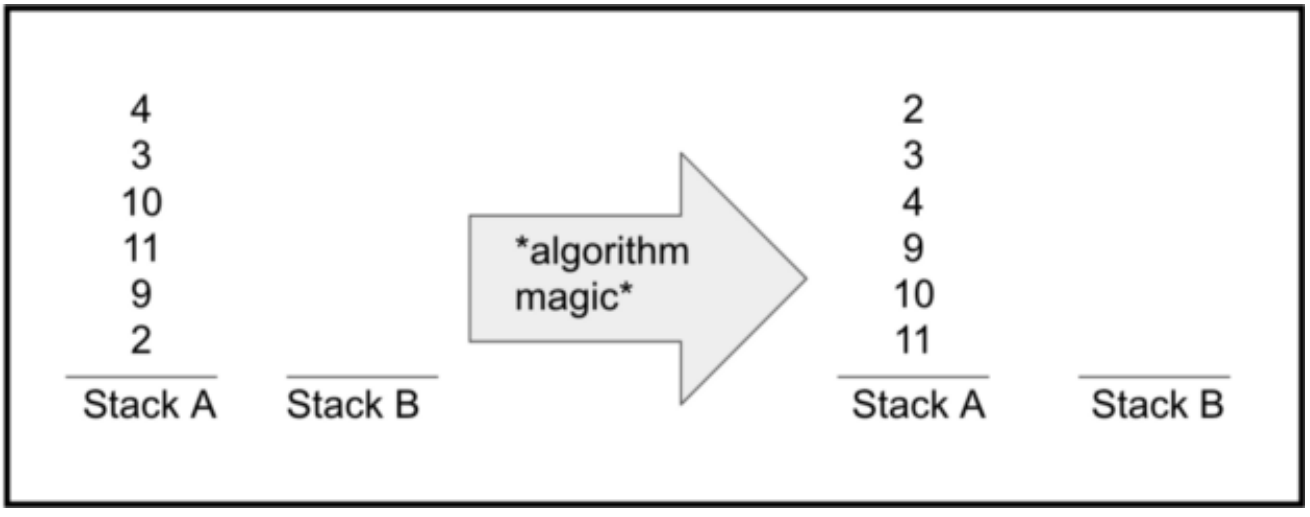
Jamie Dawson

Push_Swap: The least amount of moves with two stacks

Jamie Dawson May 11, 2019 · 7 min read ★

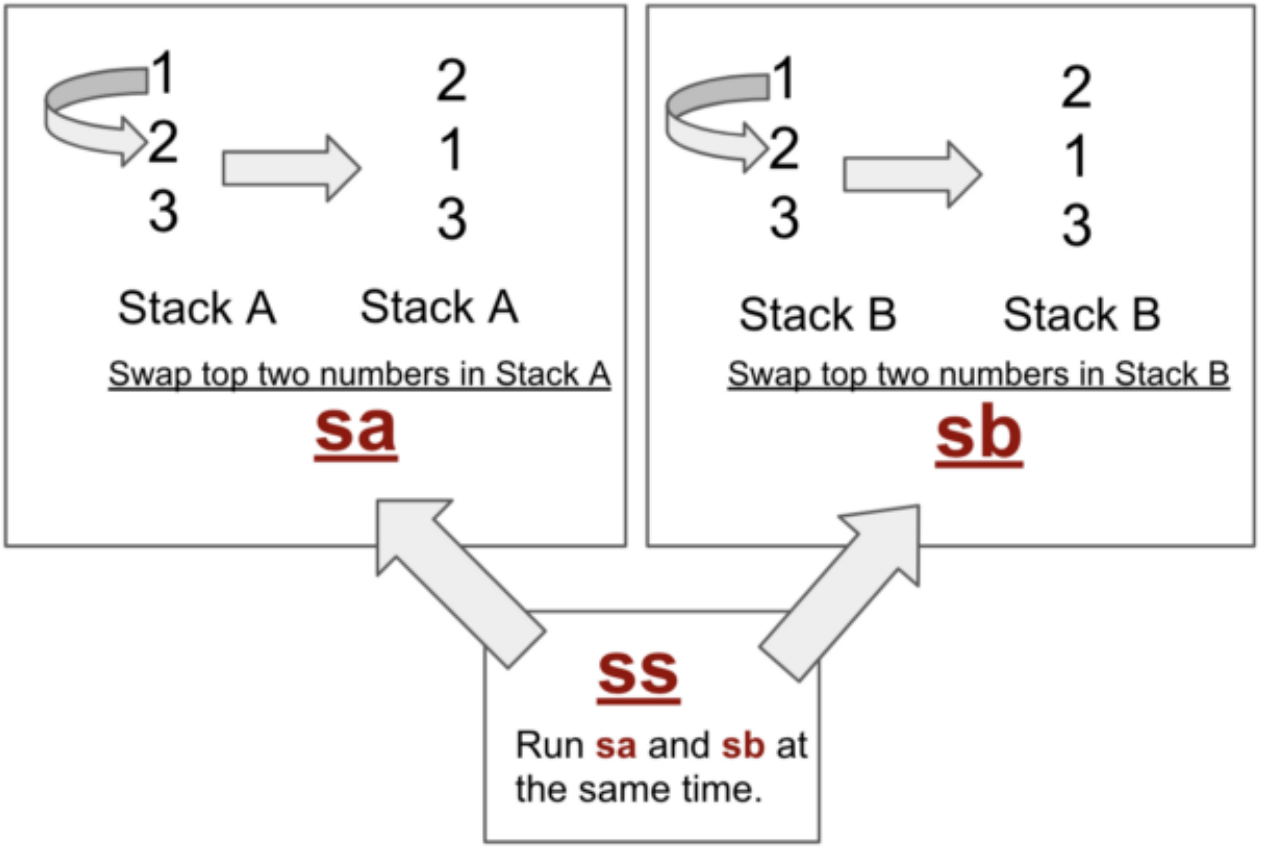
When I first became an official student at 42 Silicon Valley, I knew early on that I wanted to tackle some of the algorithms projects that you can unlock.

