CAPITOLO

5

Casualità e ammortamento

In questo capitolo descriviamo due tecniche algoritmiche molto diffuse che hanno lo scopo di ottenere costi totali più bassi di quelli forniti dall'analisi al caso pessimo. La prima utilizza la casualità per ottenere strutture di dati e algoritmi randomizzati efficienti. La seconda utilizza un'analisi più raffinata su sequenze di operazioni per strutture di dati ammortizzate.

5.1 Ordinamento randomizzato per distribuzione

Nel Capitolo 3 abbiamo osservato che l'algoritmo di ordinamento per distribuzione o *quicksort* descritto nel Codice 3.5 ha una complessità che dipende dall'ordine iniziale degli elementi: se la distribuzione iniziale degli elementi è sbilanciata si ha un costo quadratico contro un costo $O(n \log n)$ nel caso di distribuzioni bilanciate.

In questo paragrafo mostreremo un'analisi al caso medio più robusta che risulta essere *indipendente* dall'ordine iniziale degli elementi nell'array e si basa sull'uso della **casualità** per far sì che la distribuzione sbilanciata occorra con una probabilità trascurabile. Concretamente, se gli `n` elementi sono già ordinati, *quicksort* richiede sempre tempo $\Theta(n^2)$ anche se in media il tempo è $O(n \log n)$ se uno considera tutti i possibili array di ingresso. Con la sua versione randomizzata, facciamo in modo che ogni volta che *quicksort* viene eseguito su uno *stesso* array in ingresso, il comportamento non sia deterministico, ma sia piuttosto dettato da scelte casuali (che comunque calcolano correttamente l'ordinamento finale). Ne deriva che nell'analisi di complessità il numero medio di passi non si calcola più su *tutti* i possibili array di ingresso, ma su *tutte* le scelte casuali per un array d'ingresso: è una nozione più forte perché uno stesso array non può essere sempre sfavorevole a *quicksort*, in quanto anche le scelte casuali di quest'ultimo adesso entrano in gioco.

A tal fine, l'unica modifica che occorre apportare all'algoritmo *quicksort* (mostrato nel Codice 5.1) è nella riga 4 e riguarda la scelta del `pivot` che deve avvenire in modo aleatorio, equiprobabile e uniforme nell'intervallo `[sinistra...destra]`: a questo scopo viene utilizzata la primitiva `random()` per generare un valore reale `r` pseudocasuale compreso tra `0` e `1` inclusi, in modo uniforme ed equiprobabile (tale generatore è disponibile in molte librerie per la programmazione e non è semplice ottenerne uno statisticamente significativo, in quanto il programma che lo genera è deterministico). Un tale algoritmo si chiama **casuale** o **randomizzato** perché impiega la casualità per sfuggire a situazioni sfavorevoli, risultando più robusto rispetto a tali eventi (come nel nostro caso, in presenza di un array già in ordine crescente).

ALVIE **Codice 5.1** Ordinamento randomizzato per distribuzione di un array a.

```
1  QuickSort( a, sinistra, destra ):
2                              ⟨pre: 0 ≤ sinistra, destra ≤ n − 1⟩
3    IF (sinistra < destra) {
4      pivot = sinistra + (destra − sinistra) × random();
5      rango = Distribuzione( a, sinistra, pivot, destra );
```

```
6      QuickSort( a, sinistra, rango-1 );
7      QuickSort( a, rango+1, destra );
8    }
```

Teorema 5.1 *L'algoritmo* quicksort *randomizzato impiega tempo ottimo* $O(n \log n)$ *nel caso medio per ordinare* n *elementi.*

Dimostrazione Il risultato della scelta casuale e uniforme del `pivot` è che il valore di `rango` restituito da `Distribuzione` nella riga 5 è anch'esso uniformemente distribuito tra le (equiprobabili) posizioni in `[sinistra ... destra]`. Supponiamo pertanto di dividere tale segmento in quattro parti uguali, chiamate **zone**. In base a quale zona contiene la posizione `rango` restituita nella riga 5, otteniamo i seguenti *due* eventi *equiprobabili*:

- la posizione `rango` ricade nella prima o nell'ultima zona: in tal caso, `rango` è detto essere *esterno*;
- la posizione `rango` ricade nella seconda o nella terza zona: in tal caso, `rango` è detto essere *interno*.

Indichiamo con $T(n)$ il *costo medio* dell'algoritmo *quicksort* eseguito su un array di n elementi. Osserviamo che la media $\frac{x+y}{2}$ di due valori x e y può essere vista come la loro somma pesata con la rispettiva probabilità $\frac{1}{2}$, ovvero $\frac{1}{2}x + \frac{1}{2}y$, considerando i due valori come equiprobabili. Nella nostra analisi, x e y sono sostituiti da opportuni valori di $T(n)$ corrispondenti ai due eventi equiprobabili sopra introdotti. Più precisamente, quando `rango` è esterno (con probabilità $\frac{1}{2}$), la distribuzione può essere estremamente sbilanciata nella ricorsione e, come abbiamo visto, quest'ultima può richiedere fino a $x = T(n-1) + O(n) \leq T(n) + c'n$ tempo, dove il termine $O(n)$ si riferisce al costo della distribuzione effettuata nel Codice 3.6 e $c' > 0$ è una costante sufficientemente grande. Quando invece `rango` è interno (con probabilità $\frac{1}{2}$), la distribuzione più sbilanciata possibile nella ricorsione avviene se `rango` corrisponde al minimo della seconda zona oppure al massimo della terza. Ne deriva una distribuzione dei dati che porta alla ricorsione su circa $\frac{n}{4}$ elementi in una chiamata di `QuickSort` e $\frac{3}{4}n$ elementi nell'altra (le altre distribuzioni in questo caso non possono andare peggio perché sono meno sbilanciate). In tal caso, la ricorsione richiede al più $y = T\left(\frac{n}{4}\right) + T\left(\frac{3}{4}n\right) + O(n) \leq T(n) + c'n$ tempo, dove la costante c' è scelta sufficientemente grande da coprire entrambi gli eventi. Facendo la media pesata di x e y, otteniamo

$$T(n) \leq \frac{1}{2}x + \frac{1}{2}y = \frac{1}{2}\left[T(n) + T\left(\frac{n}{4}\right) + T\left(\frac{3}{4}n\right)\right] + c'n \qquad (5.1)$$

Moltiplicando entrambi i termini nella (5.1) per 2, risolvendo rispetto a $T(n)$ e ponendo $c = 2c'$, otteniamo la relazione di ricorrenza

$$T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{3}{4}n\right) + cn \tag{5.2}$$

la cui soluzione dimostriamo essere $T(n) = O(n \log n)$ nel Paragrafo 5.1.1. Il tempo medio è ottimo se contiamo il numero di confronti tra elementi. ❑

5.1.1 Alternativa al teorema fondamentale delle ricorrenze

La relazione di ricorrenza (5.2) non è risolvibile con il teorema fondamentale delle ricorrenze (Teorema 3.1), in quanto non è un'istanza della (4.2). In generale, quando una relazione di ricorrenza non ricade nei casi del teorema fondamentale delle ricorrenze, occorre determinare tecniche di risoluzione alternative come più specificatamente vedremo nella dimostrazione del risultato che segue.

Teorema 5.2 *La soluzione della relazione di ricorrenza (5.2) è* $T(n) = O(n \log n)$.

Dimostrazione Notiamo che il valore $T(n)$ nella (5.2) (livello 0 della ricorsione) è ottenuto sommando a cn i valori restituiti dalle due chiamate ricorsive: quest'ultime, che costituiscono il livello 1 della ricorsione, sono invocate l'una con input $\frac{n}{4}$ e l'altra con input $\frac{3}{4}n$ e, in corrispondenza di tale livello, contribuiscono al valore $T(n)$ per un totale di $c\frac{n}{4} + c\frac{3}{4}n = cn$.

Passando al livello 2 della ricorsione, ciascuna delle chiamate del livello 1 ne genera altre due, per un totale di quattro chiamate, rispettivamente con input $\frac{n}{4^2}$, $\frac{3}{4^2}n$, $\frac{3}{4^2}n$ e $\frac{3^2}{4^2}n$, che contribuiscono al valore $T(n)$ per un totale di $c\frac{n}{4^2} + c\frac{3}{4^2}n + c\frac{3}{4^2}n + c\frac{3^2}{4^2}n = cn$ in corrispondenza del livello 2. Non ci dovrebbe sorprendere, a questo punto, che il contributo del livello 3 della ricorsione sia al più cn (in generale qualche chiamata ricorsiva può raggiungere il caso base e terminare).

