Alma Mater Studiorum - Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

PROGRAMMAZIONE FUNZIONALE IN SPAZIO LOGARITMICO: FUNZIONI DI ORDINAMENTO

Tesi di Laurea in Sicurezza e Crittografia

Relatore:
Ill.mo Dott. Ugo Dal Lago

Presentata da:
Federico Foschini

II Sessione
Anno Accademico 2009/2010

Introduzione

Uno degli aspetti fondamentali della teoria della programmazione è quello di creare nuovi linguaggi che aiutino il programmatore a sviluppare algoritmi efficienti in maniera semplice. Il programmatore dovrebbe concentrarsi solamente sulla risoluzione del problema, senza preoccuparsi di come poi le sue idee vengano implementate e gestite dal livello sottostante.

In questa dissertazione sarà presentato IntML, un linguaggio che fornisce all'utilizzatore un meccanismo intuitivo per scrivere algoritmi in spazio sublineare, che permette, cioè, di utilizzare meno memoria di quella necessaria per memorizzare i dati in input con cui gli algoritmi lavorano. In particolare verrà trattato il problema degli algoritmi di ordinamento: saranno presentati alcuni algoritmi classici (come il merge sort e il bubble sort). Verrà descritta nel dettaglio l'implementazione del selection sort, sia in spazio logaritmico, sia nella sua implementazione tradizionale, con relativo confronto tra di esse.

Quindi tutta la tesi è incentrata sulla programmazione in spazio sublineare. Ma perché questo tipo di algoritmi non sono così diffusi? La risposta è semplice: negli ultimi anni lo sviluppo tecnologico ha portato ad avere hardware sempre più potente ed economico ed il prezzo della memoria (come quello dei processori) è sceso drasticamente.

Questo fatto può essere ricondotto alle esigenze di un utente normale che utilizza il computer per svolgere azioni semplici quali navigare nella rete, scrivere documenti o leggere email e che quindi non si troverà mai a dover gestire una quantità di dati molto più grande della memoria installata nel proprio calcolatore.

È naturale, quindi, che negli ultimi anni l'attenzione si sia concentrata soprattutto sull'ottimizzazione del *tempo* di esecuzione, sia a livello di algoritmi che di linguaggi di programmazione.

In alcuni campi, però, è necessario garantire l'utilizzo di una quantità fissata di memoria:

- le computazioni in cui la garanzia di proprietà formali è *critica*, per esempio i programmi per la gestione di macchine mediche oi programmi che regolano il funzionamento degli aerei o delle centrali elettriche. È chiaro che in questi contesti l'interruzione del programma per mancanza di memoria può produrre risultati catastrofici (compresa la morte di persone);

- i dispositivi integrati (microcontrollori) in cui la memoria è limitata e non c'è possibilità di usare memorie ausiliarie;

- tutte le applicazioni che richiedono algoritmi *online*, che devono, cioè, fornire risultato man mano che leggono dati in input in modo seriale. Questo perché in numerose situazioni non è possibile avere tutti i dati in lettura contemporaneamente ma è necessario avere in output un risultato prodotto dai dati attualmente letti.

Indice

Capitolo 1

IntML un linguaggio LOGSPACE

IntML è un linguaggio funzionale puro con la particolarità che tutti i programmi scritti con questo linguaggio sono LOGSPACE, con la caratteristica, cioè, che il loro utilizzo di memoria è asintoticamente minore di quello occupato dai dati in input.

I problemi che nascono nello sviluppo di algoritmi sub-lineari sono molteplici: pensiamo, per esempio, alla composizione di due algoritmi (situazione che possiamo trovare in qualsiasi programma più complesso di un "Hello world"): essi non possono essere eseguiti in successione perché può non esserci abbastanza spazio per memorizzare i risultati intermedi.

Per evitare questo problema quello che fa IntML è computare su piccole parti di input e output in modo da memorizzare il minor numero di dati possibile. La difficoltà nel fare ciò è evidente: presa una sola parte di input dobbiamo svolgere tutta la computazione e fornire subito un output senza poter memorizzare risultati intermedi. Pensiamo ad una semplice moltiplicazione tra numeri interi: essa ci appare banale, ma scrivere un algoritmo LOGSPACE che svolga questo compito è tutt'altro che triviale. Normalmente, per svolgere questa operazione, moltiplichiamo ogni cifra del moltiplicatore per il moltiplicando e poi sommiamo, seguendo un certo schema, tutti i

risultati intermedi. Questo tipo di algoritmo occupa uno spazio lineare dato che siamo costretti a memorizzare numerosi risultati intermedi prima di giungere al risultato finale: esistono algoritmi LOGSPACE che svolgono la moltiplicazione, ma sono molto più complessi del corrispettivo non log-space.

In un linguaggio di programmazione, normalmente, i dati vengono letti in input, gestiti in qualche modo (quasi sempre copiati in memoria centrale) e la computazione, utilizzando altra memoria aggiuntiva per tener traccia dei risultati parziali, restituisce un qualche tipo di output. In IntML l'input viene letto a pezzi: è il programma stesso a leggere, man mano, i bits necessari alla computazione. Questo stesso meccanismo viene usato anche per l'output.

E' chiaro come questo approccio modifichi radicalmente lo stile di programmazione. IntML si prefigge come obbiettivo quello di fornire all'utilizzatore un modo semplice e intuitivo per scrivere programmi LOGSPACE.

1.1 Computazione Bidirezionale

La computazione bidirezionale è una paradigma fondamentale per utilizzare dati che non possono essere caricati in memoria. Un input che non può venire memorizzato nella sua interezza, chiaramente, non può nemmeno essere letto per intero. Quello che possiamo fare, però, è interrogarlo pezzo per pezzo, o meglio bit per bit, durante il corso della computazione.

IntML si prefigge proprio questo: creare un sistema stabile ed efficiente per poter gestire una computazione bidirezionale senza l'intervento dell'utente.

Supponiamo che esista un insieme di dati che possiamo richiedere pezzo per pezzo. L'interfaccia a questi dati viene fornita da una coppia di tipi (X^-, X^+). X^- rappresenta la nostra richiesta all'input e X^+ codifica le possibili risposte.

Questo meccanismo di computazione è implementato come una rete di nodi.

