UNIVERSITÀ DEGLI STUDI DI L'AQUILA
FACOLTÀ DI INFORMATICA

PROGETTO NO. 5 – LABORATORIO DI ARCHITETTURA DEGLI ELABORATORI

HEAPSORT

ANTONIO DI FRANCESCO – 155418
PAOLO BOZZELLI – 158688
DANIELE GABRIELLI - 154849

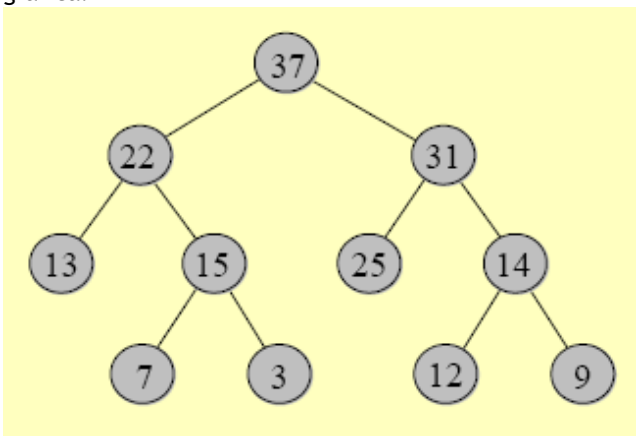
1 *Heapsort* – cenni preliminari

Tra gli algoritmi di ordinamento, uno dei più utilizzati è l'*heapsort*. Questo algoritmo procede all'ordinamento mediante lo stesso approccio incrementale adottato dal *selection sort*, differendo da quest'ultimo nella struttura dati utilizzata: l'*heapsort* utilizza l'*heap*, una struttura dati, associata ad un insieme *S*, più efficace per l'estrazione del massimo elemento da un array non ordinato ad ogni iterazione.

L'*heap* è un albero binario radicato con alcune proprietà fondamentali:

- è **completo** fino al **penultimo** livello;
- gli elementi di *S* sono memorizzati nei **nodi** dell'albero;
- per ogni nodo *v*, la chiave del padre di *v* è **maggiore o uguale** della chiave di *v*.

Per dare un'idea migliore della rappresentazione di un *heap*, di seguito ne riportiamo la rappresentazione grafica:



L'albero rappresenta una sequenza di numeri d'esempio. Non potendo lavorare su una struttura simile nell'architettura MIPS che andremo ad utilizzare, possiamo ricorrere all'utilizzo di una struttura dati a noi più comune: gli **array**.

Infatti la struttura rappresentata in figura fa riferimento ad un **array posizionale**, ovvero:

/ 37 22 31 13 15 25 14 **/ /** 7 3 **/ /** 12 9

La rappresentazione dei nodi dell'albero è calcolata in base alle posizioni dei nodi stessi all'interno dell'array posizionale.

In questa rappresentazione, l'heap è mostrato con una struttura semplice ed il suo vettore posizionale relativo permette anche la presenza di elementi vuoti (**/**). L'*heapsort* è implementato mediante l'utilizzo di un **heap con struttura rafforzata**, nella quale le foglie dell'ultimo livello sono compattate a sinistra, rendendo così il vettore posizionale di dimensione *n* (dove *n* è il numero di nodi presenti nell'heap).

Riassumiamo ora le **proprietà salienti** dell'heap:

1. Il **massimo** è contenuto nella **radice**;
2. L'albero ha altezza **O(*log*(*n*))**;
3. Gli heap con struttura rafforzata possono essere rappresentati con array di ***dimensione n***.

L'algoritmo *heapsort* ordina in loco gli elementi della struttura dati compiendo due passi fondamentali:

- costruzione dell'heap mediante la procedura ***heapify***;
- estrazione ricorsiva del massimo per *n-1* volte, memorizzandolo nella posizione dell'array appena liberata.

La procedura ***heapify*** è un algoritmo ricorsivo basato sul *divide et impera*: dapprima controlla se l'heap è vuoto, in caso contrario ripete se stessa sul sottoalbero destro e sul sottoalbero sinistro; infine, mediante la procedura ***fixHeap***, controlla che tutti i nodi siano posizionati secondo la proprietà dell'ordinamento dell'heap.

L'estrazione del massimo copia nella radice la chiave contenuta nella foglia più a destra dell'ultimo livello, rimuove la foglia e ripristina la proprietà dell'ordinamento dell'heap, richiamando la ***fixHeap*** sulla radice.

