# CS2522 — Algorithmic Problem Solving

Moving Home

## Overview

In this assessment you'll investigate algorithms and data structures related to a house-moving problem, both from the point of view of the person moving house, and from the point of view of the moving company.

### Learning Outcomes

This assessment has the following learning outcomes.

1. Demonstrate an ability to instantiate appropriate algorithms and data structures for a problem.
2. Analyse and understand the effects of different algorithms and data structures on performance.

### Overall Contribution

This assessment will count for 25% of the overall mark for the module.

### Hand-in

This assessment is due at 23:59:59 on the 29th of April 2022. Handing in up to 24 hours late will attract a 10% penalty, while handing in up to 7 calendar days late will attract a 25% penalty. Both penalties will be deducted as a percentage of the mark obtained. Work handed in more than a week late will be treated as a "no paper".

Submission should take place via *myAberdeen and Codio*. Details are as follows.

**MyAberdeen** Please upload a single PDF file containing your report to myaberdeen, covering tasks 1.3, 2.4, 2.5 and 2.6. The textual portions of the report should be no longer than 2 pages in length (single column, 11pt font, and 2cm margins), though you may include any number of graphs you wish on additional pages.

**Codio** You will see that there is a separate subdirectory for each task and subtask for which source code is expected, with a single blank python file in each subdirectory. You should replace this python file (*keeping the same name*) with your source code.

**Please adhere to the formatting instructions, as all outputs will be automatically marked. Marks will be deducted if output formats differ, or if submissions are provided in different file formats.**

Please use Python 3 for your code, and ensure that you implement all data structures except arrays, sets and dictionaries yourself. You should not need to access any libraries except for `sys, io, math` and `random` (though you may wish to use additional libraries to generate your plots). Please include all files necessary for running your code in your submission. To mark your code I will be running your output file with input as specified below.

### Plagiarism

Plagiarism is a serious offence, and will not be tolerated. If you are unsure about whether your work counts as plagiarised, please contact the course coordinator before the submission deadline. For further details, please refer to the Code of Practice on Student Discipline ( `https://tinyurl.com/y92xgkq6`).

# Assessment Tasks

## 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.

$$\begin{array}{lll} X & Y & Z \\ N & & \\ x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ & ... & \\ x_{N-1} & y_{N-1} & z_{N-1} \end{array}$$

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.

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 5 | 0 | 0 |
| 2 | 0 | 6 | 0 |
| 3 | -1 | -1 | -1 |
| 4 | 5 | 5 | 0 |

Here, item 3 could not be placed.

You can use the program `dummy_packing.py` to see example output; run it (from a Linux command line) as follows.

```
python3 packing_example.py | python3 dummy_packing.py
```