Figura 1.1

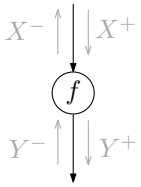

Figura 1.2

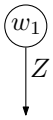

Figura 1.3

Figura 1.4

Ogni nodo ha svariati cavi e reagisce ai messaggi che gli arrivano su uno dei cavi connessi. I cavi vengono rappresentati come frecce ma è importante ricordare che essi sono *bidirezionali*: la freccia significa che per X^+ i messaggi si muovono seguendo il verso indicato, mentre con X^- vengono inviati in direzione opposta.

Diversi nodi possono essere collegati insieme in maniera semplice collegando due cavi con la stessa etichetta.

In questo modo viene garantita una computazione in spazio sub-lineare perché i dati occupano molto più spazio dei messaggi usati per l'accesso ad essi. IntML non si propone di fare soltanto questo, ma fornisce al programmatore anche una serie di costrutti per facilitare e rendere più naturale il modo di scrivere programmi LOGSPACE come le *coppie* e le *funzioni*.

Esempio Poniamo che il nostro dato sia un semplice numero binario formato da otto bit, esso sarà:

$$232 = 11101101$$

tramite il cavo X^- possiamo richiedere il bit che ci interessa, per esempio il 4.

$$11101_4101 \longrightarrow 1$$

Su X^+ verrà trasmessa la risposta alla nostra interrogazione: il risultato sarà 1.

1.2 Introduzione al linguaggio

IntML si caratterizza per essere composto da due classi di termini e tipi.

La definizione di IntML infatti inizia con un normale linguaggio funzionale, che andrà a costituire la **Working class** (WC) del linguaggio.

Per questa classe è stato scelto un semplice linguaggio del prim'ordine con tipi finiti (ma si potrebbe sceglierne uno più espressivo). Già con i tipi e i termini della WC è possibile implementare i circuiti (e quindi i programmi bidirezionali), ma con molte difficoltà legate al nuovo stile di programmazione.

Si estende quindi questo linguaggio con delle primitive per costruire e manipolare le reti di message passing in modo *naturale*: chiamiamo questa l'**Upper class** (UC).

È bene sottolineare subito alcuni aspetti dell'UC.

- Le primitive dell'UC non rappresentano semplicemente i circuiti, ma si ispirano direttamente ai classici costrutti di un linguaggio di programmazione funzionale ad alto livello. È proprio questo che permetterà al programmatore di scrivere programmi bidirezionali senza cambiare il suo stile di programmazione.

- Le primitive sono *estensioni conservative* della WC, e dunque possono programmare solo circuiti che avremmo potuto scrivere, seppur con notevoli complicazioni, anche direttamente nella WC. Questo sarà evidente quando, parlando dell'interprete, mostreremo che i termini dell'UC, prima di essere valutati, vengono *compilati* in termini della WC.

Di seguito verrà riportato e descritto brevemente il sistema di tipi e di termini della **Working class** e della **Upper class**. Per maggiori approfondimenti e per la consultazione di esempi pratici si consiglia la lettura dell'eleborato di Michael Lodi **??**, in cui vengono descritti in maniera chiara ed esaustiva numerose funzioni matematiche utili per la programmazione con IntML.

1.2.1 Working class

Il sistema di tipi della WC è formato da tipi al prim'ordine con variabili di tipo. Possono essere definiti nel seguente modo:

$$A,\ B ::= \texttt{'a} \mid \texttt{1} \mid A{*}B \mid A{+}B$$

dove :

- $\texttt{'a}$ (e $\texttt{'b}$,$\texttt{'c}$, ...) sono variabili di tipo, cioè tipi generici non ancora istanziati, che potranno assumere una qualche composizione legale di altri tipi.
- $\texttt{1}$ rappresenta il tipo del *singoletto*, ovvero un tipo (inteso come *insieme di valori possibili*) che contiene un solo elemento. La cardinalità di tale insieme sarà ovviamente $|\texttt{1}| = 1$.
- Il costruttore $+$ serve a costruire l'unione disgiunta dei tipi A e B. Ricordiamo che questo tipo di unione mantiene traccia dell'insieme di provenienza degli elementi. La cardinalità dell'insieme sarà $|A{+}B| = |A| + |B|$.
- Il costruttore $*$ serve a costruire il prodotto cartesiano tra A e B. Gli elementi di tale insieme saranno *coppie* il cui primo elemento appartiene ad A e il secondo appartiene a B. La cardinalità dell'insieme sarà $|A{*}B| = |A| * |B|$.

Da queste considerazioni, è facile evincere che, per tutti i tipi, *l'insieme dei valori possibili è finito*. Possiamo infatti vedere $+$ e $*$ come operatori sulla cardinalità dei tipi.

I termini della WC sono dati dalla grammatica seguente. Vengono usati *c,d* per indicare variabili della working class e *f,g,h* per i termini.

$$\begin{aligned} f, g, h ::= c \quad &| \quad min_A \mid succ_A(f) \mid eq_A(f, g) \mid inl(f) \mid inr(f) \\ &| \quad case\ f\ of\ inl(c) \Rightarrow g \mid inr(d) \Rightarrow h \mid * \mid \langle f, g \rangle \\ &| \quad fst(f) \mid snd(f) \mid loop(c.f)(g) \mid unbox(t) \end{aligned}$$

Dove:

- min_A, $succ_A$ e eq_A sono costanti che vengono usate rispettivamente per ottenare il minimo, il successore e come test di uguaglianza per qualsiasi tipo di dato generico A. Queste primitive, quindi, possono gestire qualsiasi tipo di dato definito dall'utente.

- *loop* fornisce l'implementazione dell'iterazione basata sull'unione disgiunta.

- *case* fornisce la distinzione tra casi. Anch'esso è basato sull'unione disgiunta

- fst, snd e $\langle f, g \rangle$ sono ripsettivamente i distruttori e i costruttori di coppie. Questi costruttori esistono sia per coppie della WC che per quelle della UC.

1.2.2 Upper class

Il sistema di tipi dell'UC può essere espresso in BNF:

$$X, Y ::= [A] \mid X * Y \mid \{A\}X \multimap Y$$

in cui X e Y sono tipi dell'UC, mentre A è un qualsiasi tipo della WC. Analizziamoli nel dettaglio.

