Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

Московский государственный технический университет имени Н.Э.Баумана
(МГТУ им. Н.Э.Баумана)

ОТЧЕТ
По лабораторной работе № 3
По курсу «Анализ алгоритмов»
на тему «Алгоритмы сортировок»

Выполнил Студент: Московец Н.С
Группа: ИУ7-51

Москва, 2017

Оглавление

Постановка задачи

Изучить и реализовать алгоритмы сортировки. Оценить трудоемкость одного из алгоритмов. Сравнить реализованные алгоритмы.

Описание модели вычислений

Опишем модель вычислений, которой будем пользоваться в дальнейшем при оценке трудоемкости алгоритмов.

Операции, имеющие трудоемкость “1”:

- арифметические операции { сложение(+), вычитание(-), умножение(*), деление(/), битовый сдвиг(<<, >>), деление нацело, взятие остатка(%) };
- логические операции { и(&&), не(!), или(||) };
- операции сравнения {<, >, =, !=, >=, <=};
- операции присваивания {=, +=, -=, *=, /=, %=};
- операция взятия индекса ([]);
- операции побитового И(&) и ИЛИ(|)
- унарный плюс и минус

- операция разыменования указателя(*)
- операции инкремента и декремента(постфиксные и префиксные) (++, --).

Операции, имеющие трудоемкость “0”:

- логический переход по ветвлению;
- операции обращения к полю структуры/класса (->, .);
- объявление переменных

Сортировка вставками

1. Описание

Сортировка вставками — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. Вычислительная сложность — $O(n^2)$.

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.

Данный алгоритм можно ускорить при помощи использования бинарного поиска для нахождения места текущему элементу в отсортированной части. Проблема с долгим сдвигом массива вправо решается при помощи смены указателей.

2. Реализация

```
1. template <class T>
2. void insertionSort(T* begin, T* end) {
3.     T tmp;
4.     T* j;
5.     for(T* i = begin + 1; i < end; i++) { //3 + 13(n-1) + 9W
6.         tmp = *i; //2
7.         j = i - 1; //2
8.         while(j >= begin && tmp >= *j) { //4 + W(4 + 5)
9.           *(j + 1) = *j; //4
10.           j--;           //1
11.         }
12.         *(j + 1) = tmp; //3
13.     }
14.  }
```

3. Теоретическая оценка

$f = 3 + 13(n-1) + 9W = 9W + 13n - 10$, где W - число заходов в цикл while

Лучший случай - отсортированный массив - W = 0:

$f_л = 13n - 10$

Худший случай - массив, отсортированный в обратную сторону.

$$w = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

$f_x = 9(n-1)n/2 + 13n - 10 = 4.5n^2 + 8.5n - 10$

Сортировка “пузырьком”

1. Описание

Сортировка простыми обменами, сортировка пузырьком — простой алгоритм сортировки. Для понимания и реализации этот алгоритм — простейший, но эффективен он лишь для небольших массивов. Сложность алгоритма: O (n^2).

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции, как пузырёк в воде - отсюда и название алгоритма).

2. Реализация

```
1. template <class T>
2. void bubleSort(T* begin, T* end)
3. {
4.     bool flag;
5.     for (T *i = end - 1; i >= begin; i--) {
6.         flag = true;
7.         for (T *j = begin + 1; j <= i; j++) {
8.             if (*(j-1) > *j) {
9.                 T tmp = *j;
10.                *j = *(j-1);
11.                *(j-1) = tmp;
12.                flag = false;
13.            }
14.        }
```

```
15.             if(flag)
16.                 break;
17.         }
18.     }
```

3. Теоретическая оценка

Трудоемкость тела внутреннего цикла не зависит от количества элементов в массиве, поэтому оценивается как Θ(1). В результате выполнения внутреннего цикла, наибольший элемент смещается в конец массива неупорядоченной части, поэтому через N таких вызовов массив в любом случае окажется отсортирован. Если же массив отсортирован, то внутренний цикл будет выполнен лишь один раз. Тогда в лучшем случае алгоритм будет иметь трудоемкость $T_{л} = O\left(\sum_{j=1}^{n-1} 1\right) = O(n)$