Per calcolare il valore finale di $T(n)$ in forma chiusa, occorre sommare i contributi di tutti i livelli. Il livello s più profondo si presenta quando seguiamo ripetutamente il "ramo $\frac{3}{4}$", ovvero viene soddisfatta la relazione $\left(\frac{3}{4}\right)^s n = 1$ da cui deriva che $s = \log_{4/3} n = O(\log n)$. Possiamo quindi limitare superiormente $T(n)$ osservando che ciascuno degli $O(\log n)$ livelli di ricorsione contribuisce al suo valore per al più $cn = O(n)$ e, pertanto, $T(n) = O(n \log n)$. ❑

Intuitivamente, dividere n elementi in proporzione a $\frac{1}{4}$ e $\frac{3}{4}$, invece che a $\frac{1}{2}$ e $\frac{1}{2}$ (come accade nel caso dell'algoritmo di ordinamento per fusione), fornisce comunque una partizione bilanciata perché la dimensione di ciascuna parte differisce dall'altra soltanto per un fattore costante. La proprietà che $T(n) = O(n \log n)$ può essere estesa a una partizione di n in proporzione a $\frac{1}{3}$ e $\frac{2}{3}$ e, in generale, in proporzione a δ e $1 - \delta$ per una qualunque costante $0 < \delta < 1$.

Da quanto discusso finora, possiamo dedurre una linea guida per lo sviluppo di una forma chiusa della soluzione $T(n)$ di una relazione di ricorrenza del tipo

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ T(\delta n) + T((1-\delta)n) + cf(n) & \text{altrimenti} \end{cases} \tag{5.3}$$

una qualunque costante $0 < \delta < 1$. Non potendo applicare il teorema fondamentale delle ricorrenze, procediamo per passaggi intermedi con le corrispondenti chiamate ricorsive. La chiamata ricorsiva iniziale (livello 0) con input n contribuisce al valore di $T(n)$ per un totale di $cf(n)$ e dà luogo a due chiamate ricorsive di livello 1, una con input $n' = \delta n$ e l'altra con input $n'' = (1-\delta)n$, dove $n' + n'' = n$. Queste ultime due chiamate contribuiscono per un totale di $cf(n') + cf(n'')$ e inoltre invocano ulteriori due chiamate ricorsive a testa, le quali costituiscono il livello 2 della ricorsione e ricevono in input m_0, m_1, m_2 e m_3 tali che $m_0 + m_1 + m_2 + m_3 = n$. Dovrebbe essere chiaro a questo punto che queste chiamate contribuiscono per un totale di $cf(m_0) + cf(m_1) + cf(m_2) + cf(m_3)$ e inoltre invocano ulteriori due chiamate ricorsive a testa. La forma chiusa per un limite superiore della (5.3) è data dalla somma dei termini noti

$$T(n) \leq cf(n) + [cf(n') + cf(n'')] + [cf(m_0) + cf(m_1) + cf(m_2) + cf(m_3)] + \cdots$$

la cui valutazione dipende dal tipo della funzione $f(n)$: per esempio, abbiamo visto che se $f(n) = n$, allora otteniamo $T(n) = O(n \log n)$. Siccome alcune chiamate potrebbero terminare prima, abbiamo che la somma dei termini noti rappresenta un limite superiore.

Esercizio svolto 5.1 Consideriamo la funzione `QuickSelect` riportata nel Codice 3.7 del Capitolo 3: rendiamola randomizzata operando la scelta del `pivot` come avviene nella riga 4 del Codice 5.1. Mostrare che la media del costo nel caso della `QuickSelect` è $O(n)$.

Soluzione L'equazione di ricorrenza per il costo al caso medio è costruita in modo simile all'equazione (5.1), con la differenza che conteggiamo una sola chiamata ricorsiva (la più sbilanciata) ottenendo $T(n) \leq \frac{1}{2}\left[T(n) + T\left(\frac{3}{4}n\right)\right] + c'n$.

Moltiplicando entrambi i termini per 2, risolvendo rispetto a $T(n)$ e sostituendo $c = 2c'$, otteniamo la relazione di ricorrenza $T(n) \leq T\left(\frac{3}{4}n\right) + cn$. Possiamo applicare il teorema fondamentale delle ricorrenze ponendo $\alpha = 1$, $\beta = \frac{4}{3}$ e $f(n) = n$ nell'equazione (4.2), per cui rientriamo nel primo caso $\left(\gamma = \frac{3}{4}\right)$ ottenendo in media una complessità temporale $O(n)$ per l'algoritmo random di selezione per distribuzione. (Osserviamo che esiste un algoritmo lineare al caso pessimo, ma di interesse più teorico.)

5.2 Dizionario basato su liste randomizzate

Abbiamo discusso nel Paragrafo 5.1 come la casualità possa essere applicata all'algoritmo di ordinamento per distribuzione (*quicksort*) nella scelta del `pivot`. Una configurazione sfavorevole dei dati o della sequenza di operazioni che agisce su di essi può rendere inefficiente diversi algoritmi se analizziamo la loro complessità nel caso pessimo. In generale, la strategia che consente di individuare le configurazioni che peggiorano le prestazioni di un algoritmo è chiamata *avversariale* in quanto suppone che un avversario malizioso generi tali configurazioni sfavorevoli in modo continuo. In tale contesto, la casualità riveste un ruolo rilevante per la sua caratteristica imprevedibilità: vogliamo sfruttare quest'ultima a nostro vantaggio, impedendo a un tale avversario di prevedere le configurazioni sfavorevoli (in senso algoritmico).

Nel seguito descriviamo un algoritmo random per il problema dell'inserimento e della ricerca di una chiave k in una lista e dimostriamo che la strategia da esso adottata è vincente, sotto opportune condizioni. In particolare, usando una lista randomizzata di n elementi ordinati, i tempi *medi* o *attesi* delle operazioni di ricerca e inserimento sono ridotti a $O(\log n)$: anche se, al caso pessimo, tali operazioni possono richiedere tempo $O(n)$, è altamente improbabile che ciò accada.

Descriviamo una particolare realizzazione del dizionario mediante liste randomizzate, chiamate **liste a salti** (*skip list*), la cui idea di base (non random) può essere riassunta nel modo seguente, secondo quanto illustrato nella Figura 5.1. Partiamo da una lista *ordinata* di $n + 2$ elementi, $L_0 = e_0, e_1, \ldots, e_{n+1}$, la quale costituisce il livello 0 della lista a salti: poniamo che il primo e l'ultimo elemento della lista siano i due valori speciali, $-\infty$ e $+\infty$, per cui vale sempre $-\infty < e_i < +\infty$, per $1 \leq i \leq n$. Per ogni elemento e_i della lista L_0 $(1 \leq i \leq n)$ creiamo r_i copie di e_i, dove 2^{r_i} è la massima potenza di 2 che divide i (nel nostro esempio, per $i = 1, 2, 3, 4, 5, 6, 7$, abbiamo $r_i = 0, 1, 0, 2, 0, 1, 0$). Ciascuna copia ha livello crescente $\ell = 1, 2, \ldots, r_i$ e punta alla copia di livello inferiore $\ell - 1$: supponiamo inoltre che $-\infty$ e $+\infty$ abbiano sempre una copia per ogni livello creato. Chiaramente, il massimo livello o *altezza* h della lista a salti è dato dal massimo valore di r_i e, quindi, $h = O(\log n)$.

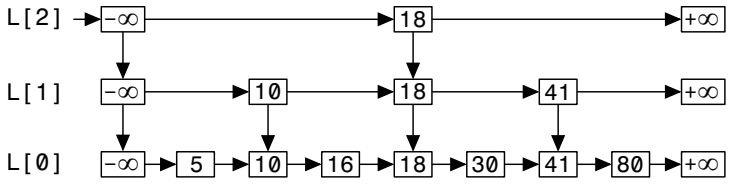


Figura 5.1 Un esempio di lista a salti di altezza h = 2.

Passando a una visione orizzontale, tutte le copie dello stesso livello ℓ ($0 \le \ell \le h$) sono collegate e formano una sottolista L_ℓ, tale che $L_h \subseteq L_{h-1} \subseteq \cdots \subseteq L_0$:[1] come possiamo vedere nell'esempio mostrato nella Figura 5.1, le liste dei livelli superiori "saltellano" (*skip* in inglese) su quelle dei livelli inferiori. Osserviamo che, se la lista di partenza, L_0, contiene $n + 2$ elementi ordinati, allora L_1 ne contiene al più $2 + n/2$, L_2 ne contiene al più $2 + n/4$ e, in generale, L_ℓ contiene al più $2 + n/2^\ell$ elementi ordinati. Pertanto, il numero totale di copie presenti nella lista a salti è limitato superiormente dal seguente valore:

$$(2 + n) + (2 + n/2) + \cdots + (2 + n/2^h) = 2(h + 1) + \sum_{\ell=0}^{h} n/2^\ell$$
$$= 2(h + 1) + n\sum_{\ell=0}^{h} 1/2^\ell < 2(h + 1) + 2n$$

In altre parole, il numero totale di copie è $O(n)$.

Per descrivere le operazioni di ricerca e inserimento, necessitiamo della nozione di predecessore. Data una lista $L_\ell = e'_0, e'_1, \ldots, e'_{m-1}$ di elementi ordinati e un elemento x, diciamo che $e'_j \in L_\ell$ (con $0 \le j < m - 1$) è il **predecessore** di x (in L_ℓ) se e'_j è il massimo tra i minoranti di x, ovvero $e'_j \le x < e'_{j+1}$: osserviamo che il predecessore è sempre ben definito perché il primo elemento di L_ℓ è $-\infty$ e l'ultimo elemento è $+\infty$.

