

Push_swap hell

computer scientist

The theory of the various files and how they work:

push_swap.c file:

Mainly the main here it is in all its splendor:

```
#include "push_swap.h"

int main(int ac, char **av) {

    t_stack *stack;

    if (ac < 2)
        return (0);
    stack = init(ac, av);
    ft_sorter(stack);
    ft_exit(stack);
    return (0);
}
```

Very simple, in reality there is not much to explain other than the fact that it is used with `av` and `ac`, inside we declare the variable `t_stack` stack an important variable to manage the two stacks, in fact the variable `t_stack` is declared as a structure in the `push_swap` library. `h` and is structured like this:

```
typedef struct s_stack
{
    int *a;
    int *b;
    int last_a;
    int last_b;
} t_stack;
```

Where variables are handled like this:

`a` and `b` are two arrays of integers, in short, our stacks.

`last_a` and `last_b` are the membership index of the last value in their respective stacks.

IMPORTANT:

Why do we set a size for our stacks knowing that their memory is expanded by a certain amount? well it's very simple because compared to arrays of strings (with which we have worked so far) you cannot make a while loop of this type:

while(a[i]), and you know why well because C would recognize the value 0 as null i.e. end of the array and we don't want this also because the 0 could be anywhere on the stack. since it's a stack of numbers and not a string

Having said this, returning to main, we basically check the number of arguments with possible exit in the case of an insufficient number, then we fill our variable with the `init` function which will be explained later and finally we free the memory of all the stacks thanks to the function `ft_exit` which will run some frees to avoid leaks.

init.c

The file that sets up our code, what provides us with all the materials to build our stacks is the init.c file.

Let's go through the actual order of functions:

void fill_stack(int ac, char **av, int i, t_stack *stack):

```
void fill_stack(int ac, char **av, int i, t_stack *stack) {

    int j;

    stack->a = (int *)malloc(sizeof(int) * (ac - 1)); stack->b = (int *)malloc(sizeof(int) * (ac - 1)); j = 0;

    while (i < ac)
        stack->a[j++] = ft_atoi(av[i++], stack->a); find_dup(stack->a, j);
    stack->last_a = j;
    stack->last_b = 0;
}
```

This function allows us to fill and complete the stack by initializing the two stacks with the malloc and filling the stack_a with the numerical values obtained from the terminal and stored in av thanks to the ft_atoi function (modified for the eventuality).

The secret behind the initialization is to stack with ac - 1 (we don't count the name of the program, right??????).

then on a whim we check the duplicate numbers with the find_dup function (which I will explain later)

I will then set the index of the last value inserted (which in this case is j) as the size of stack_a.

the variables passed are ac and av (no need to explain) the stack that must be filled and a value i which starts from 1 (skips the name of the program in av).

t_stack *init(int ac, char **av):

This function is the heart of the initialization of our stack and is done like this:

```
t_stack *init(int ac, char **av) {

    char    ** tmp;
    int     len;
    t_stack *stack;

    tmp = NULL;
    len = 0;
    if (ac == 2)
    {
        tmp = ft_split(av[1], ' '); while
        (tmp[len] != NULL)
            len++;
        stack = malloc(len * sizeof(t_stack)); fill_stack(len,
        tmp, 0, stack); free(tmp);

    }
    else if (ac >= 3) {

        stack = malloc((ac - 1) * sizeof(t_stack)); fill_stack(ac, av, 1,
        stack);
    }
    else
        return (NULL);
    return (stack);
}
```

I'll try to be concise:

we need the tmp variable to have the values taken from av (saved only on a memory line in case the values are inserted between " " in the command line) and save them thanks to the ft_split function inside the tmp matrix.

len (in case you need to split the argument between " " you will need to mark how large it is to define last_a and last_b.

t_stack *stack (doesn't even need to be said).

the first if occurs if we have two arguments, which in our case will be the name of the program plus the numbers between " " (very likely).

in sauce the argument between " " is taken and is split in such a way as to divide one number at a time also defining with the

malloc the size of the stack, then itself will be filled with the fill_stack function.

the second if is in case there are more than two arguments we define the size of the stack and then fill it, we don't have to do any operations the numbers are already divided.

the else is in case there are not enough arguments and therefore it will end returning NULL


After the various checks we will return the initialized stack.

