2G03 Homework 7, 2020

Due: Thur, Nov 19th (Hard deadline Friday Nov 20th)
The files are available on phys-ugrad in `/home/2G03/HW7`.
Do this: `cp -r /home/2G03/HW7 ~/`
This will create a directory called *HW7*.
You should write C++ source files with the names indicated and a Makefile in this directory on phys-ugrad for credit. Submit digital text files with answers to questions on avenue.

Sorting

Exercise 1 `PROGRAM` *Sorting*

The goal of this exercise is to develop a library of functions that can sort an array of N real numbers for us. The name of the game is to do this using the smallest amount of operations. The best algorithms can do this in $N \log_2 N$ operations and at most we have to use N^2 operations. We will develop two different algorithms (both quite famous) for this purpose.

Below is a main program `testsort.cpp` that generates a sequence of random real numbers and can be used to test different sorting functions. The `switch` statement is used as an alternative to `if` that can handle several integer cases.

Note that the statement,

```
A[i] = float(rand())/RAND_MAX;
```

sets the entry i of the array A to a random real number between 0 and 1. We will not discuss random number generators so just use this as is. The rand function is defined in cstdlib.

1.1 Write a `Makefile` that will compile `testsort.cpp` and produce an executable called `testsort`. You do not need to change `testsort.cpp` yet. Confirm that this program compiles with `make testsort` and when you run `testsort` and select one of the 3 options, it just prints out the set of 10 numbers twice with no sorting.

Note: We do not call the executable program `sort` because it would conflict with the unix program that has the same name. Call it `testsort`. The unmodified testsort does no sorting at all. There are 4 lines commented out with `//` in `testsort.cpp`. One is the include statement for your sorting routines header file `sort.h`. The other three are calls to the functions that do the sorting. Later, once you have written code to sort, you can get rid of the `//` in front to have it use that sort. First, You will need a more complex Makefile to incorporate your sorting code.

file: /home/2G03/HW7/testsort.cpp

```
#include <iostream>
#include <cstdlib>
//#include "sort.h"
int main() {
// A Program for testing sorting algorithms
// Filename: testsort.cpp Src: JWW 11/2020
  const int n = 10;
  int i, isort;
  float A[n];
  for (i=0;i<n;i++) {
    A[i] = float(rand())/RAND_MAX;  // generate a random real number between 0 and 1
  }
  for (i=0;i<n;i++) {
    std::cout << A[i] << " ";
  }
  std::cout << " unsorted\n";
  std::cout << "Enter 1 for insertion sort, 2 for partition test, 3 for quick sort\n";
  std::cin >> isort;
  switch (isort) {
  case 1:
  //  InsertionSort( A, n );
    break;
  case 2:
  //  std::cout << "Count for small sub-array " << Partition( A, n );
    break;
  case 3:
  //  QuickSort(A,n);
    break;
  default:
    std::cout << isort << " is not an allowed choice\n";
  }
  for (i=0;i<n;i++) {
    std::cout << A[i] << " ";
  }
}
```

The `InsertionSort` function takes the array `A` and a size n as arguments. You can see how it is used by looking at `testsort.cpp`. Start with a placeholder, mostly empty function with braces `{ }` and a print to indicate the function was called. You will add the sorting code in a later step. You also need to generate a header in your `sort.h` file. This is the same but with a semi-colon `;` after the declaration instead of braces. Once these files exist you should extend your Makefile so it now compiles `sort.cpp` as well. The placeholder `sort.cpp` should look like this:

file: placeholder sort.cpp

```
#include <iostream>
#include "sort.h"

void InsertionSort( float A[], int n ) {
   std::cout << "Running Insertion Sort\n";

}
```

1.2 Create the files `sort.h` and `sort.cpp` with the above function `InsertionSort` and modify your Makefile so that it will compile them to produce `testsort` again. There is now text printed to the screen `Running Insertion Sort` when you select option 1 to make sure it is being called. The numbers are still unsorted as the InsertionSort does nothing yet.

The `Insertion Sort` algorithm sorts the data stored in the array `A` in precisely the same manner that many card players sort their cards. We start with the first card (card 0). A sequence of one card is always ordered, so far so good. Let's assume that we somehow have i cards ordered already with the smallest card in slot 0 and the biggest in slot $i-1$. We now take a new card and we compare it to the card in slot $i-1$. If the card in slot $i-1$ is bigger than the one we want to insert we move the card in slot $i-1$ into slot i and we compare our card to the one in slot $i-2$. If that card is also bigger than the one we want to insert we also move that card one place up to slot $i-1$. We proceed this way until we find a card that is smaller than the card we want to insert. Let's suppose that card is in slot j. We can then insert the present card immediately above that in slot $j+1$ since we know that this slot must be empty. `Insertion Sort` takes N^2 operations to complete. If you're confused about this, try it with a deck of cards. In pseudo-code the algorithm looks something like this:

Insertion Sort

① **for** $i \leftarrow 1$ **to** Length(A)-1

② key$\leftarrow A[i]$ *Take a new card that's not ordered yet.*

③ $j \leftarrow i-1$

④ **while** $j >= 0$ and $A[j] >$ key *Compare to the new card*

⑤ $A[j+1] \leftarrow A[j]$ *Move up*

⑥ $j \leftarrow j-1$

⑦ **end while**

⑧ $A[j+1] \leftarrow$ key *Insert the new card*

⑨ **end for**

1.3 Now implement the full `InsertionSort` function in your file `sort.cpp`. Once it compiles, test it by sorting the list of 10 real numbers with `testsort` and selecting 1. The Makefile you just made above should work to compile this with `make testsort`. Now the last output should be sorted from lowest to highest in the final line of prints.