ESEMPIO 5.1

La ricerca di una chiave k è concettualmente semplice. Per esempio, supponiamo di voler cercare la chiave 80 nella lista mostrata nella Figura 5.1. Partendo da L_2, troviamo che il predecessore di 80 in L_2 è 18: a questo punto, passiamo alla copia di 18 nella lista L_1 e troviamo che il predecessore di 80 in quest'ultima lista è 41. Passando alla copia di 41 in L_0, troviamo il predecessore di 80 in questa lista, ovvero 80 stesso: pertanto, la chiave è stata trovata.

Tale modo di procedere è mostrato nel Codice 5.2, in cui supponiamo che gli elementi in ciascuna lista L_ℓ per $\ell > 0$ abbiamo un riferimento `inf` per raggiungere la corrispondente copia nella lista inferiore $L_{\ell-1}$. Partiamo dalla lista L_h (riga 2) e tro-

[1] Con un piccolo abuso di notazione, scriviamo $L \subseteq L'$ se l'insieme degli elementi di L è un sottoinsieme di quello degli elementi di L'.

viamo il predecessore p_h di k in tale lista (righe 4-6). Poiché $L_h \subseteq L_{h-1}$, possiamo raggiungere la copia di p_h in L_{h-1} (riga 7) e, a partire da questa posizione, scandire quest'ultima lista in avanti per trovare il predecessore p_{h-1} di k in L_{h-1}. Ripetiamo questo procedimento per tutti i livelli ℓ a decrescere: partendo dal predecessore p_ℓ di k in L_ℓ, raggiungiamo la sua copia in $L_{\ell-1}$, e percorriamo quest'ultima lista in avanti per trovare il predecessore $p_{\ell-1}$ di k in $L_{\ell-1}$ (righe 3-8). Quando raggiungiamo L_0 (ovvero, p è uguale a `null` nella riga 3), la variabile `predecessore` memorizza p_0, che è il predecessore che avremmo trovato se avessimo scandito L_0 dall'inizio di tale lista.

ALVIE **Codice 5.2** Scansione di una lista a salti per la ricerca di una chiave k.

```
1  ScansioneSkipList(k):      ⟨pre: gli elementi −∞ e +∞ fungono da sentinelle⟩
2    p = L[h];
3    WHILE (p != null) {
4      WHILE (p.succ.key <= k)
5        p = p.succ;
6      predecessore = p;
7      p = p.inf;
8    }
9    RETURN predecessore;
```

Il lettore attento avrà certamente notato che l'algoritmo di ricerca realizzato dal Codice 5.2 è molto simile alla ricerca binaria descritta nel caso degli array (Paragrafo 3.3): in effetti, ogni movimento seguendo il campo `succ` corrisponde a dimezzare la porzione di sequenza su cui proseguire la ricerca. Per questo motivo, è facile dimostrare che il costo della ricerca effettuata dal Codice 5.2 è $O(\log n)$ tempo, contrariamente al tempo $O(n)$ richiesto dalla scansione sequenziale di L_0.

Il problema sorge con l'operazione di inserimento, la cui realizzazione ricalca l'algoritmo di ricerca. Una volta trovata la posizione in cui inserire la nuova chiave, però, l'inserimento vero e proprio risulterebbe essere troppo costoso se volessimo continuare a mantenere le proprietà della lista a salti descritte in precedenza, in quanto questo potrebbe voler dire modificare le copie di tutti gli elementi che seguono la chiave appena inserita. Per far fronte a questo problema, usiamo la casualità: il risultato sarà un algoritmo random di inserimento nella lista a salti che non garantisce la struttura perfettamente bilanciata della lista stessa, ma che con alta probabilità continua a mantenere un'altezza media logaritmica e un tempo medio di esecuzione di una ricerca anch'esso logaritmico.

Notiamo che, senza perdita di generalità, la casualità può essere vista come l'esito di una sequenza di lanci di una moneta equiprobabile, dove ciascun lancio ha una possibilità su due che esca testa (codificata con `1`) e una possibilità su due che esca croce (codificata con `0`). Precisamente, diremo che la probabilità

di ottenere `1` è $q = \frac{1}{2}$ e la probabilità di ottenere `0` è $1 - q = \frac{1}{2}$ (in generale, un truffaldino potrebbe darci una moneta per cui $q \neq \frac{1}{2}$).

Attraverso una sequenza di `b` lanci, possiamo ottenere una **sequenza random** di `b` bit casuali.[2] Ciascun lancio è nella pratica simulato mediante la chiamata alla primitiva `random()`: il numero `r` generato pseudo-casualmente fornisce quindi il bit `0` se $0 \leq r < \frac{1}{2}$ e il bit `1` se $\frac{1}{2} \leq r < 1$.

Osserviamo che i lanci di moneta sono eseguiti in modo indipendente, per cui otteniamo una delle quattro possibili sequenze di $b = 2$ bit (`00`, `01`, `10` oppure `11`) in modo casuale, con probabilità $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$: in generale, le probabilità dei lanci si moltiplicano in quanto sono eventi indipendenti, ottenendo una sequenza di `b` bit casuali con probabilità $1/2^b$. Osserviamo inoltre che prima o poi dobbiamo incontrare un `1` nella sequenza se `b` è sufficientemente grande.

Nel Codice 5.3 utilizziamo tale concetto di casualità per inserire una chiave `k` in una lista a salti di altezza `h`. Una volta identificati i suoi predecessori $p_0, p_1, \ldots, p_h$, in maniera analoga a quanto descritto per l'operazione di ricerca, li memorizziamo in un vettore `pred` (riga 2). Eseguiamo quindi una sequenza di $r \geq 1$ lanci di moneta fermandoci se otteniamo `1` oppure se `r = h + 1` (riga 3). Se `r = h + 1`, dobbiamo incrementare l'altezza (righe 5-9): creiamo una nuova lista L_{h+1} composta dalle chiavi $-\infty$ e $+\infty$, indichiamo il primo elemento di tale lista come predecessore p_{h+1} della chiave `k` e incrementiamo il valore di `h`. In ogni caso, creiamo `r` copie di `k` e le inseriamo nelle liste $L_0, L_1, L_2, \ldots, L_r$ (righe 10-13): ciascuna inserzione richiede tempo costante in quanto va creato un nodo immediatamente dopo ciascuno dei predecessori $p_0, p_1, \ldots, p_r$. Come vedremo, il costo totale dell'operazione è, in media, $O(\log n)$.

ALVIE **Codice 5.3** Inserimento di una chiave in una lista a salti `L`, dove `nuovo` rappresenta un elemento allocato a ogni iterazione.

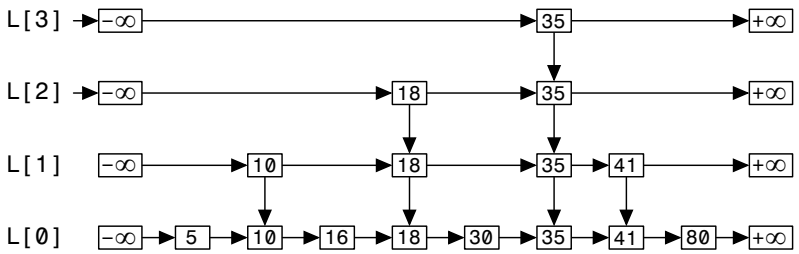
```
1  InserimentoSkipList( k ):
2    pred = [p_0,p_1,...,p_h];
3    FOR (r = 1; r <= h && random() < 0.5; r = r + 1)
4      ;
5    IF (r > h) {
6      piu.chiave = +∞; piu.succ = piu.inf = null;
7      meno.chiave = -∞; meno.succ = piu; meno.inf = L[h];
```

[2] La nozione di sequenza random `R` è stata formalizzata nella teoria di Kolmogorov in termini di incompressibilità, per cui qualunque programma che generi `R` non può richiedere significativamente meno bit per la sua descrizione di quanti ne contenga `R`. Per esempio, `R = 0101010101` non è casuale in quanto un programma che scrive per `b/2` volte `01` può generarla richiedendo solo $O(\log b)$ bit per la sua descrizione. Purtroppo è indecidibile stabilire se una sequenza è random anche se la stragrande maggioranza delle sequenze binarie lo sono.

```
8     pred[h+1] = L[h+1] = meno; h = h + 1;
9   }
10  FOR (i = 0, ultimo = null; i <= r; i = i + 1) {
11    nuovo.chiave = k; nuovo.succ = pred[i].succ; nuovo.inf = ultimo;
12    ultimo = pred[i].succ = nuovo;
13  }
```

ESEMPIO 5.2

Consideriamo l'inserimento della chiave 35 nella lista a salti illustrata nella Figura 5.1, dove h = 2. Il primo elemento dell'array pred dei predecessori, pred[0], punta all'elemento con chiave 30 di L[0] mentre pred[1] e pred[2] puntano all'elemento contenente 18 di L[1] e L[2], rispettivamente. Supponiamo che dalla riga 3 risulti r = 3 = h + 1: vengono eseguite le righe 5-9 e quindi viene aggiunta una lista L[3] contenente le due chiavi sentinella $-\infty$ e $+\infty$, inizializzando pred[3] al primo elemento della lista ($-\infty$) e aggiornando h = 3. A questo punto, con le righe 10-13 viene aggiunto un elemento con chiave 35 in L[0], L[1], L[2] e L[3] poiché r = 3, utilizzando i puntatori pred[0], pred[1], pred[2] e pred[3]. La lista a salti risultante è mostrata nella figura che segue.