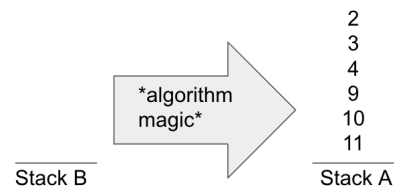
function void three_sorting(t_stack *stack)

I created it using the explanation at the following link.

Push_Swap: The least amount of moves with two stacks

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...

 <https://medium.com/@jamierobertdawson/push-swap-the-least-amount-of-moves-with-two-stacks-d1e76a71789a>



I won't dwell on the explanations since we mainly evaluate each case to use as few moves as possible with three numbers, here's the code anyway:

```
void three_sorting(t_stack *stack) {  
  
    if ((stack->a[0] < stack->a[1]) && (stack->a[0] < stack->a[2])  
        && (stack->a[1] > stack->a[2]))  
    {  
        ft_sa(stack);  
        ft_ra(stack);  
    }  
    else if ((stack->a[0] > stack->a[1]) && (stack->a[0] > stack->a[2])  
        && (stack->a[1] > stack->a[2]))  
    {  
        ft_sa(stack);  
        ft_rra(stack);  
    }  
    else if ((stack->a[0] < stack->a[1]) && (stack->a[0] > stack->a[2])  
        && (stack->a[1] > stack->a[2]))  
        ft_rra(stack);  
    else if ((stack->a[0] > stack->a[1]) && (stack->a[0] > stack->a[2])  
        && (stack->a[1] < stack->a[2])) ft_ra(stack);  
}
```

```

    else if ((stack->a[0] > stack->a[1]) && (stack->a[0] < stack->a[2])
        && (stack->a[1] < stack->a[2])) ft_sa(stack);

}

```

***int*ft_sorter(*t_stack* **stack*):**

Very simple function that chooses which sorting algorithm to do based on how many numbers are on the stack:

```

int ft_sorter(t_stack *stack) {

    if (!check_sorted_asc(stack->a, stack->last_a)) {

        if (stack->last_a == 2)
            ft_sa(stack);
        else if (stack->last_a == 3)
            three_sorting(stack);
        else
            quick_sort_a(stack, stack->last_a, 0);
    }
    return (0);
}

```

Very simple, ah obviously if the stack is already sorted it won't sort anything and I will terminate the program. more than 3 arguments will start the core function of the quick_sort_a program which I will explain (since it's a mess) last.

Checks.c

File that mainly contains functions that perform various checks to see and check whether the program can be executed or whether it can continue to be executed.

***void* print_error(*int* **stack*)**

the function is very simple and serves to send an error message and free the stack that had previously been stacked, then the program will be closed

```
void print_error(int *stack) {

    free(stack);
    write(2, "Error\n", 6); exit(0);

}
```

void find_dup(int *a, int dim)

The function looks like this:

```
void find_dup(int *a, int dim) {

    int i;
    int j;

    i = 0;
    j = 1;
    while (i < dim)
    {
        while (j < dim)
        {
            if (a[i] == a[j])
                print_error(a);
            j++;
        }
        i++;
        j = i + 1;
    }
}
```

i and j have a precise role, basically i points to a number and starts the while loop until it reaches the size of the array but why do I say point to a number? well because the second while the one where I scroll the array with j is specifically used to compare all the numbers following the one pointed by i in the array, this check will go on checking every single number pointed by i and starting from i + 1 with j

void ft_exit(t_stack *stack)

function to free the stacks and end the program:

```
void ft_exit(t_stack *stack) {
```

```

    free(stack->a);
    free(stack->b);
    exit(0);
}

```

intft_isdigit(int c)

old acquaintance from libft, I won't even explain it

```

int ft_isdigit(int c)
{
    if (c >= '0' && c <= '9')
        return (1);
    return (0);
}

```

intft_atoi(char *str, int *stack)

A variant of the source that does overflow checks by exiting the program with numbers that exceed the integer value

ATTENTION

This means that the error is normally handled as the overflow of the original atoi was handled as undefined behavior but in our case it is a requirement to check the error above INT_MAX and INT_MIN

```

int ft_atoi(char *str, int *stack) {

    long int    r;
    int         yes;
    int         the;

    r = 0;
    s = 1;
    i = 0;
    while (str[i] == 32 || (str[i] >= 9 && str[i] <= 13))
        i++;
    if (str[i] == '+')
        i++;
    else if (str[i] == '-') {

        s = -1;
        i++;
    }
    while (str[i])

```



```

{
    if (!ft_isdigit(str[i]))
        print_error(stack);
    r = r * 10 + str[i++] - '0';
}
if (r > 2147483647 || r < -2147483647)
    print_error(stack);
return (r * s);
}

```

So how does it work: we receive the string to convert into an integer and we also pass our stack, this is because in case there is an overflow we will free the memory directly in the atoi function.

