# Algoritmi e Principi dell'Informatica

Seconda prova in itinere - 4 Febbraio 2011

**Tempo a disposizione: 1h30**

## Esercizio 1 (12 punti)

Si consideri il problema di calcolare, a partire da un valore *n* in input, il valore $n2^n$.

1. Si descriva in modo informale, ma sufficientemente preciso, una macchina di Turing a k nastri con alfabeto binario {0,1} (sia per il nastro di input che per i nastri di memoria) che calcola la funzione desiderata, supponendo che il valore *n* si trovi codificato in binario nel nastro di input. Si diano le complessità temporale e spaziale della macchina scritta.
2. Si descriva (in pseudocodice) una macchina RAM che calcola la funzione desiderata. Si diano le complessità temporale e spaziale della macchina scritta, calcolate con criterio di costo logaritmico.

Quale dei 2 meccanismi proposti è il più efficiente per eseguire il calcolo in questione?

## Esercizio 2 (14 punti)

1. L'algoritmo di `MERGE-SORT` visto a lezione non ordina "sul posto", in quanto usa una quantità di memoria aggiuntiva (di dimensione non costante) al momento di fare il merge dei 2 sottoarray (che vengono ricopiati ad ogni chiamata di `MERGE` in 2 array di supporto `L` ed `R`).

   1a. Scrivere una variante `MERGE-SORT-IN-PLACE` dell'algoritmo di `MERGE-SORT`, che però ordini sul posto.

   1b. Dire se e come cambia la complessità temporale di `MERGE-SORT-IN-PLACE` rispetto a quulla di `MERGE-SORT`.

2. Definire un algoritmo `MERGE-SORT-LIST-IN-PLACE` che ordini sul posto liste doppiamente concatenate, e darne la complessità temporale.

**NB1:** E' preferibile che gli algoritmi vengano scritti in pseudocodice; tuttavia, verranno accettate anche soluzioni in cui gli algoritmi siano descritti in modo informale, purché la descrizione sia sufficientemente precisa per poter valutare la complessità degli algoritmi.
**NB2:** Il punteggio assegnato sarà tanto più alto quanto migliore sarà la complessità degli algoritmi proposti.

## Esercizio 3 (6 punti)

Dire, giustificando opportunamente le risposte, se le seguenti affermazioni sono vere o false.

1. Ogni volta che un nuovo nodo è aggiunto ad un BST mediante l'algoritmo `TREE-INSERT` visto a lezione, il nodo è aggiunto come foglia dell'albero.

2. Ogni volta che un nuovo nodo è aggiunto ad un albero red-black mediante l'algoritmo `RB-INSERT` visto a lezione, il nodo è aggiunto come foglia dell'albero (più precisamente, per come sono stati realizzati gli alberi red-black a lezione, viene inserito come nodo interno in cui entrambi i suoi figli sono il nodo sentinella `T.nil`).

## Soluzioni

## Esercizio 1

**1.**

Per risolvere in modo efficiente il problema dato con una MT, basta considerare che, per numeri codificati in binario, moltiplicare per 2 corrisponde ad aggiungere uno 0 come cifra meno significativa. Per questo, per ottenere il valore $n2^n$ è sufficiente aggiungere $n$ volte 0 alla fine della stringa binaria che rappresenta $n$ (supponendo che la cifra più significativa sia a sinistra).
Quindi, una MT (traduttrice) che esegue il calcolo desiderato può avere 1 nastro di memoria e comportarsi come segue.

- Innanzi tutto copia $n$ (in binario) sul nastro $T_1$. Contemporaneamente scrive $n$ (sempre in binario) in uscita.
- Con un ciclo, decrementa via via il contatore in $T_1$, e ad ogni decremento appende uno 0 in fondo al nastro di uscita.