- $[A]$ mi permette di utilizzare nell'UC il tipo della WC A, opportunamente "confezionatò' in un **box**.

- $X * Y$ è sostanzialmente il *prodotto cartesiano* tra i due tipi: serve a definire le *coppie*. Formalmente è definito con l'operatore di *tensore*: $X \otimes Y$.

- $\{A\}X \multimap Y$ vuole rappresentare lo spazio di funzioni indicizzato (che formalmente potremmo scrivere come $A \cdot X \multimap Y$). Come sappiamo, i

linguaggi funzionali permettono di avere in input e in output delle *funzioni*. Questo rappresenta proprio il tipo delle *funzioni che prendono in input un elemento di X e restituiscono un elemento di Y*.
In IntML bisogna però tener conto di una complicazione: per tenere la *complessità in termini di spazio* bassa, dobbiamo indicare esplicitamente *quante volte, al più, posso usare l'argomento in input*. Per fare questo indicizzo X con un tipo A della Working Class.
Si può ad esempio scrivere $2 \cdot [2] \multimap [2]$ (ipotizzando di chiamare per semplicità 2 il tipo $1 + 1$), che rappresenta l*e funzioni che prendono in input un booleano, lo utilizzano al più due volte e restituiscono un booleano*.
Se non viene indicato nulla, A viene istanziato ad 1 e quindi si potrà utilizzare quell'argomento una sola volta.

I termini della UC sono dati dalla grammatica seguente. Vengono usati *x,y* per indicare variabili della upper class e *s,t* per i termini.

$$\begin{aligned} f, g, h ::= x \quad &| \quad \langle t, t \rangle \,|\, \text{let } s \text{ be } \langle x, y \rangle \text{ in } t \,|\, \lambda x.t \,|\, s\, t \\ &| \quad [f] \,|\, \text{let } s \text{ be } [c] \text{ in } t \,|\, \mathit{case}\ f\ \mathit{of}\ \mathit{inl}(c) \Rightarrow s \,|\, \mathit{inr}(d) \Rightarrow t \\ &| \quad \text{copy } t \text{ as } x, y \text{ in } t \,|\, \text{hack}(c.f) \end{aligned}$$

Dove:

- $\lambda x.t$ rappresenta la lambda astrazione necessaria per la costruzione di funzioni.

- let s be $[c]$ in t: è il distruttore di *box*. Il funzionamento è il seguente: *let* riduce s fino a portarlo nella forma $[c]$ e poi sostituisce il valore di v a c, che si suppone essere una variabile libera in t.

- *copy*: all'interno delle funzioni è possibile utilizzare una sola volta le variabili. Per ovviare a questo problema è presente il costrutto *copy* che permette di copiare le variabili un numero finito di volte.

- *hack*: È lo strumento per scrivere nell'UC tutti quei circuiti che non possono essere realizzati usando i costrutti predefiniti. Il costrutto hack (che funziona un po' come l'assembly inline in C) permette infatti di descrivere, caso per caso, le azioni che un circuito compie a seconda degli input che riceve. È sostanzialmente un modo per implementare direttamente (usando quindi la WC) i circuiti, senza essere vincolati al limitante uso di box.

Capitolo 2

Algoritmi di ordinamento

In questo capitolo verranno brevemente descritti gli algoritmi di ordinamento confrontando la loro efficienza sia dal punto di vista del tempo di calcolo che da quello dell'utilizzo di memoria. Successivamente verrà descritta l'implementazione del Sort in ambiente Unix e infine ci sarà un confronto tra l'algoritmo LOGSPACE di cui questo lavoro fornisce un'implementazione e il corrispettivo non LOGSPACE.

2.1 Principali algoritmi di ordinamento

L'utilizzo di memoria è un'aspetto critico per tutti gli algoritmi di ordinamento e uno dei problemi più ricorrenti dell'Informatica è quello di scrivere algoritmi efficienti e, nello stesso tempo, ridurre la memoria utilizzata.

Per capire l'importanza di questo problema basti pensare alla mole di dati contenuti in un database: finché i dati da ordinare sono minori della memoria centrale disponibile essi possono essere caricati nella loro totalità in RAM e gestiti senza gravi problemi di performance. Purtroppo anche piccoli progetti spesso richiedono database molto grandi e copiare tutti i dati dai dischi alla RAM raramente è fattibile.

Da come si può osservare nella tabella 2.1 l'utilizzo di memoria di algoritmi semplici, come l'insertion sort, è minimo (solo la dimensione dei dati

in input). Questo perché il suddetto algoritmo utilizza uno *swap in-place*, una cella di memoria aggiuntiva per memorizzare l'elemento da scambiare in questo modo la memoria dei dati in input verrà utilizzata anche per fornire il risultato finale. Chiaramente algoritmi di questo tipo non sono adatti per grandi file (più grandi della memoria) perché per funzionare correttamente devono avere tutti i dati già disponibili e soprattutto devono poter modificare la memoria dove i dati risiedono.

Nome	Costo computazionale	Memoria utilizzata	Swap in-place
Insertion Sort	$O(n^2)$	1	sì
Quick Sort	$O(n^2)$	$log(n)$	no
Merge Sort	$O(nlog(n))$	n	no
IntML Sort	$O(n^2)$	1	no

Tabella 2.1 Confronto tra alcuni algoritmi di ordinamento

Per ovviare a questo problema un algoritmo largamente utilizzato ed efficiente per ordinare una mole molto grande di dati è il merge sort. Il merge sort suddivide i dati in vari *blocchi*, ognuno dei quali ha come caratteristica quella di poter essere caricato interamente in memoria centrale. Questi blocchi vengono ordinati, riscritti su disco e, infine, vengono riuniti. Al contrario dell'esempio precedente il merge sort utilizza molta più memoria, ma è più efficiente perché esegue letture e scritture dai dischi in modo sequenziale.

Per conclude possiamo notare che l'utilizzo di memoria tutti gli algoritmi di ordinamento è, nel caso migliore, $O(n)$: cioè solamente lineare rispetto all'input. Questo perché tutti gli algoritmi devono poter accedere a tutti i dati da ordinare ma, soprattutto, devono poter modificare la memoria su cui questi dati risiedono.