The conversion is practically the same as the classic atoi, checking the spaces and 1 sign if there is one, with the addition of program interruption if a non-numeric value is found or in the case of overflow

Push_swap_utils.c

Before actually touching on the algorithm I will explain various helper functions that allowed me to finish the code and that helped a lot for checking while executing the quick_sort

void tmp_sort(int *tmp_stk, int dim)

This works creates a temporary array which thanks to bubble_sort is sorted in ascending order, why do I need this temporary array? well it is used to find the pivot and in fact supports the "partition" function which I will explain later

```

void tmp_sort(int *tmp_stk, int dim) {

    int i;
    int j;
    int tmp;

    i = 0;
    while (i < dim)
    {
        j = i + 1;
        while (j < dim)

```

```

    {
        if (tmp_stk[i] > tmp_stk[j]) {

            tmp = tmp_stk[i];
            tmp_stk[i] = tmp_stk[j];
            tmp_stk[j] = tmp;
        }
        j++;
    }
    i++;
}
}

```

int* **check_sorted_desc(int *stack,int dim)*

It allows me to check if my stack is sorted in descending order (from largest to smallest)

```

int      check_sorted_desc(int *stack, int dim)
{
    int i;

    i = 1;
    while (i < dim)
    {
        if (stack[i - 1] < stack[i])
            return (0);
        i++;
    }
    return (1);
}

```

The syntax is very simple, I start from the first value with i to always compare the previous one and stop once before, it is obvious that based on the previous number it will be established with an if whether the stack is in descending order or not.

***int*check_sorted_asc(int *stack,int dim)**

The opposite of the previous function

```

int check_sorted_asc(int *stack, int dim) {

    int i;

```

```

i = 1;
while (i < dim)
{
    if (stack[i - 1] > stack[i])
        return (0);
    i++;
}
return (1);
}

```

intwhen_push(t_stack *stack,int dim,int f)

function created only for the purpose of saving lines of code when creating the algorithm.

```

int when_push(t_stack *stack, int dim, int f) {

    if (f == 1)
        ft_pb(stack);
    else if (f == 0)
        ft_pa(stack);
    dim--;
    return (dim);
}

```

it is mainly performed with the flag (f) which defines which push needs to be done, obviously chosen based on the need, then the size will be altered which ATTENTION can be defined for both stack_a and stack_b but this will be explained later.

Quick_sort.c

And here I want you, file that contains the key piece to sort my stack

Compared to before I will go in order of priority and that is to say that the functions will not be listed from bottom to top I will go with the functions that are called first:

intquick_sort_a(t_stack *stack,int dim,int count_r)

A beautiful beast, the one who starts it all let me insert the code:

```
int quick_sort_a(t_stack *stack, int dim, int count_r) {

    int pivot;
    int numbers;
    //dim is a global number passed into the two quick_sorts to handle manipulation //of the stacks

    if (check_sorted_asc(stack->a, dim) == 1)
        return (1);
    numbers = dim;
    //final part (the last sort in short)
    //sorting of the first numbers on stack a at the end of the quick sort if (dim <= 3)

    {
        quick_sort_3_a_b(stack, dim); return (1);
    }
    if (!count_r && !partition(&pivot, stack->a, dim))
        return (0);
    //printf("dim %de numbers/2 %de numbers 2 %d\n", dim, numbers / 2, numbers % 2); while (dim != numbers /
    2 + numbers % 2) //why this???
    {
        if (stack->a[0] < pivot && (dim--))
            ft_pb(stack);
        else if (++count_r)
            ft_ra(stack);
    }
    while (count_r--)
        ft_rra(stack);
    return (quick_sort_a(stack, numbers / 2 + numbers % 2, 0)
        && quick_sort_b(stack, numbers / 2, 0)); return (1);
}
```

Let's go in order, the variables it receives:

t_stack *stack, the generic stack, in short both stack_a and b (we work with a memory struct).

int dim, the size of stack_a (in this case) will be manipulated during the execution of the function it will be used in many ways, it is the heart of the program

int count_r, as the name suggests this variable defines the number of rotate or reverse rotate to do (depending on the circumstances).

