Dati e Algoritmi (Pietracaprina)

Esercizi svolti sull'Ordinamento

Problema 1 *Si consideri l'esecuzione di* `MergeSort` *su un'istanza di taglia n, con $2^d < n < 2^{d+1}$ per un qualche intero $d \geq 0$. Dimostrare che:*

a. *Per $0 \leq i \leq d$, il livello i dell'albero della ricorsione ha 2^i nodi corrispondenti a invocazioni dell'algoritmo su istanze con taglia in $[2^d/2^i, 2^{d+1}/2^i]$, e la somma delle taglie di tali istanze è n.*

b. *Il livello $d+1$ ha $x \leq n$ nodi corrispondenti a invocazioni dell'algoritmo su istanze di taglia 1.*

c. *Dai due punti precedenti, dimostrare che per una tale istanza,* `MergeSort` *richiede $O(n \log n)$ operazioni.*

Soluzione.

a. Procediamo per induzione su i. Per quanto riguarda la base dell'induzione ($i = 0$), la proprietà è vera dato che c'è un solo nodo al livello 0, associato all'istanza iniziale di taglia n, e per ipotesi $2^d < n < 2^{d+1}$. Si supponga, come ipotesi induttiva, che la proprietà valga sino al livello i, per un qualche $i \in [0, d)$ e si consideri il livello $i+1$. Ogni nodo al livello i che, per ipotesi induttiva, corrisponde a un'invocazione su un'istanza di taglia $X \in [2^d/2^i, 2^{d+1}/2^i]$, e quindi $X > 1$, avrà due figli al livello $i+1$ corrispondenti a invocazioni su istanze di taglie $\lfloor X/2 \rfloor$ e $\lceil X/2 \rceil$, rispettivamente, la cui somma è $|X|$. Se X è pari, si ha $\lfloor X/2 \rfloor = \lceil X/2 \rceil = X/2$, e dall'ipotesi induttiva deduciamo che $X/2 \in [2^d/2^{i+1}, 2^{d+1}/2^{i+1}]$. Se X è dispari, allora $X \in [(2^d/2^i) + 1, (2^{d+1}/2^i) - 1]$ ed è facile vedere che $2^d/2^{i+1} \leq \lfloor X/2 \rfloor < \lceil X/2 \rceil \leq 2^{d+1}/2^{i+1}$. Ne consegue, che al livello $i+1$ ci saranno 2^{i+1} nodi corrispondenti a invocazioni su istanze con taglie in $[2^d/2^{i+1}, 2^{d+1}/2^{i+1}]$, che sommano a n.

b. Dal punto precedente sappiamo che i nodi del livello d dell'albero della ricorsione corrispondono a invocazioni su istanze con taglie in $[2^d/2^d, 2^{d+1}/2^d] = [1, 2]$ che sommano a n. Di conseguenza, i loro figli (se esistono) corrispondono a invocazioni su istanze di taglia 1 e non possono essere più di n, dato che una stessa chiave di input non può far parte di due o più istanze a un certo livello dell'albero della ricorsione.

c. Dai due punti precedenti si ricava che l'albero della ricorsione ha $d + 2 \leq (\log_2 n) + 2$ livelli, e che le istanze associate ai nodi di ciascun livello hanno una taglia aggregata $\leq n$. Dato che un nodo contribuisce un costo lineare nella taglia dell'istanza a esso associata (che include il costo della iniziale suddivisione in due metà e il costo del merge), si ha che ciascun livello contribuisce un costo aggregato $O(n)$. Quindi la complessità è $O(dn) = O(n \log n)$.

□

Problema 2 *Sia $k = 2^d$, per un qualche intero $d > 0$, e siano $S_1, S_2, \ldots, S_k$ sequenze ordinate di taglia m ciascuna. Si vuole progettare un algoritmo efficiente per fondere le sequenze in un'unica sequenza ordinata di taglia $n = k \cdot m$.*

a. *Analizzare la complessità dell'algoritmo banale che fonde S_1 con S_2, poi $S_1 \cup S_2$ con S_3, poi $S_1 \cup S_2 \cup S_3$ con S_4 ecc.*

b. *Progettare e analizzare un algoritmo più efficiente di quello banale.*

Soluzione.