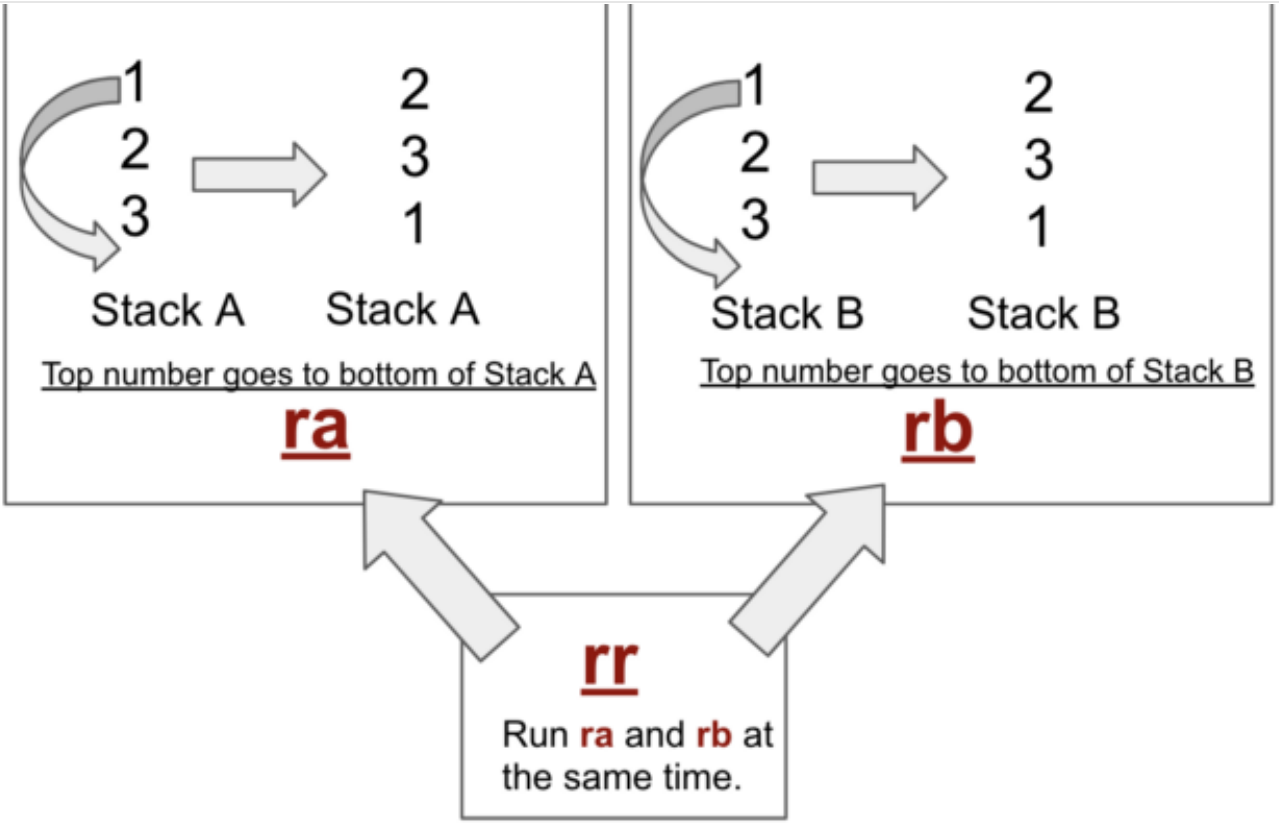
One of the algorithm projects I unlocked is called **Push_Swap**. The idea is simple, You have two stacks called **Stack A** and **Stack B**. Stack A is given a random list of unorganized numbers. You must take the random list of numbers in Stack A and sort them so that Stack A is organized from smallest to largest. There are only a few moves you're allowed to used to manipulate the stacks that we're going to call "Actions". The main goal of this project is to organize Stack A in as few actions as possible.