2 *Heapsort* – traccia progetto

Dato un vettore non ordinato residente in RAM, ordinarlo in modo crescente in loco utilizzando l'algoritmo di Heapsort, senza usare altra memoria RAM oltre quella dove sono memorizzati i dati di partenza. Dopo averlo ordinato, si permetta di modificare l'heap e si implementi una routine per ricostruire la struttura ad heap.

La dimensione *n* del vettore va passata come parametro da riga di comando; qualora non sia stata passata, il programma deve richiedere di immetterla da tastiera (console). Il valore che si vuole modificare (*k*) e la sua posizione (*j*) devono essere passati tramite riga di comando o, alternativamente, da console.

Stampare a video l'Heapsort iniziale e quello ricostruito. Includere nel programma un vettore d'esempio di almeno 10 caratteri.

3 *Heapsort* – organizzazione per passi

Il nostro progetto prevede l'implementazione dell'algoritmo di ordinamento *heapsort* mediante la copertura di **sei** passi principali:

1. ***Inserimenti iniziali***. In questa sezione, inizializzeremo le variabili stringa che permetteranno la visualizzazione di messaggi durante l'interazione con l'utente. Inoltre, questa sezione sarà responsabile del controllo di eventuali inserimenti da riga di comando o, in alternativa, degli inserimenti da console.
2. ***Build-heap***. Nella seconda fase, il programma analizzerà la stringa inserita, sia essa quella d'esempio o quella immessa dall'utente. Il *Build-heap* restituirà il vettore di caratteri ordinato secondo le proprietà dell'ordinamento ad heap in modo crescente. Durante l'ordinamento, vengono anche mantenute tutte le proprietà di una struttura *heap*.
3. ***Stampa della stringa con proprietà dell'heap***. Questa sezione stampa a video la stringa parzialmente ordinata. Per scelta di progetto, abbiamo optato per la visualizzazione dell'evoluzione dell'ordinamento della stringa, per dar modo all'utente di aver più chiaro il procedimento di ordinamento della stringa stessa.
4. ***Inserimento nuovi valori***. L'utente ha la possibilità di inserire nuovi valori all'interno della stringa da ordinare, sia essa la stringa di default, sia quella inserita dall'utente stesso. Questo passaggio può avvenire in 2 modi differenti: da riga di comando, inserendo prima il nuovo valore (carattere) e, separata da uno spazio, la posizione nella quale si vuole introdurre il nuovo valore; da console, seguendo le istruzioni mostrate a video.
5. ***Gestione parametri da riga di comando***. In questa porzione di programma, viene gestito l'inserimento di parametri da riga di comando. Ne seguono tutti i relativi controlli sulla correttezza dei parametri inseriti.
6. ***Gestione errori e stampa finale***. Infine, l'ultima sezione riguarda la gestione degli errori verificatisi nelle cinque sezioni precedenti: si mostra a video un messaggio d'errore e si restituisce il controllo al programma, ove esso è stato interrotto. Nel caso della stampa finale, il programma mostra a video la stringa ordinata in modo crescente e termina la propria esecuzione.

4 *Heapsort* – implementazione

4.1 Inserimenti iniziali

Osserviamo la struttura iniziale del programma. Nella sezione d'intestazione sono presenti tutte le dichiarazioni di stringa che verranno utilizzate durante l'esecuzione del programma.

```
STRINGA_ES: .asciiz "pso49dnie65hfm\n"

SCELTA: .asciiz "Vuoi inserire una nuova stringa? (s/n) "

SCELTA2: .asciiz "Vuoi cambiare un valore alla stringa? (s/n)"

SCELTA3: .asciiz "Inserisci il nuovo valore..."

SCELTA4: .asciiz "Inserisci la posizione del vettore..."

SCELTA5: .asciiz "La stringa finale è la seguente..."

STRINGA_LUNG: .asciiz "Lunghezza della stringa da inserire: "

STRINGA_INS: .asciiz "Inserisci la stringa: "

ERROR1: .asciiz "Il carattere inserito non è correto! Riprova...\n"

ERROR2: .asciiz "Il numero dei caratteri non corrisponde alla lunghezza del
vettore...\n"

ERROR3: .asciiz "Il valore inserito supera la posizione della stringa...Riprova\n"

FINE: .asciiz "LA PROVA è FINITA!\nLa prova sull'esempio è la seguente..."
```

```
SCELTA1: .asciiz " "
```

Tra queste compare la seguente stringa:

```
STRINGA_ES: .asciiz "pso49dnie65hfm\n"
```

`STRINGA_ES` è il vettore di caratteri d'esempio – come richiesto dalla traccia – sul quale verrà effettuato l'ordinamento di default.

