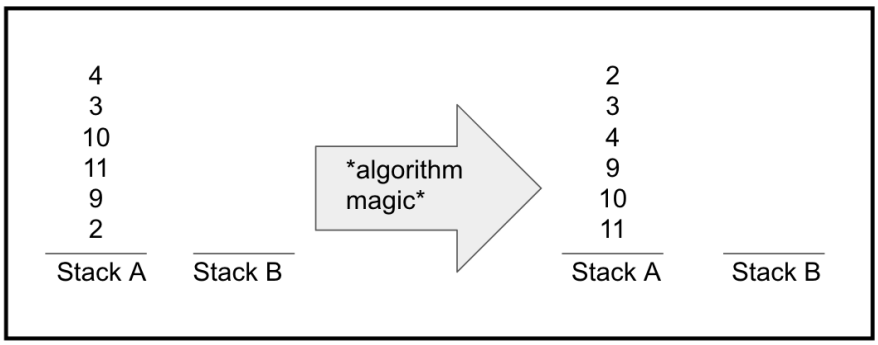
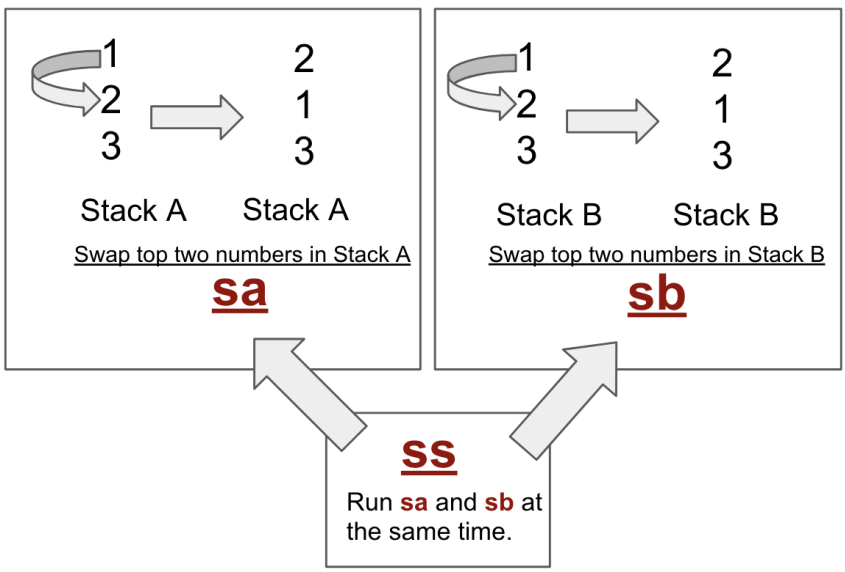
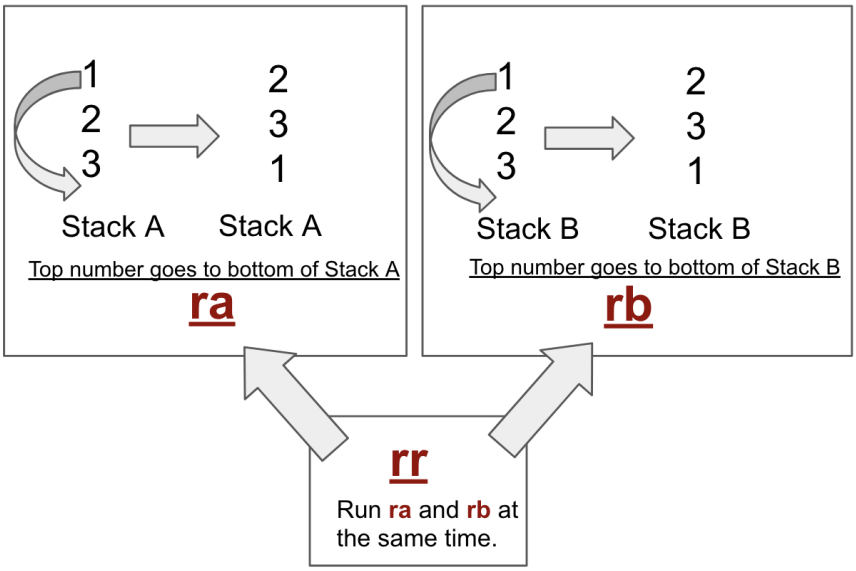
Push Swap

