая — пустая. Глубина рекурсии в худшем случае будет равна n, так как на каждом уровне остаётся только один новый элемент для дальнейшего деления.

Средний случай: обычно возникает, когда опорный элемент делит массив не идеально поровну, но и не слишком неравномерно. Даже при неидеальных разбиениях, глубина рекурсии будет логарифмической

1. Асимптотическая оценка временной сложности для лучшего, среднего и худшего случая:

Лучший случай:

Худший случай:

Средний случай:

1. Функция пространственной сложности и её асимптотическую оценка:

Быстрая сортировка имеет пространственную сложность O(), потому что при каждом делении массива рекурсия углубляется только на уровней.

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

Изображение выглядит как текст, снимок экрана, График, линия

Автоматически созданное описание

## Пирамидальная сортировка (HeapSort)

1. Асимптотическая временная и пространственная сложность:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Лучший случай | Худший случай | Средний случай |
| Временная сложность | O | O( |  |
| Пространственная сложность | O(1) | O(1) | O(1) |

1. Описание алгоритма сортировки :

Строится максимальная куча в массиве. Где каждый узел больше своих дочерних узлов. Дальше первый элемент в массиве меняется с последним. Дальше строится куча для последовательности элементов массива, не включая последний. Процесс повторяется для оставшихся элементов массива, пока все элементы не будут отсортированы.

Код на Python:

def heapify(arr, n, i):

largest = i

left = 2 \* i + 1

right = 2 \* i + 2

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

largest = left

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

largest = right

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i]

heapify(arr, n, largest)

def heap\_sort(arr):

n = len(arr)

for i in range(n // 2 - 1, -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)

1. Устойчивость алгоритма сортировки:

Пирамидальная сортировка является не устойчивой.

Проанализируем: Алгоритм пирамидальной сортировки основан на создании иерархической структуры данных — двоичной кучи. Элементы массива организуются в кучу, которая может нарушать относительный порядок одинаковых элементов.

1. Функция временной сложности для всех случаев:

Лучший случай: если массив уже отсортирован или почти отсортирован, сортировка всё равно будет строить кучу и восстанавливать её на каждом шаге.

Худший случай: когда массив отсортирован в обратном порядке или имеет самую неблагоприятную структуру для создания сбалансированной кучи, сортировка всё равно будет иметь сложность :

Средний случай: массив имеет произвольный порядок, так что построение кучи будет занимать O(n) операций. извлечение максимального элемента и восстановление кучи, занимает .

1. Асимптотическая оценка временной сложности для лучшего, среднего и худшего случая:

Лучший случай:

Худший случай:

Средний случай:

1. Функция пространственной сложности и её асимптотическую оценка:

Пирамидальная сортировка не использует дополнительную память для создания копий массива или хранения подмассивов, а работает с тем же массивом, что был подан на вход. Дополнительная память используется только для переменных внутри функций, и равна O(1).

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

Изображение выглядит как текст, снимок экрана, число, линия

Автоматически созданное описание

## Предположение о наибыстрейшей сортировки

\*Сортировки для среднего случая

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

Смотря на график, где изображены графики функций всех сортировок для среднего случая, можно сделать предположение, что самыми быстрыми сортировками будут сортировки слиянием, быстрая и пирамидальная сортировка.

# Практическая часть

## Сортировка выбором (Selection Sort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0.001254 | 0.000817 | 0.001055 | 0. 000815 |
| 2000 | 0.003114 | 0.001073 | 0.001547 | 0.001790 |
| 3000 | 0.004105 | 0.001663 | 0.002134 | 0.003235 |
| 4000 | 0.005513 | 0.003076 | 0.004024 | 0.004274 |
| 5000 | 0.007542 | 0.004561 | 0.005900 | 0.004780 |
| 6000 | 0.010712 | 0.006567 | 0.008449 | 0.007229 |
| 7000 | 0.014632 | 0.008876 | 0.013103 | 0.009212 |
| 8000 | 0.021829 | 0.011604 | 0.014829 | 0.011925 |
| 9000 | 0.026808 | 0.014719 | 0.018486 | 0.015040 |
| 10000 | 0.030868 | 0.018015 | 0.024159 | 0.01895 |
| 11000 | 0.037826 | 0.023363 | 0.030107 | 0.022747 |
| 12000 | 0.043532 | 0.025814 | 0.033047 | 0.026330 |
| 13000 | 0.052418 | 0.030157 | 0.038361 | 0.031208 |
| 14000 | 0.059402 | 0. 034843 | 0.045072 | 0.035490 |

