Problem Set 2 (con soluzioni)

docente: Luciano Gualà

Esercizio 1 (*equazioni di ricorrenza*)
Si risolvano le seguenti equazioni di ricorrenza. Si assuma sempre $T(1) = 1$.

(*a*) $T(n) = T(n-10) + 10$.

(*b*) $T(n) = T(n/2) + 2^n$.

(*c*) $T(n) = T(n/3) + T(n/6) + n^{\sqrt{\log n}}$.

(*d*) $T(n) = T(\sqrt{n}) + \Theta(\log\log n)$.

(*e*) $T(n) = T(n/2 + \sqrt{n}) + \Theta(1)$.

(*f*) $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

Soluzione esercizio 1

(*a*) Per iterazione.

$$T(n) = T(n-10) + 10 = T(n-20) + 20 = T(n-30) + 30 = \ldots = T(n-10i) + 10i$$

Quando $i = (n-1)/10$, abbiamo $T(n) = T(1) + n - 1 = n = \Theta(n)$.

(*b*) Usiamo il teorema Master. Dobbiamo confrontare $n^{\log_2 1}$ con 2^n. Quest'ultima funzione è chiaramente polinomialmente più veloce della prima. Siamo nel caso 3 (verificare per esercizio tutte le condizioni) e quindi $T(n) = \Theta(2^n)$.

(*c*) Dimostriamo che $T(n) = \Theta(n^{\sqrt{\log n}})$. Possiamo stimare $T(n)$ per eccesso nel seguente modo. $T(n) \leq S(n) = 2S(n/3) + n^{\sqrt{\log n}}$. Usando il teorema Master su $S(n)$ abbiamo (caso 3) $S(n) = \Theta(n^{\sqrt{\log n}})$. Da cui segue che $T(n) = O(n^{\sqrt{\log n}})$. La relazione $T(n) = \Omega(n^{\sqrt{\log n}})$ è ovvia.

(*d*) Facciamo un cambio di variabile, $m = \log_2 n$, da cui abbiamo che $T(n) = S(m) = S(m/2) + \log m$. Possiamo risolvere $S(m)$ con il metodo dell'albero della ricorsione: abbiamo $O(\log m)$ livelli, ognuno dei quali costa al più $\log m$. Quindi $S(m) = O(\log^2 m)$, da cui segue $T(n) = O((\log\log n)^2)$.

(*e*) Visto che $T(n)$ è monotonicamente crescente, per n abbastanza grande possiamo minorare e maggiorare il termine $T(n/2 + \sqrt{n})$ nel seguente modo:

$$T(n/2) \leq T(n/2 + \sqrt{n}) \leq T(n/2 + n/4) = T(3n/4).$$

Quindi un lower bound a $T(n)$ è $S(n) = S(n/2) + \Theta(1)$, mentre un upper bound è $S'(n) = S'(3n/4) + \Theta(1)$. Entrambe le precedenti equazioni di ricorreza hanno soluzione $\Theta(\log n)$ (si può usare il Teorema Master), da cui segue che $T(n) = \Theta(\log n)$.

(*f*) Definiamo $S(n) := \frac{T(n)}{n}$. Abbiamo quindi: $n\,S(n) = \sqrt{n}\sqrt{n}S(\sqrt{n}) + n$, ovvero $S(n) = S(\sqrt{n}) + 1$, che si può risolvere facendo il cambio di variabile $n = 2^m$ e ottenendo $S(n) = \Theta(\log\log n)$. Quindi $T(n) = \Theta(n\log\log n)$.

Esercizio 2 Un array A di n elementi è detto *unimodale* se consiste di una sequenza crescente seguita da una sequenza decrescente o più precisamente se esiste un indice $m \in \{1, 2, \ldots, n\}$ tale che:

- $A[i] < A[i+1]$, per ogni $1 \leq i < m$, e
- $A[i] > A[i+1]$, per ogni $m \leq i < n$.

In particolare $A[m]$ è il massimo elemento ed è l'unico che è circondato da due elementi più piccoli ($A[m-1]$ e $A[m+1]$).

(a) Si progetti un algorimo con complessità temporale $o(n)$ che, dato un array unimodale A, restituisce l'indice dell'elemento massimo.

(b) Si progetti un algorimo con complessità temporale $o(n \log n)$ che ordina (in ordine crescente) un array unimodale A.

Soluzione esercizio 2
Punto (a).

L'idea è quella di trovare l'indice del massimo usando l'idea della ricerca binaria. Infatti, dato un certo indice h, è possibile capire in tempo costante se $A[h]$ è il massimo guardando i valori vicini (di indice $h-1$ e $h+1$). Se $A[h]$ non è il massimo, è possibile capire in tempo costante se h si trova nella parte crescente di A (e quindi la ricerca va eseguita solo a destra di h) o nella parte decrescente di A (e quindi la ricerca del massimo può essere eseguita alla sola sinistra di h). Di seguito riportiamo lo pseudocodice dell'algoritmo `Max`(A), che usa una procedura ricorsiva ausiliaria `MaxRic`(A, i, j) che trova l'indice del valore massimo nella porzione di array compresa fra gli indici i e j. La complessità temporale dell'algoritmo è chiaramente $O(\log n)$.

