**Федеральное государственное бюджетное образовательное учреждение   
высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

**Анализ алгоритмов**

**Лабораторная работа №4**

**Отчёт на тему:**

# «Сортировка массивов»

Выполнила:

Янова Даниэлла

ИУ7-53

**Москва, 2018**

**Введение**

**Цель лабораторной работы**: изучение алгоритмов сортировки массивов, сравнительный анализ времени работы данных алгоритмов, анализ трудоемкости алгоритмов.

**Задачи лабораторной работы:**

* реализация следующих алгоритмов сортировки массивов – сортировка простыми вставками, patience sorting, плавная сортировка (smoothsort);
* оценка трудоемкости алгоритмов;
* анализ времени работы программы;
* сравнительный анализ работы алгоритмов для массивов размера от 100 до 1000 элементов.

1. **Аналитическая часть**

В данном разделе приведены алгоритмы, а также составлена модель для вычисления трудоемкости.

**1.1 Описание алгоритмов**

**Терпеливая сортировка:** имеется массив a[0...n-1], который нужно отсортировать. Для этого элементы массива раскладываются по стопкам следующим образом: если текущий элемент больше вершины стопки, мы кладем его в новую стопку, если меньше, то на вершину стопки. Элемент кладется в самую левую стопку из возможных. Далее стрjится массив стопок, после чего из всех вершин стопок выбирается минимальный из элементов. Действие повторяется, пока все элементы не будут выбраны.

**Сортировка простыми вставками:** это алгоритм, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. На вход алгоритма подаётся последовательность n чисел: M . Сортируемые числа также называют ключами. Входная последовательность на практике представляется в виде массива с n элементами. На выходе алгоритм должен вернуть перестановку исходной последовательности M , чтобы выполнялось следующее соотношение M . В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элементы удовлетворяют требованиям к выходным данным алгоритма.

**Плавная сортировка:** Сортируемый массив делится на группу подмассивов. Каждый подмассив представляет собой структуру данных куча. Каждая куча имеет размер равный одному из чисел Леонардо [3].

Числа Леонардо вычисляются по формуле:

L(0) = 1;

L(1) = 1;

L(n) = L(n-1) + L(n-2) + 1;

При этом левые элементы массива являются узлами самой большой возможной кучи. Оставшиеся элементы массива разбиваются на кучи по такому же жадному правилу. В дальнейшем эту группу подмассивов будем называть последовательность куч. При этом, элементы этой последовательности будут строго уменьшаться в размере с ростом порядкового номера. Не существует двух куч, имеющих одинаковый размер. Это утверждение можно доказать так: Числа Леонардо растут медленнее функции 2^N, поэтому для любого массива можно найти такое число Леонардо, которое будет больше середины. Исключением является массив длины 2. Никакие две кучи не будут иметь размер равным двум последовательным числам Леонардо. Исключением могут быть только две последние кучи. Если мы будем использовать подряд две кучи размерностью L(x) и L(x+1), то их

можно будет заменить одной – размерностью L(x+2)

Следствия:

* 1. В вершине каждой кучи находится максимальный из узлов данной кучи.
  2. Максимальный элемент массива нужно искать среди вершин сформированных куч.

Краткое описание алгоритма:

В начале последовательность куч пуста. Последовательно добавляем элементы из исходного массива слева направо. Элемент, который будет добавляться в последовательность куч, выступает либо в качестве вершины абсолютно новой кучи, либо в качестве вершины, которая объединит две последние кучи. Повторяем эту итерацию пока не закончатся элементы во входном массиве.

На каждой итерации определяем текущий максимум массива, используя следствие 2. Ставим максимальный элемент на последнее место и исключаем его из последовательности куч. Повторяем это действие, пока последовательность куч не окажется пуста. При объединении двух куч в одну и при обмене двух вершинных элементов нужно гарантировать сохранение свойства кучи, т.е. выполнять “просейку вниз” при необходимости.

**2**. **Конструкторская часть**

В данном разделе представлены псевдокоды алгоритмов.