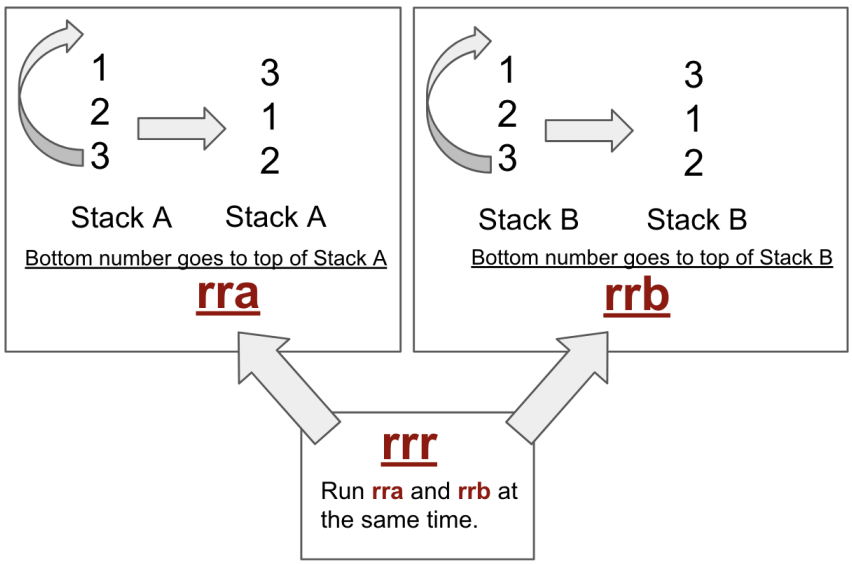
Push swap is a 42 project where we have a stack (stack A) of unorganized integer numbers, that we need to sort from smallest to biggest using two stacks A and B. You have a limited set of actions that you are allowed to use. The goal is sort “stack A” in the least amount of instructions possible.

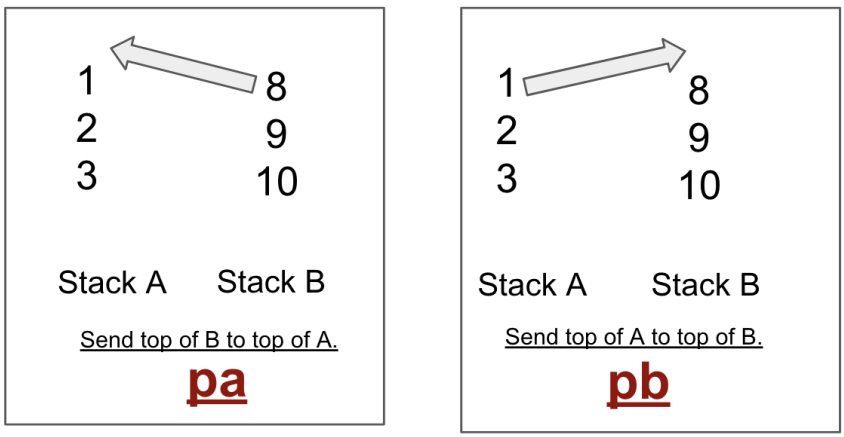


The set of actions you may use is : **sa, sb, ss, ra, rb, rr, rra, rrb, rrr, pa, pb**.







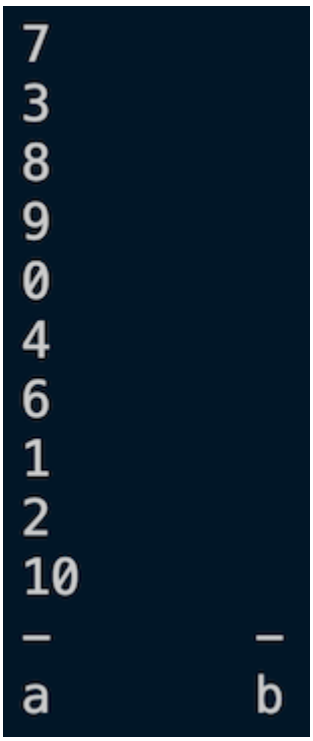


We will be using one algorithm to sort “stack A” containing 4 numbers and up. So what is it and how does it work ?

