Push_Swap: an Efficient Positional Sorting Algorithm

By Mia Combeau In 42 School Projects June 24, 2022 24 Min read Add comment

Push_swap is a vexing project to many 42 school students. The goal is to create a program in C that prints out the most optimized series of instructions to sort a random list of integers. For this project, we will need to implement an efficient sorting algorithm that respects the constraints the subject imposes on it.

The algorithm we will build here is one of many solutions to this problem. It hinges on an indexing and positioning system that makes it both very efficient and approachable.

This is not a step-by step tutorial and there will be no ready-made solutions. It is a guide to explore the issues and concepts behind the subject, one possible approach we might take, and some tips to test our code.

Read the subject [pdf] at the time of this writing.

Table of Contents

The Push_Swap Rules

Push_swap's subject sets pretty strict rules. Our program must receive a series of random positive and negative integers as arguments. It should print "error" if there are any duplicate numbers or if one of them exceeds the limits of an integer.

If the series of numbers is error-free, our program must print a series of instructions, each one on a new line, to sort the numbers in ascending order.

We are allowed two stacks for sorting purposes, stack A and stack B:

- stack A is initially filled with all the numbers to be sorted. The first number of the stack is the top one.
- stack B starts off empty.

List of Possible Push_Swap Actions

We will have to juggle between these two A and B stacks in order to sort the numbers of stack A in ascending order, i.e. from smallest on top to biggest at the bottom. For that, we have a limited number of possible actions: push, swap, rotate and reverse rotate.

Push

`pa` (push A): Take the first element at the top of stack B and put it at the top of stack A. Do nothing if B is empty. For example:

```
A : 1 3 4
B : 5 9
pa
A : 5 1 3 4
B : 9
```

`pb` (push B): Take the first element at the top of A and put it at the top of B. Do nothing if A is empty. For example:

```
A : 0 9 2
B : 1 4
pb
A : 9 2
B : 0 1 4
```

Swap

`sa` (swap A): Swap the first 2 elements at the top of stack A. Do nothing if there is only one or no elements. For example:

```
A : 8 3 9
sa
A : 3 8 9
```

`sb` (swap B): Swap the first 2 elements at the top of stack B. Do nothing if there is only one or no elements. For example:

```
B : 6 5 7
sb
B : 5 6 7
```

`ss`: `sa` and `sb` at the same time. For example:

```
A : 2 1 3
B : 5 0
ss
A : 1 2 3
B : 0 5
```

Rotate

`ra` (rotate A): Shift all elements of stack A up by 1. The first element becomes the last one. For example:

```
A : 9 2 5 8
ra
A : 2 5 8 9
```

`rb` (rotate B): Shift all elements of stack B up by 1. The first element becomes the last one. For example:

```
B : 7 3 4 6
rb
B : 3 4 6 7
```

`rr`: `ra` and `rb` at the same time. For example:

```
A : 8 0 1 2 3
B : 9 5 6
rr
A : 0 1 2 3 8
B : 5 6 9
```

Reverse rotate

`rra` (reverse rotate A): Shift all elements of stack A down by 1. The last element becomes the first one. For example:

```
A : 1 2 4 9 0
rra
A : 0 1 2 4 9
```

`rrb` (reverse rotate B): Shift all elements of stack B down by 1. The last element becomes the first one. For example:

```
B : 8 3 4 6 1
rrb
B : 1 8 3 4 6
```

`rrr` : `rra` and `rrb` at the same time. For example:

```
A : 7 6 4 9 2
B : 8 5 3 1
rrr
A : 2 7 6 4 9
B : 1 8 5 3
```

Push_Swap's Grading System

During the evaluation of the project, our grade depends on how efficient the actions our push_swap program generates are for sorting the numbers. At the time of this writing, this is the rating scale:

- Sorting 3 values: no more than 3 actions.
- Sorting 5 values: no more than 12 actions.
- Sorting 100 values: rating from 1 to 5 points depending on the number of actions:
  - 5 points for less than 700 actions
  - 4 points for less than 900 actions
  - 3 points for less than 1,100 actions
  - 2 points for less than 1,300 actions
  - 1 point for less than 1,500 actions
- Sorting 500 values: rating from 1 to 5 points depending on the number of actions:
  - 5 points for less than 5,500 actions
  - 4 points for less than 7,000 actions
  - 3 points for less than 8,500 actions
  - 2 points for less than 10,000 actions
  - 1 point for less than 11,500 actions

In order to validate this project, we need a grade of at least 80%.

