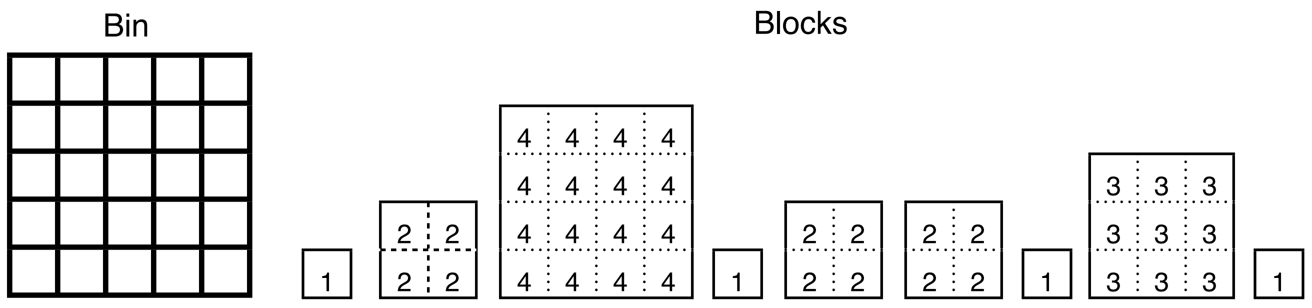
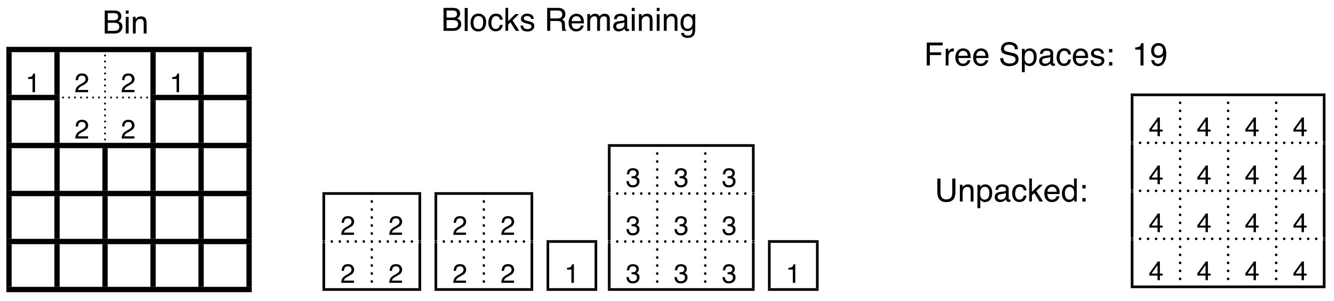
1. Problem Statement

Imagine that you are trying to pack luggage into the trunk of a car. The luggage comes in various sizes and your goal is to completely fill the trunk, if possible. In computer science, this is called the **bin packing problem**. The trunk is the bin, and the luggage are the objects that are being packed into the bin.

We will be looking at a variation of this problem called **two dimensional bin packing**. Our bin will be of some square dimension (e.g.. 5x5), and we will be packing square blocks of various dimensions (e.g. 4x4, 3x3, 2x2, and 1x1) into the bin, in the order they are presented.



We will use a greedy strategy called **first fit** to pack the blocks into the bin. We will take the items in the order they are given and place them into the bin starting at the first row, first column. If a block cannot be packed into the bin it will remain unpacked. Each attempt at placement begins trying to place the block in the lowest-numbered row that contains free spaces. For example, here is the state of the bin after the first 4 blocks are placed.



2. Problem Solving Session (20%)

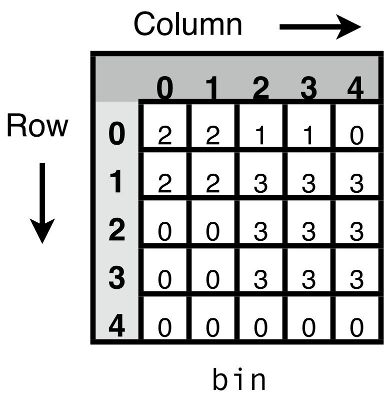
You will be working in a team of four to six students as determined by your instructor. Each team will work together to complete the following activities.

For questions 1-4, use the example above. For each question, show the following:

- **The final state of the bin.**
- **The number of free spaces.**
- **The list of the unpacked blocks.**

1. Finishing the bin packing started with the example above. Answer with the three bulleted items.
2. Another approach we could use is called **first fit ascending**. Here we place the blocks into the bin based on size from smallest to largest. Answer with the three bulleted items.
3. Yet another approach we could use is called **first fit descending**. Here we place the blocks into the bin from largest to smallest. Answer with the three bulleted items.
4. Find an order of block placement that yields the optimal solution (i.e. no free space). Your answer must include the *sequence of blocks placed*, plus the three bulleted items.