2.2 Una implementazione reale: Unix Sort

Una delle più importanti e diffuse implementazioni di algoritmo di ordinamento è certamente `sort`, disponibile in tutti gli ambienti Unix. In particolare daremo uno sguardo all'implementazione di sort disponibile in GNU coreutils.

Sort è implementato come un merge a R-vie (*R-way merge*): questo vuol dire che, se sono presenti N bytes da ordinare e M bytes di memoria, il programma creerà $\frac{N}{M}$ sottoliste che poi saranno unite a blocchi di R. Da qui è facile dedurre che l'algortmo esegue $log(\frac{\frac{N}{M}}{log(R)})$ passaggi sui dati e ha una complessità pari a $O(\frac{(\frac{N}{M})log(\frac{N}{M})}{log(R)})$

Particolare attenzione è posta nella gestione dei thread per velocizzare l'eseguzione in macchine multithread/multiprocessore, ma in questo contesto non approfondiremo questo aspetto.

L'utilizzo di memoria, invece, è stata ottimizzato in modo da ordinare rapidamente anche dati molto grandi (GiB e oltre). La memoria instanziata per l'algoritmo di sort è data dalla seguente formula:

$$Mem = \frac{MAX(\frac{Total}{8}, Avail))}{2}$$

dove Total è la memoria totale disponibile nel sistema, e Avail è la memoria attualmente libera. Se i dati sono più grandi di Mem allora vengono spezzati in blocchi al più lunghi Mem, portati in RAM, ordinati e poi scritti su file temporanei nel disco.

Esempio Supponiamo di dover ordinare un file di 1GiB: la memoria totale del sistema è pari a 512MiB e la memoria attualmente libera è 100MiB. Utilizzando la formula precendente otteniamo:

$$\begin{aligned} \text{Mem} &= \frac{MAX(\frac{1024}{8}, 100)}{2} \\ &= \frac{MAX(128, 100)}{2} \\ &= \frac{128}{2} \\ &= 64 \end{aligned}$$

A questo punto il programma di ordinamento divide l'input in blocchi di 64MiB ottenendone 16. Ogni blocco viene ordinato e scritto su disco in un file temporaneo; infine sort ne effettua il merge ottenendo il risultato desiderato.

2.3 Algoritmi di ordinamento e spazio logaritmico

Nelle sezioni precedenti abbiamo visto la stima di complessità di alcuni importanti algoritmi di ordinamento e una implementazione reale di `sort` utilizzato in tutti i sistemi UNIX-like.

La domanda che ci poniamo è la seguente: è possibile implementare qualsiasi tipo di algoritmo di ordinamento rendendolo LOGSPACE? La risposta è tutt'altro che banale. In linea di massima se un algoritmo necessita di un numero variabile di puntatori ai dati esso *non* potrà essere logaritmico. Ma vediamo in dettaglio alcuni algoritmi.

2.3.1 Selection sort

Il selection sort è uno degli algoritmi più semplici in assoluto ed è anche quello che noi usiamo più comunemente. Per esempio se dobbiamo ordinare un mazzo di carte (con un solo seme, per semplicità) banalmente scorriamo tutto il mazzo fino a trovare l'asso che mettiamo da parte, poi partiamo alla ricerca del 2 e una volta trovato lo impiliamo sopra l'asso. Continuiamo cosi fino ad avere tutto il mazzo ordinato.

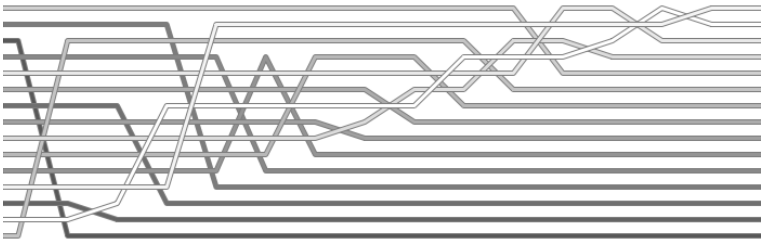

Figura 2.1 Visualizzazione grafica del Selection sort

Dal punto di vista informatico l'implementazione è la seguente:

- scorrere la lista cercando l'elemento più piccolo e spostarlo in testa.
- ripetere questa operazione fino ad avere l'elemento più grande in ultima posizione.

Selection sort è facilmente implementabile in spazio logaritmico: basta sostituire lo spostamento fisico dei dati con un puntatore che tenga traccia dell'elemento interessato.

Cocktail sort

Il Cocktail sort è una variante del selection sort (esiste anche una variante del bubble sort con lo stesso nome, attenzione a non confonderle perché gli algoritmi sono differenti). L'unico cambiamento rispetto all'algoritmo originale è il seguente: ad ogni passaggio viene selezionato sia l'elemento più piccolo che l'elemento più grande ed essi vengo posti rispettivamente ai due estremi della lista.

In questo modo vengo dimezzate le letture.

Pseudocodice

```
1  /* a[i] e' il vettore di "n" elementi da ordinare */
2  int iPos;
3  int iMin;
4
5  /* iterazione su tutto il vettore */
6  for (iPos = 0; iPos < n; iPos++)
7  {
8      iMin = iPos;
9      /* ricerca del minimo */
10     for (i = iPos+1; i < n; i++){
11         /* se l'elemento e' minore allora e' un nuovo minimo */
12         if (a[i] < a[iMin]){
13             /* memorizzazione dell'indice del minimo */
14             iMin = i;
15         }
16     /* scambio di posizione tra l'elemento corrente e il minimo
           trovato*/
17     }swap(a, iPos, iMin);
18 }
```

2.3.2 Bubble sort

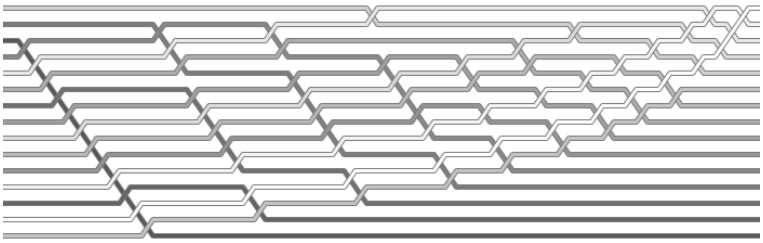

Figura 2.2 Visualizzazione grafica del Bubble sort

Il Bubble sort è uno dei primissimi algoritmi (se non il primo in assoluto) che vengono insegnati in un corso di programmazione. Il funzionamento del bubble sort è semplice: ogni coppia di elementi adiacenti della lista viene

comparata e se essi sono nell'ordine sbagliato vengono invertiti. L'algoritmo scorre poi tutta la lista finché non vengono più eseguiti scambi, situazione che indica che la lista è ordinata.