Una volta caricate le stringhe, possiamo procedere con l'esecuzione del programma. Dal `main` osserviamo che c'è subito un primo controllo, il quale è incaricato di indirizzare il programma alla label `RIGACOMANDO` nel caso sia stato inserito qualche parametro da riga di comando; in caso contrario, il programma procede con l'istruzione successiva.

```
bnez $a0, RIGACOMANDO
```

L'istruzione `bnez` (**b**ranch if **n**ot **e**qual to **z**ero) confronta `$a0` con il valore 0: nel caso in cui `$a0` abbia valore diverso da zero, il programma viene indirizzato alla label indicata (in questo caso, `RIGACOMANDO`).

Esaminiamo ora il caso in cui non viene inserito alcun parametro da riga di comando, ovvero quando l'istruzione descritta precedentemente abbia dato esito negativo (`$a0` = 0).

Non avendo inserito alcun parametro, si richiede all'utente di inserire o meno una nuova stringa, stampando a video la stringa `SCELTA`.

```
la $a0, SCELTA
li $v0, 4
syscall
```

A questo punto, l'utente dovrà digitare `s` (per inserire una nuova stringa) o `n` (per ordinare l'array di default). La risposta dell'utente viene intercettata grazie al codice che segue:

```
la $a0, SCELTA1
li $v0, 8
syscall
```

Il programma prevede quindi un controllo sulla risposta dell'utente, il quale può aver *sbadatamente* inserito più di un carattere.

```
lb $t0, 1($a0)                # Caricamento posizione successiva a lettera inserita
bne $t0, 10, ERROR_1
li $t0, 0
```

Altro controllo viene effettuato nel caso in cui il carattere sia uno solo, ma diverso da `s` o `n`.

```
lb $t0, 0($a0)                # Caricamento lettera inserita
beq $t0, 110, LOADING_ES
bne $t0, 115, ERROR_1
```

Dopo aver caricato il carattere in un registro temporaneo (`$t0`), si esamina il valore ASCII del carattere stesso, confrontandolo prima con l'ASCII code del carattere `n`, poi con quello del carattere `s`.

Nel caso di inserimento del carattere `n`, il programma viene indirizzato alla label `LOADING_ES` (incaricata di stampare l'esempio iniziale e terminare il programma). La seconda istruzione di confronto controllo se il carattere inserito sia diverso da `s`: in tal caso, il programma è indirizzato alla label `ERROR_1`, incaricato di stampare a video un messaggio d'errore e di reindirizzare il programma alla scelta dell'inserimento di una nuova stringa.

Ora il programma presenta una nuova diramazione: esamineremo quindi il caso in cui l'utente scelga di inserire una nuova stringa da riordinare.

Avendo scelto di inserire una nuova stringa, viene visualizzato il messaggio contenuto nella stringa

STRING_LUNG.

```
la $a0, STRINGA_LUNG
li $v0, 4
syscall

li $v0, 5                    # Inserimento lunghezza stringa
syscall
```

Dopo aver visualizzato la stringa, l'utente può inserire la *lunghezza* della nuova stringa da inserire. Anche in questo caso, c'è un controllo sull'input dell'utente: se viene inserito un valore *minore o uguale a zero*, il programma visualizza un messaggio d'errore. Invece, se il valore inserito è corretto, viene caricato in `$a0`.

```
ble $v0, $zero, ERROR_1
move $a0, $v0
j INSERIMENTO
```

Alla label `INSERIMENTO` troviamo il codice che segue:

```
INSERIMENTO:
move $s0, $a0
move $t9, $s0
li $a0, 0

la $a0, STRINGA_INS
li $v0, 4
syscall

li $v0, 8
syscall
move $s1, $a0
```

Viene salvato il valore inserito dell'utente prima in `$s0`, poi in `$t9` (per poterlo riutilizzare in seguito). Si visualizza quindi il messaggio che comunica di inserire la nuova stringa: una volta inserita, viene salvata in `$s1`.

Inserita la nuova stringa, bisogna controllare se questa corrisponde alla lunghezza precedentemente inserita dall'utente. Contiamo quindi i caratteri inseriti:

```
CONTATORE:
lb $t0, 0($a0)
beq $t0, $zero, CONTROLLO
addi $t1, $t1, 1
addi $a0, $a0, 1
j CONTATORE
```

Se `$t0` corrisponde al carattere `NULL` (`0` in codice ASCII, quindi `$zero`), il programma passa a controllare se i caratteri immessi sono tanti quanti l'utente aveva precedentemente dichiarato, altrimenti viene incrementato il contatore (`$t1`) dei caratteri presenti, per poi spostarsi sul carattere successivo.