**Терпеливая сортировка**

Patience\_sort:

* считать массив
* пока есть элементы массива
  + берем крайний левый элемент
  + ищем первую стопку слева, в которой вершина больше текущего элемента и кладем его туда
  + если не находим, то
    - создаем новую стопку и кладем элемент на вершину
  + для элемента создаем указатель на вершину стопки слева
* находим наибольшую возрастающую последовательность

**Сортировка простыми вставками**

* считать массив
* начать цикл от первого (индексирование ведется с нуля) до последнего элемента
* сохраняем в дополнительной переменной temp нынешний элемент и в j индекс предыдущего до нынешнего элемента
* начать цикл пока j >=0 и j > temp, то
* присваиваем элементу с индексом j+1 значение элемента с индексом j, а элементу с индексом j значение temp
* уменьшаем значение j на 1
* конец цикл пока
* конец цикла от первого до последнего элемента

**Плавная сортировка**

* считываем массив
* начать цикл от нулевого до последнего элемента
* добавляем первые два элемента в дерево
* берем элемент массива, сравниваем с верхними вершинами дерева. Если одна из них больше нынешнего элемента – меняем местами
* аналогично проверяем нижние уровни дерева, чтобы вершины были больше нижних уровней
* конец цикла от последнего до нулевого элемента
* начать цикл от последнего до нулевого элемента
* найти максимальную по значению вершину дерева
* сравнить вершину с i-м элементом массива. Если не равны, то
* присвоить значению элемента массива значение вершины дерева
* удалить нынешнюю вершину дерева
* конец цикла от последнего до нулевого элемента

1. **Технологическая часть**

В данном разделе представлена реализация алгоритмов, указан язык программирования, а также необходимые модули.

* 1. **Средства реализации**

Для выполнения данной лабораторной работы использовался язык Python 3.7.1 в среде Pycharm. Замены времени проводились с использованием функции process\_time\_ns, входящей в библиотеку time Python версии 3.7.

* 1. **Листинг кода**

1. **def** max\_elem(array):
2. m = array[0]
3. **for** i **in** range(1, len(array)):
4. **if** array[i] > m:
5. **m = array[i]**
6. **return** m
8. **def** patience\_sort(seq):
9. N = len(seq)
10. **M = [0] \* N**
11. **for** i **in** range(N):
12. M[i] = [0] \* (N + 1)
13. G = max\_elem(seq)
15. **i = j = k = 0**
17. M[i][j+1] = seq[k]
18. **for** k **in** range (1, N):
19. **if** M[i][j+1] > seq[k]:
20. **j += 1**
21. M[i][j+1] = seq[k]
22. **else**:
23. M[i][0] = j + 1
24. i += 1
25. **j = 0**
26. M[i][j+1] = seq[k]
27. M[i][0] = j + 1
29. **for** k **in** range(0, N):
30. **min\_el = G**
31. **for** j **in** range(0, i+1):
32. **if** M[j][0] != 0:
33. **if** min\_el > M[j][M[j][0]]:
34. min\_el = M[j][M[j][0]]
35. **s = j**
36. seq[k] = min\_el
37. M[s][0] -= 1
39. **return** seq

Листинг 1.1 – Терпеливая сортировка

1. **def** insertion\_sort(seq):
3. **for** i **in** range (1, len(seq)):
4. temp = seq[i]
5. **item = i - 1**
7. **while** ((item >= 0) **and** (seq[item] > temp)):
8. seq[item + 1] = seq[item]
9. seq[item] = temp
10. **item = item - 1**
12. **return** seq

Листинг 1.2: Сортировка простыми вставками.

1. **def** isAscending(v1, v2):
2. """Comparator function"""
3. **return** v1 <= v2

6. **def** up(vb, vc):
7. temp = vb
8. vb += vc + 1
9. vc = temp
10. **return vb, vc**

13. **def** down(vb, vc):
14. temp = vc
15. **vc = vb - vc - 1**
16. vb = temp
17. **return** vb, vc

