Titolo: *Relazione conclusiva per il **Progetto Assembly RISC-V** per il corso di Architetture degli elaboratori - A.A. 2021/2022 - Messaggi in Codice*

Autore:
- Cognome e nome: Diciotti Matteo
- Matricola : 7072181
- Mail: matteo.diciotti@stud.unifi.it

Data di consegna: 31/05/2022

Premesse alla trattazione del progetto:
Il codice e` stato prodotto seguendo le specifiche adottate nell'ultima stesura del documento di consegna[1], redatto il 7/4/2022.
Nella seguente trattazione verrà descritta la soluzione adottata per lo svolgimento del progetto, focalizzando l'attenzione sulle scelte effettuate durante la stesura del codice relative agli algoritmi selezionati per ciascuna funzione.
Nella trattazione verranno riportate, laddove ritenuto necessario al fine di agevolare la comprensibilità della descrizione, porzioni del codice.
Nel codice sono presenti alcuni commenti in cui si evidenziano le motivazioni che hanno portato a prendere determinate decisioni durante la realizzazione dello stesso, molte delle quali saranno inserite anche nel presente documento.
Le etichette nel codice seguono una formattazione predefinita: viene inserito il nome della funzione principale a cui appartengono le successive istruzioni seguito dalle varie specializzazioni della funzione stessa separate da un '_'. Capiterà quindi di trovare nomi quali `main_loop_decode_setup` che sta ad indicare che segue una sezione della funzione *main*, in particolare del loop principale della funzione; indica inoltre che sta per essere eseguito un passo della decodifica e che, in questo caso, sta entrando nella fase di set-up.
Per agevolare la leggibilità alle volte sono state inserite delle etichette superflue rispetto all'effettivo uso che ne viene fatto all'interno del programma. Questo e` stato ritenuto potesse aiutare nel comprendere la strutturazione delle funzioni.
La trattazione e` suddivisa per funzione: ogni funzione e` descritta singolarmente data la modularità del codice. L'unica funzione che ne richiama altre e` il *main*, che sarà trattato per primo, rimandando in seguito la descrizione delle specifiche funzioni e non trattandole nel corpo dedicato esclusivamente a questo (eccetto per alcuni accenni necessari).

Trattazione del progetto:
Il programma si suddivide in due parti: **.data** in cui vengono inseriti i valori di input, come ad esempio *myplaintext*, la stringa da codificare, e .**text** nella quale prendono corpo le istruzioni del programma, iniziando dal *main* e passando successivamente alle funzioni chiamate nello stesso *main*, che vedremo in seguito.
E` anzitutto necessario soffermarsi su **.data** che, sebbene rappresenti la porzione minoritaria del codice, nasconde un accorgimento necessario affinché il programma si concluda con successo. Questo accorgimento si nasconde nell'ordine di inserimento dei campi ed in particolare nella posizione che assume *myplaintext*: la stringa da codificare deve essere inserita come ultimo campo.
Il motivo di tale accorgimento e` dovuto al modo di operare di una particolare cifrario, ovvero il *Cifrario per Occorrenze,* che modifica la stringa stessa e che ne aumenta la lunghezza sovrascrivendo il contenuto dei byte successivi. Quindi se *myplaintext* fosse salvato in memoria precedentemente rispetto agli altri campi questo causerebbe la probabile perdita dei campi successivi.
Passando alla porzione di codice **.text** e` necessario premettere che si suddivide anch'essa in due parti: quella iniziale, nella quale giace la funzione principale, ovvero il *main*, e quella immediatamente

1 Documento di consegna integrale: https://e-l.unifi.it/mod/resource/view.php?id=701315

successiva, ovvero la sezione del codice in cui sono state inserite tutte le funzioni che vengono chiamate nel *main.*