Several Possible Methods for Push_Swap

The first thing to realize with push_swap is that the program itself doesn't need to be extremely optimized, only its output does. This means that we could very well sort our stack of numbers multiple times behind the scenes in order to choose the most efficient action to print out.

Despite the restrictions imposed in the push_swap subject, there are a multitude of ways to design a sorting algorithm. One of the most popular methods is the radix sort, which uses bitshifting and bitwise operators. Leo Fu describes it here. Another method, described here by Jamie Dawson, involves breaking the numbers up into chunks.

In this article, we will try a different very optimized algorithm that sorts by index and position.

Setting Up a Positional Sorting Algorithm for Push_Swap

This algorithm is very efficient: its results are well below the limits that the push_swap evaluation imposes. Correctly implemented, a grade of 100% is guaranteed. In addition, it is relatively easy to understand and implement.

Setting Up Linked Lists

We will use linked lists to implement this algorithm. Each element in a list will contain several variables:

```
typedef struct s_stack
{
	int				value;
	int				index;
	int				pos;
	int				target_pos;
	int				cost_a;
	int				cost_b;
	struct s_stack	*next;
}	t_stack;
```

- **value**: the integer we must sort,
- **index**: its index in the list of all values that must be sorted,
- **pos**: its current position in its stack,
- **target_pos**: for elements in stack B, the target position in stack A where it should be,
- **cost_a**: how many actions it would cost to rotate stack A so that the element at the target position gets to the top of stack A,
- **cost_b**: how many actions it would cost to rotate stack B so that this element gets to the top of stack B,
- **next**: pointer to the next element in the list.

The importance of all these variables will become clearer throughout the description of the algorithm.

Assigning Indexes

Once we've filled our stack A with the values to sort, we need to assign each element an index, from smallest to biggest. For practical reasons, the index will start at 1 and end at the total number of values to sort. For example, let's say we have a list of 10 numbers to sort, we will assign an index to each like this:

Value	Index
1900	9
42	8
18	7
-146	2
-30	3

2 147 483 647	10
3	6
0	5
-2 147 483 648	1
-2	4

Attributing indexes this way will make it much easier to know the order in which the random numbers should be.

Choosing a Sorting Algorithm Depending on The Number of Values to Sort

Most other push_swap algorithms have distinct sorting methods for 5, 100 and 500 numbers. This isn't the case here. What works for 500 numbers also works for 5. However, we still need a different algorithm for a 3 number sort.

But before we even choose between two sorting methods, it's a good idea to check if stack A is already sorted, just in case. If it is, push_swap's job ends right here.

However, if stack A is not sorted, we need to proceed differently depending on the number of values we need to sort:

- 2 values: all we need to do is `sa`.
- 3 values: jump to the 3 number algorithm
- > 3 values: jump to the more-than-three numbers algorithm