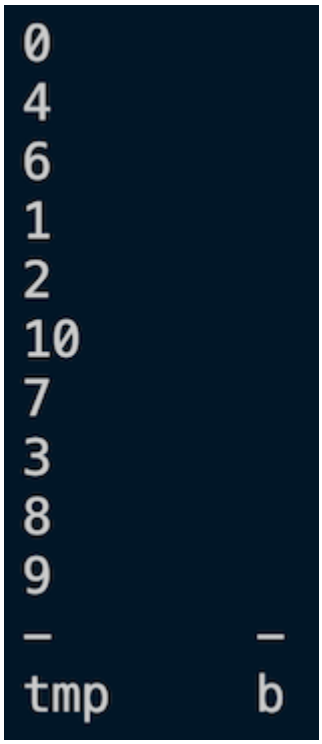
Well, I don't have a particular name for it, or if it already has one, but let's call it `Find the best element` algorithm.

How does it work ?
First we begin by finding the Largest Increasing Sequence (LIS) in stack A(starting from the smallest number in the stack).

As an example we'll define stack A as :

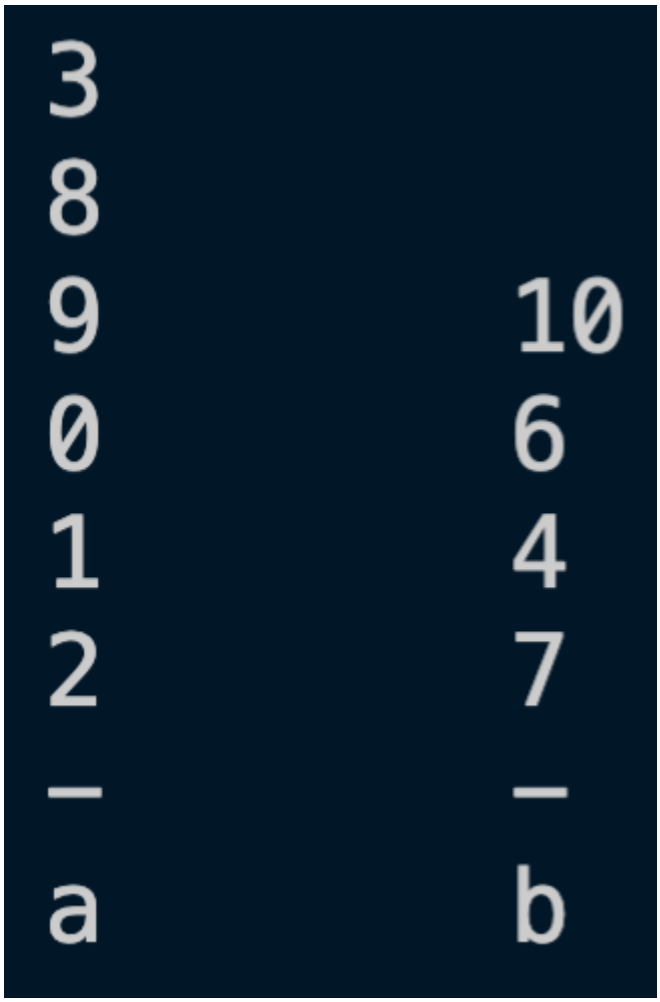


In the beginning, we need to rotate the stack A in a duplicate stack tmp and put the smallest number (in this case **'0'**) on its top, using **'ra'** or **'rra'** depending on if the number is in the top or bottom half of the stack. Here we use **'ra'** since it's in the 5th position from the top, we obtain this result :



Then we can determine the LIS as : **0, 1, 2, 3, 8, 9**. There are many tutorials on how to find the LIS, look it up yourself.

Now that we have our LIS, we push all the elements that do not belong to it to stack B, thus our two stacks will look something like this :



The next and most crucial step is determining what is the best element in stack B that we can push to its correct position in stack A. How can we do that ?