We mentioned above that `Insertion Sort` is an algorithm requiring N^2 operations to complete. We now turn to a much faster (and much more ingenious) algorithm which depending on the input can take as much as N^2 operations (*worst case*) but much more likely finishes in $N \log_2 N$ operations. This algorithm is called `Quick Sort` and is due to C. A. R. Hoare, Computer Journal 5, 1 April 1962 pp 10-15, who is now *Sir* Tony Hoare. `Quick Sort` uses a divide and conquer method usually based on recursion. Here is the basic threestep process for sorting an array $A[0 \ldots n-1]$:

Divide: Partition (rearrange) the array $A[0 \ldots n-1]$ into two (possibly empty) subarrays $A(0 \ldots q-1)$ and $A(q+1 \ldots n-1)$ such that each element of $A[0 \ldots q-1]$ is less than or equal to $A(q)$, which is, in turn, less than or equal to each element of $A[q+1 \ldots n-1]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[0 \ldots q-1]$ and $A[q+1 \ldots n-1]$ by recursive calls to `Quick Sort`. To send a subarray, such as $A[q+1 \ldots n-1]$, to a function you can specify it as `&A[q+1]`. Note that the two subarrays are shorter than the original (by about half most of the time).

Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[0 \ldots n-1]$ is sorted.

Quicksort relies on a partition step. Pseudo-code for the `Partition` function is listed below. The element used for *partitioning* is simply determined as the rightmost element $A[n-1]$ in the subarray $A[0 \ldots n-1]$ to be sorted.

① `Partition`(A, n)

② $x \leftarrow A[n-1]$ *Choose the partitioning value*

③ $i \leftarrow 0$

④ **for** $j \leftarrow 0$ **to** $n-2$

⑤ **if** $A[j] \leq x$ **then**

⑥ $A[i] \leftrightarrow A[j]$ *Swap these two*

⑦ $i \leftarrow i+1$

⑧ **end if**

⑨ **end for**

⑩ $A[i] \leftrightarrow A[n-1]$ *Swap these two*

❶ **return** i

❷ **end** `Partition`

1.4 Consider the operation of swapping two values. Explain what is wrong with this:

```
// Swap A[i] and A[j];
A[i] = A[j];
A[j] = A[i];
```

1.5 Implement the Partition function of the `Quick Sort` algorithm first as a function in the files `sort.cpp` and `sort.h`. Parition takes an array and an integer as arguments but it also returns an integer. It is not void like InsertionSort. The partition chooses a value in the array and then rearranges the array into those elements smaller and larger than this value. As part of that process it counts how many values are smaller than the partitioning value and this is the information it returns. Call `Partition` from your `testsort.cpp` and have it print out the partition count just to check it works with option 2. There is already a line of code you can uncomment to do this in `testsort.cpp`.

Simple pseudo-code for a recursive version of `QuickSort` which uses the above `Partition` function is listed below. It relies on the partition to produce two sub-arrays where every element of the first one is guaranteed to all be lesser in value than every element of the second one. Once the process has progressed until the sub-arrays are 0 or 1 long then the sort is completed.

Quick Sort

① `QuickSort`(A, n)

② if $n > 1$ then

③ $q \leftarrow$`Partition`(A, n)

④ `QuickSort`$(A[0 \ldots q-1], q)$

⑤ `QuickSort`$(A[q+1 \ldots n-1], n-1-q)$

⑥ **end if**

⑦ **end** `QuickSort`

1.6 Implement the above `Quick Sort` algorithm as a function called `QuickSort` in files `sort.cpp` and `sort.h`. It will use your `Partition` function. Modify `testsort.cpp` to test the quick sort, make it and test that it sorts correctly by running `testsort` and choosing option 3.

Now that we have verified that the sorts work, we are ready to test the performance of the algorithms.

1.7 Comment out the loops containing `std::cout` to avoid too much text printing to the terminal and then set `n=10000`. Make `testsort` again. You can now time the program with different algorithms by using:

```
time testsort
```

to run an executable called `testsort`. In the output, the first number preceeding the `u` is roughly the number of cpu-seconds used. Time the algorithms and see which one performs best. Try timing more than once to verify your results. Report your results.

1.8 Repeat the timing exercise but this time with an array of 10,000 numbers that is already ordered both top-down and bottom-up, `A[i]=i` or `A[i]=n-i+1`. Report the results and explain the outcomes.