Base RISC-V ciphering: a demonstration

Francesco Cannarozzo, francesco.cannarozzo1@stud.unifi.it, 726945

September 10, 2020

1 Assignment

As an useful assignment for the understanding of RISC-V assembly code, students in the class were given the task of writing a simple **ciphering program**, capable of encoding a string using a sequence of simple algorithms, the sequence itself being represented through another string, and then decoding it.
It is asked that the program be capable of working with **sequences of four algorithms**, processing strings **at most 100 characters long**. The encoding used for such characters is the base ASCII encoding.
Emphasis is placed on **writing modular code**: a main method will call the algorithms, forwarding the plaintext string, then retrieve the encoded string and submit it for decoding to the decoding algorithms. Therefore, they needs to be implemented in **separate procedures**.
To run our program we used **Morten Borup Petersen's RIPES**[1], in the version currently present on the Arch User Repositories.

1.1 The algorithms

Here we describe the specifications given on assignment on the four base algorithms required for a single-person assignment.

1.1.1 Caesar's cipher - Algorithm A

The simplest of them all. The algorithm receives as **inputs** the **string** to be encoded, or indeed decoded, and a **key**, in the form of a **single character** (byte). The value of the byte is then **summed** to every character in the string, thereby **encoding it**, returning the encoded string: when decoding, we merely subtract instead of summing. As an additional caveat, **we will only work on alphabetical characters**, conserving case: this means that **we will ignore numbers and symbols, and map lowercase letters strictly to lowercase letters**. The same goes for uppercase.

[1] https://github.com/mortbopet/Ripes

1.1.2 Block cipher - Algorithm B

A slightly more complex algorithm, similar in principle to Caesar's: instead of a single character, this one receives in **input a whole ciphering string**. Then, the first character of the plain-text string will be encoded with the first character of the cipher string, the second with the second, and so on. **Since the cipher string will normally be shorter than the plain-text one, we need to loop over the cipher**, so that, if it's length is e, the nth character of the plain-text string will be encoded with the character in position n mod e. For this algorithm we don't have any further restrains, except that, of course, we should ignore special characters. Just like for the previous one, decoding merely changes a sum into a subtraction.

1.1.3 Position encoding - Algorithm C

This algorithm, despite seeming almost trivial to the human reader, proved to be the most challenging to implement, and also the only one where **encoding and decoding differ significantly**. Here, **a string is transformed into a list of the occurrences of its characters**: e.g. "Hello World" becomes "H-1 e-2 l-3-4-10 o-5-8 -6 W-7 r-9 d-11". We immediately notice how this algorithm **isn't very effective on its own**: without a secret, encoding and decoding is trivial. However, when used in sequence, **increasing the length of the string is desirable**, as we shall see later. The fact that this algorithm **doesn't work in place**, however, will make implementation much more complex than the others.

1.1.4 Dictionary cipher - Algorithm D

in here, we work not with a key, but with **a rule that's hard-coded into the algorithm**: letters will be mapped to the inverted alphabet letter of the opposite case, e.g. "a" into "Z", or "B" into "y", whereas numbers will be mapped to the ASCII code of "9", minus the number. Symbols are left untouched.

2 Implementation

2.1 Further concerns

We decided to lightly expand on some of the requirements given in the assignment: as we will explain when describing the main method, we are able to apply indefinitely long sequences of encoding-decoding algorithms. As far as the character limit for the plain-text string is concerned, the only limit given by the programmer lies in this case within the position encoding/decoding module: by writing code that, from the point of view of correctness, is irrespective of the character's count number of digits, the only limit is memory, as we'll see in the implementation. Furthermore, **the code only calls instructions contained in the RISC-V base RV64I instruction set**.

2.2 global program design

We proceeded designing the main procedure. It shall **first parse the plain-text string**, writing a **newline character** at its end, for clearer output, which is maintained in encoding. Then, the **input string is parsed and appropriate algorithms are called**. After printing the encoded string to screen, the **input string is parsed again in reverse order**, calling the appropriate decoding procedures, then **printing the decoded string** after such string has been traversed.
In all of our functions, characters from the plain-text string are recovered from data memory and placed into a register one-by-one: the only one to make use of the stack will be position encoding, while position decoding will use a queue we allocate in the data field. In all cases, reading a zero byte will have the program return to caller.

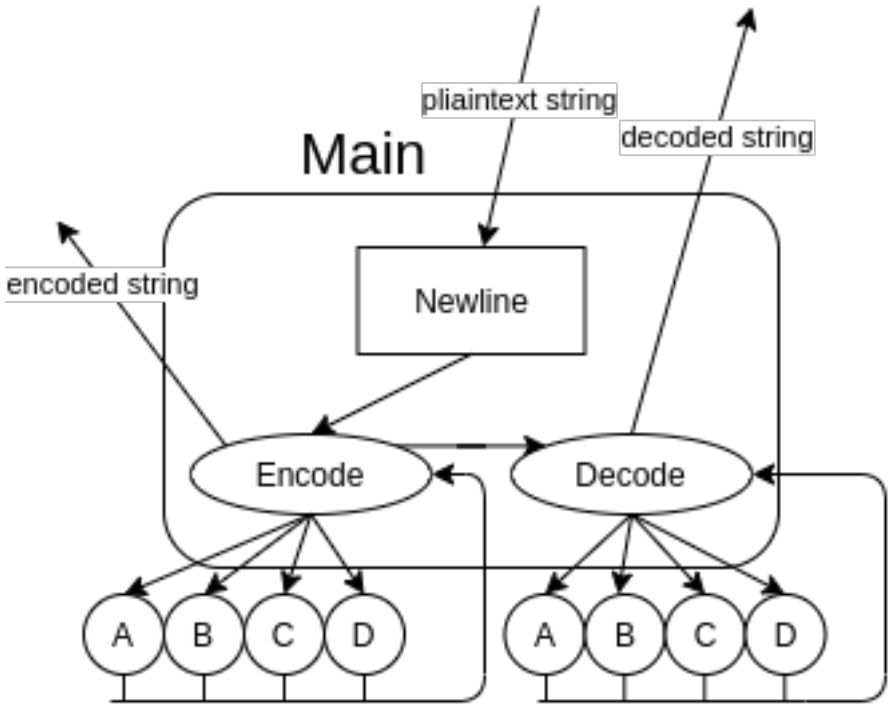


Figure 1: flowchart representing the program as a whole

2.3 in-place algorithms

as anticipated in the previous section, the position encoding and decoding algorithms are the only ones drastically different from the other six, as all the others follow a simple structure:

- (1) - check to see if a character has to be encoded. If not, check next character; when we reach end of string, return

- (2) - apply encoding/decoding function to character, then (1).