Conigli e tartarughe

Il problema principale del bubble sort è detto dei *conigli e delle tartarughe*, dove con la parola *conigli* si intendono numeri molto grandi posti all'inizio della lista e con *tartarughe* si definiscono i numeri piccoli posti sul fondo della lista. Dal funzionamento dell'algoritmo descritto precedentemente, possiamo evincere che i conigli verrano spostati molto velocemente nella loro posizione finale (da qui deriva il nome), mentre le tartarughe verrano spostate di una sola posizione alla volta verso la testa della lista.

Questo meccanismo si comporta spostando numerevoli volte gli elementi, in spazio sub-logaritmico non si ha accesso all'input e quindi non c'è la possibilità nemmeno di effettuare questi scambi tra elementi utilizzando direttamente la memoria. In un linguaggio LOGSPACE, quindi, il bubble sort dovrebbe essere implementato utilizzando puntatori. Purtroppo questi puntatori non possono essere un numero costante ma nel caso peggiore, cioè quando ogni elemento deve venir spostato in un altra posizione, sarannò tanti quanti gli elementi della lista. Da questa congettura è chiaro come il bubble sort non possa essere implementato in spazio sub-lineare, il suo utilizzo di memoria sarà sempre lineare.

Esempio: prendiamo una qualsiasi lista lunga N già ordinata tranne che per l'ultimo elemento che, casualmente, è il minore e quindi deve essere spostato in testa. Utilizzando il *selection sort* possiamo ordinare la lista in soli 2 passaggi (il primo passaggio per spostare l'elemento dalla coda alla testa e il secondo per controllare che la lista sia effettivamente ordinata), mentre con il *bubble sort* impiegheremmo n+1 passaggi poiché l'elemento viene spostato soltanto di una posizione alla volta verso la testa!

Pseudocodice

```
1  /* a[i] e' il vettore di "n" elementi da ordinare */
2  int swapped;
3
4  do{
5      swapped = false;
6      /*iterazione su tutto il vittore */
7      for (i = 0; i < n-1; i++){
8          /* l'elemento i e' maggiore di i+1. Li inverto*/
9          if (a[i] > a[i+1]){
10             swpa(a[i],a[i+1]);
11             /* viene effettuata un'inversione, la variabile e'
                   posta a  vero */
12             swapped = true;
13         }
14     }
15 /* il ciclo viene terminato quando non ci sono piu' inversioni
       da fare */
16 while{swapped}
```

2.3.3 Merge sort

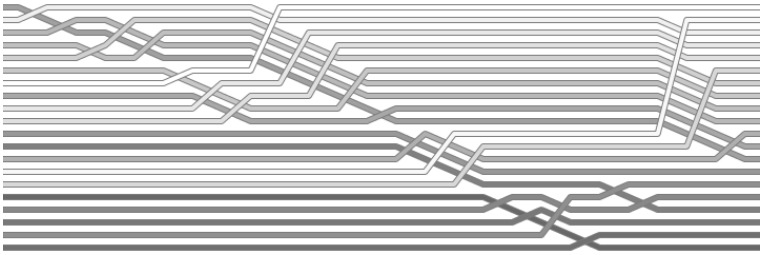

Figura 2.3 Visualizzazione grafica del Merge sort

Il merge sort, contrariamente ai due di sostituzione visti precedentemente, è un algoritmo di comparazione. Quasi tutte le implementazioni di questo algoritmo sono stabili nel senso che hanno la stessa complessità in termini

di tempo ($O(nlogn)$) sia nel caso migliore, cioè quando tutti i dati sono già ordinati, sia nel caso peggiore cioè quando i dati sono già ordinati ma nell'ordine inverso (per esempio in ordine crescente quando li vogliamo ordinati in modo decrescente).

Il merge sort è un algoritmo ricorsivo che utilizza la tecnica del *divide et impera* cioè dividere il problema iniziale in sottoproblemi più piccoli. Il funzionamento dell'algoritmo è il seguente:

- se la lista ha lunghezza 0 o 1 è già ordinata.
- dividere la lista in 2 sotto liste di uguali dimensioni. Se la lista ha un numero dispari di elementi allora una delle due sottoliste avrà un elementio in più.
- ordinare le sottoliste ricorsivamente applicando il merge sort
- unire tutte le sottoliste in una lista ordinata.

La funzione che si occupa dell'unione delle sottoliste (*il merge*) prende in input i primi due elementi di entrambe le sottoliste, li confronta e scrive in output il più piccolo (o il più grande a seconda dell'ordinamento ascendente o discendente).

Questo tipo di ordinamento non è implementabile in spazio LOGARTIMCO: infatti, da come è definito l'algoritmo stesso, occorrerebbero puntatori a tutti i dati per tenere traccia delle loro posizioni nelle sottoliste durante la ricorsione.

Pseudocodice

Per semplicità nella funzione *merge* verranno gestiti i vettori con le primitive delle liste. *append list1 to list2* inserisce il primo elemento della lista *list1* in coda alla seconda e *rest(list1)* restituisce tutti i valori della lista tranne il primo elemento.

```
1  /* a[i] e' il vettore di "n" elementi da ordinare */
2
3  /* questa funzione suddivide il vettore in 2 parti uguali */
4  int merge_sort(int a[]){
5      /* se il vettore e' lungo 1 o 0 la ricorsione termina */
6      if (lenght(a) <= 1){
7          return a;
8      }
9
10     int left[], right[], result[];
11     int middle = lenght(a)/2;
12     /* creazione di un nuovo vettore contenete la prima meta'
           degli elementi */
13     for(i = 0; i < middle; i++){
14         left[i] = a[i];
15     }
16     /* creazione di un nuovo vettore contenete la seconda meta'
           degli elementi */
17     for(i = middle; i <= n; i++){
18         right[i] = a[i];
19     }
20     /* continuo la suddivisione ricorsivamente */
21     left = merge_sort(left);
22     right = merge_sort(right);
23     /* ri-unione in un unico vettore */
24     result = merge(left, right)
25     /* risultato */
26     return result;
27 }
28
29 /* questa funzione si occupa di riunire 2 vettori ordinati */
30 int merge(int left[], right[]){
31     int result
32
33     /* il ciclo termina quando entrambi i vettori sono vuoti */
34     while(length(left) > 0 || length(right) > 0){
35         if (length(left) > 0 && length(right) > 0){
```

```
36              /* controllo quale dei due elementi e' il maggiore
                   */
37              if(first(left) <= first(right)){
38                  append(first(left), result);
39                  left = rest(left);
40                  } else {
41                      append first(right) to result;
42                      right = rest(right);
43              /* nel caso il vettore di destra sia terminato
44               * aggingo tutto quello di sinistra al
45               * risultato */
46              } else if (length(left) > 0) {
47                  append left to result;
48                  left = rest(left);
49              /* come sopra ma nel caso opposto */
50              } else if (length(right) > 0) {
51                  append right to result;
52                  right = rest(right);
53              }
54          }
55      }
56      return result;
57  }
```

