CS 321 Programming Assignment 6 – Sorting

Spring 2022 – Chapter 9 – Due on Laulima April 19, 11:55 PM

Introduction

In this assignment, we are going to compare modifications to the quicksort and shellsort algorithms, and we are going to conduct some analysis on the relative performance of these modifications. There will be more than 20 points available on this assignment, though it only counts as 20 points towards the programming assignment portion of your final grade. Thus, you have the opportunity to earn some extra credit.

Part 1 – Quicksort variants

At the end of the textbook's section covering quicksort, two alternative techniques for choosing the bound (i.e. the pivot value) were mentioned – using a randomly chosen element from within the portion of the array currently being sorted, and using a median element of the first, middle, and last elements of the portion of the array currently being sorted. Implement both of these two versions of quicksort as modifications to the quicksort implementation provided on page 514, figure 9.11. Provide test code on short arrays of ~20 random integers to demonstrate the correct operation of your modified algorithms, and of the original algorithm. Run each of the three algorithms on randomly generated data arrays of sizes 5000, 10000, 50000, 100000, 150000, and 200000. For each size, you will perform 5 runs with each algorithm and will report the median (not average) execution time for each algorithm at each size. Generate a plot using Excel or some other graphing program for the three algorithms at the various data sizes. The x-axis will be the data size; the y-axis will be the median execution time. You should draw each plot line on the same figure and include a legend that shows which line corresponds to which algorithm. You may include more data sizes if you wish. Show your figure in a word file or pdf, and provide in that file a discussion of the relative runtime performance of each of the three quicksort variants. Which variation do you recommend? Why? Provide a justification of your answer.

Part 1 Point breakdown

(3 points) – Implementation & verification of the random pivot point quicksort variant.
(3 points) – Implementation & verification of the median of 1st, middle, & last values as the bound quicksort variant.

(4 points) – Code to perform analysis, word file containing plot and discussion of the relative performance of the three quicksort variants.

Part 2 – Shellsort variants

Figure 9.8 on page 508 contains an implementation of the Shell sort algorithm that calculates increments according to the relation $h_{i+1} = 3*h_i + 1$ and uses insertion sort to sort the subarrays containing every h^{th} element. We can implement modifications of shellsort by changing the increment relationship. For each of the increment schemes mentioned below, implement a version of shellsort using insertion sort as the internal sorting algorithm:

a) $h_1 = 1$, $h_{i+1} = 3h_i + 1$ and stop with h_t for which $h_{t+2} \geq n$ (this is the version you already have, though you need to modify it for the given stopping condition)
b) $h = 2^k-1$ for k = 1 up to the largest k for which $2^k-1 < n$
c) $h_1 = 1$, then $h = 2^k+1$ for k = 1 up to the largest k for which $2^k+1 < n$
d) Fibonacci numbers starting with F(2) = 1 up to the largest F(k) < n
e) n/2 is the increment for the first iteration, then $h_i = floor(.75h_{i+1})$ until you reach 1 (this sequence is generating numbers from high to low - in the reverse order of the previous sequences. Regardless of how the numbers are generated by these sequences, remember that shellsort uses the numbers from high to low order, with the last increment always being 1. See below).

Be careful; some of these sequences count up from 1, some generate numbers in a descending sequence. Either way, once you have generated the increments, you will use them in a descending sequence**. In all cases, the final iteration increment will be 1.**

You have a total of 5 variants of shellsort, 4 of which you can implement and one of which is given for you but needs the stopping criterion added (on page 508). For each variant, provide test verification that the shellsort algorithm works on ~30 elements.

After you have competed and verified your implementations, you will perform the same analysis you provided for the quicksort algorithms, including timing, plots, discussion, etc.

Part 2 Breakdown

(2 points each) – Implementation and testing of each variant, up to 10 points.
(4 points) - Code to perform analysis, word file containing plot and discussion of the relative performance of the quicksort variants.

More extra credit

For an additional 2 points of extra credit, plot the performance of all three variants of quicksort with the performance of all variants of shell sort using insertion sort that you implemented on a single common plot, and discuss the relative performance of each. Comment on the observed difference in performance between quicksort and shellsort.

Bonus Question (3 points)

Answering this question is optional. Selection sort is another $O(n^2)$ sorting algorithm that we previously studied. Using shell sort with insertion sort as its internal sorting algorithm provides faster sorting performance than using insertion sort alone. This is not true for selection sort – if we try to use selection sort as the internal sorting algorithm for shell sort, we will not see improved performance over using selection sort alone – in fact, the shell sort will run slower than running selection sort alone. Why is this the case? Why does shell sort work with insertion sort but not with selection sort? Discuss this at the end of your analysis report.

A note on mandatory sections for shellsort

You must include the plot and discussion of relative performance of the shellsort variants. No credit for implementing extra variants will be given if the analysis is not present. No credit for any extra variant will be given if that variant is not part of the plotted and discussed data. Simply providing an implementation without including it in your analysis is insufficient.

Documentation

Include enough documentation with each sort function so that I can tell from the function header comments which variant each function represents. Failure to include this documentation will cost you at least half a point from the implementation score for each function lacking documentation.

Hints

- You need a random number generator scheme capable of generating random integers. Google will help you figure out how to do this. I suggest that you generate random numbers in the range [0, 500000].
- You must generate a new random array between each call to a sorting algorithm. If you try to pass the same array to two different sorting functions, the second function will receive sorted data. Remember that arrays are implicitly passed by reference, not by value.
- You will also need to figure out how to call the system clock to time the execution time of the sorting algorithm. You need to query it twice – immediately before and immediately after calling a sort routine. You can then

calculate the difference between the end and start times to calculate the run time of your sort function. Most likely, the returned values will be in milliseconds. Again, Google can help you out here.

- Store the runtime results for your runs in a spreadsheet. Give yourself enough time to perform your analysis and create your plots. Your analysis document will be the main deliverable that I look at when grading your assignment.

Submission

When you are finished, submit your code and a word file or pdf containing your analysis & plots on laulima. Your analysis file must be either a word file (.docx) or a pdf. It is also helpful to turn in a spreadsheet with your experiment results – please use Excel format (.xlsx). No other file formats will be accepted and will be ignored. The due date is on laulima.