Отчет по лабораторной работе №2
по курсу "Анализ алгоритмов"
по теме "Сортировки"

Студент: Доктор А.А. ИУ7-53
Преподаватель: Волкова Л.Л., Строганов Ю.В.

2018 г.

Содержание

Введение

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

Оценка алгоритма сортировки

Ниже приведены параметры оценки эффективности алгоритма [1].

1. Время — параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах мощности входного множества A. Если на вход алгоритму подаётся множество A, то обозначим n = |A|. Для типичного алгоритма хорошее поведение — это O(n log n) и плохое поведение — это O(n2). Идеальное поведение для упорядочения — O(n). Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей всегда нуждаются по меньшей мере в сравнениях. Тем не менее, существует алгоритм сортировки Хана (Yijie Han) с вычислительной сложностью O(n log log n log log log n), использующий тот факт, что пространство ключей ограничено (он чрезвычайно сложен, а за O-обозначением скрывается весьма большой коэффициент, что делает невозможным его применение в повседневной практике). Также существует понятие сортирующих сетей. Предполагая, что можно одновременно (например, при параллельном вычислении) проводить несколько сравнений, можно отсортировать n чисел за O(log2 n) операций. При этом число n должно быть заранее известно.

2. Память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют O(log n) памяти. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет O(1)). Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к сортировкам на месте.

3. Устойчивость - устойчивая сортировка не меняет взаимного расположения равных элементов. Такое свойство может быть очень полезным, если они состоят из нескольких полей,

4. Естественность поведения - эффективность метода при обработке уже отсортированных, или частично отсортированных данных. Алгоритм ведет себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

Еще одним важным свойством алгоритма является его сфера применения. Здесь основных позиций две:

1. внутренние сортировки работают с данным в оперативной памяти с произвольным доступом;

2. внешние сортировки упорядочивают информацию, расположенную на внешних носителях. Это накладывает некоторые дополнительные ограничения на алгоритм: доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только элемент, следующий за текущим объем данных не позволяет им разместиться в ОЗУ Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью.

Данный класс алгоритмов делится на два основных подкласса:

1. Внутренняя сортировка оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно сортируются на том же месте, без дополнительных затрат.

2. Внешняя сортировка оперирует с запоминающими устройствами большого объема, но с доступом не произвольным, а последовательным (сортировка файлов), т.е в данный момент мы 'видим' только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики . Это приводит к специальным методам сортировки, обычно использующим дополнительное дисковое пространство.

Задачи работы

Выше я перечислил основные сортировки, существующие на сегодняшний день, однако в данной лабораторной я разберу лишь некоторые из них, а именно:

1) сортировка вставками(Insertion) с бинарным поиском;

2) терпеливая(patience) сортировка;

2) поразрядная LSD сортировка;

1 Аналитическая часть

В данном разделе приведено описание сортировок.

1.1 Описание алгоритмов

Сортировка вставками

Сортировка вставками (Insertion Sort) — это простой алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. [3]

На вход алгоритма подаётся последовательность n чисел: a_1,a_2,...,a_n. Сортируемые числа также называют ключами. Входная последовательность на практике представляется в виде массива с n элементами. На выходе алгоритм должен вернуть перестановку исходной последовательности a'_1,a'_2,...,a'_n, чтобы выполнялось следующее соотношение a'_1 $\leq$a'$_2$ ... $\leq$a'$_n$ [4].

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 [5].

Данный алгоритм можно ускорить при помощи использования бинарного поиска для нахождения места текущему элементу в отсортированной части.

Терпеливая сортировка

Терпеливая сортировка (англ. patience sorting) — алгоритм сортировки с худшей сложностью O(n log n). Позволяет также вычислить длину наибольшей возрастающей подпоследовательности данного массива. Алгоритм назван по одному из названий карточной игры "Солитёр"— "Patience".

Алгоритм

Имеем массив source[0..n1], элементы которого нужно отсортировать по возрастанию. Разложим элементы массива по стопкам: для того чтобы положить элемент в стопку, требуется выполнение условия — новый элемент меньше элемента, лежащего на вершине стопки; либо создадим новую стопку справа и сделаем наш элемент её вершиной. Используем жадную стратегию: каждый элемент кладётся в самую левую стопку из возможных, если же таковой нет, справа от существующих стопок создаётся новая. Для получения отсортированного массива сначала построим массив стопок, затем выполним n шагов (здесь и далее нумерация шагов начинается с единицы): на i-м шаге выберем из всех вершин стопок наименьшую, извлечём её и запишем в массив ans[0..n1] на i-ю позицию.

Мы формируем новую стопку, когда встречаем элемент больший, чем вершины всех стопок, расположенных слева. В то же время стопки слева были созданы ранее, то есть элементы в них идут в исходной последовательности раньше текущего. Каждая стопка представляет собой убывающую

последовательность, то есть длина НВП в пределах стопки равна единице, поэтому появление новой стопки можно понимать как увеличение длины наибольшей возрастающей подпоследовательности на единицу (изначально длина НВП равна единице). Поэтому длина наибольшей возрастающей подпоследовательности равна количеству стопок.

Сложность

Создадим список стеков для хранения стопок. При раскладывании элементов по стопкам для поиска самой левой подходящей стопки используем бинарный поиск. Соответственно, поиск самой левой стопки занимает O(log p), где p — количество стопок (стеков). Таким образом, временная сложность раскладывания по стопкам не превышает O(nlogn).

Для получения отстортированного массива используем бинарную кучу. На каждом шаге алгоритма необходимо извлечь из кучи стек с минимальной вершиной за O(log p), где p — количество стеков в куче. Снять вершину выбранного стека и вернуть его в кучу за O(log p). Получение отсортированного массива займёт O(nlog n) времени. Получение наибольшей возрастающей подпоследовательности выполняется за O(n) по описанному выше алгоритму. Таким образом, алгоритм сортировки требует O(nlogn) времени в худшем случае и O(n) дополнительной памяти при любом раскладе.