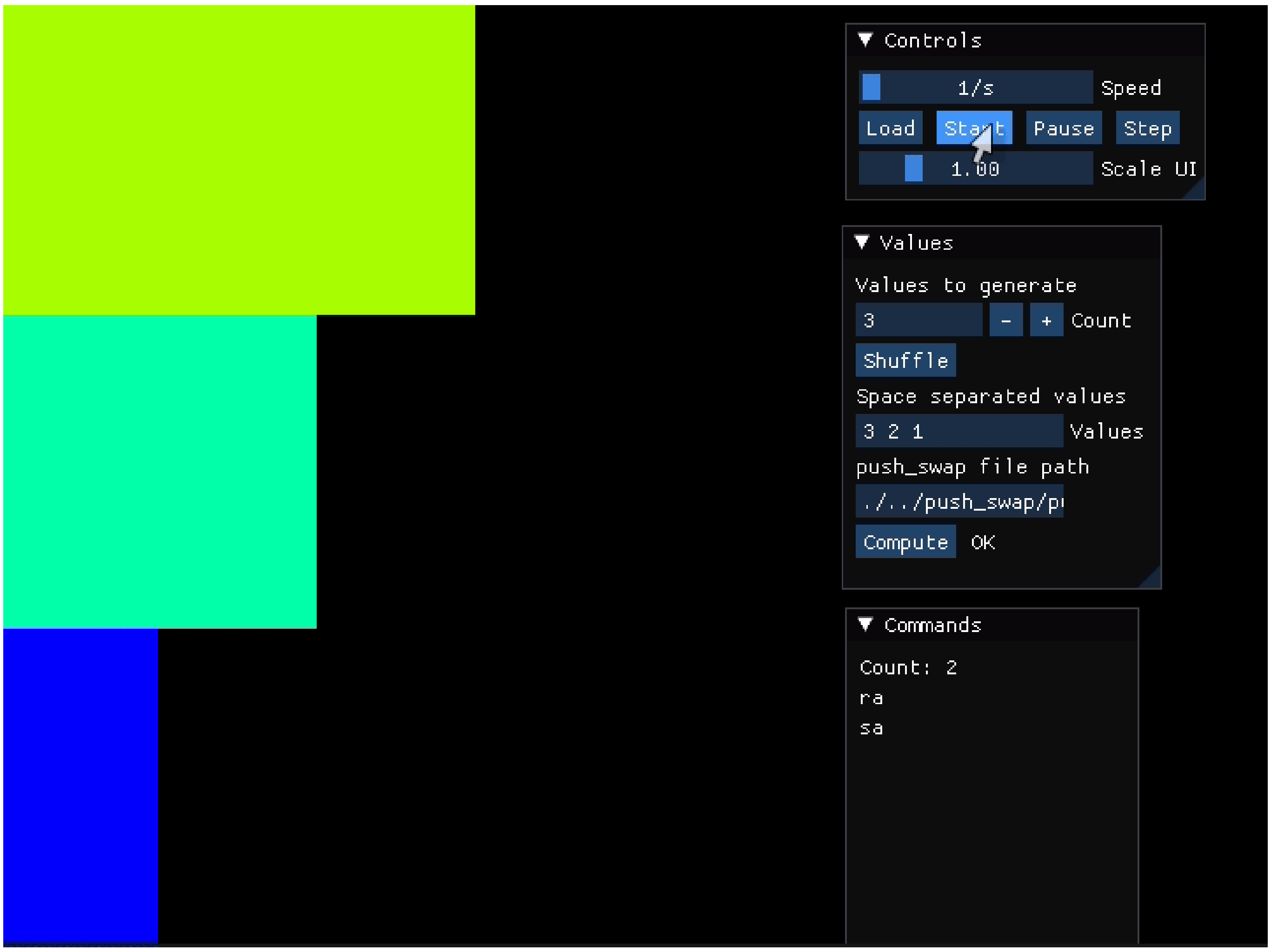
Sorting Algorithm For 3 Numbers

As per the grading scale, we should be able to sort 3 numbers with 3 or fewer actions. Here, we will never use more than two. For 3 values, there are six possible cases:

```
Case                actions
1 2 3               => no action
1 3 2               -> rra -> 2 1 3 -> sa  -> 1 2 3 => 2 actions
2 1 3               -> sa  -> 1 2 3 => 1 action
2 3 1               -> rra -> 1 2 3 => 1 action
3 1 2               -> ra  -> 1 2 3 => 1 action
3 2 1               -> ra  -> 2 1 3 -> sa  -> 1 2 3 => 2 actions
```

Our little 3 number sorting algorithm needs only pick one of three actions: `ra`, `rra` and `sa`, depending on the biggest number's position. We can sum up the simple algorithm this way:

- If the index of the **first number is highest**, do `ra`,
- Otherwise, if the index of the **second number is highest**, do `rra`,
- Then, if the index of the **first number is bigger than the index of the second number**, do `sa`.



```
Count: 2
ra
sa
```

Visualization of a three number sort with o-reo's visualizer.