В среднем и худшем случае $T = O\left(\sum_{i=n-1}^{0} \sum_{j=1}^{i} 1\right) = O(n^2)$

Быстрая сортировка

1. Описание

Быстрая сортировка, сортировка Хоара — широко известный алгоритм сортировки, разработанный английским информатиком Чарльзом Хоаром. Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем O(nlogn) обменов при упорядочении n элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

Общая идея алгоритма состоит в следующем:

- Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.
- Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».
- Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае

эффективнее, так как упрощает алгоритм разделения.

2. Реализация

```
template <class T>
void quickSort(T* begin, T* end) {
    T *i = begin, *j = end, x = *(i + (j - i) / 2);

    do {
        while (*i < x) i++;
        while (*j > x) j--;

        if(i <= j) {
              if (*i > *j) {
                  T tmp = *j;
                  *j = *i;
                  *i = tmp;
              }
              i++;
              j--;
          }
      } while (i <= j);

      if (i < end)
          quickSort(i, end);
      if (begin < j)
          quickSort(begin, j);
  }
```

3. Теоретическая оценка

Лучшим случаем для быстрой сортировки является ситуация, когда с помощью выбранного опорного элемента массив будет делится ровно пополам (+- 1 элемент). Тогда глубина рекурсии будет равна $\log_2 n$. При этом на каждом уровне рекурсии суммарная трудоемкость разделения массива на две части будет равна O(n). Тогда трудоемкость алгоритма будет равна $O(n\log_2 n)$.

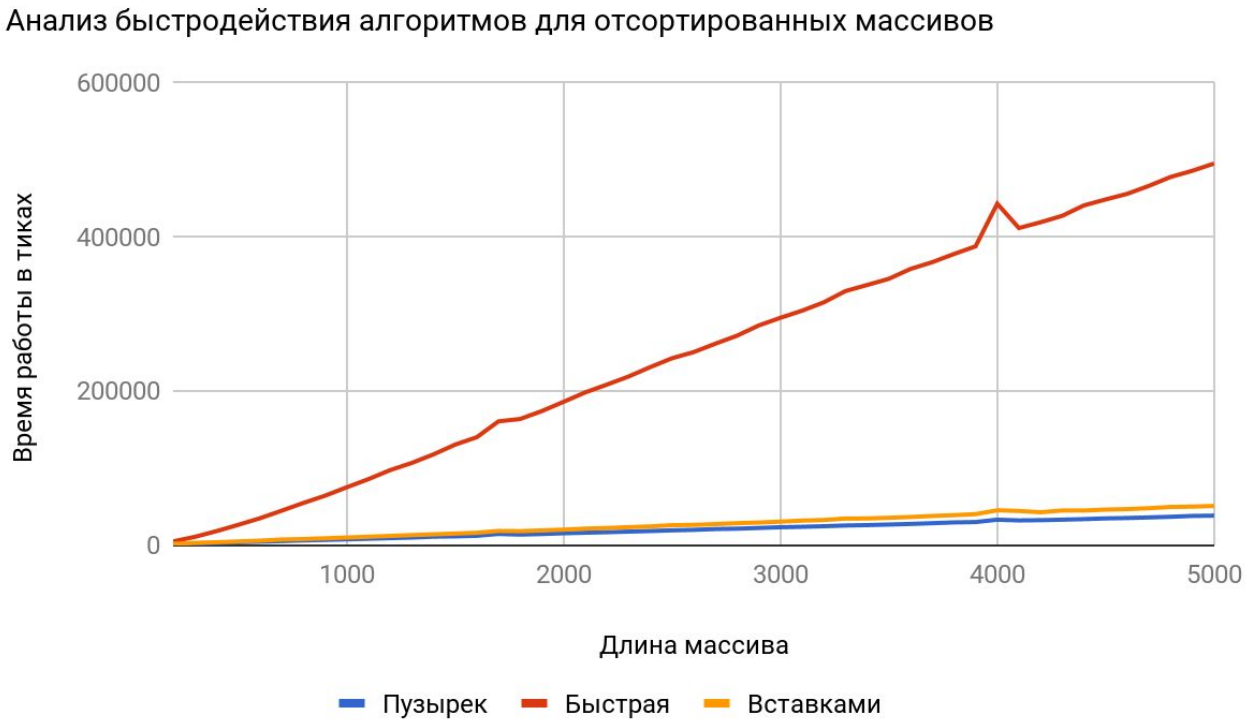
В среднем временная сложность алгоритма составляет O(nlogn).