Поразрядная сортировка

Поразрядная сортировка (англ. radix sort) — один из алгоритмов сортировки, использующих внутреннюю структуру сортируемых объектов.

Алгоритм

Исходно предназначен для сортировки целых чисел, записанных цифрами. Но так как в памяти компьютеров любая информация записывается целыми числами, алгоритм пригоден для сортировки любых объектов, запись которых можно поделить на «разряды», содержащие сравнимые значения. Например, так сортировать можно не только числа, записанные в виде набора цифр, но и строки, являющиеся набором символов, и вообще произвольные значения в памяти, представленные в виде набора байт.

Сравнение производится поразрядно: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

Так как выравнивать сравниваемые записи относительно друг друга можно в разную сторону, на практике существуют два варианта этой сортировки. Для чисел они называются в терминах значимости разрядов числа, и получается так: можно выровнять записи чисел в сторону менее значащих цифр (по правой стороне, в сторону единиц, least significant digit, LSD) или более значащих цифр (по левой стороне, со стороны более значащих разрядов, most significant digit, MSD).

LSD-сортировка (Least Significant Digit radix sort)

При LSD сортировке (сортировке с выравниванием по младшему разряду, направо, к единицам) получается порядок, уместный для чисел. Например: 1, 2, 9, 10, 21, 100, 200, 201, 202, 210. То есть, здесь значения сначала сортируются по единицам, затем сортируются по десяткам, сохраняя отсортированность по единицам внутри десятков, затем по сотням, сохраняя отсортированность по десяткам и единицам внутри сотен, и т. п.

В качестве примера рассмотрим сортировку чисел. Как говорилось выше, в такой ситуации в качестве устойчивой сортировки применяют сортировку подсчетом, так как обычно количество различных значений разрядов не превосходит количества сортируемых элементов. Ниже приведен псевдокод цифровой сортировки, которой подается массив A размера n m-разрядных чисел . Сам по себе алгоритм представляет собой цикл по номеру разряда, на каждой итерации которого элементы массива A размещаются в нужном порядке во вспомогательном массиве B. Для подсчета количества объектов, i-й разряд которых одинаковый, а затем и для определения положения объектов в массиве B используется вспомогательный массив C. Функция digit(x,i) возвращает i-й разряд числа x. Также считаем, что значения разрядов меньше k.

MSD-сортировка (Most Significant Digit radix sort)

При MSD сортировке (с выравниванием в сторону старшего разряда, налево), получается алфавитный порядок, который уместен для сортировки строк текста. Например «b, c, d, e, f, g, h, i, j, ba» отсортируется как «b, ba, c, d, e, f, g, h, i, j». Если MSD применить к числам, приведённым в примере получим последовательность 1, 10, 100, 2, 200, 201, 202, 21, 210, 9.

Будем считать, что у всех элементов одинаковое число разрядов. Если это не так, то положим на более старших разрядах элементы с самым маленьким значением — для чисел это 0. Сначала исходный массив делится на k частей, где k — основание, выбранное для представления сортируемых объектов. Эти части принято называть "корзинами"или "карманами". В первую корзину попадают элементы, у которых старший разряд с номером d=0 имеет значение 0. Во вторую корзину попадают элементы, у которых старший разряд с номером d=0 имеет значение 1 и так далее. Затем элементы, попавшие в разные корзины, подвергаются рекурсивному разделению по следующему разряду с номером d=1. Рекурсивный процесс разделения продолжается, пока не будут перебраны все разряды сортируемых объектов и пока размер корзины больше единицы. То есть останавливаемся когда d>m или lr, где m — максимальное число разрядов в сортируемых объектах, l, r — левая и правая границы отрезка массива A.

2 Конструкторская часть

В данном разделе размещены блоксхемы алгоритмов.

2.1 Разработка реализаций алгоритмов

Ниже приложены блоксхемы алгоритмов решения поставленных задач На рисунке 1 представлена реализация алгоритма сортировки вставками с бинарным поиском(Блоки A -> B и следующий после B блок).