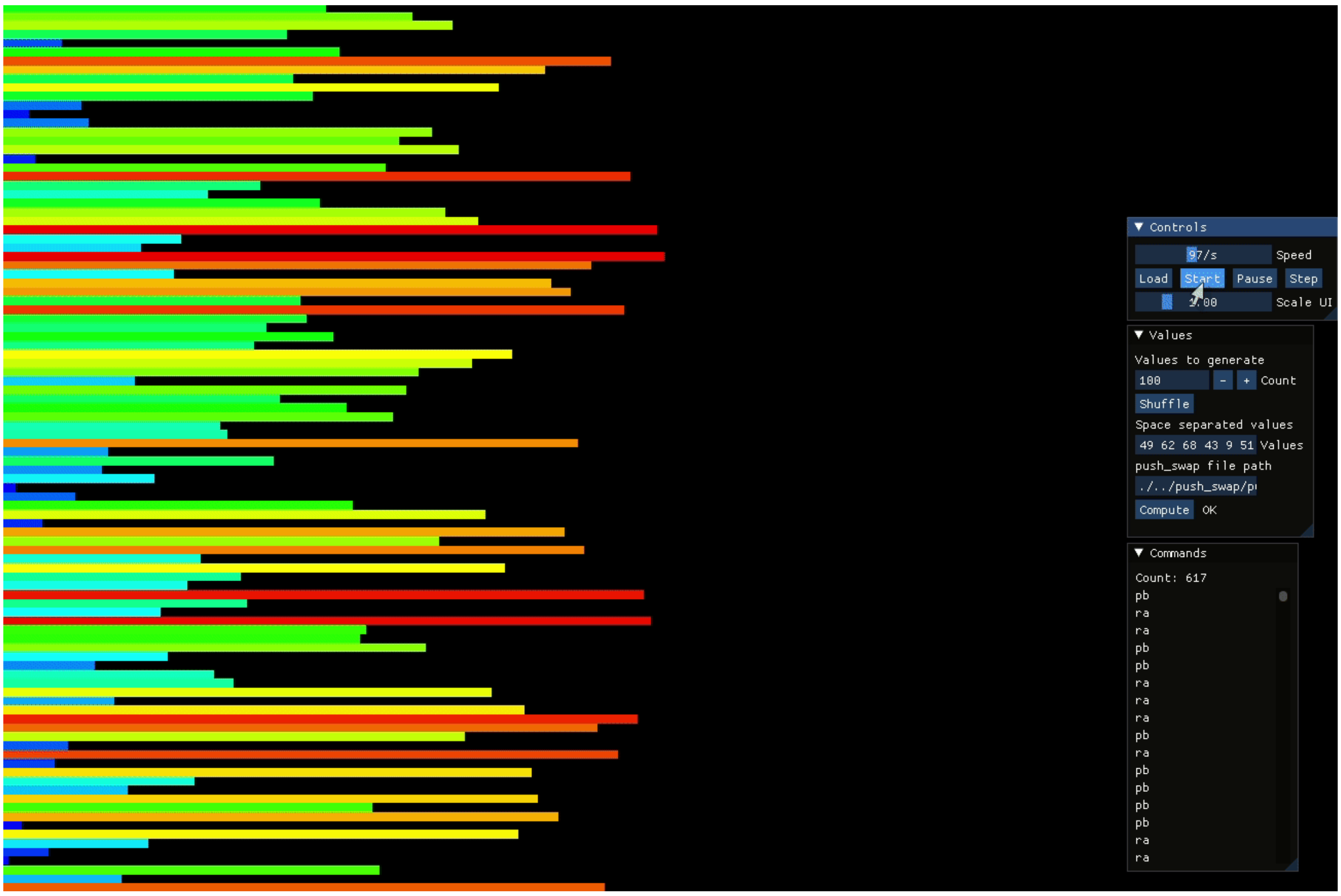
We don't use stack B at all here: a `pb` to push a number from stack A to stack B, followed by a `pa` to send it back to stack A would be a waste of actions !

Sorting Algorithm For Over 3 Numbers

For a larger series of numbers to sort, we will need to compute more actions. In order to decide which move to do next to sort each value, we need to calculate the position of each element in its stack. Then we can compare the number of actions it would take to get each value at the top of its stack. Only then can we chose the cheapest series of actions.

