ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ  
  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ   
БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ   
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»  
  
ФАКУЛЬТЕТ ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра вычислительных систем

**ОТЧЕТ  
ПО ЛАБОРАТОРНОЙ РАБОТЕ №1**

Выполнил:  
студент группы ИВ-521  
Прокопенко Р. П.  
  
Проверил:  
доцент кафедры вычислительных систем  
Курносов М. Г.

Оценка – «\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_».

Новосибирск - 2016

**Вариант 12**

**Постановка задачи**

Необходимо реализовать 3 алгоритма сортировок: Radix Sort, Odd-Even Sort, Quick Sort. Код данных алгоритмов и некоторые их характеристики приведены ниже.

**Описание алгоритмов**

**Radix Sort:**

Массив несколько раз перебирается и элементы перегруппировываются в зависимости от того, какая цифра находится в определённом разряде. После обработки разрядов (всех или почти всех) массив оказывается упорядоченным.

При этом разряды могут обрабатываться в противоположных направлениях — от младших к старшим или наоборот.

В данной работе сделан перебор от младших разрядов к старшим.

Элементы перебираются по порядку и группируются по самому младшему разряду (сначала все, заканчивающиеся на 0, затем заканчивающиеся на 1, …, заканчивающиеся на 9). Возникает новая последовательность. Затем группируются по следующему разряду с конца, затем по следующему и т.д. пока не будут перебраны все разряды, от младших к старшим.

Точное название способа LSD radix sort (Least significant digit radix sorts) — поразрядная сортировка по наименьшей значащей цифре.

В худшем случае:

- вычислительная сложность алгоритма по времени будет *O*(*k* x *n*);

- вычислительная сложность алгоритма по памяти будет *О*(*k* + *n*).

void radixSort(int \*a, int n) {

int i;

int semiSorted[n];

int significantDigit = 1;

int largestNum = findLargestNum(a, n);

while (largestNum / significantDigit > 0) {

int bucket[10] = { 0 };

for (i = 0; i < n; i++)

bucket[(a[i] / significantDigit) % 10]++;

for (i = 1; i < 10; i++)

bucket[i] += bucket[i - 1];

for (i = n - 1; i >= 0; i--)

semiSorted[--bucket[(a[i] / significantDigit) % 10]] = a[i];

for (i = 0; i < n; i++)

a[i] = semiSorted[i];

significantDigit \*= 10;

}

}

int findLargestNum(int \*a, int n) { // поиск наибольшего элемента в массиве

int i;

int largestNum = -1;

for (i = 0; i < n; i++) {

if (a[i] > largestNum)

largestNum = a[i];

}

return largestNum;

}

**Odd-Even Sort:**

Производится многократный прогон по массиву, соседние элементы сравниваются и, в случае необходимости, меняются местами. В отличие от пузырьковой сортировки шаг по массиву равен двум, а не единице.

Сначала элементы с нечётными индексами сравниваются/обмениваются с элементами с чётными индексами (1-й со 2-м, 3-й с 4-м, 5-й с 6-м и т.д.). Затем элементы с чётными индексами сравниваются/обмениваются с соседними элементами с нечётными индексами (2-й с 3-м, 4-й с 5-м, 6-й с 7-м и т.д.). Затем снова нечётные сравниваются с чётными, потом снова чётные с нечётными и т.д. Процесс завершается если в результате двух прогонов не происходило обменов — значит массив упорядочен.

В худшем случае:

- вычислительная сложность алгоритма по времени будет *O*(*n*2);

- вычислительная сложность алгоритма по памяти будет *О*(*n*).

void OddEven(int \*a, int n) { // четно-нечетная сортировка

int sorted = 0, i, temp;

while (sorted != 1) {

sorted = 1;

for (i = 1; i < n - 1; i += 2) {

if (a[i] > a[i + 1]) {

swap(a, i, i + 1);

sorted = 0;

}

}

for(i = 0; i < n - 1; i += 2) {

if(a[i] > a[i + 1]) {

swap(a, i, i + 1);

sorted = 0;

}

}

}

}

void swap(int \*a, int i, int j) { // меняем элементы местами

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

**Quick Sort:**

Алгоритм основан на подходе "разделяй-и-властвуй". Общая схема такова:

1. из массива выбирается некоторый опорный элемент a[i];

2. запускается процелура разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], влево от него, а все ключи, большие, либо равные a[i] - вправо;

3. теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правого,

http://algolist.manual.ru/sort/gif/22.gif

4. для обоих подмассивов: если в массиве более двух элементов, рекурсивно запускаем для него ту же процедура.