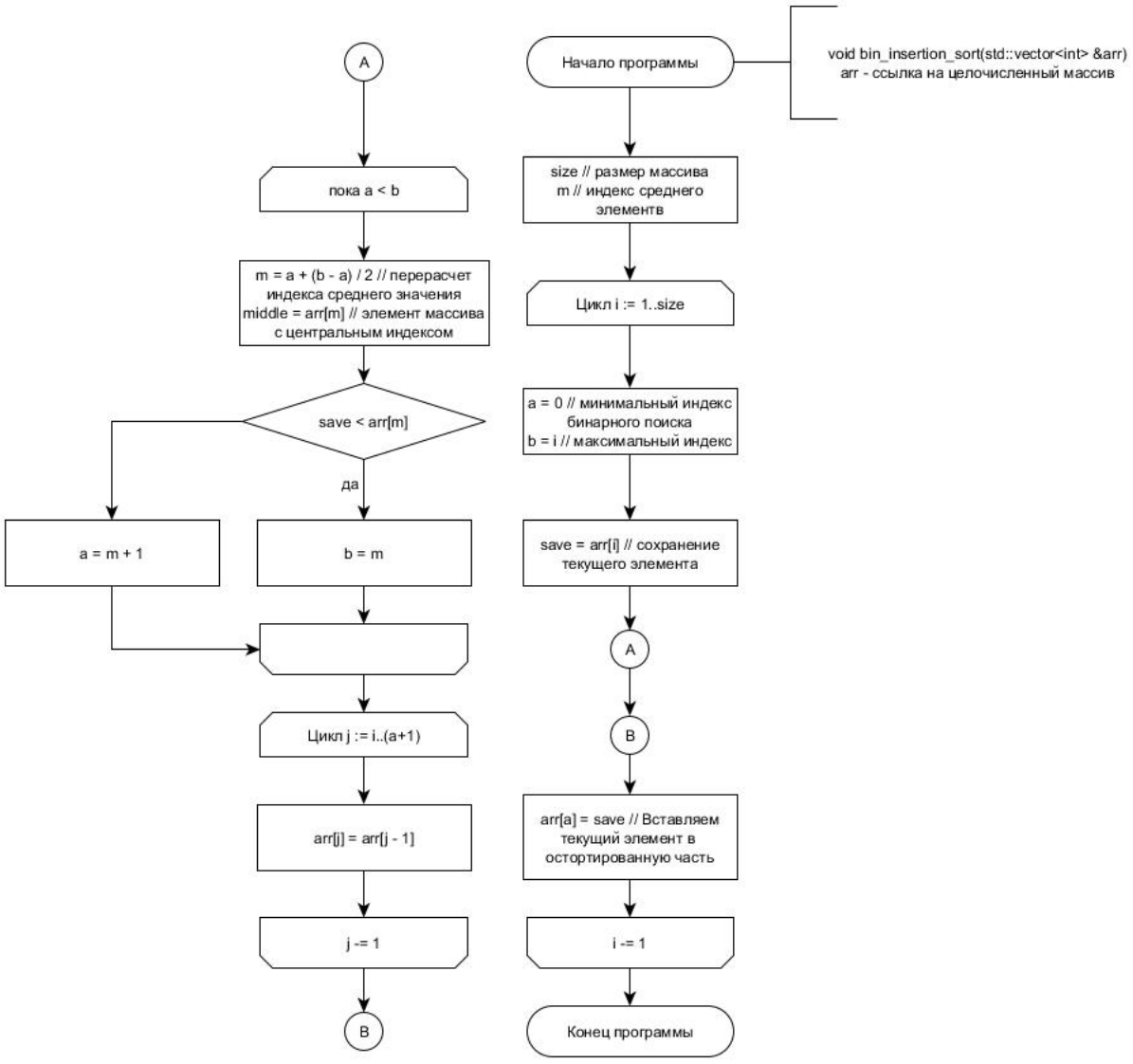

Рис. 1: Функция классического умножения матриц.

На рисунке 2 показан общий принцип работы(с комментариями!) алго-

ритма терпеливой сортировки. Детали реализации раскрываются на рисунке 3.