Trattiamo il **main**.
Queste istruzioni sono quelle che dettano il flusso del PC, chiamando, a seconda degli input inseriti, la funzione corretta per lo svolgimento delle codifiche e delle decodifiche. Nella fase preliminare vengono inseriti nei registri di memoria statica da *s0* fino ad *s6* i valori e gli indirizzi necessari al *main.*
In particolare sono stati inseriti in *s0*, *s3* e *s5* gli indirizzi nelle quali le tre stringhe passate come input nel **.data** hanno il loro primo carattere in memoria, mentre in *s1*, *s4*, ed *s6* sono stati inseriti gli indirizzi dei rispettivi caratteri null[2]. In *s2* e` stato caricato il valore di *sostK.*
Gli store degli indirizzi di null sono eseguiti successivamente alle chiamate ad una funzione creata appositamente per recuperare l'indirizzo del primo byte in memoria successivo all'indirizzo della stringa contenente il valore zero, l'equivalente al null-char e quindi all'interruzione della stringa. Questa funzione e` stata nominata `calcStringEnd`, abbreviazione di "calculation of string's end".
Eseguiti gli store preliminari e` tempo del loop del *main*, corpo principale della funzione nella quale viene eseguito il riconoscimento dei caratteri inseriti in *mycypher*[3], stringa contenente la sequenza di codifiche da effettuare, e le chiamate alle funzioni di codifica. Terminata questa parte, sempre nel loop, vengono eseguite, ancora attraverso la stringa *mycypher*, le chiamate alle funzioni di decodifica in ordine inverso rispetto alle chiamate di codifica.
Entrando in merito all'algoritmica che permette al loop di eseguire le suddette operazioni e` inizialmente da notare la sezione in cui viene eseguito il set-up: *a0* assume l'indirizzo di *myplaintext*, *a1* quello del null-char della stessa stringa e *s10* assume il valore zero. I valori *a0* e *a1* rappresentano in ogni momento del loop i valori dell'inizio e della fine della stringa da codificare o decodificare. Questa operazione e` stata posta nel set-up perché le implementazioni dei cifrari che verranno utilizzate, dato che definiscono dei metodi mutatori, non cambiano l'indirizzo iniziale della stringa e solamente uno dei cinque cifrari cambia quello finale che, passato come valore di ritorno dalla funzione stessa, viene prontamente aggiornato nel *main*. Il registro *s10* e` invece utilizzato come valore di decisione: fintato che *s10* avrà valore zero il ciclo del *main* eseguirà le codifiche, una volta terminato lo scorrimento della stringa di sequenza invece verrà cambiato il suo valore in 1 (indifferente il valore, e` sufficiente che sia diverso da zero) e la funzione eseguirà solo decodifiche da quel momento in poi. E` stata adottata questa soluzione perché permetteva di utilizzare un'unica funzione per il riconoscimento del cifrario da applicare (`main_loop_selection`). Sebbene la soluzione perda un po` di efficienza il numero di istruzioni prodotte e` proporzionale alla lunghezza di *mycypher* e in relazione agli ordini di complessità di altre funzioni diviene poco significante. Tuttavia questa scelta permette di risparmiare un'ottima porzione di codice che altrimenti avrebbe dovuto essere presente all'interno del *main.*
Il corpo del loop condiviso e` composto quindi dalla sola istruzione `bne s10, zero, main_loop_decode` che indirizza il flusso del PC verso il ciclo corretto.
Immediatamente in seguito si trova la porzione da eseguire nel ciclo in caso di codifica nella quale viene stampata subito la stringa non codificata in *a0* tramite la funzione `printString`[4] per avere una panoramica completa del processo di codifica e decodifica che esegue la stringa, avviene il caricamento del primo byte di *mycypher* e l'aggiorna del puntatore che indirizza al prossimo byte da caricare della stringa. L'istruzione cui punta l'etichetta di questa porzione di codice e` un salto condizionato che viene eseguito non appena il puntatore suddetto raggiunge il null-char di *mycypher,* rimandando all'esecuzione di una istruzione che viene eseguita solo in questo caso, ovvero `li s10, 1`, etichettata da `main_loop_decode_setup`. Questo significa che dal prossimo passo del ciclo il codice appena descritto non verrà più eseguito poiché risulterebbe superfluo e nocivo per l'efficienza