a. La strategia banale suggerita consiste di $k-1$ iterazioni dove nell'Iterazione i, per $2 \leq i \leq k$, viene fatto il merge della sequenza ordinata contenente le chiavi di $S_1 \cup S_2 \cup \cdots \cup S_{i-1}$ con la sequenza S_i. Dalla linearità dell'algoritmo di Merge, deduciamo che il numero di operazioni richiesto dall'Iterazione i è $\Theta(im)$, dato che $|S_1 \cup S_2 \cup \cdots \cup S_i| = im$. Di conseguenza, la complessità dell'algoritmo banale risulta essere:

$$\Theta\left(\sum_{i=2}^{k} im\right) = \Theta\left(k^2 m\right) = \Theta(kn).$$

b. Esiste invece una strategia più efficiente che consiste nel fondere le sequenze iniziali a coppie, poi le fusioni delle varie coppie ancora a coppie, ecc. Si ricordi che `Merge`(X, Y, Z) è l'algoritmo visto a lezione per fondere due sequenze ordinate X e Y mettendo la loro fusione ordinata in Z. L'algoritmo è il seguente:

Algoritmo `kMerge`$(S_1, S_2, \ldots, S_k)$

Input: `sequenze ordinate` $S_1, S_2, \ldots, S_k$ `di taglia` m `ciascuna`
Output: `sequenza` $\cup_{i=1}^{k} S_i$ `ordinata`
`Sia` $\mathtt{S}_i^0 = \mathtt{S}_i$`, per` $1 \leq i \leq k$`;`
for $j \longleftarrow 1$ **to** d **do**
  for $i \longleftarrow 1$ **to** 2^{d-j} **do**
    $\mathtt{S}_i^j \leftarrow$ `sequenza vuota;`
    `Merge(`$\mathtt{S}_{2i-1}^{j-1}$`,`$\mathtt{S}_{2i}^{j-1}$`,` $\mathtt{S}_i^j$`)`
return $\mathtt{S}_1^d$

È facile vedere che ciascuna iterazione del ciclo **for** esterno richiede tempo $\Theta(n)$. Quindi la complessità totale risulta essere $\Theta(nd) = \Theta(n \log k)$. Si noti che se $k = n$ e $m = 1$ l'algoritmo è equivalente a `MergeSort`. In generale, la complessità non è mai peggiore di quella `MergeSort`, ed è anzi migliore se k è piccolo.

□

Problema 3 *Siano* $S_1 = S_1[0]S_1[1]\ldots S_1[n_1-1]$ *e* $S_2 = S_2[0]S_2[1]\ldots S_2[n_2-1]$ *due sequenze ordinate di chiavi intere. In ogni sequenza le chiavi sono distinte, ma una stessa chiave può comparire sia in* S_1 *che in* S_2*. Progettare una modifica dell'algoritmo* `Merge` *che restituisca il numero di chiavi in* $S_1 \cap S_2$*. L'algoritmo deve usare spazio aggiuntivo costante oltre alle due sequenze di input, e avere complessità* $O(n_1 + n_2)$*.*

Soluzione. La modifica di `Merge` si intuisce con la seguente osservazione. Si supponga di eseguire `Merge`(S_1, S_2, S) e si consideri il primo ciclo while dell'algoritmo. È facile convincersi che se una chiave k compare sia in S_1 che in S_2, ovvero $k = S_1[i] = S_2[j]$ per una qualche coppia di indici i e j, allora ci deve essere un'iterazione di quel ciclo in cui si confrontano $S_1[i]$ ed $S_2[j]$. In quel momento si può aggiornare una variabile che conta il numero di chiavi comuni alle due sequenze. Lo pseudocodice è il seguente.

Algoritmo `CountIntersection(`S_1`,`S_2`)`

Input: `sequenze` S_1, S_2 `ordinate, ciascuna con chiavi distinte`
Output: $|S_1 \cap S_2|$
$i \leftarrow 0;\ j \leftarrow 0;\ C \leftarrow 0\ ;$
while `((`$i < |S_1|$`) AND (`$j < |S_2|$`))` **do**
  if `(`$S_1[i] = S_2[j]$`)` **then** $\{C++;\ i++;\ j++\}$;
  else
    if `(`$S_1[i] < S_2[j]$`)` **then** $i++$;
    else $j++$;
return C

Oltre alle due sequenze di input, l'algoritmo usa $O(1)$ variabili intere che richiedono quindi spazio costante. Analizziamo complessità e correttezza, per quanto non richieste dall'esercizio. La complessità chiaramente $\Theta(|S_1| + |S_2|)$ in quanto ogni iterazione esegue un numero costante di operazioni e incrementa almeno uno dei due indici i e j, i quali possono crescere, rispettivamente, sino a $|S_1|$ e $|S_2|$. La prova di correttezza dell'algoritmo si basa sul seguente invariante che vale alla fine di ciascuna iterazione dei while:

- $C = |S_1[0 \div i-1] \cap S_2[0 \div j-1]|$;
- Le chiavi in $S_1[0 \div i-1] \cup S_2[0 \div j-1]$ sono le $i \cdot j$ chiavi più piccole tra tutte le chiavi in $S_1 \cup S_2$.

