Instructions (for the actual exam)

This printed exam has been provided to you as space for scratch work and as a reference for you as you work through the problems on your laptop. We will not grade or look at any of the work written here. The Exam Reference Sheet is printed at the end of the exam.

However, since we have make-up exams scheduled on different days, you must turn in this paper midterm at the end of the exam period. Do not throw it away or take it with you. In BlueBook, the first problem asks you for the number printed above ("Your exam number") so that we can confirm everyone has returned their hard copy. It also asks you for an electronic signature agreeing that you abided by the Stanford Honor Code. Any exam that does not provide a complete answer to this problem (exam number + signature) will automatically receive a 0.

You have two hours to complete this exam. There are five total problems on the exam, and some have multiple parts. Some general tips for tackling problems:

- We recommend looking over the entire midterm before getting started so that you can budget your time well.
- Pay attention to which functions ask you to print a value versus return a value.
- Please type up your work and write down your ideas, even if you get stuck on a problem. We will do our best to give you partial credit, as long as we can understand what you wrote.
- You do not need to comment your functions (unless you find it helpful as you work). We will not be grading on style.

Good luck!
Trace: iddle?

Examine the code below and answer the following questions:

- What is returned by the function call:
  
riddle('mirrormirror')

- What is each function doing? (a brief description here is fine!)

```python
def fiddle(t, s):
    piddle = t.find(s)
    while piddle != -1:
        middle = piddle
        piddle = t.find(s, piddle + 1)
    return middle

def griddle(i):
    i += i

def riddle(s):
    t = ''
    i = 0
    while i < len(s):
        if i == fiddle(s, s[i]):
            t += s[i]
            s = s[i+1:]
            griddle(s)
            i = -1
        i += 1
    return t
```
Karel: StrikerKarel

NOTE: Just like in assignment 1, in this problem you cannot use for loops, variables, continue, break, and other Python features that were taught after Karel.

We've already worked a little bit with athlete Karel, but now we're taking it to the next level! Inspired by the events of the 2016 Rio Olympics (where Stanford athletes earned more medals than all but nine countries) as well as the 2018 Men's World Cup, Karel has decided to train for the 2020 Olympics in Tokyo. But in order to become the world's best striker, Karel first needs to learn ball control! Write a program StrikerKarel.py to help Karel learn the basics of the role.

Karel starts off at the corner of (1,1), standing on top of the ball (a beeper). There are no other beepers in the world. There is one "goal" in the final column of the world, represented by an opening in a wall between the last and second-to-last columns. Karel needs to move the ball from (1,1) to this "goal."

Two challenges make Karel's task more difficult:

- In order to simulate dribbling, Karel cannot move more than once without the ball touching the ground. That is, the ball must be put down each time Karel changes location.
- In order to simulate defenders, there are walls blocking Karel's potential paths. Karel must move around these walls.

![Figure 1](image.png)

**Figure 1:** A potential starting world for StrikerKarel (left) and Karel's ending position at the goal along the east wall (right).

You can make the following assumptions about the worlds:

- The world will have at least two columns but may be any height.
There will be exactly one "goal" in the final column. The goal corner is the only corner in the world that is blocked to the north, east, and south simultaneously.

At every reachable corner other than the goal, Karel will be able to find an opening in the wall that allows them to move east without first moving west.

Karel's ending direction does not matter, so long as Karel is in the goal and on top of a beeper.

For example, the world on the right is **not valid** because if Karel moves north following the arrow, it will be trapped by walls to the north, south, and east in the column indicated by the box. Karel would have to move west in order to move east again. Your program does not need to handle this kind of world.

The final program will call your `main()` function. In addition to completed `turn_right()` and `turn_around()` functions, we've provided two empty suggested functions (`dribble()` and `find_wall_opening()`) that you could fill in and then call in `main()` to help you break down the problem. But you should feel free to write additional functions to further decompose the program in whatever way makes sense to you! Remember strategies for top-down decomposition and pre- and post-conditions if you get stuck.

```python
def turn_right():
    turn_left()
    turn_left()
    turn_left()

def turn_around():
    turn_left()
    turn_left()
```

![Figure 2: An example of a valid world (left) and invalid world (right)](image-url)
def dribble():
    pass

def find_wall_opening():
    pass

def main():
    ```
    Karel becomes the best striker in the world!
    ```
    # YOUR CODE HERE
    pass
Images: Three is better than one

In this problem, not only will you process multiple images at once, you’ll also triplicate the original images! We’ve broken the problem down into two parts for you:

a) Creating a new image that contains three copies of the original image
b) Processing many images based on filenames written in a text file

a) Given an image filename, create an output image that is 3 times as wide as the original. Copy the original image, upside down, to the leftmost 1/3 of the output image. Leave the middle copy of the image the same as the original. The rightmost 1/3 of the output image should be a horizontal reflection of the original image (the top left corner pixel of the original image should be in the top right corner of the copied image).

```
def triplicate(image_filename):
    # YOUR CODE HERE
    pass
```

Figure 3: Triplicated version of a Karel image. The original image is shown below.

Figure 4: The original Karel image that is triplicated above

b) We have a text file with the names of the image files we need to process, but they’re all missing file extensions. There can be multiple filenames separated by spaces on a given
line, and each filename is missing the correct file extension (you’ll need to add `.png` to the end of the name).

An example text file might look like this:

```
karel sky mt-rainier
greenland-fire
stop-sign leaves curb banana
```

Write a function that takes in the filename of a text file, parses the image filenames, and then calls your `triplicate()` function from part (a) on each image file. For this problem, you can assume that your `triplicate()` function works completely and without bugs.

```python
def process_all_images(filename):
    # YOUR CODE HERE
    pass
```
Dictionaries: Counting characters

Write a function `frequency_analysis()` that takes in a string `s`, creates a dictionary to keep track of the amount of times each character shows up in the string, and then prints out the characters and their counts in alphabetic order. You should treat uppercase and lowercase chars in `s` as the same character (i.e. your function should be case insensitive). After the count dict is built, print out all the chars seen in alphabetical order, 1 per line. For example, the function call `frequency_analysis('PangolinPartyPRO')` would produce the following output:

```
    a 2
    g 1
    i 1
    l 1
    n 2
    o 2
    p 3
    r 2
    t 1
    y 1
```

YOU MUST USE A DICTIONARY TO KEEP TRACK OF COUNTS.

```python
def frequency_analysis(s):
    pass
```
Console program: Find range

Write a console program (function) called `find_range()` that reads in a list of integers, one per line, until a sentinel value of 0 (which you should be able to change easily to some other value). When you read in the sentinel value, your program should display the smallest and largest values that have been entered so far, as illustrated in the sample run below (user input is indicated in blue and bold):

```
>>> This program finds the largest and smallest numbers.
>>> Enter an integer or 0 to stop: 11
>>> Enter an integer or 0 to stop: 17
>>> Enter an integer or 0 to stop: 42
>>> Enter an integer or 0 to stop: 9
>>> Enter an integer or 0 to stop: -3
>>> Enter an integer or 0 to stop: 35
>>> Enter an integer or 0 to stop: 0
Smallest: -3
Largest: 42
```

# Constants

```python
SENTINEL = 0
```

```python
def find_range():
    # YOUR CODE HERE
    pass
```