Министерство образования Российской Федерации
Московский Государственный Технический
Университет им. Н.Э. Баумана

Отчет по лабораторной работе №3
По курсу «Анализ алгоритмов»

Тема: «Алгоритмы сортировки»

Студент: **Горохова И.Б.**
Группа: **ИУ7-51**

Преподаватель: **Волкова Л.Л.**

Москва, 2017

Содержание

Постановка задачи

В ходе лабораторной работы предстоит

1. изучить работу алгоритмов сортировки;
2. выполнить полную математическую оценку трудоёмкости для одного алгоритма сортировки с указанием лучшего, среднего и худшего случаев;
3. выполнить краткую математическую оценку трудоёмкости для двух других алгоритмов сортировки с указанием лучшего, среднего и худшего случаев;
4. реализовать три алгоритма сортировки на одном из языков программирования;
5. сравнить работу алгоритмов сортировок и сделать выводы.

Алгоритм сортировки вставками

На вход алгоритма подаётся последовательность n чисел: $a_1, a_2, ..., a_n$. 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 $a_1^{'} <= a_2^{'} <= ... <= a_n^{'}$
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.

Листинг 1: Алгоритм сортировки вставками

```
1  public static long InsertionSort(int[] array, int n) {
2    int newElem, location;
3    //...
4    for (int i = 1; i < n; i++)
5    {
6      newElem = array[i];
7      location = i-1;
8      while (location >= 0 && array[location] > newElem)
9      {
10       array[location + 1] = array[location];
11       location--;
```

```
12      }
13      array[location + 1] = newElem;
14    }
15    //...
16  }
```

Входные данные: $array$ - массив элементов, n - количество элементов
Выходные данные: $array'$ - отсортированный массив элементов

Трудоёмкость алгоритма

Для i-того элемента существует i+1 возможных положений в уже отстортированном списке. Добавление i-того элемента к отсортированному списку требует по крайней мере одного сравнения. Для того, чтобы добраться до каждой из i+1 возможных положений будет требоваться от 1 до i сравнений. Тогда чтобы найти среднее число сравнений для вставки i-того элемента составим равенство
$A_i = \frac{1}{i+1}(i + \sum\limits_{p=1}^{i} p)$.
Так как $\sum\limits_{i=1}^{N} i = \frac{N(N+1)}{2}$, то $A_i = \frac{1}{i+1}(i + \frac{i(i+1)}{2}) = \frac{i}{2} + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}$.
Теперь нужно просуммировать этот результаты для N-1 элементов списка:
$A(N) = \sum\limits_{i=1}^{N-1} A_i = \sum\limits_{i=1}^{N-1} (\frac{i}{2} + 1 - \frac{1}{i+1}) = \sum\limits_{i=1}^{N-1} \frac{i}{2} + \sum\limits_{i=1}^{N-1} 1 - \sum\limits_{i=1}^{N-1} \frac{1}{i+1}$.
Так как $\sum\limits_{i=0}^{N-L} (i + L) = \sum\limits_{i=L}^{N} i$, то $\sum\limits_{i=1}^{N-1} \frac{1}{i+1} = \sum\limits_{i=2}^{N} \frac{1}{i}$ и так как $\sum\limits_{i=L}^{N} i = \sum\limits_{i=0}^{N} i - \sum\limits_{i=0}^{L-1} i$,
то $\sum\limits_{i=2}^{N} \frac{1}{i} = \sum\limits_{i=1}^{N} \frac{1}{i} - 1 \approx \ln(N) - 1$.
В конечном итоге
$A(N) \approx (\frac{1}{2}\frac{(N-1)N}{2}) + (N-1) - (\ln(N)-1) = (\frac{N^2-N}{4}) + (N-1) - (\ln(N)-1) =$

$= \frac{N^2+3N-4}{4} - (\ln(N) - 1) \approx \mathbf{\frac{N^2}{4}}$

Анализ лучшего случая

Лучшим является случай, когда исходный массив отсортирован. Внутреннее тело цикла не выполнится ни одного раза. При этом на каждой итерации будет производиться только одно сравнение i-того элемента с i-1 элементом.

Значит трудоёмкость лучшего случая:
$V(N) = \sum_{i=1}^{N-1} 1$

$f = 2 + (n-1)(2+2+2+1+2+3) = 12n - 10$

Анализ худшего случая

Худшим является случай, когда элементы исходного массива располагаются в убывающем порядке. Тогда чтобы поставить i-тый вставляемый элемент в начало результирующего отсортированного списка, нужно сравнить его с i предыдущими элементами. Трудоёмкость худшего случая:
$W(N) = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2-N}{2} \approx \mathbf{\frac{N^2}{2}}$

Таким образом, в лучшем случае трудоёмкость алгоритма сортировки вставками имеет линейную сложность, а в худшем случае - квадратичную.

