CSCI 3301

Ivan Zelenkov

Abdullah Yasin Nur

May 1, 2022

# Programming Assignment Report

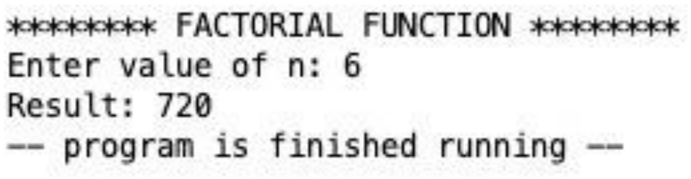
## Program for factorial of a number (*Facto.s*)

Factorial of a non-negative integer is the multiplication of all integers smaller than or equal to n. It can be calculated using the following recursive formula.

$$n! = n * (n - 1)!$$

$$n! = 1 \; if \; n = 0 \; or \; n = 1$$

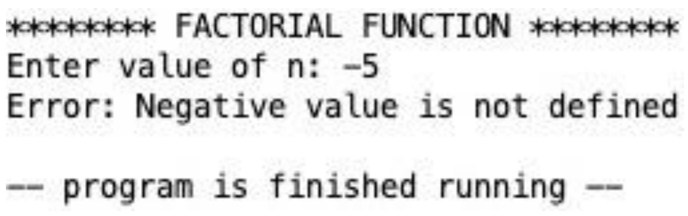
The program can calculate the factorial of numbers from 0 to 12 inclusive. For example, factorial of 6 is $6 * 5 * 4 * 3 * 2 * 1$ which is $720$. You can verify this by compiling and running my program and comparing the output with the following data.



```
******** FACTORIAL FUNCTION ********
Enter value of n: 6
Result: 720
-- program is finished running --
```

In addition, the code must be able to handle edge cases. An edge case is an issue that occurs at an extreme (maximum or minimum) operating parameter. These are the cases that are handled:

1) **Negative factorial is not defined. Display an error message.**



```
******** FACTORIAL FUNCTION ********
Enter value of n: -5
Error: Negative value is not defined

-- program is finished running --
```

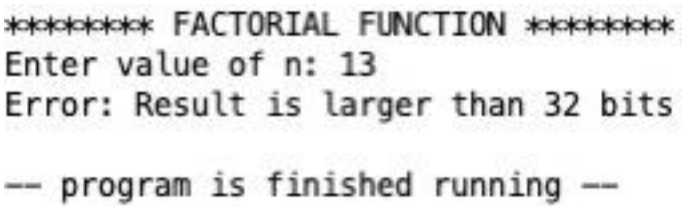
You can see lines 47 and 48 which contain the following code:

$\boldsymbol{slti\ \$t0, \$a0, 0.}$ $\boldsymbol{\#test\ for\ n < 0}$

$\boldsymbol{beq\ \$t0, 1, NegativeNumberException}$

$slti$ (set on less than immediate) will set 1 to a register $\$t0$ if value in register $\$a0$ is less than 0. If we consider the case of user input $-5$, then the register $\$t0$ is going to have value 1. Therefore, the next instruction $beq$ (branch if equal) will be true because $1 = 1$ and will branch on $NegativeNumberException$ where it will output an error message *"Error: Negative value is not defined"*.

2) **After 12 factorial, the output is larger than 32 bits. Therefore, display boundary error message.**



```
******** FACTORIAL FUNCTION ********
Enter value of n: 13
Error: Result is larger than 32 bits

-- program is finished running --
```

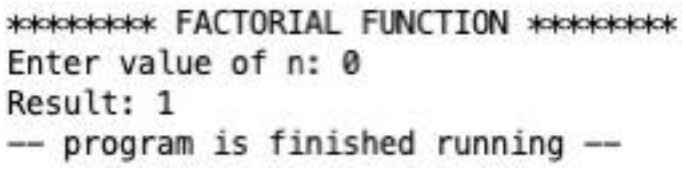
You can see lines 49 and 50 which contain the following code:

$\boldsymbol{addi\ \$t1, \$t1, 12}$ $\boldsymbol{\#\ \$t1\ =\ \$t1\ +\ 12}$

$\boldsymbol{bgt\ \$a0, \$t1, BoundaryException}$ $\boldsymbol{\#\ test\ for\ n\ >\ 12}$

First, there was created a register that contains the value 12 Then, register $\$a0$ which contains a value from the input will be compared with $\$t1$. $bgt$ (branch if greater than) is going to check if $a0 value is greater than $t1 value. If we consider the case of user input 13, then the register $\$a0$ is going to have this value and it is greater than bound 12. Therefore, a branch will be taken and the error message *"Error: Result is larger than 32 bits"* will be shown.

**3) 0! = 1**

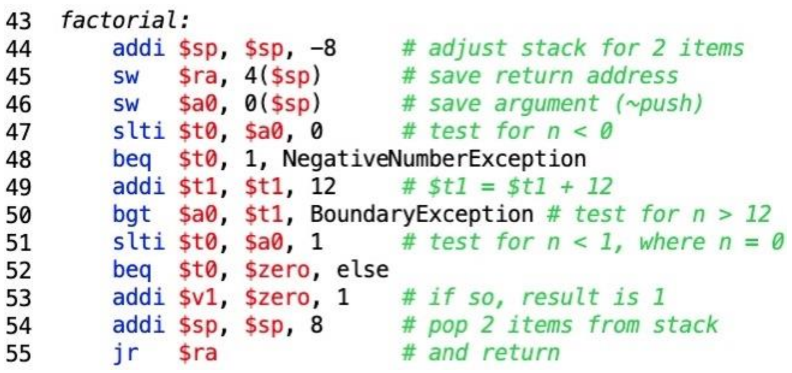


```
******** FACTORIAL FUNCTION ********
Enter value of n: 0
Result: 1
-- program is finished running --
```

You can see line 53 which contains the following code:

$\boldsymbol{addi\ \$v1, \$zero, 1 \quad \#\ if\ so, result\ is\ 1}$

Register $\$v1$ will contain value 1 and be returned to the user as output $Result$: 1 if the register $\$t0$ equals 1 at line 52.



```
43  factorial:
44      addi $sp, $sp, -8     # adjust stack for 2 items
45      sw   $ra, 4($sp)      # save return address
46      sw   $a0, 0($sp)      # save argument (~push)
47      slti $t0, $a0, 0      # test for n < 0
48      beq  $t0, 1, NegativeNumberException
49      addi $t1, $t1, 12     # $t1 = $t1 + 12
50      bgt  $a0, $t1, BoundaryException # test for n > 12
51      slti $t0, $a0, 1      # test for n < 1, where n = 0
52      beq  $t0, $zero, else
53      addi $v1, $zero, 1    # if so, result is 1
54      addi $sp, $sp, 8      # pop 2 items from stack
55      jr   $ra              # and return
```

The code above shows that there are many checks of value in register $\$a0$ done. From lines 47 to 48 was checked if the input value is negative, from lines 49 to 50 if the input value is greater than 12, and you can see that from line 51 to 52 value is also checked if it is less than 1. If we consider the case of user input 0, then all the checks above the line 51 will be false, therefore the range of values that are left to check for lines 51 to 52 is from 0 to 12, and we can be sure that the register $\$t0$ at line 51 has a value of 0, of course, if the input value was 0. Otherwise, the register $\$a0$ has a value from 1 to 12, which will branch to else block of code.

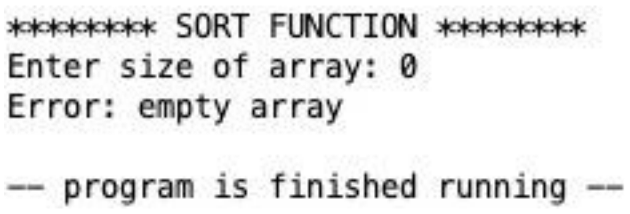
**Conclusion:** computing large factorials is a more interesting problem. There are practical issues that arise on the border between ordinary and bignum arithmetic calculations. This program works efficiently and covers all important edge cases. The improvement of this program is that instead of throwing an error, it immediately checks the input value from the user. For example, if the user input value of 50, then the program would calculate until 12 and then threw an error. This process is useless. Instead, it immediately validates the entered value and responds very quickly to the user without a long wait. Therefore, the responsiveness of this program is very high. A detailed explanation of each step is included in Facto.s and helps to understand the code.