2 Per agevolare la fluidità di linguaggio l'indirizzo in cui una stringa ha il proprio primo carattere sarà chiamato l'*indirizzo della stringa* mentre quello in cui e` presente il carattere di null (\0) sarà chiamato *null-char della stringa.*
3 In seguito nota anche come *stringa di sequenza*.
4 Vedi la trattazione a pag. 7: *Funzione di Stampa*.

del programma. Il ciclo di decodifica e` simile come struttura a quello di codifica, ma vengono modificate tre cose sostanziali: la prima istruzione e` l'aggiornamento dello stesso puntatore usato nel ciclo di codifica per riportarlo a puntare ad un valore interno di *mycypher* e non al null-char, la seconda modifica e` che l'aggiornamento avviene in senso inverso, quindi invece di incrementare il puntatore lo decrementiamo, e la terza e` che il ciclo termina quando il puntatore supera il primo elemento della stringa (superato in senso inverso, quindi dovrà essere minore di *s5*).
A questo punto trattiamo velocemente il metodo di selezione con cui viene deciso quale cifrario utilizzare basandosi sul valore caricato da *mycypher.* E` stato strutturato un albero binario di ricerca per trovare quale cifrario chiamare poiché permetteva un risparmio medio di circa una istruzione a ciclo. Sebbene non sia un guadagno eccessivo, che comunque non e` confrontabile con gli ordini di complessità delle funzioni di cifratura o calcolo del null-char, l'implementazione di tale soluzione richiede un'unica istruzione in più` rispetto all'implementazione di un algoritmo sequenziale e quindi risulta conveniente. Nel caso in cui il valore non sia presente tra quelli prescritti nel documento di consegna[1], ovvero 'A','B','C','D' ed 'E', che in codice ASCII equivalgono a 65, 66, 67, 68 e 69 in notazione decimale, e` stato deciso di interrompere immediatamente l'esecuzione del programma senza successo, ovvero non eseguendo mai il ciclo di decodifica.
All'interno dell'albero di selezione si rimanda a delle sezioni del *main* che sono adibite al set-up specifico per il cifrario selezionato e quindi alla chiamata al cifrario stesso. Nel caso del cifrario C, ovvero *Cifrario per Occorrenze,* e` necessario un ulteriore passaggio al termine dell'esecuzione della funzione di codifica o decodifica per reimpostare i puntatori di inizio e fine stringa codificata/decodificata. Per mantenere coerenza nel codice sono stati inseriti anche i set-up dei cifrari D ed E, sebbene questi non eseguano altro oltre alla chiamata (che quindi poteva essere effettuata direttamente nell'albero di selezione).
Una volta eseguite tutte le codifiche e tutte le decodifiche il ciclo si interrompe e giunge ad `end_main` nel quale il flusso del PC si interrompe tramite la chiamata a sistema.

Il primo metodo inserito nella sezione delle funzioni all'interno del codice e` la *Cifratura per Sostituzione*, ovvero la codifica A.
Trattiamo il **Cifrario per Sostituzione.** La funzione e` nominata `codeA` in relazione alla lettera predisposta come scelta di tale cifrario nella *mycypher*.[5]
La funzione richiede 4 parametri di input:

- *a0* = indirizzo della stringa da codificare/decodificare;
- *a1* = null-char della stringa da codificare/decodificare;
- *a2* = *sostK;*
- *a3* = selezionatore di codifica o decodifica (*s10*).

Questa funzione non restituisce alcun valore, tuttavia, come descritto in precedenza, non modifica gli indirizzi interni ad *a0* e ad *a1*. Questa accortezza ci permette di non dover eseguire ogni volta il set-up completo per ogni chiamata di cifratura che viene effettuata.
La funzione modifica la stringa sommando ad ogni byte contenente il valore intero corrispondente ad una lettera nella codifica ASCII il valore di *sostK* e inserendo ivi la lettera della stessa grandezza (le maiuscole rimangono maiuscole e viceversa) che corrisponde alla traslazione ciclica all'interno dell'alfabeto di un numero pari a *sostK* posizioni rispetto a carattere contenuto nel byte della stringa.
Questa funzione, con un'opportuna modifica del segno di *sostK*, può eseguire sia codifica che decodifica e per questo e` stato passato il valore contenuto in *a3* che fungerà da valore booleano.
La funzione esegue una prima parte di set-up nella quale sono da notare, in particolare, le manipolazioni che vengono operate al valore di *sostK*: innanzi tutto viene calcolato il valore del modulo di *sostK* per 26, il numero di lettere presenti nell'alfabeto; dopo di che viene cambiato di segno

