Lab Goals

- Evaluate the effectiveness of the sorting algorithms that we have discussed under various experimental parameters.
- Review the details of the sorting algorithms that we have discussed to date through some sample traces.

Assignment Details

Part One: Selection Sort vs. Insertion Sort vs. Shell Sort (40 points)

At the end of class on Wednesday, we used the SortCompare program to determine whether Selection Sort or Insertion Sort was the faster algorithm. It turned out that Selection Sort appeared to be faster with large arrays of random values (when we used N=10000), but when we dropped the array size down to N=100, we saw that the faster algorithm switched between Selection and Insertion depending on the randomly generated values and their starting order.

In this lab, we will perform a few more experiments, this time factoring in Shell Sort, to determine under which conditions each algorithm runs the fastest. In addition, we should note some special improvements in the Shell Sort case.

Before you begin these experiments, you will need some sample data reflecting the experiment that we ran at the end of Wednesday’s class. Please do the following:

1. Run the SortCompare function on Selection Sort and Insertion Sort, with an input size of 10000 and with 10 iterations per run. Perform 5 different runs of this step, and note the ratio in performance between Selection Sort and Insertion Sort. What was the average ratio, and what is the standard deviation?

2. Next, run SortCompare again with an input size of 100, changing no other parameters. Perform 5 different runs of this step, and note the ratio in performance between Selection Sort and Insertion Sort. What was the average ratio, and what is the standard deviation?

**Experiment 1: Selection vs. Insertion Sort by Datatype (10 points):** If you examine the SortCompare code provided, you will see that the timeRandomInput() function creates an array of Double type to analyze the effectiveness of the sorting algorithms. It would be interesting to see if the results that we noted before will change with a different datatype.
1. In the `timeRandomInput()` function, replace the array `Double[] a` array with a new array `Integer[] a`. You will also need to cast the `double` values returned from the call to the function `StdRandom.uniform()` to `int` type later in `timeRandomInput()`. The default call to `StdRandom.uniform()` returns values from $[0,1)$, and after integer division sorting a bunch of 0s will be boring, so let’s also add 50 parameter to the `uniform()` function call for a new upper bound. Save and compile.

2. Run `SortCompare` on Selection Sort vs. Insertion Sort again, using an array of size 10000 and 10 iterations. In our experiment during Wednesday’s class, we saw that Selection Sort was running consistently faster than Insertion Sort. Is this still the case now that we are sorting integers? Why/why not?

3. Perform 5 different runs of the above step, and note the ratio in performance between Selection Sort and Insertion Sort. What was the average ratio, and what is the standard deviation? How big is the difference when compared to the results that you saw with doubles?

4. Now, run `SortCompare` on Selection Sort vs. Insertion Sort again, using an array of size 100 and 10 iterations. In our experiment during Wednesday’s class, we saw that Selection Sort would occasionally run faster and Insertion Sort would occasionally run faster. Is this still the case now that we are sorting integers? Why/why not?

5. Perform 5 different runs of the above step, and note the ratio in performance between Selection Sort and Insertion Sort. What was the average ratio, and what is the standard deviation? How big is the difference when compared to the results that you saw with doubles?

6. Return the code back to its original state in sorting doubles for the next experiment.

**Experiment 2: Selection vs. Insertion Sort by Distribution (15 points):** The `StdRandom` library provided by the book lets us use different probability distributions for the random numbers that are generated. In the original code, we use the uniform distribution, in which there is an equal probability for each individual number to be generated. Let’s see what happens when we change that distribution.

1. Rather than a uniform distribution, let’s switch to a Gaussian distribution. The Gaussian distribution takes two parameters: a mean and a standard deviation for the distribution. Let’s set the mean of the distribution to 10 and the standard deviation to 0.5. Therefore, we want to replace `StdRandom.uniform()` with `StdRandom.gaussian(10, 0.5).

2. Run `SortCompare` on Selection Sort vs. Insertion Sort again, using an array of size 10000 and 10 iterations. Perform 5 different runs of the above step, and note the ratio in performance between Selection Sort and Insertion Sort. In our experiment during Wednesday’s class, we saw that Selection Sort was running consistently faster than Insertion Sort under the uniform distribution. Is this still the case now that we are using a Gaussian distribution? Why/why not?

3. Let’s try to reverse the parameters, so that our mean of the distribution is now 0.5 and the standard deviation is now 10. This effectively spreads out the data further, so that instead of having a tight cluster centered around 10, we have a broad dataset centered around 0.5.
4. Run \texttt{SortCompare} on Selection Sort vs. Insertion Sort again, using an array of size 10000 and 10 iterations. Perform 5 different runs of the above step, and note the ratio in performance between Selection Sort and Insertion Sort. Did anything change between the tight distribution and the broad distribution of Gaussian values? Why/why not?

5. Let’s try the exponential distribution next. With the exponential distribution, we will see a high probability of small values generated and a lower probability of larger values generated. The difference in these probabilities is controlled by a lambda parameter – a small lambda will give a distribution closer to uniform, while a large lambda will have many more values in the [0..1] range than in the [9..10] range.