Una volta terminato il conteggio dei caratteri, controlliamo che questi siano in quantità uguale alla lunghezza inserita dall'utente:

```
CONTROLLO:
addi $t1, $t1, -1
bne $t1, $s0, ERROR_2
j INIZIO
```

In `$s0` è memorizzata la lunghezza inserita dall'utente, in `$t1` è memorizzata la lunghezza effettiva della stringa. Se queste non coincidono, il programma segnala l'errore mediante la visualizzazione del messaggio `ERROR2`.

```
ERROR_2:
```

```
la $a0, ERROR2
li $v0, 4
syscall
```

In caso contrario, invece, il programma è pronto a ordinare la nuova stringa mediante l'*heapsort*: dalla label `INIZIO`, infatti, comincia la procedura di ordinamento:

```
INIZIO:
add $s2, $s1, $s0
```

In `$s1` è memorizzato l'indirizzo della stringa, in `$s0` è memorizzata la lunghezza della stessa. Sommando i due valori, otteniamo l'indirizzo del carattere finale della stringa e lo memorizziamo in `$s2`. Inizia quindi l'iterazione, che conterrà la chiamata alla procedura `BUILD_HEAP`, responsabile della costruzione dell'heap e del relativo ordinamento.

Come primo comando, troviamo un decremento: viene calcolata l'effettiva posizione finale della stringa.

```
addi $s2, $s2, -1
```

Il ciclo comincia con un controllo sulla posizione correntemente esaminata: se corrisponde alla posizione finale, vuol dire che abbiamo analizzato l'intera stringa. In caso contrario, si procede con l'ordinamento. Memorizziamo in `$t0` il valore `2`, poiché utilizzeremo questo registro per dividere l'*heap-size* (la grandezza dell'heap).

```
li $t0, 2
```

Ora dobbiamo dividere l'heap-size. Prima però controlliamo se l'array è già stato interamente esaminato: se è così, saltiamo all'etichetta `ESCI2` (paragrafo 4.3), altrimenti procediamo con la divisione dell'heap-size. Per la divisione dell'heap-size ci serviamo del registro `$t0`, che sarà divisore per il registro `$s0`, dove è memorizzato l'heap-size. Salveremo quindi il risultato della divisione (quoziente) in `$t1`, mediante l'istruzione `mflo` (**m**ove **f**rom **lo**w).

```
CICLO:
beq $s2, $s1, ESCI2

[...]

div $s0, $t0
mflo $t1
```

Ai fini dell'iterazione, controlliamo se il risultato della precedente divisione sia uguale a zero, il che indicherebbe la fine dell'analisi dell'intero heap, ovvero quando il controllo arriva alla radice dell'heap stesso, saltando alla label `ESCI1` (paragrafo 4.3).

```
RITORNA:
beq $t1, $zero, ESCI1
```

Superato questo controllo, carichiamo in `$t3` la posizione appena precedente a quella rappresentata da `$t1`, ovvero la `(n/2)-1` posizione (dove `n` è l'heap-size), detto *limite inferiore*, e la passiamo alla procedura `BUILD_HEAP`.

```
add $t3, $t1, $s1
addi $t3, $t3, -1
move $a0, $t3
jal BUILD_HEAP
```

4.2 Procedura `BUILD_HEAP`

È doveroso dedicare un paragrafo a parte per questa procedura, data la complessità della sua implementazione e del suo funzionamento. Innanzitutto, esaminiamo i parametri che riceve in ingresso.

Dall'ultima sezione di codice descritta si evince che la `BUILD_HEAP` riceve solo un parametro, memorizzato in `$a0`: questo parametro è la posizione del nodo corrente, ovvero l'indirizzo; ne carichiamo quindi il valore corrispondente, salvandolo in `$t4`.

```
lb $t4, 0($a0)
```

Si calcola quindi la posizione dei figli destro e sinistro rispetto alla posizione corrente (memorizzata nel registro `$t1`). Quindi, viene calcolato l'indirizzo del figlio destro (memorizzato in `$t5`).

```
mul $t5, $t1, 2
addi $t5, $t5, 1
add $t5, $t5, $s1
```

