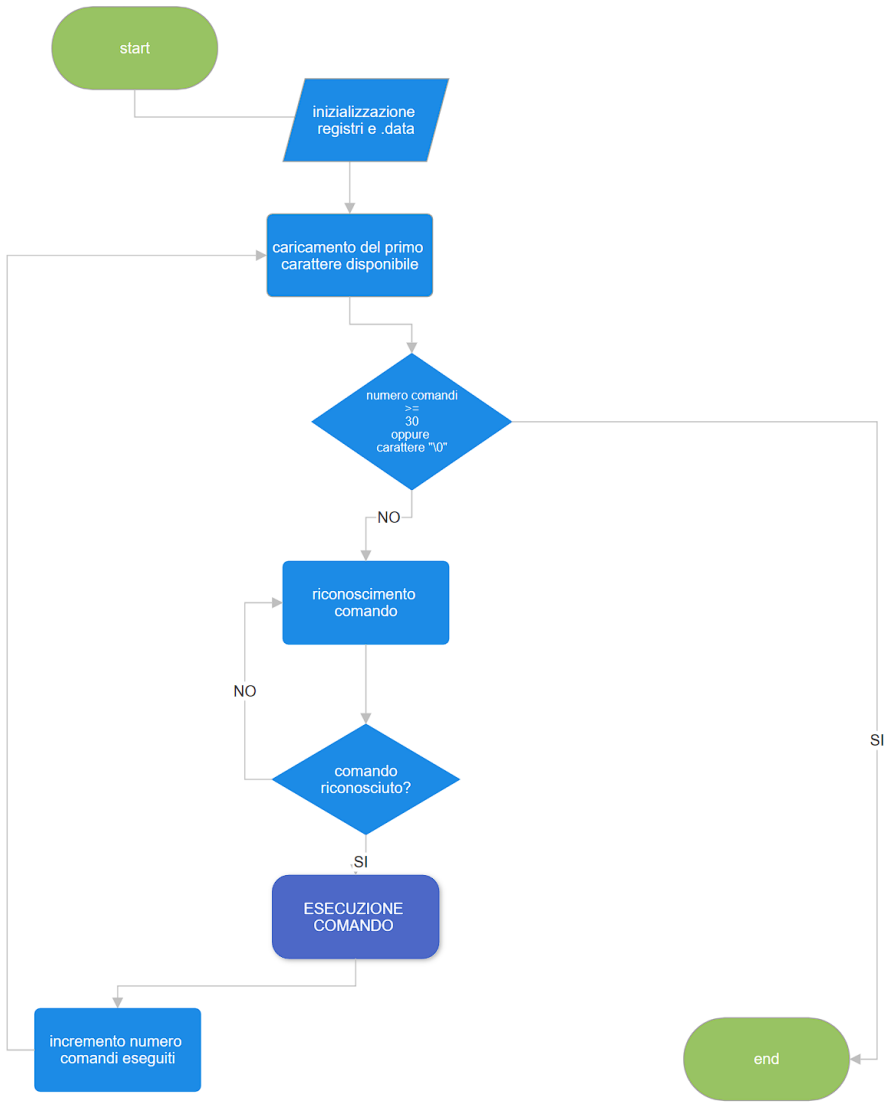
Relazione Progetto architetture degli elaboratori 2022/2023

Informazioni

Autore: Daniele Fallaci
Email: daniele.fallaci@edu.unifi.it
Matricola: 7112588

Descrizione progetto

Realizzare una lista concatenata circolare tale che ogni elemento è caratterizzato da 5 byte in memoria contigui:
1 byte che contiene il valore ascii dell'elemento e 4 byte che contengono l'indirizzo dell'elemento successivo.
La lista viene gestita da una stringa in input che utilizza come carattere separatore una tilde (~).
i comandi da realizzare sono ADD(), DEL(), PRINT, REV, SDX, SSX e SORT.



Sezione .DATA e uso dei registri

Sezione data.

- **input**: la stringa di input dei comandi
- **head**: una word che contiene l'indirizzo al primo elemento
- **print_Str**: una stringa usata per presentare meglio il print

In generale i registri sono usati come spazio di appoggio in molte parti del progetto, ma alcuni hanno degli scopi più precisi.

Registro	**Utilizzo**
a1	numero di istruzioni usate
a2	massimo istruzioni permesse
t5	contiene 0xFFFFFFFF cioè il valore puntatore di lista nulla
s1 ed s2	limite superiore ed inferiore dei caratteri inseribili. ASCII: 125 (}) ASCII: 32 ( )spazio
s6	ASCII 126 per delimitare i comandi
t1	indirizzo della stringa input
t0	carattere della stringa input che si confronta
s4	carattere gestito da ADD e DEL
s3	valore da confrontare nella analisi della stringa
s5 e s4	limite superiore e inferiore di confronto nel sort

Ad ogni comando eseguito si incrementa il registro a1.