1. Графики:

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

import java.util.Random;  
public class SelectionSort {  
 public static void selectionSort(int[] arr) {  
 int n = arr.length;  
 for (int i = 0; i < n - 1; i++) {  
 int minIdx = i;  
 for (int j = i + 1; j < n; j++) {  
 if (arr[j] < arr[minIdx]) {  
 minIdx = j;  
 }  
 }  
 int temp = arr[minIdx];  
 arr[minIdx] = arr[i];  
 arr[i] = temp;  
 }  
 }  
 public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 *selectionSort*(arr);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }  
 double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);

for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }  
 double ss\_time = *measureTime*(SortedArr);  
 Random random = new Random();  
 int[] RandomArray = new int[size];  
 for (int i = 0; i < size; i++) {  
 RandomArray[i] = random.nextInt(100000);  
 }  
 double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

## Сортировка вставками (Insertion Sort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0.000009 | 0.000237 | 0.002468 | 0.004647 |
| 2000 | 0.000008 | 0.000306 | 0.003332 | 0.001266 |
| 3000 | 0.000008 | 0.000379 | 0.004175 | 0.001151 |
| 4000 | 0.000011 | 0.000678 | 0.005447 | 0.001886 |
| 5000 | 0.000012 | 0.000689 | 0.005586 | 0.002441 |
| 6000 | 0.000013 | 0.000865 | 0.007860 | 0.003902 |
| 7000 | 0.000018 | 0.001450 | 0.009919 | 0.006169 |
| 8000 | 0.000019 | 0.001519 | 0.012881 | 0.006764 |
| 9000 | 0.000020 | 0.001929 | 0.020285 | 0.009177 |
| 10000 | 0.000021 | 0.002426 | 0.020266 | 0.011728 |
| 11000 | 0.000032 | 0.003878 | 0.026408 | 0.012658 |
| 12000 | 0.000043 | 0.003395 | 0.029430 | 0.015677 |
| 13000 | 0.000056 | 0.003830 | 0.037188 | 0.018269 |
| 14000 | 0.000163 | 0.004508 | 0.043114 | 0.020142 |

1. Графики:

Изображение выглядит как текст, снимок экрана, График, линия

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

import java.util.Random;  
public class InsertionSort {  
 public static void insertionSort(int[] arr) {  
 int n = arr.length;  
 for (int i = 1; i < n; i++) {  
 int key = arr[i];  
 int j = i - 1;  
 while (j >= 0 && arr[j] > key) {  
 arr[j + 1] = arr[j];  
 j = j - 1;  
 }  
 arr[j + 1] = key;  
 }  
 }  
 public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 *insertionSort*(arr);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }  
 double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);  
 for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }  
 double ss\_time = *measureTime*(SortedArr);  
 Random random = new Random();  
 int[] RandomArray = new int[size];  
 for (int i = 0; i < size; i++) {  
 RandomArray[i] = random.nextInt(100000);  
 }  
 double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