В конце получится полностью отсортированная последовательность.

В худшем случае:

- вычислительная сложность алгоритма по времени будет *O*(*n*2);

- вычислительная сложность алгоритма по памяти будет *О*(*n*).

void Quick(int \*a, int l, int r) { // быстрая сортировка

int i, j;

int x, buf;

i = l;

j = r;

x = a[(l + r) / 2];

do {

while (a[i] < x)

i++;

while (x < a[j])

j--;

if (i <= j) {

buf = a[i];

a[i] = a[j];

a[j] = buf;

i++;

j--;

}

}

while ( i <= j);

if (l < j)

Quick(a, l, j);

if (r > i)

Quick(a, i, r);

}

**Организация экспериментов**

Эксперименты проводились на ноутбуке Samsung RC530

(CPU: Intel Core i5-2430M, RAM 6GB);

Операционная система Windows 7 Домашняя базовая;

Среда разработки Dev C++ v5.4.2 (компилятор gcc 4.7.1 64-bit)

Ключи компиляции программы: -Wall -o

**Результаты экспериментов**

Измерим время работы каждого из алгоритмов при различном количестве элементов в массиве. Заполним для каждого алгоритма таблицу.

Radix Sort:

|  |  |  |
| --- | --- | --- |
| **#** | **Количество элементов в массиве** | **Время выполнения алгоритма, с** |
| 1 | 50000 | 0.009801 |
| 2 | 100000 | 0.016501 |
| 3 | 150000 | 0.024801 |
| 4 | 200000 | 0.032902 |
| 5 | 250000 | 0.041402 |
| 6 | 300000 | 0.050803 |
| 7 | 350000 | 0.059003 |
| 8 | 400000 | 0.064604 |
| 9 | 450000 | 0.076604 |
| 10 | 500000 | 0.084405 |
| 11 | 550000 | 0.091505 |
| 12 | 600000 | 0.099606 |
| 13 | 650000 | 0.109706 |
| 14 | 700000 | 0.117407 |
| 15 | 750000 | 0.129807 |
| 16 | 800000 | 0.141307 |
| 17 | 850000 | 0.149508 |
| 18 | 900000 | 0.155708 |
| 19 | 950000 | 0.163509 |
| 20 | 1000000 | 0.175709 |

Odd-Even Sort:

|  |  |  |
| --- | --- | --- |
| **#** | **Количество элементов в массиве** | **Время выполнения алгоритма, с** |
| 1 | 50000 | 10.188582 |
| 2 | 100000 | 41.182356 |
| 3 | 150000 | 93.404343 |
| 4 | 200000 | 166.087500 |
| 5 | 250000 | 317.844179 |
| 6 | 300000 | 437.341015 |
| 7 | 350000 | 592.691900 |
| 8 | 400000 | 809.843321 |
| 9 | 450000 | 1496.597600 |
| 10 | 500000 | 1570.314817 |
| 11 | 550000 | 1689.843653 |
| 12 | 600000 | 1859.428353 |
| 13 | 650000 | 2194.378511 |
| 14 | 700000 | 3509.762747 |
| 15 | 750000 | 3639.212151 |
| 16 | 800000 | 4872.809709 |
| 17 | 850000 | 5039.035217 |
| 18 | 900000 | 5510.547185 |
| 19 | 950000 | 6459.071437 |
| 20 | 1000000 | 8035.163585 |

Quick Sort:

|  |  |  |
| --- | --- | --- |
| **#** | **Количество элементов в массиве** | **Время выполнения алгоритма, с** |
| 1 | 50000 | 0.008801 |
| 2 | 100000 | 0.016601 |
| 3 | 150000 | 0.024802 |
| 4 | 200000 | 0.033602 |
| 5 | 250000 | 0.043603 |
| 6 | 300000 | 0.049803 |
| 7 | 350000 | 0.058804 |
| 8 | 400000 | 0.063404 |
| 9 | 450000 | 0.075204 |
| 10 | 500000 | 0.086005 |
| 11 | 550000 | 0.092014 |
| 12 | 600000 | 0.099805 |
| 13 | 650000 | 0.109606 |
| 14 | 700000 | 0.121407 |
| 15 | 750000 | 0.127007 |
| 16 | 800000 | 0.138608 |
| 17 | 850000 | 0.144008 |
| 18 | 900000 | 0.146808 |
| 19 | 950000 | 0.160609 |
| 20 | 1000000 | 0.164410 |

С помощью gnuplot построим графики зависимости времени сортировок от количества элементов в сортируемом массиве.