Худшим случаем будет та ситуация, когда выбираемый опорный элемент равен минимальному или максимальному значению на текущем интервале, так как это приведет к тому, что при рекурсивном вызове sort длина массива сокращается не в два раза, а всего лишь на один элемент. Трудоемкость алгоритма быстрой сортировки для этого случая будет равна $O(n^2)$. В связи с этим существуют различные оптимизации, в том числе и по выбору барьерного элемента, который можно выбирать случайным образом или, например,

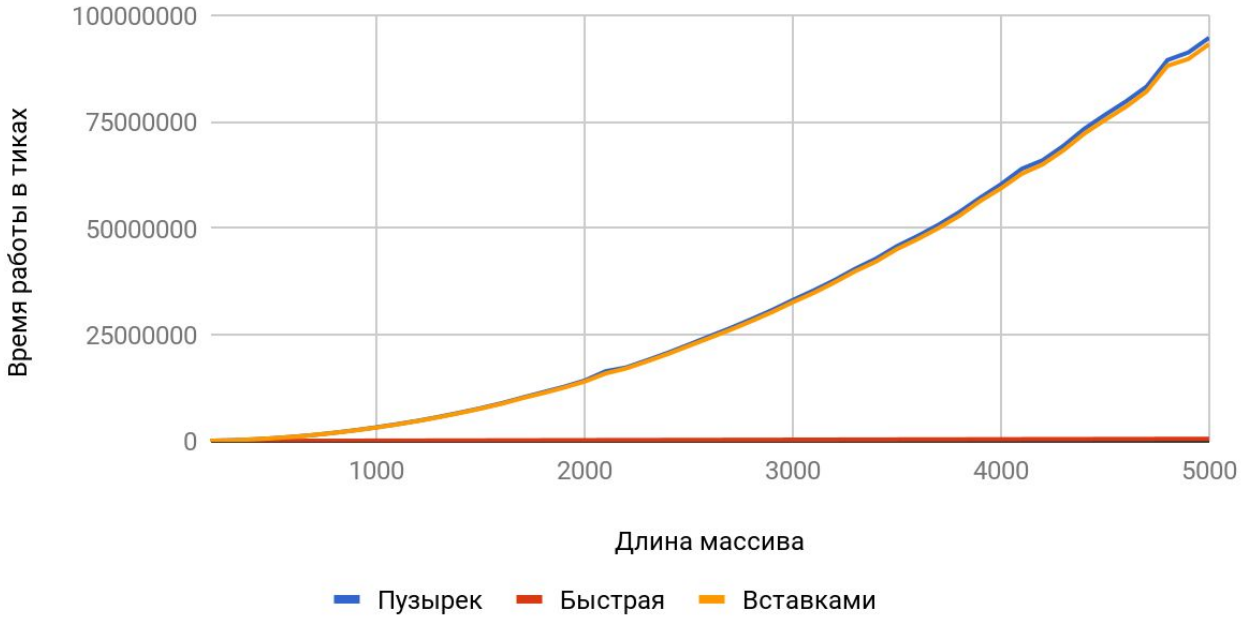
выбирать средний по значению элемент из первого, последнего и среднего элементов текущего участка массива.

