Programming Assignment 2

Introduction

In this problem set you will practice with functions, and you will use recursion to build solution for two computational problems. Don't be intimidated by the length of this problem set. It's a lot of reading, but it is very doable.

PART I: Use functions to have clean code

In PA1, we wrote a simple version of KNN algorithm using conditionals, and looping structures. Download `pa2.zip` from here. You can find a sample solution for PA1 in `dirty_knn.py`. Open `dirty_knn.py` and try to read through the code and see what it is doing. You will see that it is very difficult to read and grasp what a long spaghetti code which is full of nested loops and conditionals is actually doing, especially when there are no comments describing the code functionality!

In this part we want to take the dirty code and with the help of functions' magic, turn it into a clean and beautiful code which is easily understandable!

We use functions to **decompose** our programs into multiple sections where each section performs a single specific task. For a function name, we choose a name that reflects the essence of the specific task that it is doing.

Let's start to decompose the code in `dirty_knn.py` into meaningful self contained functions.

1. The first task that we are doing is reading and loading the `trainset`. Why not having this functionality in a dedicated function?

```
def load_trainset(path):
    pass
```

Open `neat_knn.py`, and implement this function as instructed in its docstring. Notice that we are giving the `path` to trainset file as an argument. We do this because although functions should do a specific single task, but at the same time they should be as general as possible. This is the **art** of the programmer to recognize if a function which do a very specific task can be generalized without hurting the nature of the task that it is doing!

We usually generalize a function by adding more input arguments to it. For example, in `dirty_knn.py` when we read the `trainset` file, we have the assumption that the filename is `trainset.txt` and it is located in the same directory as our python file. What if in future the filename will be different or it will be in a different location? Then we have to go over our code and wherever we have used that filename, change it to the updated one! We can avoid this by generalizing the function that reads the `trainset` and pass the path to our trainset file as an argument.

2. The next thing we do is reading and loading the information we have in the testset file. We will decompose that part of code into a dedicated function like this:

```
def load_testset(path):
    pass
```

Implement this function as instructed in its docstring in `neat_knn.py`.

3. Next we are running the KNN algorithm for the given `testset`, and we append the predicted categories to a list.

Move all this logic to the following function in `neat_knn.py` .

```
def knn(train_points, train_cats, test_points, Ks):
    pass
```

So far so good, but you should have noticed that the above function is bulky and is doing multiple tasks! For example it is finding the nearest neighbors, computing the euclidean distances, as well as finding the most frequent categories between neighbors. Hence we should still decompose these tasks into their own separate functions. We do so by following the next steps.

4. Let's decompose that piece of code which is computing the distances with categories for a particular test point under the first `for` loop in `knn` function above.

We will decompose this piece into two functions:

```
def get_distances_with_categories(train_points, test_point):
    pass

def euclidean_distance(p1, p2):
    pass
```

The first one is computing and returning the list of distances between a given test point and a list of training points with the corresponding categories. Inside this logic we are using yet another task which is computing the euclidean distance between two points, we refactor that part into `euclidean_distance` distance function and will call this function inside `get_distances_with_categories`. **Don't forget** to generalize here! Write your `euclidean_distance` function in such a way that it can work for any size of dimension.

5. After computing distances, we are sorting and selecting those elements in distances list, which are considered to be the k nearest neighbors to a test point. This can also be refactored to a function.

```
def get_nearest_neighbors(distances, k):
    pass
```

6. Now back to `knn` function, up to now we should have a list called neighbors which is returned by calling `get_nearest_neighbors` function. We then compute the category frequencies in that list. Let's refactor this logic into its own function as well!

```
def get_category_frequencies(neighbors):
    pass
```

We want to use **generalization technique** here as well as we don't want to be limited to only 3 categories. Hence we use a dictionary between categories and their associated frequencies to be able to handle different type of categories.

7. At the bottom of the for loop in `knn` function, we are finding the category which has the maximum frequency. If there were multiple categories with max frequency, we chose that one which were smaller than the others. This part also demands its own function, so decompose it into this function:

```
def find_most_frequent_category(freqs):
    pass
```

This function is already implemented for you. Don't forget to take a look at it and try to understand how it is doing its task!