### 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 $N - 1$.

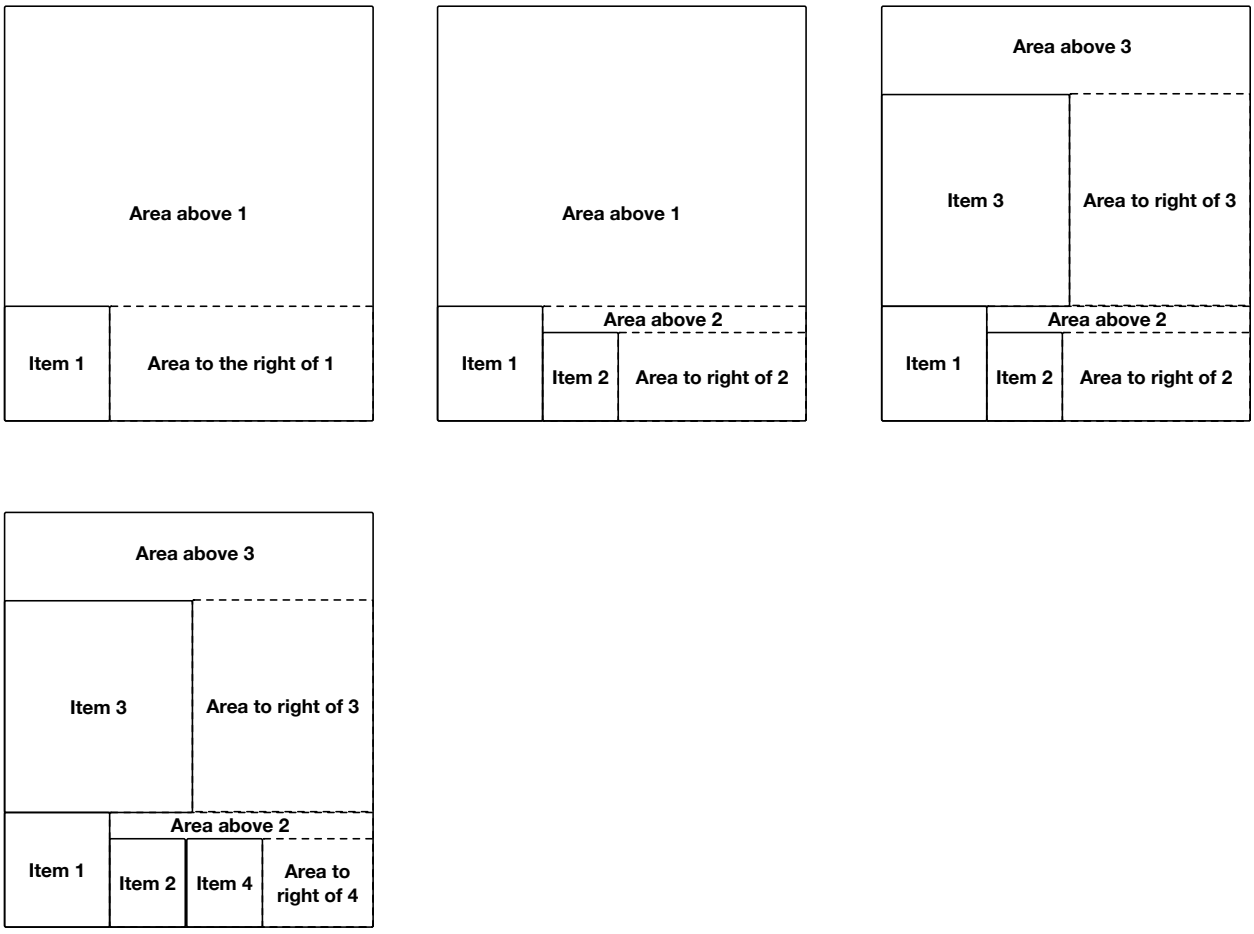


Figure 1: An example of packing

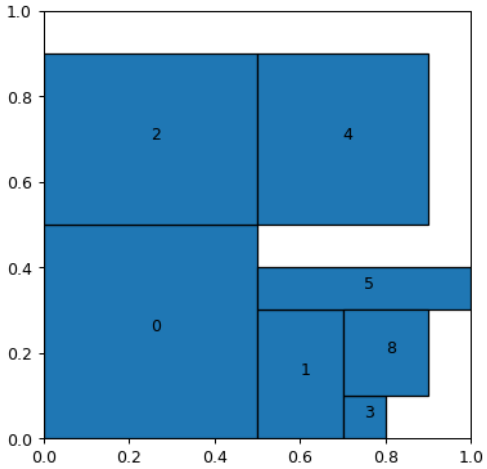


Figure 2: Packing of sample output from example.

**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
5 5 2
2 3 2
5 4 4
1 1 2
4 4 5
5 1 1
5 3 2
3 3 2
2 2 2
5 4 3
```

**Sample Output**

```
0 0 0 0
1 5 0 0
2 0 5 0
3 7 0 0
4 5 5 0
5 5 3 0
6 -1 -1 -1
7 -1 -1 -1
8 7 1 0
9 -1 -1 -1
```

Figure 2 illustrates this packing.

**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.