20. **def smoothsort(A):**
21. """The main SmoothSort function
22. Variables: q,r,p,b,c,r1,b1,c1,N
23. """
25. **def sift():**
26. **nonlocal** r1, b1, c1
27. r0 = r1
28. T = A[r0]
29. **while** b1 >= 3:
30. **r2 = r1 - b1 + c1**
32. **if** **not** isAscending(A[r1 - 1], A[r2]):
33. r2 = r1 - 1
34. b1, c1 = down(b1, c1)
35. **if isAscending(A[r2], T):**
36. b1 = 1
37. **else**:
38. A[r1] = A[r2]
39. r1 = r2
40. **b1, c1 = down(b1, c1)**
41. **if** r1 != r0:
42. A[r1] = T
44. **def** trinkle():
45. **nonlocal p, b1, c1, b, c, r1**
46. p1 = p
47. b1 = b
48. c1 = c
49. r0 = r1
50. **T = A[r0]**
51. **while** p1 > 0:
52. **while** (p1 & 1) == 0:
53. p1 >>= 1
54. b1, c1 = up(b1, c1)
55. **r3 = r1 - b1**
56. **if** p1 == 1 **or** isAscending(A[r3], T):
57. p1 = 0
58. **else**:
59. p1 -= 1
60. **if b1 == 1:**
61. A[r1] = A[r3]
62. r1 = r3
63. **elif** b1 >= 3:
64. r2 = r1 - b1 + c1
65. **if not isAscending(A[r1 - 1], A[r2]):**
66. r2 = r1 - 1
67. b1, c1 = down(b1, c1)
68. p1 <<= 1
69. **if** isAscending(A[r2], A[r3]):
70. **A[r1] = A[r3]**
71. r1 = r3
72. **else**:
73. A[r1] = A[r2]
74. r1 = r2
75. **b1, c1 = down(b1, c1)**
76. p1 = 0
77. **if** r0 != r1:
78. A[r1] = T
79. sift()
81. **def** semitrinkle():
82. **nonlocal** r1, r, c
83. r1 = r - c
84. **if** **not** isAscending(A[r1], A[r]):
85. **A[r], A[r1] = A[r1], A[r]**
86. trinkle()
88. *# Start of main function*
89. N = len(A)
90. **q = 1**
91. r = 0
92. p = 1
93. b = 1
94. c = 1
95. ***#building the tree***
96. **while** q < N:
97. r1 = r
99. **if** (p & 7) == 3:
100. **b1 = b**
101. c1 = c
102. sift()
103. p = (p + 1) >> 2
104. b, c = up(b, c)
105. **b, c = up(b, c)**
106. **elif** (p & 3) == 1:
107. **if** (q + c) < N:
108. b1 = b
109. c1 = c
110. **sift()**
111. **else**:
112. trinkle()
113. b, c = down(b, c)
114. p <<= 1
115. **while b > 1:**
116. b, c = down(b, c)
117. p <<= 1
118. p += 1
119. q += 1
120. **r += 1**
122. r1 = r
123. trinkle()
125. ***#build the sorted array***
126. **while** q > 1:
127. q -= 1
128. **if** b == 1:
129. r -= 1
130. **p -= 1**
131. **while** (p & 1) == 0:
132. p >>= 1
133. b, c = up(b, c)
134. **elif** b >= 3:
135. **p -= 1**
136. r = r - b + c
137. **if** p > 0:
138. semitrinkle()
139. b, c = down(b, c)
140. **p = (p << 1) + 1**
141. r += c
142. semitrinkle()
143. b, c = down(b, c)
144. p = (p << 1) + 1
145. ***# element q is done***
146. *# element 0 is done*

Листинг 1.3: Плавная сортировка.

* 1. **Оценка трудоемкости алгоритмов**

*Терпеливая сортировка* имеет трудоемкость в лучшем случае , в худшем случае – [1].

*Сортировка простыми вставками.*

Время работы алгоритма сортировки вставками — это сумма времён работы каждого шага:

Самым благоприятным случаем является отсортированный массив. При этом все внутренние циклы состоят всего из одной итерации, то есть M для всех j. Тогда время работы алгоритма составит

Время работы линейно зависит от размера входных данных.

Наихудшим случаем является массив, отсортированный в порядке, обратном нужному. При этом каждый новый элемент сравнивается со всеми в отсортированной последовательности. Это означает, что все внутренние циклы состоят из j итераций, то есть M для всех j. Тогда время работы алгоритма составит:

.

Время работы является квадратичной функцией от размера входных данных.

*Плавная сортировка*:

Алгоритм сортировки Smoothsort работает в два этапа:

* 1. Добавляя по одному, строится множество куч Леонардо для всех элементов со сложностью не более O(log n) [3];
  2. По очереди извлекаются максимальные элементы из построенной структуры (максимумом всегда будет корень самой правой кучи) и структура восстанавливается необходимым образом со сложностью не более O(log n).

Преимуществом использования такой структуры является то, что в случае уже отсортированной входной последовательности, операции построения куч и извлечения всех элементов будут иметь суммарную сложность O(n). И утверждается, что при уменьшении степени упорядоченности данных, которые следует отсортировать, сложность будет плавно(smoothly) возрастать от O(n) до O(n log n).

\* Лучший случай – O(n) сравнений, если числа упорядочены по возрастанию [3];

\* Худший случай – O(n log n) сравнений [3].

1. **Экспериментальная часть**

В данном разделе будут произведены замеры времени работы алгоритмов. Для исследования скоростных характеристик был использован компьютер на базе процессора Intel Core i7-4500U, содержащий 8 гигабайт оперативной памяти. Модуль тестирования запускался с жесткого диска под операционной системой Windows.

