University of Washington Bothell
CSS 342: Data Structures, Algorithms, and Discrete Mathematics
Program 3: Sorting

Purpose

This lab will serve multiple purposes. First, it will provide hands-on experience for utilizing many of the sorting algorithms we will introduce in the class. Second, it will demonstrate the cost of O(n^2) v. O(n logn) algorithms. It will also clearly show that algorithms with the same complexity may have different running times. In addition, even an O(n logn) algorithm may not perform to the expected complexity if the implementation is not careful with memory management. You will have to implement the algorithms in the most efficient manner and to make sure the theoretical complexity is observed in testing for large values of n. Finally, you will need to collect and present data in a reasonable way.
Note: this is NOT an object oriented programming assignment.

Problem:

Write a program which implements the following sorts and compares the performance for operations on arrays of integers of growing sizes 10, 100, 1000, 5000, 10000, 25000, etc…. It is recommended that you use vectors of integers instead of arrays. The word "array" would be used to describe a list of integers from here on.
Graph the performance of the different sorts as a function of the size of the array.

1) BubbleSort
2) InsertionSort
3) MergeSort
4) Non-Recursive, one extra array MergeSort (we'll call this improved version, IterativeMergeSort from here on out in this homework)
5) QuickSort
6) ShellSort

Details:

IterativeMergeSort

In-place sorting refers to sorting that does not require extra memory arrays. For example, QuickSort performs partitioning operations by repeating a swapping operation on two data items in a given array. This does not require an extra array.

MergeSort as shown in Carrano allocates a temporary array at each recursive call. Due to this MergeSort is slower than QuickSort even though their running time is upper-bounded to

O(n log n).

We can improve the performance of MergeSort by utilizing a non-recursive method and using only one additional array (instead of one array on each recursive call). In this improved version of MergeSort, IterativeMergeSort, one would merge data from the original array into the additional array and **alternatively copy back and forth between the original and the additional temporary array**.

For the IterativeMergeSort we still need to allow data items to be copied between the original and this additional array as many times as O(log n). However, given the iterative nature we are not building up state on the stack.

Other Sorts

BubbleSort, InsertionSort, MergeSort, and QuickSort are well documented and you should implement them with the aid of examples in the Carrano book. IterativeMergeSort and ShellSort will be presented in class.

Input Parameters

All the sorting functions would have an identical calling signature:
Sort(array, index1, index2);
After calling Sort, array would be modified to become sorted between index1 and index2.
For example, take array1 of size 29:

```
BubbleSort(array1, 0, 28); // would use BubbleSort to fully sort
array1. After the call array1 would be sorted.

InsertionSort(array1, 10, 20); // would use InsertionSort to sort
array1 between the indices 10 and 20. After the call, array1
would be sorted only on that segment.
```

Runtime Details

Your driver program, called Sorter, will take in two or three parameters:
1) sort type as a string of characters
2) an array size as an integer.
3) NO/YES corresponding to, do not print the array (NO), or print the array (YES or no input).

Your program will create and sort an integer array of the size with the specified sort: MergeSort, BubbleSort, InsertionSort, QuickSort, IterativeMergeSort, or ShellSort. The driver functions below will help with the creation.

Examples:

Sorter MergeSort 100 (creates and sorts an array of 100 using MergeSort, prints)

Sorter QuickSort 1000 (creates and sorts an array of 1000 using QuickSort, prints)
Sorter IterativeMergeSort 10 (creates and sorts an array of 10 using the non-recursive semi-in-place MergeSort, prints)

Here is an example of how you would execute the code and the required output:

```
./Sorter.exe QuickSort 5
Initial:
items[0] = 0
items[1] = 1
items[2] = 2
items[3] = 3
items[4] = 4
Sorted:
item[0] = 0
item[1] = 1
item[2] = 2
item[3] = 3
item[4] = 4
Time (ms): 7

./Sorter.exe QuickSort 1000 NO
Time (ms): 4618
```

```
Note: use call by reference in all sorting methods.
```

What to turn in:

Turn in, in a .zip:

(1) Your SortImpls.cpp file which has implementations of all of the sorts, and the driver file Sorter.cpp (see below).
(2) Your executable on a Linux machine: Sorter.exe
(3) A separate report in word or pdf which includes: Graphs that compares the performance among the different sorting algorithms with increasing data size. You should increase the data size to clearly show the difference in performance of the different sorts. Make sure the algorithm delivers the theoretical complexity for large n. You may need to go back and modify your implementation based on the results.
To prepare the report you may need to write a script that would run all the sorting algorithms for different arrays size, and for different type of initial condition (random, ordered, reverse ordering, partially sorted).

Driver Code

Use the code below as driver code to test and time the different Sort functions. Notice the sort definitions are in the SortImpls.cpp file that is included at the top of the driver file. This is the file you will compile and run. Please make sure to spell each of the sorts by the exact name signature below. The driver code below is only an example. You are responsible for your own testing code.