6. We’ll start with a lambda value of 0.1, keeping us somewhat close to a uniform distribution. Update the \texttt{SortCompare} code to call \texttt{a[i] = StdRandom.exp(0.1)}. Run \texttt{SortCompare} on Selection Sort vs. Insertion Sort again, using an array of size 10000 and 10 iterations. Perform 5 different runs, and note the ratio in performance between Selection Sort and Insertion Sort. What was the average ratio, and what is the standard deviation? Did you notice much of a difference from the uniform distribution experiment from Wednesday’s class? Why/why not?

7. Now let’s update the lambda value to 1.5, which will give us a large number of very small values and a lower number of large values. Update the \texttt{SortCompare} code again and run on Selection Sort vs. Insertion Sort again, using an array of size 10000 and 10 iterations. Perform 5 different runs, and note the ratio in performance between Selection Sort and Insertion Sort. What was the average ratio, and what is the standard deviation? Did you notice much of a difference from the 0.1 lambda trial in the previous step? Why/why not?

8. Return the code back to its original state in sorting under the uniform distribution for the next experiment.

\textbf{Experiment 3: Shell vs. Insertion Sort by Array Length (15 points)}: Now, let’s bring Shell Sort into the mix. For this experiment, we will go back to sorting Doubles under the uniform distribution, and will just change the length of the input array that we are sorting. In all cases for this experiment, you should run Shell as the first command line argument and Insertion as the second – we are interested in seeing if Shell Sort improves its performance as we increase the input array size.

1. Run a \texttt{SortCompare} comparison between Shell Sort and Insertion Sort on 100 random doubles, using 10 iterations in each run. Perform 5 different runs, and note the ratio in performance between the two. What is the average ratio, and what is the standard deviation? Are you seeing consistent results, or are they changing wildly? Why/why not?

2. Now, let’s bump it up to running Shell Sort vs. Insertion Sort on 1000 random doubles, again with 10 iterations in each run. Perform 5 different runs, and note the ratio in performance between the two. What is the average ratio, and what is the standard deviation? Is there a noticeable performance difference between the 100 and 1000 array sizes? Are you seeing consistent results, or are they changing wildly? Why/why not?
3. Next up is 10000 random doubles. Perform 5 different runs, and note the ratio in performance between the two. What is the average ratio, and what is the standard deviation? Is there a noticeable performance difference between the 1000 and 10000 array sizes? Why/why not?

4. Let’s do one more experiment with 100000 random doubles (you may want to save some time for this one, or jump down to the Downtime questions in Part Two). Perform 5 different runs, and note the ratio in performance between the two. What is the average ratio, and what is the standard deviation? Is there a noticeable performance difference between the 10000 and 100000 array sizes? Why/why not?

5. Let’s graph the changes in performance between Shell Sort and Insertion Sort. The x-axis of your graph should show the array length, and the y-axis should show the ratio of runtimes that you get from the output of SortCompare. As with the previous lab, create a linear axis graph and a log-log axis graph. Note that for the linear axis graph, the x-axis should show different distances between each array length (i.e., the distance in the graph between 10000 and 100000 should be different than the distance between 100 and 1000). Do you see anything interesting in these graphs, or do they look like you would expect?

Part Two: While You Have Some Downtime (10 points)

While you wait for the above program executions to complete, you should have a bit of downtime to answer some additional questions:

1. How much physical memory was taken up by the contents of the 1Kints.txt file when loaded into a one-dimensional array? How about the 1Mints.txt file? How much physical memory would be taken up if the files contained doubles rather than ints? How much physical memory would be taken up if each of the file inputs were arranged in a two-dimensional array with 100 rows (1000 ints in a 100x10 array, 1000000 ints in a 100x10000 array)? (Note: I am looking for exact numbers of bytes here, not an approximation based off of tilde notation or Big-O notation.)

2. Trace how:

   (a) The Selection Sort algorithm will sort the array SELECTIONSORT.
   (b) The Insertion Sort algorithm will sort the array DEPENDSONINPUT. (Note that you do not need to show every exchange – merely showing the final position of each letter after each iteration of the outer for loop as in the visual from the slides is sufficient.)
   (c) The Shell Sort algorithm will sort the array SHELLSORTISSOMUCHFUNYOGUYS. (Same disclaimer as Insertion Sort.)

3. Which sorting algorithm (Selection or Insertion) will run the fastest on an array where all keys are equal? Why?

4. Which sorting algorithm (Selection or Insertion) will run the fastest on an array that is in reverse order? Why?

5. What is the best case input for Shell Sort? Why?
Submission Details

For this lab, please submit the following, either to BitBucket, on paper, or both as specified:

1. (Upload) Any supporting information from your experimental runs. This can be saved output files, the different versions of the SortCompare code, Excel or LibreOffice Calc spreadsheets, or anything else you want to submit. This step is optional.

2. (Upload and Print) A well-written response to all of the questions stated through the three experiments above. Include tables showing each of the ratios that you find throughout the experiments, and the graphs that you created in experiment 3 question 5.

3. (Upload and Print) Your responses to the questions in Part Two.

4. (Upload and Print) An Assignment Information Sheet filled out for your source files, if you choose to upload any.

Please remember that all files that you submit should be your own work, though you are welcome to discuss high-level topics and algorithms with classmates.