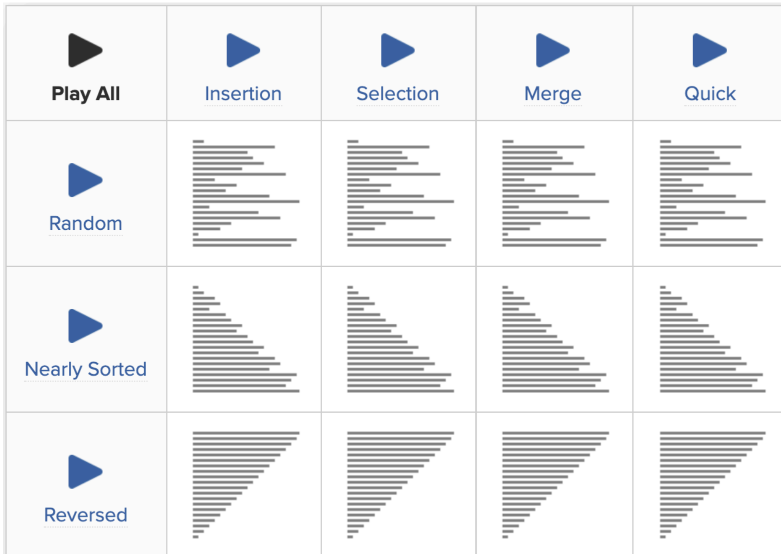
Project 6: `Sorting`



The sorting algorithms:

For this project you will **modify and analyze** 4 sorting algorithms:
SelectionSort (https://en.wikipedia.org/wiki/Selection_sort **)**
InsertionSort (https://en.wikipedia.org/wiki/Insertion_sort)
MergeSort (https://en.wikipedia.org/wiki/Merge_sort **)**
QuickSort (https://en.wikipedia.org/wiki/Quicksort**)**

You will find the starter files for this project on GitHub classroom. You can find the link to the GitHub classroom on Blackboard under Course Materials/Project6.

If you need a reminder, refer to the instructions and videos for git and GitHub Classroom in Project5.

This project is not so much about coding, rather about observation. Understanding the code of the sort functions and testing will be the learning experience here.

The starter class:

On GitHub classroom you will find an ***incomplete*** class: `SortingComparison`
The idea here is that you can instantiate a `SortingComparison` object to run different sorting algorithms on the same array and report on the number of comparisons performed by each algorithm to sort the function, allowing you to inspect the performance of each sorting function.

The data members of this class are an array called `values_` and its size `SIZE`. The parameterized constructor takes a size parameter used to initialize `SIZE` and allocate the array `values_` with that size. Because `SIZE` is `const`, it must be initialized within an initializer list.

The class has 4 sorting functions

- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Implementation:

You will complete the implementation of `SortingComparison` as directed in Parts 1 and 2 below. You are welcome to implement any additional methods you see fit to help you fulfill the requirements specified below. Your design should always be modular. If you find yourself implementing a function that is not cohesive (your function is doing more than one task, or it is very long) break it down into multiple functions.

Part 1:

You must **modify each sorting function** to keep track of and **return the number of comparisons** made to sort the input array.

- Count each time a sorting functions compares an element of the array to either another element (e.g. `a[i] < a[j]`), or to a number (e.g. `a[i] > next`)

- Do not count data insertion (e.g. a[i] = next) or swaps. The only **exception** to this is **in merge**, here **data insertion into the "merged" array actually counts as a comparison** because merging is how MergeSort sorts the array, so for merge we are counting each insert into temp_array as a comparison (e.g. `temp_array[i] = a[j]`)
- Note that for sorting algorithms with helper functions the work is likely done in the helper functions, so you need to modify these to communicate the number of comparisons back to the sorting functions. Since these already have a return, you will need to communicate the comparisons back to the sorting function through a reference parameter.

Part 2:

Implement the following function:

```
/**
@post  Sorts a copy of values_ in ascending order with each available sorting functions
       and prints the number of comparisons made by each sorting algorithm
@param array_type data_distribution of values_ in {RANDOM, INCREASING, DECREASING}
*/
void runComparison(data_distribution array_type);
```

This function will do the following:

- Populate `values_` with either random, strictly-increasing or strictly-decreasing integers, according to the value of `array_type`
- Make **a copy** of `values_` and sort it with SelectionSort
- Make **a copy** of `values_` and sort it with InsertionSort
- Make **a copy** of `values_` and sort it with MergeSort
- Make **a copy** of `values_` and sort it with QuickSort
- Print the number of comparisons made by each algorithm as follows (<u>you must include the extra \n</u>)

```
Selection sort comparisons: 49995000

Insertion sort comparisons: 25058763

Merge sort comparisons: 133616

Quick sort comparisons: 114342
```

Review:

Enums:

An *enumeration (enum)* is a user-defined type whose value is restricted to a user-defined range.

```
enum data_distribution {RANDOM, INCREASING, DECREASING};
```

Defines a new data type called `data_distribution`, variables can be instantiated with type `data_distribution` and their value can be one of RANDOM, INCREASING or DECREASING. These are generally used to make code more readable. Under the hood these are simply integers, but it would be obscure what the meaning of, say, 0, 1 and 2 would be in this case. The `runComparison` method takes a parameter of type `data_distribution`, which will determine the way this method will populate the array `values_`
These data distributions were chosen to illustrate the different behavior of the sorting algorithms on data that is organized in these ways.

Generating random numbers:

You can do this in many ways. Here is a suggestion:
First generated a seed using a timestamp, otherwise you will always generate the same "random" sequence (not so random after all).

```
srand(static_cast<unsigned>(time(0)));
```

Now you can use the rand() function to generate a random number. You should use the size of the array to set the range in which the random numbers will be generated.

```
some_random_variable = rand() % SIZE;
```

NOTE: all we can really do is generate pseudo-random numbers. In particular, rand() generates the same sequence given the same seed. While rand() is ok for this project, it may not be suitable for other applications that require more sophisticated/realistic random simulations. In future projects where you want to use random numbers or distributions you may want to explore the **<random>** library.

Testing:

This is the part of the assignment where you should inspect the differences between the sorting algorithms. I suggest yo do the following:

- Instantiate a `SortingComparison` object with size 10
- Call `runComparison` with each `data_distribution`
- Observe the difference in number of comparisons across the algorithms with respect to the `data_distribution` and make sure your numbers make sense with respect to what you know about the worst case time complexity of these algorithms.
- Do this again with size 100. How is it different from size 10?
- Do this again with size 1000
- Do this again with size 10000
- Do this again with size 100000 … what is happening??? What if you remove SelectionSort and InsertionSort from `runComparison`?

Submission:

Your project must be submitted to Gradescope <u>through GitHub</u>. The due date is Friday November 8 by 5pm. No late submissions will be accepted.

Have Fun!!!!!