Рис. 2: Общий принцип работы терпеливой сортировки

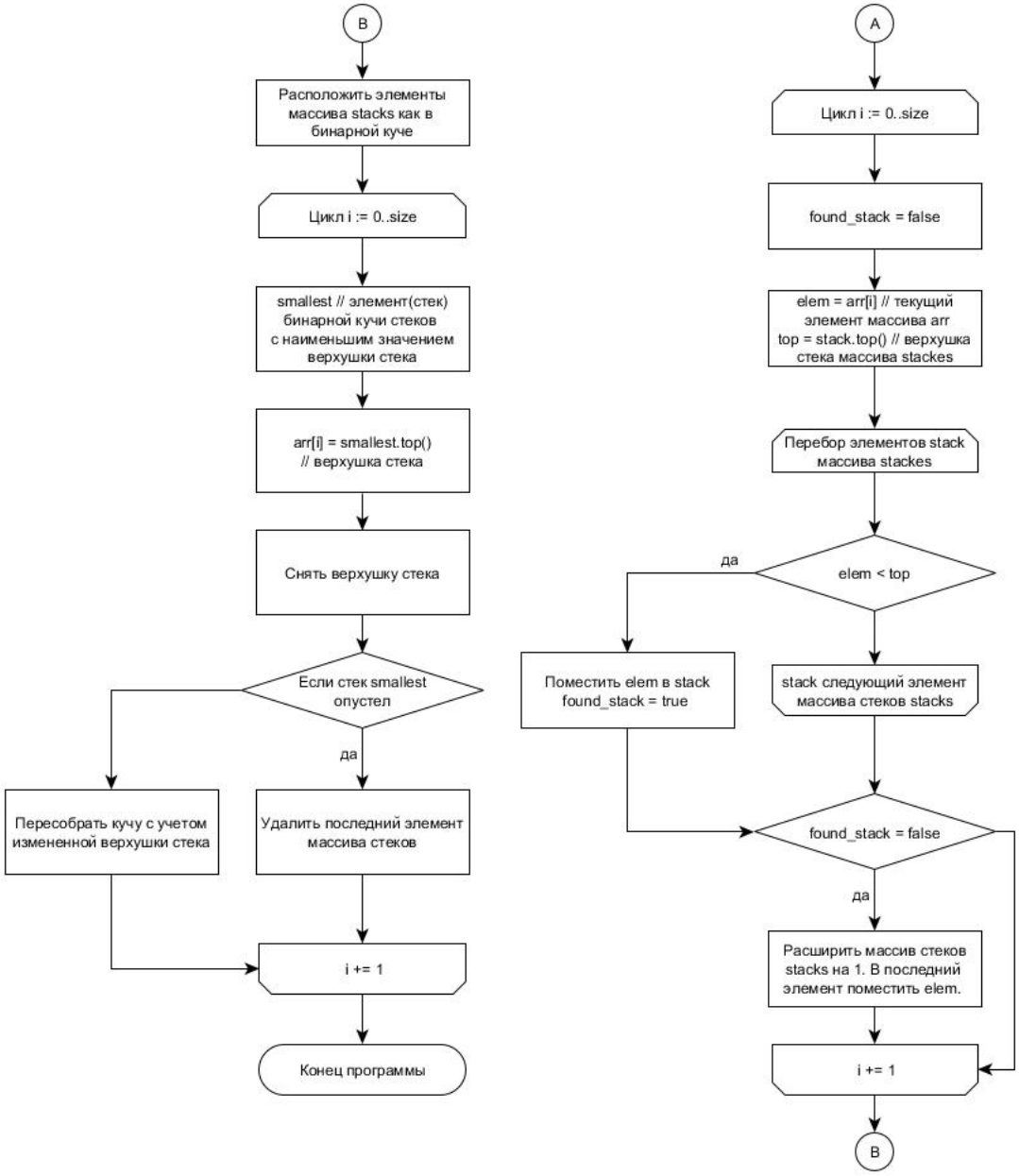

Рис. 3: Терпеливая сортировка в деталях

На рисунке 4 представлен общее представление структуры поразрядной сортировки LSD. На рисунках 6 и 5 в блоке A и E показаны циклы, корректирующие массив в случае обнаружения отрицательных чисел. в Блоке C и D реализована сортировка подсчетом, лежащая в основе поразрядной сортировки.