Completare la prova di correttezza per esercizio. □

Problema 4 (Esercizio C-13.46 [GTG14]). *Dati due insiemi A e B, ciascuno costituito da n chiavi distinte provenienti da un universo ordinato e rappresentato tramite una sequenza ordinata. Progettare un algoritmo efficiente per calcolare $A \oplus B$, ovvero l'insieme di chiavi che sono in A o in B ma non in entrambi, analizzandone correttezza e compessità.*

Soluzione. L'idea è di fare il merge di A e B evitando di inserire nell'insieme di output le chiavi di $A \cap B$. Sia x una chiave in $A \cap B$. Eseguendo il merge con lo stesso algoritmo usato all'interno di `MergeSort`, arriverà un momento in cui la scansione di A e B è ferma su x e risulta quindi facile identificare x come chiave comune e ometterla dall'output. Lo pseudocodice dell'algoritmo è il seguente:

Algoritmo `A-oplus-B(`A`,`B`)`

Input: `Insiemi` A `e` B `di` n `chiavi ciascuno`
Output: $A \oplus B$
$S \leftarrow \emptyset;\ i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0;$
while `((`$i < n$`) AND (`$j < n$`))` **do**
  if `(`$A[i] = B[j]$`)` **then** $\{i++;\ j++\}$;
  else
    if `(`$A[i] < B[j]$`)` **then** $S[k++] \leftarrow A[i++]$;
    else $S[k++] \leftarrow B[j++]$;
while `(`$i < n$`)` **do** $S[k++] \leftarrow A[i++]$;
while `(`$j < n$`)` **do** $S[k++] \leftarrow B[j++]$;
return S

Per quanto riguarda la correttezza analizziamo due casi.

CASO 1. Sia x una chiave che appartiene a uno dei due insiemi ma non a entrambi. È immediato verificare che arriverà un momento in cui x viene ispezionata dall'algoritmo e inserita in S.

CASO 2. Sia $x \in A \cap B$, ovvero $x = A[\ell] = B[r]$, per una qualche coppia di indici $0 \leq \ell, r < n$. Consideriamo il primo istante in cui si ha $i = \ell$ o $j = r$. In particolare, avremo che in questo istante vale $i = \ell$ e $j < r$, oppure $i < \ell$ e $j = r$, oppure $i = \ell$ e $j = r$. In ogni caso, deve arrivare un momento in cui $i = \ell$ e $j = r$ e x non è stata inserita in S. In questo momento però, l'algoritmo si accorge che $x \in A \cap B$ e la scarta.

La complessità dell'algoritmo è $O(n)$ e l'analisi è identica a quella della fase di merge in `MergeSort`. □

Problema 5 (Esercizio C-13.42 [GTG14]). *Sia S una sequenza di n elementi distinti sui quali è definito un ordine totale. Si ricordi che una* inversione *in S è una coppia di elementi $S[i], S[j]$ tali che $j < i$ ma $S[j] > S[i]$. Descrivere un algoritmo che determina il numero di inversioni in S in tempo $O(n \log n)$.* (**Suggerimento:** adattare la strategia di `MergeSort`.)

Soluzione. L'algoritmo si basa sulla seguente idea. Supponiamo di aver suddiviso la sequenza S in due sottosequenze S_1 e S_2. Sia m_1 il numero di inversioni dentro S_1, m_2 il numero di inversioni dentro S_2, e m_{12} il numero di inversioni tra elementi di S_1 e di S_2. Si ha quindi che il numero m di inversioni in S è uguale a $m_1 + m_2 + m_{12}$. Detto n_j il numero di elementi di S_2 più piccoli di $S_1[j]$, per $0 \leq j < |S_1|$, si vede facilmente che

$$m_{12} = \sum_{j=0}^{|S_1|-1} n_j.$$

Possiamo quindi usare una modifica di `MergeSort` che, oltre a ordinare la sequenza S, produce in output anche il numero di inversioni. Ora, m_1 e m_2 sono ottenute dalle chiamate ricorsive, mentre m_{12} può essere determinato durante il merge delle due sottosequenze S_1 e S_2, operando come segue. Si supponga di usare l'indice j per S_1 e l'indice i per S_2. Nel momento in cui $S_1[j]$ viene aggiunto alla sequenza ordinata S, siamo in grado di conoscere il valore n_j, che coincide esattamente con il numero di elementi di S_2 già inseriti in S, e quindi possiamo aggiungerlo al conteggio delle inversioni. Lo pseudocodice è il seguente. (Si noti che rispetto al `MergeSort` visto a lezione qui, per comodità di scrittura, la fase di merge è inserita nel codice invece di essere eseguita invocando un apposito algoritmo.)

