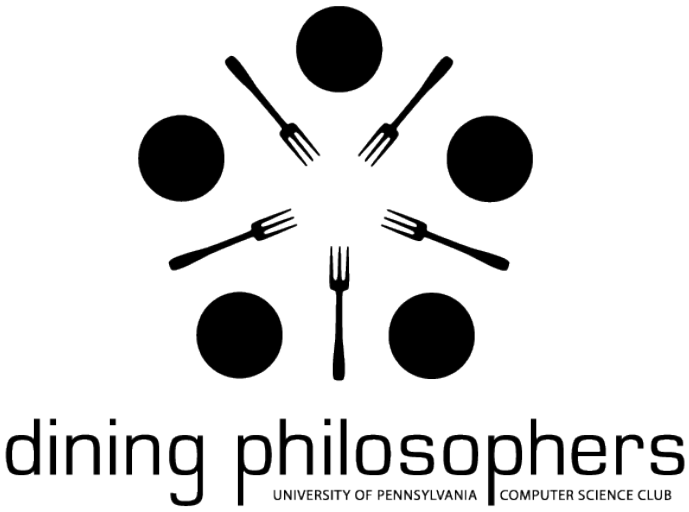
Philadelphia Classic 2012

Hosted by the Dining Philosophers
Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania

February 18, 2012



1. So much for college…

Apparently studying for the SATs is important. You didn't, and you bombed them. Nervous that you won't be able to attend an excellent school like the University of Pennsylvania, you quickly consult the Internets. As luck would have it, the importance of standardized tests has diminished in recent years! You do plenty of activities, but you don't know your GPA. Petrified with the thought of not attending Penn, you rush to your report cards. Because math is hard, you decide to write a program to calculate your GPA for you.

As a note, at your school, GPA is calculated on a 4.0 scale. An A+ or A is a 4.0, an A- is a 3.7, B+ is a 3.3, B is a 3.0, B- is a 2.7, C+ is a 2.3, C is a 2.0, C- is a 1.7, D+ is a 1.3, D is a 1.0, D- is an 0.7, and F is 0.0. Given all your grades, your task is to calculate your average GPA.

Sample Input:

The input will be in the form of a string, with grades separated by spaces. You can assume that all grades are "valid" - "A+", "A", … , "F", and that each grade will be separated from the next by a single space. You can also assume that there will be at least one grade in the input.

```
(1): "A A A B+ D- C+ A B"
(2): "C"
```

Sample Output:

Return, as a `double`, the average GPA of all your grades.

```
(1): 3.1625
(2): 2.0
```

2. When it rains it pours…
Bad news: your GPA isn't that great. This shouldn't come as a shock; after all, you didn't study for your SATs, suggesting a pretty poor work ethic. Despite this recent slew of setbacks, you're armed with a great determination to fix your problem. Over dinner last night, you remember your dad mentioning that crazy old Doc Brown had recently brought his DeLorean in to your father's Body Shop. It becomes clear what you must do. Grabbing your father's keys, you rush to the body shop. Eager to change the past, you jump into the front seat, punch in 2010, and burn rubber.

No sooner have you hit 88 miles per hour than you realize why Doc Brown needed some body work: The DeLorean is broken! It takes a couple of jumps to learn the pattern. Given a current year N and a target year T, if N > 100, T > 100, and N > T, the years the dashboard displays are actually N' and T'. These are determined in the following manner: N' is a prime number less than N, and T' is a prime number greater than T, such that (N' + T')/2 is also a prime number. The pair (N', T') displayed is the pair which minimizes abs(N' - N) + abs(T' - T).

That is, both numbers are fuzzed (N is fuzzed down, T is fuzzed up) to become prime numbers, such that their sum is two times a prime number, while minimizing the overall amount of "fuzzing". You can assume that if you had to fuzz N and T, there is at least one valid pair N' and T' that matches the constraints.