We'll now proceed to describe how algorithms A,B and D differently implement the basic algorithm.

2.3.1 Caesar's cipher - Algorithm A

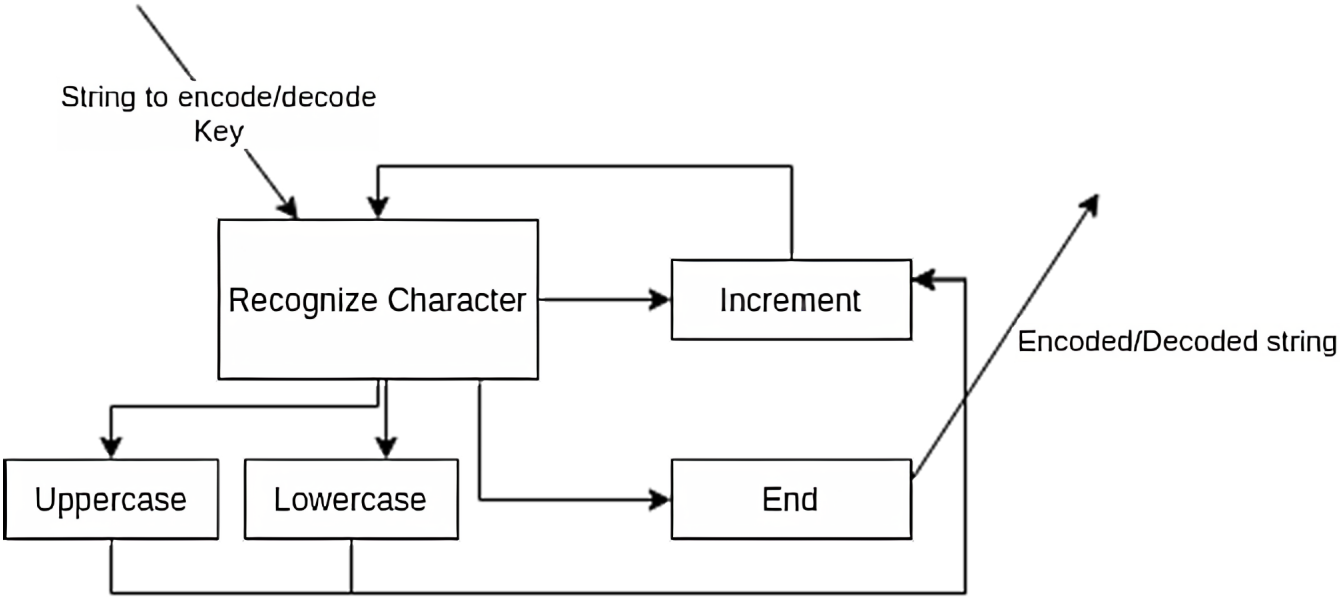
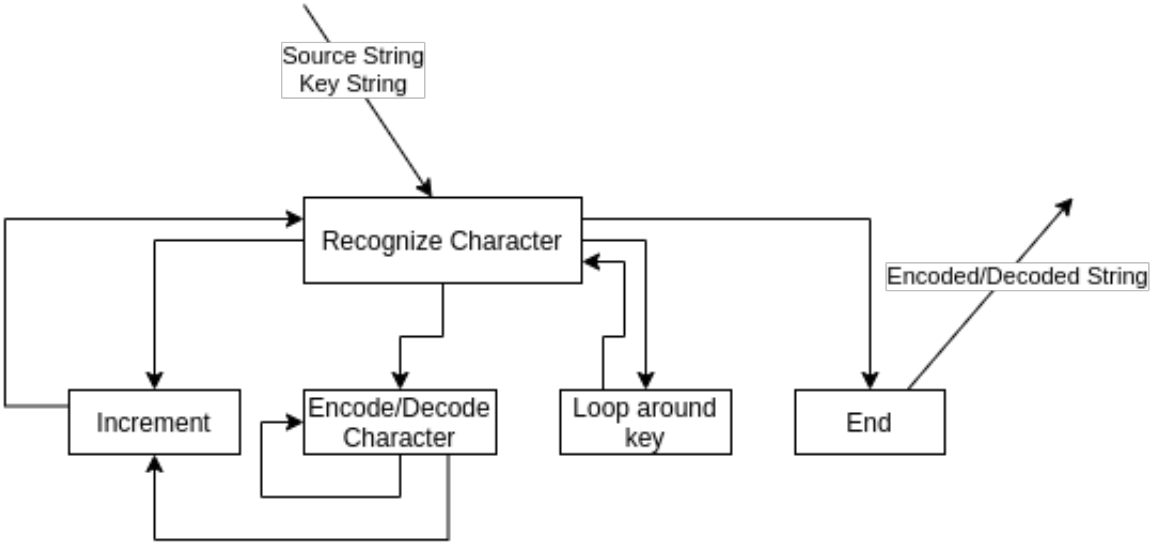


Figure 2: flowchart of our Caesar's cipher program

Starting from the very first character at the beginning of the string, the function **checks whether it is a lowercase alphabetical character, an uppercase one, or none of the two**: in this last case, it moves on to the next character, otherwise **branching to the appropriate function**, which adds (or subtracts, in case of decoding) the key byte, calculates modulo 26, then adds the lower bound of the desired character ASCII encoding range, or, in case of decoding , the upper bound. it then moves on to the next byte: when end of string is reached, it returns to caller. Below, code for both encoding and decoding: they are almost identical.

2.3.2 Block cipher - Algorithm B



Block cipher encoding follows a principle very similar to Caesar's: the function sums a plain-text byte to a key byte, obtain the encoded byte, and repeat until end of string. **This time, however, the key byte isn't a single one, but one of the characters in a string**: furthermore, bytes in the encoding string should be used one after another, so that **the nth plain-text character is encoded with the nth key character, modulo the encoding string length**. To do so, it parses both strings: whenever the key string has been completely traversed, it just gets back at the starting address, and continues encoding. Decoding is perfectly symmetrical. Furthermore, unlike the previous function, **there is little character categorization**: we expect to input a non-special ASCII character, and see any **different** non-special ASCII character to output.