Dopo il decremento di `$t5` (che permetterà di scorrere l'array in fase di iterazione), viene controllata la presenza del figlio destro, mediante l'istruzione `bgt` (**b**ranch if **g**reater **t**han).

```
addi $t5, $t5, -1
bgt $t5, $s2, LASCIA
lb $t6, 0($t5)
```

Nel caso in cui esiste un figlio destro, il valore di quest'ultimo viene memorizzato in `$t6`, altrimenti il programma passa a verificare la presenza di un figlio sinistro (dalla label `LASCIA`).

```
LASCIA:
mul $t7, $t1, 2
add $t7, $t7, $s1
addi $t7, $t7, -1
lb $t8, 0($t7)
```

Verificata la presenza del figlio destro, si passa al figlio sinistro: viene dapprima calcolata la sua posizione, poi il suo indirizzo, il quale viene memorizzato in `$t7`. Il valore del figlio sinistro viene quindi caricato in `$t8`.

```
slt $t2, $t4, $t8
bne $t2, $zero, SCAMBIA
slt $t2, $t4, $t6
bne $t2, $zero, SCAMBIA
j CONTROLLA_PADRE
```

Per mantenere la proprietà dell'ordinamento dell'heap, bisogna controllare che il padre sia maggiore dei figli destro e sinistro: in entrambi i casi, se il padre risulta essere minore di uno dei due figli, viene effettuato uno scambio tra i due nodi che non rispettano la proprietà dell'orientamento.

Questo tipo di controllo viene effettuato prima sul figlio sinistro non per scelta progettuale, ma per rispettare la proprietà dell'ordinamento dell'heap: infatti, il figlio sinistro deve essere *minore* sia del padre, che del figlio destro. Un controllo invertito altererebbe la proprietà dell'ordinamento dell'heap.

La routine di scambio è così codificata:

```
SCAMBIA:
slt $t2, $t8, $t6
bne $t2, $zero, FIGLIOSINISTRO
sb $t8, 0($a0)
sb $t4, 0($t7)
j CONTROLLA_PADRE
```

Da come si può osservare dal codice, nel caso si presenti un figlio destro minore del figlio sinistro, il programma viene indirizzato alla label `FIGLIOSINISTRO`. In caso contrario, si procede allo scambio del figlio destro con il padre.

Alla label `FIGLIOSINISTRO` viene effettuato l'eventuale scambio tra padre e figlio sinistro.

```
FIGLIOSINISTRO:
sb $t6, 0($a0)
```

```
sb $t4, 0($t5)
```

Al termine delle routine `LASCIA` e `SCAMBIA` è presente un jump (`j`) alla routine `CONTROLLA_PADRE` (`FIGLIOSINISTRO` è la routine precedente a `CONTROLLA_PADRE`, perciò non ha bisogno di jump).

```
CONTROLLA_PADRE:
li $t3, 0

lb $t3, 0($a0)

li $t4, 0
li $t5, 0

beq $t1, 1, ESCI

div $t1, $t0
mflo $t4

add $t4, $t4, $s1
addi $t4, $t4, -1
lb $t5, 0($t4)

slt $t2, $t5, $t3
bne $t2, $zero, SCAMBIA_1

j ESCI
```

Viene dapprima salvato il valore alla posizione attuale nel registro temporaneo `$t3`, poi si controlla che la posizione non sia la radice, altrimenti il programma viene indirizzato alla label `ESCI` (che vedremo in seguito).

Verificato che la posizione corrente non è la radice, si procede alla successiva divisione della corrente porzione di array. Calcoliamo l'indirizzo del padre (memorizzandolo in `$t4`), dal quale preleviamo il valore, memorizzando quest'ultimo in `$t5`.

Successivamente, controlliamo che il valore del padre sia maggiore del valore della posizione corrente: in caso contrario, viene effettuato uno scambio (`SCAMBIA_1`).

```
SCAMBIA_1:

sb $t3, 0($t4)
sb $t5, 0($a0)

j BUILD_HEAP
```

`SCAMBIA_1` è responsabile dell'eventuale scambio tra il valore della posizione attuale con il valore del padre. In seguito, viene riassegnato il controllo alla `BUILD_HEAP`.

```
ESCI:
addi $t1, $t1, -1

li $t6, 0
li $t8, 0

j RITORNA
```

`ESCI` decrementa di 1 la posizione sulla quale effettuare i controlli per l'ordinamento e reindirizza il programma alla label `RITORNA`.