If you have any hope of successfully navigating to 2010 you'll need to write a program that, given a value of N and a value of T, output the displayed values N' and T'.

Sample Input:
Your input will be two integers, N and T. You can assume that N and T are both positive integers.

```
(1): 2000, 1000
(2): 150, 120
```

Sample Output:
Return an integer array with the values N', and T'. If there are multiple pairs (N', T') that satisfy the criteria, any pair will be accepted.

```
(1): [1973, 1013]
(2): [127, 127]
```

3. Fixing the Flux Capacitor!

You worked out the fuzzing pattern, but it took a while and you're way off course, far into the future. It seems like the smartest course of action from here is to repair the Flux Capacitor in the DeLorean and head to 2010. Energized with a sense of pride for your forward thinking, you walk into the nearest storefront. It quickly becomes obvious that you're hideously unprepared to handle the local language. Deflated, you begin to work your way through the language. You see a dictionary conveniently lying on the ground, and pick it up. Oddly enough, the order of letters has changed! The alphabet is no longer A to Z; instead, it is V to $. Indeed, the ordering is different, and two symbols, '#', and '$', have been added to the alphabet as well.[1] The new alphabet is "VOFLTSUQXJGBCAHNMDEZRYKWIP#$". To help piece together modern English, you must first organize what you're seeing. Given an array of strings, you must sort them lexicographically using this new alphabet. You can assume that all strings will be exactly six characters long.

Sample Input:

Your input will be an array of strings. Each string will have six characters, in the range of V -> P or '#' or '$'.

```
[ "JUICE$", "W$#RDS", "EAGLES", "EAGERS", "W$$RDS"]
```

Sample Output:

Output the same array, with the words in the correct order as given by the new alphabet.

```
[ "JUICE$", "EAGLES", "EAGERS", "W$#RDS",  "W$$RDS"]
```

[1] This is not the first time this has happened. In the 19th century, the alphabet was 27 letters, ending with ampersand.

4. Back on task!
Armed with some rudimentary modern English, you enter the hardware store in search of parts. You're not sure which parts you need to buy, or which parts are liable to need spares (after all, you're way more Bill and Ted than Doc Brown). So, you decide to spend as much money as you can and hope for the best.

You're given an array of prices, and your goal is to spend the maximum amount of money without exceeding 100 dollars. (You may purchase the same part more than once.)

Sample Input:
You will be given an array of `double` values. You can assume that all values will be positive, and have at most two decimal points.

```
(1): [7.32, 43.19, 6.00, 12.50]
(2): [4.77, 81.42, 18.53]
```

Sample Output:
Output, as a `double`, the maximum value you can spend under $100.

```
(1): 100.00
(2): 99.95
```

5. Able to pay?

You arrive at the front of the store, parts in hand, and realize that you have no cash. You pull out your credit card, but you're not really sure if it will work in the future. The validation rule for credit card numbers in this new world works like this:

- Double the value of every second digit beginning from the right. That is, the last digit is unchanged; the second-to-last digit is doubled; the third-to-last digit is unchanged; and so on.
- Add the *digits* of all the doubled values, to the digits that were not doubled from the original number
- If this sum is divisible by 10, the credit card is valid.

Your task is to write a program which takes a credit card number (input as a String), and returns whether it is valid or not.

Sample input:
You will be given a string with digit characters. You cannot make any assumptions on the length of the String.

```
(1): 4012888888881881
(2): 4012888888881882
```

Sample output:
Your function returns a Boolean, indicating whether the given credit card number is valid.

```
(1): true
(2): false
```

6. Stand back! I'm going to try science!

It's time to repair the Flux Capacitor, but one hurdle yet remains between you and your task: Doc Brown included a security system on the casing of Flux Capacitor to avoid tampering! Fortunately, the security system mirrors the rules of Minesweeper. Finally, your years of slacking in High School will pay off!