Algoritmo `Sort-and-Inversions(`S`)`

Input: `Sequenza` S `di` n `elementi`
Output: `Numero` m `di inversioni in` S`, lasciando la sequenza` S `ordinata`

```
n ← S.size ;
if (n ≤ 1) then return S;
for i ← 0 to ÷⌈n/2⌉ − 1 do  S1[i] ← S[i];
for i ← ⌈n/2⌉ to n − 1 do  S2[i − ⌈n/2⌉] ← S[i];
m1 ← Sort-and-Inversions(S1);
m2 ← Sort-and-Inversions(S2);
m ← m1 + m2; j ← 0; i ← 0; k ← 0;
while ((j < S1.size()) AND (i < S2.size())) do
    if (S1[j] < S2[i]) then
        m ← m + i;                                   /* i = nj */
        S[k++] ← S1[j++]
    else S[k++] ← S2[i++];
while (j < S1.size()) do
    m ← m + S2.size();                          /* S2.size() = nj */
    S[k++] ← S1[j++]
while (i < S2.size()) do S[k++] ← S2[i++];
return m
```

La struttura dell'algoritmo è molto simile a quella di `MergeSort`, ed è facile verificare che la sua complessità rimane $\Theta(n \log n)$, come quella di `MergeSort`. □

Problema 6 *Dimostrare che alla fine di ciascuna iterazione del ciclo* **while** *esterno dell'algoritmo* `Partition`(S, a, b)*, usato in* `QuickSortInPlace` *vale il seguente invariante:*

- $p = S[b]$*;*
- $S[i] \leq p$ *per ogni* $a \leq i < \ell$*;*
- $S[i] \geq p$ *per ogni* $r < i \leq b$*;*
- $\ell \leq r + 1$*.*

Soluzione. È immediato vedere che le inizializzazioni eseguite prima del **while**, rendono l'invariante vero all'inizio del ciclo. Se l'invariante vale alla fine di una generica iterazione, è altrettanto immediato verificare che i due **while** interni e lo swap mantengono l'invariante vero alla fine della successiva iterazione. □

Problema 7 *Progettare un algoritmo che ordini una sequenza di* n *bit* $S = S[0]S[1] \ldots S[n-1]$ *in tempo* $\Theta(n)$*. L'algoritmo deve essere in-place e utilizzare un solo ciclo.*

Soluzione. Si può adattare l'algoritmo `Partition` usato in `QuickSortInPlace` in modo da separare gli 0 dagli 1. In particolare si può scrivere un ciclo che usa due indici ℓ e r per esaminare la sequenza rispettivamente da sinistra e da destra, lasciando gli 0 alla sinistra di ℓ e gli 1 alla destra di r. In una generica iterazione del ciclo, quando si è posizionati su $S[\ell]$

e su $S[r]$, si opera nel modo seguente: se $S[\ell] = 0$ si incrementa ℓ, se $S[r] = 1$ si decrementa r, e se $S[\ell] = 1$ e $S[r] = 0$ si scambiano i due bit. Lo pseudocodice è il seguente.

Algoritmo `SortBits(`S`)`

Input: `sequenza` $S[0], S[1], \ldots, S[n-1]$ `di` n `bit`
Output: `sequenza` S `ordinata`
$\ell \leftarrow 0$; $r \leftarrow n-1$;
while $(\ell \leq r)$ **do**
  if $((S[\ell] = 0)$ `AND` $(S[r] = 0))$ **then** {ℓ`++; continue`};
  if $((S[\ell] = 0)$ `AND` $(S[r] = 1))$ **then** {ℓ`++;`r`--; continue`};
  if $((S[\ell] = 1)$ `AND` $(S[r] = 0))$ **then** {`swap(`$S[\ell], S[r]$`); continue`};
  if $((S[\ell] = 1)$ `AND` $(S[r] = 1))$ **then** {r`--; continue`};
return S

La complessità dell'algoritmo è $\Theta(n)$ in base alle seguenti osservazioni:

- Ogni iterazione del **while** richiede $\Theta(1)$ operazioni;
- Ogni due iterazioni del **while** almeno uno dei due indici ℓ o r viene incrementato/decrementato;
- Il numero totale di incrementi/decrementi di ℓ e r è $\Theta(n)$.

