CSC 321 | **Homework 4** | your names go here

*Due November $4^{th}$, 2019*

**Directions:**

- Write your solution using LaTeX. You should add your answers to this `tex` file, then submit the compiled `pdf` using **Graddescope**.
- Try to make your answers as clear and concise as possible.
- This is a partnered homework. Partners should work collaboratively solve all problems, then work together on the write-up. You are welcome to discuss general strategies with other groups, but you should never reveal specific details of a solution, nor should you show your written solution to anyone else.
- Feel free to add more boxes if your solutions cross a page boundary or if you've solved part of the extra credit.

## Problem 1: Canonical Coins

On homework 3, you provided a greedy algorithm for making change, and proved that for one set of coins (Euro denominations) the greedy algorithm is optimal. In class, you found a counterexample showing that for some sets of coins, the greedy algorithm is not optimal. We will call a set of coins *canonical* if the greedy algorithm works (always finds the minimum number of coins) for that set, and *non-canonical* otherwise.

(a) Design and analyze a dynamic programming algorithm to determine the minimum number of coins required to make change for $n$ cents with denominations $c_1, c_2, \ldots, c_k$.

Your Problem 1a solution goes here.

(b) To check whether a given set of coins is canonical, we could run the greedy algorithm side-by-side with the dynamic programming algorithm on $1, 2, \ldots$ cents, and if the greedy algorithm always gives the right answer, we conclude that the set of coins was canonical. However, we need to know when to stop. Give (with proof) an upper bound for how high you need to check before declaring the set of coins $c_1, c_2, \ldots, c_k$ canonical. In other words, show how to find an integer $n_C$ such that for coin set $C = \{c_1, c_2, \ldots, c_k\}$ you can prove that if the greedy algorithm ever fails on $C$ it will fail before $n_C$.

Your Problem 1b solution goes here.

(c) Implement your algorithm from Problem 1: part (b) in Python. A starting point has been provided for you in `coins.py`. Your program should read from standard input, and write to standard output.

**Input:** The input contains $k$ coin denominations $c_1, c_2, \ldots, c_k$ as space-separated integers, in increasing order: $0 < c_1 < c_2 < \ldots c_k < 10^6$.

**Output:** If the set of coins is canonical, print the string `CANONICAL`, and the range of numbers you had to check to prove it. If the set of coins is not canonical, print the string `NON-CANONICAL`, and the smallest number on which the greedy algorithm fails.

**Sample Inputs and Outputs:**

```
INPUT:                  OUTPUT:
1 2 3                   CANONICAL
                        checked n=1...3

1 5 10 25               CANONICAL
                        checked n=1...50

2 3                     NON-CANONICAL
                        fails on n=1

1 9 10                  NON-CANONICAL
                        fails on n=18
```

## Problem 2: Moving on a Checkerboard

Suppose you are given an $n \times n$ checkerboard and a single checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step, you may move the checker to one of three squares:

1. the square immediately above,
2. the square that is one up and one to the left (but only if the checker is not already in the leftmost column), or
3. the square that is one up and one to the right (but only if the checker is not already in the rightmost column).

Each time you move from square $x$ to square $y$ you receive $P(x, y)$ dollars. You are given the values $P(x, y)$ for all pairs $(x, y)$ for which a move from $x$ to $y$ is legal. $P(x, y)$ may be negative.

Design and analyze a polynomial-time algorithm that figures out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible. Your algorithm is free to pick any square along the bottom edge as a starting point, and any square along the top edge as a destination, in order to maximize the number of dollars gathered along the way.

Your Problem 2 solution goes here.

## Problem 3: Pretty-printing

Suppose we have a paragraph of text, and we want to print it neatly on a page. The paragraph consists of a list of words $w_1, w_2, \ldots, w_n$; each word $w_i$ has length $\ell_i$. The maximum line length is $M$. (Assume that $\ell_i \leq M$ for all $i$.) We assume we have a fixed-width font and ignore issues of punctuation and hyphenation.

Consider a line containing words $w_i, w_{i+1}, \ldots, w_j$, and using only one space between words. Because the words must fit within the maximum line length, we know that:

$$\text{length of this line } = (\ell_i + 1) + (\ell_{i+1} + 1) + \cdots + (\ell_{j-1} + 1) + \ell_j \leq M$$

The "slack" space on a line is the number of spaces remaining at the right margin, so for this line it is the value:

$$\text{slack of this line} = M - \Big((\ell_i + 1) + (\ell_{i+1} + 1) + \cdots + (\ell_{j-1} + 1) + \ell_j\Big)$$

The penalty is the sum over all the lines (including the last) of the *squares* of the slack of all lines in the paragraph.

(a) Design and analyze a dynamic programming algorithm to find the best way to print a paragraph, where "best" means "with smallest penalty". Include a recursive definition of the optimal value that motivates your algorithm.

Your Problem 3a solution goes here.

(b) Implement your algorithm in the file `pretty-print.py`; it should print an optimal division of words into lines. The input should be a number ($M$, the maximum line length) and a file containing some words; you should assume that a "word" is any contiguous sequence of characters not including whitespace. The program should print the full text of the file, split into lines appropriately, followed by the numerical value of the penalty.

For example, consider the input in `Lem.txt`.[1]

With maximum line length 25, the output should look like:

```
Not far from here, by
a white sun, behind a
green star, lived the
Steelypips, illustrious,
industrious, and they
hadn't a care: no spats
in their vats, no rules,
no schools, no gloom,
no evil influence of the
```

[1] This is a line from Stanislaw Lem's *The Cyberiad*.

```
moon, no trouble from
matter or antimatter-for
they had a machine, a
dream of a machine, with
springs and gears and
perfect in every respect.
Penalty: 137
```

With maximum line length 75, the output should look like:

```
Not far from here, by a white sun, behind a green star, lived the
Steelypips, illustrious, industrious, and they hadn't a care: no spats
in their vats, no rules, no schools, no gloom, no evil influence of the
moon, no trouble from matter or antimatter-for they had a machine, a
dream of a machine, with springs and gears and perfect in every respect.
Penalty: 199
```

Several more test cases have been provided for you in the `pretty_print_inputs` folder. You also should feel free to create your own input examples.

### Extra Credit: Bad Recurrence

In class, we saw the following recurrence relation for the divide-and-conquer word count algorithm:

$$T(n) = \begin{cases} \sum_{i=1}^{n-1}(T(i) + T(n-i)). & n > 1 \\ c. & n = 1 \end{cases}$$

where $c$ is some constant. Prove a big-O upper bound for this recurrence.