Problem definition:

In all this problem statements, I provided *.data* section and .*text section*.
.data section contains day 1 to 5 sales with the data type of .WORD. This is the format:

```
.section .data

day_1: .word 11, 2, 3, 4, 5, 7, 8, 0        // Array for day_1 as unsigned integers
day_2: .word 2, 13, 4, 0                    // Array for day_2 as unsigned integers
day_3: .word 14, 5, 6, 2, 0                 // Array for day_3 as unsigned integers
day_4: .word 3, 5, 1, 12, 5, 8, 0           // Array for day_4 as unsigned integers
day_5: .word 4, 24, 6, 7, 1, 0              // Array for day_5 as unsigned integers
```

At the end of each array, there is 0 (Null terminator) indicating end of array.
Each problem statement has its own task, that's why I initialized variable in the data section as the following:

- total_sales: .word 0
- max_sale: .word 0

These will be template for all tasks.

Problem Statement 1 (Total sales for the whole 5 days):

In this problem, I approached the solution of this task with a specific way.
R0 register will be considered pointer in the whole program, and it will always point to sales. Then, result of this statement (R1 register) is load in.

In first iteration, R0 will be set to day_1, then branch to **sum_sales** with link.
In branch, R2 to R4 registers including LR (Link register) is saved in stack to avoid overriding necessary information. This allows to use registers efficiently without run out of them.

R2 register is assigned to 0. This register value will be our result for the day_1 array. After this, process moves to **_sum**. In this branch, it loads value from R0 memory address and store it in R3 register. That register now stores single stock value of day_1 array. Also, when this process done, R0 memory address value will be shift by 4 bytes (Post Indexing)

Next, CMP operation is applied, checking if the array value is reached to 0. It does, then it means, it reached end of the array. This will lead to branch equal operation being true and be forwarded to _**stop** branch, otherwise continue.

R3 value will be added to R2 (result of the statement in the branch) and loop back to _**sum.**

This process will continue until reaching 0. When reached _**stop** branch, it will load R1 (global total sales) pointer value to R4 register. The running sum (R2 register) is added to R4, then store that value in R1 register memory value.

Then, R2-R4 including Link register popped out of stack and branch back to where we left of in **global _start branch.**

This process will continue for each 5 days. Each day, pointer to first element of day_n ($n \in [1,5]$) will be load to R0, then branch to **sum_sales** and result of each branch call, will be added to R1 memory address value.

At the end of searching through all days, R1 memory address value is load to register R2, and the result is R2.

Then, the program is terminated with **software interrupt 0**.

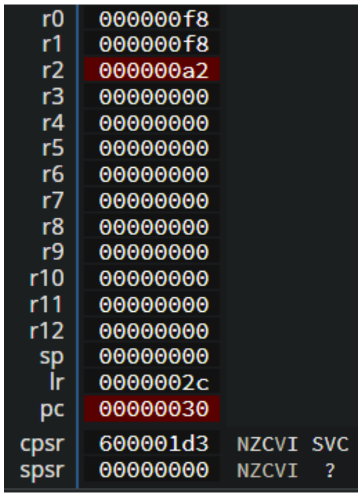
Illustration:

Input is given template. Expected Output must be **162.**

User output:

R2 register value in hexadecimal **000000a2.**

When converted into decimal, then result is 162.



```
  r0  000000f8
  r1  000000f8
  r2  000000a2
  r3  00000000
  r4  00000000
  r5  00000000
  r6  00000000
  r7  00000000
  r8  00000000
  r9  00000000
 r10  00000000
 r11  00000000
 r12  00000000
  sp  00000000
  lr  0000002c
  pc  00000030
cpsr  600001d3  NZCVI SVC
spsr  00000000  NZCVI  ?
```

Problem Statement 2 (Average sales per day):

In this problem, I approached the solution of this task with a specific way.
R6 and R7 register points to **average_val.** This is for the purpose of adding new elements to the memory address starting from R6 and incremented #4 each time, that's why the initial starting position should be stored previously to go over the elements.

R0 register will be considered pointer in the whole program, and it will always point to sales. Then, result of this statement (R1 register) is load in.

In first iteration, R0 will be set to day_1, then branch to **calc_avg** with link.
In branch, R1 to R4 registers including LR (Link register) is saved in stack to avoid overriding necessary information. This allows to use registers efficiently without run out of them.

Values are assigned to registers R1 (indicating running sum of nominator), R2 (running counter, also denominator), R4 (value of unsigned division result).

My approach is for calculating average is that, while in loop, calculate the running sum of elements in array including the count of them until null terminator meets. Then divide running sum over counter to get average.

