NTNU

Kunnskap for en bedre verden

Assignment 1: Container Ships

TPK4186 - Advanced Tools for Performance Engineering

Done by

Georg Hove Zimmer

Please note that I am working alone and have thus made some assumptions and limitations that may not always be the most efficient in practice. This is because the code could potentially become too complex to do alone, and I believe these simplifications were necessary.

Logic

The objective of this assignment is to process large datasets using Python. We will be optimizing the placement of containers on a cargo ship. To solve this problem, I have constructed a data structure that represents the ship as a list consisting of length, width, height, and sections. The first assumption and limitation in this assignment is the dimensions of the ship. The code is designed to allow for selecting different dimensions, but for the purposes of this assignment, we will only be considering a ship with a length of 24 feet, a width of 22 feet, and a height of 18 feet (see Figure 1).

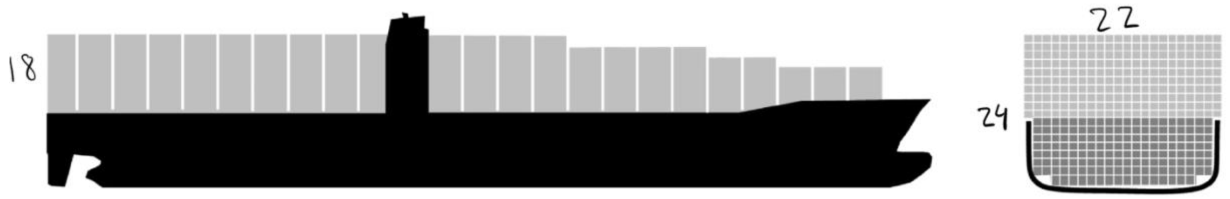


FIGURE 1

The ship consists of six sections, two in the front, two in the middle, and two in the back, and therefore three on the starboard and three on the port side (see figure 2). Each section is half the width of the ship and one-third of the length of the ship. Each section contains stacks of containers. Each container is represented as a list with a serial number, length, weight, and cargo. Note that the lists are zero-indexed, so the indices in the code will be one less than those shown in figure 2.

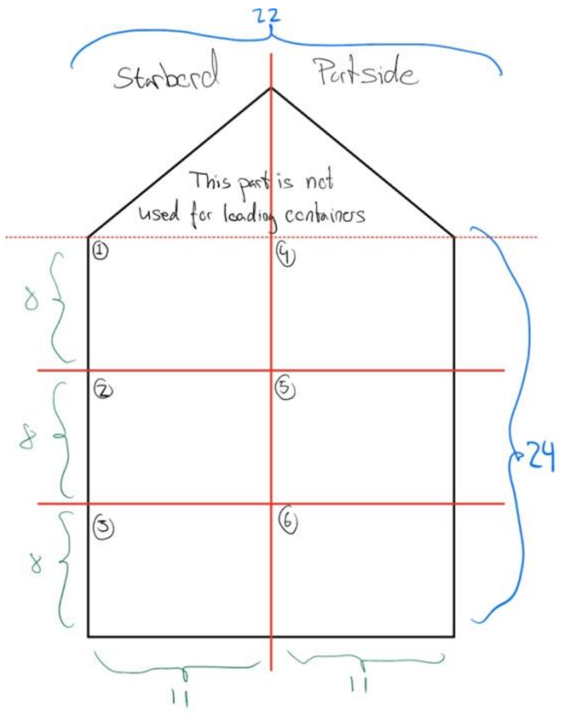


FIGURE 2

We are considering two different types of containers, either 40-foot-long containers or 20-foot-long containers. An important aspect of solving this task is that one cell is 40 feet long, so two 20-foot-long containers are needed to fill one cell. This means that each section is 11 cells wide, 4 cells long (not 8), and 18 cells high. We envision that each list in the section is a stack that goes vertically and has a length of 18. Each section will thus have 11x4 stacks, and the first element in each stack represents the first floor of containers.

So a ship has this structure:

[length, width, height, [[section 1], [section 2], [section 3], [section 4], [section 5], [section 6]]**]**

And each section has this structure:
[[stack 1], [stack 2], [stack 3], [stack 4], [stack 5], [stack 6], [stack 7], [stack 8], [stack 9], [stack 10], [stack 11], [stack 12], [stack 13], [stack 14], [stack 15], [stack 16], [stack 17], [stack 18]**]**

And each stack has this structure:
[container 1, container 2, container 3, container 4, ..., container 43, container 44]

Assumptions

Assumption for the whole assignment

- It is assumed that the ship has a length of 24, a width of 22, and a height of 18.
- It is assumed that each cell can hold a 40-foot container.
- We assume that all containers are placed lengthwise along the ship. This allows for 11 containers in width and 4 in length since each cell is 40 feet long.