We iterate the stack B from both the top and the bottom until we reach its middle(same for stack A), each time calculating and saving the number of moves necessary to put the current number on the top of stack B, and the number of moves to put it in its correct position in stack A.

For example, the number of moves necessary to put **'10**' on the top of stack B is **'0'** since it's already on the top. In stack A, **'10'** has to be placed between **'9'** and **'0'**, so the number of moves in stack A is 3, add to it **"pb"**, it becomes 4 to put **'10'** in its correct position.

How did I know where **'10'** needs to be placed ?
There are 4 cases that need to be checked to find where my number shall be placed in stack A:
P.S : Remember that we are iterating stack A from both the top and bottom.

- 1- It's between stack_A[size_of_stack_A **-** 1] and stack_A[0]. **[2, 3]** (You only need to check this once).
- (2- It's between stack_A[i] and stack_A[i + 1]. **[3, 8].** i starting from 0 to <size_of_stack_A / 2>, in our case, size_of_stack_A = 6).
- 3- It's between stack_A[j] and stack_A[j **-** 1]. **[1, 2]**, j going from <size_of_stack_A **-** 1> to <size_of_stack_A / 2>.
- 4- None of the above. In this case, the number is the maximum of both stacks. Here we have to find the position of the maximum number of stack A(which is index 3, position of **'9'** in our example), that's where **'10'** needs to be placed.

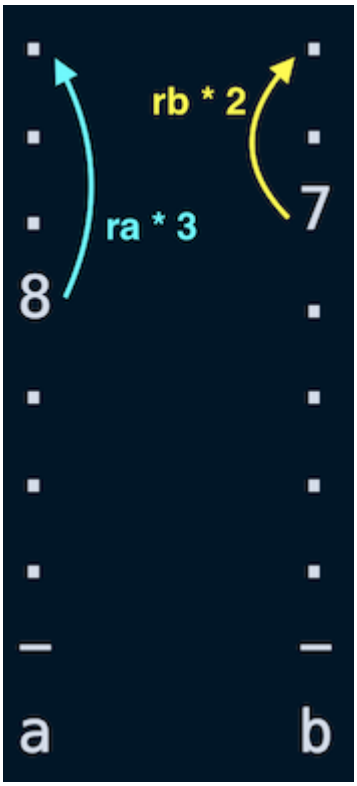
Now we have to save these positions in an array of size 2 as : **"tmp[2] = {pos_a, pos_b}"** (for example for **'10'**, tmp[2]={3, 0}) so we can compare the next number's positions with the previous ones. I know it's a bit confusing, but don't worry, you'll understand it in a bit ;).

We always consider the first number of stack B as being the best element to push back to stack A, but as we go along with our iteration through stack B, we will(or won't) find another best number to push to A.

Let's now calculate the positions for number **'7'**, we get **"tmp[2] = {1, -1}"**(don't worry about the negative value, It'll be explained later), which is better than the positions of number **'10'**, thus the best number now becomes **'7'** instead of **'10'**. Are you starting to get it now ? Cool.

For number **'6'**, **"tmp[2] = {1, 1}"** is equal to that of **'7'**, so the best number doesn't change. Can you calculate the moves for **'4'** now ? **'4'** has **"tmp[2] = {1, -2}"**, thus the best number doesn't change. Is that the result you got ? If yes, cool, if not, it's ok, just do it again and you'll get it.