Next, go to loop. In this branch, it loads value from R0 memory address and store it in R3 register. That register now stores single stock value of day_1 array. Also, when this process done, R0 memory address value will be shift by 4 bytes (Post Indexing)

Next, CMP operation is applied, checking if the array value is reached to 0. It does, then it means, it reached end of the array. This will lead to branch equal operation being true and be forwarded to **_divide** branch, otherwise continue.

R3 value will be added to R1 (result of the statement in the branch), 1 is added to R2 indicating that it went through element and loop back to **_loop.**

This process will continue until reaching 0. When reached **_divide** branch, it will go under simulation of division. Since, **UDIV** is not supported in ARM, I simulated in the following way:

- R1 / R2 =q (quotient)

  Then, R1 / R2 = R1 – R2 – R2 ….. until R1 < R2

  This means, q is the value which indicated how much R2 is in R1.

To solve this, first CMP operation is applied. If the carry flag or zero flag set, it can go to branch **division_rule** indicating R1 is still greater than R2. In division rule branch, R4 (The result of division) is added by 1 and R1 register is subtracted with R2. Then branch back to **divide.**

This will go until $R1 < R2$

After the condition met, the R4 register value is stored at R6 memory address, then post indexing is applied.

Then, R1-R4 including Link register popped out of stack and branch back to where we left of in **global _start branch.**

This process will continue for each 5 days. Each day, pointer to first element of day_n ($n \in [1, 5]$) will be load to R0, then branch to **calc_avg** and result of each branch call, will be stored at the memory address R6 including offset value of #4.

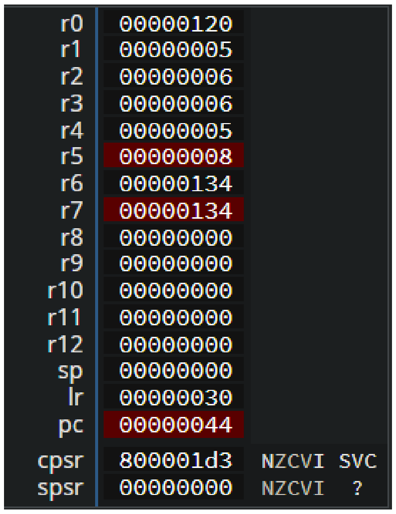
At the end of searching through all days, R7 register is used to load average values per day. First day value is loaded at R1, with post indexing. The results are labeled from R1 – R5.

Then, the program is terminated with **software interrupt 0**.

Illustration:

Input is given template. Expected Output must be **in the following:**
Day1: 5
Day2: 6
Day3: 6
Day4: 5
Day5: 8

User output:
R1 register value in hexadecimal **00000005.**
R2 register value in hexadecimal **00000008.**
R3 register value in hexadecimal **00000008.**
R4 register value in hexadecimal **00000005.**
R5 register value in hexadecimal **00000008.**



```
  r0  00000120
  r1  00000005
  r2  00000006
  r3  00000006
  r4  00000005
  r5  00000008
  r6  00000134
  r7  00000134
  r8  00000000
  r9  00000000
 r10  00000000
 r11  00000000
 r12  00000000
  sp  00000000
  lr  00000030
  pc  00000044
cpsr  800001d3  NZCVI SVC
spsr  00000000  NZCVI  ?
```

When converted to decimal, it is clear that results are the same.

Problem Statement 3 (The maximum amount sold in the whole 5 days):

In this problem, I approached the solution of this task with a specific way.
R1 register points to **max_sale.** This is for the purpose changing value of the memory address. Technically global maximum.

R0 register will be considered pointer in the whole program, and it will always point to sales. Then, result of this statement (R1 register) is load in.

In first iteration, R0 will be set to day_1, then branch to **calc_max_sale** with link.
In branch, R2 to R4 registers including LR (Link register) is saved in stack to avoid overriding necessary information. This allows to use registers efficiently without run out of them.

R2 register loads from R1 memory address value (Global Maximum load). After this, process moves to **_max**. In this branch, it loads value from R0 memory address and store it in R3 register. That register now stores single stock value of day_1 array. Also, when this process done, R0 memory address value will be shift by 4 bytes (Post Indexing)

Next, CMP operation is applied, checking if the array value is reached to 0. It does, then it means, it reached end of the array. This will lead to branch equal operation being true and be forwarded to **_stop** branch, otherwise continue.

In next step, CMP operation is applied R3, R2. This is for the purpose of checking which value is maximum. It is either current value or global maximum. If selected element, then branch to **_substitute**, otherwise loop **_max.**

In **_substitute,** R3 value is moved to R2 then branch to **_max** is applied.

When reached to **_stop** branch, the result of R2 register value will be stored in R1 memory address.

