CS106X
Winter 2017

Chris Gregg

Section Handout #4

This week has practice with sorting algorithms, classes, and recursion using memoization.

1. Insertion Sort
Suppose you are sorting the following vector using insertion sort:

```
{29, 17,  3, 94, 46,  8, -4, 12}
```

Walk through the insertion sort algorithm and show the state of the vector after each of the first three passes of the outermost loop (that is, the loop where you find the minimum element).

2. Merge Sort
Suppose you are sorting the following vector using merge sort:

```
{29, 17,  3, 94, 46,  8, -4, 12}
```

Walk through the merge sort algorithm and show the sub-vectors that are created by the algorithm and show the merging of the sub-vectors into larger sorted vectors. As a hint, this vector should have three split steps, and three merge steps.

3. It Was The Best of Cases, It Was The Worst of Cases
Recall that we said in lecture that quicksort performs in O(n log n) in the best case but $O(n^2)$ in the worst case. Let's explore why that might be. Here's an example of a vector that quicksort will sort in O(n log n) (assuming you pick the first element in the vector as the pivot):

```
{4, 1, 3, 5, 6, 7, 2}
```

And here's an example of a vector with the same values that quicksort will sort in $O(n^2)$ (again, assuming you pick the first element in the vector as the pivot):

```
{1, 2, 3, 4, 5, 6, 7}
```

Using these vectors, figure out what causes the differences in quicksort's performance, and then construct your own vectors for each case using different values. Hint: look at the patterns in the pivots that are chosen by the algorithm.

4. Sorting in O(n) Time
During lecture we looked at several general purpose sorting algorithms, but the best performance we could get was O(n log n). It turns out you can get better performance, but only with certain data.

In this problem, write a function `linearSort` that takes in a vector of integers to be sorted and an integer *k*. You can assume that every element in the vector will be between 0 and *k*. The function should run in O(n + *k*) time (that is, it's bounded by the number of elements and by the largest possible element). You can create auxiliary data structures if needed.

Thanks to Marty Stepp, Victoria Kirst, Jerry Cain, and other past CS106B and X instructors for contributing content on this handout. Thanks to Leslie Tu and Wesley Rodriguez for proofreading.

5. Reciprocate and Divide
Consider the `Fraction` class from lecture.

```
class Fraction {
public:
  Fraction();
  Fraction(int num, int denom);
  void add(Fraction f);
  void mult(Fraction f);
  float decimal();
  int getNum();
  int getDenom();
  friend ostream& operator<<(ostream &out, Fraction &frac);
private:
  int num;
  int demon;
  void reduce();
  int gcd(int u, int v);
}
```

We're going to expand the interface with two additional methods.

Add a public method named `reciprocal` to the `Fraction` class which converts the fraction to its receiprocal (note that by definition the reciprocal of a number `x` is a number `y` such that `xy == 1` holds). You can assume the numerator and denominator will always be non-zero.

Add a public method named `divide` to the `Fraction` class that takes in a `Fraction` `f` and divides the original `Fraction` by `f`. You can assume the numerator and denominator will always be non-zero.

6. Circle (Class) of Life
Write a class named `Circle` that stores information about a circle. Your class must implement the following public interface:

```
class Circle {
  // constructs a new circle with the given radius
  Circle(double r);

  // returns the area occupied by the circle
  double area();

  // returns the distance around the circle
  double circumference();

  // returns the radius as a real number
  double getRadius();

  // returns a string representation such as "Circle{radius=2.5}"
  string toString();
};
```

You are free to add any private member variables or methods that you think are necessary. It might help you to know that there is a global constant `PI` storing the approximate value of π, roughly `3.14159`.

7. First Date (Class)
Write a class named `Date` that stores a month and day. Your class must implement the following public interface:

```
class Date {
  // constructs a new date representing the given month and day
  Date(int m, int d);

  // returns the number of days in the stored month
  int daysInMonth();

  // returns the day
  int getDay();

  // returns the month
  int getMonth();

  // advances the Date to the next day, wrapping to the next month and/or year if
  // necessary
  void nextDay();
};
```

You are free to add any private member variables or methods that you think are necessary. You can also ignore leap years.

8. Align DNA Strands
Oftentimes, biologists need to find the similarity of two DNA strands (which, for the purposes of this problem, can be thought of as strings where each character is one of `A`, `C`, `G`, or `T`). One method that they use is to take two strands, `x` and `y`, and insert spaces to make strands `x'` and `y'`, both of which have the same length, but don't have spaces in the same location. For example, given the DNA strands `GATCGGCAT` and `CAATGTGAATC`, one possible alignment (with the spaces represented as underscores) is:

```
G_ATCG_GCAT_
CAAT_GTGAATC
```

As you might imagine, there are many ways to align two sequences, but some ways are better than others. To find the best one, you can score each alignment based upon the position `j` as follows:

- +1 if `x'[j]` and `y'[j]` are the same and neither is a space
- -1 if `x'[j]` and `y'[j]` are different and neither is a space
- -2 if either `x'[j]` or `y'[j]` is a space

For example, the above example has a score of -4 (take some time to walk through why that is if you're confused).

Write a function `alignStrands` that, given two DNA strands, recursively finds and returns the highest possible alignment score. In order to make sure this returns in a reasonable time, cache the results of recursive calls so that you don't unnecessarily repeat recursive calls.