Capitolo 3

Implementazione del Selection Sort in IntML

Sin dall'inizio, il problema dell'ordinamento ha destato molto interesse tra gli informatici teorici. Attualmente esistono centinaia di algoritmi che svolgono questa funzione, ognuno ottimizzato per certi tipi di dati o per un certo tipo di hardware. Non esiste, però, nessun algoritmo sub-lineare di ordinamento.

Il motivo principale di questa assenza è che, inanzi tutto, esistono pochissime implementazioni di linguaggi LOGSPACE e la loro diffusione è soltanto in ambito accademico. E' chiaro che in assenza di un linguaggio potente per scrivere algoritmi di ordinamento la loro diffusione resta limitata e si preferisce creare algoritmi intrensicamente non efficenti e poi risolvere i problemi di gestione di memoria in un altro livello, per esempio usando la memoria virtuale fornita dal sistema operativo.

3.1 L'algoritmo

L'algoritmo di sort che si è scelto di implementare è una variante del *selection sort*. Una variante perché, come già detto nel Capitolo 1, IntML, i dati non sono memorizzati internamente e la computazione non viene effet-

tuata su tutto l'input contemporaneamente: non è quindi nemmeno possibile effettuare lo scambio *in-place* all'interno della lista dati presa in ingresso e nemmeno fornire una lista ordinata intera come output.

Per ovviare a questo problema, l'algoritmo prende in input la posizione del numero che interessa all'utente, per esempio il secondo elemento più piccolo, e la posizione del bit che vogliamo avere in output.

A questo punto l'algoritmo interroga l'input bit a bit facendo il confronto tra tutti i numeri e mantenendo un puntatore all'elemento correntemente più piccolo, come avviene nel selection sort classico. Una volta trovato l'elemento richiesto dall'utente viene stampato in output il bit selezionato.

È chiaro che nel caso l'utilizzatore richieda l'intero numero in output e non solo un suo bit basterà eseguire l'algoritmo su tutti i bit in cui il numero è composto.

3.1.1 Complessità algoritmica

Prima di calcolare la complessità a livello di memoria analizziamo la complessità in termini di tempo. Il nostro sort eseguirà tanti cicli sull'input quanto è la posizione del numero richiesto:

Esempio Se disponiamo di una lista di lunghezza 7 e ci serve sapere il valore del terzo numero più piccolo, effettueremo 3 cicli sulla nostra lista. Nel primo ciclo troveremo l'elemento più piccolo in assoluto che chiameremo A, nel secondo passaggio avremo il primo elemento più piccolo ma maggiore di A (quindi il secondo elemento minore globalmente); e infine nella terza iterazione otterremo l'elemento richiesto: il terzo elemento più piccolo.

A questo punto è molto semplice dare un limite superiore alla complessità di questo algoritmo: essa sarà in termini di tempo nel caso peggiore $O(n^2)$. Infatti se l'input è lungo n e l'utente richiede il numero più grande in assoluto verranno fatte n iterazioni.

La complessità, nel caso migliore, è invece $O(n)$, che si presenta quando l'utente richiede l'elemento minimo della lista.

3.1.2 Memorizzazione dei dati

Al contrario di un normale algoritmo di ordinamento non viene memorizzato nessun dato aggiuntivo, ma soprattutto non viene nemmeno utilizzata la memoria dell'input per effettuare uno scambio *al volo* come invece viene fatto, in altri algoritmi di ordinamento. Tutta la computazione viene gestita utilizzando soltanto due puntatori ai dati Essi sono: *curr_min* e *old_min*.

- *current_min*: mantiene la posizione del minimo nell'iterazione corrente
- *old_min*: mantiene la posizione del minimo dell'iterazione precedente.

Esempio di computazione :
Il **grassetto** rappresenta *current_min* e *l'italico* rappresenta *old_min*.

2	3	2	**1**	4	al primo passaggio *old_min* non è settato
2	3	**2**	*1*	4	trovo i più piccoli elementi maggiori di 1
2	**3**	*2*	1	4	trovato il quarto elemento più piccolo

Tabella 3.1 Esempio di ricerca del quarto elemento più piccolo

In questo esempio si è cercato il quarto elemento più piccolo e si può notare come in tre passaggi si è ottenuto il risultato finale. Ad ogni iterazione vengono aggiornati entrambi i puntatori e un contatore viene utilizzato per memorizzare quanti numeri sono già stati ordinati in modo da terminare l'esecuzione al raggiungimento dell'elemento richiesto.

3.2 Il codice

Di seguito viene riportato il codice dell'algoritmo implementato con relativo commento.

