Sorting

Due: 23:59, Monday, February 22, 2016

In this assignment you are to implement four different sorting algorithms, and count the number of barometer operations performed by each algorithm. Your solution should not be OS-dependent; use only libraries that are part of the C++ standard (i.e. no `stdafx.h`). Please heed the notes below:

- Your sorting functions should be template functions that take a template variable for the type to be sorted (see examples near the end of this document)
- Each sorting function should return an integer equal to the number of barometer operations performed during the function execution; the barometer instruction is the instruction that is executed the greatest number of times. Please refer to Lab exercise #4 for information on counting barometer instructions.
- The functions should all be written in a sorting.cpp file containing the implementations of the sorting functions and their helpers declared in the driver file; you are not required to write any classes for this assignment.

Part 1 – Selection Sort

Write a template function called `SelectionSort` that sorts its array parameter using the selection sort algorithm. The function should have two parameters, an array of type T (where T is a template variable), and an integer that records the size of the array. The function should return an integer that equals the number of times its barometer comparison is made during its execution.

Part 2 – Quicksort

Write a template function called `Quicksort` that sorts its array parameter using the quicksort algorithm. The function should have two parameters, an array of type T (where T is a template variable), and an integer that records the size of the array. The function should return an integer that equals the number of times its barometer comparison is made during its execution.

The partition helper function `QSPartition` should use the *last* element of the subarray as the pivot value.

Part 3 – Randomized Quicksort

Write a template function called `RQuicksort` that sorts its array parameter using the quicksort algorithm. The function should have two parameters, an array of type T (where T is a template variable), and an integer that records the size of the array. The function should return an integer that equals the number of times its barometer comparison is made during its execution.

The partition helper function `RQSPartition` should perform a swap of the *last* element of a subarray with a random element of the subarray, followed by the same partition logic used for your basic Quicksort partition function.

Part 4 – Mergesort

Write a template function called `Mergesort` that sorts its array parameter using the Mergesort algorithm. The function should have two parameters, an array of type T (where T is a template variable), and an integer that records the size of the array. The function should return an integer that equals the number of times its barometer comparison is made during its execution.

Part 5 – Shell Sort

Write a template function called `ShellSort` that sorts its array parameter using the Shell sort algorithm. The function should have two parameters, an array of type T (where T is a template variable), and an integer that records the size of the array. The function should return an integer that equals the number of times its barometer comparison is made during its execution.

We have not covered the Shell sort algorithm in class, it is up to you to research how to write it. If you use code that you find on the web as your starting point you should cite the web page in comments above your `ShellSort` function. You should use an initial gap of size $n/2$ and divide the gap by two in each step (this sentence will make more sense when you've done more research).

Counting barometer operations

For Quicksort the number of barometer operations is comparisons in the partition function. This function should return the index of the pivot element, so it therefore cannot also return the number of comparisons it made! The way to deal with this is to add a reference parameter to the function that you can use to record the comparison count.

Here is an example (un-related to the assignment) of how you could count the amount of work performed by a function that returns the maximum value in an array:

```
int Max(int arr[], int n, int& comparisons)
{
  comparisons = 0;
  int maximum = arr[0];
  for (int i = 1; i < n; i++)
  {
    comparisons += 1;
    if (arr[i] > maximum)
      maximum = arr[i];
  }
  return maximum;
}
```

And here is a short test that shows how you can call this function (i.e. from a main method):

```
int arr100[100];
for (int i = 0; i < 100; i++)
{
  arr100[i] = i+1;
}
int count = 0;
cout << "max value = " << Max(arr100, 100, count) << endl;
cout << "comparisons = " << count << endl;
```

Running the sorting functions

Download the test driver, the sorting.cpp skeleton file and test input files from the course website. You are to write your templated sorting functions into sorting.cpp. If you are unable to complete any function, simply write a stub returning a default value of the correct type to ensure that the program compiles without errors. Your submitted code will be tested using a similar driver and a different set of test input files.

Deliverables

Submit a ZIP archive titled "assign-3.zip" which includes the following files:

- title page with names of contributing group members and the resources consulted
- sorting.cpp