

Name _____

Instructions Follow instructions *carefully*. This is an individual homework consisting of two parts; a set of analytical problem from the class text, Levitin, and a programming assignment. For the analytical part, you should solve all of them and submit them individually on or before the due date. Clarity and legibility of presentation of your submission are as important as your answers to problems. You are very strongly advised to typeset your solution using some document processing system. If the grader cannot easily read your writings, you may not be awarded full points even if you claim your answers are correct. Staple this cover page to the front of your assignment for easier grading. For the programming part, be sure to submit all your program files via the webhandin (the address is to be announced).

Analytical Problems

Problem	Page	Points	Score
2.2.3	60	10	
2.3.1c-g	67	5	
2.3.6a-d	68	10	
2.4.1	76	10	
2.4.4	76	10	
2.4.9a	77	5	
Program			
Correctness		25	
Style/Documentation		5	
Analysis		20	
Total		100	

Programming Assignment

You are to implement three brute-force style sorting algorithms and perform empirical tests and analysis on various data sizes. You must program in the C++/JAVA language using good style and documentation in a true objected oriented manner (NOTE: it will be very helpful for the TA if you did the programming in JAVA). Specifically, you will implement the following algorithms (as presented in Levitin) which will sort arrays of integers in *increasing* order: Selection Sort, Bubble Sort and Cocktail Sort. Cocktail Sort is essentially the same as Bubble Sort but instead of starting at the beginning of the array, it alternates between going from left-to-right to right-to-left on each pass of the array.

So that your functions can be tested for correctness, you *must* implement each sort such that functions use the following names:

- C++ - SelectionSort(int *sort_array, int size)
Java - SelectionSort(int [] sort_array, int size)
- C++ - BubbleSort(int *sort_array, int size)
Java - BubbleSort(int *sort_array, int size)
- C++ - CocktailSort(int *sort_array, int size)
Java - CocktailSort(int *sort_array, int size)

Implementations should be placed in source files with *exactly* the following names: SelectionSort.cpp(java), BubbleSort.cpp(java), and CocktailSort.cpp(java) respectively. There will not be a need to define classes or use header files for these. Note: you are passing an array pointer to each sort function, so you will need to send a COPY of each array to each sort function to ensure that the three different sorts are acting on the same array.

In addition, you will also create a file (name it `main.cpp(java)`) that will do some empirical tests on your algorithms. First, you will analyze each algorithm on randomly generated instances of various array sizes. You can choose your own test sample sizes if you wish but be sure to justify your choices in your analysis. A good suggestion would be $n = 1000, 5000, 10,000, 15,000, \dots 50,000$. See the `man` pages on `srand()` and `rand()` or the `Random` class in Java to populate your arrays with random integers. In addition to random data samples, you may want to see how the algorithms behave on other types of data sets—already sorted, completely unsorted, almost sorted, small integers, large variations etc.

To compare and contrast the three different algorithms you will need to keep track of how many comparisons, swaps, and CPU time (see `clock()` or `System.currentTimeMillis()`) it takes for each instance to be sorted by each algorithm, plus any other characteristics you can think of. You may want to run each instance of n a number of times and take an *average*, the choice is yours. However, you must detail and justify the data sets you use.

Since you are not defining any classes, you will have to keep track of the comparisons and swaps in *global variables*. To keep them standard, declare them as `int COMPS` and `int SWAPS` in your `main.cpp` file *before* you include your `cpp` files.

To report the final results, the main program should output a nice looking table. The following is merely a *suggestion*. As long as it is readable and conveys all the necessary information you may design your own table.

n	10	100	...	10000
Selection Sort	comps: 32.5	comps: xx	...	comps: xx
	swaps: 4.3	swaps: xx	...	swaps: xx
	time: xx	time: xx	...	time: xx
Bubble Sort	comps: 32.5	comps: xx	...	comps: xx
	swaps: 4.3	swaps: xx	...	swaps: xx
	time: xx	time: xx	...	time: xx
Cocktail Sort	comps: 32.5	comps: xx	...	comps: xx
	swaps: 4.3	swaps: xx	...	swaps: xx
	time: xx	time: xx	...	time: xx

Be sure to hand in all of your files individually (*not* in a zip or tar file), hand in your analysis as HARD COPY including a printout of your test results. Include a makefile that compiles your code into an executable called `sorttest` (via the command `make sorttest`) using the `g++` compiler.

Points will be awarded based on the following categories:

- **Correctness** – Your code should execute as described above, naming conventions should be followed. You must include a makefile that will compile your code into an executable.
- **Style/Documentation** – Your code should be readable, properly indented and spaced with sufficient comments to explain. Generally good Object Oriented style should be followed including separation of header/implementation files, appropriate naming of methods and variables, etc.
- **Analysis** – Your analysis should reflect your understanding and critical analysis of the algorithms as well as demonstrate that you gave some thought to the assignment beyond mindlessly coding the algorithms. Discuss how each algorithm works. For each of the criteria, (comparisons, swaps, CPU time) compare and contrast your empirical results with each algorithm’s asymptotic characterization, with each other, and with regard to increasing data set size. On what types of data sets would each algorithm perform well (already in order, out of order, etc)? Which criteria are more relevant and in what context? Are there any other criteria that could be explored? Did the data match the theoretical asymptotic behavior? How and why or why not? Where you able to approximate the multiplicative constant for each algorithm? Discuss any modifications that you tried to the algorithms and justify why you did so. Discuss anything else that you feel is relevant, pitfalls that you faced, etc. Turn in your analysis as a hard copy with the rest of your assignment.