8. Finally at the end of `dirty_knn.py`, we are writing the result list into a file. This is also something that should be refactored into a function:

```
def write_results(path, result_list):
    pass
```

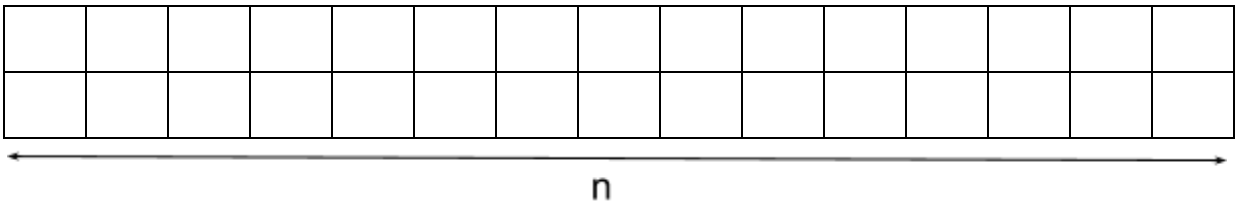
Now take a look and compare the `dirty_knn.py`, and `neat_knn.py`. Isn't the code in the latter one just beautiful compared to that spaghetti code in the former one? :D

PART II: The *art* of recursion

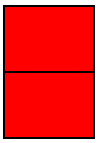
We practice recursion by solving the following two problems step by step recursively.

Tiling a $2 \times n$ board

We have a $2 \times n$ board like this:



We also have an unlimited number of 1×2 tiles like this which can be rotated by 90 degrees.



The problem is to find the number of different ways that we can tile the whole board in such a way that there will be no cell that is not covered.

This problem may look mind boggling at first glance! How are we supposed to count the number of different ways!! But if we think recursively, we will see what a simple and elegant solution it has.

When you want to think recursively, the **first step** is always assuming that there exists a magical function which gets state of the main problem as input argument, and returns the correct solution for that state.

What does define the state of the tiling problem? The simplest thing that comes to mind, is the board's length which is n here. Hence we assume there is a magical function like f, where $f(n)$ is the solution for our main problem.

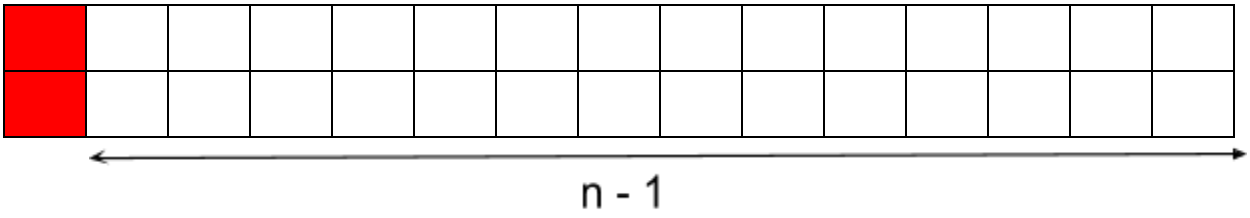
The **second step** is to figure out if we had the solution to smaller subproblems (by using the same magical function), how can we combine those solutions to build the solution for the main problem. If you can figure this out, then you're almost done!

One way is to explore the actions that you can make and see how taking those actions may result in smaller subproblems. In our example if we put one tile on the board, the state will change. We are usually interested in those actions which change the state to similar states but with smaller sizes.

The trick in tiling problem is to use this assumption that we know the leftmost cells on the board are going to be covered some time, why not try to tile them first?

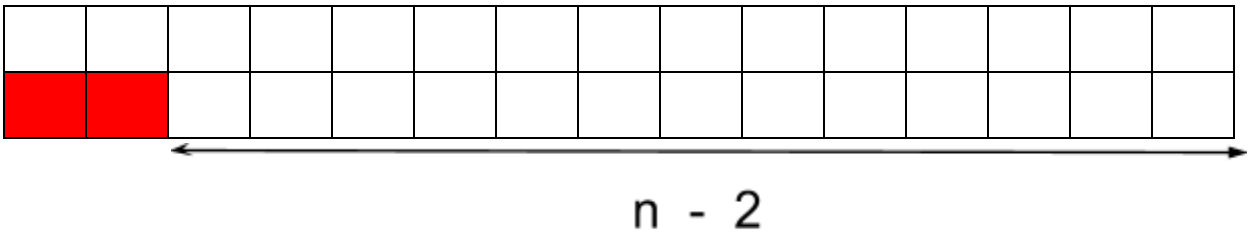
If we want to cover the leftmost cells first, then there are two possible actions:

1) put a vertical tile like this:



In this case, the number of ways that we can tile the board, is the number of ways that we can tile the remaining cells in the $2 \times (n-1)$ board. We already know that $f(n-1)$ has the solution for this problem as the remaining cells are a board of length $n-1$! So we say the number of ways to tile a $2 \times n$ board, given that a red tile like in the picture is put on the board is $f(n-1)$.

2) put a horizontal tile like this:



If we do that, we know we are forced to put a same horizontal tile on top of the first tile, to cover those two cells, and all remain is to tile the $2 \times (n-2)$ board. In how many ways can we do that?

