EECE.3220: Data Structures

Spring 2017

Programming Assignment #2: Algorithmic Complexity

Due **Tuesday, 2/21/17***, 11:59:59 PM

***Assignments submitted by Wed., 2/15 earn a 10% bonus on initial grade only**

1. Introduction

This assignment will provide a different look at algorithmic complexity by asking you to measure the execution time of some algorithms discussed in class. You will write functions to implement the linear search, binary search, and selection sort algorithms, and use functions from the time library to test how long they take to run.

2. Deliverables

You should submit three files for this assignment:

- ***prog2_main.cpp:*** Source file containing your main function.
- ***prog2_functions.h:*** Header file containing prototypes of relevant functions.
- ***prog2_functions.cpp:*** Source file containing definitions of relevant functions.

Submit all three files by uploading them to your Dropbox folder. Ensure your file names match the names specified above. Do not place these files in a subdirectory—place them directly in your shared folder. Failure to meet this specification will reduce your grade, as described in the program grading guidelines.

3. Specifications

The main program (***prog2_main.cpp***) should do the following:

- Prompt for and read a seed value for the random number generator. (See Section 4 for hints on random number generation.)
- Prompt for and read three array sizes to be tested; repeat that prompt if the user enters any invalid array size.
- Successively call functions to do the following:
    - Allocate and fill the three arrays with random values. The array sizes should be based on the user input described above. (See Section 4 for hints on dynamic memory allocation in C++.)
    - Measure the time required to do each of the following operations (see Section 4 for hints on measuring time):
        - Perform a linear search on each array
        - Perform a selection sort on each array
        - Perform a binary search on each array
    - **Clarification: You want to measure worst-case execution time, so searches should look for a value that won't be in the array.**

3. Specifications (continued)

prog2_functions.h should contain prototypes for the following functions, which you are required to implement in ***prog2_functions.cpp*****:**

`int *create_array(int n);`

Dynamically allocate an integer array of size `n` and fill that array with random values. Return a pointer to the first element in the array.

`int linear_search(int arr[], int n, int req);`

Perform a linear search for the value `req` on array `arr[]`, which contains `n` integer values. Return the position of the requested value if found, and -1 if not found.

`int binary_search(int arr[], int n, int req);`

Perform a binary search for the value `req` on array `arr[]`, which contains `n` integer values. Return the position of the requested value if found, and -1 if not found.

`void selection_sort(int arr[], int n);`

Perform a selection sort on array `arr[]`, which contains `n` integer values. The array should be sorted from lowest to highest value.

4. Hints

Random number generation: The C standard library header (`<cstdlib>` or `<stdlib.h>`) includes the following functions for pseudo-random number generation:

- `rand()`: Returns a pseudo-random integer value between 0 and RAND_MAX, a constant defined in the standard library.
- `srand()`: Provide a seed value to the random number generator.
  - Providing the same seed value to `srand()` in multiple program runs will cause `rand()` to generate the same sequence of values.
  - One way to simulate true randomness is to use the current system time as the seed value, using the following function call: `srand(time(NULL));`

Dynamic memory allocation: As we will discuss in class, dynamic allocation in C++ is done using the `new` operator. Expressions using `new` evaluate as pointers to the given type. So, for example, the following line of code creates an array of 10 `double` values:

```
double *dlist = new double[10];
```

Dynamically allocated memory can be freed using the `delete` operator. `delete` should be followed by a pointer to a block to be deallocated. For example, to deallocate the array above, you would write the following:

```
delete dlist;
```

All arrays allocated in this program should be deleted before the program ends.

4. Hints (continued)

Measuring time: The C time library (<ctime> or <time.h>) contains functions related to elapsed time. One way to measure the amount of time consumed by a piece of code is the clock() function, which returns a value of type clock_t representing the number of clock cycles consumed by the program.

Taking the difference between two clock measurements and dividing by the predefined constant CLOCKS_PER_SEC converts that time difference to seconds. Scaling your division by another amount allows you to get different units of time. For example, the code snippet below prints the number of milliseconds it takes to call the function f():

```
clock_t t1, t2;
t1 = clock();       // First time measurement
f();
t2 = clock();       // Second time measurement
cout << "The function call took "
     << (t2 - t1) / (CLOCKS_PER_SEC / 1000)
     << " milliseconds" << endl;
```

For this program, execution times should be shown in microseconds (10^{-6} seconds).

5. Test Cases

Your output should closely match these test cases exactly for the given input values. Please note that execution times may differ significantly on different computers, but the relative execution time for a given search or sort function on different array sizes should be roughly the same across systems.

I will use these test cases in grading of your assignments, but will also generate additional cases that will not be publicly available. Note that these test cases may not cover all possible program outcomes. You should create your own tests to help debug your code and ensure proper operation for all possible inputs.

The test cases were generated in Xcode, so I've copied and pasted the output below, rather than showing a screenshot of the output window. User input is underlined, although it won't be when you run the program.

Case 1:

```
Enter seed for RNG (-1 for truly random seed): 1
Enter three array sizes: 25000 50000 75000
Creating arrays ...
Linear search test ...
Time to linear search 25000 element array: 72 uS
Time to linear search 50000 element array: 140 uS
Time to linear search 75000 element array: 212 uS

Selection sort test ...
Time to selection sort 25000 element array: 816356 uS
Time to selection sort 50000 element array: 3271099 uS
Time to selection sort 75000 element array: 7311509 uS

Binary search test ...
Time to binary search 25000 element array: 1 uS
Time to binary search 50000 element array: 1 uS
Time to binary search 75000 element array: 2 uS
```

Case 2:

```
Enter seed for RNG (-1 for truly random seed): -1
Enter three array sizes: 100000 20000 4000
Creating arrays ...
Linear search test ...
Time to linear search 100000 element array: 283 uS
Time to linear search 20000 element array: 57 uS
Time to linear search 4000 element array: 11 uS

Selection sort test ...
Time to selection sort 100000 element array: 13275580 uS
Time to selection sort 20000 element array: 536484 uS
Time to selection sort 4000 element array: 21678 uS

Binary search test ...
Time to binary search 100000 element array: 1 uS
Time to binary search 20000 element array: 1 uS
Time to binary search 4000 element array: 1 uS
```