S	a	t	o	r		a	r	e	p	o		t	e	n	e	t		o	p	e	r	a		r	o	t	a	S
S	g	r	u	n	t	!	S	g	r	u	n	t	!	S	g	r	u	n	t	!	S	g	r	u	n	t	!	S
G	i	(	&	"	5	#	f	m	$	&	/	*	'	b	m	(	6	~	&	'	f	i	3	)	~	*	#	G

Figure 3: Example of block ciphering

2.3.3 Dictionary cipher - Algorithm D

In dictionary cipher encoding, we don't encode with a key: instead, the program's character recognition function can discern several subsets of characters, passing every

byte to the correct encoding function, which follows a fixed, hard-coded rule: characters corresponding to letters are encoded to the opposite alphabet letter of the opposite case, while numbers have their pure value subtracted from the ASCII value of character "9". Characters different from those will have the program load the next, while a zero byte will return to caller. It's easy to notice how the rule specified in the requirements is an **involution**: that is, if c is a character and f is the function, $f(f(c)) = f^{-1}(f(c)) = c$. What this means, is that we only need one procedure to perform both encoding and decoding.

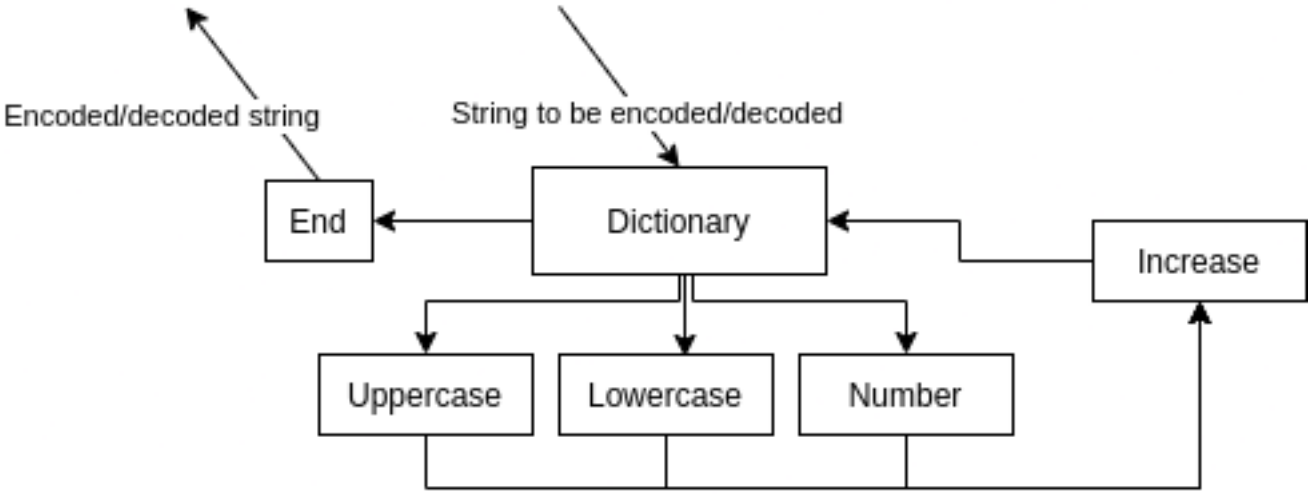


Figure 4: Chart of our Dictionary encoding/decoding

2.4 Position encoding and decoding

These two algorithms have been the **most challenging to implement**. As we noted in the preamble, these **cannot be done in** place, not only because **string length isn't conserved** after encoding or decoding, but also because a character that's further in the String to be encoded might end up before a character that appears earlier. In other words, while encoding, the placement of every character sub-string **requires full knowledge of the string**. Working out of place, both in encoding end decoding, we will need to write the encoded (or decoded) string in a support memory location, and then move the original string's starting pointer point to it.

2.4.1 Encoding

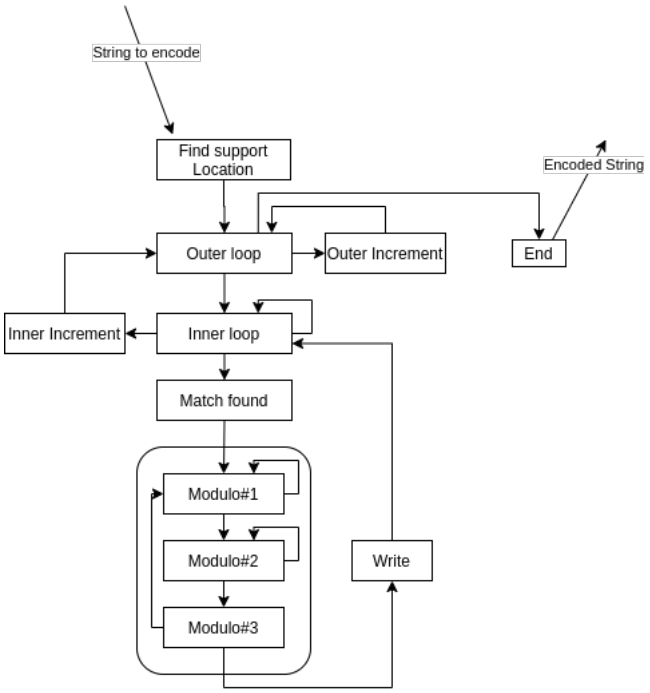


Figure 5: flowchart of our position encoding function

Encoding will have the program **examining a single character**, writing it, then **parsing the string for all occurrences of the same characters**, keeping track in a register of the **byte offset** from the beginning of the string: **this offset will then be converted to a series of ASCII characters**. To do so, the program **will calculate modulo 10, add 48, push the result onto the stack, then subtract the modulo 10 result to the original number, and repeat**. A succession of pops from the stack will write the number in ASCII, from the most significant value to the least. Notice how every time a character is mapped, the program has to parse all the characters to the right of the one examined.

