

Corso di Algoritmi e Strutture Dati—Informatica per il Management

Prima prova parziale, 28/01/2010

Note: Il tempo a disposizione per lo svolgimento della prova è di **due ore e 30 minuti**. Durante la prova non è possibile consultare libri o appunti, né interagire in alcun modo con gli altri studenti. Chi copia in tutto o in parte il compito verrà escluso dalla prova; l'esclusione potrà avvenire anche dopo il termine della prova stessa, e riguarderà entrambi gli “estremi” della copiatura. È consentita la consultazione del formulario allegato a questo compito. Ove si richieda la descrizione di un algoritmo, è possibile darla mediante pseudo-codice oppure mediante codice Java (in quest'ultimo caso verranno ignorati eventuali errori minori di sintassi che non compromettano la comprensione dell'algoritmo né la sua correttezza). Tutte le risposte devono essere adeguatamente motivate.

Esercizio 1.

Si consideri il seguente problema: dato un array $A[1..n]$ di $n > 0$ elementi tra di loro confrontabili, restituire il valore minimo in A , ossia restituire un valore v presente in A tale che per ogni $i=1..n$ risulti $v \leq A[i]$.

1. Scrivere un algoritmo *ricorsivo* di tipo *divide et impera* per risolvere il problema di cui sopra; non è consentito usare variabili globali. [punti 5]
2. Calcolare la complessità asintotica dell'algoritmo proposto, motivando la risposta; [punti 5]
3. Dare un limite inferiore alla complessità del problema, nell'ipotesi in cui l'algoritmo risolutivo si basi solo su confronti. Giustificare la risposta. [punti 3]

Soluzione 1.

L'algoritmo può essere descritto come segue:

```
Algoritmo minDivideImpera( Array A[1..n], int i, int f ) : Elem
  if (i==f) then
    return A[i];
  else
    m = (f+i)/2; // si assume arrotondamento per difetto
    v1 = minDivideImpera(A, i, m);
    v2 = minDivideImpera(A, m+1, f);
    if (v1 < v2) then
      return v1;
    else
      return v2;
    endif
  endif
```

I parametri i e f indicano rispettivamente l'indice del primo e dell'ultimo elemento (estremi inclusi) in cui effettuare la ricerca. La prima invocazione dell'algoritmo quindi sarà $\text{minDivideImpera}(A, 1, n)$, assumendo che l'array A abbia n elementi indicizzati a partire da 1.

La complessità asintotica si ricava risolvendo la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} c_1, & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + c_2, & \text{se } n > 1 \end{cases}$$

Applicando il Master Theorem (caso 1), la soluzione è $T(n) = O(n)$.

Per ricavare un limite inferiore, è sufficiente considerare che un qualsiasi algoritmo basato su confronti deve necessariamente considerare ogni elemento dell'array A almeno una volta (altrimenti se viene saltato un elemento, questo potrebbe essere il minimo e l'algoritmo non sarebbe corretto). Quindi il limite inferiore alla complessità del problema è $\Omega(n)$.

Esercizio 2.

Scrivere un algoritmo efficiente per risolvere il seguente problema: dato un array $A[1..n]$ di $n > 0$ valori reali, restituire *true* se l'array A rappresenta un min-heap binario, *false* altrimenti. [punti 5]

Calcolare la complessità nel caso pessimo e nel caso ottimo dell'algoritmo proposto, motivando le risposte. [punti 5]

Soluzione 2.

L'algoritmo può essere espresso come segue:

```
Algoritmo isMinHeap( Array A[1..n] )
  for i=1 to n do
    if ( 2*i <= n && A[i] > A[2*i] ) then
      return false;
    endif
    if ( 2*i+1 <= n && A[i] > A[2*i+1] ) then
      return false;
    endif
  endfor
  return true;
```

Si noti i controlli ($2*i \leq n$) e ($2*i+1 \leq n$): tali controlli sono necessari per essere sicuri che i figli dell'elemento $A[i]$ (rispettivamente $A[2*i]$ e $A[2*i+1]$) siano presenti nell'array. Se tali controlli sono omessi, l'algoritmo è sbagliato in quanto può causare l'indicizzazione di un array al di fuori dei limiti.

Il caso pessimo si ha quando il ciclo “for” viene eseguito interamente, ossia quando l'array $A[1..n]$ effettivamente rappresenta un min-heap. In questo caso il costo è $O(n)$.

Il caso ottimo si verifica quando la radice $A[1]$ risulta maggiore di uno dei due figli ($A[2]$ o $A[3]$, se esistono). In questo caso il ciclo “for” esce alla prima iterazione restituendo *false*, e il costo risulta $O(1)$.

Esercizio 3.

Considerare la seguente variante dell'algoritmo selectionSort():

```
Algoritmo mioSelectionSort(array A[1..n])
  for k=0 to n-2 do
    m=k+1;
    for j=k+2 to n do
      if (A[j] <= A[m]) then
        m=j;
      endif
    endfor
    scambia A[k+1] con A[m];
  endfor
```

Ricordiamo che un algoritmo di ordinamento è *stabile* se preserva l'ordinamento relativo degli elementi uguali presenti nell'array da ordinare.

L'algoritmo mioSelectionSort() è un algoritmo di ordinamento stabile? In caso negativo, mostrare come modificarlo per farlo diventare stabile. Motivare le risposte. [punti 5]

Soluzione 3.

L'algoritmo mioSelectionSort() è quasi identico all'algoritmo SelectionSort() descritto nel libro di testo, con la differenza che la condizione è stata scritta come “if ($A[j] \leq A[m]$)” (cioè usando l'operatore minore o uguale).

Questo fa sì che mioSelectionSort non sia un algoritmo di ordinamento stabile, perché in caso di elementi uguali mette in prima posizione l'ultimo elemento trovato. Per renderlo stabile è sufficiente riscrivere l'if come “if (A[j] < A[m])”

Esercizio 4.

Dato un array A[1..n] di $n \geq 3$ elementi confrontabili, tutti distinti, descrivere un algoritmo che in tempo $O(1)$ determina un elemento qualsiasi dell'array che non sia né il minimo né il massimo. [punti 4]

A cosa può servire un algoritmo del genere? Ovvero, esiste qualche problema che abbiamo affrontato nel corso la cui soluzione potrebbe beneficiare da questo algoritmo? [punti 1]

Soluzione 4.

Per risolvere il problema basta considerare i primi tre elementi di A, ossia A[1], A[2] e A[3]. Si cercano il massimo e il minimo tra questi tre, e si restituisce quello tra di loro che non è né massimo né minimo. Un tale elemento esiste sempre, perché per ipotesi tutti gli elementi sono distinti.

```
Algoritmo noMinMax( Array A[1..n] )
  if ( A[1] < A[2] ) then
    min12 = A[1];
    max12 = A[2];
  else
    min12 = A[2];
    max12 = A[1];
  endif
  if ( A[3] < min12 ) then
    return min12;
  elseif ( A[3] < max12 ) then
    return A[3];
  else
    return max12;
  endif
```

Si osservi che le variabili min12 e max12 contengono rispettivamente il valore minimo e massimo tra A[1] e A[2].

Un simile algoritmo può essere utilizzato per scegliere il pivot (in modo deterministico) in modo da evitare il caso pessimo nell'algoritmo quickSort e quickSelect, assumendo che tutti i valori in input siano distinti.