Our sorting algorithm will have 8 steps, provided, of course, that stack A is not already sorted:

1. **`pb` all elements** from stack A, **except three**.
2. **Sort the 3 numbers left in stack A**.
3. Loop as long as there are elements in stack B:
    1. **Find the current position** of every element in stack A and B.
    2. **Calculate the target position** in stack A where each element in stack B should be.
    3. **Calculate the number of actions** (the cost) to put each element in stack B at its target position in stack A and **choose the element that's cheapest to move**.
    4. **Execute the sequence of actions** needed to move the element from stack B to stack A.
4. If stack A is not sorted, chose between `ra` and `rra` to rotate it into ascending order.



```
97/s Speed
1.00 Scale UI
Values to generate
100 Count
Space separated values
49 62 68 43 9 51 Values
push_swap file path
./../push_swap/pu
OK
Count: 617
pb
ra
ra
pb
pb
ra
ra
ra
pb
ra
pb
pb
pb
pb
ra
ra
```

Visualization of the 100 number sort with o-reo's visualizer.

In this visualization, we can see stack A to the left and stack B to the right. The color blocks represent the random numbers that we must sort. We start by pushing all the values but three to stack B. As push_swap gives its instructions, we can see the values of stack B being correctly inserted back into stack A. In the end, stack A must rotate to be in ascending order.

Step 1: Send Everything to Stack B

The first step is to send all the elements of stack A to stack B, except 3. We keep three elements in stack A to avoid extra actions. After all, we already have a simple algorithm that we can use to sort 3 numbers directly in stack A.

However, pushing all our elements to stack B doesn't mean we can't do it in a somewhat ordered way. In fact, thanks to our indexing system, we can already do a crude 2 step sort.

First, we can push all of the smaller values. So, if an element has a smaller index than the middle index of all the elements (the total number of values to sort divided by 2), we can push it to stack B. Otherwise, we rotate A. After that, we can freely `pb` all the rest of the elements except for the last three that will remain in stack A.

This way, stack B will already be vaguely sorted, which will reduce the number of actions we will have to do later.

Step 2: Sort the 3 Numbers Left in Stack A

Chances are, the three elements we kept in stack A will need to be sorted. Thankfully, we have our little 3 number sorting algorithm to help out.

With these three values in order, all we need to do is insert each element in stack B at the right position in stack A. But finding the most efficient sequence of actions is a little more complicated than that. Let's see how we can manage it.

Step 3: Calculating Positions

A key step in our push_swap algorithm is to find the position of each element in its stack, as well as the target position in stack A where an element of stack B should go. With this positional information, we will be able to calculate the cheapest sequence of actions and choose which element to move first.

Finding the Position of Each Element

An element's position is simply a representation of where it currently is in its stack. Let's take a random list of four values:

```
value:          3       0       9       1
index:         [3]     [1]     [4]     [2]
position:      <0>     <1>     <2>     <3>
```

All we need to do to find the position of each element is scan the stack from top to bottom, and assigning each element a position that we increment at each iteration. We must find the positions of each element in both stacks this way. Of course, we will have to update these positions often, every time we want to calculate the most efficient sequence of actions.

Finding the Target Position of Each Element in Stack B

For each element in stack B, we need to calculate a target position. **The target position** is where an element of stack B needs to be in stack A. More precisely, it's the position of the stack A element that must be at the top stack A when we push this element from B to A.

We can spot this target position simply by scanning stack A for the closest superior index to the one of the stack B element. The position of the element in stack A with that index will be the target position of our stack B element.

Example of Finding Target Positions

```
Stack A contains
        value:          8       0       1       3
        index:         [6]     [1]     [2]     [4]
        position:      <0>     <1>     <2>     <3>

Stack B contains
        value:          2       6       9
        index:         [3]     [5]     [7]
        position:      <0>     <1>     <2>
        target pos.:   (3)     (0)     (1)
```

Here, the element at the top of stack B has a value of 2 and an index of 3. This means that it is the third value in ascending order of all the numbers that need sorting (0, 1, 2…). Since this element is at the top of pile B, we could push it to the top of pile A. But it wouldn't be at the right spot. This element needs to be between indexes 2 and 4. If the element with index 4, currently at the third position in stack A was at the top of its stack, we could safely push our element from stack B on top of it. So the target position of this first element in stack B is the position of the element with index 4 of stack A, which is 3.