Sezione .TEXT

Riconoscimento del comando

Presa in input la stringa viene iniziata a scorrere mediante registri di appoggio, viene caricato il primo byte non ancora analizzato e si va nella label di riferimento per proseguire la analisi.
Viene usato **next_letter** per ottenere il carattere successivo.
(Gli spazi vengono ignorati a inizio e fine comando corretto).

- ricevo una A e vado nella sezione di controllo di ADD
- ricevo una D e vado nella sezione di controllo di DEL
- ricevo una P e vado nella sezione di controllo di PRINT
- ricevo una S e controllo SSX, SDX o SORT a seconda degli input
- se ricevo un carattere diverso da quello atteso vado al comando successivo e ignoro gli spazi con **ignore_until_tilde** e **gotoValidLetter**.
- se il carattere è di fine stringa finisce il programma nella label **end**

una volta individuato il comando il programma controlla, tramite **after_brackets,** se contiene prima del successivo dei caratteri errati diversi dallo spazio.
(es: “ADD(A) a~...”).

```
## Trovare la lettera successiva nell'input
next_letter:
    addi t1,t1,1 #next letter
    lb t0, 0(t1)
    jr ra
```

```
ignore_until_tilde:
beq t0,s6,analize_string

loop_tilde:
    beq t0,zero,end #####fine se \0
    addi t1,t1,1 #next letter
    lb t0, 0(t1)
    bne t0,s6,loop_tilde

gotoValidLetter:
    beq t0,zero,end
    jal next_letter

    beq t0,s2,gotoValidLetter
    j analize_string
```

```
## Questo metodo serve per individuare se e' un reale comando o una istruzione tipo:
## ADD(A)AA ~ REV    A~
after_brackets:
    addi sp,sp,-4
    sw ra,0(sp)
    jal next_letter
    lw ra, 0(sp)
    addi sp,sp,4
    beq t0,zero,end_brackets ## l'ultima istruzione ha '\0'
    beq t0,s6,end_brackets   ## senno' ha '~'
    bne t0,s2,ignore_until_tilde
    j after_brackets
end_brackets:
    jr ra
```

ADD

Il comando prende in input tra le parentesi un carattere compreso tra 32 e 125, in caso contrario si va al comando successivo.

- Si trova i primi 5 byte contigui liberi tramite **find_space**;
  questo metodo dato un indirizzo nel registro a3, restituisce un indirizzo con 5 byte successivi liberi nel registro a3
- Nel caso in cui si sta aggiungendo il primo elemento abbiamo la label **first_add**

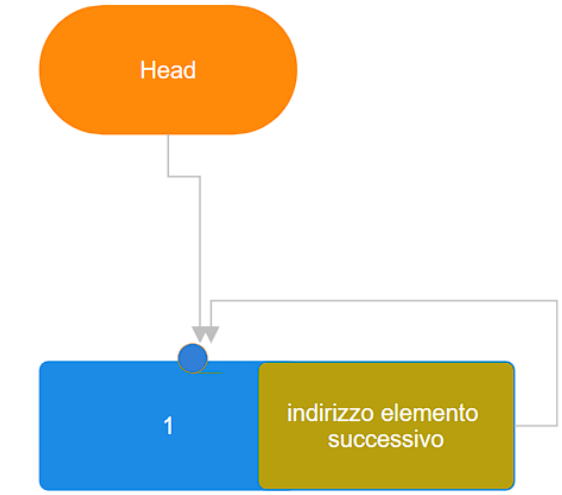
```
add:
    lw t3 head
    lw s10,head
    beq t3,t5, first_add    #controllo lista vuota
 loop_to_last:
     addi s11,t3,0
     lw t3,1(t3)
     bne t3,s10,loop_to_last
     addi a3,s11,0
     addi a6,s11,0

     addi sp,sp,-4
     sw ra,0(sp)
     jal find_space
     lw ra, 0(sp)
     addi sp,sp,4

    sw a3,1(a6)             #upd puntatore in memoria
    sb s4,0(a3)             #metto il valore
    lw t3,head
    sw t3,1(a3)
    j end_ADD
 first_add:
    la a3,head
    addi a6,a3,0

    addi sp,sp,-4
    sw ra,0(sp)
    jal find_space
    lw ra, 0(sp)
    addi sp,sp,4

    sw a3,0(a6)#upd head in memoria
    sb s4,0(a3)#metto il valore
    sw a3,1(a3)
 end_ADD:
    addi a1,a1,1
    jr ra

find_space:
                #a3 conterra' il primo indirizzo da cui iniziare ad analizzare
    lb s11, 0(a3) #vedere se e' vuoto il primo byte che conterra l'elemento
    addi a3,a3,1  #byte successivo
    bne s11, zero,find_space #se non lo e' rianalizzo quello dopo
    lw s11,0(a3)  #4 byte successivi(notare sono gia andato avanti di uno quindi off=0)
    bne s11,zero,find_space
    addi a3,a3,-1
                #a3 conterra' 5 byte liberi successivi
    jr ra
```