5 Seguiranno la medesima logica tutti i cifrari seguenti. Non verrà quindi ripetuta la motivazione della scelta.

se la funzione sta eseguendo una decodifica ed infine viene calcolato il modulo positivo del numero in caso questo risulti inferiore a zero.[6]
A questo punto inizia un ciclo che si interromperà non appena il puntatore al prossimo indirizzo della stringa (*t0*) raggiungerà il null-char. All'interno del ciclo viene caricato il byte puntato e, una volta aggiornato il puntatore, vengono eseguiti una serie di controlli al fine di comprendere a quale gruppo di lettere appartiene. Nel caso il valore non corrisponda ad una lettera viene eseguito il prossimo passo del ciclo, altrimenti la cifratura: se il valore del byte rappresenta una lettera in *t2* sarà stato caricato il valore dalla lettera *'a'* appartenente allo stesso gruppo, ovvero 65 per le lettere maiuscole e 97 per le lettere minuscole. Questo valore permette di comprendere la posizione del carattere del byte all'interno dell'alfabeto, la sua *posizione relativa*[7], tramite una semplice sottrazione. A questo punto prendono luogo la somma del valore modificato di *sostK*, il modulo del valore così trovato per 26 e la reintroduzione del valore sottratto della *'a'(o 'A').* Viene eseguito lo store del valore così calcolato nella posizione dell'ultimo byte caricato, ovvero *t0-1* dato che *t0* era già stato incrementato nel caso in cui il valore del byte non rappresentasse una lettera.
Terminato il ciclo la funzione esegue un salto ad *ra*.

Il secondo cifrario presente e` noto come **Cifrario a Blocchi**.
La funzione e` nominata `codeB`. La funzione richiede 5 parametri di input:

- *a0* = indirizzo della stringa da codificare/decodificare;
- *a1* = null-char della stringa da codificare/decodificare;
- *a2* = indirizzo di *blocKey;*
- *a3* = null-char di *blocKey;*
- *a4* = selezionatore di codifica o decodifica (*s10*).

Come il *Cifrario a Sostituzione,* il *Cifrario "Dizionario"* e il *Cifrario "Inversione"* non ha valori di ritorno ma non modifica i valori di *a0* e *a1.* La funzione `codeB` esegue sia la codifica che la decodifica, risparmiando una grande porzione di codice che sarebbe stata altrimenti ripetuta al costo di una istruzione a ciclo.
Sebbene sia evidentemente molto simile al cifrario per sostituzione e` da notare che in questo caso si cifrano anche simboli e cifre e che non si calcola più la traslazione ciclica all'interno dell'alfabeto ma si calcola all'interno dell'intervallo di valori compresi in [32, 127].
In questo caso l'algoritmo e` un po' più complesso. Nella fase di set-up e` da notare il calcolo della lunghezza della stringa *blocKey,* cosa che non viene eseguita altrove. Questa e` necessaria per impostare un loop nel quale si carica un'unica volta i valori dei byte della *blocKey* ed un'unica volta i valori dei byte della stringa da cifrare, risparmiando quindi numerose istruzioni, dell'ordine della differenza tra la lunghezza della stringa da cifrare e la lunghezza della *blocKey.* Nel caso la *blocKey* sia più lunga della stringa il ciclo si interrompe appena tutti i byte sono stati cifrati. Per comprendere quando questo avviene controlla se il primo byte da cifrare con uno specifico valore calcolato dalla *blocKey* sia superiore o uguale al null-char della stringa. Oltre a questo controllo il loop esegue il calcolo dell'indirizzo da codificare con il valore appena calcolato dalla *blocKey* ed esegue il controllo sul valore di *a4,* ovvero indica al PC se eseguire una codifica o una decodifica.
La sezione relativa alla codifica esegue il caricamento del primo byte da cifrare, lo somma al valore ottenuto dalla *blocKey*, ne esegue il modulo per il numero di elementi nell'intervallo di destinazione

6 Al fine di accorciare la trattazione questo e` l'unico caso in cui viene presa in esame ogni istruzione, in seguito verrà descritto più generalmente l'algoritmo dando per scontato cose come l'aggiornamento dei puntatori ecc.