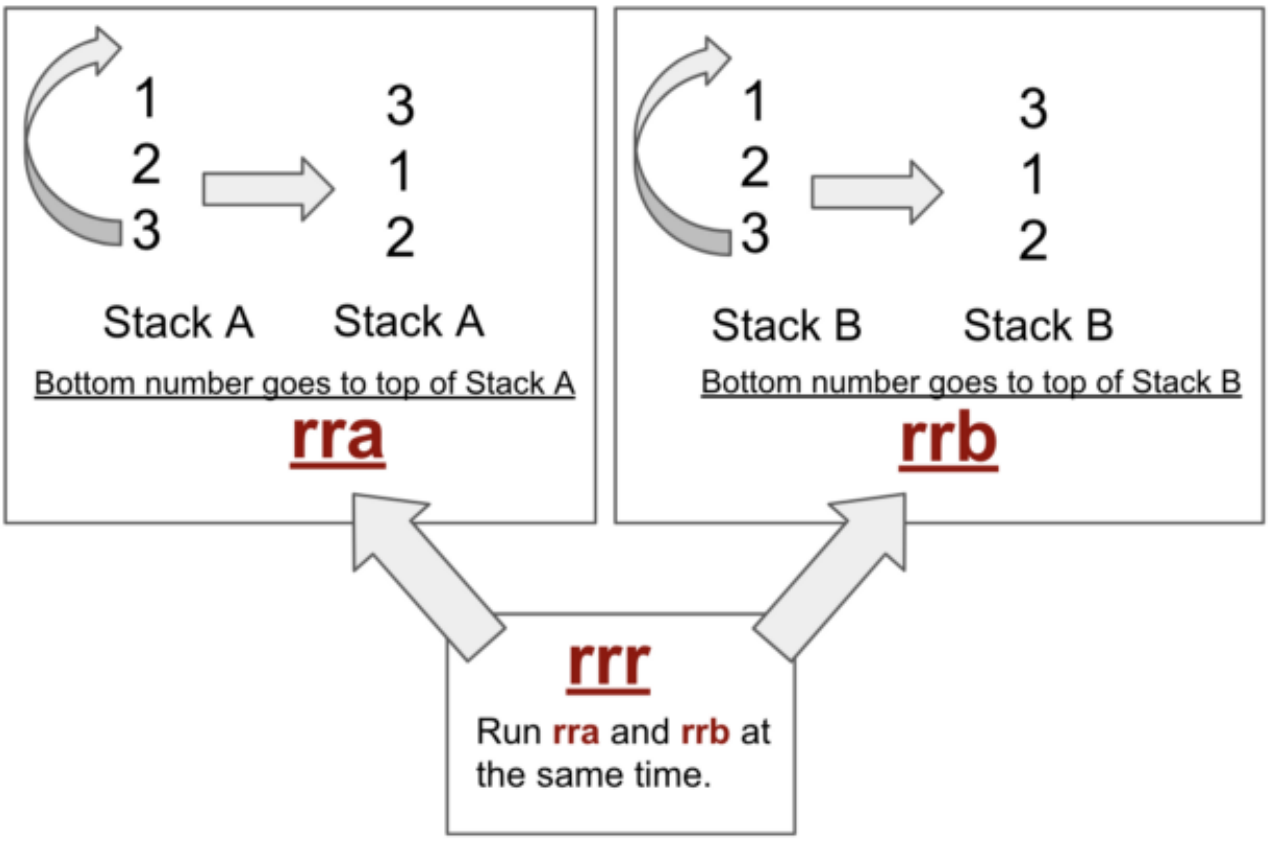


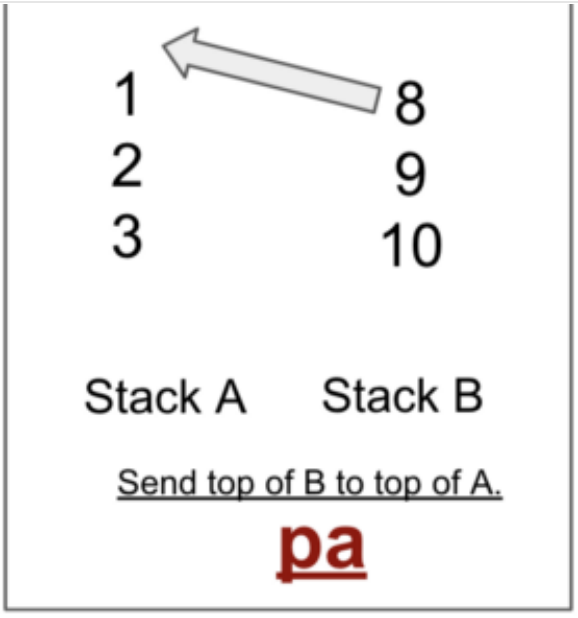
The actions are named: **sa, sb, ss, ra, rb, rr, rra, rrb, rrr, pa, pb**.

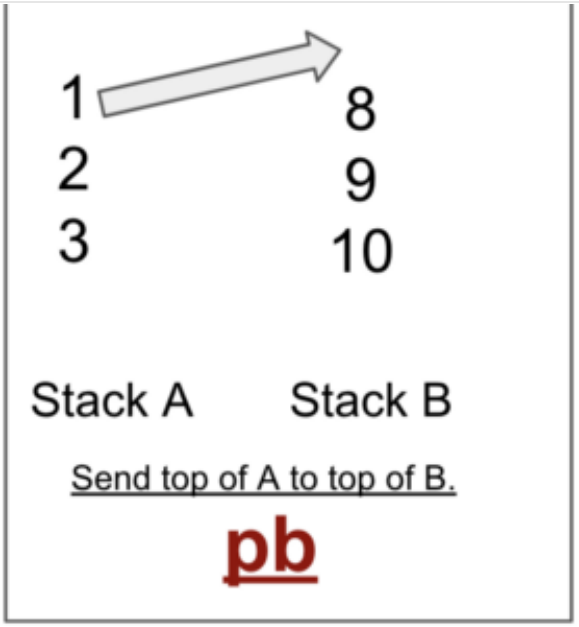
Here's how they all work:







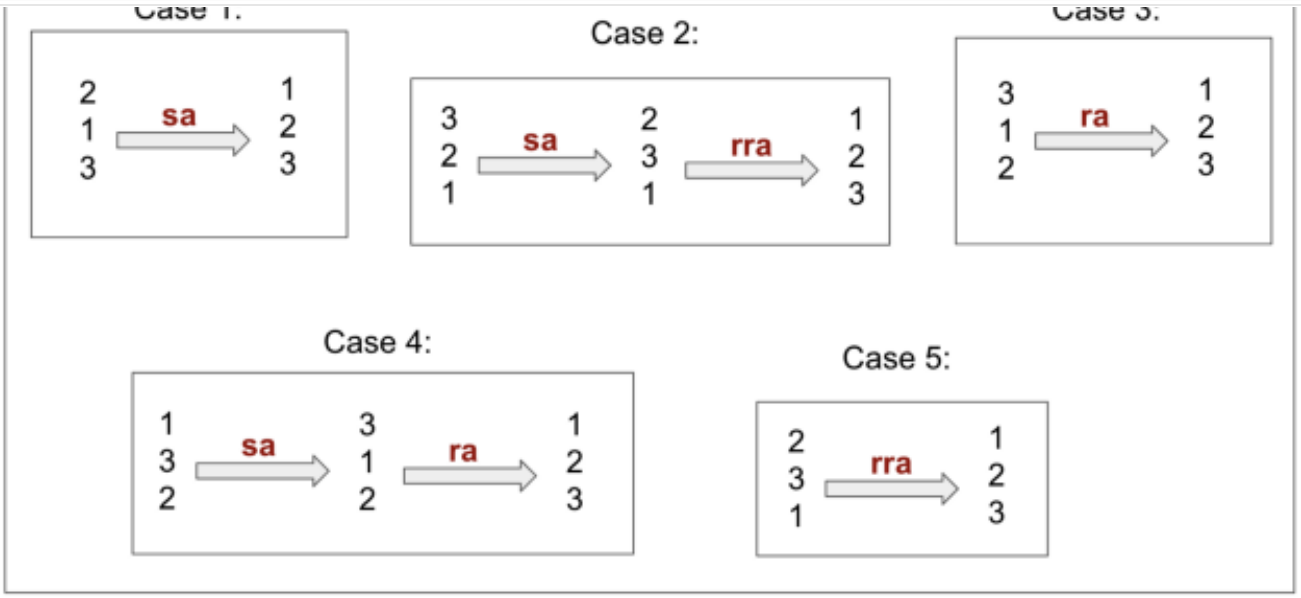




So now that we have the actions we are allowed to use, how should we use them? The algorithms we'll use depends on how many random numbers are getting applied to Stack A. There are 4 main test cases