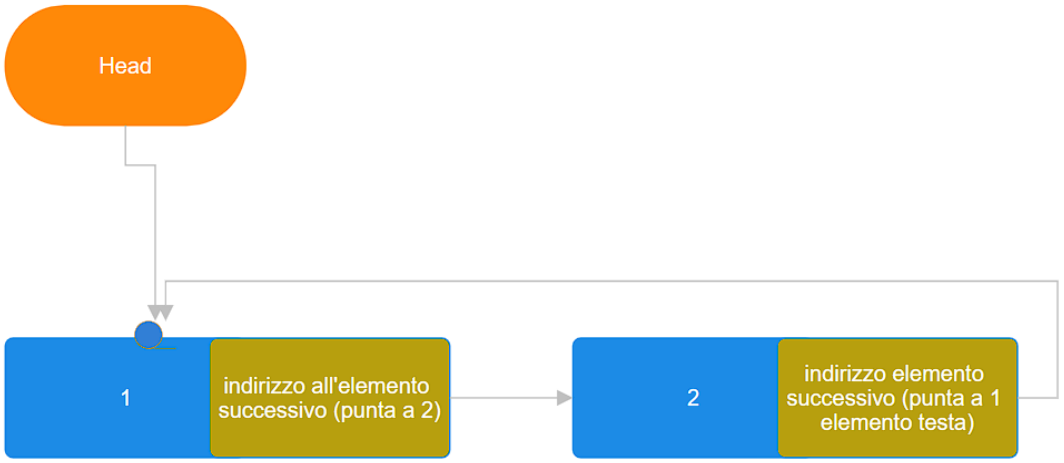
Esempio di ADD

input = “ADD(1)~ADD(2)”

in blu l’elemento aggiunto i 4 byte successivi rappresentano l’indirizzo all’elemento successivo.

Logica dietro l’istruzione add:





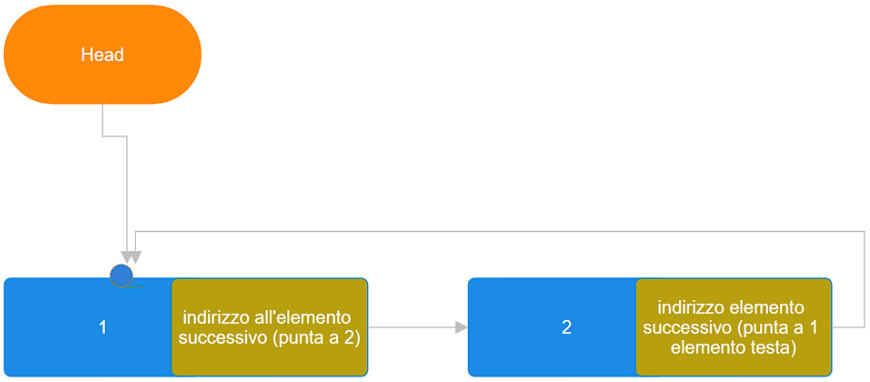
DEL

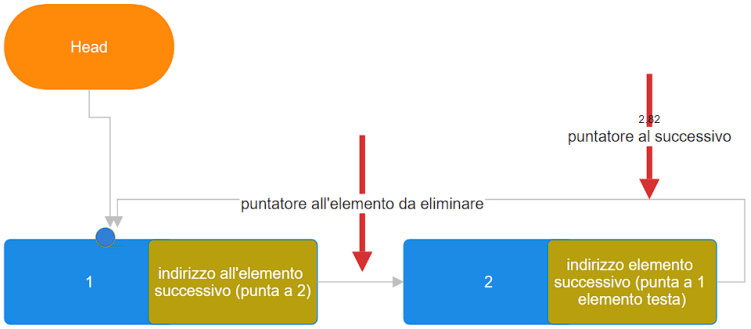
Il comando prende in input tra le parentesi un carattere tra 32 e 125, proprio come ADD. Una volta accertato che il carattere è nel range giusto, che avviene prima della chiamata al comando:

- si controlla che la lista abbia elementi
- si scorre successore e predecessore
- ogni volta che si trova un elemento uguale a quello contenuto in s4 lo si elimina andando alla label **rem** (che è un loop)
- aggiorna i puntatori

```
del:
    lw t3, head                 #s4 conterra' l'elemento da togliere
    la s10,head #phead
    lw s11,head
    beq t3,t5,end_del

find_rem:
    lb s9,0(s11) #letter
    lw s7,1(s11)
    beq s9,s4,rem
update_pointers:

    lw s10,0(s10)               #scorro il predecessore
    addi s10,s10,1

    lw s7,1(s11)                #scorro l'elemento attuale
    beq s7,t3,upd_coda
    lw s11,1(s11)
    j find_rem
rem:
    lw s11,1(s11)
    lb s7,0(s11)

    beq t3,s11,end_rem
    beq s7,s4,rem
end_rem:
    lw s7,1(s11)
    sw s11,0(s10)
    lw t3,head
    j update_pointers

upd_coda:

    lw t3,head
    lb s7,0(t3)

    bne s7,s4,upd_not_void
    la t3,head
    sw t5,0(t3)
    j end_del

upd_not_void:
    sw t3,1(s11)

end_del:
    addi a1,a1,1
    jr ra
```

esempio: data una lista con 2 elementi (1,2) si vuole procedere ad eliminare 2 con l'istruzione DEL(2)





cambio il puntatore al successivo dell’elemento 1 e al suo posto metto il puntatore al successivo dell’elemento 2.

l’elemento 2 rimane escluso dalla lista.

PRINT

Il comando, come suggerisce il nome, stampa tramite le system call il contenuto della lista. In particolare abbiamo queste fasi:

- stampa a video della stringa **print_Str** (presentazione del print)
- scorrimento della lista con stampa di tipo char per ogni ascii
- chiusura del print con il carattere “}”

```
print:
    lw t3,head
    li a7,4
    la a0, print_Str
    ecall

    li a7,11
    lw s11, head
    #li s9, 0

    beq s11, t5, fine_stampa #vuoto

    stampa:

    lb a0, 0(s11)
    ecall

    lw s11,1(s11)
    beq s11,t3,fine_stampa

    j stampa

    fine_stampa:

    li a0,125 #}
    ecall

    li a0,32
    ecall

    addi a1,a1,1
    jr ra
```

REV

Data la lista se ne inverte il contenuto.
Per implementare questo metodo ho usato come appoggio lo stack e due loop:

- push_rev: aggiungo allo stack i caratteri in ordine (1 byte)
- pop_rev: si estrae dall'ultimo inserito al primo inserendo nella lista

Così facendo ottengo la lista invertita scorrendo due volte la lista tramite i loop.

```
rev: #DONE
    lw t3, head
    lw s11,head
    beq s11, t5, end_rev

  push_rev:

    lb a0, 0(s11) #character to save

    #salvataggio del char nella pila
    addi sp,sp,-1
    sb a0,0(sp)


    lw s11,1(s11)
    beq s11,t3,pop_rev
    j push_rev

  pop_rev:
    lw s11,head

  ciclo_rev:

    lb a0,0(sp)
    addi sp,sp,1

    sb a0,0(s11)
    lw s11,1(s11)
    beq s11,t3,end_rev
    j ciclo_rev
  end_rev:
     addi a1,a1,1
     jr ra
```

SDX

Questo metodo fa uno shift a destra della lista.
tramite il loop interno si va all'ultimo elemento della lista(salvato in s8) e lo si mette come testa.

```
sdx: #DONE
    lw s11,head

    lw s10, head
    la s9,head

   loop_sdx:

      addi s8,s11,0
       lw s11,1(s11)
       beq s11, s10,end_ssx

       j loop_sdx

   end_ssx:
       sw s8,0(s9)
       addi a1,a1,1

     jr ra
```