As a reminder, the rules of Minesweeper are as follows: You start with a grid of blank squares. Each square will either have a mine, or nothing, behind it. When you click on a square, one of a few things may happen.

a. If there is a mine behind that square, you lose and the game is over.
b. If there is no mine behind that square, but at least one of the (up to 8) adjacent squares (where adjacent means in the 4 cardinal directions, or either of the 4 diagonals) contains a mine, that blank is "removed", and behind it is revealed the number of adjacent squares that do contain a mine.
c. If there is no mine behind that square, and none of the adjacent squares contain a mine, a 0 is revealed in that square. In addition, since all adjacent squares are "safe", those squares are all revealed as well (in this case, "revealing" a square is equivalent to clicking on it). Of course, if any of those squares have no adjacent mines, they also recursively reveal all of their adjacent squares. We have provided extensive sample input/output which should clarify these rules.

You will be given a minesweeper board, in the form of a 2-dimensional array of characters. This board will indicate which squares have mines behind them. In addition, you will be given a 2-dimensional array indicating which squares you click and the order in which you click them. Your task is to display what the board looks like, to the player, after the specified squares have been clicked. If you click a square that has a mine behind it, you immediately lose, and no further moves are played out (even if the array has more moves remaining).

Sample Input:

The board will be a 2-dimensional array. You can assume the number of rows is equal to the number of columns. Squares with mines will have the '*' character, and squares without mines will have the ' ' (space) character.

The moves will also be stored in a 2-dimensional array of integers. This is an array of height *h*, where *h* is the number of moves performed, and width 2. Each move is stored as an array with 2 elements. The first element represents the "y" coordinate of the square you click, and the second represents the "x" coordinate of the square you click. These coordinates are zero-indexed, meaning that in a 3 by 3 array, to click the middle element in the top row, the move would be [0, 1]. You can assume that we will only specify a click on a valid square that has not already been revealed.

As described above, you will be given two inputs. For each of the sample inputs below, assume that the input "board" is the following 2-dimensional array:

```
[[ ' ', '*', ' ', ' ', '*', '*' ],
 [ '*', ' ', ' ', '*', '*', ' ' ],
 [ '*', ' ', '*', ' ', ' ', ' ' ],
 [ '*', '*', '*', ' ', ' ', ' ' ],
 [ ' ', ' ', ' ', ' ', ' ', ' ' ],
 [ ' ', '*', '*', '*', ' ', '*' ] ]

(1): [ [0, 0], [0, 2] ]
(2): [ ] (no moves)
(3): [ [5, 1], [0, 0] ]
(4): [ [3, 5], [5, 0] ]
```

Sample Output:
You should return a 2-dimensional array, representing what the board looks like to the user. Squares that have not been revealed should have an '_' underscore character. Squares that have been revealed that do not have mines behind them should have a character representing the number of adjacent mines. If a square with a mine behind it was clicked, you should show a '*' asterisk character.

```
(1):
[[ '2', '_', '2', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ] ]

(2):
[[ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ] ]

(3):
[[ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '*', '_', '_', '_', '_' ] ]

(4):
[[ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '_', '_', '_' ],
 [ '_', '_', '_', '4', '2', '1' ],
 [ '_', '_', '_', '2', '0', '0' ],
 [ '_', '_', '_', '3', '2', '1' ],
 [ '1', '_', '_', '_', '_', '_' ] ]
```

7. Gotta go back in time!
Is the Flux Capacitor repaired? Look, maybe you didn't fix every single little tiny bug, but basically you fixed it. Sadly, your repair job has made your destination harder to specify. Namely, you have to specify your destination with two pieces of information: 1) the number of days backward or forward that you will be traveling, and 2) the month that you will be landing in. But hey, you just fixed a completely incomprehensible piece of scientific genius despite a serious lack of scientific knowhow, so you're in pretty high spirits. All you have to do is figure out a way to determine which month you would land in, given the number of days forward or backward you are traveling.