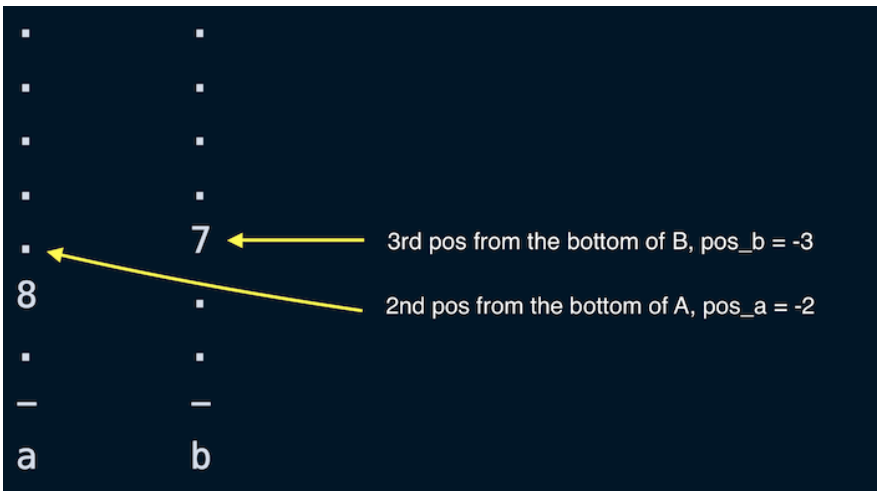
In this example, you can push **'7'** to stack A easily, but what if **'7'** had **"tmp[2]={3, 2}"**, and it is the best number we can push ? Of course you can do **"rb"** twice, and **"ra"** three times, but that wouldn't minimize the number of instructions, would it ? 2 * **"rb"** + 3 * **"ra"** = 5.



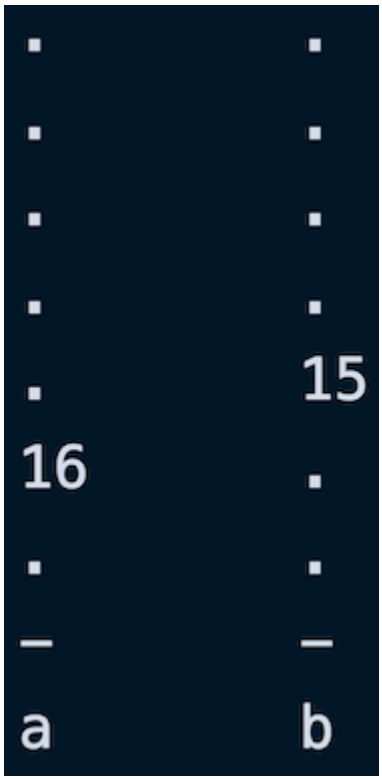
In this case we can use smart rotate **“rr”** to minimize the number of instructions 2 * **“rr”** + **“ra”** = 3.

You may be wondering, but how would I know when I should use **“rr”** or **“rrr”** instead of **“ra/rb”** or **“rra/rrb”** ? and how to use it ?.

When iterating through stack B, first you have to check if the current number's positions are either both in the top half or the bottom one. The way I did it is by using negative indexes, which means if I have a number that's in the bottom half of stack B, I save its **pos_b** as a negative number, and if its correct position in stack A is in the bottom half, **pos_a** will be negative too. But you may use whatever that's suitable for you to handle the bottom half numbers.

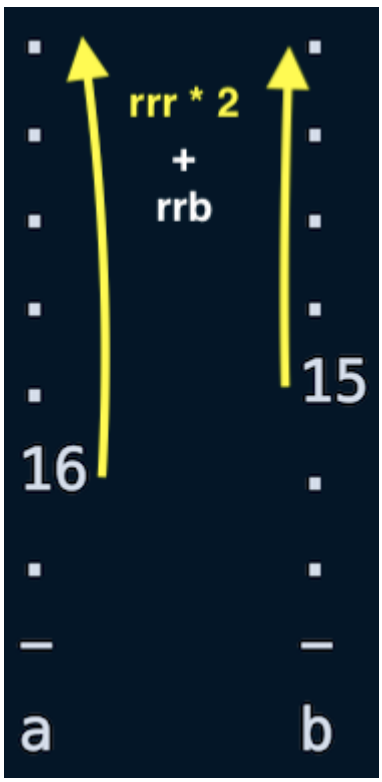


Now we have to take in consideration these smart rotates to determine the best number. To do that, you calculate the maximum moves the current number can do to be in its correct position in stack A. For example, if our best number is, let's say **'15',** is in the bottom half of stack B that has **"tmp[2]={-2, -3}"**, the maximum moves it needs to do is 3.

