

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теории функций и стохастического анализа

ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ (БАЗОВОЙ) ПРАКТИКЕ

студента 4 курса 451 группы
направления 38.03.05 — Бизнес-информатика

механико-математического факультета
Чайковского Петра Ильича

Место прохождения: завод "Тантал"

Сроки прохождения: с 29.06.2019 г. по 26.07.2019 г.

Оценка:

Руководитель практики от СГУ

доцент, к. ф.-м. н.

Н. Ю. Агафонова

Руководитель практики от организации

ведущий программист

Д. Э. Кнутов

Саратов 2022

Тема практики: «Правила оформления курсовых и дипломных работ»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	4
1 Постановка задачи. Описание метода её решения.....	5
2 Вычисление сложности алгоритма.....	7
3 Программа решения на языке C++.	8
ЗАКЛЮЧЕНИЕ	12

ВВЕДЕНИЕ

Главное требование при разработке алгоритмов сортировки массивов - экономное использование доступной оперативной памяти. Это означает, что перестановки, с помощью которых упорядочиваются элементы, должны выполняться без затрат дополнительной временной памяти. Ограничивая методы сортировок таким образом, попробуем классифицировать их в соответствии с их временной эффективностью, выражающейся числом перестановок элементов M или числом необходимых сравнений ключа C . Такая временная эффективность является функцией числа сортируемых элементов n . Хотя хорошие алгоритмы сортировки ограничиваются порядком $n \cdot \log(n)$ сравнений, наиболее простые требуют порядка n^2 сравнений ключей. Рассмотрим один из наиболее простых алгоритмов сортировки - сортировку пузырьком.

1 Постановка задачи. Описание метода её решения.

Среди алгоритмов сортировки сортировка пузырьком занимает своё место как один из наиболее простых алгоритмов. Как и в других несложных алгоритмах, здесь выполняются повторные проходы по массиву, причём каждый раз наименьший элемент оставшегося множества просеивается в направлении одного из концов массива (в зависимости от того, происходит ли сортировка по убыванию или по возрастанию). Своё название алгоритм получил как раз из-за схожести процесса подъёма пузырьков воздуха в сосуде с водой с просеиванием элемента к одному из концов массива. Рассмотрим код "наивной" реализации алгоритма сортировки пузырьком:

```
1  PROCEDURE BubbleSort;  
2    VAR i, j: INTEGER; x: Item;  
3  BEGIN  
4    FOR i := 1 TO n - 1 DO  
5      FOR j := n - 1 TO i BY -1 DO  
6        IF a[j - 1] > a[j] THEN  
7          x := a[j - 1]; a[j - 1] := a[j]; a[j] := x  
8        END  
9      END  
10   END  
11 END BubbleSort
```

Листинг 1: Алгоритм сортировки пузырьком.

Этот алгоритм нетрудно улучшить. Рассмотрение примеров показывает, что в некоторых случаях алгоритм продолжает работу при уже отсортированном массиве. Очевидный способ улучшить алгоритм - запоминать, был ли осуществлён хотя бы один обмен во время прохода. Тогда проход без обменов будет означать, что алгоритм может быть остановлен. Возможно также ещё одно улучшение - запоминать не только факт обмена, но и позицию (индекс) последнего обмена. Ясно, что все пары соседних элементов левее этого значения индекса уже упорядочены, поэтому последующие проходы могут останавливаться на этом значении индекса, вместо того, чтобы продолжаться до i . Также возможно и третье усовершенствование: менять направление последовательных проходов. Получившийся алгоритм называ-