Teorema 5.3 *La complessità media delle operazioni di ricerca e inserimento su una lista a salti (random) è* O(log n).

Dimostrazione La complessità del Codice 5.3 è la stessa della complessità della ricerca, ovvero del Codice 5.2. Pertanto analizziamo la complessità dell'operazione di ricerca.

Iniziamo col valutare un limite superiore per l'altezza media e per il numero medio di copie create con il procedimento appena descritto. La lista di livello più basso, L_0, contiene n + 2 elementi ordinati. Per ciascun inserimento di una chiave k, indipendentemente dagli altri inserimenti abbiamo lanciato una moneta equiprobabile per decidere se L_1 debba contenere una copia di k (bit 0) o meno (bit 1): quindi L_1 contiene circa n/2 + 2 elementi (una frazione costante di n in generale), perché i lanci sono equiprobabili e all'incirca metà degli elementi in L_0 ha ottenuto 0, creando una copia in L_1, e il resto ha ottenuto 1. Ripetendo tale argomento ai livelli successivi, risulta che L_2 contiene circa n/4 + 2 elementi, L_3 ne

contiene circa $n/8 + 2$ e così via: in generale, L_ℓ contiene circa $n/2^\ell + 2$ elementi ordinati e, quando $\ell = h$, l'ultimo livello ne contiene un numero costante $c > 0$, ovvero $n/2^h + 2 = c$. Ne deriviamo che l'altezza h è in media $O(\log n)$ e, in modo analogo a quanto mostrato in precedenza, che il numero totale medio di copie è $O(n)$ (la dimostrazione formale di tali proprietà sull'altezza e sul numero di copie richiede in realtà strumenti più sofisticati di analisi probabilistica).

Mostriamo ora che la ricerca descritta nel Codice 5.2 richiede tempo medio $O(\log n)$. Per un generico livello ℓ nella lista a salti, indichiamo con $T(\ell)$ il numero medio di elementi esaminati dall'algoritmo di scansione, a partire dalla posizione corrente nella lista L_ℓ fino a giungere al predecessore p_0 di k nella lista L_0: il costo della ricerca è quindi proporzionale a $O(T(h))$.

Per valutare $T(h)$, osserviamo che il cammino di attraversamento della lista a salti segue un profilo a gradino, in cui i tratti orizzontali corrispondono a porzioni della stessa lista mentre quelli verticali al passaggio alla lista del livello inferiore. Percorriamo a ritroso tale cammino attraverso i predecessori p_0, p_1, p_2, ..., p_h, al fine di stabilire induttivamente i valori di $T(0)$, $T(1)$, $T(2)$, ..., $T(h)$ (dove $T(0) = O(1)$, essendo già posizionati su p_0): notiamo che per $T(\ell)$ con $\ell \geq 1$, lungo il percorso (inverso) nel tratto interno a L_ℓ, abbiamo solo due possibilità rispetto all'elemento corrente $e \in L_\ell$.

1. Il percorso inverso proviene dalla copia di e nel livello inferiore (riga 7 del Codice 5.2), nella lista $L_{\ell-1}$, a cui siamo giunti con un costo medio pari a $T(\ell-1)$. Tale evento ha probabilità $\frac{1}{2}$ in quanto la copia è stata creata a seguito di un lancio della moneta che ha fornito 0.
2. Il percorso inverso proviene dall'elemento $\hat{e}$ a destra di e in L_ℓ (riga 5 del Codice 5.2), a cui siamo giunti con un costo medio pari a $T(\ell)$: in tal caso, $\hat{e}$ non può avere una corrispettiva copia al livello superiore (in $L_{\ell+1}$). Tale evento ha probabilità $\frac{1}{2}$, perché è il complemento dell'evento al punto 1.

Possiamo quindi esprimere il valore medio di $T(\ell)$ attraverso la media pesata (come per il *quicksort*) dei costi espressi nei casi 1 e 2, ottenendo la seguente relazione di ricorrenza per un'opportuna costante $c' > 0$:

$$T(\ell) \leq \frac{1}{2}\left[T(\ell) + T(\ell - 1)\right] + c' \tag{5.4}$$

Moltiplicando i termini della relazione (5.4), risolvendo rispetto a $T(\ell)$ e ponendo $c = 2c'$, otteniamo

$$T(\ell) \leq T(\ell - 1) + c \tag{5.5}$$

Espandendo (5.5), abbiamo $T(\ell) \leq T(\ell-1) + c \leq T(\ell-2) + 2c \leq \cdots \leq T(0) + \ell c = O(\ell)$. Quindi $T(h) = O(h) = O(\log n)$ è il costo medio della ricerca. ❑

Esercizio svolto 5.2 Discutere come realizzare l'operazione di cancellazione dalle liste randomizzate, valutandone la complessità in tempo.

Soluzione Sia `k` la chiave da cancellare e p_0, p_1, p_2, ..., p_h i suoi predecessori *stretti*, identificati mediante una variante della ricerca di `k`: notare che un predecessore p_i è stretto per `k` quando p_i < `k`. Dopo averli memorizzati nell'array `pred` come descritto all'inizio del Codice 5.3, è sufficiente eseguire `pred[i].succ = pred[i].succ.succ` per tutte le liste `L[i]` che contengono `k` (ossia, per cui `pred[i].succ.chiave = k`, essendo p_i un predecessore stretto). Se `L[h]` non contiene più chiavi (a parte $-\infty$ e $+\infty$), decrementiamo anche `h`. Il costo è dominato da quello della ricerca, quindi è $O(h) = O(\log n)$. È importante osservare che la cancellazione di `k` non incide sul numero di copie degli altri elementi. Per convincersene, consideriamo come vengono inseriti gli elementi: quando decidiamo il numero `r` di copie nel Codice 5.3, questo dipende solo dal lancio delle monete e non dagli elementi inseriti fino a quel momento. Quindi la presenza o meno della chiave `k` non influenza questo aspetto e le liste risultanti sono ancora randomizzate, ipotizzando che l'algoritmo di cancellazione non conosca quante copie ci sono per ciascun elemento ai fini dell'analisi probabilistica.

In conclusione, i dizionari basati su liste randomizzate sono un esempio concreto di come l'uso accorto della casualità possa portare ad algoritmi semplici che hanno in media (o con alta probabilità) ottime prestazioni in tempo e in spazio.

5.3 Unione e appartenenza a liste disgiunte

Le liste possono essere impiegate per operazioni di tipo insiemistico: avendo già visto come inserire e cancellare un elemento, siamo interessati a gestire una sequenza arbitraria `S` di operazioni di unione e appartenenza su un insieme di liste contenenti un totale di `m` elementi. In ogni istante le liste sono *disgiunte*, ossia l'intersezione di due liste qualunque è vuota. Inizialmente, abbiamo `m` liste, ciascuna formata da un solo elemento. Un'operazione di unione in `S` prende due delle liste attualmente disponibili e le concatena (non importa l'ordine di concatenazione). Un'operazione di appartenenza in `S` stabilisce se due elementi appartengono alla stessa lista.

Tale problema viene chiamato di *union-find* e trova applicazione, per esempio, in alcuni algoritmi su grafi che discuteremo in seguito. Mantenendo i riferimenti al primo e all'ultimo elemento di ogni lista, possiamo realizzare l'operazione di unione in tempo costante. Tuttavia, ciascuna operazione di appartenenza può richiedere tempo $O(m)$ al caso pessimo (pari alla lunghezza di una delle liste dopo una serie di unioni), totalizzando $O(nm)$ tempo per una sequenza di `n` operazioni.

Presentiamo un modo alternativo di implementare tali liste per eseguire un'arbitraria sequenza `S` di `n` operazioni delle quali n_1 sono operazioni di unione e

n_2 sono operazioni di appartenenza in $O(n_1 \log n_1 + n_2) = O(n \log n)$ tempo totale, migliorando notevolmente il limite di $O(nm)$ in quanto $n_1 < m$. Rappresentiamo ciascuna lista con un riferimento all'inizio e alla fine della lista stessa nonché con la sua lunghezza. Inoltre, corrediamo ogni elemento `z` di un riferimento `z.lista` alla propria lista di appartenenza: la regola intuitiva per mantenere tali riferimenti, quando effettuiamo un'unione tra due liste, consiste nel cambiare il riferimento `z.lista` negli elementi `z` della lista *più corta*. Vediamo come tale intuizione conduce a un'analisi rigorosa.