Let's move on to the variables:

int pivot, is the number that in a hypothetical ordered stack is in the middle of it (for example a stack of 1 3 7 9 11 21 will have 7 as a pivot).

int numbers, takes into account the dim value in case it is manipulated or changed during the code it can be treated as a flag for the size of my stacks

Ok now that we've got the hard part out of the way it's time for the code, so the first if checks if the array is sorted (in ascending order), but why this? well this function is rather particular, in fact it is recursive and I need this control to end it.

We have control over the dimension which is used to define whether or not to sort by 3 numbers or less (various cases are covered)

then we check that the pivot has been found or that the r moves are still to be made and so at the same time we find the pivot.

This while control is rather peculiar but once you think about it it makes sense

```
while (dim != numbers / 2 + numbers % 2)
```

Basically this check is used to determine if our stack has reached the value for which the numbers lower than the pivot have been pushed, but why the numbers % 2? well if our stack is of odd size we won't risk making a mistake (very clever who told me this).

in the while we check exactly all the numbers below the pivot and push them into stack_b otherwise we increase the number of ras we execute until we find the value lower than the pivot.

```
if (stack->a[0] < pivot && (dim--))
    ft_pb(stack);
else if (++count_r)
    ft_ra(stack);
```

immediately afterwards, for reasons of ordering, I rearrange the stack with the reverse rotates in such a way as not to further vary the number of moves that will be made

```
while (count_r--)  
    ft_rra(stack);
```

Finally I return by calling not one but TWO functions recursively in such a way that I also start executing the function that will sort the stack_b (obviously to execute quick_sort_b the quick_sort_a function must return at least once without recursion, mind you, this stuff messes with the brain)

```
return (quick_sort_a(stack, numbers / 2 + numbers % 2, 0)  
        && quick_sort_b(stack, numbers / 2, 0)); return (1);
```

Small detail: note that the sorting of b only takes numbers / 2 and without the % 2 this is because, if in the previous case we rounded, here we certainly do not take an extra number and therefore it does not keep the % 2

int quick_sort_b(t_stack *stack, int dim, int count_r)

Operation is similar with some small differences

```
int quick_sort_b(t_stack *stack, int dim, int count_r) {  
  
    int pivot;  
    int numbers;  
  
    if (!count_r && check_sorted_desc(stack->b, dim) == 1)  
        while (dim--)  
            ft_pa(stack);  
    if (dim <= 3)  
    {  
        sort_3_b(stack, dim);  
        return (1);  
    }  
    numbers = dim;  
    if (!partition(&pivot, stack->b, dim))  
        return (0);  
    while (dim != numbers / 2)
```

```

{
    if (stack->b[0] >= pivot && dim--)
        ft_pa(stack);
    else if (++count_r)
        ft_rb(stack);
}
while (numbers / 2 != stack->last_b && count_r--)
    ft_rrb(stack);
return (quick_sort_a(stack, numbers / 2 + numbers % 2, 0)
        && quick_sort_b(stack, numbers / 2, 0));
}

```

We can say that the differences are these:

unlike quick_sort_a, if it is sorted in a DECRESENTLY way, it will not return anything but rather will push all its contents into a, if the size is less than or equal to 3, a sorting takes place such that the stack_b is sorted in a descending way.

if ONLY the partition function doesn't work return a. look for numbers / 2 in the while for the reason explained above. and the check is the mirror image of stack_a, I look at the BIGGER numbers of my pivot found in stack_b and push them back, otherwise rb, the return is the same because stack_b must also make use of recursion to work (but it will only stop when it cannot no longer calculate a pivot and alternates with stack_a by calling it first, so that it always has numbers unless stack_a is already sorted) in short the two create such a synergy that by doing a recursion the sorting will take place in blocks that are continuously divided thanks to a continuous calculation of the pivot and the passing of the values between a and b.

intsort_3_b(t_stack *stack, int dim)

particular function that works with the dimension (which is split and managed by both stacks).

```

int sort_3_b(t_stack *stack, int dim) {

    if (dim == 1)
        ft_pa(stack);
    else if (dim == 2) {