Алгоритм сортировки пузырьком

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.

Листинг 2: Алгоритм сортировки пузырьком

```
1 public static long BubbleSort(int[] array, int n) {
2   int alreadyRight = n;
3   boolean swapped = true;
4   //...
5   while (swapped)
6   {
7     alreadyRight--;
```

```
8       swapped = false;
9       for (int i = 0; i < alreadyRight; i++)
10        if (array[i] > array[i+1])
11        {
12          int tmp = array[i];
13          array[i] = array[i+1];
14          array[i+1] = tmp;
15          swapped = true;
16        }
17    }
18    //...
19  }
```

Входные данные: $array$ - массив элементов, n - количество элементов
Выходные данные: $array'$ - отсортированный массив элементов

Трудоёмкость алгоритма

Будем считать, что появление момента, в который перестановка элементов осуществляться не будет, равновероятно в каждом из N-1 прохождений по циклу. Нужно посчитать сколько сравнений будет произведено в каждом из случаев.

Пусть $C(i)$ - число сравнений, выполненных на первых i проходах, тогда
$A(N) = \frac{1}{N-1} \sum_{i=1}^{N-1} C(i)$,

$C(i)$ выражается по формулам суммирования и равна $Ni - \frac{i^2+1}{2}$.
Подставив это выражение в уравнение для $A(N)$ получим $A(N) = \frac{2N^2-N}{6} \approx \mathbf{\frac{1}{3}N^2}$

Анализ лучшего случая

Лучшим является случай, когда массив уже отсортирован, но при этом N-1 сравнение (в цикле for) будет выполнено даже при отсутствии перестановок на первом проходе.
Значит трудоёмкость лучшего случая:
$V(N) = \sum_{i=1}^{N-1} 1$

$f = 2 + 1 + 2 + 2 + (n-1)(2+4) + 1 = 6n + 2$

Анализ худшего случая

Худшим является случай, когда элементы упорядочены в обратном порядке. При таком раскладе наибольший элемент, стоящий в начале, будет сравниваться и переставляться со всеми остальными элементами вплоть до конца массива.

На первом проходе будет выполнено N-1 сравнений, на следующем N-2 и так далее, следовательно в худшем случае число сравнений будет равно $W(N) = \sum_{i=N-1}^{1} i = \frac{N^2-N}{2} \approx \frac{1}{2}N^2$

Таким образом,как и в сортировке вставками в лучшем случае трудоёмкость алгоритма сортировки пузырьком имеет линейную сложность, а в худшем случае - квадратичную.

Алгоритм сортировки слиянием

Сортируемый массив разбивается на две части примерно одинакового размера. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).Два упорядоченных массива соединяются в один.

На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.

Листинг 3: Алгоритм сортировки слиянием

```
1
2 public static long MergeSort(int [] array,int n) {
3     //...
4   Merge(array, 0, n-1);
5   //...
6 }
7
8 public static void Merge(int[] array, int first, int last)
    {
9     int middle;
```

```
10      if (first < last)
11      {
12        middle = (first + last)/2;
13        Merge(array, first, middle);
14        Merge(array, middle+1, last);
15        MergeLists(array, first, middle, middle+1, last);
16      }
17
18  }
19
20  public static void MergeLists(int [] array, int start1, int
        end1, int start2, int end2) {
21    int finalStart = start1;
22    int finalEnd = end2;
23    int index = 0;
24    int[] result = new int[finalEnd-finalStart+1];
25    while(start1 <= end1 && start2 <= end2) {
26      if (array[start1] < array[start2]) {
27        result[index] = array[start1++];
28      }
29      else {
30        result[index] = array[start2++];
31      }
32      index++;
33    }
34
35    if (start1 <= end1)
36      for(int i = start1; i <= end1; i++) {
37        result[index++] = array[i];
38      }
39    else
40      for (int i = start2; i <= end2; i++) {
41        result[index++] = array[i];
42      }
43
44    index = 0;
45    for (int i = finalStart; i <= finalEnd; i++) {
46      array[i] = result[index++];
47    }
48
49  }
```

Функция MergeSort(int [ ] array,int n)