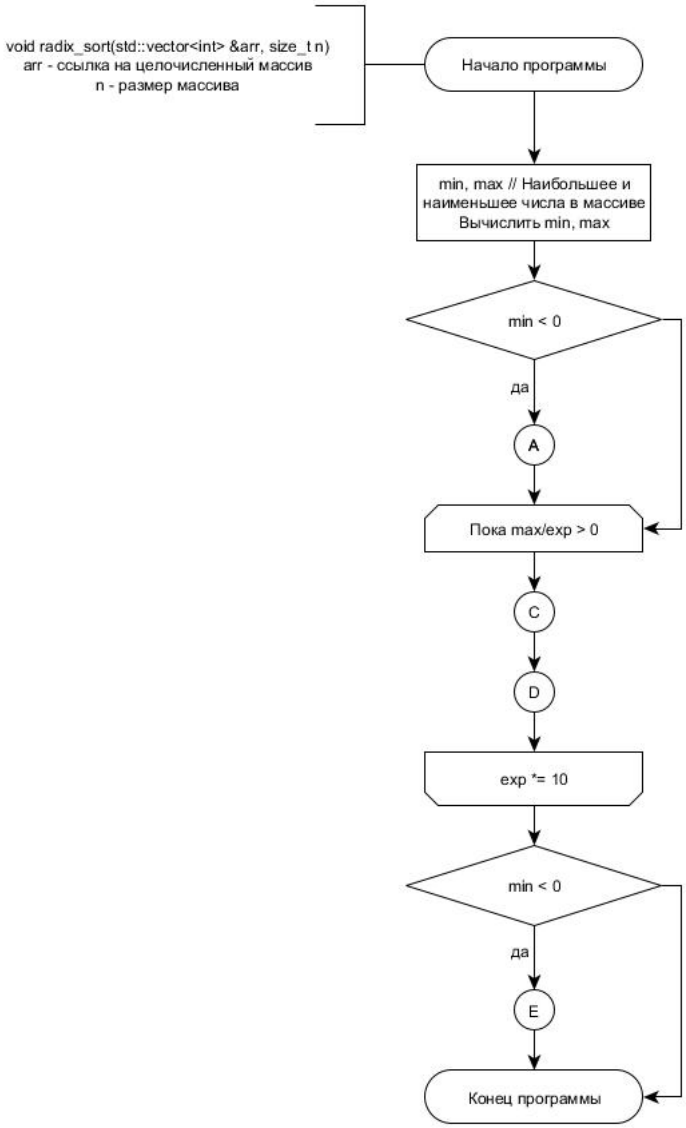

Рис. 4: Общий принцип поразрядной сортировки LSD

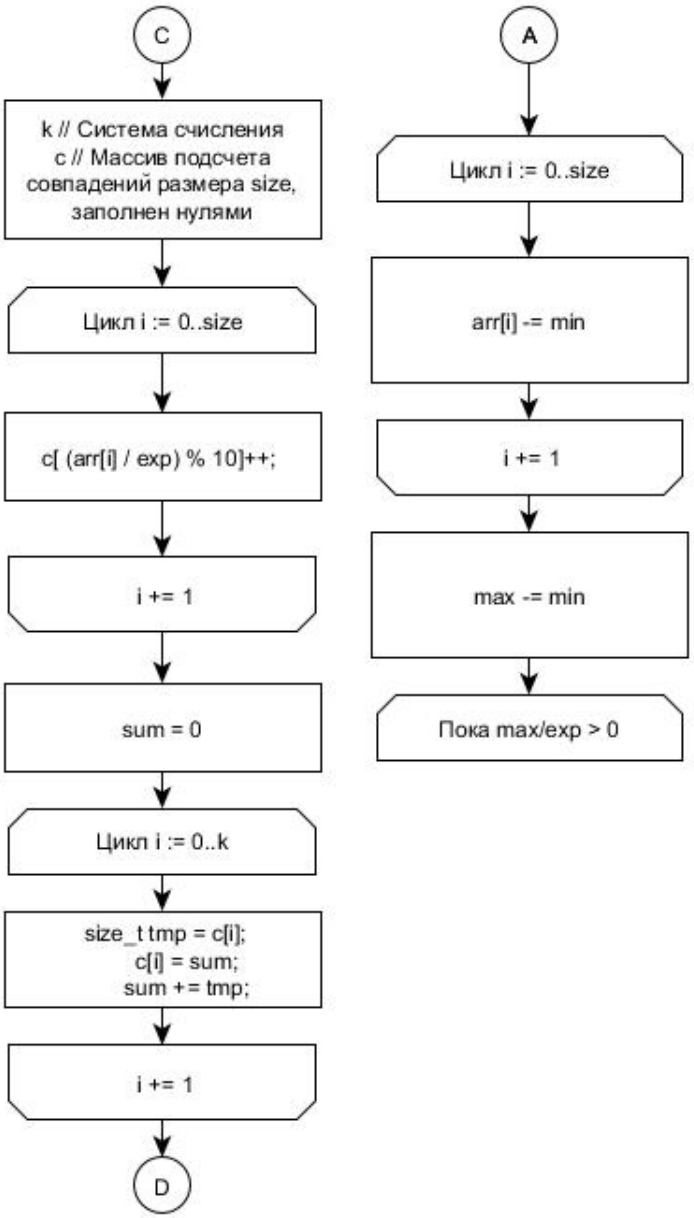

Рис. 5: Сортировка подсчетом. Часть 1

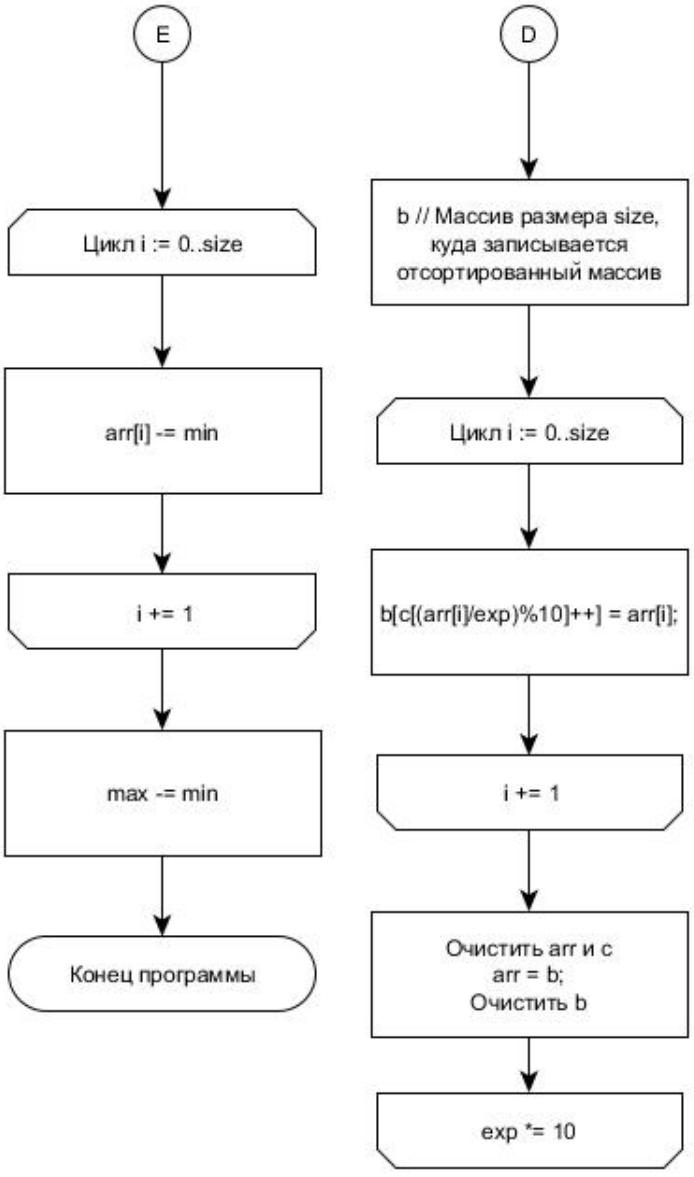

Рис. 6: Сортировка подсчетом. Часть 2

3 Технологическая часть

В данном разделе будут приведены требования к программнму обеспечению и средства реализации и листинг кода.

3.1 Требования к программному обеспечению

Требуется указать размер массива и его элементы.Размер массива должен быть больше нуля. Элементы массива, как и размер, представляют из себя целые числа в диапозоне от -2 147 483 648 до +2 147 483 64.

3.2 Средства реализации

Для реализации программ я выбрал язык программирования - C++, так как имею опыт работы на нём. Среда разработки - Qt. Замеряется время работы процессора с помощью функции, продемонстрированной в листинге 1.

Листинг 1: Функция замера процессороного времени

```
1 unsigned long long tick(void)
2 {
3    unsigned long long d;
4    __asm__ __volatile__ ("rdtsc" : "=A"     (d));
5    return d;
6 }
7
```

Эта функция в отличие от встроенной функции таймера, способна считать реальное процессорное время работы программы в тиках[6].

3.3 Реализация алгоритмов

В листинге 2 представлена реализация сортировки вставками с бинарным поиском.

Листинг 2: Сортировка вставками с бинарным поиском

```
void bin_insertion_sort(std::vector<int> &arr) {
    size_t size = arr.size();
    size_t m = 0;
    for (size_t i = 1; i < size; i++) {
        size_t a = 0;
        size_t b = i;
        int save = arr[i];

        while (a < b) {
            m = a + (b - a) / 2;
            if (save < arr[m]) {
                b = m;
            } else {
                a = m + 1;
            }
        }
        for (size_t j = i; (j > a); j--) {
            arr[j] = arr[j - 1];
        }
        arr[a] = save;
    }
}

```

В листинге 3 представлена реализация терпеливой сортировки. Для ее реализации были написаны дополнительные структуры moreStack и lessStack, которые необходимы для использования функций бинарной кучи библиотеки STL, продемонстрированная в листинге 4.