□

Problema 8 *Svolgere i seguenti punti:*

a. Dire per quali istanze `QuickSortInPlace` *implementa fedelmente la strategia L-E-G descritta a lezione.*

b. Dire come si comporta `QuickSortInPlace` *su un'istanza con tutte chiavi uguali.*

c. (Esercizio C-13.33 [GTG14]). *Modificare* `QuickSortInPlace` *in modo che implementi fedelmente la strategia L-E-G* per tutte le possibili istanze*, e dire come si comporta la modifica su un'istanza con tutte chiavi uguali.*

Soluzione.

a. `QuickSortInPlace` implementa fedelmente la strategia L-E-G quando di ogni pivot scelto si ha una sola copia nella sequenza, e quindi, in particolare, quando le chiavi sono tutte distinte.

b. Su un'istanza con chiavi tutte uguali, `QuickSortInPlace` si comporta come nel caso di istanza ordinata: ha un albero della ricorsione di n livelli e richiede $\Theta(n^2)$ operazioni.

c. Si supponga di invocare l'algoritmo nel segmento $S[a \div b]$ della sequenza, per una data coppia di indici a, b con $0 \leq a < b \leq n-1$. L'idea è quella di modificare la fase di divide di `QuickSortInPlace` visto a lezione in modo da accumulare alla immediata sinistra dell'indice ℓ e alla immediata destra dell'indice r le chiavi uguali al pivot incontrate durante la scansione della sequenza. Per far questo si possono utilizzare 4 indici, ℓ, ℓ_1, r, e r_1 impostati inizialmente come segue:

$$\ell = \ell_1 = a, \quad r = b-1, \quad r_1 = b.$$

Detto $p = S[b]$ il pivot, alla fine di ciascuna iterazione del **while** esterno l'algoritmo mantiene il seguente invariante:

- $S[i] < p$ per ogni $a \leq i < \ell_1$;
- $S[i] = p$ per ogni $\ell_1 \leq i < \ell$;
- $S[i] = p$ per ogni $r < i \leq r_1$;
- $S[i] > p$ per ogni $r_1 < i \leq b$;
- $\ell \leq r + 1$.

Lo pseudocodice è il seguente:

Algoritmo `QuickSortInPlace-v2`(S, a, b)

Input: `sequenza` S `di` n `chiavi, indici` $0 \leq a, b < n$
Output: `sequenza` S `ordinata in senso crescente tra` $S[a]$ `e` $S[b]$
if $(a \geq b)$ **then return** S;
$p \leftarrow S[b]$;
$\ell \leftarrow a$; $\ell_1 \leftarrow \ell$; $r \leftarrow b - 1$; $r_1 \leftarrow b$;
while $(\ell \leq r)$ **do**
  while $((\ell \leq r)$ `AND` $(S[\ell] \leq p))$ **do**
    if $(S[\ell] < p)$ **then**
      `swap`$(S[\ell], S[\ell_1])$;
      $\ell++$; ℓ_1++
    else $\ell++$;
  while $((\ell \leq r)$ `AND` $(S[r] \geq p))$ **do**
    if $(S[r] > p)$ **then**
      `swap`$(S[r], S[r_1])$;
      $r--$; r_1--
    else $r--$;
  if $(\ell < r)$ **then** `swap`$(S[\ell], S[r])$;
`QuickSortInPlace-v2`$(S, a, \ell_1 - 1)$;
`QuickSortInPlace-v2`$(S, r_1 + 1, b)$;

L'analisi è praticamente identica a quella della versione vista a lezione (`QuickSortInPlace`). Si osservi che quando le chiavi sono tutte distinte `QuickSortInPlace-v2` essenzialmente coincide con `QuickSortInPlace`, mentre quando ci sono chiavi ripetute le prestazioni asintotiche non peggiorano e anzi possono migliorare notevolemente. In particolare, quanto tutte le chiavi sono uguali, l'algoritmo termina con la prima invocazione, eseguendo un numero lineare di operazioni.

□

Problema 9 (Esercizio C-13.48 [GTG14]). *Sia dato un insieme A di n viti e un insieme B di n dadi. Ogni vite va bene per un solo dado. Progettare un algoritmo per trovare tutte le coppie (vite,dado) giuste avendo a disposizione una primitiva che data una vite a e un dado b decide in tempo costante se a è* piccola, giusta, *o* grande *per b. Dare una stima in ordine di grandezza del numero di volte in cui la primitiva viene invocata.*

Soluzione. L'idea è quella di adattare la strategia di QuickSort. Si prende un dado b arbitrario e lo si confronta con tutte le viti, trovando la vite a giusta per b e separando le rimanenti viti tra quelle piccole per b e quelle grandi per b. Poi, si confronta la vite a con tutti i dadi (escluso b), separando i dadi piccoli per a da quelli grandi per a. Siano A_L e A_G gli insiemi delle viti rispettivamente piccole e grandi per b, e siano B_L e B_G gli insiemi dei dadi rispettivamente piccoli e grandi per a. L'algoritmo viene chiamato ricorsivamente su (A_L, B_L) e su (A_G, B_G). Un'analisi analoga a quella fatta per QuickSort mostra che al caso pessimo la primitva verrà invocata $\Theta(n^2)$ volte, ma scegliendo a caso il dado b che funge da pivot in ogni chiamata ricorsiva le invocazioni diventano $O(n \log n)$ con alta probabilità. □