SSX

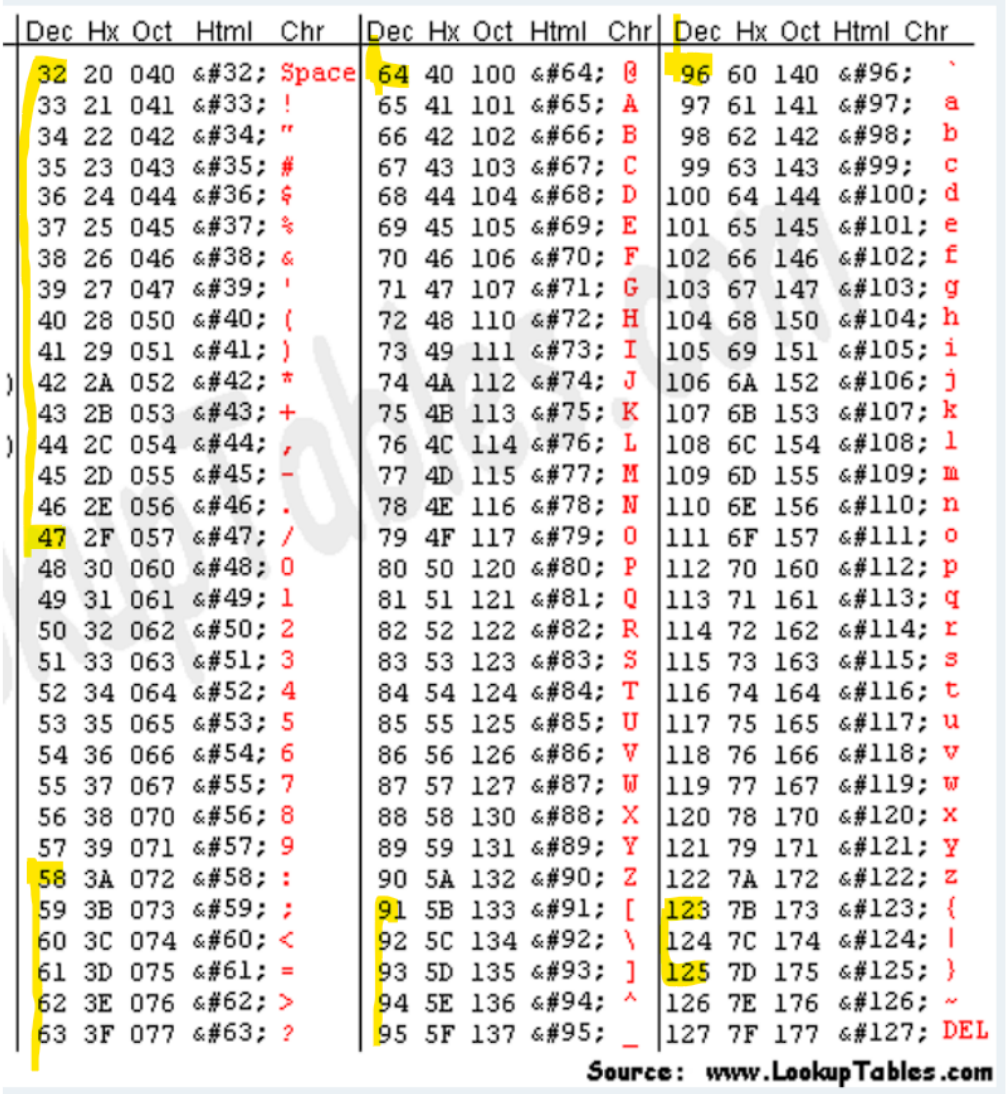
Scorre di un elemento la lista e si salva l'indirizzo in head.

```
ssx: #DONE
  #modificata la testa:
    lw s11, head
    #addi s11,s11,1
    lw s11,1(s11)
    la s10,head

    sw s11,0(s10)

    addi a1,a1,1
    jr ra
```

SORT

ANALISI PRELIMINARE

Dobbiamo rispettare il seguente ordine
simboli < numeri < minuscole < maiuscole (ASCII fino al 125)



Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
32	20	040		Space	64	40	100	@	@	96	60	140	`	`
33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Come osserviamo da questa tabella un semplice ordinamento per ASCII rispetto al valore non basta.

Problemi

- Le maiuscole vengono prima delle minuscole.
- I simboli hanno 4 intervalli diversi evidenziati in giallo.

Soluzione trovata

- Richiamare gli ordinamenti per intervalli
- se non appartiene a quell'intervallo un elemento lo si ignora
- implementazione **ricorsiva**

Analizzando il codice troviamo:

- **recorsive_label** che esegue ricorsivamente la ricerca dei caratteri del sottoinsieme richiamato; utilizza il registro s2 per segnalare l'effettiva presenza di un elemento del sottoinsieme (quindi aggiornare il puntatore s10).
- **loop** dove si cerca l'elemento del sottoinsieme nella lista
- **exc_anyway** si occupa di scambiare l'elemento.
- **not_exc** non fa lo scambio, perché l'elemento analizzato non rientra nel sottoinsieme di ordinamento corrente (manda avanti il puntatore s11).
- **end_rec**: la ricorsione è finita.

Di seguito è riportato il codice del sort.