For this problem, you only have to determine, given a number of days, which month you will land in if you travel that many days in the future. Assume today is, indeed, today (February 18, 2012). Also, don't forget leap years! All years divisible by 400 are leap years, all other years divisible by 100 are not leap years, and all other years divisible by 4 are leap years.

Sample Input:
The input will be a positive integer, representing the number of days in the future you are traveling. You can assume that the input will be less than the max value of integers (2^31 - 1).

```
(1): 11
(2): 12
(3): 1155
```

Sample Output:
Output the month that you arrive to the future in. The first letter of the month should be capitalized.

```
(1): February
(2): March
(3): April
```

8. Getting the message across.
You've successfully arrived in 2010, and it's time to convince your past self to study harder! The obvious plan would be to confront your past self, face-to-face. You're no palooka[2], however, and you know that a face-to-face meeting would cause chaos for the space-time continuum!

To ensure the origin of your message is carefully hidden, you decide to encode it using anagrams. Concerned that your past self will take years parsing the message, thus rendering the message pointless, you decide to include an anagram grouper to help your past self along.

Two words are anagrams if they contain the same letters, in the same frequencies. For example, READ and DARE are anagrams, but RABBLE and BARREL are not anagrams.

Your anagram solver will take a string array, and return a list of string arrays, where each new array is a subset of the original, containing a set of anagrams.

Sample input:
Your input will be an array of strings. You can assume that all strings are composed solely of alphabetic characters.

```
["dog", "god", "cat", "act", "fish", "tac"]
```

Sample output:

Output the words such that all anagrams are in the same list, in a list of list of words. The order of the lists, as well as the order of the words in each list, doesn't matter. You simply need to have the lists separated correctly.

```
[ ["dog", "god"], ["cat", "act", "tac"], ["fish"] ]
```

[2] Because you're a a time traveler, the fact that "palooka" is no longer modern slang by any stretch of the imagination is not relevant to you.

9. Better encryption

After all of that work, you decide that anagrams aren't the most secure way to decode a message. You come up with a more improved method for decoding your messages. Your new method is to use a substitution cipher – each letter gets substituted with another letter in the alphabet (or possibly itself).

To aid the writing of such an encrypted message, you give yourself a way to figure out which words an encrypted word might stand for. For example, if the encrypted word is XJJXI, words that it could stand for include "PEEPS" and "TOOTH", because they have the same 'pattern' (with respect to repeated letters). You want to find all words with that pattern.

Sample input:
You will have two inputs – the first will be an array of words, representing all the words in the "dictionary", and the second will be a word that has been encrypted, and the second will be the encrypted String. You can assume that all letters are capitalized.

```
["AXE", "TOOTH", "BANANA", "PEEPS", "APPLE"], "XJJXI"
```

Sample output:
Output an array of words that the word could stand for. This array should be a subset of the words in the dictionary. There may be no words in the dictionary that the word stands for, in which case you should output an empty array. The words should be returned in the same order as they appear in the dictionary.

```
["TOOTH", "PEEPS"]
```

10. Don’t bet your future.
You got the message out, but there’s no way to be certain your past self can decode it in time, or that studying will pay off. You need another edge. Perhaps a basic arithmetic parser can help you work through those tough SAT questions! Eager to waste no time, you get right to work!

Your task is to write a program that takes in a string and produces the evaluation of that string as an arithmetic expression.

Sample Input:
Your input is a string consisting of integers, parentheses, addition symbols, and multiplication symbols representing a valid arithmetic expression.

```
(1): "(3+2)*5"
(2): "1+((4+7)*3)"
(3): "(12)"
```

Sample Output:
Output an integer that is the value of the given expression, using correct order of operations.

```
(1): 25
(2): 34
(3): 12
```

Epilogue.

You changed the past and improved your grades! Now you’re sure to blow the minds of every college admissions officer who looks at your application (even the ones from great schools like Penn)! With renewed self-confidence and a step in your stride, you go to Doc Brown’s house and thank him for letting you borrow his DeLorean.