Входные данные: $array$ - массив элементов, n - количество элементов
Выходные данные: $array'$ - отсортированный массив элементов

Функция Merge(int[] array, int first, int last)
Разбивает список пополам, двигаясь вниз по рекурсии и на обратном пути соединяет отсортированные половинки списка.
Входные данные: $array$ - массив элементов, $first$ - индекс первого элемента в сортируемой части списка, $last$ - индекс последнего элемента в сортируемой части списка
Выходные данные: $array'$ - отсортированный массив элементов

Функция MergeLists(int [] array, int start1, int end1, int start2, int end2)
Упорядочиваниет элементы двух списков, объединяя их в один результирующий список.
Входные данные: $array$ - массив элементов, $start1, end1$ - первый и последний индекс первого списка, $start2, end2$ - первый и последний индекс второго списка
Выходные данные: $array'$ - отсортированный массив элементов

Трудоёмкость алгоритма

Для алгоритма сортировки справедливо рекуррентное соотношение: $A(N) = 2A(N/2) + O(N)$, где $O(N)$ тратится на слияние двух массивов длины N. Тогда

$A(N) = 2A(N/2)+O(N) = 4A(n/4)+2O(N) = ... = A(1)+log(N)O(N)$
Следовательно, сложность алгоритма сортировки слиянием равна $O(Nlog(N))$

Анализ лучшего случая

Сравнением элементов занимается процедура MergeLists, которая сливает два списка в один. Анализируя ее, можно понять, что наилучшим случаем будет случай, когда все элементы первого списка меньше, чем первый элемент второго списка. При этом алгоритм выполнит сравнений столько, сколько элементов в первом списке, т.е N/2.

Подставив $O(N) = N/2$ в рекуррентном соотношении из предыдущего пункта имеем $V(N) = 2V(N/2) + N/2 = 4V(N/4) + N/2 + N/2 = 8V(N/8) + N/2 + N/2 + N/2 = ... = NV(1) + log_2(N) * N/2 = \frac{N}{2}log_2(N)$

Анализ худшего случая

Худшим случаем в слиянии двух списков в один является случай, когда элементы в двух списках идут "через один"(например A = [1, 3], B = [2, 4]). Элементы в результирующий список заносятся поочередно, сначала из одного, затем из другого, следовательно производится N-1 сравнений, где N - сумма количеств элементов в обоих списках.

Подставим $O(N) = N - 1$ в рекуррентное соотношение и получим $W(N) = 2W(N/2) + N - 1 = 4W(N/4) + N - 2 + N - 1 = ... = NW(1) + Nlog_2(N) - \sum_{i=0}^{log_2(N-1)} 2^i = Nlog_2(N) - N + 1$

Таким образом, трудоемкость алгоритма и в лучшем, и в худшем случае имеет логарифмическую сложность. Следовательно, даже в худшем случае, сортировка слиянием эффективнее, чем сортировки пузырьком и вставками.

Из минусов стоит отметить, что для слияния списков требуется дополнительная память.

Эксперимент

В качестве эксперимента были произведены замеры времени для следующих случаев:

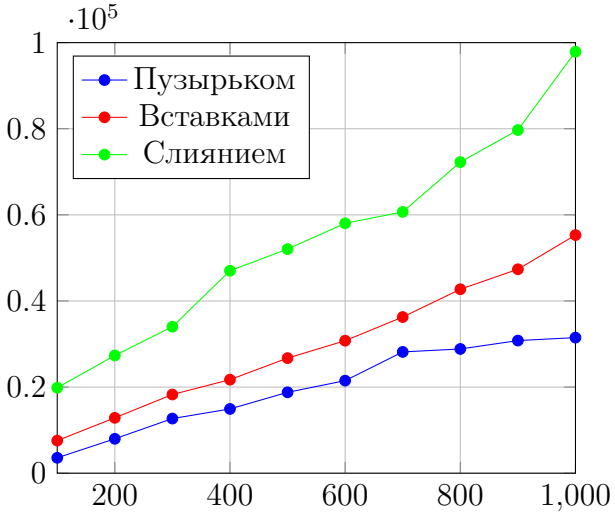
1. по 10 замеров для каждого из массивов по 100, 200, ..., 1000 элементов, заполненных **случайными** числами;
2. по 10 замеров для каждого из массивов по 100, 200, ..., 1000 элементов, заполненных **упорядоченными по возрастанию** числами;
3. по 10 замеров для каждого из массивов по 100, 200, ..., 1000 элементов, заполненных **упорядоченными по убыванию** числами.

