CS 115 Homework 6: Navigating a City Grid... But Loopy!

The goal of this homework is to compute shortest paths in a city grid in a different way.

- We will calculate the distance from every square to the start rather than just one.
- Instead of traversing the grid with recursion, we will use **loops**
- We will calculate the path as well as the cost.
- We will reuse previous computation in a different way.

City grid representation

After a reminder of the city grid representation overview from Homework 3, this section describes *how cities are stored in files*.

We represent a city as a collection of blocks arranged in a grid with width m and height n.

The blocks are numbered from $(0,0)$ in the top left to $(m-1, n-1)$ in the bottom right.

$(0,0)$	$(1,0)$	$(2,0)$	$(3,0)$
$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$
$(0,2)$	$(1,2)$	$(2,2)$	$(3,2)$

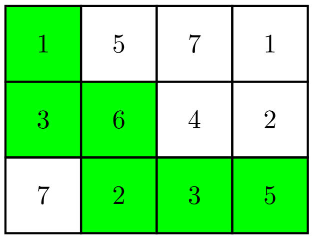
Our goal is to get from some block at (x, y) to the block at $(0,0)$, while only being able to travel **up and to the left**.

To reflect the fact that different blocks are harder to cross than others, (e.g. obstructions, topography, traffic), each block has a cost corresponding to the amount of time it takes to cross it.

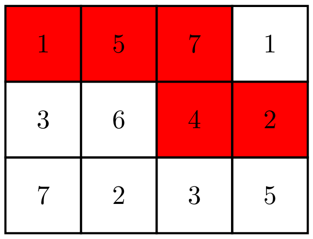
1	5	7	1
3	6	4	2
7	2	3	5

When we start at a block (x, y), we take time equal to its cost to cross it. Then, whichever block we go into costs us time equal to its cost. This continues until we reach block $(0,0)$, at which point we incur its cost as well.

For example, the following are two paths from the above grid, with their costs listed below. The paths start at different coordinates, but both end at $(0,0)$.

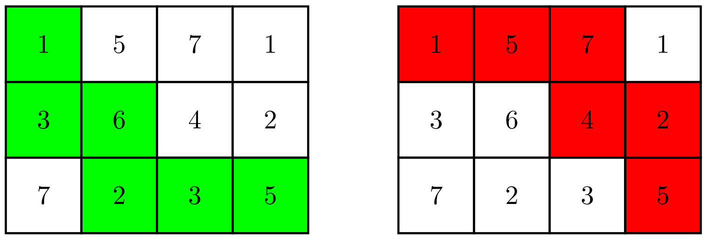


$5+3+2+6+3+1=20$



$2+4+7+5+1=19$

Note, however, that there is more than one way to get to $(0, 0)$ from most coordinates. And these paths do not have equal costs!



$5 + 3 + 2 + 6 + 3 + 1 = 20$ $\quad$ $5 + 2 + 4 + 7 + 5 + 1 = 24$

Representing Grid in Memory

Like the previous homework, we represent a grid as a list of lists (2D list).
A `grid` with width m and height n has a `list` of n rows where each row is a `list` of m numbers.
The example given above would, for example, be represented by the following list:

```
[[1,5,7,1],[3,6,4,2],[7,2,3,5]]
```

Our grids are rectangular, so all rows of a grid have the same length.

The Grid File Format

Instead of storing the grids in code, we will load city grids from a file. This is the file format we use, which is stored as text.

1. A line consisting of m and n : two positive integers separated by a space.
2. After the above, n lines with m positive integers follow. Each line represents a row.

For example, the previously given grid would be represented in a file as follows:

```
4 3
1 5 7 1
3 6 4 2
7 2 3 5
```

Task 1: Reading Grids

Your first task is to load the grid files. Implement the function `parse_file`, which takes in the name of a single file and returns the grid stored in that file as the list of lists as specified in Section *Representing the Grid*.

If the file is invalid, then your code should throw a `ValueError` describing what was wrong in the error message (how is up to you). Example ways a grid is invalid that we expect your code to detect:

- Not listing the dimensions at the top of the file, or having too many/few numbers
- The number of items in a row is smaller than m
- There aren't n rows following the first line

To load them, be careful about how you store files! Finding files is one of the hardest parts of File IO.

1. Make a folder called `grids` **in the same folder as your** `hw6.py` **and** `test_hw6.py`
2. Download the grid txt files from Canvas
3. **Move them all into the grids folder**
4. **For example, if your** `hw6.py` **is in your** `homework6` **folder, the** `grids` **folder should** ***also*** **be in your** `homework6` **folder. The** ***grid txt files*** **should be in the** `grids` **folder.**

Task 2: Computing Shortest Distances (Redux)

Your next task is to implement the `distances_from` function. This takes in a `grid` of costs, and outputs a 2D list of shortest distances from each block (x, y) in the grid.

We can do this bottom-up using for loops. Start from the initial point of $(0, 0)$, and then calculate the distances moving outwards.

A key observation that makes this problem easier is that if you go through the coordinates in ascending order, then by the time you get to point (x, y), you have already calculated the distances from $(x-1, y)$ and $(x, y-1)$.[1]

This thus gives us our algorithm to generate our array of distances:

- Create a 2D array `dists` which we will gradually fill out.
- Fill in the first row and column of `dists`.
- Loop over every remaining coordinate (x, y), and for each:
  - Find the minimum of the distances from $(x - 1, y)$ and $(x, y - 1)$, by looking at earlier quantities in `dists`.
  - That minimum plus the cost to traverse (x, y) is then exactly the distance from (x, y), letting us fill in that coordinate of `dists`.