If an element in stack B has a superior index to all of the elements in stack A, as is the case with the last element of stack B in our example, we need to use the smallest index's position as its target position. This is because we want the smallest index of stack A to be at the top before we push our higher index onto it. This ensures that the largest value will be at the bottom of stack A after we rotate it.

Why Calculate Positions Like This ?

With these positional values, we will be able to determine how many actions it will take to get each stack in the right position for each element. Then we can figure out which element of stack B is cheapest to move first.

Obviously, we only need to calculate target positions for elements in stack B. Elements in stack A don't need a target position since they are (hopefully) already in the right position.

Of course, several B elements will have the same target position, especially when stack B has more elements than stack A. Their position in B will be the determining factor, cost-wise.

Step 4: Calculating the Cheapest Action Cost

Once we've determined the positions and target positions of every element in both stacks, we will be able to compare their costs in terms of number of actions it takes to put each element at the top of B and at the right spot in stack A.

Calculating Costs for Stack A and B

For each element in stack B, we need to calculate two costs. The first is the cost to bring the element to the top of B. The second cost calculation is the same, but for pile A, in order to determine how many actions we need to move the element of the target position up to the top of stack A.

In both cases, we need to distinguish between rotate (**`ra`** or **`rb`**) and reverse rotate (**`rra`** or **`rrb`**) in order to correctly calculate the costs. As a reminder, rotate moves the top element to the bottom of the stack, while reverse rotate does the opposite, bringing the bottom element to the top of the stack.

Distinguishing between rotate and reverse rotate

To determine if we need to reverse rotate rather than simply rotate, we can measure the size of the stack and divide it in half. This way, it's easy to check if the position of our element is in the top or bottom half of the stack. If the element's position is smaller than the middle position of the stack, it is in the top half, which means we need to do **`rb`**, otherwise, we will have to go with **`rrb`**.

It would be very useful to make the reverse rotate cost negative. That way, during the execution of the action sequence, we will be able to check the cost's sign to know whether to rotate or reverse rotate. And if the costs of both A and B are the same sign, we can further save on actions by choosing to do **`rr`** or **`rrr`**, which will rotate or reverse rotate both stacks simultaneously.

Finally, we can compare the cost of moving each element of stack B to stack A to choose the cheapest. This choice is simple, all we need to do is add the costs for stack A and stack B for each element in stack B. Rather, we need to add the absolute values of the costs so that the addition stays true even with the reverse rotate negative cost values.

Why Not Use Swap?

This algorithm relies on calculating the positions of each element in a stack in order to pick which one to move first. This implies using primarily rotate and reverse rotate actions to get each stack in position. We will not be using swaps (`sa`, `sb` or `ss`) ; these actions are not optimized because they don't change the positions of all the elements in the stack. After an `sb + pa`, for example, we are back at the same element as before at the top of our stack, which does not move our push_swap forward.

Examples of Cost Calculation

Let's take our earlier example of the position calculations and modify it a little to illustrate the cost calculation.

```
Stack A contains
        value:          8       0       1       3       4
        index:         [7]     [1]     [2]     [4]     [5]
        position:      <0>     <1>     <2>     <3>     <4>

Stack B contains
        value:          2       6       9
        index:         [3]     [6]     [8]
        position:      <0>     <1>     <2>
        target pos.:   (3)     (0)     (1)
```

The first element of stack B would like to be slotted in before the element in 3rd position of stack A. Since this element is already at the top of B, it's cost for B will be 0. However, there will be a cost to move its target from 3rd position to the top of stack A. Since the target element is in the bottom half of stack A, we need two `rra` to get it to the top. This means that its cost for A is -2. The total cost of moving the first element of B is `(abs)0 + (abs)-2 = 2`.

The second element of stack B needs to be at position 0 of pile A. We need to do a `rb` for it to be at the top of stack B, and pile A is already in the correct position. The cost to move this element in B is 1 and the cost to move stack A into position is 0. The total cost to get this element to the proper place is `(abs)1 + (abs)0 = 1`.