3.2.1 NextMin

```
1  nextMin =U
2     fun vecchiomin ->
3     fun lista ->
4      copy lista as lista1,lista23 in
5      copy lista23 as lista2,lista3 in
6
7      let vecchiomin be [oldmin] in
8      loop(
9       fun v ->
10
11      let v be [vw] in
12      let [(fst vw)] be [counter] in
13      let [(fst (snd vw))] be [elem] in
14      let [(fst (snd (snd vw)))] be [minimo] in
15      let [(snd (snd (snd vw)))] be [i] in
16
17      (ifU [counter = zero] (* first iteration *)
18       (let (cmp [elem] [oldmin] lista1) be [vali] in
19        (ifU [( vali = res_gt )] (* piu' grande di oldmin *)
20         (ifU [( i = max)]
21          ([return <uno, elem>])
22          ([continue <uno, <succ elem, <elem, succ i>>>])
23         )
24         (ifU [( i = max)]
25          ([return <zero, elem>])
26          ([continue <zero, <succ elem, <minimo, succ i>>>])
27         )
28        )
29       )
30       (let (cmp [elem] [oldmin] lista2) be [vali] in
```

```
31        (ifU [( vali = res_gt )]
32         (
33          let (cmp [elem] [minimo] lista3) be [res] in
34          ( ifU [( res = res_lt )] (* if we found a new min *)
35            (ifU [( i = max )] (* checking for the last element
                 *)
36             ([return <uno, elem>])
37             ([continue <uno, <succ elem, <elem, succ i>>>])
38            ) (* else it's not a new minimum*)
39            (ifU [( res = res_eq )]
40             (ifU [( i = max )]
41              ([return <succ counter, elem>])
42              ([continue <succ counter, <succ elem, <elem, succ
                    i>>>])
43             )
44             (ifU [( i = max )]
45              ([return <counter, minimo>])
46              ([continue <counter, <succ elem, <minimo, succ
                    i>>>])
47            )
48           )
49          )
50         )
51         ([continue <counter, <succ elem, <minimo, succ i >>>])
52        )
53       )
54      )
55     )[<min : 1+1+1+1, <min : 1+1+1+1, <min : 1+1+1+1, min :
          1+1+1+1>>>];
```

La funzione *nextMin* si occupa della ricerca del più piccolo elemento maggiore del precedente. In altre parole se ci interessa trovare il terzo elemento più piccolo della lista *nextMin* farà 2 iterazioni: nella prima verrà cercato l'elemento più piccolo maggiore del minimo assoluto fornito dalla funzione *trovaMin* e nella seconda iterazione verrà trovato l'elemento più piccolo maggiore dell'elemento trovato precedentemente.

Entrando più nel dettaglio vediamo come *nextMin* prenda in input il minimo precedente e la lista su cui operare. Il ciclo agisce su una tupla composta in questo modo:

```
[<contatore, <elemento, <minimo, iteratore>>>];
```

Dove le variabili vengono utilizzate nel seguente modo:

- contatore: memorizza i numeri già trovati, ogni volta che trovo un minimo valido aumento il contatore di una unità. In questo modo se devo trovare n-esimo minimo interromperò il ciclo quando il contatore avrà un valore pari ad n
- elemento: rappresenta il puntatore all'elemento attualmente analizzato la lista
- minimo: memorizza il minimo elemento trovato.
- iteratore: si occupa di contare il numero di iterazioni già effetuate.

3.2.2 trovaMin

```
trovaMin =U (* find the min elem of the list and return its
    position in the list*)
    fun lista ->
    loop
    (
     fun v ->
     let v be [vw] in
     let [(fst vw)] be [counter] in
     let [(fst (snd vw))] be [elem] in
     let [(fst (snd (snd vw)))] be [minimo] in
     let [(snd (snd (snd vw)))] be [i] in

     (ifU [counter = zero] (* first iteration we set the first
         elem *)
      ([continue <uno, <succ elem, <elem, succ i>>>])
      (let (cmp [elem] [minimo] lista) be [res] in
```

```
15        ( ifU [( res = res_lt )] (* if we found a new min *)
16          (ifU [( i = max )] (* checking for the last element *)
17           ([return <uno, elem>])
18           ([continue <uno, <succ elem, <elem, succ i>>>])
19          ) (* else it's not a new minimum*)
20          (ifU [( res = res_eq )]
21           (ifU [( i = max )]
22            ([return <succ counter, elem>])
23            ([continue <succ counter, <succ elem,<minimo, succ
                 i>>>])
24           )
25           (ifU [( i = max )]
26            ([return <counter, minimo>])
27            ([continue <counter, <succ elem, <minimo, succ i>>>])
28           )
29         )
30        )
31       )
32      )
33      ) [<min : 1+1+1+1, <min : 1+1+1+1, <min : 1+1+1+1, min :
          1+1+1+1>>>];
```

trovaMin è una versione semplificata di *nextMin* che si occupa semplicemente di trovare il minimo assoluto presente nella lista. Viene utilizzata come prima funzione chiamata per iniziare la computazione.

3.2.3 trovaIndice

```
1 trovaIndice =U
2     fun obj ->
3     fun lista ->
4     copy lista as lista1,lista23 in
5     copy lista23 as lista2,lista3 in
6     let obj be [goal] in
7
8     let (trovaMin lista1) be [res] in
9     let (sub [goal] [fst res]) be [num] in
```

```
10      (ifU [ num = zero ] (* ho gia' trovato l'elemento*)
11       ([snd res])
12       (loop(
13             fun v ->
14             let v be [vw] in
15             let [fst vw] be [ob] in
16             let [snd vw] be [minimo] in
17             let (nextMin [minimo] lista2) be [res] in
18
19             let (sub [ob] [fst res]) be [num] in
20             (ifU [ num = zero]
21              ([return (snd res)])
22              ([continue <num, (snd res)>])
23             )
24            )[<num, (snd res)>])
25      );
```

Questa funzione ha il compito di richiamare, inizialmente *trovaMin* e succesivamente *nextMin*, per poi controllare quando si è trovato l'elemento richiesto dall'utente. Notiamo che la funzione prende in input il numero richiesto dall'utente e la lista su cui operare. Il ciclo viene effettuato su una coppia:

```
1       [<passi, vecchiomin>]
```

Dove:

- passi: viene utilizzato per contare quanto siamo vicini al numero richiesto dall'utente. Quando *passi* è pari a zero significa che abbiamo trovato l'elemento richiesto dall'utente e la computazione si interrompe.
- vecchiomin: viene utilizzato per memorizzare il minimo precedente. Questa informazione viene passata alle funzione *nextMin* descritta precedentemente.

