Отчет о выполнении задания 1  
«Алгоритмы сортировки»

**Меняйлов Владислав Олегович**

Группа ИС-641

# Описание алгоритмов

В задании 1 требовалось реализовать и исследовать эффективность трех алгоритмов сортировки: Radix Sort, Selection Sort и HeapSort. Все алгоритмы реализованы на языке C под операционной системой GNU/Linux.

Radix Sort.- Числа сортируются по разрядам. Существует два варианта: least significant digit (LSD) и most significant digit (MSD). При LSD сортировке, сначала сортируются младшие разряды, затем старшие. При MSD сортировке все наоборот. При LSD сортировке получается следующий порядок: короткие ключи идут раньше длинных, ключи одного размера сортируются по алфавиту, это совпадает с нормальным представлением чисел: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. При MSD сортировке получается алфавитный порядок, который подходит для сортировки строк. 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 разной длины, то получим последовательность 1, 10, 2, 3, 4, 5, 6, 7, 8, 9.

Код на С.

|  |
| --- |
| #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <sys/time.h> |
|  | #define size 1000000 |
|  |  |
|  | double wtime() |
|  | { |
|  | struct timeval t; |
|  | gettimeofday(&t, NULL); |
|  | return (double)t.tv\_sec + (double)t.tv\_usec \* 1E-6; |
|  | } |
|  |  |
|  | int getrand(int min, int max) |
|  | { |
|  | return (double)rand() / (RAND\_MAX + 1.0) \* (max - min) + min; |
|  | } |
|  |  |
|  | int findLargestNum(int \* array){ |
|  |  |
|  | int i; |
|  | int largestNum = -1; |
|  |  |
|  | for(i = 0; i < size; i++){ |
|  | if(array[i] > largestNum) |
|  | largestNum = array[i]; |
|  | } |
|  |  |
|  | return largestNum; |
|  | } |
|  |  |
|  | void radixSort(int \* array){ |
|  | double t,g; |
|  | FILE \*file; |
|  | file=fopen("file1.dat","w"); |
|  | int i,l=50000; |
|  | t = wtime(); |
|  | int \*semiSorted=(int\*)malloc(size\*sizeof(int)); |
|  | int significantDigit = 1; |
|  | int largestNum = findLargestNum(array); |
|  |  |
|  | while (largestNum / significantDigit > 0){ |
|  | int \*bucket=(int\*)malloc(size\*sizeof(int)); |
|  | for (i = 0; i < size; i++) |
|  | bucket[(array[i] / significantDigit) % 10]++; |
|  |  |
|  | for (i = 1; i < size; i++) |
|  | bucket[i] += bucket[i - 1]; |
|  |  |
|  | for (i = size - 1; i >= 0; i--) |
|  | semiSorted[--bucket[(array[i] / significantDigit) % 10]] = array[i]; |
|  |  |
|  |  |
|  | for (i = 0; i < size; i++){ |
|  | if (i==l-1){ |
|  | g=t; |
|  | t = wtime() - t; |
|  | fprintf(file,"%d %.6f\n",l,t); |
|  | printf("%.6f\n",t); |
|  | t=g; |
|  | l=l+50000; |
|  | } |
|  | array[i] = semiSorted[i]; |
|  | } |
|  |  |
|  | significantDigit \*= 10; |
|  | } |
|  | fclose(file); |
|  | } |
|  |  |
|  | int main(){ |
|  | int i; |
|  | int \*list=(int\*)malloc(size\*sizeof(int)); |
|  | for (i=0;i<size;i++){ |
|  | list[i]=getrand(1,1001); |
|  | } |
|  | radixSort(&list[0]); |
|  | printf("\n"); |
|  |  |
|  | return 0; |
|  | } |

Selection sort—алгоритм сортировки. Может быть как устойчивый, так и неустойчивый. На массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае Θ(n2), предполагая что сравнения делаются за постоянное время.