Again we have our magical function f, and number would be $f(n-2)$ as the remaining cells are a similar board of length $n-2$.

Let's summarize what we have achieved so far. We assumed that there exists a magical function like f that if it gets the state of the tiling problem as an input, it will return the number of different ways that we can cover a board of cells. We also saw how the actions of covering the leftmost cells will result in similar smaller problems for which we had also a solution because of the magical function f.

Now we need to think how we can **combine** the solution to smaller subproblems to come up with a solution for the main problem. In our tiling example, this step is easy. We had only two possible actions to cover the leftmost cells, we know there are $f(n-1)$ ways to cover the whole board if we take the first action, and there are $f(n-2)$ ways if we take the second action. As the two actions can't be taken at the same time, we use the **rule of sum**, and say, the number of ways to cover the whole board is $f(n-1)+f(n-2)$. As we assumed that the solution to the main problem is also $f(n)$, we have:

$f(n) = f(n-1) + f(n-2)$.

Now as we have the recurrence relation, the **final step** is to think about base cases. Base cases are the smallest possible shape of the main problem, for which we can easily return the solution without any heavy computations.

In our example if the board is 2×1 (has length 1), we know the number of ways to cover it, is 1, and if the board is 2×0 (has length 0), we have only 1 way to tile it and it is doing nothing!

In other words, we have:

$f(0) = 1$ and $f(1) = 1$.

Now you have all the materials to write a recursive solution for this problem. Open `recur.py` and implement the following function based on the above description:

```
def tiling_ways(n):
    pass
```

Fast Matrix Exponentiation

Assume we have the following 2×2 matrix, and the problem is to find its n'th power:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

One way to do so is just start from the base matrix and multiply it by itself n times. This works, but there is a much faster way to do it, and it uses recursion!

Let's think recursively about this problem.

The **first step** was making the assumption of having a magical function which has the solution to our problem. What will we use to define as the state of the problem is the matrix itself and the power n. Therefore we assume there is a function like pow, and $pow(A, n)$ returns the n'th power of A.

The second step was exploring the smaller subproblems and figuring out if we can somehow combine the solution to them to come up with a solution to our main problem. This is the trickiest step in thinking recursively. In the fast matrix exponentiation problem, it is easy to figure this out if we consider the exponentiation rules.

Assume, n is an even integer, then based on the exponentiation rules, we have $A^n = A^{n/2} \times A^{n/2}$, we know $A^{n/2} = pow(A, n/2)$, therefore we have a recurrence relation here:

$$pow(A, n) = pow(A, n/2) \times pow(A, n/2)$$

And if n is an odd integer, then $n-1$ should be even, thus we have $A^n = A \times A^{(n-1)/2} \times A^{(n-1)/2}$, or in other words:

$$pow(A, n) = A \times pow(A, (n-1)/2) \times pow(A, (n-1)/2)$$

Alright, we only need to figure out what are the base cases.

We know if $n = 0$, $pow(A, 0) = I_{2\times 2}$, where I is the identity matrix. If $n = 1$, then $pow(A, 1) = A$

This way of matrix exponentiation is much much faster than just multiplying A, n times with itself. One notice though, if you look at each of the recurrence relations above, you will see

there is a term on the right side which is repeated. You should avoid to compute that repeated term again and again to achieve the fast nature of this solution. For example, in $pow(A, n) = pow(A, n/2) \times pow(A, n/2)$, when you compute the first $pow(A, n/2)$ recursively, store the result in a variable and multiply it by itself, rather than recomputing the second $pow(A, n/2)$ again recursively.

You have to implement the following function in `recur.py`:

```
def fast_mat_exp(A, n):
    pass
```

Deliverables

1. Local

Download *pa2.zip* from here. It contains three files. You have to complete the code in `neat_knn.py` and `recur.py` based on the above description. Do not forget to write down the number of hours (roughly) you spent on this problem, and the names of the people you collaborated with, and of course your name.

2. HackerRank

Open this link, and sign up in this contest as before.

a) Open the knn challenge, and copy and paste your code that you wrote in `neat_knn.py` to the code editor in that challenge in the space that is marked between:

YOUR CODE starts here:

YOUR CODE ends here!

You can run your code to see if you can pass the sample test case. If you are successful, submit your code to see if you can pass all the test cases.

b) Open the tiling problem challenge, copy and paste your `tiling_ways` implementation in the code editor in its appropriate place. Test your code and submit to see if you can pass all the test cases.

c) Open the fast matrix exponentiation challenge, copy and paste your `fast_mat_exp` implementation in the code editor in its appropriate place. Test your code and submit to see if you can pass all the test cases.