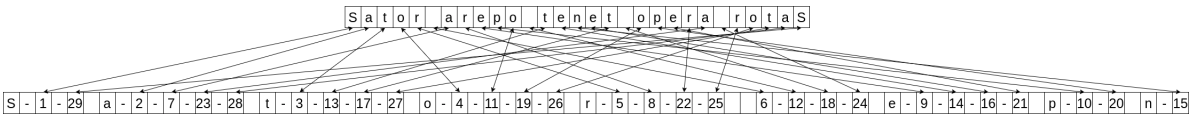


Figure 6: Example of position encoding

2.4.2 Decoding

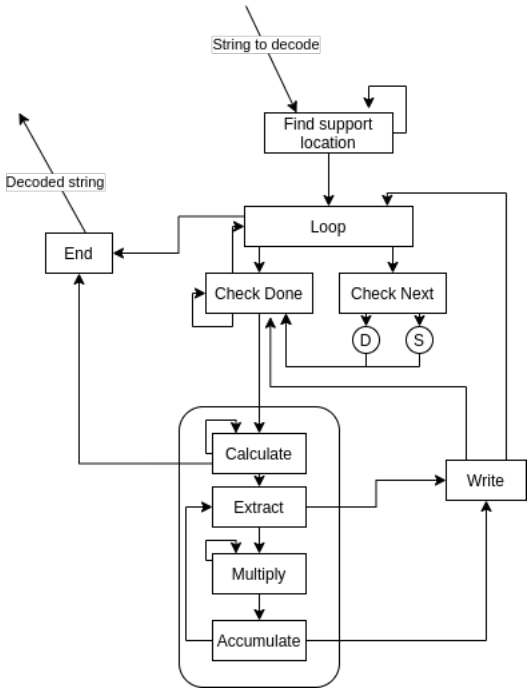


Figure 7: logical structure of our position decoding function

Whereas in encoding we used the stack as a support structure for turning pure numbers into ASCII, **the decoding program will make use of a queue**: every **sub-string describing character occurrences** is parsed, while ASCII-encoded **digits are inserted in such queue** occurrence by occurrence.
Thus, by keeping appropriate counters, it **processes ASCII-encoded digits from most to least significant**, accumulating the result and adding it (minus one) to the starting address of the decoded string its building; it then stores the character from the beginning of the sub-string, repeating this for every occurrence in the sub-string, then repeating for the next sub-string, and so on.
to implement this, it is critical to be able to distinguish when dashes and subspace signify a character to be written, or separators dictated by the encoding: to do so, we wrote an appropriate "checkNext" function.

3 Code operation

We ran a few tests of the code, as they can be seen in the attached video. As the program can detect when the plain-text string, input string, or block encoding string are empty, and return an appropriate error message, we don't have to worry about these fields being invalid. we ran the program with the text string

”N3l m3zz0 d3l c4mm1n d1 n057r4 v174
m1 r17r0v41 p3r un4 53lv4 05cur4
, che l4d1r1774v14er4sm4rr174
ah1qu4n70ad1rqu4ler4e co54dur4
3s745elv45elv466143 4spr43 for73
ch3nelpen513rrin0v4lapaur4
Tan7e amar4chep0c0pie m0rte
m4pertr47t4rd3lb3nch1’vitr0va1
d1rò d3l’al7r3 c053ch1 vh0 scor7e
10non5ob3nrid1r c0miv’intr4i
74n7’ 3r4pi3nd150nn0 4qu3lpun70
ch3lav3r4cev144bb4nd0n41”

obtaining the following results:

for input string ”A” and caesar key 64:

```
Z3x y3ll0 p3x o4yy1z p1 z057d4 h174 y1 d17d0h41 b3d gz4 53xh4 05ogd4,
otq x4p1d1774h14qd4ey4dd174
mt1cg4z70mp1dcg4xqd4q oa54pgd43e745qxh45qxh466143 4ebd43
 rad73ot3zqxbqz513dduz0h4xmbmgd4Fmz7q mymd4otqb0o0buq
 y0dfqy4bqdfd47f4dp3xn3zot1’hufd0hm1p1dò p3x’mx7d3 o053ot1 ht0 eoad7q10zaz5an3zdup1d
  o0yuh’uzfd4u74z7’ 3d4bu3zp150zz0 4cg3xbgz70 ot3xmh3d4oqh144nn4zp0z41
```

for input string ”B” and block string ”http://www3.google.com”:

```
WH"1HbJ3H88at0s<zs_r0r:5$AofBL8/de<0}9-x_;#>
 IF1KbB8.’gN=C|˜A&ˆ9s${IA1>858%L8_zAG?A|_8u!=)#EMB‘OL8579"&<{=ˆet?{’+EG5BL}87>=Dt}
  :awGB>z"(nd5%/Lid9DC(Ay?vDA){%$qb3!K’9;xu˜=>9Avy
  |9,I}<@1.+L(0vGu(ns0vDqqz&A>_@"}8Aˆz%uuAv4v%!=
  L*EM4b%zKB2pA7˜vzA4’o:yF$òO|K%Z0tG#;-
  iˆ9CqqF5(C_O,{(FemA@v|tcsrAw(˜ukBO{H&=E/y˜
  | :8;D|@<5DMc@"K’8_=@˜v=&bu&Au&+ q_O{!K@0˜C#<pkE5DBkwI ?_>LI
```

for input string ”C”

```
N-1 3-2-6-12-50-58-129-147-154-160-163-172-227-230-251-258-263-286-312-317-330-340-344
 l-3-13-59-75-115-135-140-166-180-228-252-255-331-341
   -4-10-14-21-24-31-36-39-48-52-56-62-70-74-98-120-148-
   155-192-208-249-259-267-271-293-311-326-337
   m-5-17-18-37-91-194-209-214-296 z-7-8 0-9-26-44-63-107-177-202-204-
   210-240-261-270-279-295-322-325-336-358
    d-11-22-77-109-125-226-244-250-290-319-357
    c-15-65-71-121-161-198-203-232-260-264-273-294-338-347
     4-16-30-35-46-55-61-68-76-83-86-89-92-97-104-114-118-124-128-132-
     137-142-146-149-153-179-186-197-215-221-224-304-307-314-327-346-351-352-355-360
```

```
      1-19-23-33-38-41-47-78-80-85-95-101-110-145-171-234-243-245-266-278-291-320-350-361
    n-20-25-54-105-164-169-176-189-231-280-282-287-301-308-318-323-324-334-356-359
   5-27-57-64-123-133-138-170-262-283-321
   7-28-34-42-81-82-96-106-131-159-190-222-256-276-306-309-335
        r-29-40-43-51-67-79-88-93-94-111-117-127-152-158-173-174-185-196-
        211-218-220-225-239-246-
     257-275-288-292-303-313-345 v-32-45-60-84-136-141-178-236-241-268-298-343-349
      p-49-151-167-182-201-205-216-315-332 u-53-66-103-113-126-184-329-333 ,-69
       h-72-100-162-199-233-265-269-339 e-73-87-116-119-134-
       139-165-168-191-200-207-213-217-277-348
        s-90-130-150-272 a-99-108-181-183-188-193-195-242-254-342 q-102-112-328
        o-122-157-274-281-284 6-143-144
         f-156 i-175-206-237-289-297-300-305-316 T-187 t-212-219-223-238-302
         b-229-285-353-354 '-235-253-299-310
```

