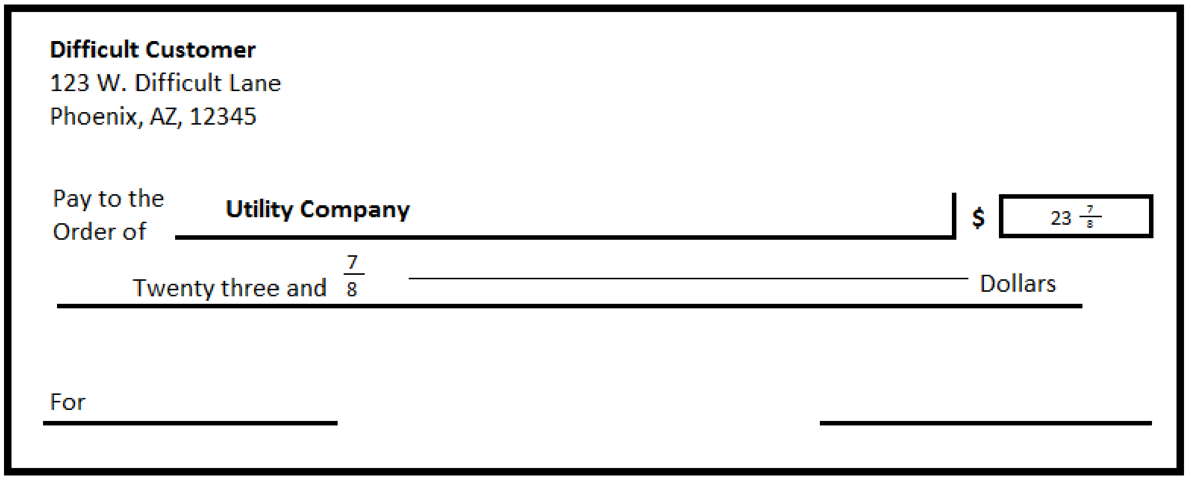
# Pre-Interview Programming Challenges

Directions: Complete one of the following challenge problems using either **C#**, **Java**, **JavaScript** ***(ES5 or ES6)***, or **Python**. Reply to the email that contained this document and attach your solution code file(s). If you complete more than one problem then you will be evaluated based on your best/most-correct solution. Be ready to discuss your code if you are selected for a follow-up interview.

## Challenge Problem 1: Writing Checks the Hard Way

### Description

In an effort to be as difficult of a customer as possible, you decide that from now on you will write all of your checks using a fraction with the smallest possible positive denominator for the cents portion of the amount. For example, to pay for a bill that is \$23.87 you want to fill out your check like the one below.



**Difficult Customer**
123 W. Difficult Lane
Phoenix, AZ, 12345

Pay to the Order of **Utility Company** \$ $23\,\frac{7}{8}$

Twenty three and $\frac{7}{8}$ Dollars

For

*Figure 1*

Note that any fraction of a dollar that you write into a check will be rounded **down** to the nearest cent, meaning that multiple fractions exist that will round down to the same number of cents (e.g. both \$$\frac{1}{51}$ and \$$\frac{1}{100}$ round down to \$0.01). Given that your goal is to be a difficult customer, you should always use the fraction with the smallest possible positive denominator. The check in Figure 1 could have been written as \$$23\frac{87}{100}$ but it is slightly more difficult to convert \$$\frac{7}{8}$ into cents than it is to convert \$$\frac{87}{100}$.

### Example(s)

- To convert 1 cent (\$0.01), any fraction between $\frac{1}{51}$ and $\frac{1}{100}$ might be used (or an infinite number of other fractions) since, when rounded down to the nearest hundredth, each converts to .01. However, the one with the smallest denominator is $\frac{1}{51}$
- To convert 3 cents (\$0.03), any fraction between $\frac{1}{26}$ and $\frac{1}{33}$ might be used (or an infinite number of other fractions) since, when rounding down to the nearest hundredth, each converts to .03. However, the one with the smallest denominator is $\frac{1}{26}$
- The solution fractions for 7 cents, 8 cents, and 9 cents are $\frac{1}{13}$, $\frac{1}{12}$, and $\frac{1}{11}$ respectively

## Code

Create a function (and supporting fraction class/object, depending on your language choice) that takes in some fraction of a dollar in units of cents (*1¢* to *99¢*) and returns a Fraction object in units of dollars. Here is some example code (and a snippet of output) written in ES6 that you may use as a starting point if you would like.

```
class Fraction {
   constructor(numerator, denominator, units) {
      this.numerator = numerator;
      this.denominator = denominator;
      this.units = units;
   }
   toString() {
      return this.numerator + '/' + this.denominator + ' ' + this.units;
   }
}
function decimalToFraction(cents) {
   // TODO: complete this function
}

// Create a non-sparse array of values from 1-99 (inclusive)
Array.apply(null, Array(99)).map((i,v) => v+1).forEach(
   // Print out each value in the array as cents converted to fraction form
   cents => console.log(cents + ' cent(s) => ' + decimalToFraction(cents))
);
```

