Due: Friday, May 17 **at 5 pm!**

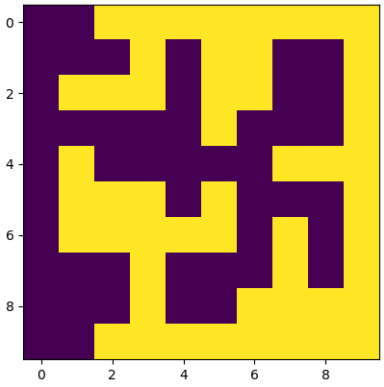
In this homework assignment, you will be implementing a version of Metropolis-Hastings to perform some simplified Gerrymandering analysis. Your solutions should go in the provided `hw7.py` file, which will be submitted through the course gradescope.

Part 1. Representing a Districting Plan

To start us off, we need to formalize the specific problem we will be working on in this assignment. We are going to perform redistricting on a square domain, which has been divided into a grid of "precincts" which can be assigned to one of two possible districts. We will represent our domain as an $N \times N$ matrix, with the entry at index $[i, j]$ representing the precinct located at row i and column j in our grid. The matrix will be filled with 0s or 1s, indicating the assigned district for each of our precincts.

For example, the matrix below will result in the corresponding districting plan displayed beside it (note the color scheme, where pythons displays 1s as yellow and 0s as purple):

```
[[0 0 1 1 1 1 1 1 1 1]
 [0 0 0 1 0 1 1 0 0 1]
 [0 1 1 1 0 1 1 0 0 1]
 [0 0 0 0 0 1 0 0 0 1]
 [0 1 0 0 0 0 0 1 1 1]
 [0 1 1 1 0 1 0 0 0 1]
 [0 1 1 1 1 1 0 1 0 1]
 [0 0 0 1 0 0 0 1 0 1]
 [0 0 0 1 0 0 1 1 1 1]
 [0 0 1 1 1 1 1 1 1 1]]
```



To perform our Gerrymandering analysis, our goal will be to generate thousands of example districting plans, like the one above, that will act as an ensemble.

To do this sampling, you will employ Metropolis-Hastings with the Single Node Flip algorithm to sample the space of all possible districting plans. You will also explore the use of various energies imposed on the space to see a variety of different types of solutions.

The core idea behind the Single Node Flip algorithm as a generative sampling algorithm is to take a current plan and consider flipping a single border precinct to an adjacent district. In order to accomplish this task, it will be useful for us to track the set of boundary precincts in the current districting plan.

In addition to our matrix of 1s and 0s representing the `plan` itself, we will also track the `border`, represented as a list of coordinate pairs of the form (r, c), which would indicate that the precinct at row r and column c is on the border between our two districts.

As a final note before we begin working on the implementation of this algorithm, there are a few hard constraints that must be included: (1) each of the districts must be contiguous; and (2) the size of the districts must be roughly equal (more on this later). These constraints will become an important part of your implementation.

Part 2. **Helper Functions**

The actual implementation of Metropolis Sampling and the Single Node Flip algorithm can be rather involved, so there are a few helper functions we should set up first to prepare us to fully implement our solution.

You have been provided with a file `gerry.py` that includes some of these helper function that have been implemented for you. You are welcome to simply import this file and access these helper functions, or, if you would like a challenge, you can work to implement them yourself.

The first helper function we will need is a function that will check if a given district is contiguous (if one of the districts is not, the proposed plan must be automatically rejected). This check is implemented in the function `isContiguous(plan, val, N)`, which takes in the current `plan`, the `val` indicating which district is being examined (so `val` should be 1 or 0 for this project), and the size of the $N \times N$ plan, `N`. The function will return `True` if and only if the district in question is contiguous. Note that this can be done utilizing Depth-First Search, which is able to detect connected components of a graph. If we start at a precinct in our current district, and search the graph for all other precincts with the same label, DFS will visit all of the precincts in the district unless it is not contiguous. This provides us with a direct check for contiguity.

Another useful function to have is one that can check whether a specific precinct is on the border between the two districts. This can be useful when trying to update the border after a proposed node flip. The function `onBorder(row, col, plan, N)` performs precisely this check on the precinct located at `[row,col]`. The basic idea here: check the up, down, left, and right neighbors (if they exist) and flag as a border precinct if one of the neighbors if from a different district than the current precinct.