Листинг 3: Терпеливая сортировка

```
1  void patience_sort(std::vector<int> &arr, size_t size) {
2      std::vector<std::stack<int>> stacks; // массив стеков
3
4      typedef std::vector<std::stack<int>>::iterator Iterator;
5
6      for (auto it = arr.begin(); it != arr.end(); it++)
7      {
8          std::stack<int> stack;
9          stack.push(*it);
10         Iterator iter = std::lower_bound(stacks.begin(), stacks.end(), stack,
11                                          lessStack());
12         if (iter == stacks.end())
13             stacks.push_back(stack);
14         else
15             iter->push(*it);
16     }
17
18     // Превратить массив стеком в контейнер
19     std::make_heap(stacks.begin(), stacks.end(), moreStack());
20     for (size_t i = 0; i < size; i++) {
21         // Первый стек становится последним, остальные перестраиваются
   под кучу
22         std::pop_heap(stacks.begin(), stacks.end(), moreStack());
23         // Достаем последний стек, который на самом деле имеет
   наибольший приоритет
24
25         std::stack<int> &smallest = stacks.back();
26         // На верху стека лежит наименьшее число, а до этого был
   выбран стек с
27         // наибольшим приоритетомнаименьшей( верхушкой)
28
29         arr[i] = smallest.top();
30         smallest.pop();
31
32         if (smallest.empty()) {
33             stacks.pop_back();
34         }
35         else {
36             // Поскольку последний стек обновился, надо пересобрать кучу,
    имитируя
37             // вставку нового элемента. push_heap вставляет"" последний
   элемент
38             std::push_heap(stacks.begin(), stacks.end(), moreStack());
39         }
40     }
41  }
42
```

Листинг 4: Структуры lessStack и moreStack

```
struct moreStack{
   bool operator()(std::stack<int> s1, std::stack<int> s2) {
      return s1.top() > s2.top();
   }
};

struct lessStack{
   bool operator()(std::stack<int> s1, std::stack<int> s2) {
      return s1.top() < s2.top();
   }
};


```

В листинге 5 представлена реализация LSD поразрядной сортировки. Для ее реализации были написаны несколько дополнительных функций. Так, функция из листинга 6 реализует возможность работы данной сортировки с отрицательными числами. А функция get из 7 определяет минимальное и максимальное значения массива для определения наличия отрицательных чисел(по минимальному числу) и количество разрядов, требуемых для обхода(по максимальному числу). И замыкает перечисление функций сортировка подсчетом из листинга 8, лежащая в основе работы любой поразрядной сортировки.

Листинг 5: Поразрядная LSD сортировка

```
    void radix_sort(std::vector<int> &arr, size_t n) {
       int min = 0, max = 0;
       get_min_max_numbers(arr, n, min, max);
       if (min < 0) {
         add_to_all(arr, −min);
         max −= min;
       }
       for (int exp = 1; max/exp > 0; exp ∗= 10) {
          counting_sort(arr, n, exp);
       }
       if (min < 0) {
          add_to_all(arr, min);
       }
    }

```

Листинг 6: Функция прибавляющая число к каждому элементу массива

```
    void add_to_all(std::vector<int> &arr, int num) {
       size_t n = arr.size();
       for(size_t i = 0; i < n; ++i ) {
          arr[i] += num;
       }
    }

```

Листинг 7: Функция получающая минимум и максимум в массиве

```
1    void get_min_max_numbers(std::vector<int> arr, size_t size,
2                             int &min, int &max) {
3       min = max = arr[0];
4       for (size_t i = 1; i < size; i++) {
5          if (arr[i] > max) {
6             max = arr[i];
7          } else if (arr[i] < min){
8             min = arr[i];
9          }
10      }
11   }
12
```

Листинг 8: Сортировка подсчетом

```
1       void counting_sort(std::vector<int> &arr,
2                size_t size, int exp) {
3        size_t k = 10;
4        std::vector<size_t> c(k, 0);
5        for(size_t i = 0; i < size; ++i )
6           c[ (arr[i] / exp) % 10]++; ;
7        size_t sum = 0;
8        for(size_t i = 0; i < k; ++i ) {
9           size_t tmp = c[i];
10          c[i] = sum;
11          sum += tmp;
12       }
13       std::vector<int> b(size);
14       for( size_t i = 0; i < size; ++i ) {
15          b[c[(arr[i]/exp)%10]++] = arr[i];
16       }
17       arr.clear();
18       c.clear();
19       arr = b;
20       b.clear();
21    }
22
```

4 Экспериментальная часть

В данном разделе будут приведена сложность реализованных алгоритмов и их сравнительный анализ на основе экспериментальных данных.

4.1 Сложность

Сортировка вставками с бинарным поиском

Для того, чтобы наглядно показать сложность алгоритма сортировки вставками с бинарным поиском приведу листинг кода 9, из которого видно, что в общем случае сложность равна 2 + 1 + 2 + (size - 1)(2 + 1 + 1 + 2 + 1 + log(i) * (1 + 7 || 8) + 2 + size2(2 + 4)) или же 12*size -7 + 9*log((n)!) + 6*size*size2. Трудность вызывает 4*size*size2, поскольку size2 зависит от поступающего на вход массива, поэтому предлагается рассмотреть отдельные 2 случая:лучший и худший.

Лучший случай

Пусть на вход пришёл массив [1,2,3,4,5,6,7,8]. на первой итерации перед заходом в цикл(16-ая строчка), i = 1, a = 1, size2 = a - i = 0, аналогично на второй итерации, третьей и т.д. Это связано с тем, что мы хотим вставить элемент в то место, в котором он уже находится, поэтому size2 каждый раз будет равняться нулю и общая сложность становится следующей: -7 + 84 + 9log(8!) или $\mathbf{O(log(n!))}$ для лучшего случая

Худший случай

Пусть на вход пришёл массив [8,7,6,5,4,3,2,1]. На первой итерации i = 1, a = 0, size2 = 1, на второй i = 2, a = 0, size2 = 2 и т.д. На каждой итерации a = 0, поскольку крайний левый элемент неотсортированной части каждый раз меньше всех элементов отсортированной части, поэтому size2 будет n!. Общая сложность: -7 + 84 + 9log(8!) + 6*8*8! или $\mathbf{O}(n^2)$ для худшего случая