## Сортировка пузырьком (Bubble Sort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0. 0000029 | 0. 003261 | 0. 006313 | 0.008472 |
| 2000 | 0. 000001 | 0. 001420 | 0. 004305 | 0. 003996 |
| 3000 | 0. 000002 | 0. 003260 | 0. 009743 | 0. 007956 |
| 4000 | 0. 000002 | 0. 006164 | 0. 017316 | 0. 013519 |
| 5000 | 0.000003 | 0. 009566 | 0. 027547 | 0. 021314 |
| 6000 | 0.000004 | 0. 013316 | 0. 039204 | 0. 032232 |
| 7000 | 0.000004 | 0. 018897 | 0. 053498 | 0. 043563 |
| 8000 | 0.000004 | 0. 027003 | 0. 070469 | 0. 059688 |
| 9000 | 0.000007 | 0. 034056 | 0. 088904 | 0. 091830 |
| 10000 | 0.000012 | 0. 039090 | 0. 109600 | 0. 131424 |
| 11000 | 0.000008 | 0. 055372 | 0. 132782 | 0. 152201 |
| 12000 | 0.000011 | 0. 060844 | 0. 157830 | 0. 209016 |
| 13000 | 0.000006 | 0. 076952 | 0. 186904 | 0. 259033 |
| 14000 | 0.000006 | 0. 089155 | 0. 222801 | 0. 304515 |

1. Графики:

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

import java.util.Random;  
public class BubbleSort {  
 public static void BubbleSort(int[] arr) {  
 int n = arr.length;   
 for (int i = 0; i < n - 1; i++) {  
 boolean swapped = false;   
 for (int j = 0; j < n - 1 - i; j++) {  
 if (arr[j] > arr[j + 1]) {   
 int temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 swapped = true;  
 }  
 }   
 if (!swapped) {  
 break;  
 }  
 }  
 }  
 public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 *BubbleSort*(arr);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }  
 double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);  
 for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }

double ss\_time = *measureTime*(SortedArr);  
 Random random = new Random();  
 int[] RandomArray = new int[size];  
 for (int i = 0; i < size; i++) {  
 RandomArray[i] = random.nextInt(100000);  
 }  
 double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

## Сортировка слиянием (Merge Sort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0. 001010 | 0. 000158 | 0. 000147 | 0. 000261 |
| 2000 | 0. 000236 | 0. 000223 | 0. 000205 | 0. 000323 |
| 3000 | 0. 000383 | 0. 000340 | 0. 000281 | 0. 000498 |
| 4000 | 0. 000406 | 0. 000453 | 0. 000380 | 0. 000671 |
| 5000 | 0. 000537 | 0. 000634 | 0. 000491 | 0. 000861 |
| 6000 | 0. 000614 | 0. 000745 | 0. 000572 | 0. 001324 |
| 7000 | 0. 000709 | 0. 000915 | 0. 000715 | 0. 001295 |
| 8000 | 0. 000765 | 0. 001371 | 0. 001142 | 0. 001410 |
| 9000 | 0. 001256 | 0. 001573 | 0. 001296 | 0. 001619 |
| 10000 | 0. 001462 | 0. 001719 | 0. 001415 | 0. 004729 |
| 11000 | 0. 000969 | 0. 001322 | 0. 000941 | 0. 001699 |
| 12000 | 0. 001163 | 0. 001122 | 0. 000850 | 0. 001924 |
| 13000 | 0. 001035 | 0. 001667 | 0. 000945 | 0. 002061 |
| 14000 | 0. 001084 | 0. 001376 | 0. 001025 | 0. 003172 |

1. Графики

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, линия, снимок экрана, График

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

import java.util.Random;  
public class MergeSort {  
 public static void mergeSort(int[] array, int left, int right) {  
 if (left < right) {  
 int mid = (left + right) / 2 ;  
 *mergeSort*(array, left, mid);   
 *mergeSort*(array, mid + 1, right);   
 *merge*(array, left, mid, right);  
 }  
 }   
 public static void merge(int[] array, int left, int mid, int right) {   
 int n1 = mid - left + 1;  
 int n2 = right - mid;   
 int[] leftArray = new int[n1];  
 int[] rightArray = new int[n2];   
 System.*arraycopy*(array, left, leftArray, 0, n1);  
 System.*arraycopy*(array, mid + 1, rightArray, 0, n2);   
 int i = 0, j = 0, k = left;   
 while (i < n1 && j < n2) {  
 if (leftArray[i] <= rightArray[j]) {  
 array[k] = leftArray[i];  
 i++;  
 } else {  
 array[k] = rightArray[j];  
 j++;  
 }  
 k++;  
 }

while (i < n1) {  
 array[k] = leftArray[i];  
 i++;  
 k++;  
 }  
 while (j < n2) {  
 array[k] = rightArray[j];  
 j++;  
 k++;  
 }  
}

public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 *BubbleSort*(arr);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;  
}  
  
public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }  
 double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);  
 for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }