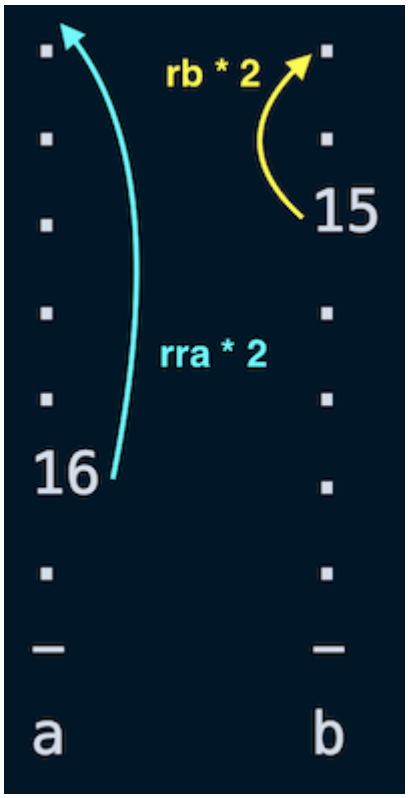


Do you get how I calculated it ?
If not, here's how, I check if **<pos_a * pos_b>** is positive, if it is, then I use :

> ***<"rrr">*** ** < the minimum between (abs(**pos_a**) and abs**(pos_b)***
> **>**



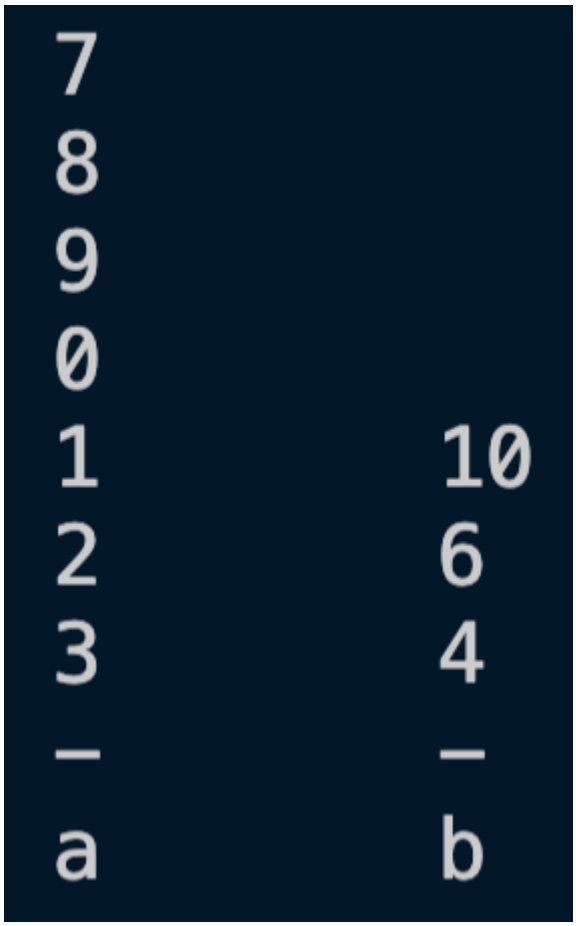
Otherwise I use normal rotates **“ra/rra”** and **“rrb/rb”**.