that I want to cover. The cases are **3**, **5**, **100**, and **500**. Each case requires me to handle them differently, so I'll break down how I optimized each case one at a time.

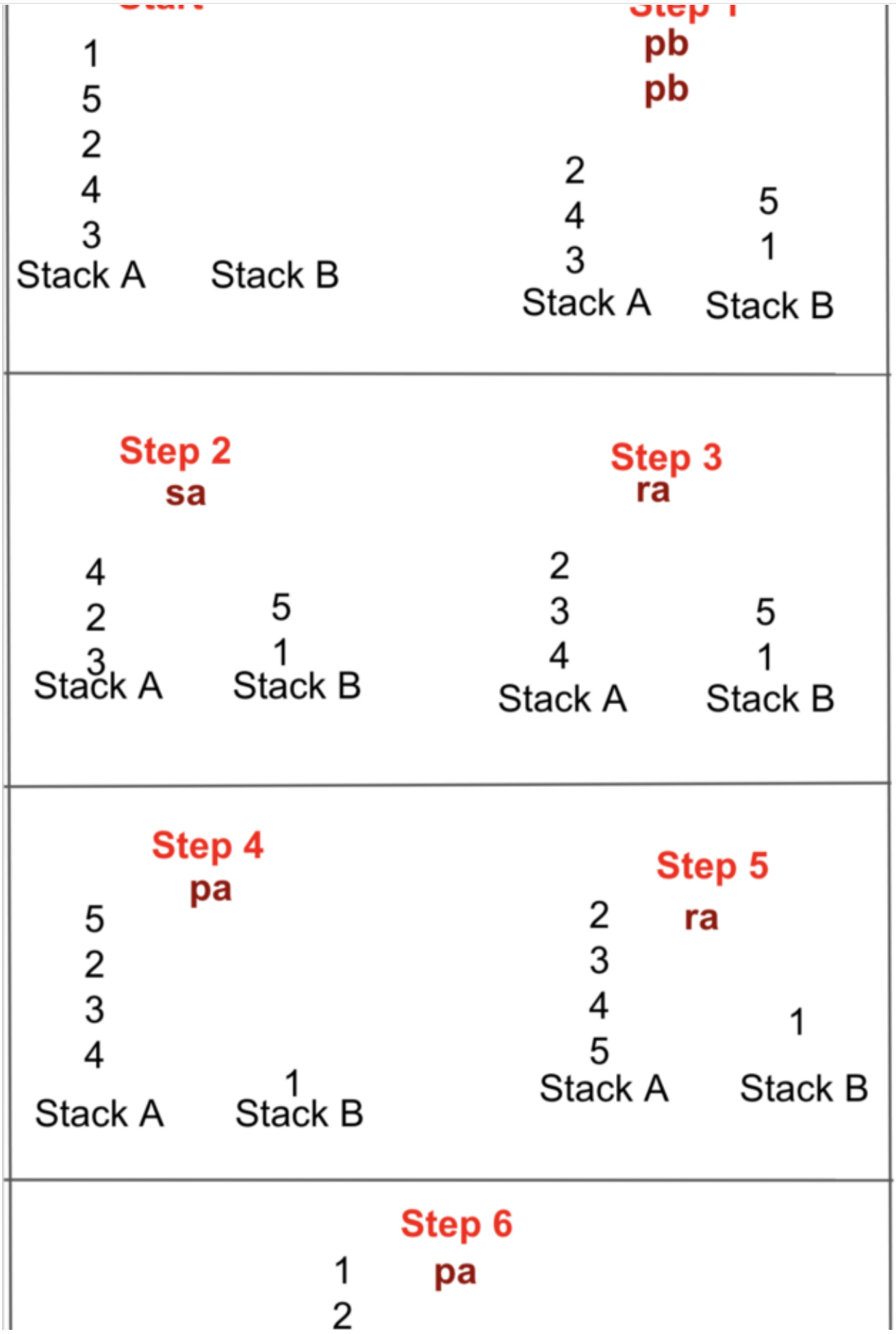
3 random numbers:

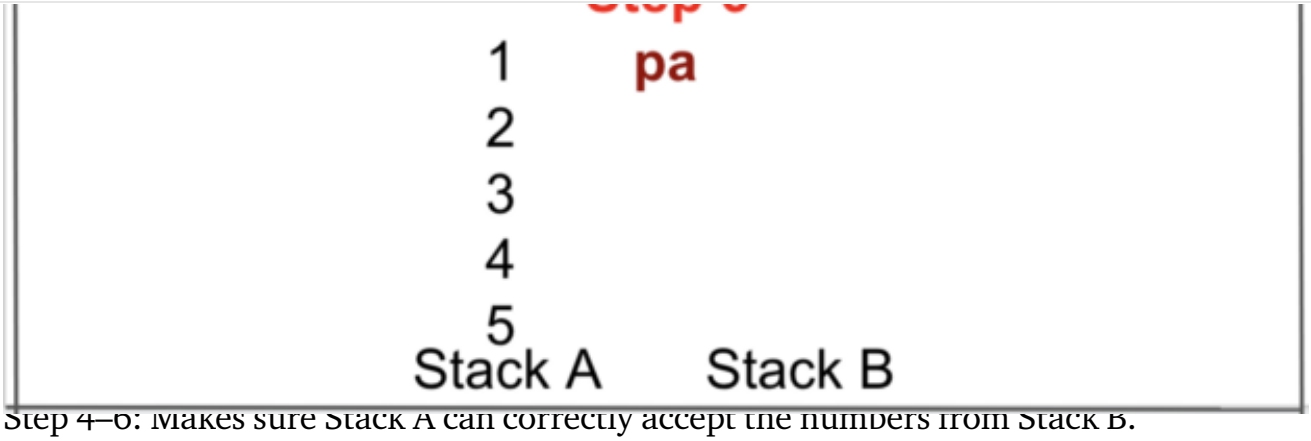
I realized there are five possible cases for only three random numbers being put into Stack A. My goal is to make sure I sort them from smallest to largest in no more than two actions. The way I determine which actions I should use depends on the position of the top number, middle number, and bottom number. With every case, I compare top to middle, middle to bottom, and bottom to top. Depending on which number is bigger or smaller will affect which actions I call.



5 random numbers:

Now we have to deal with 5 random numbers getting put into Stack A. We are only allowed 12 actions. Anything beyond that will fail this section of the project. What's awesome about this is that we can use the logic from **3 Random Numbers** to optimize our code. All we have to do is just move the first two numbers from the top of Stack A and move them to Stack B. We'll bring those numbers back once the three numbers in Stack A are sorted from smallest to largest. I'm going to apply the test case **1 5 2 4 3** and show how my logic will work.





