Randy Dang
Project README

Part 1: Steps for building the code:

- Create a main.cpp file, write boilerplate code, and print out "Hello World!"
- Create a read_input.h and read_input.cpp file and write a function that reads in multiset data from a file and stores it in a C++ vector (a list).
- Test the read input function by printing out stored elements, visually checking to ensure they match the elements in the file.
- Create selection.h, selection.cpp, subarrays.h, and subarrays.cpp. Begin designing the select function and write functions that split the input into subarrays of a specified size.
- Test subarray functions by printing out elements of subarrays in order, visually checking that they match inputted elements.
- Create partition.h and partition.cpp and implement partition.
- Create sort.h and sort.cpp and implement selection sort and quick sort. Test on randomly generated vectors of various sizes.
- Code the selection algorithm with recursive faith, using brute force methods on small input and calling functions from other files.
- Test selection as described in "Tests ran:"

Part 2: How to run the code:

- To compile the code, type "make selection" on the command line.
- To run the code, type "./selection inputFile" on the command line, where inputFile is the name of a file containing the elements to select from.
  - Each element in inputFile should be separated by some amount of whitespace, and there should be at least two elements. Elements can be integers or decimal numbers.

  - The code will prompt for a subarray size. Select a subarray size between 2 and the number of elements in the input. 5 is a common choice.
  - The code will prompt for the rank of the element to be selected. Input a rank between 1 and the number of elements. For example, to find the median, input approximately number of elements divided by 2.
  - The code will output the selected element and the time it took to run selection.

- To generate random integers between 0 and 100, type “make generate_inputs” followed by “./generate_inputs outputFile” where outputFile is the name of the file to output random numbers to.

An example of finding the median of a random 51 numbers between 0 and 100:

```
make generate_inputs
./generate_inputs input_ints.txt
51
make selection
./selection input_ints.txt
5
26
```

Part 3: Tests ran:

- Tests ran on pieces of the code as described in Step 1.
- Select 1st, 4th, and 7th element of a custom 7-element input file of decimal numbers, using subarray sizes 2, 5, and 7, and sort elements by hand to verify answers.
- Select 20th element of 55 random elements using subarray sizes 3, 5, 11, and 55, and verify answers by copying elements into a column of a spreadsheet and sorting that column.
- Generate 100000 random numbers and use a random number generator to select a subarray size and a rank. Verify answers by copying into a spreadsheet and sorting the spreadsheet. Repeat many times.

Part 4: Conclusions Drawn:

I collected runtime data for selection on various input sizes (using randomly generated integers between 0 and 100) and subarray sizes. For each input-subarray size combination in the table below, I completed three trials and averaged the results. For each trial, I used a random number generator to determine the rank of the element being selected. The full data set of all trials can be found here: `https://docs.google.com/spreadsheets/d/1MKH0aAngHKb6tHjgiC-VmRXPmiV_wkyj/edit?usp=sharing&ouid=107599951240719805947&rtpof=true&sd=true`

Runtime of Selection Algorithm for Various Input Sizes n and Subarray Sizes k in Milliseconds

	n=100	n=1000	n=10000	n=100000
k=3	0.148	2.784	34.156	336.471
k=5	0.123	1.795	20.447	219.192
k=15	0.110	1.232	11.994	122.748
k=$\sqrt{n}$	0.151	3.024	33.551	390.658
k=$n/2$	0.222	5.610	93.770	7311.300
k=n	0.070	2.998	280.73	27503.800

Observations:

- As expected, the runtime for $k = 3$ was significantly longer than for $k = 5$ or $k = 15$ because the selection algorithm runs in $\Theta(n \log n)$ time for $k = 3$ but $\Theta(n)$ time for k equalling some other constant.
- As k became a value that depended on n, the selection runtime got considerably longer, likely because many previously constant operations became dependent on n.
- The "best" value to choose for k would be a constant that is large but does not approach the value of n. In almost all cases, a subarray size of 15 was the most efficient.
- The above trends are much more apparent in the data for large input sizes than for small input sizes.

Part 5: Learning Outcomes:

- Several pieces of the algorithm (such as partition) could have been done in place but I decided not to for ease of implementation. I learned how difficult it can be to figure out an in-place algorithm that works for some operations.
- I learned how to use C++'s rand() function to generate random numbers to use for testing.
- I learned about C++'s chrono library and how to use it to store and print runtimes of operations.
- I learned the importance of recursive faith when implementing a recursive algorithm as complex as selection (especially when it came to finding the median of medians).
- I learned the importance of using brute force methods as base cases for small input sizes (for selection and sorting), and how quickly such methods become unideal as input sizes increase.
- As described in the "Conclusions" section, I learned that a relatively large but clearly bounded constant is the ideal subarray size for selection.