## Сортировка Шелла (Shell Sort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0. 000415 | 0. 000642 | 0. 000770 | 0. 000968 |
| 2000 | 0. 000911 | 0. 000278 | 0. 000182 | 0. 000354 |
| 3000 | 0. 000179 | 0. 000404 | 0. 000202 | 0. 000458 |
| 4000 | 0. 000190 | 0. 000586 | 0. 000280 | 0. 000820 |
| 5000 | 0. 000190 | 0. 000802 | 0. 000401 | 0. 000541 |
| 6000 | 0. 000329 | 0. 000565 | 0. 000247 | 0. 000815 |
| 7000 | 0. 000296 | 0. 000681 | 0. 000157 | 0. 000952 |
| 8000 | 0. 000183 | 0. 000788 | 0. 000214 | 0. 001127 |
| 9000 | 0. 000138 | 0. 000879 | 0. 000205 | 0. 001186 |
| 10000 | 0. 000153 | 0. 001017 | 0. 000230 | 0. 001215 |
| 11000 | 0. 000171 | 0. 001187 | 0. 000279 | 0. 001366 |
| 12000 | 0. 000196 | 0. 001303 | 0. 000295 | 0. 001530 |
| 13000 | 0. 000297 | 0. 001303 | 0. 000340 | 0. 001616 |
| 14000 | 0. 000210 | 0. 001583 | 0. 000485 | 0. 001770 |

1. Графики

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

import java.util.Random;  
public class ShellSort {  
 public static void shellSort(int[] array) {  
 for (int gap = array.length / 2; gap > 0; gap /= 2) {   
 for (int i = gap; i < array.length; i++) {  
 int temp = array[i];   
 int j = i;  
 while (j >= gap && array[j - gap] > temp) {  
 array[j] = array[j - gap];   
 j -= gap;  
 }  
 array[j] = temp;   
 }  
 }  
 }  
 public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 *insertionSort*(arr);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }  
 double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);  
 for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }  
 double ss\_time = *measureTime*(SortedArr);  
 Random random = new Random();  
 int[] RandomArray = new int[size];  
 for (int i = 0; i < size; i++) {  
 RandomArray[i] = random.nextInt(100000);  
 }  
 double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

## Сортировка Шелла – Последовательность Хиббарда (Shell Sort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0. 000392 | 0. 000669 | 0. 000479 | 0. 000834 |
| 2000 | 0. 000850 | 0. 000375 | 0. 000197 | 0. 000376 |
| 3000 | 0. 000188 | 0. 000424 | 0. 000206 | 0. 000505 |
| 4000 | 0. 000255 | 0. 000838 | 0. 000287 | 0. 000753 |
| 5000 | 0. 000320 | 0. 000595 | 0. 000401 | 0. 000560 |
| 6000 | 0. 000159 | 0. 000669 | 0. 000218 | 0. 000792 |
| 7000 | 0. 000230 | 0. 000715 | 0. 000367 | 0. 000823 |
| 8000 | 0. 000211 | 0. 000846 | 0. 000195 | 0. 001005 |
| 9000 | 0. 000121 | 0. 000987 | 0. 000186 | 0. 001119 |
| 10000 | 0. 000139 | 0. 001085 | 0. 000212 | 0. 001234 |
| 11000 | 0. 000158 | 0. 001208 | 0. 000246 | 0. 001403 |
| 12000 | 0. 000173 | 0. 001374 | 0. 000255 | 0. 001621 |
| 13000 | 0. 000190 | 0. 001756 | 0. 000307 | 0. 001919 |
| 14000 | 0. 000203 | 0. 001595 | 0. 000300 | 0. 001862 |