Step 4–6: Makes sure Stack A can correctly accept the numbers from Stack B.

In total, you'll use 7 action points. Way below our max limit of 12.

100 random numbers:

When I originally started handling 100 random numbers, I figured I could just use the Insertion Sort algorithm to finish the project. The steps were simple:

1. Find the smallest number in Stack A.-
2. Move the smallest number found to the top of Stack A.
3. Push that number to Stack B.
4. Repeat steps 1–3 until Stack A is empty.
5. Push everything back to stack A once Stack B has all the numbers from biggest to smallest.

By sending the smallest number one at a time from Stack A to Stack B. I was able to make Stack B hold all the numbers from biggest to smallest. Then when I sent all the numbers back to Stack A. They will be sorted from smallest to biggest.

This approach *technically worked*… I mean, it *is* an algorithm and it *did* sort all the numbers… But it was far from being optimized enough to pass this project.

In order to optimize my idea, I realized I was sorting 1 large chunk of 100 numbers. I could cut down on my actions by sorting 5 smaller chunks of 20.

Let's say I have a random list of 100 numbers from 0–99.

Chunk 1 is 0–19

Chunk 3 is 40–59

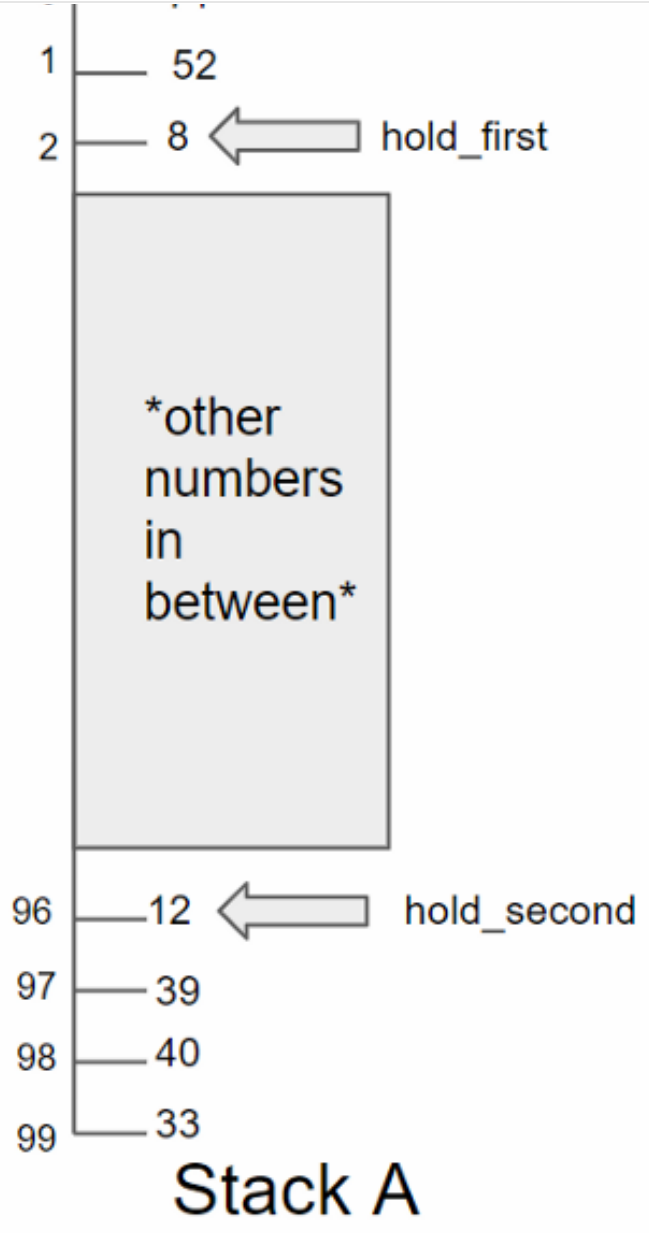
Chunk 4 is 60–79

Chunk 5 is 80–99

Step 1: Scan Stack A from the top to confirm if one of the numbers from Chunk 1 exist inside of it. Let's call that number **hold_first**.

Step 2: Scan Stack A again from the bottom and see if a different number from Chunk 1 exists in that list. I'll call that number **hold_second**.

Step 3: Compare how many moves it would take to get hold_first and hold_second to the top.