Una tale macchina di Turing deve ripetere il ciclo $n$ volte, ed ogni ripetizione costa $\log(n)$ per decrementare il contatore in $T_1$. In totale, quindi, la complessità temporale è $n \cdot \log(n)$, con $n$ valore del numero in input. La complessità spaziale è invece $\log(n)$ (vengono contati solo i nastri di memoria).
Si noti che se, come è abitudine, per la MT consideriamo la lunghezza $m$ della stringa in input come "dimensione del dato", essendo come noto $m = \log(n)$ (cioè $n = 2^m$) allora la complessità temporale diventa $m2^m$, e quella spaziale $m$.

**2.**

Il seguente pseudocodice risolve il problema in questione.

```
1 read(n)
2 ris := 1
3 for i := 1 to n
4   ris := ris*2
5 ris := n*ris
6 write(ris)
```

Il ciclo 3-4 viene eseguito $n$ volte, e ad ogni iterazione *ris* vale $2^i$. Il suo costo, a criterio di costo logaritmico, è quindi $\Sigma^n_{i=1} \log(2^i)$ cioè $\Sigma^n_{i=1} i$, che è $\Theta(n^2)$.

Dei 2 meccanismi di calcolo presentati, quindi, quello realizzato dalla MT è il più efficiente.
Tuttavia, è possibile scrivere una macchina RAM che realizza il calcolo desiderato, ma con complessità migliore.
L'algoritmo è il seguente

```
1  read(n)
2  m := n
3  ris := 1
4  temp := 1
5  while m > 0
6    if m mod 2 = 1
7      ris := ris * temp
8    temp := temp * temp
9    m := m div 2
10 ris := ris*n
11 write(ris)
```

In questo caso il ciclo 5-9 viene eseguito solo $\log(n)$ volte, e l'operazione più onerosa del ciclo è la 7 (che nel caso pessimo, quello in cui $n$ sia una potenza di 2 meno 1, per esempio 15 viene eseguita ad ogni iterazione del ciclo), che costa $\log(2^{2^i} \cdot 2^{2^i})$, cioè $\log(2^{2^{i+1}})$, cioè $2^{i+1}$. Quindi, il costo del ciclo 5-9 è $\Sigma^{\log n}_{i=1} 2^{(i+1)}$ che è $2^{(\log(n)+2)}$, che è $\Theta(n)$.
In realtà, con un meccanismo per certi versi analogo a quello descritto ora per la macchina RAM, è possibile ottenere una complessità $\Theta(n)$ anche per la MT.

## Esercizio 2

**1.**

Per fare in modo che MERGE-SORT ordini sul posto occorre modificare l'algoritmi di MERGE, evitando di usare gli array L ed R. Ciò si ottiene "shiftando" la parte del sottoarray di sinistra ancora da sistemare quando viene preso un elemento da destra.

Lo pseudocodice per il nuovo algoritmo di MERGE, chiamato ora MERGE-IN-PLACE. è il seguente:

```
MERGE-IN-PLACE (A, p, q, r)
1 j := q+1
2 k := p
3 while k < j and j <= r
4   if A[k] > A[j]
5     temp := A[j]
6      // ora spostiamo la parte del sottoarray di sinistra ancora da
sistemare una posizione a destra
7     for l := j downto k+1
8       A[j] := A[j-1]
9     A[k] := temp
10    j := j + 1
11  k := k+1
12  // non c'e' ramo else, perche' se A[k] <= A[j], allora A[k] e' gia' a
posto
```

In questo caso l'algoritmo di MERGE-IN-PLACE costa $\Theta(n^2)$ nel caso pessimo, in quanto, ad ogni ripetizione del ciclo 3-12, può richiedere di spostare $j$-$k$+1 celle che è un numero $\Theta(n)$ (per esempio nel caso in cui tutto il sottoarray di destra sia più piccolo del sottoarray di sinistra, ogni ripetizione del ciclo 7-8 costa $n/2$, con $n=r-p+1$ la lunghezza del sottoarray che si sta ordinando).

