Dati e Algoritmi I (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 $[2^d/2^i, 2^{d+1}/2^i]$ (induzione su i).*

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

Soluzione.

a. 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 l'ipotesi induttiva, corrisponde a un'invocazione su un'istanza di taglia $X \geq 2^d/2^i > 1$, avrà due figli al livello $i+1$ corrispondenti a invocazioni su istanze di taglia $\geq \lfloor X/2 \rfloor$. Se X è pari, allora $\lfloor X/2 \rfloor = X/2 \geq 2^d/(2 \cdot 2^i) = 2^d/2^{i+1}$. Se invece X è dispari, allora $X \geq (2^d/2^i) + 1$ e quindi $\lfloor X/2 \rfloor = (X-1)/2 \geq 2^d/2^{i+1}$. Ne consegue, che al livello $i+1$ ci saranno almeno 2^{i+1} nodi corrispondenti a invocazioni su istanze di taglia $\geq 2^d/2^{i+1}$.

b. Dal punto precedente sappiamo che i nodi del livello d dell'albero della ricorsione corrispondono a invocazioni su istanze di taglia in $[2^d/2^d, 2^{d+1}/2^d] = [1, 2]$. 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.

□

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. Progettare e analizzare un algoritmo efficiente per fondere le k sequenze in un'unica sequenza ordinata di taglia $n = k \cdot m$.*

Soluzione. Si osservi che se si adotta la strategia banale di fondere prima 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., il tempo di esecuzione risulta essere:

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

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`. □

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 che compaiono sia in* S_1 *che in* S_2*. L'algoritmo deve usare spazio aggiuntivo costante oltre alle due sequenze di input.*

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 `CountDuplicates(`S_1`,`S_2`)`

Input: `sequenze` S_1, S_2 `ordinate di chiavi distinte`
Output: `numero` C `di chiavi presenti sia in` S_1 `che in` S_2

$i \leftarrow 0$; $j \leftarrow 0$; $C \leftarrow 0$;
while `((`$i < n_1$`) AND (`$j < n_2$`))` **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. □

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 sia n_r il numero di elementi di S_2 più piccoli di $S_1[r]$, per $0 \leq r < S_1.size$. Si vede facilmente che il numero di inversioni in S è:

$$m_1 + m_2 + \sum_{r=0}^{S_1.size-1} n_r.$$

Ora, m_1 e m_2 possono essere determinate ricorsivamente. Se poi ricorsivamente ordiniamo anche le sequenze, i valori n_r possono essere determinati facilmente durante il merge delle

sottosequenze S_1 e S_2, una volta che queste sono ordinate. 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: `Sequenza` S `ordinata e numero` m `di inversioni in` S
if $(n \leq 1)$ **then return** S;
$S_1 \leftarrow S[0 \div \lceil n/2 \rceil - 1]$;
$S_2 \leftarrow S[\lceil n/2 \rceil \div n - 1]$;
$\{S_1, m_1\} \leftarrow$ `Sort-and-Inversions(`S_1`);`
$\{S_2, m_2\} \leftarrow$ `Sort-and-Inversions(`S_2`);`
$m \leftarrow m_1 + m_2$; $r \leftarrow 0$; $t \leftarrow 0$; $k \leftarrow 0$;
while $((r < S_1$`.size()) AND (`$t < S_2$`.size()))` **do**
  if $(S_1[r] < S_2[t])$ **then**
    $m \leftarrow m + t$; `/*` $t = n_r$ `*/`
    $S[k++] \leftarrow S_1[r++]$
  else $S[k++] \leftarrow S_2[t++]$;
while $(r <$ S_1`.size())` **do**
  $m \leftarrow m +$ S_2`.size();` `/*` S_2`.size()` $= n_r$ `*/`
  $S[k++] \leftarrow S_1[r++]$
while $(t <$ S_2`.size())` **do** $S[k++] \leftarrow S_2[t++]$;
return $\{S, m\}$

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 la correttezza dell'algoritmo* `Partition`(S, a, b) *visto a lezione (invocato da* `QuickSortInPlace` con $a < b$*) basandosi sul seguente invariante per il ciclo while esterno:*