4.3 Stampa della stringa con proprietà dell'heap

Nel paragrafo 4.1 abbiamo tralasciato la descrizione delle label `ESCI1` ed `ESCI2`: queste sono responsabili della stampa della stringa ordinata secondo i criteri dell'heapsort. `ESCI1` stampa la stringa durante l'evoluzione dell'ordinamento, `ESCI2` stampa la stringa ordinata in modo **crescente**.

```
ESCI1:
```

```
move $a0, $s1
li $v0, 4
syscall

li $t3, 0
li $t4, 0
li $t5, 0

lb $t3, 0($s2)
lb $t4, 0($s1)

sb $t4, 0($s2)
sb $t3, 0($s1)

addi $s2, $s2, -1
addi $s0, $s0, -1

j CICLO

ESCI2:

move $a0, $s1
li $v0, 4
syscall
```

4.4 Inserimento nuovi valori

Una volta ordinata la stringa di default o quella inserita dall'utente, quest'ultimo può decidere se inserire un valore nuovo in una qualsiasi delle posizioni del vettore di caratteri.

```
RITENTA:
la $a0, SCELTA2
li $v0, 4
syscall

la $a0, SCELTA1
li $v0, 8
syscall
```

Con questa porzione di codice, viene visualizzato a video il messaggio "`Vuoi cambiare un valore alla stringa? (s/n)`", la quale risposta viene catturata e salvata in `SCELTA1`.

Effettuo quindi i soliti controlli di correttezza sull'input da tastiera.

```
lb $t0, 1($a0)
bne $t0, 10, ERROR_3
li $t0, 0

lb $t0, 0($a0)
```

Nel caso sia stato inserito più di un carattere, il programma è indirizzato alla label `ERROR_3`. In seguito, viene controllata la volontà dell'utente: se viene premuto `n` (codice ASCII: 110), il programma viene indirizzato alla label `PRINT_STR` (che termina anche l'esecuzione); se viene premuto un carattere diverso da `s`, viene mostrato un messaggio d'errore (label `ERROR_3`); se viene inserito `s` (codice ASCII: 115) il programma procede con l'esecuzione delle istruzioni successive, quindi dell'inserimento del valore da cambiare.

```
beq $t0, 110, PRINT_STR
bne $t0, 115, ERROR_3
```

Supponendo che l'utente voglia inserire un nuovo valore, vedremo quindi visualizzato a video un messaggio che chiederà all'utente di immettere il valore.

```
la $a0, SCELTA3
li $v0, 4
syscall
```

```
la $a0, SCELTA1

li $v0, 8
syscall
```

Una volta inserito il nuovo valore, vengono effettuati i controlli relativi alla correttezza dell'input: prima si controlla se si è inserito più di un carattere, poi che sia presente almeno un carattere. In entrambi i casi, se il controllo dà esito negativo, il programma viene indirizzato alla label `ERROR_3`.

```
lb $t0, 1($a0)
bne $t0, 10, ERROR_3

lb $t0, 0($a0)

beq $t0, 10, ERROR_3
```

Se il valore è stato inserito correttamente, si procede all'inserimento della posizione nella quale lo si vuole inserire, salvandone il valore in `$t1`.

```
la $a0, SCELTA4
li $v0, 4
syscall

la $a0, SCELTA1

li $v0, 5
syscall

move $t1, $v0
```

Bisogna quindi controllare che il valore immesso sia compreso tra 1 e *n*, dove *n* è la dimensione della stringa. Nel caso sia *minore o uguale* a 0, verrà mostrato il messaggio d'errore alla label `ERROR_3`; se il valore è maggiore della dimensione della stringa, verrà visualizzato il messaggio d'errore alla label `ERROR_4`.

```
ble $t1, $zero, ERROR_3
bgt $t1, $t9, ERROR_4
```

Superati i controlli, bisogna procedere al caricamento del valore intero inserito e al calcolo del relativo indirizzo della posizione inserita da tastiera.

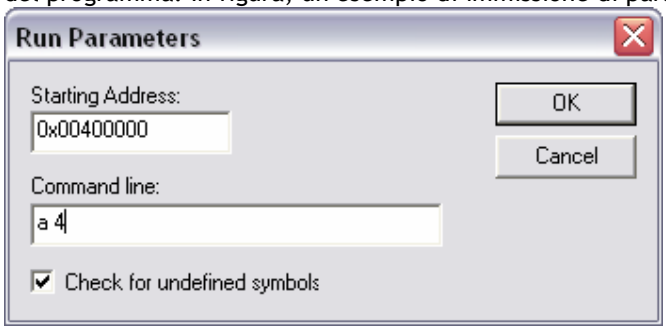
```
li $t2, 0
add $t1, $t1, $s1
addi $t1, $t1, -1

sb $t0, 0($t1)

move $a0, $s1

li $v0, 4
syscall

li $t1, 0
move $s0, $t9

j CONTATORE
```

Salvata in `$s0` la lunghezza della stringa e caricata quest'ultima con il nuovo valore inserito nella posizione indicata dall'utente, possiamo procedere all'ordinamento, reindirizzando il programma alla label `CONTATORE`.