7 Con posizione relativa si intende l'indice intero che un valore assumerebbe se il conteggio iniziasse a partire dal primo elemento di un gruppo di valori. Nel caso dell'alfabeto la posizione relativa di 'd' e` 4, mentre nel caso della stringa "Assembly ti amo!" la posizione relativa della 'y' e` 8 (o 7 nel caso in cui si inizi a contare da 0)

(96) e ci somma il minimo valore dell'intervallo di destinazione (32). Cosi facendo si ottiene un valore compreso tre 32 e 127 che rappresenta la codifica del byte e viene eseguito lo store di questo valore nell'indirizzo da cui e` stato caricato il byte della stringa. Infine viene calcolato il prossimo indirizzo da codificare sommando all'indirizzo del byte appena codificato la lunghezza della *blocKey*. Se questo indirizzo appartiene alla stringa allora inizia un altro passo del ciclo di codifica con la stessa chiave di cifratura (il valore del byte caricato dalla *blocKey)*, altrimenti passa al prossimo byte della *blocKey*.
Prima del ciclo di decodifica viene sommato al valore calcolato dalla *blocKey* il valore minimo dell'insieme su cui si cerca la traslazione (32) per risparmiare alcune istruzioni (evitando di eseguire due sottrazioni ad ogni passo del loop) e il ciclo carica il byte da decodificare, gli sottrae il valore appena calcolato e ci somma il numero di elementi dell'intervallo di destinazione (96) finché il valore risultante non e` compreso tra 32 e 127. Sono state quindi eseguite le operazioni inverse rispetto a quelle di codifica. Infine esegue lo store del valore così ottenuto e calcola, come per il ciclo di codifica, il prossimo indirizzo da decodificare con la stessa chiave calcolata da *blocKey*.
Il ciclo principale termina anche nel caso in cui tutti i valori della *blocKey* sono stati caricati.
Terminato il ciclo la funzione esegue `jr ra (o ret` in pseudoistruzioni`)`.

Il *Cifrario per Occorrenze* e` l'unico cifrario per cui e` stato scelto di dividere la funzione di cifratura da quella di decifratura. Questa decisione e` dovuta al fatto che gli algoritmi utilizzati nelle due funzioni eseguono operazioni completamente distinte. A conferma di questo fatto si tenga presente il calcolo di complessità: la codifica risulta avere ordine $O(n^2)$, dove n e` la lunghezza della stringa da codificare, mentre la decodifica un ordine O(n), considerando come operazione fondamentale il caricamento dei byte dalla memoria nei registri.
Le uniche caratteristiche che sono in comune, data la definizione del cifrario, sono i parametri di input e il valore di ritorno: gli input sono due, ovvero gli indirizzi che delimitano la stringa da codificare (inizio stringa e null-char), e il valore di ritorno e` l'indirizzo del null-char della stringa codificata.
Trattiamo ora esclusivamente la **Funzione di codifica per il Cifrario per Occorrenze**.
Durante la progettazione dell'algoritmo e` stata ritenuta necessaria una stringa d'appoggio. Questo perché per ogni carattere la codifica non equivale ad un unico byte, come per le altre codifiche, ma per ogni byte da codificare la sequenza codificata occuperà almeno il doppio della memoria (mediamente molto più del doppio). Vista la maggiore occupazione di memoria della stringa codificata e` stata valutata come più efficiente l'esecuzione della copia della stringa da codificare nella memoria dinamica, ed in questo caso nello stack, e poi l'esecuzione della codifica lavorando sulla copia e salvandola nella posizione corretta di memoria, ovvero dove giaceva la stringa da codificare. Cosi facendo si dimezzano (almeno) i tempi necessari per la gestione della stringa d'appoggio. La decisione di utilizzare lo stack risulta evidente nel momento in cui si esegue una lettura del loop di codifica: ad ogni passo si esegue il caricamento del byte puntato da *sp* e al ciclo dopo si passa a quello successivo, come in una pila. In realtà sulla pila in questione vengono eseguite anche delle operazioni distinte all'interno del ciclo, ma questo non ne modifica il comportamento.
Eseguita la copia della stringa segue il set-up del loop di codifica nel quale vengono operati gli store dei valori necessari all'interno del codice e la copia di *sp*: quest'ultimo occorre per il calcolo della posizione relativa di un carattere all'interno della stringa essendo sufficiente l'operazione di una sottrazione per ottenere il valore ricercato. Uno dei valori inseriti negli store propedeutici al loop e` quello presente in *a7,* ovvero 129: questo valore e` stato selezionato come flag per riconoscere quali caratteri sono già stati codificati e che quindi devono essere non considerati nuovamente. Il valore 129