1. Графики

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

public class Shell\_HibbartSort {  
 public static void HibbardSort(int[] array) {  
 int n = array.length;  
 int k = 1;  
 while ((1 << k) - 1 < n) {  
 k++; }  
 for (int gap = (1 << (k - 1)) - 1; gap > 0; gap = (1 << (--k)) - 1) {  
 for (int i = gap; i < n; i++) {  
 int temp = array[i];  
 int j;  
 for (j = i; j >= gap && array[j - gap] > temp; j -= gap) {  
 array[j] = array[j - gap];  
 }  
 array[j] = temp;  
 } } }

public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 HibbardSort(arr);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }  
 double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);  
 for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }  
 double ss\_time = *measureTime*(SortedArr);  
 Random random = new Random();  
 int[] RandomArray = new int[size];  
 for (int i = 0; i < size; i++) {  
 RandomArray[i] = random.nextInt(100000);  
 }  
 double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

## Сортировка Шелла – Последовательность Пратта (Shell Sort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0. 003329 | 0. 000497 | 0. 002306 | 0. 000188 |
| 2000 | 0. 000489 | 0. 000566 | 0. 000409 | 0. 000228 |
| 3000 | 0. 000729 | 0. 000843 | 0. 000493 | 0. 000304 |
| 4000 | 0. 000654 | 0. 000928 | 0. 000621 | 0. 000410 |
| 5000 | 0. 000850 | 0. 001102 | 0. 000753 | 0. 000803 |
| 6000 | 0. 000975 | 0. 001832 | 0. 001351 | 0. 000909 |
| 7000 | 0. 001513 | 0. 001798 | 0. 000367 | 0. 000881 |
| 8000 | 0. 001369 | 0. 002062 | 0. 001257 | 0. 001332 |
| 9000 | 0. 001548 | 0. 002358 | 0. 001257 | 0. 001541 |
| 10000 | 0. 001780 | 0. 003389 | 0. 002466 | 0. 001284 |
| 11000 | 0. 002108 | 0. 002907 | 0. 002108 | 0. 001413 |
| 12000 | 0. 002156 | 0. 003550 | 0. 002486 | 0. 001584 |
| 13000 | 0. 002411 | 0. 003683 | 0. 002499 | 0. 001710 |
| 14000 | 0. 002665 | 0. 003851 | 0. 002716 | 0. 001890 |

1. Графики

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
  
public class Shell\_PrattSort {  
 public static void prattSort(int[] array) {  
 int n = array.length;  
 List<Integer> prattSequence = *generatePrattSequence*(n);  
 for (int gap : prattSequence) {  
 for (int i = gap; i < n; i++) {  
 int temp = array[i];  
 int j = i;  
 while (j >= gap && array[j - gap] > temp) {  
 array[j] = array[j - gap];  
 j -= gap;  
 }  
 array[j] = temp;  
 }  
 }  
 }  
 public static List<Integer> generatePrattSequence(int max) {  
 List<Integer> prattSequence = new ArrayList<>();  
 int p = 1, q = 1;  
 while (p <= max) {  
 q = p;  
 while (q <= max) {  
 prattSequence.add(q);  
 q \*= 3;  
 }  
 p \*= 2;  
 }  
 prattSequence.sort((a, b) -> b - a);  
 return prattSequence;  
 }

public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 prattSort(arr);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }

double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);  
 for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }  
 double ss\_time = *measureTime*(SortedArr);  
 Random random = new Random();  
 int[] RandomArray = new int[size];  
 for (int i = 0; i < size; i++) {  
 RandomArray[i] = random.nextInt(100000);  
 }  
 double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