4.5 Gestione parametri da riga di comando

Come abbiamo accennato nel paragrafo 4.1, è possibile inserire dei parametri direttamente da riga di

comando. Ricordiamo che per *riga di comando* (o *command line*) intendiamo la fase di precaricamento del programma. In figura, un esempio di immissione di parametri da riga di comando.



Ogni parametro viene inserito separato da uno spazio.

In SPIM, quando vengono inseriti dei valori da riga di comando, questi vengono memorizzati nei registri.

Nel dettaglio, il numero di parametri inseriti è memorizzato nel registro **`$a0`**, mentre in **`$a1`** vengono memorizzati gli indirizzi dei valori dei parametri inseriti.

I valori reali sono memorizzati all'interno dello stack, ognuno di essi separati da un carattere nullo (00 in esadecimale).

Nel nostro progetto, si presentano due casi nei quali bisogna gestire l'immissione di parametri da riga di comando: all'inserimento della lunghezza della stringa da esaminare e all'inserimento di un nuovo valore con relativa posizione.

Il primo controllo che l'applicazione effettua è proprio sul registro `$a0`, per verificare l'eventuale presenza di parametri passati da riga di comando.

```
bnez $a0, RIGACOMANDO
```

Nel caso in cui sia presente anche un solo parametro (quindi `$a0` è diverso da *0*), il programma è indirizzato alla label `RIGACOMANDO`.

```
RIGACOMANDO:
beq $a0, 2, RIGACOMANDO2
bgt $a0, 1, LOADING_ES
lw $t0, 0($a1)
```

Nel caso in cui vengano inseriti due parametri, vuol dire che l'utente ha intenzione di inserire un nuovo valore in una determinata posizione all'interno della stringa già presente in memoria, perciò il programma va ad eseguire le istruzioni alla label `RIGACOMANDO2` (che esamineremo tra poco); se ha inserito più di un valore, viene caricato l'esempio presente in memoria, mediante le istruzioni alla label `LOADING_ES`.

Supponendo che sia verificata la seconda condizione, andiamo ad esaminare le istruzioni alla label `LOADING_ES`.

```
LOADING_ES:
li $a0, 0
li $s0, 0

la $a0, FINE
li $v0, 4
syscall

la $a0, STRINGA_ES
li $v0, 4
syscall

li $s0, 14
move $s1, $a0
move $t9, $s0
```

Viene caricata in `$t9` la lunghezza della stringa d'esempio ed in `$s1` l'indirizzo della posizione iniziale di tale stringa. In seguito, viene effettuato l'ordinamento su `STRINGA_ES`, ripetendo l'iterazione descritta nel paragrafo 4.2.

Supponiamo invece che l'utente abbia inserito due parametri dalla command line: si deve procedere all'esecuzione delle istruzioni alla label `RIGACOMANDO2`.

```
RIGACOMANDO2:
li $a0, 0
```

```
li $s0, 0
li $t3, 0
li $t4, 0
li $t5, 1

la $a0, STRINGA_ES
li $v0, 4
syscall

li $s0, 14
move $s1, $a0
move $t9, $s0

lw $t0, 0($a1)

lb $t1, 0($t0)

addi $t0, $t0, -2

lb $t2, 0($t0)
```

In questa prima parte, vengono caricati i due parametri inseriti e salvati in `$t1` e `$t2`, ove si trovano rispettivamente il nuovo valore da inserire nella stringa e la relativa posizione nella quale si desidera memorizzarlo. Nel registro `$t5` è memorizzato il fattore di moltiplicazione decimale, che ci occorrerà per calcolare la posizione che si intende modificare: nel caso in cui la stringa presenti più di 10 caratteri, sarà necessario moltiplicare per 10 il secondo valore immesso da riga di comando.

Ad esempio, se si inserisce "`a 12`", il valore `12` verrà memorizzato come `1` e `2` (in ASCII esadecimale): sarà quindi necessario dapprima calcolare il valore decimale di `1` e `2`, per poi moltiplicare per 10 il valore `1`.