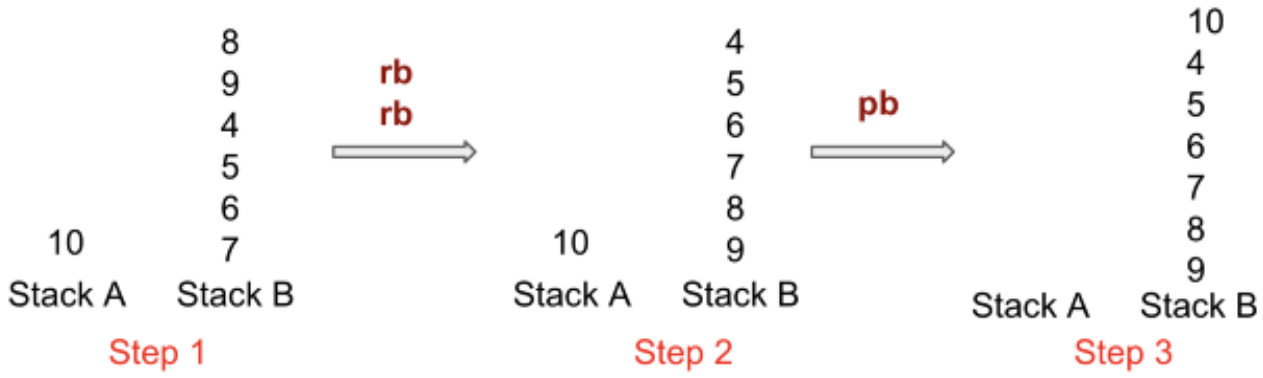
Here's an example of two numbers found in Stack A. We see 8 and 12 are both numbers inside of Chunk 1 (0–19) and they're inside of Stack A. After comparing both numbers, we see that in order to get 8 to the top of the list, we would have to run the **ra** command 2 times. But in order to get the number 12 at the top of the list, we would need to run the **rra** command 4 times. Because 8 takes fewer moves, we prioritize 8 and send it to the top of Stack A.

Before we move on to Step 4, I want to explain how I determine if those two numbers need to use **ra** or **rra**. For that, I just use basic math.

First I take the total amount of numbers in the list which is 100. Divide that number by 2, which gives us 50 which would represent the middle of the list. Then I find the place

the list). So I take the location of 8 and I try to figure out if the location of the number is greater than or less than 50. Because 2 is less than 50, I know it would be quicker to run ra on the number 8 instead of rra. But the number 12 is in the 96th spot. Because 96 is greater than 50, I know I have to use rra on it.

Step 4: So the correct number is now on to top of Stack A. But there are two things we need to check for before we push the number over to Stack B. You have to check if the number you're pushing is either bigger or smaller than all the other numbers in Stack B. Programming 42 Silicon Valley Programming Tutorials Insertion Sort Algorithms make sure we're not gonna cause an infinite loop by trying to find the perfect spot to insert that number.



In the above example, the quickest way to get the number 10 in the correct spot is to make sure that we get the smallest number (4) and place it on the top of Stack B. After that we can move the number 10 to the top of Stack B.

Regardless of the number at the top of Stack A being bigger or smaller than all the numbers in Stack B. It's still good to check if you're placing the number in the correct spot before sending it over.

Step 5: Repeat steps 1–4 until all the numbers in Chunk 1 no longer in Stack A.

Step 6: Repeat steps 1–4 for the rest of the chunks so they are handled in the same way and all of Stack A is inside of Stack B.

Step 7: Now that Stack A is empty, find the biggest number in Stack B, move it to the top, and push it over to Stack A. Repeat this until Stack B is empty. You can use the logic from Step 3 to determine if you need to use rb or rrb to quickly get the numbers to the

Your Stack A should now have all the numbers ordered from smallest to biggest.

500 random numbers:

So this will be easy.

Just use the exact same logic from 100 random numbers. But instead of splitting it into 5 chunks, just split it into 11 chunks. Why 11?

11 chunks are what I decided to use after running several tests on it. The range of action poi I got was way less than other numbers I tested it on.

Tools you can use to help:

I can't recommend this push_swap visualizer enough. This tool was made for push_swap an it really helped me optimize my code.

Click here to view the GitHub account for the visualizer.