## Быстрая сортировка (Quick Sort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0. 002918 | 0. 000072 | 0. 002306 | 0. 000057 |
| 2000 | 0. 001827 | 0. 000074 | 0. 000409 | 0. 000090 |
| 3000 | 0. 004085 | 0. 000169 | 0. 000493 | 0. 000141 |
| 4000 | 0. 007554 | 0. 000152 | 0. 000621 | 0. 000196 |
| 5000 | 0. 011320 | 0. 000237 | 0. 000753 | 0. 000247 |
| 6000 | 0. 016604 | 0. 000266 | 0. 001351 | 0. 000299 |
| 7000 | 0. 022144 | 0. 000459 | 0. 000367 | 0. 000357 |
| 8000 | 0. 028957 | 0. 000308 | 0. 001257 | 0. 000441 |
| 9000 | 0. 037623 | 0. 000368 | 0. 001257 | 0. 000465 |
| 10000 | 0. 045961 | 0. 000570 | 0. 002466 | 0. 000526 |
| 11000 | 0. 054710 | 0. 000457 | 0. 002108 | 0. 000580 |
| 12000 | 0. 065155 | 0. 000563 | 0. 002486 | 0. 000656 |
| 13000 | 0. 076476 | 0. 000519 | 0. 002499 | 0. 000692 |
| 14000 | 0. 088596 | 0. 000567 | 0. 002716 | 0. 000741 |

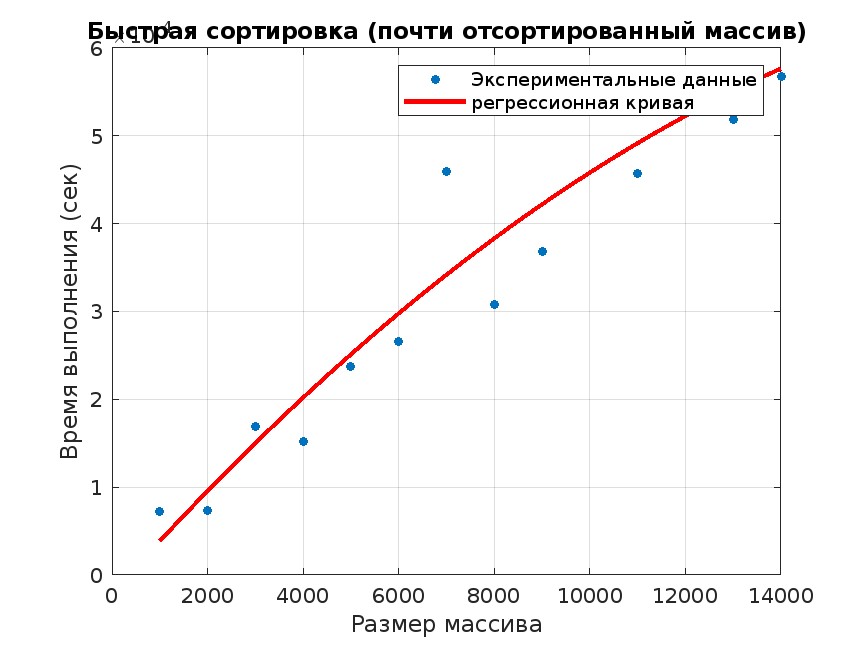
1. Графики:

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание



Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

import java.util.Random;  
public class QuickSort {  
 public static void quick\_Sort(int[] array, int low, int high) {  
 if (low < high) {  
 int pivotIndex = *partition*(array, low, high);  
 *quick\_Sort*(array, low, pivotIndex - 1);  
 *quick\_Sort*(array, pivotIndex + 1, high);  
 }  
 }  
 public static int partition(int[] array, int low, int high) {  
 int pivot = array[high];  
 int i = low - 1;  
 for (int j = low; j < high; j++) {  
 if (array[j] <= pivot) {  
 i++;  
 int temp = array[i];  
 array[i] = array[j];  
 array[j] = temp;  
 }  
 }  
 int temp = array[i + 1];  
 array[i + 1] = array[high];  
 array[high] = temp;  
 return i + 1;  
 }  
 public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 *quick\_Sort*(arr,0,arr.length - 1);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;  
 }  
 public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }

double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);

double r\_time = *measureTime*(ReverseSortedArr);  
 for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }  
 double ss\_time = *measureTime*(SortedArr);  
 Random random = new Random();  
 int[] RandomArray = new int[size];  
 for (int i = 0; i < size; i++) {  
 RandomArray[i] = random.nextInt(100000);  
 }  
 double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

