Homework 5

- Due: Tuesday, October 23, 11:59pm
- Directory name in your github repository for this homework (case sensitive): `hw5`.
- Provide a `README.md` file.

Problem 1 (Class Structures and Inheritance, 20%)

Imagine that you are writing (yet another) car racing game. In it, the player can customize her race car in a number of ways:

- There are (currently - future versions might add more) three different type of chassis available: the Truck, the Racer, and the Buggy. They drive differently.
- The player can customize the color of her car.
- There are (again: currently) three different engines available: the Nitrox, the Warp, and the PowerBlast. They have different properties.
- The player can accumulate arbitrarily many "Items". These currently are
  1. Car Stereo
  2. Spoiler
  3. Bike Rack
  4. Flower Vase
  5. The following Weapons: Harpoon, Machine Gun, Grenade Launcher, Oil Dispenser
- While a player can accumulate arbitrarily many weapons, at most one can be in use at a given time.

The players also have various characteristics, like strength, experience, and control. The game supports arbitrary numbers of players playing at any given time.

Diagram the classes involved in the game so far. (We
ignore such things as racetracks and other parts of th
game.) Indicate which classes are abstract, and whic
aren't. Show which classes inherit from each other, publicly or privately. Also show which classes have a "has-a" relationship, possibly using sets or maps or lists where appropriate.

You can choose to draw this by hand and scan it (and submit as a JPG or PDF), or to use some graphics software (and produce a JPG or PDF), or to use UML (if you know it), or draw it using ASCII art (this may be a lot of work). Provide an explanation with your choices, i.e., tell us why you chose to have certain classes inherit from each other (or not inherit). (Notice that you don't need to do any actual programming for this problem.)

Of course, there isn't just one solution, but some solutions are better than others.

Make sure to indicate what file your answer is in, in your README.

Problem 2 (Break QuickSort, 10%)

Consider the following code for QuickSort's partition algorithm:

```
int partition(int *a, int L, int R) {
  int i=L;
  int index = medianof3(a,L,R);
  a.swap(index,R);
  int p = a[R];
  for (int j = L; j < R; j++) {
    if (a[j] <= p) {
      a.swap(i,j);
      i++;
    }
  }
  a.swap(i,R);
  return i;
}
```

The median-of-3 algorithm finds the middle value out of `a[L]`, `a[(L+R)/2]` (rounded down), and `a[R]`.

Provide an array of `12` different values that achieves
worst-possible pivot selection over all calls to partitio
explain your answer.

Problem 3 (Cave Exploration, 15%)

When people explore caves, they frequently use ropes to be able to find their way out. You attach the rope at the entrance, and keep a large spool with you. When you go deeper into the cave, you unroll some rope. As you retrace your steps, you roll up the rope again, so you can unroll it again later when you explore a different passage. You always keep the rope pretty tight, both as you go deeper and as you return.

In this problem, you will be given a sequence of steps that a cave explorer took. You assume that the rope was all rolled up at the beginning, and you are supposed to output whether the rope is guaranteed to be again all rolled up at the end of the sequence of steps. The sequence of steps is denoted by a string of lowercase letters `e`, `w`, `n`, `s` (the 4 directions). If the input contains any other characters (even spaces), you should output an error. Let's illustrate what we mean by this whole "rope rolled up" thing with some examples.

- If the sequence is "news", then the person took a step North, then East, then West (rolling up the rope a bit), and then South, finishing the rolling up. (Try it out if you want.) So the rope is all rolled up.
- If the sequence is "nesw", the the person took a step North, then East, then South, then West. So they went in a circle. If there was a rock in the middle of the circle, then the rope now loops around the rock, so it can't be all rolled up.
- If the sequence is "new", then the rope is half rolled-up, but because the person ends up in a different place than they started, there's still some rope unrolled.
- If the sequence is "newwes", then the rope is all rolled up. The (third) West step undid the rope from the (second) East step, and the (fifth) East step undid the rope from the (fourth) West step. Finally, the South step undid the rope from the North step. Again, try it out in person if you want to get a feel for how this works.