Radix Sort:

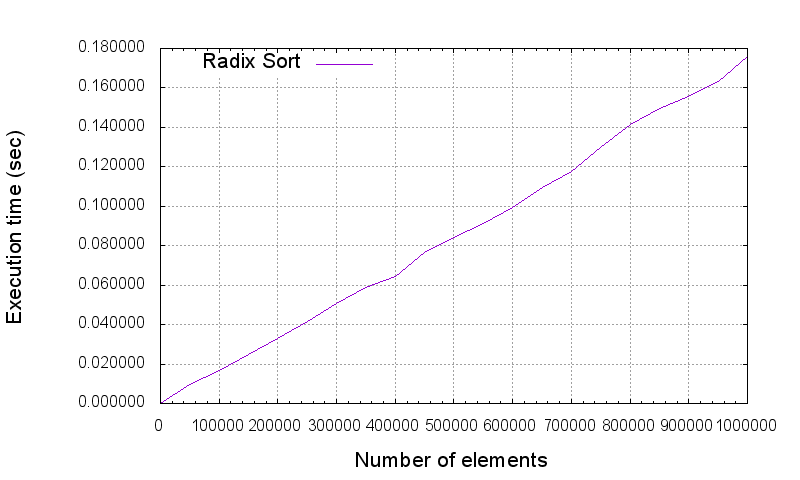
****

Рисунок - Зависимость времени выполнения алгоритма Radix Sort от количества элементов массива

Odd-Even Sort:

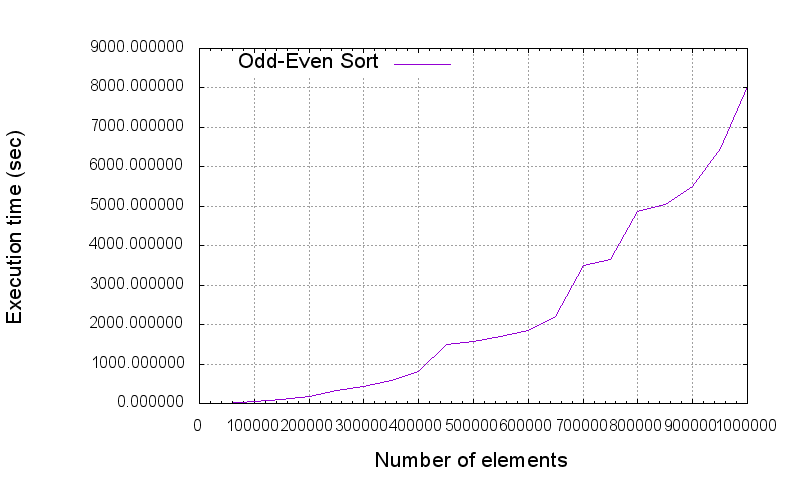
****

Рисунок - Зависимость времени выполнения алгоритма Odd-Even Sort от количества элементов массива

Quick Sort:

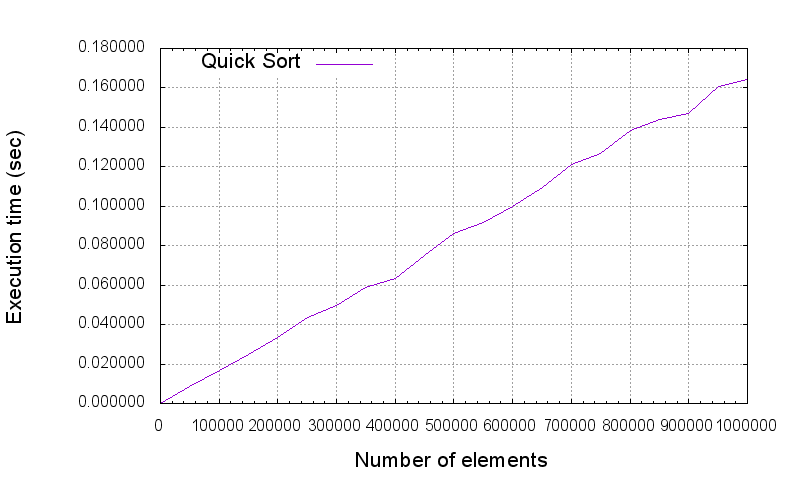
****

Рисунок - Зависимость времени выполнения алгоритма Quick Sort от количества элементов массива