## Пирамидальная сортировка (HeapSort)

1. Таблица времени сортировки:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Отсортированный массив | Почти отсортированный массив | Обратно отсортированный массив | Случайный массив |
| 1000 | 0. 001107 | 0. 000130 | 0. 000161 | 0. 000136 |
| 2000 | 0. 000191 | 0. 000188 | 0. 000194 | 0. 000220 |
| 3000 | 0. 000271 | 0. 000283 | 0. 000278 | 0. 000335 |
| 4000 | 0. 000368 | 0. 000386 | 0. 000372 | 0. 000433 |
| 5000 | 0. 000494 | 0. 000267 | 0. 000288 | 0. 000332 |
| 6000 | 0. 000287 | 0. 000322 | 0. 000311 | 0. 000401 |
| 7000 | 0. 000336 | 0. 000383 | 0. 000348 | 0. 000488 |
| 8000 | 0. 000430 | 0. 000497 | 0. 000459 | 0. 000592 |
| 9000 | 0. 000489 | 0. 000543 | 0. 000534 | 0. 000671 |
| 10000 | 0. 000541 | 0. 000612 | 0. 000770 | 0. 000836 |
| 11000 | 0. 000676 | 0. 000646 | 0. 000645 | 0. 000580 |
| 12000 | 0. 000633 | 0. 000687 | 0. 000628 | 0. 000845 |
| 13000 | 0. 000642 | 0. 000730 | 0. 000680 | 0. 000954 |
| 14000 | 0. 000679 | 0. 000784 | 0. 000754 | 0. 001019 |

1. Графики:

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, График, линия, снимок экрана

Автоматически созданное описание

1. Код для расчёта времени сортировки на Java:

import java.util.Random;  
public class HeapSort {  
 public class heap\_Sort {  
 public static void heapsort(int[] array) {  
 int n = array.length;  
 for (int i = n / 2 - 1; i >= 0; i--) {  
 *heapify*(array, n, i);  
 }  
 for (int i = n - 1; i >= 0; i--) {  
 int temp = array[0];  
 array[0] = array[i];  
 array[i] = temp;  
 *heapify*(array, i, 0);  
 }  
 }  
 public static void heapify(int[] array, int n, int i) {  
 int largest = i;  
 int left = 2 \* i + 1;  
 int right = 2 \* i + 2;  
 if (left < n && array[left] > array[largest]) {  
 largest = left;  
 }  
 if (right < n && array[right] > array[largest]) {  
 largest = right;  
 }  
 if (largest != i) {  
 int swap = array[i];  
 array[i] = array[largest];  
 array[largest] = swap;  
 *heapify*(array, n, largest);  
 }  
 }  
 }  
 public static double measureTime(int[] arr) {  
 long startTime = System.*nanoTime*();  
 heap\_Sort.*heapsort*(arr);  
 long endTime = System.*nanoTime*();  
 return (endTime - startTime)/ 1e9;

}  
 public static void main(String[] args) {  
 System.*out*.println("Размер массива\t\t\t\tВремя (с)");  
 for (int size = 1000; size<15000;size+=1000 ) {  
 int[] arr = new int[size];  
 int[] SortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 SortedArr[i] = i;  
 }

double s\_time = *measureTime*(SortedArr);  
 int[] ReverseSortedArr = new int[size];  
 for (int i = 0; i < size; i++) {  
 ReverseSortedArr[i] = size - i;  
 }  
 double r\_time = *measureTime*(ReverseSortedArr);  
 for (int i = 0; i < size / 10; i++) {  
 int index1 = new Random().nextInt(size);  
 int index2 = new Random().nextInt(size);  
 int temp = SortedArr[index1];  
 SortedArr[index1] = SortedArr[index2];  
 SortedArr[index2] = temp;  
 }  
 double ss\_time = *measureTime*(SortedArr);  
 Random random = new Random();  
 int[] RandomArray = new int[size];  
 for (int i = 0; i < size; i++) {  
 RandomArray[i] = random.nextInt(100000);  
 }  
 double ra\_time = *measureTime*(RandomArray);  
 System.*out*.printf("%-20d %-20f %-20f %-20f %-20f%n", size, s\_time, r\_time, ss\_time, ra\_time);  
 }  
 }  
}