ется "шейкер-сортировкой" и является модификацией алгоритма сортировки пузырьком. Рассмотрим код этого алгоритма:

```
1  PROCEDURE ShakerSort;  
2    VAR j, k, L, R: INTEGER; x: Item;  
3  BEGIN  
4    L := 1; R := n - 1; k := R;  
5    REPEAT  
6      FOR j := R TO L BY -1 DO  
7        IF a[j - 1] > a[j] THEN  
8          x := a[j - 1]; a[j - 1] := a[j]; a[j] := x; k := j;  
9        END  
10     END  
11     L := k + 1;  
12     FOR j := L TO R BY +1 DO  
13       IF a[j - 1] > a[j] THEN  
14         x := a[j - 1]; a[j - 1] := a[j]; a[j] := x; k := j  
15       END  
16     END  
17     R := k - 1  
18   UNTIL L > R  
19 END ShakerSort
```

Листинг 2: Алгоритм шейкер-сортировки.

2 Вычисление сложности алгоритма.

Оценим число сравнений в простой пузырьковой сортировке. В среднем оно будет равно $C = \frac{n^2 - n}{2}$, а минимальное, среднее и максимальное числа присваиваний элементов равны:

$$M_{min} = 0, \quad M_{avg} = \frac{3}{4}(n^2 - n), \quad M_{max} = \frac{3}{2}(n^2 - n).$$

Анализ улучшенных вариантов, особенно шейкер-сортировки, довольно сложен. Наименьшее число сравнений здесь равно $C_{min} = n - 1$. Для улучшенной пузырьковой сортировки Дональд Кнут нашёл, что среднее число проходов пропорционально величине $n - k_1\sqrt{n}$, а среднее число сравнений - величине: $(n^2 - n \cdot (k_2 + \ln n))/2$. Однако ни одно из упомянутых улучшений не может повлиять на число обменов, уменьшается только число избыточных проверок, поэтому все улучшения имеют меньший эффект, чем могло ожидатьс - обмен двух элементов более затратная, нежели сравнение ключей, операция.

Подобный анализ показывает, что сортировка пузырьком и её небольшие улучшения хуже, чем сортировка вставками и сортировка выбором, поэтому является малоэффективным алгоритмом сортировки. Шейкер-сортировка эффективна в тех случаях, когда элементы уже стоят в почти правильном порядке.

3 Программа решения на языке C++.

По полученным псевдокодам напишем программу на языке программирования C++. Программа будет содержать реализацию алгоритма сортировки пузырьком и шейкерной сортировки.

```
1 #include<iostream>
2 #include<vector>
3
4 using namespace std;
5
6 const int numberOfValues = 10;
7 const int divisionValue = 97;
8
9 void generateVector(vector<int>& values)
10 {
11     int i;
12
13     for (i = 0; i < numberOfValues; ++i)
14         values[i] = rand() % divisionValue;
15 }
16
17 void displayVector(vector<int> values)
18 {
19     for (int value : values)
20         cout << value << " ";
21 }
22
23 void shakerSort(vector<int>& values)
24 {
25     int left, right, i;
26
27     left = 0; right = values.size() - 1;
28
29     do
30     {
31         for (i = right; i > left; --i)
32         {
33             if (values[i] < values[i - 1])
34                 swap(values[i], values[i - 1]);
35         }
```



```

36     left++;
37
38     for (i = left; i < right; ++i)
39     {
40         if (values[i] > values[i + 1])
41             swap(values[i], values[i + 1]);
42     }
43     right--;
44 }
45 while (left <= right);
46 }
47
48 void bubbleSort(vector<int>& values)
49 {
50     int i, j;
51
52     for (i = 0; i < numberOfValues; ++i)
53         for (j = numberOfValues - 1; j > i; --j)
54             if (values[j - 1] > values[j])
55                 swap(values[j - 1], values[j]);
56 }
57
58 int main()
59 {
60     srand(time(0));
61
62     vector<int> values(numberOfValues, 0);
63     generateVector(values);
64
65     cout << "Случайно сгенерированные числа:\n";
66     displayVector(values);
67
68     cout << "\nОтсортированные элементы:\n";
69     bubbleSort(values);
70     displayVector(values);
71
72     cout << "\n\nСлучайно сгенерированные числа:\n";
73     generateVector(values);
74     displayVector(values);
75