Algorithm 1: `Max`(A)

```
n = size of A ;
if A[1] > A[2] then
    return 1
if A[n] > A[n − 1] then
    return n
return MaxRic(A, 2, n − 1)
```

Dove, la procedura ricorsiva ausiliaria è:

Algorithm 2: `MaxRic`(A, i, j)

```
if i > j then
    return −1
m = ⌊(i+j)/2⌋ ;
if A[m] > A[m − 1] e A[m] > A[m + 1] then
    return m
if  A[m] < A[m + 1] then
    return MaxRic(A, m + 1, j)
else
    return MaxRic(A, i, m − 1)
```

Punto (b).

Mostriamo come è possibile ordinare un array unimodale di n elementi in tempo $O(n)$. L'idea è quella di trovare l'indice m dell'elemento massimo e sfruttare il fatto che le due porzioni $A[1; m]$ e $A[m + 1; n]$ sono già ordinate. Sappiamo già che possiamo fondere due sequenze ordinate in tempo lineare nella loro dimensione usando la procedura `Merge`. Qui l'unico dettaglio a cui dobbiamo fare attenzione è che le due porzioni di A sono ordinate in modo diverso, ovvero, la prima in modo crescente, mentre la seconda in modo decrescente.

Di seguito riportiamo lo pseudocodice dell'algoritmo che usa al suo interno due procedure ausiliarie. Una è la procedura (presentata a lezione) `Merge`(A, i_1, f_1, f_2) che fonde le due porzioni di $A[i_1, f_1]$ e $A[f_1 + 1, f_2]$ (e che si aspetta ordinate in modo crescente al momento della chiamata). L'altra è `Inverti`(A, i, f), che inverte gli elementi della porzione di array $A[i; f]$ in tempo lineare nel numero di elementi da invertire e usa memoria ausiliaria costante. Tale procedura è semplice ed è lasciata come esercizio allo studente.

Algorithm 3: `OrdinaUnimodale`(A)

```
n = size of A ;
m = Max(A);
if m ≠ −1 and m ≠ n then
    Inverti(A, m, n);
    if m ≠ 1 then
        Merge(A, 1, m − 1, n);
```

Esercizio 3 L'elemento *mediano* di un insieme S di m elementi (distinti) è quell'elemento che ha esattamente $\lfloor m/2 \rfloor$ elementi minori in S. Per esempio, l'insieme $S = \{1, 4, 6, 8, 9, 12\}$ ha mediano 8. Siano dati due insiemi A e B di n elementi ciascuno, rappresentati come sequenze ordinate memorizzate negli array $S_A[1 : n]$ e $S_B[1 : n]$. Progettare un algoritmo che trovi il mediano di $A \cup B$ in tempo $O(n)$ e che usi memoria ausiliaria $O(1)$. Si faccia attenzione al fatto che i due insiemi possono avere intersezione non vuota.

Soluzione esercizio 3

L'idea fondamentale è che, dati i due insieme A e B, è possibile scandire in tempo lineare e usando solo memoria aggiuntiva costante gli elementi di $A \cup B$ in ordine crescente di valore. Si può fare nel seguente modo. Poiché i due vettori S_A e S_B sono ordinati, si può usare l'idea utilizzata nella procedura `Merge`: si posizionano due indici i e j all'inizio di S_A e S_B e si scandiscono (in ordine crescente) gli elementi di $A \cup B$ estraendo ripetutamente il minimo fra $S_A[i]$ e $S_B[j]$ e incrementando opportunamente gli indici. Usando questa idea, è possibile individuare il mediamo di $A \cup B$ eseguendo due scansioni lineari, la prima che calcola $N := |A \cup B|$, e la seconda che si ferma quando raggiunge il $(\lfloor N/2 \rfloor + 1)$-esimo minimo di $A \cup B$. Un possibile pseudocodice è fornito di seguito.