Problema 10 *Sia S una sequenza di n chiavi distinte e non ordinate.*

a. *Progettare un algoritmo in-place che dato un intero $k > 0$ e due indici a e b tali che $0 \leq a < b < n$ e $b \geq a + k$, esegua la partizione delle chiavi comprese tra $S[a]$ e $S[b-k]$ intorno ai k pivot $S[b-k+1], S[b-k+2], \ldots, S[b]$, e restituisca k indici $\ell_1 < \ell_2 < \cdots < \ell_k$ che rappresentano le posizioni dei pivot dopo la partizione. In particolare, alla fine dell'esecuzione deve valere che $S[\ell_1] < S[\ell_2] < \cdots < S[\ell_k]$ sono i k pivot e che, per ogni indice j tale che $\ell_i < j < \ell_{i+1}$, $S[\ell_i] < S[j] < S[\ell_{i+1}]$ (assumendo $\ell_0 = a - 1$ e $\ell_{k+1} = b + 1$). Chiamare l'algoritmo:* `k-Partition`(S, a, b).

b. *Analizzare la complessità dell'algoritmo* `k-Partition`(S, a, b) *progettato nel punto precedente.*

Soluzione.

a. L'idea è di ordinare i k pivot tenendoli inizialmente alla fine del segmento di sequenza $S[a \div b]$, e poi invocare ripetutamente l'algoritmo `Partition` usato dall'algoritmo `QuickSortInPlace` visto a lezione per partizionare le chiavi in $S[a \div b]$ intorno a un pivot alla volta partendo dal più piccolo di essi.

Algoritmo `k-Partition`(S, a, b)

Input: `Sequenza` S `di` n `chiavi distinte, indici` a `e` b `tali che` $0 \leq a < b < n$ `e` $b \geq a + k$
Output: `Sottosequenza` $S[a \div b]$ `riorganizzata separando le chiavi intorno ai` k `pivot originariamente in` $S[b-k+1], S[b-k+2], \ldots, S[b]$`, i quali in output occupano, ordinati, le posizioni` $\ell_1 < \ell_2 < \cdots < \ell_k$

`Ordina in-place` $S[b-k+1], S[b-k+2], \ldots, S[b]$;
$\ell_0 \longleftarrow a - 1$;
for $i \longleftarrow 1$ **to** k **do** $\ell_i + 1 \leftarrow$ `Partition`$(S, \ell_{i-1}, b-k+i)$;
return $\ell_1, \ell_2, \ldots, \ell_k$

b. Sia $m = b - a + 1$, il numero di chiavi sulle quali è invocato l'algoritmo. La complessità di `k-Partition` è determinata dall'ordinamento iniziale dei k pivot e dalle k chiamate a `Partition`, ciascuna delle quali richiede un tempo al più lineare in m. Limitando superiormente il tempo di ordinamento dei k pivot con $O(k^2)$ (ottenibile ad esempio usando InsertionSort), e ricordando che $k < m$, abbiamo che

$$T_{\text{k-Partition}}(m, k) \in O\left(k^2 + km\right) = O(km).$$

□

Problema 11 (Esercizio C-13.39 [GTG14]). *Dato un array A di n interi nell'intervallo $[0, n^2 - 1]$, descrivere un metodo semplice per ordinare A in tempo $O(n)$.*

Soluzione. Si noti che utilizzando `BucketSort` la complessità risulterebbe $\Theta(n + n^2) = \Theta(n^2)$, mentre utilizzando un qualsiasi algoritmo basato su confronti la complessità risulterebbe $\Omega(n \log n)$. Dato che comunque l'intervallo di valori per le chiavi è limitato, possiamo applicare `RadixSort` sfruttando un'opportuna rappresentazione per le chiavi. Infatti, ciascun intero $k \in [0, n^2 - 1]$ può essere rappresentato *in base* n, tramite due cifre (c_1, c_0), dove $c_1, c_0 \in [0, n - 1]$ sono univocamente determinate e tali che $k = c_1 \cdot n + c_0$. Le due cifre sono ricavabili come segue:

$$c_i = \left\lfloor \frac{k}{n^i} \right\rfloor \bmod n, \quad i = 0, 1.$$

Applicando `RadixSort` a tale rappresentazione, si ordina A in tempo $O(n)$. □

Problema 12 *Si supponga di utilizzare* `RadixSort` *per ordinare un array A di n interi nell'intervallo $[0, M - 1]$, con $M > n$, vedendo ciascun intero rappresentanto in base N. Determinare la complessità in funzione dei parametri n, M, e N, e dire qual è la scelta migliore per N.*