Prima di tutto, però, bisogna controllare i parametri immessi da riga di comando.

```
MAX_10:
slt $t3, $t2, 48
bne $t3, $zero, ERROR_1

slt $t3, $t2, 58
beq $t3, $zero, ERROR_1
```

Dopo esserci assicurati che i valori inseriti siano effettivamente dei numeri (in caso contrario viene visualizzato un messaggio d'errore), possiamo procedere all'inserimento del valore passato da command line.

```
addi $t0, $t0, -1
lb $t2, 0($t0)

bne $t2, $zero, MAX_10

bgt $t4, $s0, ERROR_5

add $t4, $t4, $s1
addi $t4, $t4, -1

sb $t1, 0($t4)

j INIZIO
```

Com'è possibile osservare dal codice, vengono effettuati due ulteriori controlli: se il numero è maggiore di 10, il programma è reindirizzato alla label `MAX_10`, per poter calcolare il valore corretto della posizione che si intende modificare; quindi, viene verificato che il parametro immesso sia minore della lunghezza della stringa: in caso contrario, viene visualizzato un messaggio d'errore.

Una volta superati positivamente tutti i controlli, si procede alla sostituzione del valore e si indirizza il programma alla label `INIZIO`, cosicché possa iniziare la procedura di ordinamento *heapsort*.

4.6 Gestione errori e stampa finale

Riepiloghiamo ora la stampa a video degli errori finora non esaminati.

```
ERROR_1:
la $a0, ERROR1
li $v0, 4
syscall

j DECISIONE
```

Stampa la stringa `ERROR1` (“`Il carattere inserito non è correto! Riprova...\n`”) e indirizza al programma alla label `DECISIONE`.

```
ERROR_3:
la $a0, ERROR1
li $v0, 4
syscall

j RITENTA
```

Stampa la stringa `ERROR1` (“`Il carattere inserito non è correto! Riprova...\n`”) e indirizza al programma alla label `RITENTA`.

```
ERROR_4:
la $a0, ERROR3
li $v0, 4
syscall

j RITENTA
```

Stampa la stringa `ERROR3` (“`Il valore inserito supera la posizione della stringa...Riprova\n`”) e indirizza al programma alla label `RITENTA`.

```
ERROR_5:
la $a0, ERROR3
li $v0, 4
syscall

j DECISIONE
```

Stampa la stringa `ERROR3` (“`Il valore inserito supera la posizione della stringa...Riprova\n`”) e indirizza al programma alla label `DECISIONE`.

Infine, la porzione di codice che stampa la stringa finale ordinata in modo crescente.

```
PRINT_STR:
la $a0, SCELTA5
li $v0, 4
syscall

move $a0, $s1
li $v0, 4
syscall
```

5 Heapsort – riscontro problemi

Durante lo svolgimento del progetto non sono stati riscontrati particolari problemi di natura implementativa, a parte una contenuta difficoltà nella codifica della procedura di build-heap, la quale ha richiesto uno sforzo maggiore, data la complessità dell’algoritmo che la descrive.

In fase di testing, abbiamo analizzato tutti gli errori che possono manifestarsi durante l’inserimento (da console o da riga di comando) da parte degli utenti.

Volendo scendere nel dettaglio, abbiamo riscontrato l’unico problema in fase di input da console: nel caso in cui l’utente voglia scrivere una nuova stringa da ordinare, l’eventuale correzione della stringa stessa è ostacolata da un malfunzionamento non dovuto al codice da noi implementato. È nostra opinione che il malfunzionamento di cui sopra sia dovuto al comportamento dell’ambiente nel quale l’applicazione è eseguita.

Sottolineato questo particolare, possiamo concludere confermando di non aver trovato alcun ostacolo

durante lo svolgimento di questo progetto.

6 *Heapsort* – bibliografia

1. *Algoritmi e strutture dati* - (Demetrescu, Finocchi, Italiano – McGraw-Hill - 2004)
2. *Slides di algoritmi e strutture dati* – (Demetrescu, Finocchi, Italiano – McGraw-Hill - 2004)
3. *Appunti di algoritmi e strutture dati* - (Proietti - 2004)
4. *Struttura e progetto dei calcolatori, l'interfaccia hardware e software* – (Patterson, Hennessy – Zanichelli – 1995)
5. *Slides di laboratorio di architettura degli elaboratori* - (Muccini – 2005)