*Время замерялось в наносекундах с помощью функции System.nanoTime().

Средние значения из 10 замеров представлены в таблицах и на графиках.

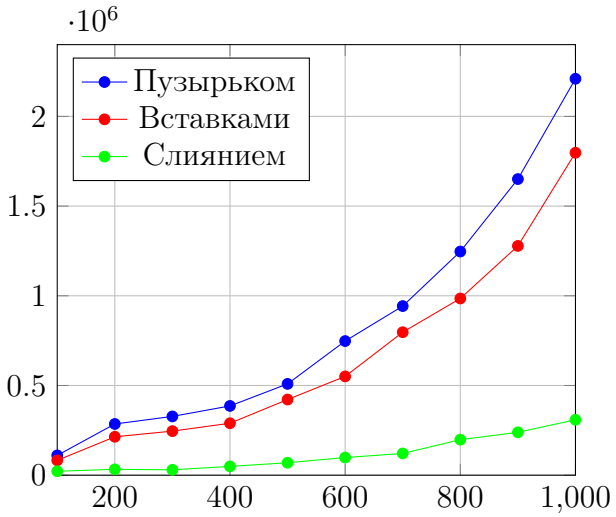
Лучший случай

Количество элементов	Сортировка пузырьком	Сортировка вставками	Сортировка слиянием
100	3575	7576	19853
200	7988	12861	27346
300	12687	18275	34023
400	14926	21728	47018
500	18784	26728	52053
600	21496	30774	58025
700	28168	36266	60688
800	28853	42724	72258
900	30799	47364	79711
1000	31493	55303	97889



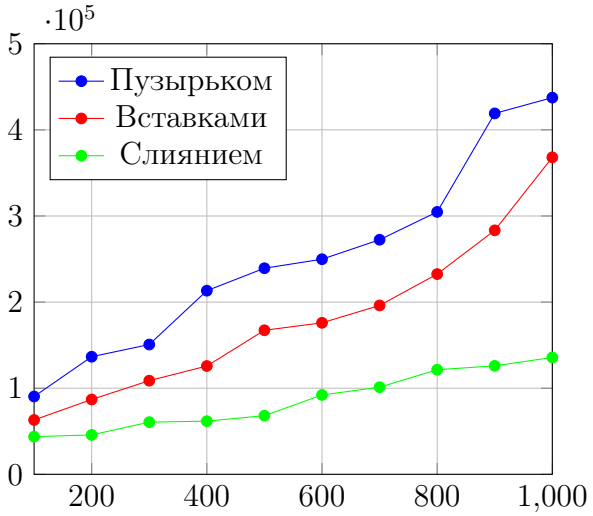
Худший случай

Количество элементов	Сортировка пузырьком	Сортировка вставками	Сортировка слиянием
100	110353	82773	21984
200	285465	214098	32715
300	327605	245703	29950
400	386112	289585	49027
500	508956	421718	69373
600	747590	550692	98495
700	942695	797022	121179
800	1246721	985040	198086
900	1650604	1277953	238993
1000	2209843	1797382	309063



Средний случай

Количество элементов	Сортировка пузырьком	Сортировка вставками	Сортировка слиянием
100	90358	63122	43763
200	136566	86900	45745
300	150732	108751	60604
400	213239	125774	61735
500	239333	167273	68122
600	249795	175940	92155
700	272405	196083	101175
800	304712	232538	121471
900	419114	283331	125962
1000	437437	368130	135746



Выводы из экспериментов

В результате экспериментов были подтверждены результаты теоретической оценки алгоритмов. В полностью отсортированном массиве сортировка слиянием работает дольше, так как она имеет логарифмическую сложность, в то время как сортировка пузырьком и вставками - линейную. В среднем и худшем случае сортировка слиянием работает намного быстрее двух других, так как она имеет логарифмическую сложность, а алгоритмы сортировки вставками и пузырьком - квадратичную. При этом подтвержден факт, что сортировка вставками работает быстрее сортировки пузырьком примерно 1.35-1.5 раза.

Заключение

В ходе лабораторной работы я изучила работу алгоритмов сортировки, выполнила полную математическую оценку трудоёмкости для алгоритма сортировки вставками с указанием лучшего, среднего и худшего случаев, выполнила краткую математическую оценку трудоёмкости для алгоритмов сортировки пузырьком и слиянием с указанием лучшего, среднего и худшего случаев, реализовала три алгоритма сортировки на языке программирования Java и сравнила работу этих алгоритмов сортировок.