Complessità del calcolo

Caso pessimo

$T_M(n) = \max\{T_M(x), |x| = n\}$
$S_M(n) = \max\{T_M(x), |x| = n\}$

Notazioni

- O-grande: limite asintotico superiore.
  Data $g(n)$, $O(g(n)) = \{f(n) \mid \exists c, n_0\,(c, n_0 > 0 : \forall n \geq n_0 0 \leq f(n) \leq cg(n))\}$
- Ω-grande: limite asintotico inferiore.
  Data $g(n)$, $\Omega(g(n)) = \{f(n) \mid \exists c, n_0\,(c, n_0 > 0 : \forall n \geq n_0 0 \leq cg(n) \leq f(n))\}$
- Θ-grande: limite asintotico sia superiore sia inferiore.
  Data $g(n)$, $\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0\,(c_1, c_2, n_0 > 0 : \forall n \geq n_0 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n))\}$

Teoremi di accelererazione lineare

- Se L è accettato da una MT M a k nastri con complessità $S_M(n)$, per ogni $c > 0 (c \in R)$ si può costruire una MT M' a k nastri con complessità $S_{M'}(n) < cS_M(n)$
- Se L è accettato da una MT M a k nastri con complessità $S_M(n)$, si può costruire una MT M' a 1 nastro (*non* a nastro singolo) con complessità $S_{M'}(n) = S_M(n)$
- Se L è accettato da una MT M a k nastri con complessità $S_M(n)$, per ogni $c > 0 (c \in R)$ si può costruire una MT M' a 1 nastro con complessità $S_{M'}(n) < cS_M(n)$
- Se L è accettato da una MT M a k nastri con complessità $T_M(n)$, per ogni $c > 0 (c \in R)$ si può costruire una MT M' (a $k+1$ nastri) con complessità $T_{M'}(n) = \max\{n+1, cT_M(n)\}$

Conseguenze pratiche

- Lo schema di dimostrazione è valido per qualsiasi tipo di modello di calcolo, quindi anche per calcolatori reali (es.: aumentare il parallelismo fisico (16bit $\rightarrow$ 32bit $\rightarrow \ldots$)).
- Aumentando la potenza di calcolo in termini di risorse disponibili si può aumentare la velocità di esecuzione, ma il miglioramento è al più lineare.
- Miglioramenti di grandezza superiore possono essere ottenuti solo cambiando algoritmo e non in modo automatico.

Macchina RAM

Glossario

Accumulatore: è la prima cella del modello della memoria, indicata con `M[0]`.
Immediato: è un numero intero.
ADD: Sono le operazioni elementari di somma (`ADD`), sottrazione (`SUB`), moltiplicazione (`MULT`), divisione (`DIV`).

Costi Logaritmici

Il costo della copia di un numero n da una cella all'altra è tante micro-operazioni elementari quanti sono i bit necessari a codificare n, cioè $\log(n)$.
Il costo dell'accesso ad una cella di posizione n-esima è l'apertura di $\log(n)$ gate logici ad altrettanti banchi di memoria.
In forma sintetica:

$$l(i) = \textit{if } i = 0 \textit{ then } 1 \textit{ else } \lfloor \log_2 |i| \rfloor + 1$$

Teorema di correlazione polinomiale

Sotto "ragionevoli" ipotesi di criteri di costo (il criterio di costo costante per la RAM non è ragionevole) se un problema è risolvibile mediante un modello di calcolo M_1 con complessità $C_1(n)$, allora è risolvibile da qualsiasi altro modello di calcolo M_2 con complessità $C_2(n) \leq P_2(C_1(n))$, essendo P_2 un opportuno polinomio.

Comando		Operazione	Complessità	Descrizione
`LOAD`	`X`	`M[0] = M[X]`	$l(x)$	Carica in `M[0]` il contenuto della cella X
`LOAD=`	`X`	`M[0] = X`	$l(x) + l(M[x])$	Carica in `M[0]` l'immediato X
`LOAD*`	`X`	`M[0] = M[M[X]]`	$l(x) + l(M[x]) + l(M[M[x]])$	Carica in `M[0]` dall'indirizzo M[X]
`STORE`	`X`	`M[X] = M[0]`	$l(x) + l(M[0])$	Carica in `M[X]` il contenuto di M[0]
`STORE*`	`X`	`M[X] = M[M[0]]`	$l(x) + l(M[x]) + l(M[0])$	Carica in `M[X]` dall'indirizzo M[0]
`ADD`	`X`	`M[0] = M[0] + M[X]`	$l(M[0]) + l(x) + l(M[x])$	Carica in `M[0]` il risultato dell'operazione
`ADD=`	`X`	`M[0] = M[0] + X`	$l(M[0]) + l(x)$	
`ADD*`	`X`	`M[0] = M[0] + M[M[X]]`	$l(M[0]) + l(x) + l(M[x]) + l(M[M[x]])$	
`READ`	`X`	`M[X] = read()`	$l(\textit{input value}) + l(x)$	Salva in X il valore letto in input
`READ*`	`X`		$l(\textit{input value}) + l(x) + l(M[x])$	
`WRITE`	`X`	write(M[X])	$l(x) + l(M[x])$	Scrive in output il valore di X
`WRITE=`	`X`	write(X)	$l(x)$	Scrive in output l'immediato X
`WRITE*`	`X`	write(M[M[X]])	$l(x) + l(M[x]) + l(M[M[x]])$	Scrive in output l'indirizzo di X
`JUMP`	`label`	`PC= b(label)`	1	Salta alla label indicata
`JZ`	`label`	`if M[0] == 0`	$l(M[0])$	Salta alla label indicata se l'accumulatore è 0.
`JGZ`	`label`	`if M[0] > 0`	$l(M[0])$	Salta alla label indicata se l'accumulatore è maggiore di 0.
`HALT`			1	Interrompe l'esecuzione del programma.