for input string "D":

```
m6O N6AA9 W6O X5NN8M W8 M942I5 E825 N8 I82I9E58 K6I FM5 46OE5 94XFI5, XSV
 O5W8I8225E85VI5HN5II825 ZS8JF5M29ZW8IJF5OVI5V XL45WFI56H254VOE54VOE533856 5HKI56
  ULI26XS6MVOKVM486IIRM9E5OZKZFI5gZM2V ZNZI5XSVK9X9KRV
   N9IGVN5KVIGI52G5IW6OY6MXS8'ERGI9EZ8W8Iò W6O'ZO2I6 X946XS8 ES9 HXLI2V89MLM4LY6MIRW8I
    X9NRE'RMGI5R25M2' 6I5KR6MW849MM9 5JF6OKFM29 XS6OZE6I5XVE855YY5MW9M58
```

Instead, with combination "CBAD" we obtained:

```
rmi8B\NjajK‘4j@4m>[4mh3ifgH‘KcjfF^4ni;:2J3=@;cmlB_
\ebf‘‘=g=:l0[3gk3gilHM‘zjdI^4lk1:0M19O3gmmB\KxjzG[0nj4>
0L8nm0mihY\‘ygjJ‘@=m=?6‘0j;<gi>BL‘11jH[0@=0n6‘4=@=mgm
HMJjdx‘M@=j:;K8i@3cj>XL\xyjK‘1=n<j6_0j;:ah>C_HjecC[:jf4?<J8mj
:mheB\MffjI‘>=l;k&<8j;:cmmW\MzjxK[0fk4?3H8m?=mheYzejzjDa1=
f4?<[1k;?gmmEI\fzz‘‘1m=:=:[3n>3hfnHNJfjeE^4mhn:1H0=n;hmkCK\
dda‘N=g9C:2_8m@3cc>D_GjfeG[:mi4?:M8ml0mheE\MfxjIL?9V4>;[:j;@
immC‘\faf‘_ng=:=0[3l@3hdnHNJcjeEN4mf<:0N<=n=c4jH‘Jjdg‘N==k>:;
L8i?3db>XJ\ydjDK4gf4i1[=h;:jf>D‘LjffD[0mk4>1I8nn;mikX\‘cejKM>=n<i
6_0l;:ca>DHJjfxE[:nj4?1_8m@=mgnA\MgzjI_<=l:k6N1i;<ei>BKNjdbG[;i@(
>6_==@<mgkHMHjcf‘M?=h@:>^8gl3ae>D_‘jffL[0kj4>=_8mn=mhjB\Ncb
jJK>=m?j6‘=n;<hj>BK_jdaKaS=m1:1L8jm3ijiH‘JcjfFH4nh>:2I==@<imlW_\
eye‘‘@h=;=2[2@i3gifHMNdjdJM4ll<:0L:=n>a4iHNIjbz‘K<=n:@6_2l;:gb>D
I_jeaJ[:gl4@1_$h;;bmkA\LejyK[@m=nl6_5i;:gi>DKGjfxL[:mm4?;K8mj?mg
nY\MgxjIN=9#4??[1@;=gmiD\JzjzC[@g=n@6H1=?:immDI\fez‘_=m=0m>[
4hn3icjH‘HbjfCK4mn0:1_<=@;jmlCK\edx‘‘<i=:m=[3hl3hbfHNGejdLN4ln;:
0MO9%3gh>AK\agjDM4nl>:2M4=?@bmlBJ\ecf‘‘>g=:i>[2kn3gfeo@\cxjKL
0=n>k6_<m;;ji>C_KjefF[;nj4@0‘$&;>gmhY\‘gdjK_;=n:l6_<k;<ha>BMM1kjF
H(Q=??6_5@;:dh>DGGjedI[:ij4?<H8lnm4O>XM\yzjK_>=n0i6_2k;:ga>DJKjf
aD[0fn4?3^8m>@mhmB\NfzjJJ?=l<j&m8f>3ignH‘KgjeE‘(X=ni6_5g;:bi>DH
MjfyD[0fl4>?L8mm;mhiA\Mce1j[0@m4>2‘8l@n4%>DNNjfbE[:hk4?>_8mi
=4d>DLMjfcHaA=n=l&18nj>mhnY\NdzjJIn=mnk6N5@;<je>B‘J1CjKI?9%4
?2‘8m?mmhlB\NdyjI^:9W4?1H8mi>mgiB\Mbc1o[:lj4?;N8mhmmgmEz
```

In each of these cases, decoding returned the initial string, as intended.

4 Appendix

4.1 time complexity

In evaluating our algorithms' time complexity proportional to string length, we still find position encoding behave differently, if only in the encoding half, as every other algorithm has time complexity in $O(n)$, its being in $O(n^2)$, if we consider the fact that the $log_{10}n$ factor we get from operating with figures can be taken, as far as the operation of our program is concerned, a small constant value.

4.2 Difficulty

Considering the case of having to guess the key, what is the number of guesses we have to take, in the worst case, before we find it? A factor is of course the number of possible values that the key can take: in our case, it's never bigger than the amount of values a byte can store, thus 256. So, if k is the number of possible values, in the case of Caesar's cipher, the worst case scenario sees k guesses.

For block encoding, the key is itself a string: if we assume it is not longer than the string to be encoded, the worst case amounts to trying all the strings long l made up of k characters, in total k^l tries, observing that the set of strings that are long exactly l contains all strings formed by repetition of a shorter string.

Position encoding, of course, doesn't have a key to find out: decoding is straightforward, and we need exactly one try.