Soluzione. In generale, per ordinare una sequenza S di n chiavi nel range di interi $[0, M-1]$, con $M > n$, conviene scegliere $N = n$ come base delle chiavi. Vediamo il motivo. Supponiamo di scegliere una base $N \leq M$ generica ($N > M$ non ha senso!). È facile vedere che una qualsiasi chiave $k \in [0, M - 1]$ può essere rappresentata con $d = \lceil \log_N M \rceil$ cifre in base N, ovvero $k = \sum_{i=0}^{d-1} c_i \cdot N^i$, dove

$$c_i = \left(\left\lfloor \frac{k}{N^i} \right\rfloor \bmod N \right) \in [0, N - 1], \qquad \text{per } 0 \leq i < d.$$

Applicando RadixSort con tale rappresentazione la complessità è

$$\Theta(d(n + N)) = \Theta((\log_N M)(n + N)) = \Theta\left(\frac{\log M}{\log N}(n + N) \right),$$

e tale formula è minimizzata scegliendo $N \in \Theta(n)$. □

Problema 13 *Sia S una sequenza di n chiavi distinte $\in [0, 10n]$. Progettare un algoritmo di complessità lineare che restituisca l'indice i della* mediana*, cioè tale che esistono esattamente $\lfloor n/2 \rfloor$ chiavi più piccole di $S[i]$ in S.*

Soluzione. Il seguente algoritmo risolve il problema utilizzando `BucketSort`

Algoritmo `MedianIndex`(S)

Input: `Sequenza` S `di` n `chiavi distinte` $\in [0, 10n]$
Output: `Indice` i `tale che` $S[i]$ `è la mediana di` S

`Crea una sequenza vuota` S'`;`
for $i \longleftarrow 0$ **to** $n - 1$ **do**
    `crea una entry` $e_i = (S[i], i)$`;`
    $S'[i] \leftarrow e_i$
`Ordina` S' `con BucketSort;`
return $S'[\lfloor n/2 \rfloor]$`.getValue()`

La complessità è dominata dall'esecuzione di `BucketSort` ed è quindi lineare in n. □

Problema 14 (Esercizio C-13.40 [GTG14]). *Siano $S_1, S_2, \ldots, S_k$ k sequenze non vuote di entry con chiavi nel range $[0, N-1]$, per un qualche $N \geq 2$. Descrivere un algoritmo che ordini separatamente tutte le sequenze in tempo $O(n+N)$, dove n è la somma delle taglie delle sequenze. (In output le sequenze devono rimanere distinte e quindi l'esercizio non chiede di ordinare l'unione delle sequenze, che sarebbe banale.)*

Soluzione. L'idea è di ricopiare le entry di tutte le sequenze in un'unica sequenza S trasformando ciascuna entry $(c, v) \in S_i$ in una entry $((i, c), v)$ dove la nuova chiave (i, c) è composta di due cifre: la cifra più significativa è l'indice $i \in [1, k]$ della sequenza di origine, mentre la cifra meno significativa è la chiave $c \in [0, N-1]$ originale. Si applica quindi RadixSort a S usando le nuove chiavi. Dopo l'ordinamento, le entry di ciascuna sequenza risulteranno contigue e ordinate in base alla loro chiave originale e sarà sufficiente estrarle da S. Lo pseudocodice è il seguente:

Algoritmo `kSort`$(S_1, S_2, \ldots, S_k)$

Input: `sequenze` $S_1, S_2, \ldots, S_k$ `di entry con chiave in` $[0, N-1]$
Output: `sequenze` $S_1, S_2, \ldots, S_k$ `ordinate`

$S \longleftarrow$ `sequenza vuota;` $\ell \longleftarrow 0$`;`
for $i \longleftarrow 1$ **to** k **do**
  for $j \longleftarrow 0$ **to** $|S_i| - 1$ **do**
    `Sia` $(c, v) = S_i[j]$`;`
    $S[\ell++] \longleftarrow ((i, c), v)$`;`
`RadixSort(S);`
for $i \longleftarrow 1$ **to** k **do** $\ell_i \longleftarrow 0$`;`
for $j \longleftarrow 0$ **to** $|S| - 1$ **do**
  `Sia` $((i, c), v) = S[j]$`;`
  $S_i[\ell_i++] \longleftarrow (c, v)$`;`

Per quanto riguarda la correttezza, è semplice vedere che alla fine dell'algoritmo le sequenze $S_1, S_2, \ldots, S_k$ risultano individualmente ordinate. Riguardo alla complessità, sia $N' = \max\{N, k+1\}$. RadixSort viene eseguito su una sequenza di n entry con chiavi formate da $d = 2$ cifre in $[0, N'-1]$, e richiede quindi tempo $O(d(n+N')) = O(n+N')$. Dato che le sequenze sono non vuote, deve valere che $k \leq n$ e, di conseguenza, $N' \in O(\max\{N, n\}) = O(n+N)$. Poiché gli altri passi dell'algoritmo richiedono tempo $O(n)$, la complessità totale è $O(n+N)$. □