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

Soluzione. Proviamo anzitutto che l'invariante vale alla fine di ciascuna iterazione del while. All'inizio del ciclo l'invariante vale dato che $p = S[b]$, $\ell = a$, $r = b - 1$ e $a < b$. Supponiamo che esso valga alla fine di una generica iterazione del while che non sia l'ultima, e consideriamo l'iterazione successiva. È immediato vedere che né i due cicli while interni né l'eventuale swap effettuato nell'istruzione seguente possono invalidare l'invariante, e quindi esso continua a valere alla fine dell'iterazione. Dato che dal ciclo si esce appena si verifica che $\ell > r$, ovvero quando $\ell = r + 1$, a questo punto l'invariante garantisce che

- $S[j] \leq p$, per ogni $a \leq j < \ell$
- $S[j] \geq p$, per ogni $\ell \leq j \leq b - 1$.

Di conseguenza, lo swap finale tra $S[\ell]$ e $S[p]$ assicura la correttezza dell'algoritmo. □

Problema 7 *Con riferimento a* `QuickSortInPlace`*:*

a. *Dire per quali istanze l'algoritmo implementa fedelmente la strategia L-E-G descritta a lezione.*

b. *Dire come si comporta l'algoritmo 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:

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

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

- $\ell \leq r+1$;
- $S[i] < p$ 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 $r_1 < i \leq b$.

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 ≥ b) then return S;
p ← S[b];
ℓ ← a; ℓ1 ← ℓ; r ← b − 1; r1 ← b;
while (ℓ ≤ r) do
    while ((ℓ ≤ r) AND (S[ℓ] ≤ p)) do
        if (S[ℓ] < p) then
            swap(S[ℓ], S[ℓ1]);
            ℓ++; ℓ1++
        else ℓ++;
    while ((ℓ ≤ r) AND (S[r] ≥ p)) do
        if (S[r] > p) then
            swap(S[r], S[r1]);
            r−−; r1−−
        else r−−;
    if (ℓ < r) then swap(S[ℓ], S[r]);
QuickSortInPlace-v2(S, a, ℓ1 − 1);
QuickSortInPlace-v2(S, r1 + 1, b);
return S;
```

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 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 8 (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 primitva 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 V_L e V_G gli insiemi delle viti rispettivamente piccole e grandi per b, e siano D_L e D_G gli insiemi dei dadi rispettivamente piccoli e grandi per a. L'algoritmo viene chiamato ricorsivamente su (V_L, D_L) e su (V_G, D_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 9 *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 10 (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 11 (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**
    `let` $(c, v) = S_i[j]$;
    $S[\ell++] \longleftarrow ((i, c), v)$;
`RadixSort(S);`
for $i \longleftarrow 1$ **to** k **do** $\ell_i \longleftarrow 0$;
for $j \longleftarrow 0$ **to** $|S| - 1$ **do**
  `let` $((i, c), v) = S[j]$;
  $S_i[\ell_i++] \longleftarrow (c, v)$;
return $S_1, S_2, \ldots, S_k$

La correttezza è banalmente verificata. Riguardo alla complessità, sia $M = \max\{N, k+1\}$. RadixSort viene eseguito su una sequenza di n entry con chiavi formate da $d = 2$ cifre in $[0, M-1]$. Tale passo richiede quindi tempo $O(d(n + M)) = O(n + N)$ in quanto $k \leq n$, e quindi $M = O(n + N)$. Gli altri passi dell'algoritmo richiedono tempo $O(n)$. □

Problema 12 *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:

$S' \leftarrow$ `sequenza vuota;`
for $i \longleftarrow n-1$ **downto** 0 **do**
  $S'[B[S[i]\texttt{.getKey()}]-1] \leftarrow S[i]$;
  $B[S[i]\texttt{.getKey()}]--$
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 13 *Sia S una sequenza di n chiavi intere. Non si fanno ipotesi sul range delle chiavi, che può essere arbitrariamente grande, ma si sa che in S ci sono solo $k = O(\log n)$ chiavi distinte. Una chiave si dice* frequente *se compare almeno 10 volte in S. Progettare un algoritmo di complessità $o(n \log n)$ che determini quante sono le chiavi frequenti in S. Far vedere che la complessità è $o(n \log n)$.*

Soluzione. L'idea è di fare una scansione di S mantenendo in una mappa, a ogni iterazione della scansione, delle entry le cui chiavi sono le chiavi incontrate in S sino a quel momento e i cui valori sono, per ciascuna chiave, il numero di occorrenze di quella chiave incontrate sino a quel momento. Durante la scansione si mantiene anche il numero di chiavi frequenti. Lo pseudocodice è il seguente:

Algoritmo `FrequentKeys`(S)

Input: `Sequenza` S `di` n `chiavi, con` $k = O(\log n)$ `chiavi distinte`
Output: `Numero di chiavi frequenti`