Driver code example (Linux)

```
#include <iostream>
#include <vector>
#include <string>
#include "SortImpls.cpp"
#include <sys/time.h>
//#include <windows.h>
using namespace std;
void InitArray(vector<int> &, int);
void InitArrayMid(vector<int> &, int);
void PrintArrayDetails(const vector<int> &, string);
int elapsed( timeval &startTime, timeval &endTime);
int main(int argc, char *argv[])
{
  int size = 0;
  string sort_name = "";
  bool printOut = true;
  if ((argc != 3) && (argc != 4))
  {
    cerr << "Usage: Sorter SORT_TYPE ARRAY_SIZE [YES|NO]" <<
endl;
    return -1;
  }
  sort_name = string(argv[1]);
  size = atoi(argv[2]);
  if (size <= 0)
  {
    cerr << "Array size must be positive" << endl;
    return -1;
  } if (
    argc == 4)
    {
      string printArr = string(argv[3]);

      if (printArr == "NO")
      {
        printOut = false;
      }
      else if (printArr == "YES")
      {
        printOut = true;
      }
      else
      {
        cerr << "Usage: Sorter SORT_TYPE ARRAY_SIZE [YES|NO]" <<
endl;
        return -1;
      }
    }
    srand(1);
    vector<int> items(size);
    InitArrayMid(items, size);
    if (printOut)
    {
      cout << "Initial:" << endl;
      PrintArrayDetails(items, string("items"));
    }
    struct timeval startTime, endTime;
    gettimeofday(&startTime, 0); // Linux
    //int begin = GetTickCount();  // GetTickCount is windows
specific.
    //
    // PLACE YOUR CODE HERE
    if (sort_name == "SelectionSort")
    {
      SelectionSort(items, 0, size - 1);
    }
    if (sort_name == "BubbleSort")
    {
      BubbleSort(items, 0, size - 1);
    }
    if (sort_name == "InsertionSort")
    {
      InsertionSort(items, 0, size - 1);
    }
    if (sort_name == "MergeSort")
    {
      MergeSort(items, 0, size - 1);
    }
    if (sort_name == "IterativeMergeSort")
    {
      IterativeMergeSort(items, 0, size - 1);
    }
    if (sort_name == "QuickSort")
    {
      QuickSort(items, 0, size - 1);
    }
    if (sort_name == "ShellSort")
    {
      ShellSort(items, 0, size - 1);
```

```
  }
  gettimeofday(&endTime, 0);
  if (printOut)
  {
    cout << "Sorted:" << endl;
    PrintArrayDetails(items, string("item"));
  }
  int elapsed_secs = elapsed(startTime,endTime);
  cout << "Time (ms): " << elapsed_secs << endl;
  return 0;
}
void InitArray(vector<int> &array, int randMax)
{
  if (randMax < 0)
  {
    return;
  }
  vector<int> pool(randMax);
  for (int i = 0; i < randMax; i++)
  {
    pool[i] = i;
  }
  int spot;
  for (int i = 0; i < randMax; i++)
  {
    spot = rand() % (pool.size());
    array[i] = pool[spot];
    pool.erase(pool.begin() + spot);
  }
}
void InitArrayMid(vector<int> &array, int randMax)
{
  if (randMax < 0)
  {
    return;
  }
  vector<int> pool(randMax);
  for (int i = 0; i < randMax; i++)
  {
    pool[i] = i;
    array[i] = i;
  }
  int spot;
  int i1 = 0.49*randMax;
  int i2 = 0.51*randMax;
  for (int i = i1; i < i2; i++)
```

```
    {
      spot = rand() % (pool.size());
      array[i] = pool[spot];
      pool.erase(pool.begin() + spot);
    }
  }
  void PrintArrayDetails(const vector<int> &array, string name)
  {
    int size = array.size();
    for (int i = 0; i < size; i++)
    cout << name << "[" << i << "] = " << array[i] << endl;
  }
  // Function to calculate elapsed time if using gettimeofday
(Linux)
  int elapsed( timeval &startTime, timeval &endTime )
  {
    return ( endTime.tv_sec - startTime.tv_sec ) * 1000000 + (
endTime.tv_usec - startTime.tv_usec );
  }
```

Grading guidelines

1. Compiling on Linux 10 points
2. Grading code not compiling 10 points
   That could be due to a different command line then specified, or function arguments not matching, or not including the implementation file in the driver file as specified.
3. Defining classes and OOP concepts where it is not necessary 10 points
4. Efficiency 20 points.
   For example, copying or creating arrays when it is not necessary.
5. Document 20 points.
   Graphs should be clear, with data readable, and convergence behavior analysis (best case, worst case, average case, etc.). The document should be written as a lab report.
   There would be -5 points deduction for each occurrence of a convergence behavior that is not consistent with the theory, with no discussion explaining the source of the wrong behavior.
6. BubbeSort Correctness 5
7. InsertSort Correctness 5
8. MergeSort Correctness 5
9. QuickSort Correctness 5
10. IterativeMergeSort Correctness 5
11. ShellSort Correctness 5

Note: Correctness includes edge cases, empty arrays, reversed arrays, ordered arrays, array of length 29 for example, middle cases, etc. (Every method that is not handling middle cases -2 points.)