Assumption for task 5

- Since we need to place each container individually without creating any "gaps," we must ensure that a 40-foot container is not placed on top of a 20-foot container (as the 40-foot container would fall). Therefore, I will "combine" all 20-foot containers and place them in pairs. This results in a new container with the total weight of both containers. This method ensures that there will never be gaps in the placement, and we can still sort the containers so that the heaviest ones are on the bottom. We assume that it is sufficient to match the heaviest 20-foot container with the first available one, rather than creating pairs of the heaviest containers. The most important thing is to ensure that the heaviest 40-foot containers are on the bottom

Assumption for task 5

- I do assume we have to place at least 1000 containers before the balance matter. This is because the ships own weight will handle the first containers.

Assumption for task 10

- Since the load function always places a container on the lightest section and the lightest stack within that section, we will always take into account balancing the ship by checking whether it meets the requirements. I assume that it is sufficient to check the balance of the ship after each container is placed and to have an algorithm that always places the container in the lightest section. I could have checked for each container whether it would unbalance the ship (exceeding balance requirements), but this would have led to a very long run time without adding much value to the code. Therefore, I will place each container in the lightest section and the lightest stack within that section, and then check if the ship is balanced.

Assumption for task 11& 12

- Since the ship is divided into six sections, with three sections lengthwise, it is assumed that there are three cranes
- Since containers are loaded onto the ship in such a way that the ship is always balanced, each section will be filled at approximately the same rate as more containers are added. Since three containers would place containers in each of the ship's sections, i.e., one crane in sections 1 and 4, one crane in sections 2 and 5, and one crane in sections 3 and 6 (see figure 2), the total number of operations would only be three times as few as my code is implemented. Therefore, it is assumed that three cranes would only shorten the loading time by three.

Tasks and documentation

Task 1-3:
My code is structured as described on page 2. Containers have this structure:

```
def Container_New(serialNumber, length, weight, cargo):
    return [serialNumber, length, weight, cargo]
```

The ship and sections has this structure:

```
def Ship_Stack():
    return [None for _ in range(18)]
```

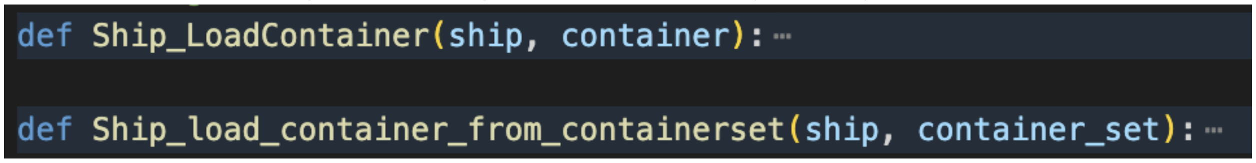
```
def Ship_Section():
   return [Ship_Stack() for _ in range(11 * 4)]


def Ship_New(length, width, height):
   return [length, width, height,
         [Ship_Section() for _ in range(6)]]
```

Task 4 & 6:
Write to file is completed and the results are in “container_set.tsv”. Load to file can be excecuted by running:

```
print(load_containers_from_file("container_set.tsv"))
```

And you will see that it prints a list with container.

Task 5 & 7:
I take one container and I place it in the lightest section and in the lightest stack in that section, and I check the weight of all containers in that stack. I always pair the 20-for-containers together. I obtain the requirements of “no holes”, sorted stack with heaviest at the bottom and a balanced ship. This is the core logic to the assignment, and its mostly done by these two functions:

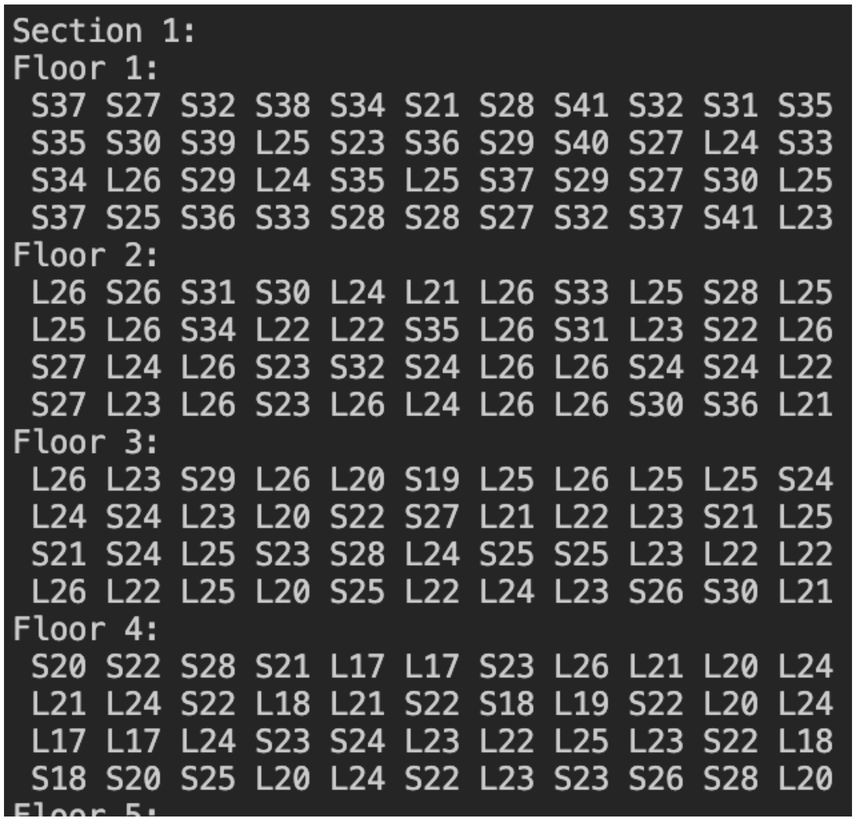


```
def Ship_LoadContainer(ship, container):⋯

def Ship_load_container_from_containerset(ship, container_set):⋯
```