```
count ← 0;
M ← mappa vuota;
for i ⟵ 0 to n − 1 do
    x ← M.get(S[i]);
    if (x = null) then x ← 0;
    M.put(S[i], x + 1);
    if (x + 1 = 10) then count ← count+1;
return count
```

In ogni momento M contiene al più $k = O(\log n)$ entry. Quindi, implementando M tramite un (2,4)-tree, ciascuna operazione su M richiede tempo $O(\log k) = O(\log \log n)$ e la complessità finale dell'algoritmo risulta essere $O(n \log \log n) = o(n \log n)$. □

Problema 14 (Esercizio C-13.35 [GTG14]). *Sia data una sequenza S di n elementi che rappresentano i voti per l'elezione di un presidente, dove ciascun voto è espresso tramite un intero associato a un particolare candidato. Si supponga di sapere che ci sono $k < n$ candidati. Descrivere un algoritmo che in tempo $O(n \log k)$ determina chi ha vinto l'elezione, ovvero chi ha ricevuto più voti. Si assuma che ogni candidato sia rappresentato da una chiave numerica, ma che non si abbiano informazioni sul range della chiave che quindi può essere anche molto grande. Analizzare la correttezza dell'algoritmo e far vedere che ha complessità $O(n \log k)$.*

Soluzione. L'idea è di fare una scansione di S mantenendo in una mappa, ad ogni iterazione della scansione, delle entry le cui chiavi sono i candidati incontrati e i cui valori sono i voti accumulati sino a quel momento. Durante la scansione è anche possibile mantenere il candidato con il massimo numero di voti e i voti ricevuti. Lo pseudocodice è il seguente:

Algoritmo `Election`(S)

Input: `Sequenza` S `di` n `interi, con` $k < n$ `interi distinti`
Output: `Intero piu' frequente in` S

```
max ← 0; winner ← null;
M ← mappa vuota;
for i ⟵ 0 to n − 1 do
    if (M.get(S[i])=null) then
        M.put(S[i], 1);
        if (1 > max) then max ← 1; winner ← S[i];
    else
        x ← M.get(S[i]);
        M.put(S[i], x + 1);
        if (x + 1 > max) then max ← x + 1; winner ← S[i];
return winner
```

La correttezza dell'algoritmo è basata sul seguente invariante che vale alla fine dell'iterazione i del for, per ogni $i \geq -1$ (la fine dell'iterazione -1 è l'inizio del ciclo):

- Il candidato con il massimo numero di voti in $S[0 \div i]$ è `winner` e ha ricevuto `max` voti in $S[0 \div i]$
- La mappa M contiene una entry per ogni candidato che ha ricevuto voti in $S[0 \div i]$. La chiave della entry è l'intero che identifica il candidato, e il valore della entry è il numero di voti ricevuti dal candidato in $S[0 \div i]$

La verifica della correttezza tramite l'invariante sopra indicato è molto semplice ed è lasciata come esercizio. Si osservi che in ogni momento M contiene al più k entry. Quindi, implementando M tramite un (2,4)-tree, ciascuna operazione su M richiede tempo $O(\log k)$ e la complessità finale dell'algoritmo risulta essere $O(n \log k)$. □