Problema 15 *Sia S una sequenza di n entry con chiavi intere in $[0, N-1]$, e sia B un vettore ausiliario di taglia N tale che $B[k]$ è uguale al numero di entry in S con chiave $\leq k$, per ogni $0 \leq k < N$.*

a. Scrivere un ciclo `for` *che in tempo $\Theta(n)$ produca una sequenza S' con le entry di S ordinate, sfruttando le informazioni del vettore B. (Si noti che N, che non compare nella complessità, può essere anche $\gg n$.)*

b. Analizzare la correttezza del ciclo trovato al punto precedente tramite un opportuno invariante.

c. Dire se l'ordinamento ottenuto è stabile e, se non lo fosse, dire come modificare il ciclo scritto al punto (a) per renderlo tale.

Soluzione.

a. Il ciclo è il seguente:

```
S' ← sequenza vuota;
for i ⟵ n − 1 downto 0 do
    S'[B[S[i].getKey()] − 1] ← S[i];
    B[S[i].getKey()] − −
return S'
```

b. Alla fine di ogni iterazione i del ciclo **for** vale il seguente invariante:

- Tutte le entry $S[j]$ con $j \geq i$ sono state memorizzate nelle posizioni finali corrette di S'; entry con la stessa chiave vengono memorizzate in posizioni consecutive procedendo da destra verso sinistra.
- $B[k]$ è uguale al numero di entry con chiave $\leq k$ in S meno il numero di entry con chiave esattamente uguale a k già incontrate, ovvero entry appartenenti al suffisso $S[i \div n-1]$. Di conseguenza, $B[k]-1$ rappresenta la posizione che dovrà occupare la prossima entry incontrata con chiave uguale a k, se tale entry esiste.

L'invariante vale all'inizio del ciclo ($i = n$) in quanto non ci sono entry $S[j]$ con $j \geq n$ e S' è vuota. Supponendo che valga alla fine dell'iterazione $i \leq n$, nella iterazione $i-1$ esso garantisce che la entry $S[i-1]$ venga inserita nella posizione corretta di S'. Questo fatto unito all'aggiornamento di $B[S[i-1]$.getKey()], rende vero l'invariante alla fine di questa nuova iterazione. È immediato verificare che alla fine del ciclo la validità dell'invariante implica che S' contiene le entry di S ordinate.

c. L'ordinamento è stabile perchè entry con la stessa chiave vengono prelevate da S da destra verso sinistra, e memorizzate in S' ugualmente da destra verso sinistra.

□

Problema 16 *Siano $S_1[0 \div n-1]$ and $S_2[0 \div n-1]$ due array ordinati di n chiavi intere ciascuno. In ogni array le chiavi sono distinte, ma una chiave può comparire in entrambi gli array.*

a. Progettare un algoritmo `CountNeg`*,* **contenente un solo ciclo***, che determina il* numero di chiavi negative distinte *che compaiono in S_1 e/o S_2. Ad esempio, se $S_1 = [-3,-2,0,34]$ e $S_2 = [-7,-2,34,45]$ l'algoritmo restituisce 3 (chiavi negative distinte: $-7,-3,-2$). Per avere punteggio pieno, l'algoritmo deve utilizzare spazio costante, oltre a S_1 e S_2.*

b. Specificare un opportuno invariante che serva per provare la correttezza dell'algoritmo.

Soluzione.

a. L'algoritmo è il seguente

Algoritmo `CountNeg(`S_1`,`S_2`)`

Input: `sequenze` S_1, S_2 `ordinate, ciascuna con chiavi distinte`
Output: `numero di chiavi negative distinte in` S_1 `e/o` S_2

```
i ← 0; j ← 0; C ← 0;
while ((i < n) AND (S1[i] < 0)) OR ((j < n) AND (S2[j] < 0)) do
    if (j = n) then {i++; C++};
    else
        if (i = n) then {j++; C++};
        else
            if (S1[i] = S2[j]) then {i++; j++; C++};
            else
                if (S1[i] < S2[j]) then {i++; C++};
                else {j++; C++};
return C
```

Oltre a S_1 e S_2 l'algoritmo usa spazio costante per le variabili i, j e C.

b. L'invariante che vale alla fine di ciascuna iterazione del while è il seguente:

- C è il numero di chiavi negative distinte in $S_1[0 \div i-1]$ e/o $S_2[0 \div j-1]$.
- Le chiavi in $S_1[0 \div i-1] \cup S_2[0 \div j-1]$ sono le $i \cdot j$ chiavi più piccole tra tutte le chiavi in $S_1 \cup S_2$.

□