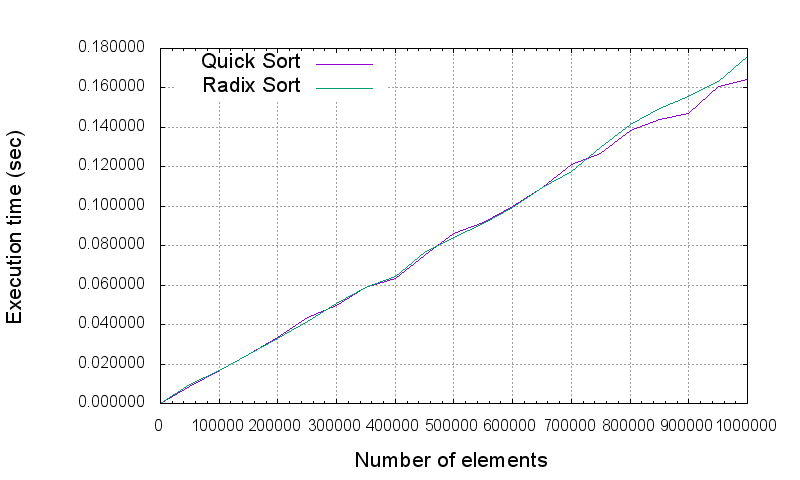
Построим на одном рисунке графики для Radix Sort и Quick Sort.

Рисунок - Сравнение эффективности выполнения алгоритмов Quick Sort и Radix Sort

Видим, что алгоритмы работают практически одинаково эффективно, но Quick Sort отработал более быстро чем Radix Sort.

Построим на одном рисунке графики для всех трех сортировок.

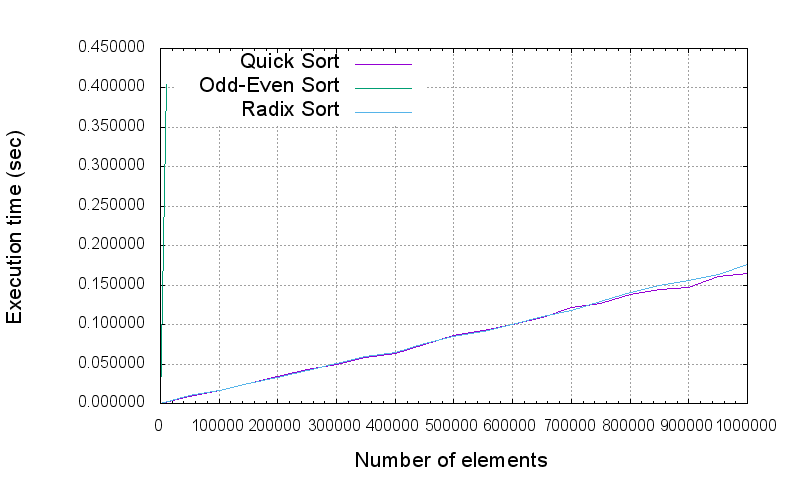
****

Рисунок - Сравнение эффективности выполнения алгоритмов Quick Sort, Odd-Even Sort и Radix Sort

Ввиду того, что алгоритм Odd-Even Sort работает крайне неэффективно и долго сортирует элементы массива, построен график, где для алгоритма Odd-Even Sort количество элементов массива взято 10000, а для алгоритмов Quick Sort и Radix Sort взяты все 20 контрольных точек (50000-1000000 элементов), т.к. они сортируют даже такие огромные массивы за доли секунды, ввиду своей эффективности. Если для Odd-Even брать значения более 10000 элементов, то графики двух других сортировок буду практически не видны.

Из проведенных экспериментов следует вывод, что алгоритмы по эффективности идут в следующем порядке:

1. Quick Sort;

2. Radix Sort;

3. Odd-Even Sort.

Хотя следует отметить, что Quick Sort и Radix Sort очень близки по своей эффективности.

Алгоритм Odd-Even Sort работает так неэффективно в связи с тем, что в ходе его работы элементы массива перебираются от первого и до последнего огромное количество раз(сначала четные, затем нечетные) и меняются местами элементы, стоящие не на своих местах. Работа прекращается лишь после обоих проходов алгоритма по четным и нечетным элементам и если обмена элементов не произошло, что свидетельствует о том, что все элементы стоят на своих местах.