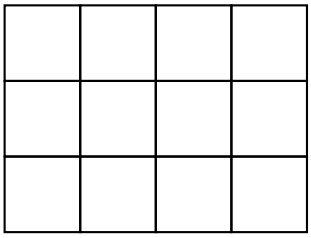
# CS 115 Homework 3: Navigating a City Grid

Navigation from a starting location to a goal location is one of the most foundational capabilities for agents that operate in our world. In addition to computing directions for ourselves to get from point A to point B, autonomous navigation is critical for applications such as self-driving cars, delivery robots, and any other robot that needs to move!

In this lab, we will use branching recursion to implement an algorithm for **computing the shortest path** between a start location and target location.

In general, the shortest path problem is about finding the "lowest cost" path, which is usually in terms of the most precious resource: time.

To keep this homework simple, we will model the city as a collection of blocks, arranged in a grid.



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

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

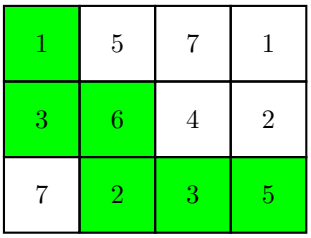
Our goal will be to get from some block at $(x, y)$ to the block at $(0, 0)$, while only being able to travel **up and to the left**.

To reflect the fact that not all blocks are equally easy to cross, (e.g. obstructions, topography, traffic), each block has a value corresponding to the amount of time it takes to cross it.

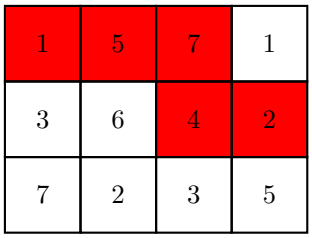
| | | | |
|---|---|---|---|
| 1 | 5 | 7 | 1 |
| 3 | 6 | 4 | 2 |
| 7 | 2 | 3 | 5 |

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 $(0, 0)$.

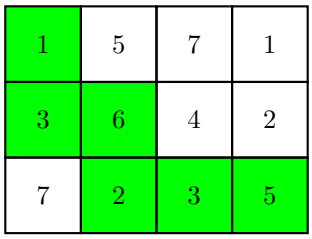



$5+3+2+6+3+1=20$

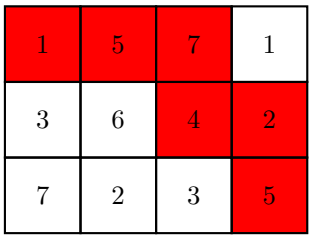



$2+4+7+5+1=19$

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!




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




$5+2+4+7+5+1=24$

The goal of this homework is primarily to develop our skills in designing recursive algorithms while also reinforcing skills in working with lists. To guide us, we solve the following problem: Given the grid specification above and location $(x, y)$, find the cost of the cheapest (the quickest: if cost is time) path to get to $(0, 0)$.

Once we have an algorithm to compute the cost of the cheapest path, returning the actual path is relatively straightforward.

## Grid Implementation Details

We begin by first talking about how we want to represent our data. In this homework, we will represent the grids mentioned above as 2D arrays: lists of lists. In other words, we will have a `list` of $m$ rows, where each row is itself a `list` of $n$ blocks.

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]]
```

For the purposes of this homework, we will assume that our grid is always rectangular (so all the rows of a given grid have the same length), and you do not need to consider non-rectangular grids (especially because paths become ill-defined in that case). In other words, your code is allowed to break when given non-rectangular grids.

We also note that grid are mutable. They are lists of lists, and we want the ability to modify individual blocks. This means that you have to keep careful track of pointers to avoid situations where you don't properly edit a list.

## Task 1: Getting Started with Grids

Your first challenge is to just have basic grids we can work with.

First, you should implement the `copy` method. As we are working with mutable lists, we sometimes want to be able to duplicate a grid to keep the original unchanged. The `copy` method should create a deep copy of the grid: If I run `new_grid = copy(grid)`, I should not be able to affect either `grid` or `new_grid` by mutating the other.

Then, you should implement the `empty` method. This takes in two integers `m` and `n`, and should return an empty grid with `m` rows and `n` columns. This must be a valid grid, in the proper format, and it must be initialized with all blocks initially set to 0. Note that the rows should be fully independent: I should not be able to edit one row by mutating another.

## Task 2: Mass-Mutating the Grids

Once we have the ability to make empty grids, we now wish to fill them with numbers. Changing one coordinate should be easy, using our standard list mutations. But we also want to be able to update entire rows or columns at once.

For this, your next goal is to implement the `increase_row` and `increase_col` methods. These methods must mutate a given grid by increasing the cost of all cells in a given row or column by a given amount. More specifically, `increase_row` takes in a grid, a coordinate `y` and a value `cost`, and it should increase all the blocks in row `y` by `cost`.

The following are some example calls of these methods.