**Output:**

```
1 cent(s) => 1/51 dollar
2 cent(s) => 1/34 dollar
3 cent(s) => 1/26 dollar
4 cent(s) => 1/21 dollar
5 cent(s) => 1/17 dollar
6 cent(s) => 1/15 dollar
7 cent(s) => 1/13 dollar
8 cent(s) => 1/12 dollar
9 cent(s) => 1/11 dollar
10 cent(s) => 1/10 dollar
11 cent(s) => 1/9 dollar
12 cent(s) => 1/8 dollar
13 cent(s) => 2/15 dollar
```

## Challenge Problem 2: Tournament Brackets

### Description

Given a list of $2^N$ ranked sports teams (ranked 1 through $2^N$) entering a tournament, you want to determine a set of initial matchups where a specific team T will advance as far as possible in the tournament before being eliminated (or winning). Assume that a team with a higher ranking (lower number) will always win when paired against any team with a lower ranking (e.g. team 1 beats team 2).

### Example(s)

Consider the above problem when N = 2 ($2^2$ = 4 teams) and the team we want to favor is team 3 (T = 3). Compare the brackets in Figure 2. Given the initial matchups in the first bracket, Team 3 doesn't make it past the first round. However, given the matchups in the second bracket, Team 3 is able to progress to the second round. There is no set of initial matchups where Team 3 can win the tournament. Therefore, the initial matchups in the second bracket would be a valid solution for (N=2, T=3).

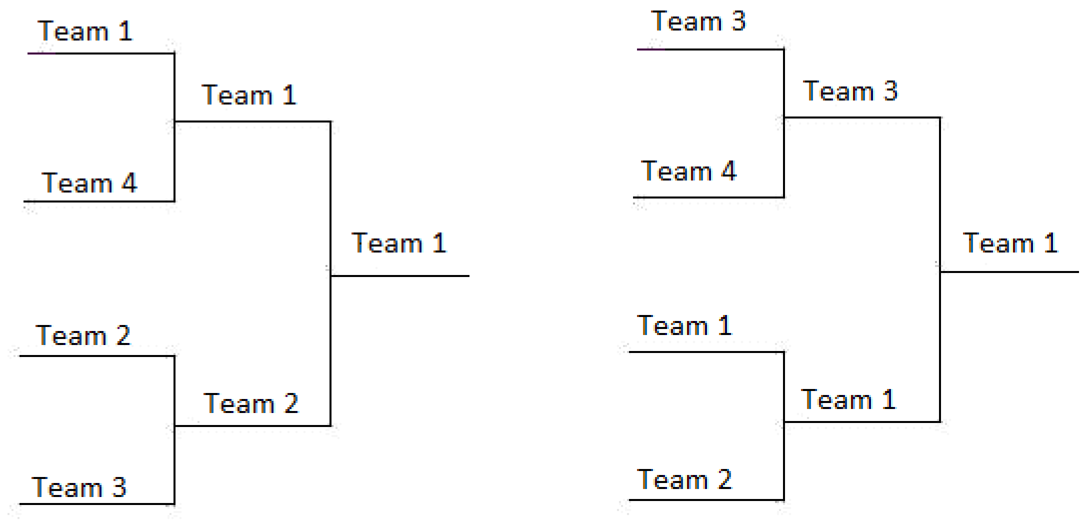


*Figure 2*

### Code

Write a function that, given valid integer values for N and T, outputs the pairings for the first round of the tournament. There may be numerous equivalent solutions, but your function only needs to output one of them. The function should output the solution as a comma-separated list in a format similar to "3-4, 1-2", indicating that Team 3 plays Team 4, and Team 1 plays Team 2. Note that the order of the teams in each matchup does not matter as long as the specified team can advance as far as possible (e.g. 1-2 is equivalent to 2-1).

## Challenge Problem 3: Tricky Pricing

### Description

In an effort to start phasing out pennies, a law has been passed allowing companies to round prices to the nearest nickel. You are starting a company that allows customers to buy items in fixed quantities, but the actual items in the order don't matter, and you want to set the prices of the individual items to maximize the amount of times that the final order sum will round up to the nearest nickel by the highest margins. As such, you need a pricing algorithm that maximizes the number of times the order total rounds up to the nearest nickel.

### Example(s)

Consider a menu of two items, Apples and Oranges, where we set the initial prices to \$20 and \$14 respectively. We are also allowing customers to order in batches of 3 items at a time. The optimal solution is to price apples to \$19.96 and oranges to \$13.96, because any order total of 3 items from this menu will result in the total rounding up by 2 cents.

### Code

Given a list of prices PRICES, and the size of each customer's order, write a function that returns a modified list of prices with the following criteria:

- Reduces the price of each item by 1-5 cents, so that the amount of cents will be between 95 and 99. This makes the price look lower than it is. For example, a price of \$10 should be reduced to something between \$9.95 and \$9.99. All prices provided to the algorithm will be integer dollar values.
- Selects prices that, with some combination of N items of any type, has an order total that will be rounded up by the maximum number of cents. Remember that we are rounding to the nearest nickel, so you can only round up 2 cents.
- If there are multiple prices that round up by the same number of cents, then the best price is the one that results in the highest final order total. For example, an order total of \$9.98 is better than an order total of \$9.93, but an order total of \$9.93 is better than an order total of \$9.99, because \$9.93 rounds up by two cents, while \$9.99 rounds up by only one cent.