Algoritmi

Si adotta il criterio di **costo costante** (manipoliamo numeri che non richiedono quantità di memoria molto più grandi della dimensione dell'input).
Ogni istruzione viene eseguita in un tempo costante c_i.

Complessità di un algoritmo *divide et impera*

- Si divide il problem in b sottoproblemi, ciascuno con dimensione $\frac{1}{b}$.
- Se il problema ha dimensione n piccola a sufficienza ($n < c$, c costante caratteristica del problema), esso può essere risolto in tempo costante ($\Theta(1)$).
- $D(n)$ è il costo di dividere il problema, e $C(n)$ è il costo di ricombinare i sottoproblemi. $T(n)$ è il costo per risolvere il problema totale.

$$\text{Equazione di Ricorrenza} \quad T(n) = \begin{cases} \Theta(1) & \text{se } n < c \\ D(n) + aT(\frac{n}{b}) + C(n) & \text{altrimenti} \end{cases}$$

Insertion Sort

```
INSERTION-SORT(A)
    for j = 2 to A.length
        key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

Merge Sort

```
MERGE-SORT(A, p, r)
    if p < r
        q = ⌊(p+r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q+1, r)
        MERGE(A, p, q, r)

MERGE (A, p, q, r)
n1 = q-p+1
n2 = r-q
CreaArray(L[1...n1+1] e R[1...n2+1])
for i = 1 to n1
    L[i] = A[p+i-1]
for j = 1 to n2
    R[j] = A[q+j]
L[n1+1] = ∞
R[n2+1] = ∞
i = 1
j = 2
for k = p to r
    if L[i] <= R[j]
        A[k] = L[i]
        i = i+1
    else
        A[k] = R[j]
        j = j+1
```

Heapsort

```
PARENT(i)
    return ⌊i/2⌋
LEFT(i)
    return 2*i
RIGHT(i)
    return 2*i+1

MAX-HEAPIFY(A, i)
    l = LEFT(i)
    r = RIGHT(i)
    if l <= A.heapsize and A[l] > A[i]
        max = l
    else
        max = i
    if r <= A.heapsize and A[r] > A[max]
        max = r
    if max != i
        swap A[i] ↔ A[max]
        MAX-HEAPIFY(A, max)

BUILD-MAX-HEAP(A)
    A.heapsize = A.length
    for i = A.length/2 to 1
        MAX-HEAPIFY(A, i)

HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length to 2
        swap A[1] ↔ A[i]
        A.heapsize = A.heapsize - 1
        MAX-HEAPIFY(A, 1)
```

Quicksort

```
QUICKSORT(A, p, r)
    if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q-1)
        QUICKSORT(A, q+1, r)

PARTITION(A, p, r)
    x = A[r]
    i = p-1
    for j = p to r-1
        if A[j] <= X
            i = i+1
            swap A[i] ↔ A[j]
    swap A[i+1] ↔ A[r]
    return i+1
```

Counting Sort

```
COUNTING-SORT(A, B, k)
    for i = 0 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    for i = 1 to k
        C[i] = C[i] + C[i-1]
    for j = A.length to 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Risoluzione di ricorrenze

- Metodo della sostituzione
  - formulare un'ipotesi di soluzione
  - sostituire la soluzione nella ricorrenza, e dimostrazione (per induzione) che è in effetti una soluzione.
- Teorema dell'esperto (Master Theorem)
  - Data la ricorrenza $T(n) = aT(\frac{n}{b}) + f(n)$, in cui $a \geq 1$, $b > 1$, $\lfloor \frac{n}{b} \rfloor$ o $\lceil \frac{n}{b} \rceil$.
    1. se $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ per qualche $\varepsilon > 0$, allora $T(n) = \Theta\left(n^{\log_b a}\right)$
    2. se $f(n) = \Theta\left(n^{\log_b a}\right)$, allora $T(n) = \Theta\left(n^{\log_b a} \log(n)\right)$
    3. se $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ per qualche $\varepsilon > 0$, e $af\left(\frac{n}{b}\right) \leq cf(n)$ per qualche $c < 1$ e per tutti gli n grandi a sufficienza, allora $T(n) = \Theta(f(n))$
  - Se $f(n) = \Theta\left(n^k\right)$, con k una qualche costante:
    1. se $k < \log_b a \rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$
    2. se $k = \log_b a \rightarrow T(n) = \Theta\left(n^k \log(n)\right)$
    3. se $k > \log_b a \rightarrow T(n) = \Theta\left(n^k\right)$