Код на С

|  |
| --- |
| #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <sys/time.h> |
|  | #define size 1000000 |
|  | double wtime() |
|  | { |
|  | struct timeval t; |
|  | gettimeofday(&t, NULL); |
|  | return (double)t.tv\_sec + (double)t.tv\_usec \* 1E-6; |
|  | } |
|  |  |
|  | int getrand(int min, int max) |
|  | { |
|  | return (double)rand() / (RAND\_MAX + 1.0) \* (max - min) + min; |
|  | } |
|  |  |
|  |  |
|  |  |
|  | int main(){ |
|  | double t,g; |
|  | int i,j,minindex,temp,l=50000; |
|  | int \*A=(int\*)malloc(size\*sizeof(int)); |
|  |  |
|  | for (i=0;i<size;i++){ |
|  | A[i]=getrand(1,1001); |
|  | } |
|  | t = wtime(); |
|  | FILE \*file; |
|  | file=fopen("file1.dat","w"); |
|  | for (i=0;i<size-1;i++){ |
|  | minindex=i; |
|  | for (j=i+1;j<size;j++){ |
|  | if (A[j]<A[minindex]){ |
|  | minindex=j; |
|  | } |
|  | } |
|  | if (minindex != i){ |
|  | temp=A[i]; |
|  | A[i]=A[minindex]; |
|  | A[minindex]=temp; |
|  | } |
|  | if (i == l-2){ |
|  | g=t; |
|  | t = wtime() - t; |
|  | fprintf(file,"%d %.6f\n",l,t); |
|  | printf("%.6f\n",t); |
|  | t=g; |
|  | l=l+50000; |
|  | } |
|  | } |
|  | fclose(file); |
|  |  |
|  | return 0; |
|  | } |

Heap sort— алгоритм сортировки, работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за Θ(n log n) операций при сортировке n элементов.[2] Количество применяемой служебной памяти не зависит от размера массива (то есть, O(1)).

Код на С

|  |
| --- |
| #include<stdio.h> |
|  | #include <stdlib.h> |
|  | #include <sys/time.h> |
|  | #define MAX 1000000 |
|  | double wtime() |
|  | { |
|  | struct timeval t; |
|  | gettimeofday(&t, NULL); |
|  | return (double)t.tv\_sec + (double)t.tv\_usec \* 1E-6; |
|  | } |
|  | int getrand(int min, int max) |
|  | { |
|  | return (double)rand() / (RAND\_MAX + 1.0) \* (max - min) + min; |
|  | } |
|  | void max\_heapify(int a[], int i, int heapsize) |
|  | { |
|  | int tmp, largest; |
|  | int l = (2 \* i) + 1; |
|  | int r = (2 \* i) + 2; |
|  | if ((l <= heapsize) && (a[l] > a[i])) |
|  | largest = l; |
|  | else |
|  | largest = i; |
|  | if ((r <= heapsize) && (a[r] > a[largest])) |
|  | largest = r ; |
|  | if (largest != i) |
|  | { |
|  | tmp = a[i]; |
|  | a[i] = a[largest]; |
|  | a[largest] = tmp; |
|  | max\_heapify(a, largest, heapsize); |
|  | } |
|  |  |
|  | } |
|  | void build\_max\_heap(int a[], int heapsize) |
|  | { |
|  | int i; |
|  | for (i = heapsize/2; i >= 0; i--) |
|  | { |
|  | max\_heapify(a, i, heapsize); |
|  | } |
|  |  |
|  | } |
|  | void heap\_sort(int a[], int heapsize) |
|  | { |
|  | double t; |
|  | int g; |
|  | int log=50000; |
|  | t = wtime(); |
|  | int i, tmp,l=950000; |
|  | build\_max\_heap(a, heapsize); |
|  | FILE \*file; |
|  | file=fopen("file1.dat","w"); |
|  | for (i = heapsize; i > 0; i--) |
|  | { |
|  | if (i==l){ |
|  | log=log+50000; |
|  | g=t; |
|  | t = wtime() - t; |
|  | fprintf(file,"%d %.6f\n",log,t); |
|  | printf("%.6f\n",t); |
|  | t=g; |
|  | l=l-50000; |
|  |  |
|  | } |
|  | tmp = a[i]; |
|  | a[i] = a[0]; |
|  | a[0] = tmp; |
|  | heapsize--; |
|  | max\_heapify(a, 0, heapsize); |
|  | } |
|  | fclose(file); |
|  | } |
|  | int main() |
|  | { |
|  | int i, r, heapsize; |
|  | int \*a=(int\*)malloc(MAX\*sizeof(int)); |
|  | double t; |
|  | for (i=0;i<MAX;i++){ |
|  | a[i]=getrand(1,1001); |
|  | } |
|  | heapsize = MAX - 1; |
|  | heap\_sort(a, heapsize); |
|  |  |
|  | return 0; |
|  | } |

