CPSC 1160 SECTION 001 INSTRUCTOR: GLADYS MONAGAN

ASSIGNMENT #4: SORTING AND COMPLEXITY

DUE DATE WITH BRIGHTSPACE: FEBRUARY 1 AT 11:50 PM

THE LEARNING OUTCOMES

- to follow an algorithm from a description and to implement it in C++ (a sorting algorithm)
- to do a theoretical analysis of some code and to express it in big O notation
- to do a theoretical analysis of code written by the student and to express it in big O notation
- to write a recurrence relation with initial value if the function being analyzed is recursive
- to analyze code for best, average, worst cases

READINGS

- class notes
- §3.9 and §4.4 on Generating Random Numbers
- §7.10 Sorting Arrays
- chapter 17 on Recursion
    - not covered §17.3, §17.6, §17.7
- chapter 18 on Developing Efficient Algorithms
    - §18.1 - §18.4
    - not covered: Towers of Hanoi analysis §18.4.3
    - not covered: §18.5, (§18.6, §18.7), §18.8, §18.9, §18.10
    - yes Chapter Summary
- chapter 19 on Sorting §19.1 - §19.5
    - not covered §19.6, §19.7, §19.8

THEORETICAL ANALYSIS ON PAPER

Do a theoretical analysis of the following C++ function `multiply`.

Count the (exact) number of **multiplications** done in `multiply` (and its helper functions) as a function of n.

In addition to giving your answer as a function of n, express the answer in big O notation.

Show your workings.

```
int I(int i, int j, int n)
{
   return n * i + j;
}

int dotProduct(const int A[], const int B[], int i, int j, int n)
{
   int t = 0;
   for (int k = 0; k < n; k++) {
      t += A[I(i,k,n)] * B[I(k,j,n)] * A[I(i,k,n)] * B[I(k,j,n)];
   }
   return t;
}

void multiply( const int A[], const int B[], int C[], int n )
{
   for (int i = 0; i < n; i++ ) {
      for (int j = 0; j < n; j++ ) {
         C[I(i,j,n)] = dotProduct(A, B, i, j, n );
      }
   }
}
```

DESCRIPTION OF A SORTING ALGORITHM

Suppose we want to sort n integers of an array A in **ascending** order using the algorithm described below.

For the **following explanation given below**, **assume that the number of integers is n > 5 and that there are no duplicates in the array**. A is an array with n integers.

- We place the largest of the array A of n integers into A[n-1] by comparing pairs of numbers:
  - we compare A[0] with A[1] and if A[0] > A[1] then A[0] and A[1] are swapped
  - we compare A[1] with A[2] and if A[1] > A[2], then A[1] and A[2] are swapped
  - we continue comparing in pairs
  - ..
  - we compare A[n-2] with A[n-1] and if A[n-2] > A[n-1], then A[n-2] and A[n-1] are swapped
- We place the smallest of the first *n-1* integers into A[0] by comparing pairs of numbers "in the opposite direction" from the previous pass:
  - we compare A[n-2] with A[n-3] and if A[n-2] < A[n-3], then A[n-2] and A[n-3] are swapped
  - we compare A[n-3] with A[n-4] and if A[n-3] < A[n-4], then A[n-3] and A[n-4] are swapped
  - we continue comparing in pairs
  - ..
  - we compare A[1] with A[0] and if A[1] < A[0], then A[0] and A[1] are swapped
- We place the largest of the integers from A[1] up to and including A[n-2] into A[n-2] by comparing pairs of numbers "in the opposite direction" from the previous pass
  - we compare A[1] with A[2] and if A[1] > A[2] then A[1] and A[2] are swapped
  - we compare A[2] with A[3] and if A[2] > A[3], then A[2] and A[3] are swapped
  - we continue comparing in pairs
  - ..
  - we compare A[n-3] with A[n-2] and if A[n-3] > A[n-2], then A[n-3] and A[n-2] are swapped
- We place the smallest of the integers from A[1] up to and including A[n-3] into A[1] by comparing pairs of numbers "in the opposite direction" from the previous pass:
  - we compare A[n-3] with A[n-4] and if A[n-3] < A[n-4], then A[n-2] and A[n-3] are swapped
  - we compare A[n-4] with A[n-5] and if A[n-4] < A[n-5], then A[n-4] and A[n-5] are swapped
  - we continue comparing in pairs
  - ..
  - we compare A[2] with A[1] and if A[2] < A[1], then A[2] and A[1] are swapped
- We continue comparing in pairs in both directions until the whole array is sorted in ascending order.