1 Gestione delle chiamate per intervalli

s5 e s4 contengono i valori che delimitano i vari intervalli.

```
sort:
    #simbolo<numeri<minuscole<maiuscole
    #s10 posizione fin dove sono arrivato a confermare l'ordinamento
    #s11 scorrimento generale degli elementi
    #s2 viene usato per il numero di scambi in questo contesto del programma
    #s5 e s4 tengono traccia degli intervalli di ordinamento

    lw s10,head
    lw s11,head
    addi t3,s11,0
    addi a6,s11,0

    beq t3,t5,end_rec #lista vuota
    lw s2,1(s10)
    beq t3,s2,end_rec ##1 solo elemento

###caratteri da space a '/'###
    li s5,32
    li s4,47
    addi sp,sp,-4
    sw ra,0(sp)
    jal recorsive_label
    lw ra, 0(sp)
    addi sp,sp,4

###caratteri da ':' a '@'###
    li s5,58
    li s4,64
    addi sp,sp,-4
    sw ra,0(sp)
    jal recorsive_label
    lw ra, 0(sp)
    addi sp,sp,4

###caratteri da 'parentesi quadrata' a '`'###
    li s5,91
    li s4,96
    addi sp,sp,-4
    sw ra,0(sp)
    jal recorsive_label
    lw ra, 0(sp)
    addi sp,sp,4

###caratteri da '{' a '}' ###
    li s5,123
    li s4,125
    addi sp,sp,-4
    sw ra,0(sp)
    jal recorsive_label
    lw ra, 0(sp)
    addi sp,sp,4

###Numeri###
    li s5,48
    li s4,57
    addi sp,sp,-4
    sw ra,0(sp)
    jal recorsive_label
    lw ra, 0(sp)
    addi sp,sp,4

###caratteri minuscoli###
    li s5,97
    li s4,122
    addi sp,sp,-4
    sw ra,0(sp)
    jal recorsive_label
    lw ra, 0(sp)
    addi sp,sp,4

###caratteri maiuscole###
    li s5,65
    li s4,90
    addi sp,sp,-4
    sw ra,0(sp)
    jal recorsive_label
    lw ra, 0(sp)
    addi sp,sp,4

    li s2,32
    addi a1,a1,1
    j end_rec
```

2 Algoritmo di ordinamento:

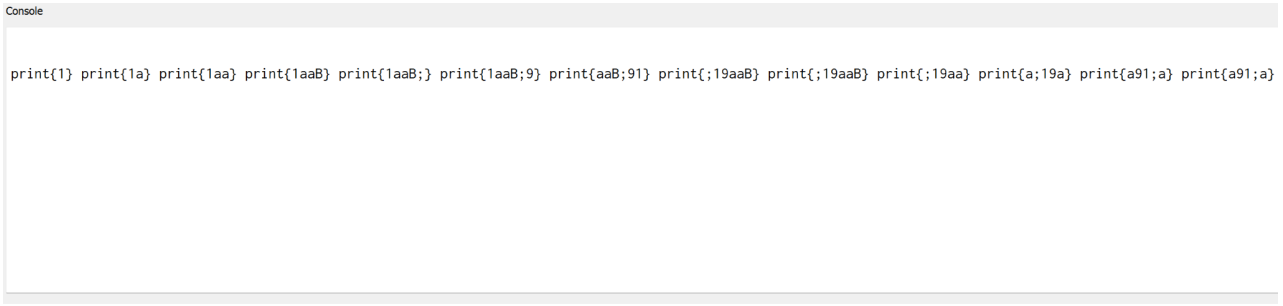
A ogni ricorsione viene richiamata questa porzione di codice in cui vengono effettuati scambi per ordine ASCII e se nel range aspettato nelle chiamate.

```
recorsive_label:

    lb s9,0(s10)
    lw s11,1(s10)
    li s2,0 # azzeramento numero scambi

    #start with a valid letter
    blt s9,s5,loop
    bgt s9,s4,loop
    li s2,1
    ##confronto qui
loop:
    beq s11,t3,end_loop
    lb s7,0(s11)
    blt s7,s5,not_exc
    bgt s7,s4,not_exc

    blt s9,s5,exc_anyway
    bgt s9,s4,exc_anyway
    #sia s9 che s7 sono nel range se si arriva a questa parte del codice
    bgt s7,s9,not_exc
exc_anyway:
    #si ricerca il migliore da mettere in quella posizione scorrendo tutto e poi
    #Si passa al successivo e si sposta s10 di una posizione
    addi a7,s9,0 #salva carattere scambio
    addi a6,s11,0 #indirizzo di scambio
    addi s9,s7,0 #scambio
    sb a7,0(a6)
    sb s9,0(s10)
    li s2,1

not_exc:
    lw s11,1(s11)
    j loop

end_loop:
    #se non ci sono stati scambi non mando avanti il puntatore dell' ordinamento-
    #-ne ciclo di nuovo per la stessa lettera
    beq s2,zero,end_rec
    lw s10,1(s10)
end_loop_no_upd:
    beq s10,t3, end_rec
    addi sp,sp,-4
    sw ra,0(sp)
    jal recorsive_label
    lw ra, 0(sp)
    addi sp,sp,4

end_rec:
    jr ra
```