Finally, we have one last element in pile B. Since it is in the bottom half of its pile, we need an `rrb` to get it to the top. So its B cost will be -1. Its target position is the second element in stack A. We will need to do an `ra` to get it to the top of stack A, which means the A cost is 1. The total cost to move this last B element is `(abs)-1 + (abs)1 = 2`.

In this case, it's clear that the best choice that our push_swap algorithm could make right now is to move the second element of stack B.

Step 5: Executing the Chosen Sequence of Actions

Once we've found the cheapest element in stack B, we need to execute the action sequence that will push it to the top of stack A.

Since we've saved this element's A and B costs in two separate variables, we can use them to figure out how many actions we need to execute in each pile. Plus, we made it so a reverse rotate cost is negative unlike the rotate cost. So we can easily distinguish between these two actions to execute them perfectly. We can also save on actions by doing an `rrr` instead of an `rra` and an `rrb` separately, for example.

Let’s take an element with an A cost of 2 and a B cost of 3, for example. Both are positive, so we know we have to rotate and not reverse rotate. Plus, we can clearly see that we can just do two `rr` and one `rb`. For an element with an A cost of -4 and a B cost of 1, we will have to do four `rra` and one `rb`. However, if the A cost if -3 and the B cost is -1, we can do one `rrr` and two `rra`.

With the elements in stack A and stack B in position, we can finally **`pa`**. And, as long as there still are elements in B, we go back to step 3 to reevaluate all the positions and all the costs for the next sequence of actions.

Step 6: Rotate Stack A to the Right Position

When there are no more elements in stack B, that doesn’t mean that stack A is totally in order ! That depends on which element was last placed from stack B. We can’t forget to adjust stack A with `ra` or `rra` until the element with the smallest index is at the top and the element with the largest index is at the bottom.

There ! Stack A should now be correctly sorted in ascending order, and our push_swap program should have printed out the smallest number of instructions to make it so.

Tips to Test Push_Swap

Before we even implement the algorithm, it’s important to check that the linked list and that the functions for each action (`pa`, `pb`, `sa`, `sb`, `ss`, `ra`, `rb`, `rr`, `rra`, `rrb` and `rrr`) work correctly. For that, we need to print the values or indexes of all the elements in a stack in a loop to check the results.

We also need to check that our push_swap program is correctly detecting bad input. In particular, we should be vigilant about duplicate numbers, like 0, +0, -0, 00000 or even 1, +1, 00001 and 1.0. We should also be careful to check that no numbers exceed INT_MAX or INT_MIN.

Testing Push_Swap with the Checker

42 provides us with a checker to help with testing our push_swap. In order to make use of it, we have to execute it with a list of numbers in a shell variable, like this:

```
ARG="1 -147 2 89 23 30"; ./push_swap $ARG | ./checker_linux $ARG
```

The checker only prints `OK` or `KO`, depending on whether the stack was correctly sorted or not. To see how many actions it took to get the sorting done, we can just use the wc command:

```
ARG="1 -147 2 89 23 30"; ./push_swap $ARG | wc -l
```

Testing Push_Swap with the Shuf Command

That was useful for small series of numbers, but it's far from ideal for 100 or 500 numbers! To test those, we can use the shuf command, which shuffles a series of integers in a random way. Let's note however that when we generate a list of random numbers with shuf, there will never be any negative numbers. To generate 500 numbers between 0 and 1000 with the shuf command, we can do this:

```
ARG=$(shuf -i 0-1000 -n 500); ./push_swap $ARG | ./checker_linux $ARG
```

Thanks to this command, we can even add some Makefile rules for quick testing:

```
test3:                $(NAME)
                      $(eval ARG = $(shell shuf -i 0-50 -n 3))
                      ./push_swap $(ARG) | ./checker_linux $(ARG)
                      @echo -n "Instructions: "
                      @./push_swap $(ARG) | wc -l

test5:                $(NAME)
                      $(eval ARG = $(shell shuf -i 0-50 -n 5))
                      ./push_swap $(ARG) | ./checker_linux $(ARG)
                      @echo -n "Instructions: "
                      @./push_swap $(ARG) | wc -l

test100:              $(NAME)
```