## Сравнение теоретических и практических результатов

1. Сортировка выбором (Selection Sort)

Изображение выглядит как текст, снимок экрана, График, линия

Автоматически созданное описание

1. Сортировка вставками (Insertion Sort)

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

1. Сортировка пузырьком (Bubble Sort)

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

1. Сортировка слиянием (Merge Sort)

Изображение выглядит как текст, линия, снимок экрана, График

Автоматически созданное описание

1. Сортировка Шелла (Shell Sort)

Изображение выглядит как текст, линия, снимок экрана, График

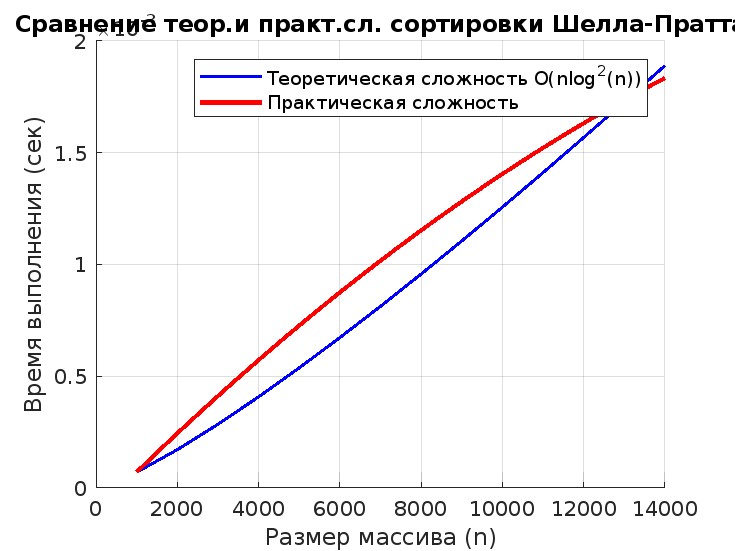
Автоматически созданное описание

1. Сортировка Шелла – последовательность Хиббарда (Shell Sort)

Изображение выглядит как текст, линия, снимок экрана, График

Автоматически созданное описание

1. Сортировка Шелла – последовательность Пратта (Shell Sort)



1. Быстрая сортировка (Quick Sort)

Изображение выглядит как текст, линия, снимок экрана, График

Автоматически созданное описание

1. Пирамидальная сортировка (Heap Sort)

Изображение выглядит как текст, линия, снимок экрана, График

Автоматически созданное описание

## Сравнение практических результатов сортировок

Изображение выглядит как текст, снимок экрана, линия, График

Автоматически созданное описание

Опираясь на график, где показаны все функции сложности для среднего случая, можно сделать вывод, что быстрее всего будут работать сортировки Шелла с последовательностью Хиббарда и Пратта, а также быстрая и пирамидальная сортировки. Результат чуть разнится с нашим предположением в теории, но это может быть связано, что на практике было обработано меньшее количество элементов, и с их возрастанием сортировка слиянием и правда будет становится быстрее сортировок Шелла.

# Вывод

Посмотрев наглядно, как и с какой скоростью работают сортировки сравнением, можно сделать вывод, что теоретическая оценка сложности — это абстрактная модель, которая показывает, как алгоритм ведет себя при увеличении размера задачи. Она игнорирует многие аспекты практического выполнения, поэтому допустимо расхождение между теорией и практикой. Но сравнив результаты практики и предположения теории, мы выяснили, что наибыстрейшими сортировками являются пирамидальная и быстрая.