Quindi, la ricorrenza di MERGE-SORT-IN-PLACE nel caso pessimo diventa

$T(n) = 2T(n/2) + n^2$

la cui soluzione è facile vedere essere, usando il metodo dell'esperto, $\Theta(n^2)$, con un peggioramento quindi rispetto a MERGE-SORT.

**2.**

Nel caso di una lista, invece, è possibile fare il MERGE senza dovere ricorrere ad array ausiliari, né di dover spostare interi sottoarray: è sufficiente modificare un numero costante di puntatori per ottenere l'effetto di posizionare i nodi nel punto giusto durante il MERGE, come nel seguente algoritmo (la lista L va aggiunta come parametro per poter modificare l'attributo head nel momento in cui un nodo va messo in testa alla lista):

```
SUBLIST-MERGE-IN-PLACE(L, h, m, t)
1 j := m.next
2 k := h
3 while k ≠ j and j ≠ r.next
4   if k.key > j.key
5     temp := j.next
6     // ora stacchiamo il nodo j, per riattaccarlo prima di k
7     j.prev.next := j.next
8     if j.next /= NIL
9       j.next.prev := j.prev
10    if k.prev /= NIL
11      k.prev.next := j
12    else L.head := j
13    j.prev := k.prev
14    j.next := k
15    k.prev := j
```

```
15    j := temp
17  else k := k.next // k va spostato solo se j non è stato staccato e
attaccato prima di esso
```

In questo caso c'è un solo ciclo, quello delle linee 3-17, ed esso viene ripetuto al più *n* volte.
Quindi, la complessità di `SUBLIST-MERGE-IN-PLACE` è ancora Θ(n), la ricorrenza di `LIST-MERGE-SORT-IN-PLACE` è la solita (`LIST-MERGE-SORT-IN-PLACE` è lo stesso di `LIST-MERGE-SORT` visto a esercitazione, salvo che la chiamata a `SUBLIST-MERGE` in `SUBLIST-MERGE-SORT` va sostituita con quella a `SUBLIST-MERGE-IN-PLACE`):
$T(n) = 2T(n/2) + n$
e la complessità è ancora Θ(n log(n)).

## Esercizio 3

Entrambe le affermazioni sono vere.
E' banale vedere che l'affermazione 1 è vera: l'algoritmo di `TREE-INSERT` scende nell'albero fino ad arrivare ad un nodo `NIL`, e attacca il nuovo nodo al posto del `NIL`, non attaccando niente altro sotto al nuovo nodo, che quindi è una foglia.

Per vedere che anche l'affermazione 2 è vera basta notare che `RB-INSERT` si comporta esattamente come `TREE-INSERT` fino a che il nuovo nodo viene attaccato come foglia; quindi viene invocato `RB-INSERT-FIXUP` sul nodo *z* appena inserito. Si noti che i figli di un nodo su cui viene invocato `RB-INSERT-FIXUP` non cambiano nei casi 1 e 3, e possono cambiare solo nel caso 2 (in cui il fratello di *z* diventa suo figlio).
Tuttavia, il caso 2 non può verificarsi per il nodo appena attaccato ad un albero da `RB-INSERT`. Infatti, se il caso 2 si verificasse per il nodo appena inserito, siccome il padre *x* di *z* è rosso, esso prima dell'inserimento di *z* poteva solo avere entrambi i figli uguali a `T.nil`, o le proprietà degli alberi RB sarebbero state violate.
Tuttavia, per avere il caso 2 il fratello *y* di *x* dovrebbe essere nero, e quindi le altezze nera del sottoalbero di radice *x* e di quello di radice *y* sarebbero diverse (quella di *x* è 0, quella di *y* è almeno 1), il che di nuovo violerebbe le proprietà degli alberi RB.
Di conseguenza il nodo *z* rimarrà una foglia dell'albero anche dopo `RB-INSERT-FIXUP`.