**4.1 Исследование скорости работы алгоритмов**

Для каждого алгоритма производятся замеры времени в секундах для массивов размерности от 500 до 5000 с шагом 500. Используются три типа массивов: отсортированные по возрастанию, отсортированные по убыванию и массивы, заполненные случайным образом. Для всех типов массивов эксперимент проводится 100 раз, время в графиках приводится среднее из 100 экспериментов.

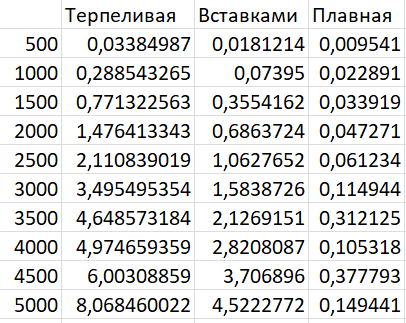


Таблица 1 – таблица времени выполнения сортировок случайно заполненного массива.

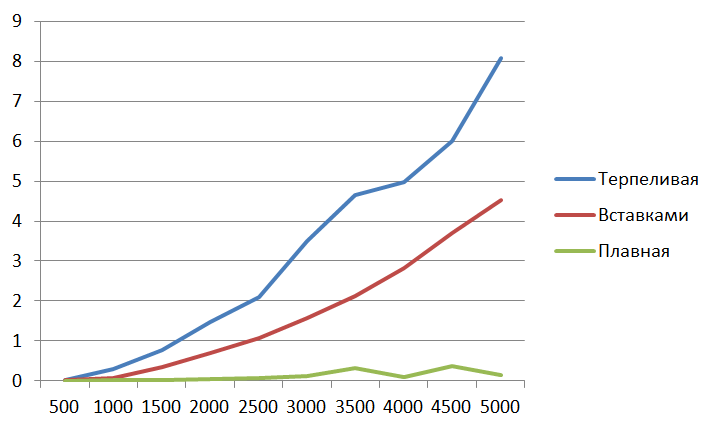


Рис.1 – график времени сортировки разными методами массива, заполненного в случайном порядке



Таблица 2 - таблица времени выполнения сортировок массива, заполненного по возрастанию.

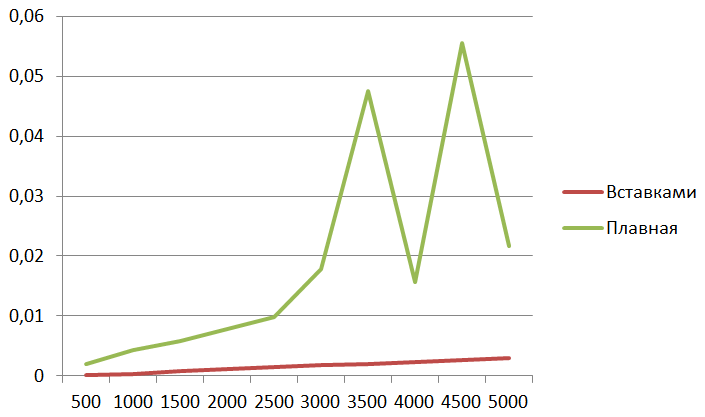


Рис.2-1 - график времени сортировки вставками и плавной сортировкой массива, заполненного по возрастанию

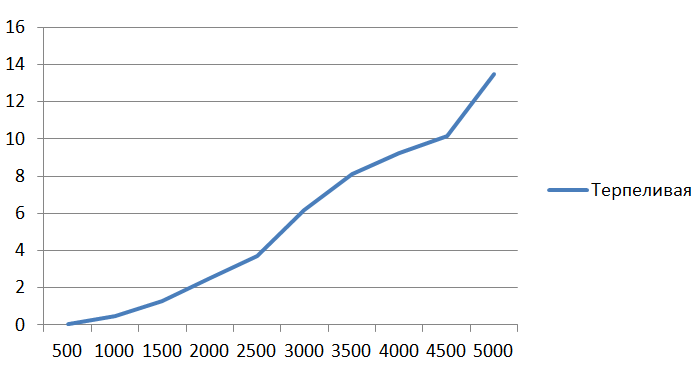


Рис.2-2 - график времени терпеливой сортировки массива, заполненного по возрастанию.



Таблица 3 - таблица времени выполнения сортировок массива, заполненного по убыванию.