```

```

    if (stack->b[0] < stack->b[1])
        ft_sb(stack);
    while (dim--)
        ft_pa(stack);
}
else if (dim == 3) {

    while (dim || !(stack->a[0] < stack->a[1] && stack->a[1] < stack->a[2])) {

        if (dim == 1 && stack->a[0] > stack->a[1])
            ft_sa(stack);
        else if (dim == 1
            || (dim >= 2 && stack->b[0] > stack->b[1]) || (dim == 3 &&
            stack->b[0] > stack->b[2])) dim = when_push(stack, dim, 0);

        else
            ft_sb(stack);

    }
}
return (0);
}

```

The function is used to manage the peculiarity in stack_b of being sorted in a decreasing way by looking at the numbers to push or sort in multiple cases:

case dim 1 then stack b will be emptied by pushing the last number into
stack_a

case size 2

I look at the larger values and swap them or push them

case 3

in the first if I prepare stack_a to receive the value by sorting the first two values (note that the while will not stop until the size is 0 or until the first values of stack_a are sorted).

if not, I check that the values in b are ordered and push them or I have to exchange the first with the second which is certainly larger.

intpartition(int *pivot,int *stack,int dim)

function to find the pivot, i.e. the number that is exactly in the middle of my stack, uses the tmp_sort function to understand the average value in the stack and assigns it to the pivot variable

```
int partition(int *pivot, int *stack, int dim) {

    int *tmp;
    int i;
    int j;
    //the dim is global and applies to both stacks tmp = (int
    *)malloc(sizeof(int) * dim);
    if (!tmp)
        return (0);
    i = 0;
    j = 0;
    while (i < dim)
        tmp[j++] = stack[i++];
    tmp_sort(tmp, dim);
    * pivot = tmp[dim / 2];
    free(tmp);
    return (1);
}
```

void quick_sort_3_a_b(t_stack *stack, int dim)

Function that performs the final checks in stack_a and manages the various sorting cases of stack_a is like a distant brother of the function

then *tsort_3_b(t_stack *stack, int dim)*.

```
void quick_sort_3_a_b(t_stack *stack, int dim) {

    //check size and management of stack_a with three elements if (dim == 3 &&
    stack->last_a == 3)
        three_sorting(stack);
    //same thing with two elements else if
    (dim == 2)
    {
        if (stack->a[0] > stack->a[1])
            ft_sa(stack);
    }
    //case management of the first numbers in stack_a //obviously there
    are also cases where //to fix the stack_a everything else if (dim == 3)
    is rejected

    {
```

```

while (dim != 3 || !(stack->a[0] < stack->a[1] && stack->a[1] < stack->a[2])) {

    if (dim == 3 && stack->a[0] > stack->a[1] && stack->a[2])
        ft_sa(stack);
    else if (dim == 3 && !(stack->a[2] > stack->a[0]
        && stack->a[2] > stack->a[1])) dim =
        when_push(stack, dim, 1); else if (stack->a[0]
    > stack->a[1])
        ft_sa(stack);
    else if (dim++)
        ft_pa(stack);
    }
}
}
}

```

first simple check that orders the stack_a with three numbers (fortified by last_a, be careful not to confuse it with dim), second check the same but with two.

third check checking the numbers in stack_a with size 3, which continues with a loop until the first three numbers are sorted,

first if

```

if (dim == 3 && stack->a[0] > stack->a[1] && stack->a[2])
    ft_sa(stack);

```

Is the first number bigger than the second? swapped (and is the third number actually there?)

else if

```

else if (dim == 3 && !(stack->a[2] > stack->a[0]
    && stack->a[2] > stack->a[1]))

```

the dimension is three but the numbers are not ordered? pusho in b.

third if similar to the first but be careful (we look at the size here not because it is not said how big the stack is but be careful also because the while also works as long as dim is different from 3)

```

else if (stack->a[0] > stack->a[1])
    ft_sa(stack);

```

Last else if manages the size of the stack by sending the values back to it happens last because it means the stack is pseudo-ordered.

VOILÀ IT'S OVER TO HELL I'M PLANNING A MUSIC THAT HAS PUT ME UP WITH MY BACK TO THE WALL AND MADE ME DEPRESS

It's not over yet I have to manage the leaks:

Fixed some general leaks missing case with a topic
everything done ready to push