Il Codice 5.4 realizza tale semplice strategia per risolvere il problema di *union-find*, specificando l'operazione `Crea` per generare una lista di un solo elemento `x`, oltre alle funzioni `Appartieni` e `Unisci` per eseguire le operazioni di appartenenza e unione per due elementi `x` e `y`. In particolare, l'appartenenza è realizzata in tempo costante attraverso la verifica che il riferimento alla propria lista sia il medesimo. L'operazione di unione tra le due liste disgiunte degli elementi `x` e `y` determina anzitutto la lista più corta e quella più lunga (righe 2-8): cambia quindi i riferimenti `z.lista` agli elementi `z` della lista più corta (righe 9-13), concatena la lista lunga con quella corta (righe 14-15) e aggiorna la dimensione della lista risultante (riga 16).

ALVIE **Codice 5.4** Operazioni di creazione, appartenenza e unione nelle liste disgiunte.

```
1  Crea( x ):                                          ⟨pre: x non null⟩
2    lista.inizio = lista.fine = x;
3    lista.lunghezza = 1;
4    x.lista = lista;
5    x.succ = null;
```

```
1  Appartieni( x, y ):                                 ⟨pre: x, y non null⟩
2    RETURN (x.lista == y.lista);
```

```
1  Unisci( x, y ):                  ⟨pre: x, y non vuoti e x.lista ≠ y.lista⟩
2    IF (x.lista.lunghezza <= y.lista.lunghezza) {
3      corta = x.lista;
4      lunga = y.lista;
5    } ELSE {
6      corta = y.lista;
7      lunga = x.lista;
8    }
9    z = corta.inizio;
10   WHILE (z != null) {
11     z.lista = lunga;
12     z = z.succ;
13   }
```

```
14    lunga.fine.succ = corta.inizio;
15    lunga.fine = corta.fine;
16    lunga.lunghezza = corta.lunghezza + lunga.lunghezza;
```

L'efficacia della modalità di unione può essere mostrata in modo rigoroso facendo uso di un'analisi più approfondita, che prende il nome di **analisi ammortizzata** e che illustreremo in generale nel Paragrafo 5.5. Invece di valutare il costo al caso pessimo di una *singola* operazione, quest'analisi fornisce il costo al caso pessimo di una *sequenza* di operazioni, distribuendo quindi il costo delle poche operazioni costose nella sequenza sulle altre operazioni della sequenza che sono poco costose. La giustificazione di tale approccio è fornita dal fatto che, in tal modo, non ignoriamo gli effetti correlati delle operazioni sulla medesima struttura di dati. In generale, data una sequenza `S` di operazioni, diremo che il *costo ammortizzato* di un'operazione in `S` è un limite superiore al costo effettivo (spesso difficile da valutare) totalizzato dalla sequenza `S` diviso il numero di operazioni contenute in `S`. Naturalmente, più aderente al costo effettivo è tale limite, migliore è l'analisi ammortizzata fornita.

Teorema 5.4 *Il costo di* $n_1 < m$ *operazioni* `Unisci` *è* $O(n_1 \log m)$*, quindi il costo ammortizzato per operazione è* $O(\log m)$*. Il costo di ciascuna operazione* `Crea` *e* `Appartieni` *è invece* $O(1)$ *al caso pessimo.*

Dimostrazione È facile vedere che ciascuna delle operazioni `Crea` e `Appartieni` richiede tempo costante. Partendo da `m` elementi, ciascuno dei quali costituisce una lista di un singolo elemento (attraverso l'operazione `Crea`), possiamo concentrarci su un'arbitraria sequenza `S` di n_1 operazioni `Unisci`. Osserviamo che, al caso pessimo, la complessità in tempo di `Unisci` è proporzionale direttamente al numero di riferimenti `z.lista` che vengono modificati alla riga 11 del Codice 5.4: per calcolare il costo totale delle operazioni in `S`, è quindi sufficiente valutare un limite superiore al numero totale di riferimenti `z.lista` cambiati.

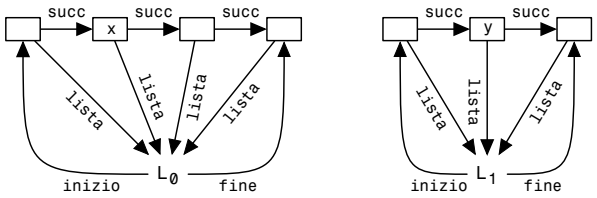
Possiamo conteggiare il numero di volte che la sequenza `S` può cambiare `z.lista` per un qualunque elemento `z` nel seguente modo. Inizialmente, l'elemento `z` appartiene alla lista di un solo elemento (se stesso). In un'operazione `Unisci`, *se* `z.lista` cambia, vuol dire che `z` va a confluire in una lista che ha una dimensione almeno *doppia* rispetto a quella di partenza. In altre parole, la prima volta che `z.lista` cambia, la dimensione della nuova lista contenente `z` è almeno 2, la seconda volta è almeno 4 e così via: in generale, l'`i`-esima volta che `z.lista` cambia, la dimensione della nuova lista contenente `z` è almeno 2^i. D'altra parte, al termine delle n_1 operazioni `Unisci`, la lunghezza di una qualunque lista è minore oppure uguale a $n_1 + 1 \leq m$: ne deriva che la lista contenente `z` ha lunghezza compresa tra 2^i e `m` (ovvero, $2^i \leq m$) e che vale sempre $i = O(\log m)$. Quindi, ogni elemento `z` vede cambiare il riferimento `z.lista` al più $O(\log m)$ volte. Sommando tale quantità per gli $n_1 + 1$ elementi `z` coinvolti nelle n_1 operazioni

`Unisci`, otteniamo un limite superiore di $O(n_1 \log m)$ al numero di volte che la riga 11 viene globalmente eseguita: pertanto, la complessità in tempo delle n_1 operazioni `Unisci` è $O(n_1 \log m)$ e, quindi, il costo *ammortizzato* di tale operazione è $O(\log m)$. Al costo di queste operazioni, va aggiunto il costo $O(1)$ per ciascuna delle operazioni `Crea` e `Appartieni`. ❑

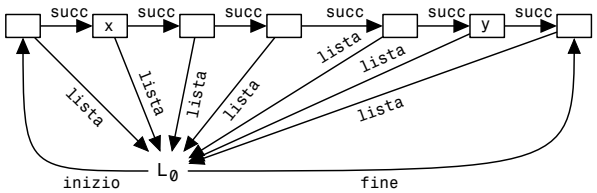
Lo schema adottato per cambiare i riferimenti `z.lista` è piuttosto generale: ipotizzando di avere insiemi disgiunti i cui elementi hanno ciascuno un'etichetta (sia essa `z.lista` o qualunque altra informazione) e applicando la regola che, quando due insiemi vengono uniti, si cambiano solo le etichette agli elementi dell'insieme di cardinalità minore, siamo sicuri che un'etichetta non possa venire cambiata più di $O(\log m)$ volte. L'intuizione di cambiare le etichette agli elementi del più piccolo dei due insiemi da unire viene rigorosamente esplicitata dall'analisi ammortizzata: notiamo che, invece, cambiando le etichette agli elementi del più grande dei due insiemi da unire, un'etichetta potrebbe venire cambiata $\Omega(n_1)$ volte, invalidando l'argomentazione finora svolta.

ESEMPIO 5.3

L'esecuzione dell'operazione `Unisci(x, y)` sulle liste mostrate nella figura che segue per prima cosa confronta la lunghezza della lista L_0 a cui appartiene x con quella della lista L_1 a cui appartiene y. Questa informazione viene reperita utilizzando i puntatori `lista` di x e y.



La lista L_0 è più lunga di L_1 quindi i puntatori `lista` degli elementi di L_1 vanno a puntare L_0. L'ultimo elemento di L_0 punta con `succ` al primo elemento di L_1 e al campo `fine` di L_0 viene assegnato il valore nel campo fine di L_1.



Infine viene aggiornato il campo `lunghezza` di L_0. La lista risultante è mostrata nella figura in alto. Si noti che non è più mostrato L_1 con i puntatori `inizio` e `fine`. Questi tuttavia continueranno a persistere ma d'ora in poi saranno ignorati.

5.4 Liste ad auto-organizzazione

L'auto-organizzazione delle liste è utile quando, per svariati motivi, la lista *non è necessariamente ordinata* in base alle chiavi di ricerca (contrariamente al caso delle liste randomizzate del Paragrafo 5.2). Per semplificare la discussione, consideriamo il solo caso della ricerca di una chiave k in una lista e adottiamo uno schema di scansione sequenziale: percorriamo la lista a partire dall'inizio verificando iterativamente se l'elemento attuale è uguale alla chiave cercata. Estendiamo tale schema per eseguire eventuali operazioni di auto-organizzazione al termine della scansione sequenziale (le operazioni di inserimento e cancellazione possono essere ottenute semplicemente, secondo quanto discusso nel Paragrafo 1.3).