## Program for sorting a set of given integers (*Sort.s*)

The program sorts a set of given integers using the bubble sort algorithm. The primary advantage of the bubble sort is that it is popular and easy to implement. Furthermore, in the bubble sort, elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

The array can store signed integers as input and print an error message if it is empty. Let's consider some types of input data for testing the program:

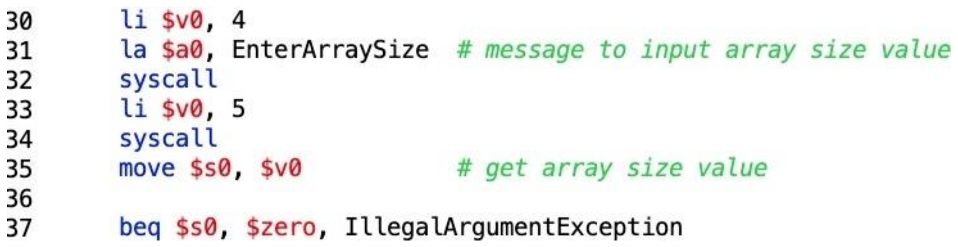
1) **Empty array is not defined. Display an error message.**



```
******** SORT FUNCTION ********
Enter size of array: 0
Error: empty array

-- program is finished running --
```

You can see lines 30-37 which contain the following code:

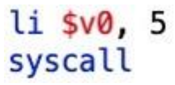


```
30      li $v0, 4
31      la $a0, EnterArraySize  # message to input array size value
32      syscall
33      li $v0, 5
34      syscall
35      move $s0, $v0           # get array size value
36
37      beq $s0, $zero, IllegalArgumentException
```

First, output message to the user to input an array size value, then get that value from input and assign to $\$s0$ register. After that is completed, check if the input value is equal to 0. If it is true, branch on $IllegalArgumentException$, output error message *"Error: empty array"* and exit.

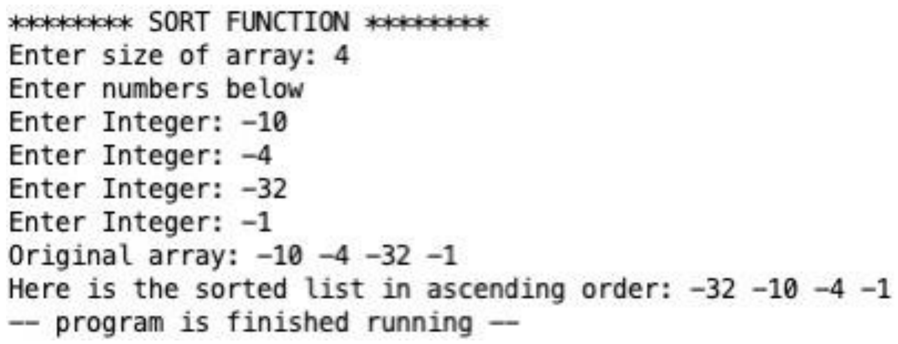
2) All negative, all positive, or both types of integers in the array.

Load immediate (***li***) operation is used in code to accept signed integers on input.