Algorithm 4: `Mediano`(S_A, S_B, n)

```
i = 1, j = 1; N = 2n;
while i ≤ n and j ≤ n do
    if S_A[i] < S_B[j] then
        i = i + 1;
    else
        if S_A[i] > S_B[j] then
            j = j + 1;
        else
            N = N − 1;
            i = i + 1;
            j = j + 1;
m = ⌊N/2⌋ + 1; k = 1; i = 1; j = 1;
while k ≠ m do
    if S_A[i] < S_B[j] then
        i = i + 1;
    else
        if S_A[i] > S_B[j] then
            j = j + 1;
        else
            i = i + 1;
            j = j + 1;
    k = k + 1;
if i > n then
    return S_B[1];
if j > n then
    return S_A[1];
return min{S_A[i], S_B[j]};
```

Esercizio 4 (*sui modelli di calcolo e i numeri di Fibonacci*)
Nell'analizzare la complessità temporale degli algoritmi per calcolare l'n-esimo numero di Fibonacci, abbiamo stimato il numero (asintotico) di linee di codice eseguite dai vari algoritmi in funzione del valore n preso in ingresso dall'algoritmo stesso. Ora che conosciamo cosa si intende per modello di calcolo, ci rendiamo conto che le complessità temporali sono state derivate assumendo un modello di calcolo RAM a *costi uniformi*. Sfortunatamente, tale modello non è particolarmente adatto quando si ha a che fare con algoritmi che, come nel caso del calcolo dei numeri di Fibonacci, devono gestire interi il cui valore cresce esponenzialmente all'aumentare del valore di input.

Questo esercizio vi chiede di porre rimedio a questa imprecisione. In particolare, si chiede di derivare nuovamente un upper bound asintotico al tempo di esecuzione degli algoritmi `Fibonacci2`, `Fibonacci3`, e `Fibonacci6`, nei due seguenti modelli di calcolo (di fatto più ragionevoli per il problema in esame):

- Il primo modello è quello RAM a *costi logaritmici*. In tale modello, ogni operazione (di lettura, scrittura, somma e prodotto) che coinvolge un operando di valore x ha

un costo (temporale) pari a $\Theta(\log x)$, o equivalentemente ha un costo proporzionale al numero di bit necessari per rappresentare x.

- In alcuni casi il modello precedente risulta essere ancora impreciso, perché non distingue fra i costi delle varie operazioni. Di fatto, l'operazione di prodotto fra due interi è un'operazione più costosa rispetto all'operazione di somma. Mentre è possibile sommare due numeri di k bit in tempo $O(k)$[1], non è noto un algoritmo con la stessa complessità per moltiplicare due numeri di k bit. Si assuma che il prodotto fra due interi di k bit possa essere calcolato in tempo $O(k^{\log_2 3})$. Come cambiano le complessità temporali dei vari algoritmi?

In entrambi i modelli considerati, si può dire ancora che `Fibonacci3` è esponenzialmente più veloce di `Fibonacci2`? E `Fibonacci6` di `Fibonacci3`?

Soluzione esercizio 4
L'osservazione cruciale è che il numero di Fibonacci di ordine n vale circa ϕ^n, quindi il numero di bit necessari per rappresentarlo è (circa) $\log_2 \phi^n = n \log_2 \phi = \Theta(n)$. Quindi nel modello RAM a costi logaritmici ogni operazione costa $O(n)$. A questo punto è possibile derivare nuovamente le complessità dei tre algoritmi.

La complessità temporale di `Fibonacci2` diventa $O(\phi^n n)$, perché l'albero della ricorsione ha $\Theta(\phi^n)$ nodi ognuno dei quali costa al più $O(n)$. `Fibonacci3`, invece richiede tempo $O(n^2)$, perché effettua $O(n)$ somme ognuna delle quali costa $O(n)$. Per quanto riguarda `Fibonacci6`, l'equazione di ricorrenza che descrive la sua complessità temporale è $T(n) = T(n/2) + O(n)$ che, risolta con il Teorema Master, ha soluzione $T(n) = O(n)$.

Nel secondo modello di calcolo, che distingue i costi di operazioni di somma e di prodotto, i primi due algoritmi continuano ad avere complessità $O(\phi^n n)$ e $O(n^2)$, mentre `Fibonacci6` ha complessità $O(n^{\log_2 3})$. Infatti, ogni chiamata ricorsiva di tale algoritmo esegue un numero costante di operazioni di somma e prodotto, e gli operandi coinvolti hanno un numero di bit pari a $O(n)$. La sua equazione di ricorrenza è quindi $T(n) = T(n/2) + O(n^{\log_2 3})$ che ha appunto soluzione $O(n^{\log_2 3})$.

Si noti a questo punto che in entrambi i modelli di calcolo, l'algoritmo `Fibonacci2` ha una complessità esponenzialmente più grande degli altri due, mentre `Fibonacci6` non risulta esponenzialmente migliore di `Fibonacci3` (come avveniva nel modello RAM a costi uniformi). Un'ultima osservazione: `Fibonacci6` è un algoritmo ottimo nel modello RAM a costi logaritmici, perché anche solo scrivere l'output (che ha $\Omega(n)$ bit) richiede appunto tempo $\Omega(n)$.

[1] Abbiamo imparato come si fa alle scuole elementari: si allineano i due numeri e si sommano cifra-cifra dalla meno significativa alla più significativa.