An Example Run of Our Algorithm

For example, let's try to calculate all the distances for the earlier example. We begin by noting that the distance from $(0, 0)$ is just the cost of traversing $(0, 0)$.

Grid:

1	5	7	1
3	6	4	2
7	2	3	5

Distances:

1			

Now, we can try to calculate the distance from $(1, 0)$. This is the cost of traversing the cell $(1, 0)$, followed by the cost of traversing the cell $(0, 0)$.

Grid:

1	5	7	1
3	6	4	2
7	2	3	5

Distances:

1	6		

But what is interesting is the distance from $(2, 0)$. See, the shortest path from $(2, 0)$ involves having to cross the cell $(2, 0)$, and then taking the shortest path from $(1, 0)$... but we just worked out the distance from $(1, 0)$. After all, we have already filled out that value of our distances grid. So we can find the distance from $(2, 0)$ just by adding those two values.

[1] This is a technique known as dynamic programming. While we explain above how to apply it for this homework, if you wish to learn more about it, you are welcome to look it up.

Grid:

1	5	7	1
3	6	4	2
7	2	3	5

Distances:

1	6	13	

We can continue this pattern to fill out the entire first row and first column. At each step, calculating the next entry just requires knowing the previous one. So if we ensure that we do all our work while moving away from $(0, 0)$, all the relevant information needed to work out each cell is already computed and stored in the array by the time we get to it.

Grid:

1	5	7	1
3	6	4	2
7	2	3	5

Distances:

1	6	13	14
4			
11			

Now, let us consider the cell $(1, 1)$. The shortest path here involves either going to $(1, 0)$ or to $(0, 1)$. But due to our ordering, we have already pre-computed the distances from those two points! So we can reuse that earlier work, which has already been stored in the array.

Grid:

1	5	7	1
3	6	4	2
7	2	3	5

Distances:

1	6	13	14
4	10		
11			

Then, calculating the distance from cell $(2, 1)$ can rely on the distances from $(2, 0)$ and $(1, 1)$. And this can be continued to fill out the entire $(x, 1)$ row.

Grid:

1	5	7	1
3	6	4	2
7	2	3	5

Distances:

1	6	13	14
4	10	14	16
11			

And then this can be repeated for the $(x, 2)$ row, taking advantage of the fact that we've computed the $(x, 1)$ row. And then the $(x, 3)$ row, and so on. At each step, we only have to consider the immediate neighbours, which we'd already computed earlier due to the order we went over our cells. The result is that we can efficiently calculate the entire array of distances without ever having to waste effort recomputing earlier results.

Grid:

1	5	7	1
3	6	4	2
7	2	3	5

Distances:

1	6	13	14
4	10	14	16
11	12	15	20

Task 3: Finding the Shortest Path

In this next task, you will compute the list of points on the shortest path from the start to any point (x, y). Implement `shortest_path`, which list of lists of shortest distances for a grid and a point (x, y), returns a list of points from the start to that point.

Here is a algorithm for finding the shortest path for a point with only the shortest distances.

1. Initialize the path as an empty list
2. Start at point (x, y)
3. At each iteration (use a while loop)
   - Add the current point to the path
   - Check if the current point is the start, if so, exit the loop
   - Move the current point to the point above or to the left that minimizes the distance to the start.

For example, consider the earlier example. The shortest path has been highlighted below.

Grid:

1	5	7	1
3	**6**	4	2
7	**2**	**3**	**5**

Distances:

1	6	13	14
4	**10**	14	16
11	**12**	**15**	**20**

If we call `shortest_path` with this grid and array of distances, we would want it to return the following list.

```
[(3,2), (2,2), (1,2), (1,1), (0,1), (0,0)]
```

In other words, this returns a list containing tuples representing the points that were passed through, starting at the given point and ending at $(0, 0)$.

Task 4: Labor log

Please complete the labor log at the end of `hw6_report.txt`. Some prompting questions are on the `.txt` file. Only two sentences are required, but we are happy to read anything you have to share about how this homework went.

Submission Instructions

For this homework, you must submit the following files, with exactly the following names:

- `hw6.py`
- `hw6_report.txt`

Rubric

Category	**Description**	**Points**
Name and pledge	For including your name and the full Stevens Honor Pledge on every submitted file.	5
Task 1: `parse_file`	Can correctly parse correctly formatted grid files.	10
Task 1: `parse_file`	Throws ValueError exceptions for listed invalid grid files.	20
Task 2: `distances_from`	Efficiently and correctly computes shortest distances from all points using loops.	30
Task 3: `shortest_path`	Computes shortest paths, correct implementation.	30
Task 4: Labor log	Completes labor log.	5
Total		100

Notes and Reminders

- Keep track of the work you put into this homework, **especially in how you overcame difficulties**. Share in the "labor log" of the report.
- **Remember to include docstrings!** If your submission does not have docstrings, or has docstrings that are not related to your code, you will not get credit for them.
- Additionally, if your functions are more complicated than usual, you should add comments inside them explaining how they work.
- We have provided a test file, `test_hw6.py` for you to test your solutions to ensure that files are named correctly and to check the correctness of your code. Try out many different cases to verify that your code runs. Our provided test file may not be comprehensive, and so you should ensure to test your code on many other cases as well.
- Code that doesn't compile (throws a `SyntaxError` on being run) will receive a 0 for the assignment! Make sure you submit code that can be run, even if it does not pass all the test cases.
- **Don't forget to add your name and pledge!**