It should query the user for an input and read a singl from `cin`. Then, it should output whether (1) the rop rolled up by outputting `rolled up` to terminal, (2) the rope is not fully rolled up (output `not rolled up`), or (3) the input was invalid (output `invalid`, when there are contained characters other than the four we listed).

You may use anything from the STL you like to help solve this problem. You can also import any data structures and ADTs you created in a previous assignment.

If you think about this problem carefully, you can solve this very elegantly in about 15 lines of code total. If not, you may end up with 100+ lines of messy code that still don't get it right in all cases. So think before you start coding.

Your algorithm should run in linear time, and should be contained in `cave.cpp`, and the command `make cave` should produce the executable `cave`.

Problem 4 (k-way templated MergeSort, 40%)

Background: Functors and Comparators (0%)

The following is background info that will help you understand how to do the next step.

If you saw the following:

```
int x = f();
```

you'd think `f` was a function. But with the magic of operator overloading, we can make `f` an object and `f()` a member function call to `operator()` of the instance, `f` as shown in the following code:

```
struct RandObjGen {
  int operator() { return rand(); }
};

RandObjGen f;
int r = f(); // translates to f.operator() which return
```

An object that overloads the `operator()` is called a **functor** and they are widely used in C++ STL to provide a kind of polymorphism.

We will use functors to make a k-way Merge Sort alg
be able to use different sorting criteria (e.g., if we are
strings, we could sort either lexicographically/alphabetically or by length of string). To do so, we supply a functor object that implements the different comparison approaches.

An object/class whose job it is to compare two elements of some other class is called a **Comparator**. Thus, a comparator is a special type of functor in charge of comparing elements of some other type. If you give a sorting algorithm an array/vector of some type `T` as well as a comparator for objects of type `T`, then the algorithm will sort the array/vector by exactly the criterion that the comparator specifies. Here are two examples of string comparators; they are provided to you in `homework-resources` in files `functor.h` and `functor.cpp`.

```
struct AlphaStrComp {
  bool operator() (const std::string& lhs, const std::st
  { // Uses string's built in operator<
    // to do lexicographic (alphabetical) comparison
    return (lhs < rhs);
  }
};

struct LengthStrComp {
  bool operator()(const std::string& lhs, const std::str
  { // Compares strings by size;
    // Breaks ties by lexicographic comparison
    return (   (lhs.size() < rhs.size())
            || (lhs.size() == rhs.size() && lhs < rhs))
  }
};

string s1 = "Blue";
string s2 = "Red";
AlphaStrComp comp1;
LengthStrComp comp2;

// Here comp1(s1,s2) is calling comp1.operator() (s1, s
cout << "Blue compared to Red using AlphaStrComp yields
```

```
  // Here comp2(s1,s2) is calling comp2.operato
  cout << "Blue compared to Red using LenStrCom
```

This would yield the output

```
1  // Because "Blue" is alphabetically less than "Red"
0  // Because the length of "Blue" is 4 which is NOT less
```

We can now make a templated function (not class, just a templated function) that lets the user pass in which kind of comparator object they would like:

```
template <class Comparator>
void DoStringCompare(const std::string& s1, const std::str
{
  cout << comp(s1, s2) << endl;  // calls comp.operator()
}


  string s1 = "Blue";
  string s2 = "Red";
  AlphaStrComp comp1;
  LengthStrComp comp2;

  // Uses alphabetic comparison
  DoStringCompare(s1,s2,comp1);
  // Use string length comparison
  DoStringCompare(s1,s2,comp2);
```

In this way, you could define a new type of comparison in the future, make a functor struct for it, and pass it in to the `DoStringCompare` function and the `DoStringCompare` function never needs to change.

These comparator objects are used by the C++ STL `map` and `set` class to compare keys to ensure no duplicates are entered. They can also be passed to functions that sort objects, which is what you will use them for in this problem.

```
template < class T,                        // set::key_ty
           class Compare = less<T>,        // set::key_co
           class Alloc = allocator<T>      // set::alloca
           > class set;
```

You could pass your own type of Comparator object
class, but it defaults to C++'s standard less-than func
`less<T>` which is simply defined as:

```
template <class T>
struct less
{
  bool operator() (const T& x, const T& y) const {return ›
};
```

For more reading on functors, search the web or try this link.