sarà inserito al posto di tutti i byte di cui e` già stata inserita l'occorrenza nella stringa. Il loop se troverà un byte contenente 129 salterà al passo successivo del ciclo e quindi al prossimo byte. Il motivo della scelta di 129 e` che il primo valore a non avere un carattere assegnato in codifica ASCII e` il suddetto. E` stato preso in esame anche l'inserimento di valori illegali per le varie codifiche, come ad esempio 10 (già inserito tra i valori necessari per la codifica), pero` questo, sebbene completamente coerente e funzionante, e` stato ritenuto ledesse leggermente la generalità del codice e quindi e` stato preferito l'inserimento di un valore più stabile in questo senso.
Nel loop una volta caricato il byte dalla stringa e valutato come differente da 129 viene inserito immediatamente nella stringa codificata. Infatti se il valore e` differente da 129 significa che in nessun passo precedente del ciclo e` stata evidenziata un'occorrenza di quel carattere e quindi e` la prima apparizione di quel carattere all'interno della stringa. A questo punto viene eseguita la ricerca del carattere all'interno della stringa a partire proprio dalla posizione in cui e` stato appena trovato (le posizioni precedenti sono già state valutate tutte) e una volta riconosciuta una equivalenza il ciclo salva il valore 129 nella posizione del byte caricato e l'immediato inserimento dell'occorrenza (codificata come richiesto dal progetto, ovvero con un byte per ogni cifra dell'occorrenza, in modo tale che la stampa a video mostri la posizione relativa) all'interno della stringa finale preceduto da un simbolo '-' (45 in ASCII). Il calcolo di quali cifre inserire e` eseguito attraverso un ulteriore breve ciclo nel quale viene eseguito lo store del modulo della cifra dell'occorrenza per 10 nello stack (dalla prima posizione di cui non ci interessa più mantenere l'informazione, ovvero quella su cui il ciclo principale e` attualmente, dato che il valore e` già salvato in un registro) e viene diviso il valore dell'occorrenza per 10. Questo ciclo si interrompe appena il valore dell'occorrenza diviene zero a causa delle divisioni eseguite e cosi facendo, appena passa al ciclo successivo, ovvero quello adibito allo store delle occorrenze, calcola la codifica dell'ultimo valore inserito nello stack e lo inserisce nella stringa, finché non sono stati codificati e quindi salvati tutti i valori aggiunti allo stack nel calcolo del modulo (l'ordine di inserimento nella stringa e` corretto, la cifra più significativa sarà inserita per prima). Alla fine della ricerca di ogni carattere la parte finale del ciclo inserisce uno spazio come richiesto dalle specifiche del cifrario. Uscita dal ciclo, la funzione termina dopo aver sostituito all'ultimo carattere, ovvero uno spazio data l'implementazione del ciclo, il carattere null e dopo aver aggiornato *a0* con il valore di ritorno.

