Task 1, packing

Congratulations, you've found the house of your dreams. The only problem is that you need to move out of your old house. You've got a lot of stuff, it's all in boxes, and the movers are on their way. You're not sure how many truckloads they'll need to move your stuff, and how long it'll take. The fewer truckloads, and the shorter time, the cheaper it'll be for you.

The first problem we're going to consider involves packing the truck with your boxes. We'll start with a very simple approach to doing so, and build up the complexity of the problem as we go along.

Input format

The problem we're trying to solve can (broadly) be specified as follows.
Given a set of N boxes of size x_i,y_i,z_i (where $0 \leq i < N$), and an area of size X,Y,Z, maximise the number of

boxes which can be placed within the area with no overlap.
Input to our program will therefore take the form of a number of lines, with elements separated by spaces, as

follows.

XYZ

N

x0 y0 z0

x1 y1 z1 ...

xN−1 yN−1 zN−1
Where X, Y, N, x_i, y_i and z_i are all integers.

You can use the program packing example.py to generate inputs for testing. Output format

As output, you'll print out the index of the item, and the location of its bottom left corner. If an item can't be placed, print -1 for its x,y and z coordinates. For example we might have the following 5 item in the vehicle.

0000

1500

2060

3 -1 -1 -1

4550

Here, item 3 could not be placed.

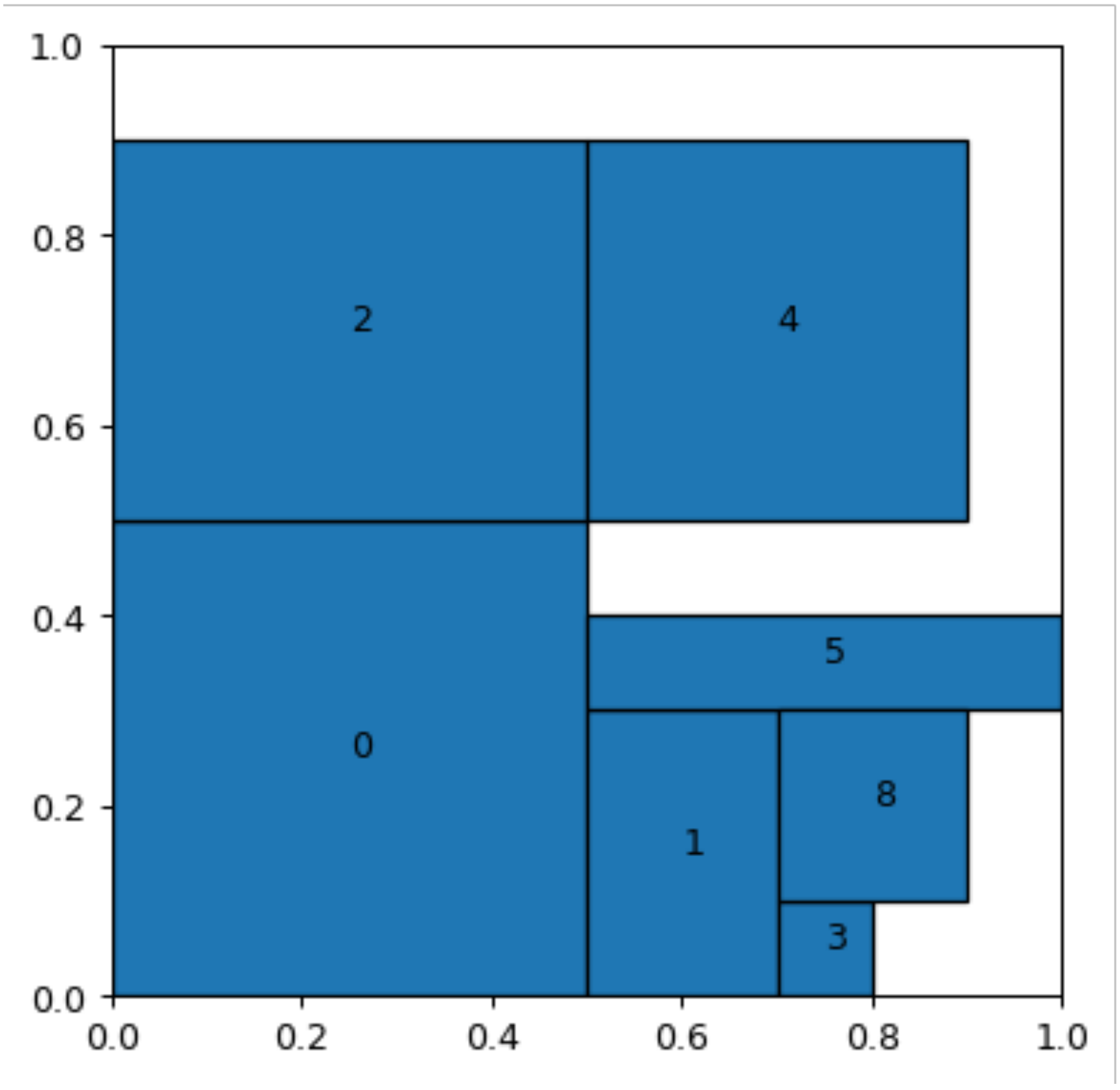
2-D packing

For a start, let’s assume that we can’t stack things on top of each other. Let’s also assume that we can’t rotate boxes as we receive them. We will therefore begin by ignoring the Z and z_i elements of our input file (which will always be 0). The basic approach we’ll take is very simple. We’ll place an object in the bottom left corner of our area. We’ll then subdivide our area into two, namely an area to the right of our object, and an area above it. We’ll then try fit the next item to the right of our object (not allowing it into the area above it). If it doesn’t fit there, we’ll try fit it in the area above. We’ll repeat this for every object that fits. This is graphically shown in Figure 1. We will always try place

an object to the right of placed objects before trying to place it above (note that the area above item 4 as 0 space). Note that you should place items in the order in which they appear in the file, i.e., item 0, then item 1, until item

N – 1.