Problem 4(a) Write another comparator (5%)

Implement another comparator `NumStrComp` on strings that does the following: for each digit and letter, it assigns a number ("0" = 0, "1" = 1, "2" = 2, ..., "9" = 9, "A" = 10, "B" = 11, "C" = 12, ..., "Z" = 35; same numbers for lowercase letters as for uppercase letters); all other characters are 0. It then simply adds up all the numbers assigned to all digits and letters in the string, and sorts by the sum. Ties are broken lexicographically, i.e., with the "standard" `string` comparison for C++.

Ad the functor `NumStrComp` to `functor.cpp` and `functor.h`.

Use the `DoStringCompare` function we gave you above to thoroughly test your new functor. You should not submit `DoStringCompare`, though - it's just for testing.

Problem 4(b) (Implement k-way templated Merge Sort, 25%)

Write your own templated implementation of the Merge Sort algorithm that works with any class `T`, and any Comparator functor for `T`. Put your implementation in a file `msort.h`; don't make a `msort.cpp` file (because you're using templates). Your `mergeSort()` function must take a `vector<T>`, an integer `k`, and a comparator object (i.e., functor) that has an `operator()` defined for it. In other words, it **must** have the following signature.

```
  template <class T, class Comparator>
  void mergeSort (std::vector<T>& myArray, int k, Compara1
```

Passing in different Comparator objects allows you to change the sorting criterion. The integer k is used as follows:

In normal Merge Sort, you divide the array into two e
parts, sort each of them, and then call the `merge` fur
we saw in class. In k-way Merge Sort, you divide the
into k equal parts, sort each part, and then run a k-way `merge` function that you will need to write. Whereas 2-way `merge` needs two indices to traverse the two halves of the array, k-way `merge` will need k indices, so an array of indices may be a good idea.

You may assume that k is at least 2, and at most `myArray.size()`, so you don't need to error-check this. You can define any helper functions that you think would be useful. Test your implementation carefully with all three functors from your file `functor.h`/`functor.cpp`. Do the testing in a separate file (e.g., `test.cpp` or `main.cpp`), which you should not submit.

Problem 4(c) Analyze your k-way Templated Merge Sort (10%)

Analyze the running time of your new k-way templated Merge Sort as a function of k and the array size `n = myArray.size()`. Which algorithm would you say that your k-way `mergeSort` resembles when k=n? What is its running time in that case? Submit your analysis in a file called `hw5.txt` (or if you want to use a different file type such as a pdf, indicate where your answer is in your README).

Problem 5 (The Fatalist Hypothesis, 15%)

A couple instructors are wondering whether there is a high correlation between grades earned in CSCI 170 and CSCI 104. Professor Fatalist has a very bold hypothesis: for any two students A and B, if A got a strictly higher grade than B in 170, then A will have a grade greater or equal to B's in 104. In addition, if A got a strictly higher grade than B in 104, then A will have a grade greater or equal to B's in 170.

Professor Optimist doesn't think the Fatalist Hypothesis is accurate, and he wants your help to prove it. You must write the following function in file `fatalist.cpp`:

```
bool FatalistHypothesis(std::vector<std::pair<int,int> > {
```

The input is a vector of pairs. Each pair refers to the CSCI 104 grade (the first value in the pair) and the CSCI 170 grade (the second value) for a specific student. Your should return true if Professor Fatalist's Hypothesis t to be correct for this data set, and false otherwise.

You may use anything from the STL you like to help solve this problem. You can also import any data structures and ADTs you created in a previous assignment, or use anything you wrote for the current assignment.

Your algorithm must run in worst-case O(n log n) time.

Submission Link

You can submit your homework here. Please make sure you have read and understood the submission instructions.

WAIT! You aren't done yet. Complete the last section below to ensure you've committed all your code.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your hw5 directory to your `hw-usc-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. Go to your home directory: `$ cd ~`
2. Create a `verify` directory: `$ mkdir verify`
3. Go into that directory: `$ cd verify`
4. Clone your hw-username repo: `$ git clone git@github.com:usc-csci104-fall2018/hw-username.git`
5. Go into your hw5 folder `$ cd hw-username/hw5`
6. Recompile and rerun your programs and tests to ensure that what you submitted works.