How will we represent the bin? It needs to have two dimensions, the first is the row and the second is the column; this is row-major order. Here is a visualization which shows how the row and column indices are used to access elements in the bin.



	0	1	2	3	4
0	2	2	1	1	0
1	2	2	3	3	3
2	0	0	3	3	3
3	0	0	3	3	3
4	0	0	0	0	0

```
bin[ 0 ][ 0 ] == 2
bin[ 0 ][ 2 ] == 1
bin[ 2 ][ 3 ] == 3
bin[ 4 ][ 1 ] == 0
```

Notice the values we are using in our bins. If an element contains a `0`, it means it is a free space; otherwise it contains a block whose number indicates its dimension.

5. Write pseudocode for a boolean function, `isSpaceFree`. It takes the bin (a 2-D list as described above), a row index, a column index, and a block size. It returns whether a block can be packed into the bin starting at the upper left coordinate specified by (row, column). **It should leave the bin unchanged**. For example, using the bin above:

```
>>> isSpaceFree(bin, 2, 0, 2)
True
>>> isSpaceFree(bin, 3, 1, 2)
False
>>> isSpaceFree(bin, 4, 3, 2)
False
```

3. Implementation (80%)

List Comprehensions

How will we represent the two dimensional bin in Python? A regular list in python is only one dimensional. We will need to make a **list of lists** in order to accommodate the second dimension.

One way to do this is to start with an empty list, and then append new lists repeatedly into it.

```
>>> bin = []
>>> for row in range(5):
...     bin.append([])
...     for col in range(5):
...             bin[row].append(0)
...
>>> bin
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0,
0, 0]]
```

This can be simplified into a single statement using a **list comprehension**.

```
>>> bin = [ [0 for col in range(5) ] for row in range(5) ]
>>> bin
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0,
0, 0]]
```

Now that we have a 2-D list, we can access any element in it by indexing into it with the row and column.

```
>>> for row in range(5):
...     for col in range(5):
...             print(bin[row][col], end=" ")
...     print()
...
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

To access or change an individual element in the bin, index to it with the row and column.

```
>>> bin[0][0] = 1
>>> bin[0][1] = 2
>>> bin[0][2] = 2
>>> bin[1][1] = 2
>>> bin[1][2] = 2
>>> bin
[[1, 2, 2, 0, 0], [0, 2, 2, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0,
0, 0]]
```

Program Details

You will write a program, `pack.py`, which will implement the **first fit descending** algorithm (question 3 from problem solving).

Your program will prompt for two inputs from the user. The first is the **square dimension of the bin (an integer)**. This tells you how large to make your bin. The second is the **name of the block file (a string)**. The file will contain a single line of integers representing the block sizes, separated by space/s. Here is a sample file, `blocks.txt`:

```
1 2 4 1 2 2 1 3 1
```

When run, the program should show the **final state of the bin**, the **number of free spaces**, and the **list of unpacked blocks**. Here is a sample run:

```
$ python3 pack.py
Enter square bin dimension: 5
Block file: blocks.txt
4 4 4 4 1
4 4 4 4 1
4 4 4 4 1
4 4 4 4 1
0 0 0 0 0
Free Spaces: 5
Unpacked blocks:  [3, 2, 2, 2]
```

To make displaying the bin easier, we guarantee that the block sizes will range between `1` and `9` inclusively.

In order to do first fit descending you will need to sort the blocks in descending order first. Assuming the blocks are in a list of integers called `blockList`, the way to do this in Python 3 is:

```
blockList = sorted(blockList, reverse=True)
```

Testing

Assume we are packing the following blocks into a 7x7 bin:

Block size	Count
6x6	1
5x5	1
4x4	1
3x3	3
2x2	2
1x1	8

In addition to your program, you will submit two test files:

- `blocks1.txt` contains a *subset* of the blocks listed in the table. When supplied with this test set, your program should produce a filled bin with no free space. You may remove any of the blocks for your submission.
- `blocks2.txt` contains the *entire* set of the blocks listed in the table. When supplied with this test set, your program should produce a partially filled bin, a non-optimal result.

5. Grading (100 points)

Problem Solving (20 points): As determined by your instructor from problem solving work.

Implementation (80 points): As determined by your SLI from dropbox submission.

Functionality (70 points):
- (5 points) Program prompts for dimension and block file.
- (5 points) Program reads block file into a list of integers and sorts them.
- (60 points) Program produces correct output using the first fit descending algorithm.
  - (48 points) Final bin state.
  - (6 points) Free spaces.
  - (6 points) Unpacked block list.

Code Style (5 points):
- Comment block at head of file with name and program description.
- All functions docstring'd according to the coding standard.

Test Files (5 points):
- Test files produce optimal and non-optimal results according to the writeup.

6. Submission

Zip your program, `pack.py`, and two test files, `blocks1.txt` and `blocks2.txt` to a zip file named `pack.zip`. Submit the zip file to the **myCourses dropbox** for this lab.