```

```

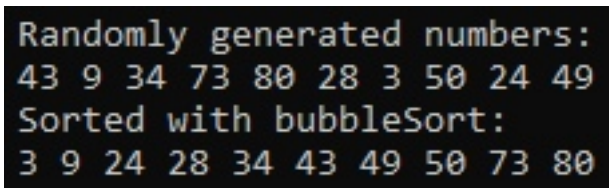
76     cout << "\nОтсортированные элементы:\n";
77     shakerSort(values);
78     displayVector(values);
79 }

```

Листинг 3: Программа алгоритмов сортировки пузырьком и шейкерной сортировки.

Для хранения элементов, которые нужно отсортировать, будем пользоваться структурой "вектор". Для демонстрации работы алгоритма будем генерировать вектор случайных значений, содержащий 10 чисел. Генерацию и вывод случайных значений будем осуществлять при помощи вспомогательных функций *generateVector()* и *displayVector()*.

Сначала протестируем алгоритм сортировки пузырьком, расположенный на 48 - 56 строках программы. Сгенерируем вектор случайных значений и попробуем его отсортировать:



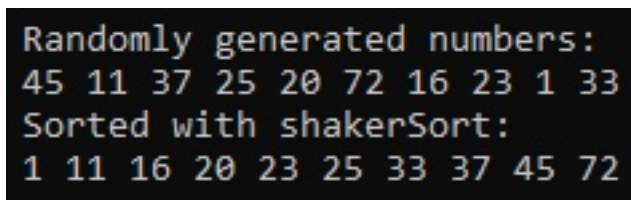
```

Randomly generated numbers:
43 9 34 73 80 28 3 50 24 49
Sorted with bubbleSort:
3 9 24 28 34 43 49 50 73 80

```

Рис. 1 - Входные и выходные данные для алгоритма сортировки пузырьком.

Как видно, алгоритм справляется со своей задачей. Теперь рассмотрим шейкерную сортировку. Сгенерируем новый вектор из 10 случайных значений и посмотрим на результат выполнения алгоритма.



```

Randomly generated numbers:
45 11 37 25 20 72 16 23 1 33
Sorted with shakerSort:
1 11 16 20 23 25 33 37 45 72

```

Рис. 2 - Входные и выходные данные для алгоритма шейкерной сортировки.

Видно, что алгоритм справляется с поставленной задачей. Разберём его подробнее. Основная часть работы выполняется на строках 29 - 45. При входе в функцию инициализируются два значения - *left* и *right* - изначально они указывают на начало и конец вектора соответственно. Далее, элементы

начинают обрабатываться в цикле. В строках 31 - 35 выполняется сдвиг к началу вектора более "легких"элементов - меньших по значению. В строках 38 - 42 осуществляется сдвиг к концу вектора более "тяжёлых"элементов, больших по значению. Как видно, такие сдвиги осуществляются поочерёдно - сначала продвигаются меньшие, а затем большие значения. После каждого такого цикла значение *left* увеличивается, а значение *right* уменьшается, что означает, что все элементы, стоящие до *left* и после *right*, стоят на своих местах. Внешний цикл продолжается до тех пор, пока значения *left* и *right* не совпадут.

ЗАКЛЮЧЕНИЕ

В ходе проведённого анализа было выявлено, что алгоритм пузырьковой сортировки и его разнообразные модификации являются малоэффективными алгоритмами сортировок. Причиной этому является тот факт, что все простые методы сортировок перемещают элемент на одну позицию на каждом элементарном шаге, поэтому всегда требуют порядка n^2 таких шагов. Любое серьёзное усовершенствование должно иметь целью увеличение расстояния, на которое перемещаются элементы в каждом прыжке. Такой способ перемещения элементов реализован, например, в сортировках вставками, выбором и обменами.