2



Task 1.1 Implement this algorithm with input and output as specified above.
An example of the input and output you may receive and should produce is as follows.

Sample Input

```
10 10 10 10
552 232 544 112 445 511 532 332 222 543
```

Sample Output

```
0000 1500 2050 3700 4550 5530
6 -1 -1 -1 7 -1 -1 -1 8710

9 -1 -1 -1
```

Task 1.2 Implement the following simple variations of the basic algorithm.

- 1.2.1 Sort the items by their x coordinate and attempt to insert the largest (in terms of x) items first. In case of a tie, a box with a lower index should be considered first. Your output should still index the first box received in the list as box 0, the second as box 1, etc. For the sample input above, your output should be as follows.
  ```
  0000 1850 2500 3280
  4 -1 -1 -1 5540 6050 7550 8080
  9 -1 -1 -1
  ```

- 1.2.2 Sort the items by their y coordinate and attempt to insert the largest (in terms of y) items first.

- 1.2.3 Sort the items by their area. Attempt to insert the item into the area with the smallest remaining available area first. In other words, while you will partition the space as you previously did, rather than placing ”right” then ”above”, try place items by considering smallest to largest available areas instead. For the sample input, you should get the following output.
  ```
  0000
  1 -1 -1 -1 2500 3950 4550 5540
  6 -1 -1 -1 7 -1 -1 -1 8 -1 -1 -1 9050
  ```

- 1.2.4 Assume you are now allowed to rotate an item before placing it, and try to rotate an item so that—if possible— it is placed into the location with the smallest remaining area first. You should sort items by area before starting the process.

Task 1.3 Empirically determine which of the approaches is most efficient in terms of fitting the most items into a space, in terms of how much space is left over for each approach, and in terms of runtime. Explain how you came up with your results. Hint: Think about how an evaluation should take place. Consider running multiple instances of the problem with different sizes of inputs and see how that affects efficiency and runtime. A good analysis would include some sort of statistical analysis to support confidence in your results.

Task 1.4 Extend the approaches in Task 1.2 to a 3D volume (prioritizing the vertical coordinate last when necessary). Don't worry about gravity, items can float in midair if needed. Carry out the same evaluation as in Task 1.3 for these approaches. If you need to prioritize between different dimensions, prioritize x over y over z.

Task 2, Unpacking

You've now reached your new home, and we're going to switch perspective to the moving company, who wants to unpack the truck as quickly as possible. The moving company has devised an ingenious system for unpacking their trucks. They form multiple human chains, from the truck to the home, and pass items along the chain. Of course, some people pass items faster than others, and to overcome blockages, they allow people to pass items across chains. Figure 3 illustrates one such scenario, consisting of 2 lines. We can see that if an item follows the bottom line, it'll take a total of 10 units of time to get to the house (1+4+3+2), while following the top line means it'll take 13 time units to do so. However, starting along the bottom, switching to the top line and then switching back to the bottom at the last person takes only 9 units of time.

The problem we face is therefore finding the fastest way of moving an item from the truck to home.

Input Format

Our input file format is as follows

NS
<pos> <pos> <time>

Here, N and S are the number of human chains present, and the number of people in the human chain.

There then follow a large number of triples. Each <pos> element identifies a chain and position in the chain, with time describing the time to transfer an item, or to handle the item themselves (in case the two position elements are identical). The example of Figure 3 could thus be represented as the file

24 00003 00112 01012 01123 02022 02131

6

End (house)

03036 10101 10011 11114 11022 12123 12032 13132

You can use the program unpacking example.py to generate sample input for testing. Output Format

For Tasks 2.1-2.3, your program will output a single number. Your code will be run as

```
python3 unpacking_example.py | python3 unpacking.py
```

Task 2.1 The problem can clearly be represented as one of finding the shortest path through the graph. Implement Dijkstra's algorithm and print out the minimum time needed for an item to move from the truck to the house. The output for this task is a single number printed on its own line.

Task 2.2 We can also view this as a recursive problem. In the example, the shortest time taken to get to the house is the minimum time taken to get to the end of the chain; the shortest time to get to the end of the top chain is the minimum time taken to get to the previous person in the top chain, or the minimum time taken to get to the 3rd person in the bottom chain plus the transfer time, and so on. Implement a recursive algorithm which follows this logic to print out the minimum time taken for an item to move from the truck to the house. The output for this task is a single number printed out on its own line.

Task 2.3 Implement a dynamic programming based approach for the recursive algorithm of Task 2.2. The output for this task is a single number printed out on its own line.

Task 2.4 In your report, provide a *theoretical* analysis of the worst-case runtime of each approach, as well as its worst-case space complexity (i.e., describe worst-case runtime in O-notation and prove why these results come about).

Task 2.5 Provide an empirical evaluation of each of the approaches, plotting the time taken for different numbers of chains and different chain lengths. Use this to identify average case runtime complexity, and justify your analysis (c.f., Task 1.3).

Task 2.6 Now consider what would happen if you have multiple items to transfer. If the first item goes along the bottom chain, then the person will be busy for a while which ”blocks” that chain. One way to adapt the approach we used previously is to add times to the system as items pass through it to reflect this blockage. Given a list of times at which items are offloaded:

```
<input as described above>
N
t1
t2
... tn
```