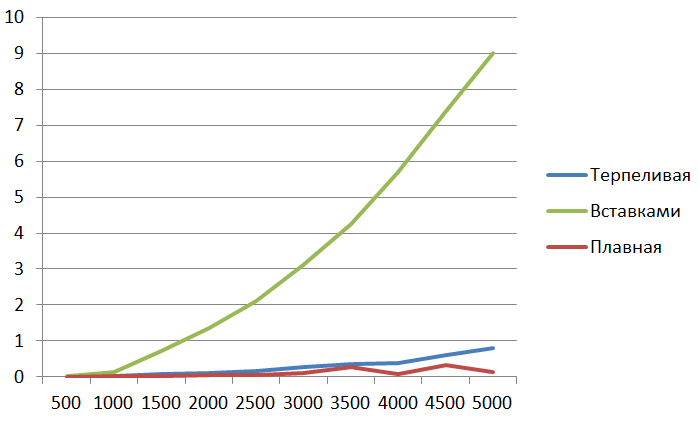


Рис.3-1 – график времени сортировки разными методами массива, заполненного по убыванию

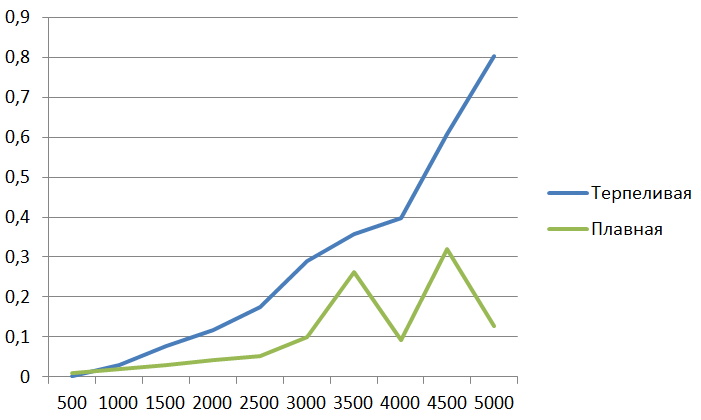


Рис.3-2 – более подробный график времени терпеливой и плавной сортировки, заполненного по убыванию

По графику на рис.1 видно, что медленнее всего работает терпеливая сортировка, она дольше сортировки вставками в 2 раза. При этом плавная сортировка остается на примерно одном и том же уровне времени, когда остальные монотонно возрастают.

По таблицам значений замеров времени заметно, что плавная сортировка при любом виде заполнения массива всегда работает в среднем меньше секунды, хотя используются массивы огромного количества элементов, а другие сортировки работают в среднем по 8-10 секунд. Это определенный плюс для плавной сортировки – всегда работает быстро.

Также стоит отметить, что для сортирования массива имеет большое значение, как массив изначально будет заполнен: по возрастанию или убыванию. Сортировка со вставками очень быстро работает с массивами, заполненными по возрастанию, и очень медленно с массивами, заполненными по убыванию. Терпеливая сортировка же наоборот – очень медленно работает с возрастающим массивом и очень быстро с убывающим.

Дело в работе сортировок. Терпеливая сортировка создает кучи с элементами, где вершина является минимальным элементом в данной стопке. То есть идет отбор по минимальным элементам, которые заносятся в вершины стопок, из-за чего потом легко доставать первые элементы для отсортированного возрастающего массива. В случае с убывающим массивом будет создана только одна стопка, что заметно облегчает работу сортировки, но в случае с возрастающим массивом будет наоборот – будет создано столько стопок, сколько находится элементов в массиве, а это заметно нагружает работу сортировки.

Сортировка вставками же при возрастающем массиве будет работать очень быстро, так как при проходе по массиву ей не придется ничего вставлять, двигать значения элементов, то есть будет только один проход по массиву. При убывающем же массиве сортировка будет каждый раз вставлять элемент и двигать остальную часть массива. Поэтому терпеливая сортировка быстрая при убывающем массиве, а сортировка с вставками – на возрастающем.

* 1. **Вывод**

Из экспериментальных данных видно, что самой проверенной на постоянную высокую скорость является плавная сортировка, как на небольших объемах массива, так и на огромных.

**Заключение**

В данной лабораторной работе вычислена трудоемкость алгоритмов сортировки массивов. Для каждого из алгоритмов была реализована программа, проведены замеры времени, произведен сравнительный анализ алгоритмов.

**Список литературы**

1. [https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8](https://neerc.ifmo.ru/wiki/index.php?title=Сортировки)
2. **http://algolab.valemak.com/gnome**
3. **Анализ алгоритмов. Дж. Макконнелл, 2004**