An important aspect of the code is how the tostring prints out the code and how to interpret the output.
Each stack is sorted with the heaviest element in the beginning at the list. So each floor in one section is the first element of each stack in this section. The second floor is the second element in each stack in each section.

Example with the first 4 floors in section 1:
Note that “S” means short, so every “S” is two 20-foot containers. “L” is long containers, and the number is that containers total weight.

The first stack has these four elements [S37, L26, L26, S20,…]. Each floor is like a matrix and a stack has its elements in the same position in each floor, starting from floor 1.



```
Section 1:
Floor 1:
 S37 S27 S32 S38 S34 S21 S28 S41 S32 S31 S35
 S35 S30 S39 L25 S23 S36 S29 S40 S27 L24 S33
 S34 L26 S29 L24 S35 L25 S37 S29 S27 S30 L25
 S37 S25 S36 S33 S28 S28 S27 S32 S37 S41 L23
Floor 2:
 L26 S26 S31 S30 L24 L21 L26 S33 L25 S28 L25
 L25 L26 S34 L22 L22 S35 L26 S31 L23 S22 L26
 S27 L24 L26 S23 S32 S24 L26 L26 S24 S24 L22
 S27 L23 L26 S23 L26 L24 L26 L26 S30 S36 L21
Floor 3:
 L26 L23 S29 L26 L20 S19 L25 L26 L25 L25 S24
 L24 S24 L23 L20 S22 S27 L21 L22 L23 S21 L25
 S21 S24 L25 S23 S28 L24 S25 S25 L23 L22 L22
 L26 L22 L25 L20 S25 L22 L24 L23 S26 S30 L21
Floor 4:
 S20 S22 S28 S21 L17 L17 S23 L26 L21 L20 L24
 L21 L24 S22 L18 L21 S22 S18 L19 S22 L20 L24
 L17 L17 L24 S23 S24 L23 L22 L25 L23 S22 L18
 S18 S20 S25 L20 L24 S22 L23 S23 S26 S28 L20
Floor 5:
```

The loading function uses insertContainer and popLighterConteiners which covers this task, but I added a removeContainer and findContainer to the code, just to be sure I answered every aspect of this task. These could be called on in main:

```
new_container = ContainerManager_NewRandomContainer()
Ship_LoadContainer(ship, new_container)
print(Ship_FindContainer(ship, new_container))
Ship_RemoveContainer(ship, new_container)
```

Task 8 -10:

The ship is always balanced with the algoritm always placing a container at the lightest section and the lightest stack in this section, but I also check all the constraint in the code separately. Please note the assumption for these tasks. Run this code:

```
print("The total weight of the ship is: ", Ship_GetTotalWeight(ship))

for i in range(Ship_GetNumberOfSections(ship)):
    sectionWeight = Ship_GetTotalWeightOfSection(Ship_GetSection(ship, i))
    print("Section", i+1, "has weight: ", sectionWeight)

print("The total weight on startboard of the ship is: ",
     Ship_GetTotalWeightOnStarboard(ship))
print("The total weight on portside of the ship is: ",
     Ship_GetTotalWeightOnPortSide(ship))
print("\t")
print("The total weight in the front of the ship is: ",
    Ship_GetTotalWeightInFront(ship))
print("The total weight in the middle of the ship is: ",
    Ship_GetTotalWeightInMiddle(ship))
print("The total weight in the back of the ship is: ",
    Ship_GetTotalWeightInBack(ship))

Ship_CheckLoadBalance(ship, 5, 10)
```

And the rest should be self-explanatory with function-names.

Task 11 & 12:

By running this code, you get how many days it would take to load the ship.

```
print("Number of operations is: ", operations)
print("It takes ", calculate_loading_time_with_one_crane(operations), "to load the ship with one crane")
```

It takes around 150-180 days to load the ship completely, which is much higher than it should in practice. This is mostly because we don't sort the container_set beforehand. We take a random set of containers, and structure is many nested lists. This leads to many iterations and operations.