For the dictionary cipher, we have to think differently: we are not to find a key, but a rule: that is, a function from a set of character to the same set of characters. Furthermore, as decoding has to be possible, such a function should be bijective: the number of those is exactly $k!$: since our admissible symbols are 95, $95! \approx 10^{148}$, an obviously absurd number. It is easy to notice, however, how any information on the function can drastically reduce this number: for example, knowing only that our function maps letters to letters and numbers to numbers, brings the worst case number of tries to $2 \times 26! + 10! \approx \times 10^{26}$. Furthermore, if we knew that the function is an involution without fixed points, the number of such functions in a set of $2n = k$ elements can be computed[2] as $(2n-1)!! = \frac{2n!}{2^n n!}$: in the case we also knew that letters are mapped onto letters and numbers to numbers, that would reduce the possible choices to $2 \times \frac{26!}{2^{26}13!} + \frac{10!}{2^{10}5!} \approx 10^9$, enormously less than the original figure. The 2 multiplication is here introduced by the two separate sets of uppercase and lowercase letters.

[2] Philippe Flajolet and Robert Sedgewick, Analytic Combinatorics, CUP, 2009.

4.3 Combining ciphers

As our program applies a sequence of different ciphers to a plain-text message, it is interesting to understand how much better the same ciphers used one after another can be when compared to using them individually. Of course, when we combine cipher, the combination itself is part of the secret: even knowing the single algorithms used, only being able to check if our guess is correct after after every decoding is performed would leave us with a worst case of $\left(\sum_0^k d_i\right)^s$, where we have k algorithms with worst cases $d_0 \dots d_k$, and s stages: having instead the possibility to check correctness at every step would reduce the s to a multiplicative factor.
However, were an encoding to modify a property of the string that impacts difficulty, the order of the sequence itself would become relevant: in our case, position encoding greatly increases length, roughly twofold to threefold, which alters the difficulty of block encoding in the worst case.