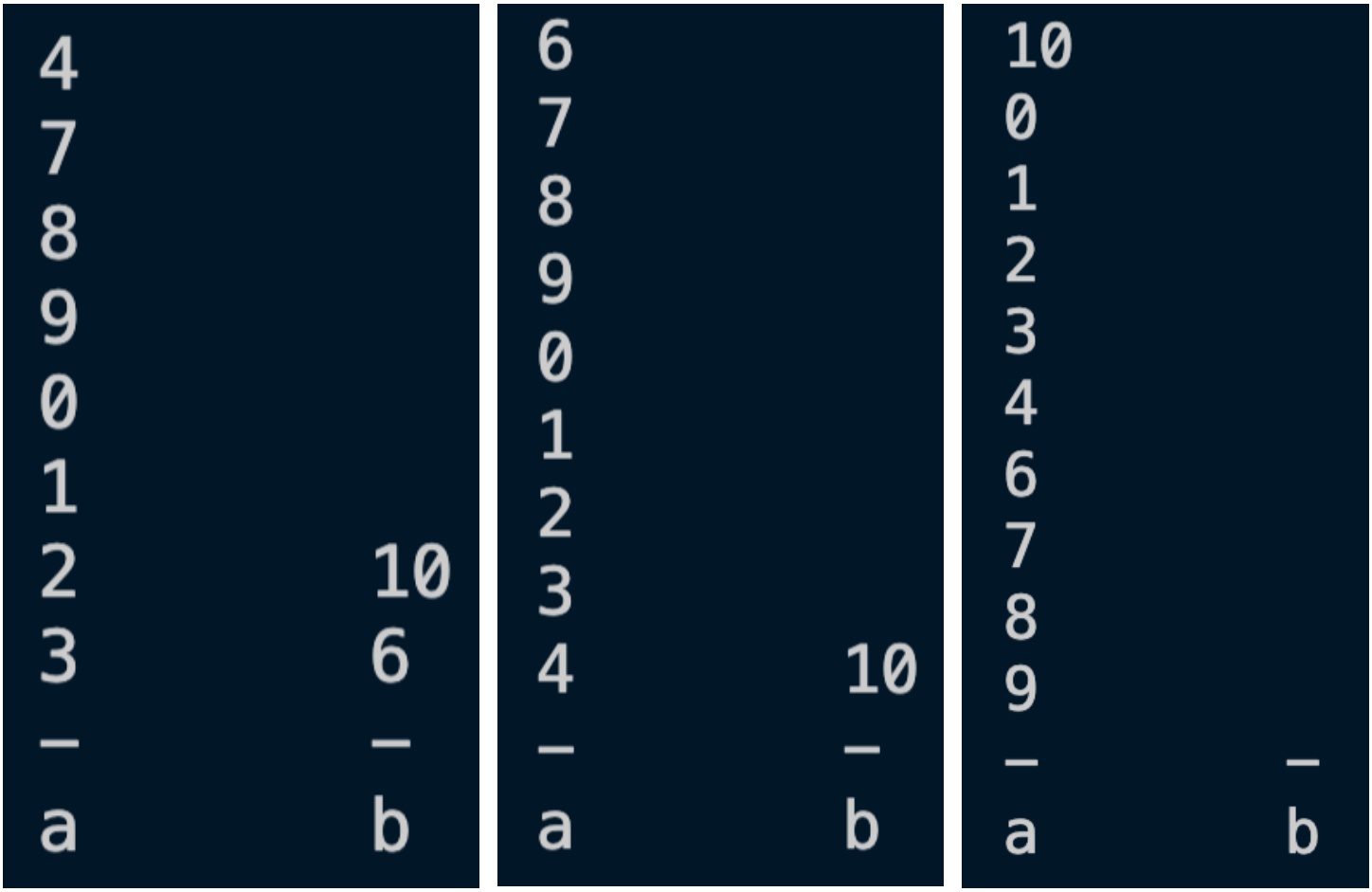
The same applies if the best number is in the top half, you just use positive **pos_a** and **pos_b**.

The negative numbers are only used to represent numbers that are in the bottom half of the stack, but when you rotate the stacks, you have to use the absolute values of **pos_a** and/or **pos_b,** duh !.

Back to our example, **‘7’** can be pushed to stack A, which leads to this :