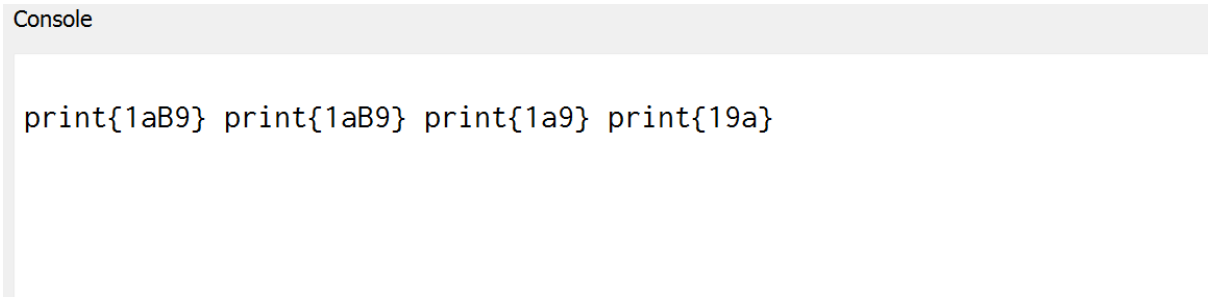
Test

*L'esecuzione di **alcuni** test si trova nel video allegato al progetto*.

test 1 (preso dalla versione 4 del progetto):
"ADD(1)~PRINT~ADD(a)~PRINT~ADD(a)~PRINT~ADD(B)~PRINT~ADD(;)~PRINT~ADD(9)
~PRINT~SSX~PRINT~SORT~PRINT~DEL(b)~PRINT~DEL(B)~PRINT~PRI~ SDX
~PRINT~ REV ~PRINT~ PRINT"



test 2 (preso dalla versione 4 del progetto):
"ADD(1) ~ SSX ~ ADD(a) ~ add(B) ~ ADD(B)~ ADD ~ ADD(9) ~PRINT ~ SORT(a) ~ PRINT
~ DEL(bb) ~DEL(B) ~PRINT~ REV ~SDX ~PRINT"



test 3
finalità del test: comandi nulli o non validi.
"ADD(K)~ADD(2)~AD~~~ADD(4)~ADD(d)~PRINT~REV~PRINT~PRI~A~REV~PRINT~
~ADD(8)~PRINT"

ISTRUZIONE	LISTA	OUTPUT	ESECUZIONE
ADD(K)	K		si

ADD(2)	K2		si
AD	K2		no
~	K2		no
ADD(4)	K24		si
ADD(d)	K24d		si
PRINT	K24d	print{K24D}	si
REV	d42K		si
PRINT	d42K	print{d42K}	si
PRI	d42K		no
A	d42K		no
REV	K24d		si
PRINT	K24d	print{K24d}	si
(spazio)	K24d		no
ADD(8)	K24d8		si
PRINT	K24d8	print{K24d8}	si

Console

```
print{K24d} print{d42K} print{K24d} print{K24d8}
```

test 4

finalità del test: il programma deve riconoscere i comandi anche nel caso si presentino con spazi prima e dopo il carattere di delimitazione tilde(~).

“ ADD(6) ~ADD(7) ~REV K~ADD(f)~PRINT ~SORT~PRINT”

ISTRUZIONE	**LISTA**	**OUTPUT**	**ESECUZIONE**
ADD(6)	6	-	si

ADD(7)	67	-	si
REV K	67	-	no
ADD(f)	67f	-	si
PRINT	67f	print{67f}	si
SORT	67f	-	si

Console

```
print{67f} print{67f}
```

test 5

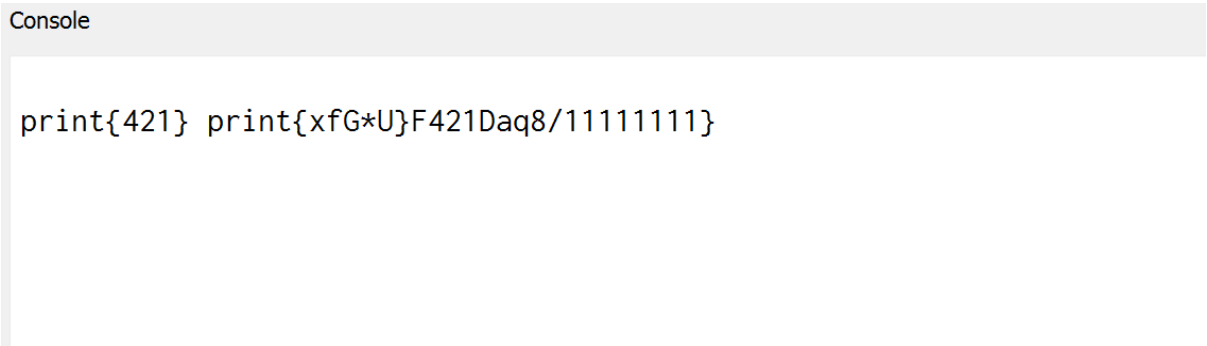
finalità del test: il programma non prende più di 30 comandi.
“ADD(1)~ADD(2)~ADD(4)~REV~PRINT~SORT~ADD(F)~ADD(})~ADD(U)~ADD(*)~ADD(G)~ADD(f)~ADD(x)~REV~ADD(D)~ADD(a)~ADD(q)~ADD(8)~ADD(/)~ADD(1)~ADD(1)~ADD(1)~ADD(1)~ADD(1)~ADD(1)~ADD(1)~ADD(1)~PRINT~ADD(1)~ADD(1)~PRINT~SORT~REV~PRINT”