```
>>> A = [[0,0,1], [0,1,2], [1,2,3]]
>>> increase_row(A, 1, 10)
>>> A
[[0,0,1], [10,11,12], [1,2,3]]
>>> increase_col(A, 2, 100)
>>> A
[[0,0,101], [10,11,112], [1,2,103]]
```

These are mutating methods, and they do not return anything, only modifying the given grid.

## Task 3: Finding the Fastest Path

Finally, we reach the main goal of this homework. You have to implement the `distance_from` function. This takes in a grid, and two coordinates `x` and `y`. And your goal is to output the length of the shortest path from $(x, y)$ to $(0, 0)$, as per the rules laid out in the earlier sections.

This method needs to run in a reasonable timeframe on large inputs, and must return the total distance. Some example inputs are listed below.

```
>>> grid = [[1,5,3], [9,4,7], [2,8,6]]
>>> distance_from(grid, 2, 2)
22
>>> distance_from(grid, 2, 0)
12
```

The first example takes advantage of the following shortest path, while the second example has only one possible path to begin with, which is the shortest path by necessity.

| | | |
|---|---|---|
| 1 | 5 | 3 |
| 9 | 4 | 7 |
| 2 | 8 | 6 |

$6 + 7 + 3 + 5 + 1 = 22$

| | | |
|---|---|---|
| 1 | 5 | 3 |
| 9 | 4 | 7 |
| 2 | 8 | 6 |

$3 + 5 + 1 = 9$

## Labor log

For this homework and each going forward, we would like you to share your software development **process**. Especially, we want you to write about how you **debugged** your code, which is one of the most important parts of creating code.

In your labor log in `hw3_report.txt`, write about:

**Required components**:

1. How did you locate and correct any mistakes in your code? What strategies did you use? (At least 2 sentences)

2. Which strategies for correcting your code were the most and least effective? (At least 2 sentences)

In particular, we are looking at how you model what you *expect* your code to do and how you observe what your code *actually* does.

In addition to the required reflection, we also suggest logging other important aspects of software development. This is not graded, but we hope it helps you raise awareness of your engineering process.

**Recommended optional components**:

1. Timing: when you started the homework, how many hours you worked on it
2. Environment: where you worked on the homework, with whom, ...
3. What you will continue and/or do differently for next homework
4. What you're proudest of?

## Submission Instructions

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

- `hw3.py`
- `hw3_report.txt`

**Remember to put your name and the Stevens Honor Pledge on every file you submit!** Failure to put the pledge can result in marks being deducted, with repeat penalties potentially getting stricter between assignments.

Submissions must be handled through Gradescope (accessible through the homework page on Canvas). Grades will be released through Gradescope once the homework is graded.

## Rubric

| Category | Description | Points |
|---|---|---|
| Task 1 | For properly implementing `empty` and `copy`. | 15 |
| Task 2 | For correctly implementing `increase_row` and `increase_col`. | 20 |
| Task 3 | For correctly implementing `distance_from`. | 40 |
| Labor log | For answering the required questions about debugging code in the `hw3_report.txt`. | 15 |
| Docstrings | For correctly written and formatted docstrings. | 5 |
| Name | For stating your name on every file for submission. | 1 |
| Collaboration statement | For listing your collaborators or stating None. | 2 |
| Pledge | For writing the full Stevens Honor Pledge on every file for submission. | 2 |
| **Total** | | 100 |

## Hints and Suggestions

As this problem set is quite large, we provide several hints to help you get started and to help you debug your code.

- While conceptually difficult, a lot of these functions are a lot simpler than they first appear. In particular, the first three functions all have single line solutions. While yours do not have to be that simple, if you find your solution starting to get longer than about 10 lines, you may wish to try a different approach.

- We actually recommend tackling `copy` before `empty`. It turns out that `copy` can be quite useful for implementing `empty`.
- Of particular use in this homework is list multiplication. If I do something like `res = [expr] * k`, this is equivalent to the following code:

```
a = expr
res = [a, a, a, a, a, ..., a]
```

  where there are `k` copies of `a` in the list. This can be quite useful for building large lists with repeated elements, but be careful! The above code expansion is quite deliberate, as it copies the same value each time. This might mean unexpected behaviour when `expr` is not a primitive value, but something that is stored as a pointer.
- Keep careful track of which functions mutate and which return a modified copy.
- Remember your higher-order functions. These can be quite handy for several of these problems (although not all).
- `increase_row` is a bit easier than `increase_col`, and is definitely the one I recommend starting from.
- `distance_from` is definitely the most tricky problem in the homework, so you'll want to spend a lot of time figuring things out. In particular, how you can express the shortest distance to a given coordinate using the shortest distance to other points is a challenge we leave to you to figure out.

## 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_hw3.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.
- **Until we tell you otherwise, you are not allowed to use loops!** We want you to learn to use recursion, so you are not allowed to use loops to solve these problems.
- **Don't forget to add your name and pledge!**