Trattiamo nell'immediato seguito la **Funzione di decodifica per il Cifrario per Occorrenze**.
Con la stessa logica con cui e` stato deciso nella funzione dei codifica del medesimo cifrario di eseguire la copia della sequenza di byte più corta per risparmiare istruzioni, nella presente funzione si esegue la copia della stringa codificata, anziché di quella da modificare, nelle posizioni corrette di memoria. E` infatti necessaria memoria ausiliaria anche in questo caso dato che verranno inseriti i caratteri all'interno della stringa via via che saranno calcolate le occorrenze in cui posizionarli. La stringa di appoggio sarà un vettore di char che seguirà il null-char presente in *a1*. E` stato valutato anche l'inserimento della sequenza decodificata nello stack, ma non possedendo a priori la dimensione che occuperà (a meno di non eseguire un primo scorrimento della stringa da codificare con un passo di selection-sort sulle occorrenze, non molto efficiente quindi) l'utilizzo dello stack risultava un po` forzato.
L'algoritmo implementato per la decodifica, escludendo il ciclo di copia della stringa d'appoggio che risulta molto banale, esegue uno scorrimento sulla stringa codificata, carica il carattere in un registro di memoria e ne calcola tutte le varie occorrenze inserendo di volta in volta il carattere nella porzione

di memoria ausiliaria selezionata. Quindi il primo carattere verrà inserito in posizione *a1+1,* ovvero null-char della stringa da codificare più uno, e gli altri in posizione coerente basandosi sull'indirizzo del null-char. Dopo aver inserito tutte le occorrenze di un carattere, la funzione confronta l'ultima posizione in cui questo e` stato inserito con un registro (*a3*) in cui e` presente l'indirizzo maggiore in cui e` stato inserito un qualsiasi valore tra tutti quelli fino ad ora decodificati. Il mantenimento del valore dell'occorrenza più elevata ci serve per l'inserimento del null-char nella stringa d'appoggio. Una volta finito il ciclo la stringa passata come input sarà seguita dal null-char e dalla stringa decodificata. Viene inserito quindi il carattere di null nella posizione indicata da *a3+1* e la copia della stringa dalla posizione *a0* a seguire.
Infine viene aggiornato il registro *a0* perché contenga l'indirizzo del null-char della stringa decodificata.

Trattiamo il **Cifrario "Dizionario"**.
Come per il *Cifrario per Occorrenze* anche in questo caso gli input sono i limiti della stringa da codificare. La funzione non restituisce niente, ma mantiene la proprietà di invarianza dei valori in *a0* e *a1*. La funzione modifica i valori relativi a lettere e a cifre, i simboli rimangono immutati.
L'algoritmo che esegue la funzione e` inserito in un ciclo che scorre ogni byte della stringa e, similmente a come svolto per il *Cifrario per Sostituzione*, comprende a quale gruppo di valori appartiene. Se appartiene all'intervallo [48, 57] allora corrisponde ad un numero e sarà codificato tramite la sostituzione del valore all'interno del byte di memoria dal valore corrispondente alla sottrazione da 105 del valore del byte. Questa codifica corrisponde all'individuazione del complementare di 9 del valore, intatti 48+57=105, che equivale a ASCII(0)+ASCII(9). Se invece il valore appartiene agli intervalli [65, 90] o [97, 122] allora il valore corrisponde ad una lettera e in questo caso maiuscole e minuscole sono codificate ugualmente. Infatti la richiesta della consegna e` che il valore codificato equivalga al carattere in medesima posizione nell'alfabeto invertito cambiato di grandezza. Questo risultato lo si ottiene semplicemente eseguendo la sottrazione a 187 del valore del byte. Infatti 65+122=187 che equivale a ASCII(A)+ASCII(z) (la somma dei valori rimane costante per ogni coppia di valori).
Dato che la funzione calcola di fatto i complementari ad un valore per ogni byte che non sia un simbolo, e` evidente che la funzione sia l'inversa di se` e che quindi un'applicazione consecutiva non faccia che restituire la stringa decodificata.