```
0 0 0 0
1 8 5 0
2 5 0 0
3 2 8 0
4 -1 -1 -1
5 5 4 0
6 0 5 0
7 5 5 0
8 0 8 0
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.

```
0 0 0 0
1 -1 -1 -1
2 5 0 0
3 9 5 0
4 5 5 0
5 5 4 0
6 -1 -1 -1
7 -1 -1 -1
8 -1 -1 -1
9 0 5 0
```

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. Here, for example, for Task 1.2.1 when you again only sort by $x$ coordinate, your output for the sample input should be

```
0 0 0 0
1 8 5 0
```

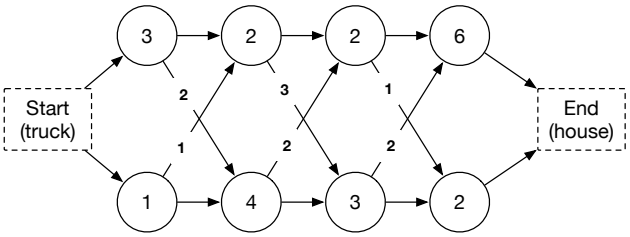


Figure 3: Example of two lines delivering items.

```
2 0 0 2
3 2 8 0
4 -1 -1 -1
5 5 0 0
6 0 5 0
7 5 5 0
8 0 8 0
9 5 0 2
```

If you need to prioritize between different dimensions, prioritize $x$ over $y$ over $z$.

## 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

```
N S
<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

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

```
0 3 0 3 6
1 0 1 0 1
1 0 0 1 1
1 1 1 1 4
1 1 0 2 2
1 2 1 2 3
1 2 0 3 2
1 3 1 3 2
```

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
```

Where `N` is the number of items to offload and $t1$ is the time the first item is offloaded, $t2$ the time the second is offloaded etc., implement a greedy approach to offloading the items from the truck, printing out the time taken to offload all N items.

The `offloading_example.py` program generates output in this format.

In your report, prove that this greedy approach is, or is not optimal.

# Marking Scheme

Note that sub-tasks build on each other. You will not receive marks for later parts of task 1 if earlier parts are not completed. Similarly, you will not receive marks for later parts of task 2 if earlier parts of task 2 are not completed.

Task 1 and Task 2 will be marked independently of each other; each count for half of the final mark for this assignment. CGS marks therefore reflect the marks given for the parent task.

| Task | Marking Notes |
|---|---|
| 1.1 | Completing this task will yield a CGS E3 |
| 1.2 | Completing 1.2.1-1.2.3 will yield a CGS D3. Completing these and 1.2.4 will yield a CGS C3. |
| 1.3 | This task will yield up to a CGS B3 depending on the quality of evaluation etc. |
| 1.4 | Successful implementation of an extension of 1.2.1-1.2.3 will result in up to a CGS A4; 1.2.4 will yield a CGS A3, while the evaluation will yield up to a CGS A1. |
| 2.1 | Completing this task will yield a CGS E3 |
| 2.2 | Completing this task will yield a CGS D3 |
| 2.3 | Completing this task will yield a CGS C3 |
| 2.4 | Completing this task will yield a CGS B1 |
| 2.5 | Completing this task will yield up to a CGS A3 |
| 2.6 | Completing this task will yield up to a CGS A1 |

# Changelog

| | |
|---|---|
| 31/03/2022 | Sample output updated for 1.4(.1) |
| 25/03/2022 | Sample output given for 1.2.3. |
| 23/03/2022 | More information given for Task 1.2.1. |
| 21/03/2022 | More example output added. |
| 16/03/2022 | More explanation given for 1.2.1-3. |
| 14/03/2022 | Gave a bit more information for Task 1.2.4. |
| 10/03/2022 | Corrected sample input and output for Task 1.1 and added figure. |
| 28/02/2022 | Added Sample input and output for Task 1.1. |
| 24/02/2022 | Corrected hand-in time. |
| 24/02/2022 | Clarified the order in which items should be placed. |
| 14/02/2022 | Initial version. |