For your assignment and in general, do **not** assume that n > 5.

For your assignment and in general, do **not** assume that there are no duplicates. Duplicates are allowed in the array A.

I just made those two simplifications to explain the algorithm above.

IMPLEMENT

1. Implement the algorithm described in the previous section and use the function prototype

    ```
    void sort(int A[], int n);
    ```

    You may overload the function `sort` but you must still use the function prototype above to call the overloaded function.

    Do not keep track of the swaps. No need to keep a Boolean variable that would help you optimize whenever any integers were not swapped in a pass.

2. Now using the rand() library function, generate an array of 19 random integers in the range
    0 <= random number <= 999
    Print onto standard output the array before you `sort` it and after you `sort` it. Print 5 integers per line (use a named constant for 5) with the last line having at most 5 numbers. Format your output so that you can tell quickly whether the sorting is working.

3. For `sort`
    a. Give the time complexity in big O notation: no need to show your workings or how you came up with the answer.
    b. What is the worst case and the best case and the average case?
    c. Is the running time complexity different for the different cases? Why yes or why not?

4. Repeat 1. and 2. but this time write a **recursive function** that implements the sorting algorithm. Use the function prototype

    ```
    void sortR(int A[], int n);
    ```

    Yes, you may overload the function.
    Like in the class notes for selection sort and insertion sort, recursive functions *can* have loops

5. Given that n is the size of the array (i.e. n is the number of integers in the array), do a theoretical analysis for the running time of the function `sortR`.
    In your **theoretical** analysis, count the number of comparisons that `sortR` does to sort `n` values. Count comparisons between the array values the way we have done in class.
    Show your workings.
    In addition give the complexity in big O notation.
    Do not modify your code for the theoretical analysis. The assignment is different from the lab.

6. For `sortR`
    a. What is the worst case and the best case and the average case?
    b. Is the running time complexity different for the different cases? Why yes or why not?

SUBMIT WITH BRIGHTSPACE AS A SINGLE ZIP FILE: PART A) TO PART F) AS LISTED BELOW

A) The source code of the complete sorting programs: one program with an iterative version `sort` function and one program with a recursive version `sortR` function (you may put both versions in a single program if you want). Document clearly your functions.

B) Some "screen captures" or "print screens" that show that your iterative version `sort` works:
   1. Show the input array of 19 random integers (at most 5 integers per line)
   2. Show the sorted output array of 19 random integers (at most 5 integers per line)

C) Some "screen captures" or "print screens" that show that your recursive version `sortR` works:
   1. Show the input array of 19 random integers (at most 5 integers per line
   2. Show the sorted output array of 19 integers (at most 5 integers per line)

Submit as a pdf file (or as raster files from handwritten notes)

D) The theoretical analysis of the recursive version `sortR` as described in 5 and the answers to the questions 6a. 6b.

   - Give a recurrence relation with initial value.
   - Solve the recurrence relation with initial value and give your answer in closed-form as a function of n.
   - Write the resulting expression in big O notation
   - Show your work.

E) As per 3.a give the complexity of `sort` in big O notation (no need to show your work) and give the answer to the questions 3b. and 3c.

F) The theoretical analysis of multiply

<u>**Do not plagiarize.**</u>