Листинг 9: Сортировка вставками с бинарным поиском

```
1  //
2   // Модель
3   //'/ ','<', '=', '[]', '++', '--', '.'
4   // стоимость 1
5   //
6  void bin_insertion_sort(std::vector<int> &arr) {
7   size_t size = arr.size(); //2 +
8   size_t m = 0; //1 +
9   for (size_t i = 1; i < size; i++) { // 2 + (size-1) * ( 2 +
10     size_t a = 0; // 1 +
11     size_t b = i; // 1 +
12     int save = arr[i]; // 2 +
13
14     while (a < b) { // 1 + log(i) * (1 +
15        m = a + (b - a) / 2; // 4 +
16        if (save < arr[m]) { // 2 + (1 || 2))
17           b = m;
```

```
18            } else {
19               a = m + 1;
20            }
21         } // int size2 = i - a
22         for (size_t j = i; (j > a); j--) { // 2 + size2 * (2 +
23            arr[j] = arr[j - 1]; // 4)
24         }
25         arr[a] = save; // + 1)
26      }
27   }
28
```

Терпеливая сортировка

Создадим список стеков для хранения стопок. При раскладывании элементов по стопкам для поиска самой левой подходящей стопки используем бинарный поиск. Соответственно, поиск самой левой стопки занимает O(log p), где p — количество стопок (стеков). Таким образом, временная сложность раскладывания по стопкам не превышает O(nlogn).

Для получения отстортированного массива используем бинарную кучу. На каждом шаге алгоритма необходимо извлечь из кучи стек с минимальной вершиной за O(log p), где p — количество стеков в куче. Снять вершину выбранного стека и вернуть его в кучу за O(log p). Получение отсортированного массива займёт O(nlog n) времени. Получение наибольшей возрастающей подпоследовательности выполняется за O(n) по описанному выше алгоритму. Таким образом, алгоритм сортировки требует **O(nlogn)** времени **в худшем случае** и **O(n) дополнительной памяти** при **любом** раскладе[7]. В **лучшем** же случае сложность **O(N)** [8].

Поразрадная LSD сортировка

Сначала отсортируем их по первому(старшему) разряду. Сортировка в таком случае выполняется с помощью сортировки подсчетом (count sort). Сложность — o(n). Мы получили 10 «корзин» — в которых старший разряд 0, 1, 2 и т.д. Далее в каждой из корзин запускаем ту же процедуру, но только рассматриваем уже не старший разряд, а следующий за ним, и т.д.

Шагов столько, сколько разрядов в числах. Соответственно, сложность алгоритма — O(n*k), k — число разрядов. [9]

4.2 Сравнительный анализ

Для экспериментов использовались массивы, размер которых варьируется от 1000 до 1000 с шагом 1000. Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными. Каждая таблица состоит из 4-ёх столбцов. Первый столбец представляет собой количество элементов. Оставшиеся три - время, затраченное в худшем, обычном и лучшем вариантах. Под обычным вариантом подразумевается сортировка массива из случайных чисел. В таблице 1 приведены результаты замера сортировки вставками с бинарным поисков. Здесь лучший случай, когда массив был отсортирован заранее. Худший же, когда массив обратно-отсортированный. Замеры терпеливой сортировки показаны в таблице 2. Лучший случай, когда таблица обратно отсортирована, худший, когда прямо. Результаты работы LSD сортировки можно увидеть в таблице 3. Так как эта сортировка зависит от количества разрядов, то лучший случай, когда разряд всего 1. За обычный случай принята ситуация, когда максимальное количество разрядов равно трём. Худший случай, когда числа отрицательные и количество разрядов равно максимально допустимому в рамках выбранного типа.

Таблица 1: Результаты замеров сортировки вставками с бинарным поиском.

Количество элементов	Отсортированный (в тиках)	Из случайных чисел (в тиках)	Обратно-отсортированный (в тиках)
1000	206514	8145084	15627474
2000	385240	23286510	41709682
3000	463698	44311186	69365279
4000	624200	60320225	118394474
5000	726015	92174428	185447617
6000	851555	131531028	267783319
7000	1087951	183235489	365067903
8000	1239921	239472118	479276444
9000	1438931	296470986	597333496
10000	1611345	373288185	748708369

Таблица 2: Результаты замеров терпеливой сортировки

Количество элементов	Отсортированный (в тиках)	Из случайных чисел (в тиках)	Обратно-отсортированный (в тиках)
1000	26969365	20183614	7555004
2000	41837848	30510740	11882920
3000	58297726	43699749	15566613
4000	73692411	57359246	20776705
5000	101998422	77268127	26669980
6000	118373450	90142691	31857210
7000	135697894	104254801	36132328
8000	156642849	120577387	42425616
9000	201309386	141705472	48113387
10000	220771653	158540637	53537901

Таблица 3: Результаты замеров поразрядной LSD сортировки

Количество элементов	Из случайных чисел от 0 до 10(в тиках)	Из случайных чисел от 0 до 1000(в тиках)	Из отрицательных чисел больших -1000000 (в тиках)
1000	124241	314779	808965
2000	235804	549783	1447132
3000	344252	744940	2094131
4000	416376	1094003	2894579
5000	527703	1333514	3421013
6000	653100	1616399	4800591
7000	720129	1988292	5213457
8000	810016	2116511	5743089
9000	890413	2411941	6521953
10000	1062303	2724617	7134253

Ниже приведены графики, на которых наглядно видна сравнительная скорость работы обсуждаемых алгоритмов. Смотрите на графики 7, 8 и 9.