A third helper function to create is `countBorder(border, plan)`, which will return the number of 1s and the number of 0s in the current border (which can be done by looping over the border precincts and checking their district value in the current plan). This calculation is very useful when calculating the forward and reverse probabilities used by Metropolis: the odds that we selected a 1 to flip to a 0 is related to the number of 1s and 0s on the border!

You have been provided with a plotting function as well, which will be used to display your districting plans. You should not have to directly interact with this function, as the provided `main` block in the `hw7.py` file will handle all of the plotting for you. However, if you write your own version of the `gerry.py` file for a challenge, you should just copy-paste the provided `plotDistrict(plan, filename='''')` code.

Part 3. **Metropolis Sampling**

To perform the actual Metropolis Sampling, you will need to implement three specific functions in your `hw7.py` file (note: you should feel free to create additional helper functions if you want, but this is not required since you are writing code for this specific problem and do not need to worry about generalizing). The three functions signatures should be:

- `updateBorder(old_border, fliprow, flipcol, plan, N)`
- `calcQ(offset, borderOnes, borderZeros, offset_thresh)`
- ```
  metropolis(plan, offset, N, offset_thresh,
             border, borderOnes, borderZeros,
             measureFunc,
             maxIters=10, maxTime=-1,
             plotIters=-1, filename="", path="")
  ```

The `updateBorder` function: You should write a function to update the border given the old border and the indices of the flipped node. Note that when a node flips, it remains a border node, but all of its neighbors (up, down, left, right) may either become new border nodes, or become interior nodes (no longer on the border). Your function should return the new border list, along with the new count of 1s and 0s in the new border.

The `calcQ` function: As part of Metropolis Sampling, you must calculate the value of $Q(X \to Y)$, the transition probability from one state to another. For the Single Node Flip algorithm, this probability will simply be 1 divided by the number of border nodes, since the algorithm is to simply pick a border node to flip. There is one piece of added difficulty: we require that the two districts have roughly equal number of precincts. The parameter `offset_thresh` defines the allowable offset between the 1s and 0s: the difference between those quantities is not allowed to exceed that value. If the total difference 1s or 0s in the current plan is too close to that offset threshold, then certain moves will be disallowed. If, for example, 1s are disallowed, then Q should only include the border precincts that are currently 0 (because only those border nodes can be flipped!).

Note that this function takes an `offset` as input. This should be a signed quantity tracked by your Metropolis code: if positive, there are more 1s than 0s. Note that flipping a 1 to a 0 will increase this offset by 2, while the opposite flip will decrease it by 2. You will need to use this `offset` value along with `offset_thresh` to determine the number of allowable flips. The returned value `Q` should be the reciprocal of this number of *allowable flips*.

The `metropolis` function: You are now ready to create the full Metropolis implementation. This function is very involved, and you have been provided some starter code to get setup.

Let's start with a discussion of the various inputs, which come in 5 groups:

- Plan details: your function will take as input the current `plan` and `offset` (number of 1s - number of 0s), the size of the $N \times N$ plan, `N`, and the allowable offset threshold, `offset_thresh`.
  ```

- Border details: your function will take as input the current `border`, along with info about the current number of `borderOnes` and `borderZeros` (the number of 1s and 0s in the current border). These will be updated as the algorithm progresses.
- Measure details: your function will take as input a function `measureFunc`, which will take in a proposed `plan` and `border`, and return the value of μ corresponding to that plan. Specific energy functions will be discussed in detail later.
- Termination conditions: your function will take as input two values related to termination conditions of the Metropolis implementation: the maximum number of iterations, `maxIters`, and the maximum allowable computation time in seconds, `maxTime`. The provided code has already made use of these inputs to perform the sampling loop, so you should not need to interact with these variables. Note: the `maxTime` input will allow you to limit the computation time as you perform tests (though in my experience, this code runs reasonably fast and will not require use of this mechanism).
- Plotting parameters: your function will take as input three values related to producing and saving images of the districting plans as your algorithm runs. The variable `plotIters` will determine how many iterations to allow between plotting, while the `filename` and `path` specify where to save the generated image. Again, the provided code has already made use of these variables, so you should not need to interact with them.

The bulk of your work in this assignment will be to fill in the sampling loop in the `metropolis` function (marked with a `FIXME` statement in the provided code). Below is a general outline for each iteration that you should use as a guide when you write your function:

- First, you should randomly pick a border node to flip, and record its row and column.
- Now calculate the new offset value based on the flipped node and the previous value (note that flipping a node changes the offset by ± 2).
- If the new offset exceeds the allowable threshold, revert to the old plan and continue to the next iteration. Importantly: this does not count as an iteration, so you will need to decrement the variable `iter` before you `continue`.
- Now check if the new plan with the flipped node is contiguous. If not, revert everything, decrement `iter`, and continue.
- Calculate the updated border info. Note: if you have already made the flip, you may have to undo the flip for this calculation. It is important to keep track of whether you are interacting with the old plan or the new plan when performing these updates.
- Once you have access to both the old and new borders, you will be able to calculate the forward and reverse probabilities Q using your `calcQ` function. The forward probability will rely on border info from the old border, while the reverse probability will use the new border.
- Now you are ready to calculate the new value of μ, the measure for the new plan. Note: there already exists a variable `mu` which should have the measure for the old plan, so you will need to calculate a `new_mu` value with the new plan. Again, it is important to remember whether or not you have already made the node flip. You will likely have to make and un-make this flip multiple times during this process.

- Now that you have both the forward and reverse values of Q, as well as both the old and new values of μ, you are ready to calculate the Metropolis acceptance probability.
- Get a random value between 0 and 1, and accept the move with the probability you just calculated. If you accept the move, update everything in preparation for the next iteration (make sure the flip has been performed, update the `offset`, the `border`, the `borderOnes` and `borderZeros`, and the value of `mu`).
- The final step in the iteration is plotting, which has already been provided for you in the code.

I will again make an important note as you create your implementation: be aware of the current state of your plan. It is likely that you will flip the value of the node back and forth multiple times while calculating all of these various quantities, and it is critical that you know whether the node is flipped or not for each line of code you write.

An additional hint to help you: the following code snippet will flip/un-flip the node in question: `plan[fliprow, flipcol] = 1 - plan[fliprow, flipcol]`.

Part 4. **Energy Functions**

Now that you have your implementation of the Metropolis sampling ready, it is time to discuss the various measures that we might try to impose on our space of districting plans. Recall the we often times will define a measure in terms of an energy function we are trying to minimize: $J(\xi)$, where ξ is the current districting plan. Our measure then takes the form

$$\mu(\xi) = \frac{1}{Z} e^{-\beta J(\xi)}.$$

We can extend this idea to a set of energy scores, with $J(\xi)$ defined as the weighted sum of the different energies. The idea here is that each energy defines a quantity that we are interesting in minimizing, and so we attempt to minimize their weighted sum.

You have been provided with a file `energy.py` that has a number of different energies implemented for use with your Metropolis Sampling. Again, if you would like a challenge, you can try to implement some of these functions yourself.

The first function is `weightedMu(energyLst, plan, border)`. This function takes in a list of energies and weights (represented as `(weight, energy)` pairs), and returns the corresponding value of μ after computing their weighted sum, negating, and exponentiating.

The next provided function is `getHVmatrices(N)`, which will produce two $N \times N$ matrices defined as