5 Full code

```
 .data
mycipher: .string ""
sostK: .word 64
blocKey: .string ""
plaintexterror: .string "Error: plaintext string is empty"
blockerror: .string "Error: block encoding string is empty"
inputerror: .string "Error: input string is empty"
myplaintext: .string ""
queue: .string ""
.text
#main function
la a0 myplaintext
lb a1 0(a0)
beq a1 zero plaintextempty
#addnewline - adds newline character at end of string
add a2 a0 zero
nlLoop:  #parses string to be encoded until end, then adds newline character
addi a0 a0 1
lb t0 0(a0)
bne t0 zero nlLoop
li t1 10
sb t1 0(a0)
add a0 a2 zero
li a7 4
ecall   #outputs string to be encoded
la s1 mycipher
lb t0 0(s1)
beq t0 zero inputempty
idLoopEncode:  #parses input string, then calls appropriate encoding functions
lb s4 0(s1)
```

```
li s3 65
beq s4 zero idLoopDecode  #when end of string isreached, go to decoding
bne s4 s3 notA  #checks if current input string character is A
jal caesarE
addi s1 s1 1
lb s4 0(s1)
notA:
addi s3 s3 1
bne s4 s3 notB #checks if current input string character is B
jal blockE
addi s1 s1 1
lb s4 0(s1)
notB:
addi s3 s3 1
bne s4 s3 notC #checks if current input string character is C
jal occurrencesE
addi s1 s1 1
lb s4 0(s1)
notC:
addi s3 s3 1
bne s4 s3 notD #checks if current input string character is D
jal dictionaryPE
addi s1 s1 1
lb s4 0(s1)
notD:
j idLoopEncode
idLoopDecode: #parses input string in reverse, calls appropriate decoding functions
ecall  #outputs encoded string
la s1 mycipher
scrolloop:
lb t0 0(s1)
beq t0 zero decode
addi s1 s1 1
j scrolloop
decode:
addi s1 s1 -1
lb t0 0(s1)
beq t0 zero result
li t1 65
bne t0 t1 notAA
jal  caesarD
notAA:
addi t1 t1 1
bne t0 t1 notBB
jal blockD
notBB:
addi t1 t1 1
bne t0 t1 notCC
jal occurrencesD
```

```
notCC:
addi t1 t1 1
bne t0 t1 notDD
jal dictionaryPE
notDD:
j decode
result:
ecall  #outputs decoded string
addi a7 zero 93
ecall  #terminates program
#caesar's cypher - Encode
caesarE:
lw t3 sostK
add a4 a0 zero
caesarEChara:  #takes string address and key value, modifies alphabetical characters in string
lb t0 0(a0)                          #load character
beq t0 zero endECaesar               #checks character isn't null (RETURN TO CALLER)
slti t1 t0 65                        # lower external alphabetical character boundary
not t1 t1
slti t2 t0 123                       # upper external alphabetical character boundary
and t1 t1 t2
beq t1 zero incrementECaesar         #checks if t0 encoding value is within boundary
           #(REPEAT OUTER)
slti t1 t0 91                        #lower internal alphabetical character boundary
not t1 t1
slti t2 t0 97                        #upper internal alphabetical character boundary
and t1 t1 t2
bne t1 zero incrementECaesar               #check if t0 encoding value is within internal boundary
     #(REPEAT OUTER)
#both bne fail: chara is alphabetic
slti t1 t0 91
beq t1 zero caseELowerCase  #state if character encoding corresponds to lowercase letter
add t0 t0 t3          #add key value
ucELoop: #apply modulo until conditions on output satisfied
slti t1 t0 91
bne t1 zero incrementECaesar
addi t0 t0 -91 #subtract upper uppercase character bound
addi t0 t0 65  #add lower uppercase character bound
j ucELoop
caseELowerCase:
add t0 t0 t3
lcELoop: #apply modulo until conditions on output satisfied
slti t1 t0 123
bne t1 zero incrementECaesar
addi t0 t0 -123  #subtrasct upper lowercase character bound
addi t0 t0 97         #add lower lowercase character bound
j lcELoop
incrementECaesar: #parses string
sb t0 0(a0)
```

```
addi a0 a0 1
j caesarEChara
endECaesar:
add a0 a4 zero
ret
#caesar's cypher - Decode
caesarD:
add a4 a0 zero
lw t3 sostK
caesarDChara: #takes string address and key value, modifies alphabetical characters in string
lb t0 0(a0)                                          #load character
beq t0 zero endDCaesar        #checks character isn't null (RETURN TO CALLER)
slti t1 t0 65         #lower external alphabetical character boundary
not t1 t1
slti t2 t0 123        #upper external aphabetical character boundary
and t1 t1 t2          #checks if t0 encoding value is in inside boundary
beq t1 zero incrementDCaesar  #if the charachter is outside of these bounds,
                          #goes to next character (REPEAT OUTER)
slti t1 t0 91         # lower internal alphabetical character boundary
not t1 t1             #invert bit: results 1 if character encoding is above boundary, else 0
slti t2 t0 97         #upper internal alphabetical character boundary
and t1 t1 t2          #checks if character encoding is within bounds
bne t1 zero incrementDCaesar  #if it is, goes to next character (REPEAT OUTER)
slti t1 t0 91         #upper uppercase bound
beq t1 zero caseDLowerCase  #if higher, go to lowercase (UPPERCASE/LOWERCASE)
sub t0 t0 t3        #subtract key
ucDLoop:            #uppercase modulo loop
slti t1 t0 65       #checks if t0 is below lower uppercase bound after subtraction
beq t1 zero incrementDCaesar  #if it isn't, move on to next character (REPEAT OUTER)
addi t0 t0 91                  #add upper bound
addi t0 t0 -65                #subtract lower bound: we have computed one step of modulo
j ucDLoop                     # (REPEAT INNER)
caseDLowerCase:
sub t0 t0 t3            #subtract key
lcDLoop:            #lowercase modulo loop
slti t1 t0 97         #checks if t0 is below lower lowercase bound after subtraction
beq t1 zero incrementDCaesar      #if it is,  move on to next character (REPEAT OUTER)
addi t0 t0 123            #add upper bound
addi t0 t0 -97            #subtract lower bound: we have computed one step of modulo
j lcDLoop                 # (REPEAT INNER)
incrementDCaesar: #modifies data and parses string
sb t0 0(a0)       #save the eventually modified character
addi a0 a0 1      # go to next character in string
j caesarDChara    # (REPEAT OUTER)
endDCaesar:
add a0 a4 zero
ret # (RETURN TO CALLER)
#block cypher
blockE:
```

```
la a2 blocKey
lb t0 0(a2)
beq t0 zero blockEmpty
add t2 a2 zero  #copy encoding string starting address for looping over
add a4 a0 zero
blockEChara: #Takes two strings, uses one to modify the other
lb t0 0(a0) # load byte from string to be encode
lb t1 0(a2) #load byte from encoding string
beq t0 zero endEBlock # checks if t0 is null (RETURN TO CALLER)
beq t1 zero loopEKey
slti t3 t0 32 #checks lower non-control character bound
slti t4 t0 127  #checks upper characters bound
not t3 t3       #invert byte: results 0 when below 127, 1 otherwise
and t3 t3 t4     #checks if any of the conditions is true
beq t3 zero incrementEBlock    #if we're beyond boundaries,
                  #we increment the addresses (REPEAT OUTER)
add t0 t0 t1 # add encoding string value
blockModELoop:  #Modulo loop
slti t3 t0 127  #checks if modified byte is within upper boundary
bne t3 zero incrementEBlock   #if it is, increment (REPEAT OUTER)
addi t0 t0 -127 #subtract upper bound
addi t0 t0 32     #one step of modulo
j blockModELoop   # (REPEAT INNER)
incrementEBlock:  #parses through both strings
sb t0 0(a0)       #save the eventually modified string
addi a0 a0 1      #go to next character in string to be encoded
addi a2 a2 1      #go to next character in encoding string
j blockEChara     #  (REPEAT OUTER)
loopEKey:       #allows for encoding string to loop over string to be encoded
add a2 t2 zero    #simply retrieve starting address from memory
j blockEChara     # (REPEAT OUTER)
endEBlock:
add a0 a4 zero
ret    # (RETURN TO CALLER)
blockD:
la a2 blocKey
add t2 a2 zero #Copy encoding string starting address for looping over
add a4 a0 zero
blockDChara:
lb t0 0(a0)   #load byte from string to be decoded