3.2.4 sort

```
1 sort =U
2     fun lista ->
3     fun pos ->
4     fun bit ->
5     copy lista as lista1,lista2 in
6     let (trovaIndice pos lista1) be [res] in
7     lista2 [(res)] bit;
```

La funzione *sort*, infine, è molto semplice e si occupa di fare da wrapper tra le funzioni descritte e l'utente. *sort* prende in input la lista su cui lavorare e la posizione dell'elemento e del bit richiesto. In output viene stampato il bit richiesto dall'utente.

Conclusioni e Sviluppi Futuri

Riepilogo del lavoro svolto

Inizialmente è stato presentato IntML, una implementazione reale di un linguaggio LOGSPACE. Sono stati visti i problemi che comporta la programmazione in spazio sub-lineare: è molto difficile per un programmatore scrivere algoritmi che gestiscano una quantità molto vasta di dati senza uno strumento adatto a questo scopo.

Per questo motivo sono state descritte, in breve, la sintassi e la semantica di IntML in modo da rendere chiaro al lettore come questo linguaggio possa aiutare il programmatore a scrivere e creare programmi utilizzando la *programmazione bidirezionale* senza preoccuparsi di comprendere il funzionamento delle reti e delle macchine di turing offline che implementano questo paradigma.

Una volta introdotto questo nuovo linguaggio si è passati alla descrizione di vari algoritmi di ordinamento e di come questi potrebbero essere implementati in spazio sub-logaritmico.

Si è visto come il problema dell'ordinamento sia un problema critico: al giorno d'oggi con la diffusione cosi vasta di internet, sempre più spesso, capita di dover gestire grandi quantita di dati che risiedono, frequentemente, su server remoti. Per questo è cresciuta l'importanza di creare algoritmi che possano lavorare e gestire informazioni anche se esse non possono venir copiate e memorizzate nella macchina su cui viene eseguito il programma. In questi casi IntML si rivela essere un linguaggio dotato di buone potenzialità

per risolvere questo tipo di problemi permettendo al programmatore di scrivere algoritmi con la proprietà formale di effettuare tutta la computazione utilizzando *meno* memoria di quella occupata dall'input.

Infine, nell'ultima parte di questo elaborato, viene fornita un'implementazione di un algoritmo di ordinamento LOGSPACE con relativa descrizione e spiegazione del codice.

Ottimizzazioni dell'algoritmo

Durante l'implementazione dell'algoritmo di ordinamento effettuata utilizzando il linguaggio IntML si è posta particolare attenzione all'ottimizzazione della memoria, giungendo, alla fine dello sviluppo, soltanto all'utilizzo di due puntatori per gestire i dati. La complessità del codice non è stata ottimizzata per due motivi:

- lo scopo della tesi era presentare una versione funzionante e completamente utilizzabile di un algoritmo di ordinamento LOGSPACE. Aggiungere ottimizzazioni riguardanti la velocità di esecuzione avrebbe soltanto reso il codice più difficile da comprendere e difficile da mantenere.

- Anche l'implementazione dell'interprete di IntML predilige la semplicità e la chiarezza a discapito delle prestazioni (almeno in questo fase iniziale del suo sviluppo). Le ottimizzazioni possibili a livello di algoritmo, perciò, non avrebbero portato benefici osservabili.

Come descritto nel Capitolo 2 la complessità del selection sort è pari ad $O(n^2)$ una semplice ottimizzazione possibile sarebbe, conoscendo la lunghezza della lista da ordinare, quella di confrontare se la posizione dell'elemento desiderato dall'utente sia più vicino al minimo o al massimo. Se il numero fosse molto vicino al minimo, inizieremmo allora a ricercare l'elemento normalmente, se invece fosse più vicino al massimo, partiremmo dall'elemento più grande.

Esempio: consideriamo una lista di 10 elementi: se l'utente ci richiede il secondo elemento più piccolo, allora utilizzeremo l'algoritmo standard. Nel caso l'utente richiedesse il nono elemento più piccolo, invece di iterare nove volte sulla lista partendo dalla ricerca del minimo elemento, l'algoritmo partirebbe alla ricerca del secondo elemento più grande, cioè proprio del nono elemento più piccolo! In questo modo le iterazioni sulla lista vengono dimezzate nel caso peggiore.

Sviluppi futuri

Come già menzionato nell'introduzione esistono molti ambiti *potenziali* in cui un lavoro di questo tipo può essere utilizzato. È ovvio che, essendo IntML un linguaggio di recentissima introduzione, moltissimi aspetti possono essere sviluppati e migliorati.

Per quanto riguarda il linguaggio, esso parte da una base molto semplice. Si potrebbe sicuramente sostituire la Working Class con un linguaggio più espressivo e, d'altro canto, aumentare il numero di costrutti che l'Upper Class mette a disposizione per facilitare e rendere ancora più naturale la programmazione in IntML.

L'interprete può essere esteso in modo da migliorarne l'usabilità (interfaccia grafica, evidenziazione della sintassi...) e facilitare il compito del programmatore (messaggi di errori chiari ed esplicativi, debugger...).

Un'ambito in cui si sta lavorando attualmente è la realizzazione di un sistema che permetta al programmatore di ottenere direttamente tutti gli output di una determinata funzione, senza interrogarla in ogni suo input. Come abbiamo visto nell'algoritmo di ordinamento questa nuova caratteristica sarebbe molto utile per ottenere direttamente una lista ordinata senza che l'utente debba passare manualmente ogni singolo bit alla funzione di *sort*.

Bibliografia

[DG06] A. Dovier and R. Giacobazzi. Dispense per il corso di fondamenti dell'informatica: Linguaggi formali, calcolabilità e complessità. 2006. URL: `www.dimi.uniud.it/~dovier/DID/dispensa.pdf`.

[DLS10a] U. Dal Lago and U. Schöpp. Functional programming in sublinear space. *European Symposium on Programming (ESOP2010), Cyprus, LNCS 6012, ©Springer-Verlag*, 2010.

[DLS10b] U. Dal Lago and U. Schöpp. Type inference for sublinear space functional programming. *ASIAN Symposium on Programming Languages and Systems (APLAS 2010), Shangai, China*, 2010.

[M.10] Lodi M. Programmazione in spazio sublogaritmico: una libreria di funzioni matematiche. 2010. URL: `http://amslaurea.cib.unibo.it/1444/`.

[MG06] S. Martini and M. Gabbrielli. *Linguaggi di programmazione: principi e paradigmi*. McGraw-Hill Italia, 2006.