Then, R2-R4 including Link register popped out of stack and branch back to where we left of in **global _start branch.**

This process will continue for each 5 days. Each day, pointer to first element of day_n ($n \in [1,5]$) will be load to R0, then branch to **calc_max_sale** and result of each branch call, will be added to R1 memory address value.

At the end of searching through all days, R1 memory address value is load to register R2, and the result is R2.

Then, the program is terminated with **software interrupt 0**.

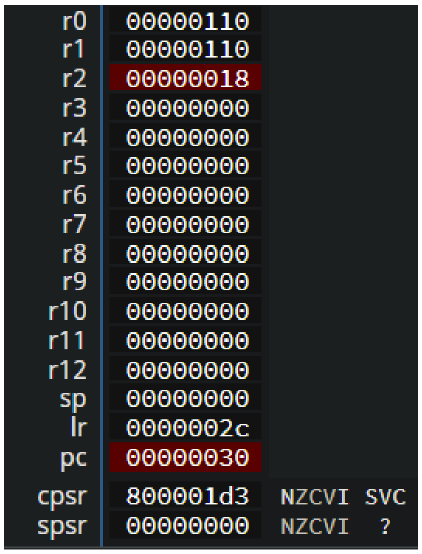
Illustration:

Input is given template. Expected Output must be **24.**

User output:

R2 register value in hexadecimal **00000018.**

When converted into decimal, then result is 24.



```
  r0  00000110
  r1  00000110
  r2  00000018
  r3  00000000
  r4  00000000
  r5  00000000
  r6  00000000
  r7  00000000
  r8  00000000
  r9  00000000
 r10  00000000
 r11  00000000
 r12  00000000
  sp  00000000
  lr  0000002c
  pc  00000030
cpsr  800001d3  NZCVI SVC
spsr  00000000  NZCVI  ?
```

Problem Statement 4 (The minimum amount sold in the whole 5 days):

In this problem, I approached the solution of this task with a specific way.
R1 register points to **min_sale.** This is for the purpose changing value of the memory address. Technically global maximum. In this case, min_sale is maximum value that can be assigned in ARM.

R0 register will be considered pointer in the whole program, and it will always point to sales. Then, result of this statement (R1 register) is load in.

In first iteration, R0 will be set to day_1, then branch to **calc_min_sale** with link.
In branch, R2 to R4 registers including LR (Link register) is saved in stack to avoid overriding necessary information. This allows to use registers efficiently without run out of them.

R2 register loads from R1 memory address value (Global Minimum load). After this, process moves to **_min**. In this branch, it loads value from R0 memory address and store it in R3 register. That register now stores single stock value of day_1 array. Also, when this process done, R0 memory address value will be shift by 4 bytes (Post Indexing)

Next, CMP operation is applied, checking if the array value is reached to 0. It does, then it means, it reached end of the array. This will lead to branch equal operation being true and be forwarded to _**stop** branch, otherwise continue.

In next step, CMP operation is applied R3, R2. This is for the purpose of checking which value is minimum. It is either current value or global minimum. If selected element, then branch to _**substitute**, otherwise loop _**min.**

In _**substitute,** R3 value is moved to R2 then branch to _**min** is applied.

When reached to _**stop** branch, the result of R2 register value will be stored in R1 memory address.

Then, R2-R4 including Link register popped out of stack and branch back to where we left of in **global _start branch.**

This process will continue for each 5 days. Each day, pointer to first element of day_n ($n \in [1, 5]$) will be load to R0, then branch to **calc_min_sale** and result of each branch call, will be added to R1 memory address value.

At the end of searching through all days, R1 memory address value is load to register R2, and the result is R2.

Then, the program is terminated with **software interrupt 0**.

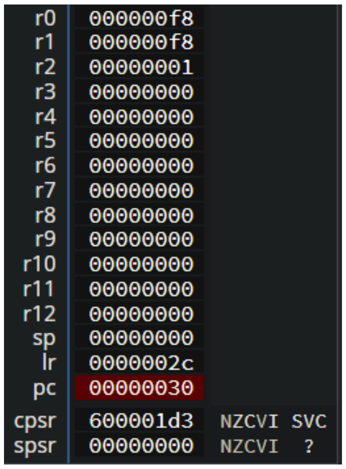
Illustration:

Input is given template. Expected Output must be **1.**

User output:

R2 register value in hexadecimal **00000001.**

When converted into decimal, then result is 1.



```
  r0  000000f8
  r1  000000f8
  r2  00000001
  r3  00000000
  r4  00000000
  r5  00000000
  r6  00000000
  r7  00000000
  r8  00000000
  r9  00000000
 r10  00000000
 r11  00000000
 r12  00000000
  sp  00000000
  lr  0000002c
  pc  00000030
cpsr  600001d3  NZCVI SVC
spsr  00000000  NZCVI  ?
```