# Организация экспериментов

* Эксперименты проводились на ноутбуке Samsung   
  (Intel(R) Core(TM) i3-3110M CPU @2.40GHz 2.40GHz Ozu 4gb
* Операционная система Opensuze 20 x86\_64 (компилятор gcc 4.8.2)
* Ключи компиляции программы (см. README): -Wall -O2

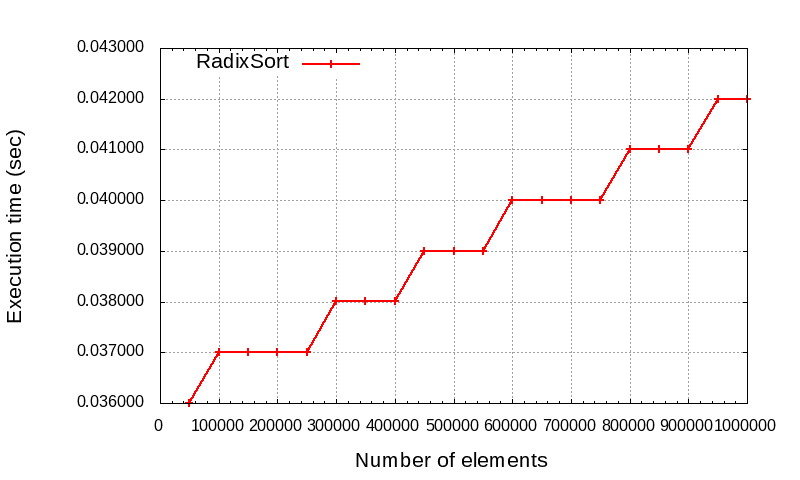
# Результаты экспериментов

**Radix sort**

На рис. показаны графики зависимости Кол-во элементов в массиве от времени

*Radix sort*

|  |  |  |
| --- | --- | --- |
| **#** | **Количество элементов в массиве** | **Время выполнения алгоритма, с** |
| 1 | 50 000 | 0,036002 |
| 2 | 100 000 | 0,037002 |
| 3 | 150 000 | 0,037002 |
| ... | ... |  |
| 20 | 1 000 000 | 0,042002 |
|  |  |  |

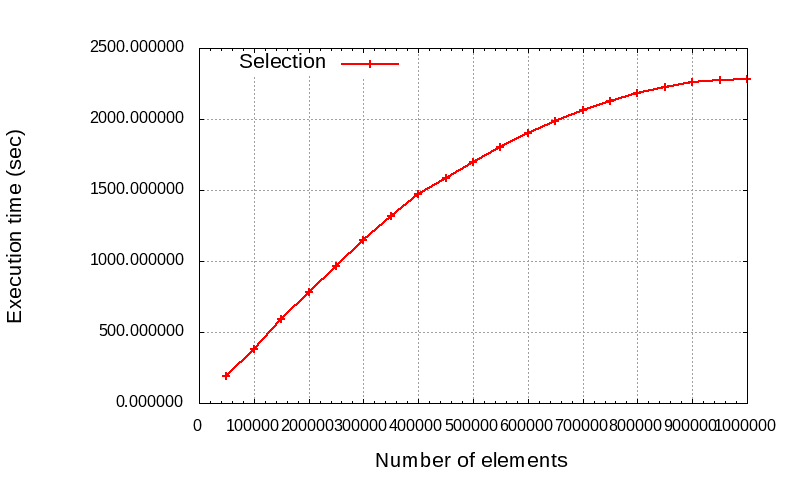


**Selection sort**

На рис. показаны графики зависимости Кол-во элементов в массиве от времени

*Selection sort*

|  |  |  |
| --- | --- | --- |
| **#** | **Количество элементов в массиве** | **Время выполнения алгоритма, с** |
| 1 | 50 000 | 188.576120 |
| 2 | 100 000 | 381.346541 |
| 3 | 150 000 | 590.588618 |
| ... | ... |  |
| 20 | 1 000 000 | 2282.125928 |



**Heap sort**

На рис. показаны графики зависимости Кол-во элементов в массиве от времени

*Heap sort*

|  |  |  |
| --- | --- | --- |
| **#** | **Количество элементов в массиве** | **Время выполнения алгоритма, с** |
| 1 | 50 000 | 0.066589 |
| 2 | 100 000 | 1.048164 |
| 3 | 150 000 | 1.079475 |
| ... | ... |  |
| 20 | 1 000 000 | 1.569195 |
|  |  |  |