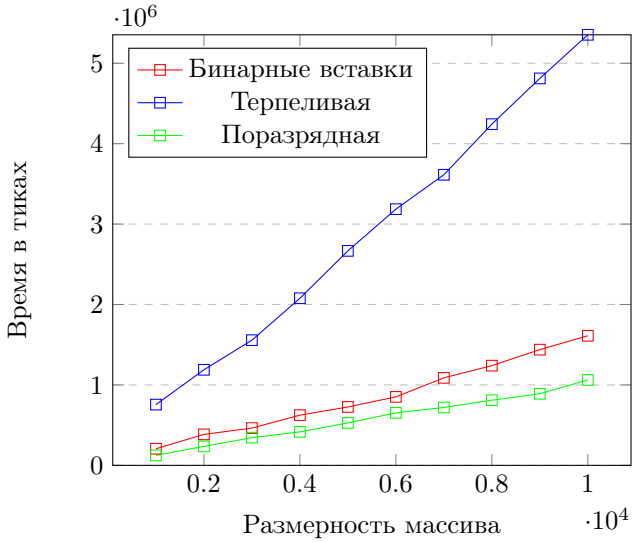

Рис. 7: График лучших случаев сортировок

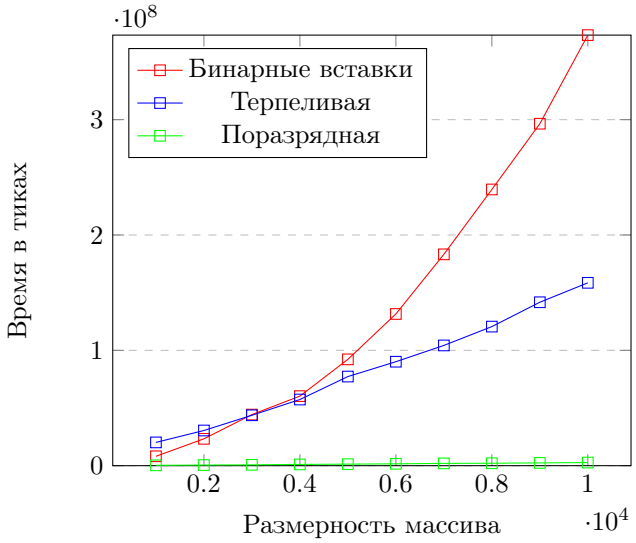

Рис. 8: График обычных случаев сортировок

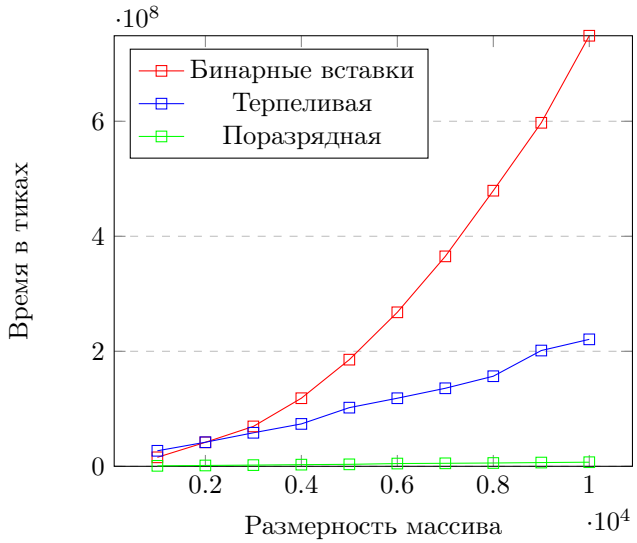

Рис. 9: График худших случаев сортировок

Вывод

В результате проведенного эксперимента был получен следующий вывод: алгоритм Бинарных вставок справляктся лучше всех в случае, когда массив заранее отсортирован. Поразрядная сортировка лучше всего работает, когда разброс значений(а значит и количество разрядов) не столь велико, однако и в остальных случаях, LSD сортировка ведет очень расторопно - быстрее всех остальных рассмотренных здесь сортировок. Терпеливая сортировка тратит много времени на распределение элементов по стопкам и преобразование массива к бинарной куче, за счет чего не может потягаться с той же поразрядной сортировкой и даже проигрывает вставкам с бинарным поиском в своем лучшем случае.

Заключение

В ходе работы были изучены и реализованы алгоритмы сортировки вставками с бинарным поиском, терпеливой сортировки и поразрядной. Подтвердилось, что поразрядная сортировка наиболее эффиективная при различных наборах данных. Бинарные вставки не смотря на свою простоту в некоторых случаях(когда массив частично отсортирован) могут потягаться с более продвинутыми своими 'коллегами'. Терпеливую же следует использовать по своему назначению - для поиска наибольшей подпоследовательности в последовательности.

Список литературы

[1] Кантор Илья, http://algolist.manual.ru/sort/

[2] Российский государственный социальный университет, Кафедра «Финансы и кредит», Великохатько Кира, https://studfiles.net/preview/4652654/

[3] Кнут Д. Э. 5.2.1 Сортировка путём вставок // Искусство программирования. Том 3. Сортировка и поиск = The Art of Computer Programming. Volume 3. Sorting and Searching / под ред. В. Т. Тертышного (гл. 5) и И. В. Красикова (гл. 6). — 2-е изд. — Москва: Вильямс, 2007. — Т. 3. — 832 с. — ISBN 5-8459-0082-1.

[4] Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. 2.1. Сортировка вставкой // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 3-е изд. — М.: Вильямс, 2013. — С. 38-45. — ISBN 5-8459-1794-8.

[5] Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72-76. — ISBN 5-94836-005-9.

[6] И.В. Ломовской. Курс лекций по языку программирования С, 2017

[7] Университет ИТМО, статья про терпеливую сортировку, http://neerc.ifmo.ru/wiki/index.php?title=Терпеливаясортировка

[8] Chandramouli, Badrish; Goldstein, Jonathan (2014). Patience is a Virtue: Revisiting Merge and Sort on Modern Processors (PDF). SIGMOD/PODS.

[9] https://habr.com/post/268261/