Trattiamo il **Cifrario "Inversione"**.
L'implementazione di questa funzione e` la più semplice a livello algoritmico tra tutti i cifrari trattati. I valori di input sono i limiti della stringa da codificare e non restituisce valori di ritorno.
L'algoritmo consiste in un ciclo nel quale vengono caricati contemporaneamente il primo e l'ultimo carattere effettivi della stringa (quindi non il null-char, che rimane nella posizione originale) e vengono salvati nelle posizioni invertite. Il ciclo termina quando i due puntatori che scorrono nei sensi opposti della stringa si sovrappongono o si scambiano, che equivale all'interruzione del ciclo dopo n/2 passaggi, con n la lunghezza della stringa.
Anche in questo caso la funzione inversa e` equivalente alla funzione di codifica.

Trattiamo le due funzioni ausiliare create per modulare il codice.
Entrambe hanno un unico valore di input: l'indirizzo della stringa da trattare.
La prima e` la **Funzione per il calcolo del Fine Stringa.**

L'algoritmo della funzione equivale allo scorrimento di un puntatore su una stringa finché il contenuto del byte puntato non equivale a zero. Quel byte e` quindi il null-char della stringa.
La funzione restituisce l'indirizzo del carattere di null.

La **Funzione di Stampa** e` un metodo creato per compattare, all'interno del codice, le istruzioni per il set-up della stampa di una stringa e la consecutiva stampa di un capoverso (due ne sono stati inseriti in realtà per rendere più comprensibile quando termina la stampa di una codifica). Come ultima istruzione e` posto il ripristino dell'indirizzo della stringa stampata perché la funzione possieda la proprietà di invarianza dei registri posti in input.

Test di corretto funzionamento del programma:
Nella seguente sezione verranno riportati alcuni esempi di applicazione del programma per dimostrare la correttezza delle implementazioni proposte. Verranno mostrati i valori passati in input e i conseguenti output. Per rendere immediata l'evidenza della correttezza delle implementazioni e` stato deciso di mostrare anzitutto gli esempi presenti nel documento di consegna, più un altro paio di casi.
La colonna degli output mostra le stringhe derivanti dall'applicazione di ogni codifica presente in *mycypher*.

INPUT	**OUTPUT (STRINGA COMPLETAMENTE CODIFICATA)**
myplaintext = "AMO AssEMBLY" mycypher = "A" sostK = -2	YKM YqqCKZJW
myplaintext = "LAUREATO" mycypher = "B" blocKey = "OLE"	[MZaQFc[
myplaintext = "sempio di messaggio criptato - 1" mycypher = "C"	s-1-13-14 e-2-12 m-3-11 p-4-24 i-5-9-18-23 o-6-19-28 -7-10-20-29 d-8 a-15-26 g-16-17 c-21 r-22 t-25-27 --30 1-31
myplaintext = "myStr0ng P4ssW_" mycypher = "D"	NBhGI9MT k5HHd_
myplaintext = "BUONANOTTE" mycypher = "E"	ETTONANOUB
myplaintext = "Ciao Mondo!" mycypher = "AED" sostK = -765	Rxpd Bdcsd! !dscdB dpxR !WHXWy WKCi
myplaintext = "Ciao Mondo!" mycypher = "CEB" blovKey = "OLE ole ale ALE !!!"	C-1 i-2 a-3 o-4-7-10 -5 M-6 n-8 d-9 !-11 11-! 9-d 8-n 6-M 5- 01-7-4-o 3-a 2-i 1-C @=2aOeRDAdRN!B2-avn/,5q\cRtN;Es.m%rnJa@9H

Soffermandosi sul sesto test valutiamo l'andamento del terzo byte da codificare, quello relativo alla 'a', come esempio. Inizialmente si esegue `codaA`, quindi si codifica 'a' con (-765)%26, ovvero 15, e quindi 'a' diventa 'p'; si esegue poi `codeE`, quindi in posizione 3 dovrà essere contenuto il valore in posizione 9 (*I -3 +1)*, ovvero 's', ed equivale a quanto ottenuto. Infine il complemento di 's' nell'alfabeto inverso e mutato di dimensione e` 'H', quindi applicando `codeD` risulta quanto atteso.