lb t1 0(a2)   #load byte from decoding string
beq t0 zero endDBlock       # checks if t0 is null (RETURN TO CALLER)
beq t1 zero loopDKey        # checks if t1 is null: if it is, we go back to the beginning of the
slti t3 t0 32               # checks lower non-control character bound
slti t4 t0 127              #checks upper character bound
not t3 t3                   # invert byte: 0 when below 127, 1 otherwise
and t3 t3 t4                 #checks if one of the conditions is true
beq t3 zero incrementDBlock   #if we're beyound boundaries,
                         #we increment the addresses (REPEAT OUTER)
```

```
sub t0 t0 t1      #subtract encoding string value
blockModDLoop:    #Modulo loop
slti t3 t0 32     #checks if t0 value lower than 32 after subtraction
beq t3 zero incrementDBlock   # if it isn't, increment (REPEAT OUTER)
addi t0 t0 127      #add upper bound
addi t0 t0 -32      #subtract lower bount: we have performed one step of modulo
j blockModDLoop     # (REPEAT INNER)
incrementDBlock: #parses through both strings
sb t0 0(a0)   #saves eventually modified byte
addi a0 a0 1  # go to next character in string to be encoded
addi a2 a2 1  # go to next character in encoding string
j blockDChara # (REPEAT OUTER)
endDBlock:
add a0 a4 zero
ret           # (RETURN TO CALLER)
loopDKey:   #allows for encoding string to loop over string to be encoded
add a2 t2 zero   #simply retrieve starting address from memory
j blockDChara # (REPEAT OUTER)
#occurrences
occurrencesE:
li a3 -1   #read-data placeholder
li a4 45   # "-"
li t2 0    #ext counter
add a1 a0 zero
locationFinderE:  #finds free memory location for storing encoded string,
                            #since this encoding is not in-place
lw t1 0(a1)
beq t1 zero occurrencesEOuter
addi a1 a1 4
add a2 a1 zero
j locationFinderE
occurrencesEOuter:  #parses s till end of string
lb t0 0(a0)
slti t4 t0 32   #valid character check
addi t2 t2 1 #outer loop counter
beq t0 zero endOccurrencesE
bne t4 zero occurrencesEIncreaseInv
sb t0 0(a1)  #save character in encoded string
add t1 a0 zero    #copy a0 address
addi a1 a1 1
add t3 t2 zero
occurrencesEInner:
lb t4 0(t1)   #load from a0 address copy
beq t4 t0 Ematch  #for every character, parse string looking for instances of the same char
addi t3 t3 1 #inner loop counter
beq t4 zero occurrencesEIncrease #end inner loop
addi t1 t1 1
j occurrencesEInner
Ematch: #when we have a match, write "-" and obscure charachter in the string
```

```
          # to be encoded with placeholder value -1
sb a4 0(a1)
sb a3 0(t1)
addi a1 a1 1
positionCodECalculator:
add t4 t3 zero #copy inner loop counter
modulor:  #calculates ascii charachters corresponding to placement
add a6 t4 zero
moduloop: #subtracts 10 increasing counter until value in t4<10
slti t6 t4 10
bne t6 zero modulor2
addi t4 t4 -10
addi t5 t5 1
j moduloop
modulor2:
add a5 t5 zero
li a4 0
othermod:
beq t5 zero subMod
addi a4 a4 10
addi t5 t5 -1
j othermod
subMod: #finish calculating characters by adding 48, then push them onto the stack:
         #less significant digits are pushed first
sub t4 a6 a4
slti t6 a6 10
addi t4 t4 48
addi sp sp -1
sb t4 0(sp)
bne t6 zero write
add t4 a5 zero
j modulor
write:  #writes characters to encoded string by popping them from the stack;
        #most significant digits are popped first
li a4 45
lb t4 0(sp)
beq t4 zero occurrencesEInner
addi sp sp 1
sb t4 0(a1)
addi a1 a1 1
j write
occurrencesEIncrease: #ends inner loop and prepares for next iteration of outer loop
li t6 32
sb t6 0(a1)
addi a0 a0 1
addi a1 a1 1
j occurrencesEOuter
occurrencesEIncreaseInv: #for invalid characters,
                        #we continue parsing without writing anything to destination
```

```
addi a0 a0 1
j occurrencesEOuter
endOccurrencesE: #puts encoded string starting address in a0 and returns
li t0 10
sb t0 0(a1)
add a0 a2 zero
ret
occurrencesD:
li a5 32 #load space chara
li a6 45 #load "-" chara
add a1 a0 zero
li a4 10
add t3 a1 zero
locationFinderD:  #finds appropriate memory location for decoded string,
               # as the procedure is not in-place
lw t1 0(a1)
beq t1 zero occurrencesDLoop
addi a1 a1 4
j locationFinderD
occurrencesDLoop:
add t3 a1 zero
lb t0 0(a0)
#sb zero 0(a0)
#beq t0 t5 endOccurrencesD
beq t0 zero endOccurrencesD
addi a0 a0 1
beq t0 a5 checkNext
#beq t0 a6 checkNext
occurrencesDInner:
lb t1 0(a0)
checkDone:
la a2 queue    #load queue address.
addi a0 a0 1   #increase source string pointer
beq t1 a6 occurrencesDInner  #if we get "-", continue parsing substring
beq t1 a5 occurrencesDLoop   # if we get space, start parsing next substring
li t2 -1      #start tens counter at -1
add t5 a2 zero   #copy starting queue address, t5 is back of the queue
li t4 0          #start number translation at 0
positionCodDCalculator:      #here we push ascii numbers into queue
beq t1 a5 extractorLoop    #when we get to end of number substring, start extracting from queue
beq t1 a6 extractorLoop
beq t1 zero endOccurrencesD
sb t1 0(t5)     #store current byte at back of queue
addi t2 t2 1    #increase tens counter
addi t5 t5 1    #increase queue back pointer
lb t1 0(a0)     #load next byte
addi a0 a0 1    #increase source string pointer
j positionCodDCalculator
extractorLoop:  #here we dequeue
```

```
lb t1 0(a2)   #take byte from front of queue
addi a2 a2 1  #increase queue front pointer
#beq t2 zero writeD  #when we reach end of number substring, go to write
addi t1 t1 -48     #subtract 48 to get pure number
add t5 t2 zero  #we put current tens counter in t5

multiplier:     #here we multiply by powers of ten
beq t5 zero accumulate
slli t6 t1 1    #multiply by ten
slli t1 t1 3
add t1 t6 t1
addi t5 t5 -1
j multiplier
accumulate:
add t4 t4 t1   #accumulate extracted values one by one into number translation
beq t2 zero writeD
addi t2 t2 -1
j extractorLoop
writeD:
add t4 a1 t4 #add number translation minus one to destination string starting address
addi t4 t4 -1
sb t0 0(t4)   #store byte in t0 there
lb t1 0(a0)
addi a0 a0 -1
beq t1 a5 occurrencesDLoop
j occurrencesDInner
endOccurrencesD:
lb t0 0(t3)
addi t3 t3 1
bne t0 zero endOccurrencesD
sb t5 0(t3)
add a0 a1 zero
ret
checkNext:
lb t1 0(a0)
beq t1 a6 dash
beq t1 a5 space
j occurrencesDLoop
dash:
add t0 t1 zero
j checkDone
space:
addi a0 a0 2
lb t1 0(a0)
j checkDone
#dictionary
dictionaryPE:
add a4 a0 zero
dictionaryE: #takes in a string, modifies the string according to a set of different rules
```

```
      #depending on character subsets
lb t0 0(a0) #load character
beq t0 zero dictEEnd

# we first need to define a larger acceptable character boundary:
                #it goes from 48 to 122, and has two internal boundaries
slti t1 t0 48
not t1 t1
slti t2 t0 123
and t1 t1 t2
beq t1 zero dictEIncrease
#now we need to separate different cases
slti t1 t0 58
bne t1 zero dictENum
slti t1 t0 91
bne t1 zero dictEMai
slti t1 t0 97
bne t1 zero dictEIncrease
li t1 90   #lowercase letters become uppercase opposite
addi t0 t0 -32
addi t0 t0 -65
sub t0 t1 t0
j dictEIncrease
dictENum: #we load the ascii encoding for 9,
        # obtain the ascii number in t0 by subtracting 48, then subtract
li t1 57
addi t0 t0 -48
sub t0 t1 t0
j dictEIncrease
dictEMai:    #Uppercase letters become lowercase opposite
slti t1 t0 65
bne t1 zero dictEIncrease
li t1 122
addi t0 t0 32
addi t0 t0 -97
sub t0 t1 t0
j dictEIncrease
dictEIncrease:   #go on parsing
sb t0 0(a0)
addi a0 a0 1
j dictionaryE
dictEEnd:
add a0 a4 zero
jr ra
#errors
plaintextempty:
li a7 4
la a0 plaintexterror
ecall
```

```
li a7 93
ecall
blockEmpty:
li a7 4
la a0 blockerror
ecall
li a7 93
ecall
inputempty:
li a7 4
la a0 inputerror
ecall
li a7 93
ecall
```