Tale organizzazione sequenziale può trarre beneficio dal **principio di località temporale**, per il quale, se accediamo a un elemento in un dato istante, è molto probabile che accederemo a questo stesso elemento in istanti immediatamente (o quasi) successivi. Seguendo tale principio, sembra naturale che possiamo *riorganizzare* proficuamente gli elementi della lista dopo aver eseguito la loro scansione. Per questo motivo, una lista così gestita viene riferita come struttura di dati ad **auto-organizzazione** (*self-organizing* o *self-adjusting*). Tra le varie strategie di auto-organizzazione, la più diffusa ed efficace viene detta *move-to-front* (MTF), che consideriamo in questo paragrafo: essa consiste nello spostare l'elemento acceduto dalla sua posizione attuale alla cima della lista, senza cambiare l'ordine relativo dei rimanenti elementi, come mostrato nel Codice 5.5. Osserviamo che MTF effettua ogni ricerca senza conoscere le ricerche che dovrà effettuare in seguito: un algoritmo operante in tali condizioni, che deve quindi servire un insieme di richieste man mano che esse pervengono, viene detto **in linea** (*online*).

ALVIE **Codice 5.5** Ricerca di una chiave k in una lista ad auto-organizzazione.

```
1  MoveToFront( a, k ):
2    p = a;
3    IF (p == null ¦¦ p.dato == k) RETURN p;
4    WHILE (p.succ != null && p.succ.dato != k)
5      p = p.succ;
6    IF (p.succ == null) RETURN null;
7    tmp = p.succ;
8    p.succ = p.succ.succ;
9    tmp.succ = a;
10   a = tmp;
11   RETURN a;
```

Un esempio quotidiano di lista ad auto-organizzazione che utilizza la strategia MTF è costituito dall'elenco delle chiamate effettuate da un telefono cellulare: in effetti, è probabile che un numero di telefono appena chiamato, venga usato nuovamente nel prossimo futuro. Un altro esempio, più informatico, è proprio dei sistemi operativi, dove la strategia MTF viene comunemente denominata *least recently used* (LRU). In questo caso, gli elementi della lista corrispondono alle pagine di memoria, di cui solo le prime `r` possono essere tenute in una memoria ad accesso veloce. Quando una pagina è richiesta, quest'ultima viene aggiunta alle prime `r`, mentre quella a cui si è fatto accesso meno recentemente viene rimossa. Quest'operazione equivale a porre la nuova pagina in cima alla lista, per cui quella originariamente in posizione `r` (acceduta meno recentemente) va in posizione successiva, `r + 1`, uscendo di fatto dall'insieme delle pagine mantenute nella memoria veloce.

Per valutare le prestazioni della strategia MTF, il termine di paragone utilizzato sarà un algoritmo **non in linea** (*offline*), denominato OPT, che ipotizziamo essere a conoscenza di *tutte* le richieste che perverranno. Le prestazioni dei due algoritmi verranno confrontate rispetto al loro costo, definito come la somma dei costi delle singole operazioni, in accordo a quanto discusso sopra. In particolare, contiamo il numero di elementi della lista attraverso cui si passa prima di raggiungere l'elemento desiderato, a partire dall'inizio della lista: quindi *accedere all'elemento in posizione* `i` *ha costo* `i` in quanto dobbiamo attraversare gli `i` elementi che lo precedono. Lo spostamento in cima alla lista, operato da MTF, non viene conteggiato in quanto richiede sempre un costo costante.

Tale paradigma è ben esemplificato dalla gestione delle chiamate in uscita di un telefono cellulare: l'ultimo numero chiamato è già disponibile in cima alla lista per la prossima chiamata e il costo indica il numero di *clic* sulla tastierina per accedere a ulteriori numeri chiamati precedentemente (occorre un numero di clic pari a `i` per scandire gli elementi che precedono l'elemento in posizione `i` nell'ordine inverso di chiamata).

È di fondamentale importanza stabilire le regole di azione di OPT, perché questo può dare luogo a risultati completamente differenti. Nel nostro caso, esaminate tutte le richieste in anticipo, OPT *permuta* gli elementi della lista solo una volta all'inizio, *prima* di servire le richieste. Per semplificare la nostra analisi comparativa, OPT e MTF partono con la stessa lista iniziale: quando arriva una richiesta per l'elemento `k` in posizione `i`, OPT restituisce l'elemento scandendo i primi `i` elementi della lista, senza però muovere `k` dalla sua posizione, mentre MTF lo pone in cima alla lista. Inoltre, presumiamo che le liste non cambino di lunghezza durante l'elaborazione delle richieste.

Notiamo che OPT permuta gli elementi in un ordine che rende minimo il suo costo futuro. Chiaramente un tale algoritmo dotato di chiaroveggenza non esiste, ma è utile ai fini dell'analisi per valutare le potenzialità di MTF.

A titolo esemplificativo, è utile riportare i costi in termini concreti del numero di clic effettuati sui cellulari. Immaginiamo di essere in possesso, oltre al cellulare di marca MTF, di un futuristico cellulare OPT che conosce in anticipo le `n` chiamate che saranno effettuate nell'arco di un anno su di esso (l'organizzazione della lista delle chiamate in uscita è mediante le omonime politiche di gestione). Potendo usare entrambi i cellulari con gli stessi `m` numeri in essi memorizzati, effettuiamo alcune chiamate su tali numeri per un anno: quando effettuiamo una chiamata su di un cellulare, la ripetiamo anche sull'altro (essendo futuristico, OPT si aspetta già la chiamata che intendiamo effettuare). Per la chiamata `j`, dove `j = 0, 1, ..., n - 1`, contiamo il numero di clic che siamo costretti a fare per accedere al numero di interesse in MTF e, analogamente, annotiamo il numero di clic per OPT (ricordiamo che MTF pone il numero chiamato in cima alla sua lista, mentre OPT non cambia più l'ordine inizialmente adottato in base alle chiamate future). Allo scadere dell'anno, siamo interessati a stabilire il costo, ovvero il numero totale di clic effettuati su ciascuno dei due cellulari.

Mostriamo che, sotto opportune condizioni, il *costo di* MTF *non supera il doppio del costo di* OPT. In un certo senso, MTF offre una forma limitata di chiaroveggenza delle richieste rispetto a OPT, motivando il suo impiego in vari contesti con successo.

Formalmente, consideriamo una sequenza arbitraria di `n` operazioni di ricerca su una lista di `m` elementi, dove le operazioni sono numerate da `0` a `n - 1` in base al loro ordine di esecuzione. Per $0 \le j \le n-1$, l'operazione `j` accede a un elemento `k` nelle lista come nel Codice 5.5: sia c_j la posizione di `k` nella lista di MTF e c'_j la posizione di `k` nella lista di OPT. Poiché vengono scanditi c_j elementi prima di `k` nella lista di MTF, e c'_j elementi prima di `k` nella lista di OPT, definiamo il costo delle `n` operazioni, rispettivamente,

$$\texttt{costo}(\text{MTF}) = \sum_{j=0}^{n-1} c_j \quad \text{e} \quad \texttt{costo}(\text{OPT}) = \sum_{j=0}^{n-1} c'_j. \tag{5.6}$$

Teorema 5.5 *Partendo da liste uguali, vale* $\texttt{costo}(\text{MTF}) \le 2 \times \texttt{costo}(\text{OPT})$.

Dimostrazione Vogliamo mostrare che

$$\sum_{j=0}^{n-1} c_j \le 2 \sum_{j=0}^{n-1} c'_j \tag{5.7}$$

quando le liste di partenza sono uguali. Da tale diseguaglianza segue che MTF scandisce asintoticamente non più del doppio degli elementi scanditi da OPT. Nel seguito proviamo una condizione più forte di quella espressa nella diseguaglianza (5.7) da cui possiamo facilmente derivare quest'ultima: a tal fine, introduciamo la nozione di **inversione**. Supponiamo di aver appena eseguito l'operazione `j` che accede all'elemento `k`, e consideriamo le risultanti liste di MTF e OPT: un esempio di configurazione delle due liste in un certo istante è quello riportato nella Figura 5.2.

Lista MTF	=	e_4	e_2	e_6	e_0	e_1	e_3	e_7	e_5
Lista OPT	=	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7

Figura 5.2 Un'istantanea delle liste manipolate da MTF e OPT.