Сравнение алгоритмов

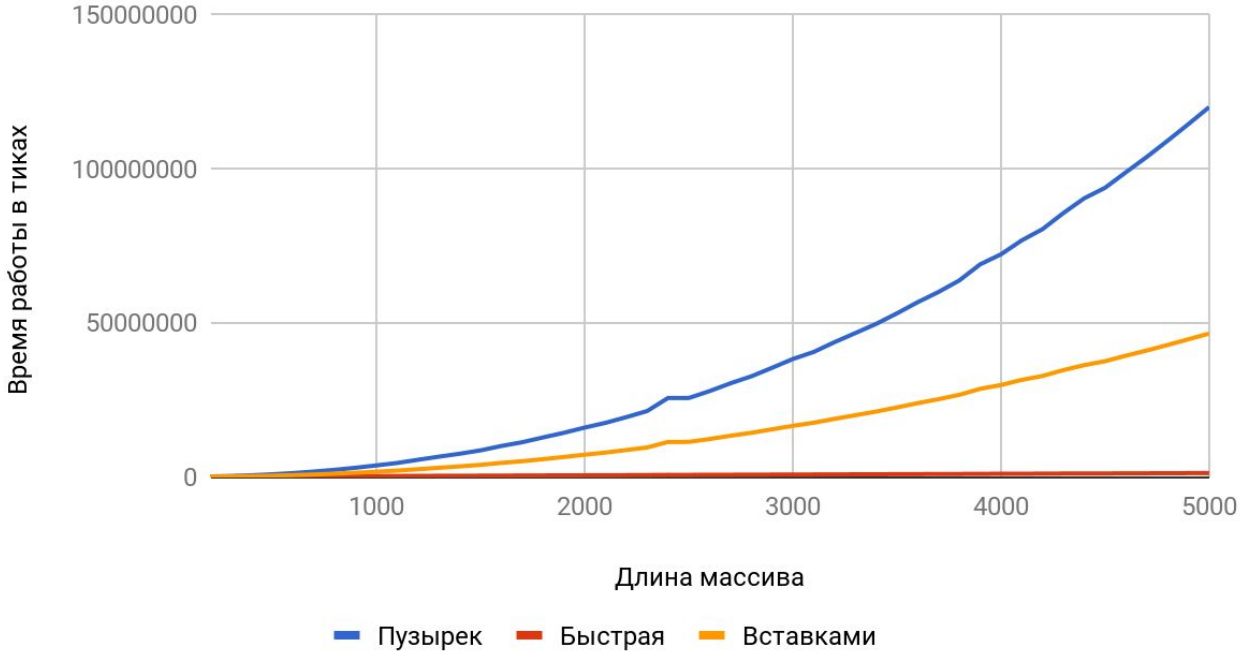
Для сравнения алгоритмов было посчитано среднее время работы для массивов различной длины: 100, 200, …, 5000. При этом были проведены отдельные тесты для произвольных массивов, отсортированных и отсортированных в обратном порядке массивов.



Анализ быстродействия алгоритмов для отсортированных в обратном порядке массивов



Анализ быстродействия алгоритмов для произвольных массивов



Выводы:

- Для произвольных массивов и массивов отсортированных в обратную сторону, зависимость времени работы алгоритма от длины массива квадратичная для сортировок вставками и пузырьком, и линейная для быстрой сортировки, что и подтверждает теоретическую оценку трудоемкости этих алгоритмов.

- В лучшем случае - для отсортированного массива, алгоритмы сортировки вставками и пузырьком работают быстрее, чем алгоритм быстрой сортировки.
- Когда массив частично отсортирован и имеет небольшое количество элементов, то наиболее эффективным алгоритмом сортировки является сортировка вставками. Низкая эффективность быстрой сортировки в таком случае объясняется наличием дополнительных затрат на обслуживание вызовов рекурсивной функции.

Заключение

Во время выполнения работы были изучены и реализованы алгоритмы сортировки. Была произведена оценка трудоемкости сортировки вставками и сравнение быстродействия работы реализованных алгоритмов.