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}, \quad V = H^T.$$

These two matrices will allow us to define a group of energies related to the symmetry/asymmetry of the current plan across the horizontal or vertical axis.

Recall that our plans are represented as $N \times N$ matrices of 1s and 0s. So we are going to define a series of energies by taking the sum of the pairwise product of matrices H or V with the given plan. Anywhere that H or V have a 1 will contribute $+1$ to the energy when they align with a 1 in the plan. Similarly, anywhere a -1 in H or V align with a 1 from the plan will contribute -1 to the energy.

This effect can then be used to define the following four energies:

- Horizontal Symmetry: This energy is computed as the absolute value of the sum of the pairwise product of the matrix H and the current plan. If the 1s in the plan are horizontally balanced, then the $+1$ and -1 contributions to the energy score will cancel. Meanwhile, if all of the 1s are on top (or bottom), then there will be entirely $+1$ (or -1) contributions to this energy. So minimizing this energy will result in a top/bottom balancing of the 1s in the plan, i.e., horizontal symmetry.
- Vertical Symmetry: In a similar fashion, minimizing the absolute value of the sum of the pairwise product of matrix V and the current plan will lead to a left/right balancing of the 1s in the plan, i.e., vertical symmetry.
- Horizontal Asymmetry: If we leave the sum of the pairwise product signed, we end up with a score that will optimize for asymmetry. The score will be minimized when the 1s of the plan align with the -1s of the matrix H. So using this energy with a postive weight will prioritize placing the 1s of the plan in the bottom half of the domain. A negative weight will prioritize placing them in the top of the domain.
- Vertical Asymmetry: In a similar fashion, this energy can prioritize the 1s to appear on the left or right (with a negative or positive weight respectively).

In addition to these symmetry/asymmetry energies, there is one more critical energy to include: **compactness**. Compactness is a measure of how compact a shape is, how close it is to a "blob" instead of a "snake" shape. The typical score used for this measure is called the isoperimetric ratio: the ratio of the squared perimeter to the area of the shape. This quantity is minimized when the perimeter is "as short as possible" compared to the area, i.e., when the shape is a circle (the ideal "blob").

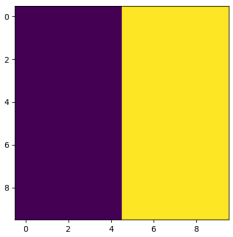
This energy has been implemented for you in the function `isoperimetric(plan, N, border)`, which computes the average compactness for the two districts in the plan. To implement this function, we will say that each precinct is a unit square, with each side of length 1, and an area of 1. The area of a district is then just the number of precincts in that district. The perimeter is a more difficult calculation. For each of the internal border nodes (the nodes that separate the two districts), the contribution to the perimeter will equal the number of this node's neighbors that are from the other district. Meanwhile, we must also include the perimeter of the domain itself: any node on the boundary of the domain will add a perimeter of 1 to the district, with the four corner precincts adding a perimeter of 2.

Now that we have defined the specific energies to consider in this assignment, you could, if you would like, try to define your own energies using these ideas. In particular, note that you can use the trick with matrices H and V to instead prioritize different shapes. If you define an $N \times N$ matrix, you can fill it with various values that will define how preferred it is to have a 1 in that precinct: the more negative the value in your matrix, the higher the preference for a 1. You could use this idea to draw some interesting shapes to sample towards.

Part 5. **Putting it all together**

Now that all of the code is written, you are ready to run some tests. You have been provided with testing code in the `main` block of the `hw7.py` file. In this block, you are able to set the various parameters and run your own tests, though I would encourage you to mainly rely on the provided parameters (though you might want to decrease the number of iterations or plotting iterations for debugging purposes).

The provided code also sets up an initial plan to use for all of your testing. This plan starts will all 0s on the left and all 1s on the right. This will make it easy to see the result of our various energies on the sampling: if you prioritize 1s on the left, you should see your algorithm attempt to "flip" the image and swap the sides. Below is the image of the initial plan:



If you set the `filename` variable to a few specific options, you will enable specific provided scenarios for testing:

- "uniform.png" will run a scenario with uniform sampling, where every districting plan is equally likely. This should result in mostly "snake-like" plans, since the vast majority of districting plans have that property.
- "left-asym.png" will prioritize placing 1s (yellow) on the left. The initial plan has all 1s on the right, so this is a good test for mixing - how long does it take to see 1s on the left? Your code should typically make it there after 10,000 iterations.
- "compact.png" will run a scenario with only the compactness energy turned on. This will prioritize "blob-like" districts. You should not see any of the "snakes" show up here.
- "top_left.png" will run a scenario that prioritizes placing 1s (yellow) in the top left corner of the domain. To help with this sampling, we have also turned on the compactness score. Otherwise, the "snake-like" results will tend to struggle to actually achieve the "top-left" priority.

Once you have verified that your sampling is working correctly using these various tests, you should submit your code to the course gradescope.