Presi due elementi distinti `x` e `y` in una delle due liste, questi devono occorrere anche nell'altra: diciamo che l'insieme `{x, y}` è un'inversione quando l'ordine relativo di occorrenza è diverso nelle due liste, ovvero quando `x` occorre prima di `y` (non necessariamente in posizioni adiacenti) in una lista mentre `y` occorre prima di `x` nell'altra lista. Nel nostro esempio, $\{e_0, e_2\}$ è un'inversione, mentre $\{e_1, e_7\}$ non lo è. Definiamo con Φ_j il numero di inversioni tra le due liste dopo che è stata eseguita l'operazione `j`: vale $0 \le \Phi_j \le \frac{m(m-1)}{2}$, per $0 \le j \le n-1$, in quanto $\Phi_j = 0$ se le due liste sono uguali mentre, se sono una in ordine inverso rispetto all'altra, ognuno degli $\binom{m}{2}$ insiemi di due elementi è un'inversione. Per dimostrare la (5.7), non possiamo utilizzare direttamente la proprietà che $c_j \le 2c'_j + O(1)$, in quanto questa proprietà in generale non è vera. Invece, ammortizziamo il costo usando il numero di inversioni Φ_j, in modo da dimostrare la seguente relazione (introducendo un valore fittizio $\Phi_{-1} = 0$ con l'ipotesi che MTF e OPT partano da liste uguali):

$$c_j + \Phi_j - \Phi_{j-1} \le 2c'_j \tag{5.8}$$

Possiamo derivare la (5.7) dalla (5.8) in quanto quest'ultima implica che

$$\sum_{j=0}^{n-1} (c_j + \Phi_j - \Phi_{j-1}) \le 2\sum_{j=0}^{n-1} c'_j$$

I termini Φ nella sommatoria alla sinistra della precedente diseguaglianza formano una cosiddetta **somma telescopica**, $(\Phi_0 - \Phi_{-1}) + (\Phi_1 - \Phi_0) + (\Phi_2 - \Phi_1) + \cdots + (\Phi_{n-1} - \Phi_{n-2}) = \Phi_{n-1} - \Phi_{-1} = \Phi_{n-1} \ge 0$, nella quale le coppie di termini di segno opposto si elidono algebricamente: da questa osservazione segue immediatamente che

$$\sum_{j=0}^{n-1} c_j \le 2\sum_{j=0}^{n-1} c'_j - \Phi_{n-1} \le 2\sum_{j=0}^{n-1} c'_j \tag{5.9}$$

ottenendo così la disuguaglianza (5.7).

Possiamo quindi concentrarci sulla dimostrazione dell'equazione (5.8), dove il caso $j = 0$ vale per sostituzione del valore fissato per $\Phi_{-1} = 0$, in quanto $\Phi_0 = c_0 = c'_0$ visto che inizialmente le due liste sono uguali e quindi dopo la prima operazione vengono create c_0 inversioni perché la chiave cercata viene spostata solo da MTF.

Ipotizziamo quindi che l'operazione $j > 0$ sia stata eseguita: a tale scopo, sia `k` l'elemento acceduto in seguito a tale operazione e ipotizziamo che `k` occupi la posizione `i` nella lista di MTF (per cui $c_j = i$): notiamo che la (5.8) è banalmente soddisfatta quando $i = 0$ perché la lista di MTF non cambia e, quindi, $\Phi_j = \Phi_{j-1}$.

Prendiamo l'elemento k' che appare in una generica posizione $i' < i$. Ci sono solo due possibilità se esaminiamo l'insieme $\{k', k\}$: è un'inversione oppure non lo è. Quando MTF pone k in cima alla lista, tale insieme diventa un'inversione se e solo se non lo era prima: nel nostro esempio, se $k = e_3$ (per cui $i = 5$), possiamo riscontrare che, considerando gli elementi k' nelle posizioni da 0 a 4, due di essi, e_4 ed e_6, formano (assieme a e_3) un'inversione mentre i rimanenti tre elementi non danno luogo a inversioni. Quando e_3 viene posto in cima alla lista di MTF, abbiamo che gli insiemi $\{e_3, e_4\}$ ed $\{e_3, e_6\}$ non sono più inversioni, mentre lo diventano gli insiemi $\{e_2, e_3\}$, $\{e_0, e_3\}$ e $\{e_1, e_3\}$.

In generale, gli i elementi che precedono k nella lista di MTF sono composti da f elementi che (assieme a k) *danno* luogo a inversioni e da g elementi che *non danno* luogo a inversioni, dove $f + g = i$. Dopo che MTF pone k in cima alla sua lista, il numero di inversioni che cambiano sono esclusivamente quelle che coinvolgono k. In particolare, le f inversioni *non* sono più tali mentre appaiono g *nuove* inversioni, come illustrato nel nostro esempio. Di conseguenza, pur non sapendo stimare individualmente il numero di inversioni Φ_j e Φ_{j-1}, possiamo inferire che la loro differenza dopo l'operazione j è $\Phi_j - \Phi_{j-1} = -f + g$. Ne deriva che $c_j + \Phi_j - \Phi_{j-1} = i - f + g = (f + g) - f + g = 2g$.

Consideriamo ora la posizione c'_j dell'elemento k nella lista di OPT: sappiamo certamente che $c'_j \geq g$ perché ci sono almeno g elementi che precedono k, in quanto appaiono prima di k anche nella lista di MTF e non formano inversioni con k prima dell'operazione j. A questo punto, otteniamo l'equazione (5.8), in quanto $c_j + \Phi_j - \Phi_{j-1} = 2g \leq 2c'_j$, concludendo di fatto l'analisi ammortizzata. ❑

Esercizio svolto 5.3 Siano $s_0, s_1, \ldots, s_{m-1}$ gli m elementi della lista nell'ordine iniziale stabilito dall'algoritmo OPT. Indicando con f_i il numero di volte in cui s_i viene richiesto dalla sequenza di n accessi, dove $\sum_{i=0}^{m-1} f_i = n$, mostrare che $f_0 \geq f_1 \geq \cdots \geq f_{m-1}$: cioè, mostrare che OPT organizza gli elementi della sua lista in ordine non crescente di frequenza.

Soluzione Poiché OPT paga un costo i per accedere a s_i (senza cambiare la lista) e questo accade f_i volte, possiamo derivare che $\texttt{costo}(\text{OPT}) = \sum_{i=0}^{m-1} i \times f_i$. Per assurdo, ipotizziamo che esistano $i' < i$ tali che $f_{i'} < f_i$, ossia la lista di OPT non è in ordine (non crescente di frequenza). Scambiando di posto $s_{i'}$ e s_i nella lista prima dell'esecuzione di OPT, otteniamo un costo strettamente inferiore, producendo una contraddizione sul fatto che OPT è ottimo: infatti, $i' \times f_i + i \times f_{i'} < i' \times f_{i'} + i \times f_i$.

Osserviamo che tale analisi della strategia MTF sfrutta la condizione che l'algoritmo OPT non può manipolare la lista una volta che abbia iniziato a gestire le richieste. È possibile estendere la dimostrazione del Teorema 5.5 al caso in cui *anche* OPT possa portare un elemento in cima alla lista.

In generale, il Teorema 5.5 non è più valido se permettiamo a OPT di manipolare la sua lista in altre maniere. Accedendo all'elemento in posizione `i`, l'algoritmo può per esempio riorganizzare la lista in tempo $O(i + 1)$: pensiamo a un impiegato con la sua pila disordinata di pratiche dove, pescata la pratica in posizione `i`, può metterla in cima alla pila ribaltando l'ordine delle prime `i` nel contempo. In tal caso, è possibile dimostrare che un algoritmo che adotta una tale strategia, denominato REV, ha un costo pari a $O(n \log n)$ mentre il costo di MTF risulta essere $\Theta(n^2)$, invalidando l'equazione (5.7) per `n` sufficientemente grande.

Tuttavia, MTF rimane una strategia vincente per organizzare le informazioni in base alla frequenza di accesso. L'economista giapponese Noguchi Yukio ha scritto diversi libri di successo sull'organizzazione aziendale e, tra i metodi per l'archiviazione cartacea, ne suggerisce uno particolarmente efficace. Il metodo si basa su MTF e consiste nel mettere l'oggetto dell'archiviazione (un articolo, il passaporto, le schede telefoniche e così via) in una busta di carta etichettata. Le buste sono mantenute in un ripiano lungo lo scaffale e le nuove buste vengono aggiunte in cima. Quando una busta viene presa in una qualche posizione del ripiano, identificata scandendolo dalla cima, viene successivamente riposta in cima dopo l'uso. Nel momento in cui il ripiano è pieno, un certo quantitativo di buste nel fondo viene trasferito in un'opportuna sede, per esempio una scatola di cartone etichettata in modo da identificarne il contenuto. L'economista sostiene che è più facile ricordare l'ordine temporale dell'uso degli oggetti archiviati piuttosto che la loro classificazione in base al contenuto, per cui il metodo proposto permette di recuperare velocemente tali oggetti dallo scaffale.

5.5 Tecniche di analisi ammortizzata

Le operazioni di unione e appartenenza su liste disgiunte e quelle di ricerca in liste ad auto-organizzazione non sono i primi due esempi di algoritmi in cui abbiamo applicato l'analisi ammortizzata. Abbiamo già incontrato un terzo esempio di tale analisi per valutare il costo delle operazioni di ridimensionamento di un array di lunghezza variabile nel Teorema 1.1.[3] Questi tre esempi illustrano tre diffuse modalità di analisi ammortizzata di cui diamo una descrizione utilizzando come motivo conduttore il problema dell'incremento di un contatore.

In tale problema abbiamo un contatore binario di `k` cifre binarie, memorizzate in un array `contatore` di dimensione `k` i cui elementi valgono `0` oppure `1`. In particolare, il valore del contatore è dato da $\sum_{i=0}^{k-1} (\texttt{contatore[i]} \times 2^i)$ e supponiamo che inizialmente esso contenga tutti `0`.