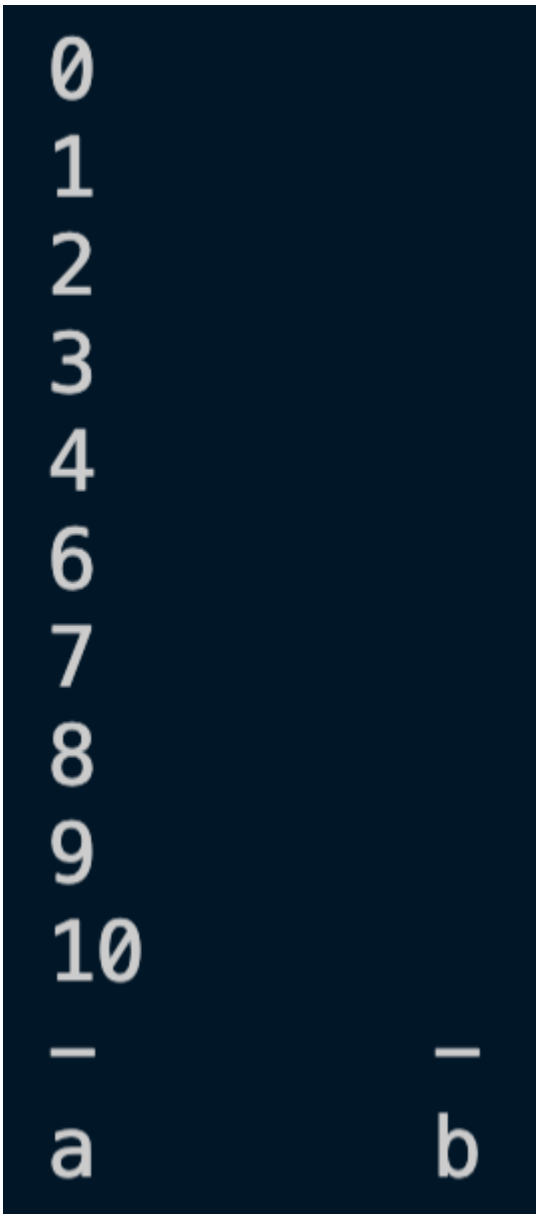


Now it’s just repetitive work until stack B is empty:



The final step is to check if the smallest number is in the top of stack A. If it is, that's what we want, if not we'll have to rotate it to the top with of course **'ra'** or **'rra'** like we did in determining the LIS.

Congratulations, you have successfully sorted the stack A in the least amount of instructions possible, yaay!

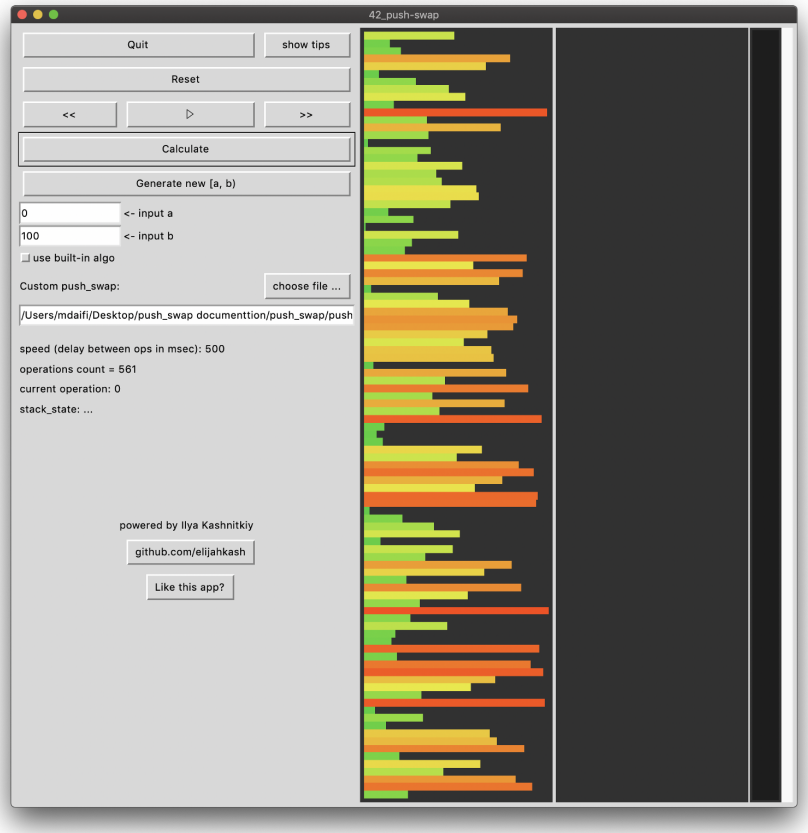


There are many visualizers to help you see how the numbers are being sorted, but it will be more useful to see if you did something wrong.

push-swap-gui

Implementation of push-swap (42-school project) on python with GUI. Be welcome to use my ...

pypi.org



You need to use python3 to execute this visualizer, or use a virtual environment by doing:

- python3 -m venv env
- source env/bin/activate
- python3 -m push_swap_gui

Credits to the owner of the visualizer who helped me figure out how his algorithm works and allowed me to use it in my project.

Good luck to you all.