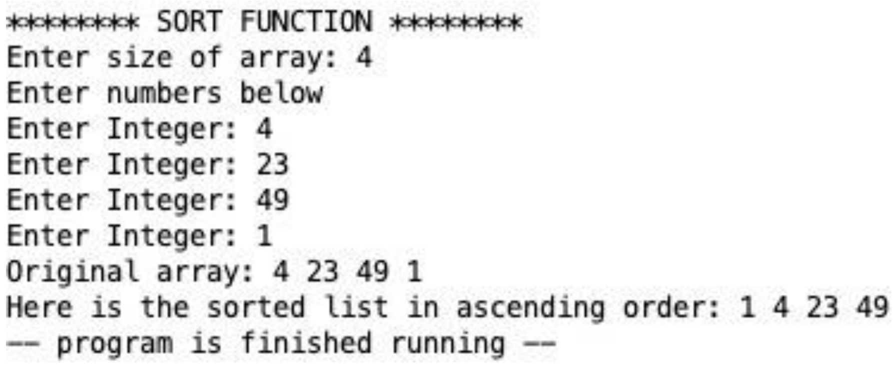


```
li $v0, 5
syscall
```

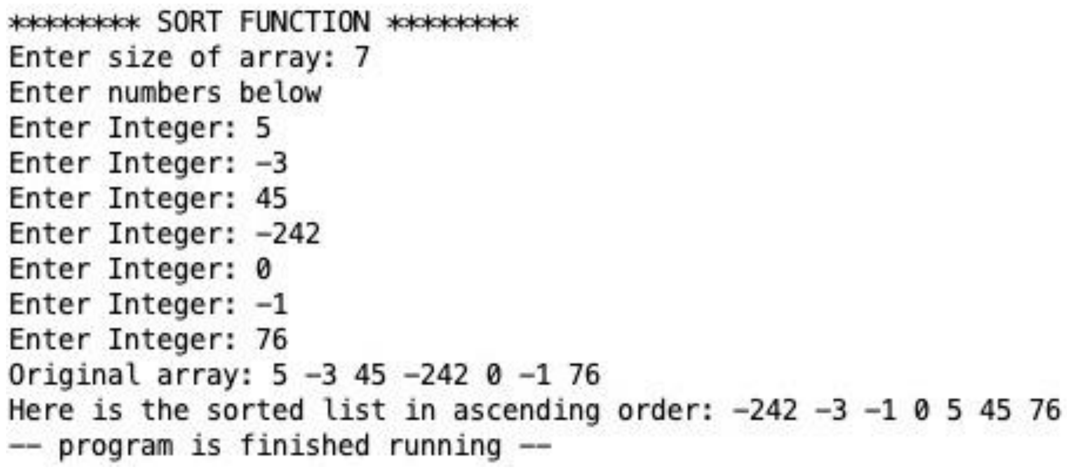
It loads a register with a value that is immediately available.



```
******** SORT FUNCTION ********
Enter size of array: 4
Enter numbers below
Enter Integer: -10
Enter Integer: -4
Enter Integer: -32
Enter Integer: -1
Original array: -10 -4 -32 -1
Here is the sorted list in ascending order: -32 -10 -4 -1
-- program is finished running --
```



```
******** SORT FUNCTION ********
Enter size of array: 4
Enter numbers below
Enter Integer: 4
Enter Integer: 23
Enter Integer: 49
Enter Integer: 1
Original array: 4 23 49 1
Here is the sorted list in ascending order: 1 4 23 49
-- program is finished running --
```



```
******** SORT FUNCTION ********
Enter size of array: 7
Enter numbers below
Enter Integer: 5
Enter Integer: -3
Enter Integer: 45
Enter Integer: -242
Enter Integer: 0
Enter Integer: -1
Enter Integer: 76
Original array: 5 -3 45 -242 0 -1 76
Here is the sorted list in ascending order: -242 -3 -1 0 5 45 76
-- program is finished running --
```

**Conclusion:** program handles all edge cases and works as expected. The major disadvantage is the amount of time it takes to sort. The average time increases almost exponentially as the number of elements increases.

Efficiency could be improved by implementing Quick Sort. The quick sort is regarded as the best sorting algorithm. This is because of its significant advantage in terms of efficiency because it can deal well with a huge list of items. It sorts in place, therefore no additional storage is required as well. This could be a good new problem to solve in the future that will improve skills in MIPS programming.

A detailed explanation of each bubble sort step is included in Sort.s to help understand the code.