dal comando numero 31 (colorato in rosso) notiamo che il programma non esegue più i successivi.

ISTRUZIONE	**LISTA**	**OUTPUT**	**ESECUZIONE**
ADD(1)	1	-	si
ADD(2)	12	-	si
ADD(4)	124	-	no
REV	421	-	si
PRINT	421	print{421}	si
SORT	124	-	si
ADD(F)	124F		si
ADD(})	124F}		si

ADD(U)	124F}U		si
ADD(*)	124F}U*		si
ADD(G)	124F}U*G		si
ADD(f)	124F}U*Gf		si
ADD(x)	124F}U*Gfx		si
REV	xfG*U}F421		si
ADD(D)	xfG*U}F421D		si
ADD(a)	xfG*U}F421Da		si
ADD(q)	xfG*U}F421Daq		si
ADD(8)	xfG*U}F421Daq8		si
ADD(/)	xfG*U}F421Daq8/		si
ADD(1)	xfG*U}F421Daq8/1		si
ADD(1)	xfG*U}F421Daq8/11		si
ADD(1)	xfG*U}F421Daq8/111		si
ADD(1)	xfG*U}F421Daq8/1111		si
ADD(1)	xfG*U}F421Daq8/11111		si
ADD(1)	xfG*U}F421Daq8/111111		si
ADD(1)	xfG*U}F421Daq8/1111111		si
ADD(1)	xfG*U}F421Daq8/11111111		si
ADD(1)	xfG*U}F421Daq8/111111111		si
PRINT	xfG*U}F421Daq8/111111111	print{xfG*U}F421Dq8/111111111}	si

ADD(1)	xfG*U}F421Daq8/ 1111111111		si
ADD(1)	xfG*U}F421Daq8/ 11111111111		si
PRINT	END		NO(massimo istruzioni raggiunto)
SORT			NO
REV			NO
PRINT			NO



Possiamo notare che il registro a1 contiene 30 comandi (il massimo eseguibile).

test 6

finalità del test: testare l'intero funzionamento del programma "ADD(a)~ADD(8)~ADD(a)~ADD(a)~ADD(a)~REV~DEL(8)~PRINT~DEL(a)~ADD(1)~SSX~PRINT~ADD(0)~PRINT~SORT~PRINT~ADD(A)~ADD(F)~~ADD(*)~ADD(F)~ADD([)~SORT~SDX~PRINT~DEL(0)~A ~ PRI~PRINT~ADD(<)~ADD(})~SORT~PRINT~SORT~PRINT"

ISTRUZIONE	**LISTA**	**OUTPUT**	**ESECUZIONE**
ADD(a)	a		si
ADD(8)	a8		si

ADD(a)	a8a		si
ADD(a)	a8aa		si
REV	aa8a		si
DEL(8)	aaa		si
PRINT	aaa	print{aaa}	si
DEL(a)	-		si
ADD(1)	1		si
SSX	1		si
PRINT	1	print{1}	si
ADD(0)	10		si
PRINT	10	print{10}	si
SORT	01		si
PRINT	01	print{01}	si
ADD(A)	01A		si
ADD(F)	01AF		si
~			no
ADD(*)	01AF*		si
ADD(F)	01AF*F		si
ADD([)	01AF*F[		si
SORT	*[01AFF		si
SDX	F*[01AF		si
PRINT	F*[01AF	print{F*[01AF}	si
DEL(0)	F*[1AF		si
A	F*[1AFA		no
PRI	F*[1AFA		si

PRINT	F*[1AF	print{F*[1AF}	si
ADD(<)	F*[1AF<		si
ADD(})	F*[1AF<}		si
SORT	F*[1AF<}		si
PRINT	*<[}1AFF	print{*<[}1AFF}	si
SORT	END		NO(massimo istruzioni raggiunto)
PRINT			NO

Console

```
print{aaaa} print{1} print{10} print{01} print{F*[01AF} print{F*[1AF} print{*<[}1AFF}
```