Come mostrato nel Codice 5.6, l'operazione di incremento richiede un costo in tempo pari al *numero di elementi cambiati* in `contatore` (righe 4 e 7), e quindi

[3] Nel Capitolo 2 abbiamo utilizzato questo risultato nell'analisi della complessità delle operazioni di inserimento e cancellazione nelle pile, nelle code e negli heap.

O(k) tempo al caso pessimo: discutiamo tre modi di analisi per dimostrare che il costo ammortizzato di una sequenza di $n = 2^k$ incrementi è soltanto O(1) per incremento.

ALVIE **Codice 5.6** Incremento di un contatore binario.

```
1  Incrementa( contatore ):          ⟨pre: k è la dimensione di contatore⟩
2    i = 0;
3    WHILE ((i < k) && (contatore[i] == 1) ) {
4      contatore[i] = 0;
5      i = i+1;
6    }
7    IF (i < k) contatore[i] = 1;
```

Il primo metodo è quello di **aggregazione**: conteggiamo il numero totale T(n) di passi elementari eseguiti e lo dividiamo per il numero n di operazioni effettuate. Nel nostro caso, conteggiamo il numero di elementi cambiati in contatore (righe 4 e 7), supponendo che quest'ultimo assuma come valore iniziale zero. Effettuando n incrementi, osserviamo che l'elemento contatore[0] cambia (da 0 a 1 o viceversa) a ogni incremento, quindi n volte; il valore di contatore[1] cambia ogni due incrementi, quindi n/2 volte; in generale, il valore di contatore[i] cambia ogni 2^i incrementi e quindi $n/2^i$ volte. In totale, il numero di passi è $T(n) = \sum_{i=0}^{k-1} n/2^i = \left(\sum_{i=0}^{k=1} 1/2^i\right) n < 2n$. Quindi il costo ammortizzato per incremento è O(1) poiché T(n)/n < 2. Osserviamo che abbiamo impiegato il metodo di aggregazione per analizzare il costo dell'operazione di unione di liste disgiunte.

Il secondo metodo è basato sul concetto di **credito** (con relativa metafora bancaria): utilizziamo un fondo comune, in cui depositiamo crediti o li preleviamo, con il vincolo che il fondo non deve andare mai in rosso (prelevando più crediti di quanti siano effettivamente disponibili). Le operazioni possono sia depositare crediti nel fondo che prelevarne senza mai andare in rosso per coprire il proprio costo computazionale: il costo ammortizzato per ciascuna operazione è il numero di crediti depositati da essa. Osserviamo che tali operazioni di deposito e prelievo di crediti sono introdotte solo ai fini dell'analisi, senza effettivamente essere realizzate nel codice dell'algoritmo così analizzato. Nel nostro esempio del contatore, partiamo da un contatore nullo e utilizziamo un fondo comune pari a zero. Con riferimento al Codice 5.6, per ogni incremento eseguito associamo i seguenti movimenti sui crediti:

1. preleviamo un credito per ogni valore di contatore[i] cambiato da 1 a 0 nella riga 4;
2. depositiamo un credito quando contatore[i] cambia da 0 a 1 nella riga 7.

Da notare che la situazione al punto 1 può occorrere un numero variabile di volte durante un singolo incremento (dipende da quanti valori pari a `1` sono esaminati dal ciclo); invece, la situazione al punto 2 occorre al più una volta, lasciando un credito per quando quel valore da `1` tornerà a essere `0`: in altre parole, ogni volta che necessitiamo di un credito nel punto 1, possiamo prelevare dal fondo in quanto tale credito è stato sicuramente depositato da un *precedente* incremento nel punto 2. Ogni operazione può essere dotata di `O(1)` crediti e quindi il costo ammortizzato per incremento è `O(1)`. Possiamo applicare il metodo dei crediti per l'analisi ammortizzata del ridimensionamento di un array a lunghezza variabile: ogni qualvolta che estendiamo l'array di un elemento in fondo, depositiamo `c` crediti per una certa costante $c > 0$ (di cui uno è utilizzato subito); ogni volta che raddoppiamo la dimensione dell'array, ricopiando gli elementi, utilizziamo i crediti accumulati fino a quel momento.

Infine, il terzo metodo è basato sul concetto di **potenziale** (con relativa metafora fisica). Numerando le n operazioni da 0 a $n-1$, indichiamo con Φ_{-1} il potenziale iniziale e con $\Phi_j \geq 0$ quello raggiunto dopo l'operazione j, dove $0 \leq j \leq n-1$. La difficoltà consiste nello scegliere l'opportuna funzione come potenziale Φ, in modo che la risultante analisi sia la migliore possibile. Indicando con c_j il costo richiesto dall'operazione j, il costo ammortizzato di quest'ultima è definito in termini della differenza di potenziale, nel modo seguente:

$$\hat{c}_j = c_j + \Phi_j - \Phi_{j-1} \tag{5.10}$$

Quindi, il costo totale che ne deriva è dato da $\sum_{j=0}^{n-1} \hat{c}_j = \sum_{j=0}^{n-1} (c_j + \Phi_j - \Phi_{j-1}) = \sum_{j=0}^{n-1} c_j + (\Phi_{n-1} - \Phi_{-1})$: utilizzando il fatto che otteniamo una somma telescopica per le differenze di potenziale, deriviamo che il costo totale per la sequenza di n operazioni può essere espresso in termini del costo ammortizzato nel modo seguente:

$$\sum_{j=0}^{n-1} c_j = \sum_{j=0}^{n-1} \hat{c}_j + (\Phi_{-1} - \Phi_{n-1}) \tag{5.11}$$

Nell'esempio del contatore binario, poniamo Φ_j uguale al numero di valori pari a `1` in `contatore` dopo il `(j+1)`-esimo incremento, dove $0 \leq j \leq n-1$: quindi, $\Phi_{-1} = 0$ in quanto il contatore è inizialmente pari a tutti `0`. Per semplicità, ipotizziamo che il contatore contenga sempre uno `0` in testa e, fissato il `(j+1)`-esimo incremento, indichiamo con ℓ il numero di volte che viene eseguita la riga 4 nel ciclo `WHILE` del Codice 5.6: il costo è quindi $c_j = \ell + 1$ in quanto ℓ valori pari a 1 diventano `0` e un valore pari a `0` diventa `1`. Inoltre, la differenza di potenziale $\Phi_j - \Phi_{j-1}$ misura quanti `1` sono cambiati: ne abbiamo ℓ in meno e `1` in più, per cui $\Phi_j - \Phi_{j-1} = -\ell + 1$. Utilizzando la formula (5.10), otteniamo un costo ammortizzato pari a $\hat{c}_j = (\ell + 1) + (-\ell + 1) = 2$. Poiché $\Phi_{n-1} \geq 0$ e $\Phi_{-1} = 0$, in base all'equazione (5.11) abbiamo che $\sum_{j=0}^{n-1} c_j \leq \sum_{j=0}^{n-1} \hat{c}_j \leq 2n$. Osserviamo che abbiamo utilizzato il metodo del potenziale per l'analisi della strategia MTF scegliendo come potenziale Φ_j il numero di inversioni rispetto alla lista gestita da OPT.

5.6 Esercizi

5.1 Mostrare che l'analisi al caso medio del *quicksort* randomizzato è $O(n \log n)$ anche dividendo il segmento `[sinistra ... destra]` in tre parti (invece che in quattro).

5.2 Estendere la `QuickSelect` randomizzata in modo da trovare gli elementi di rango compreso tra r_1 e r_2, dove $r_1 < r_2$. Studiare la complessità al caso medio dell'algoritmo proposto.

5.3 Modificare il Codice 5.2 in modo da restituire in un array tutti i predecessori sulle liste L_i della chiave data.

5.4 Dimostrare che la complessità dell'operazione di ricerca in una lista a salti non casuale è logaritmica nel numero degli elementi.

5.5 Scrivere lo pseudocodice che, prese due liste a salti, produce l'intersezione degli elementi in esse contenuti. Discutere la complessità dell'algoritmo proposto.

5.6 Descrivere una rappresentazione degli insiemi per il problema dell'unione di liste disgiunte che, per ogni insieme, utilizzi un albero in cui i soli puntatori siano quelli al padre.

5.7 Mostrare che, nonostante sia $costo(\text{MTF}) \leq 2 \times costo(\text{OPT})$, alcune configurazioni hanno $costo(\text{MTF}) < costo(\text{OPT})$ (prendere una lista di $m = 2$ elementi e accedere a ciascuno $n/2$ volte).

5.8 Consideriamo un algoritmo non in linea REV, il quale applica la seguente strategia ad auto-organizzazione per la gestione di una lista. Quando REV accede all'elemento in posizione i, va avanti fino alla prima posizione $i' \geq i$ che è una potenza del 2, prende quindi i primi i' elementi e li dispone in ordine di accesso futuro (ovvero il successivo elemento a cui accedere va in prima posizione, l'ulteriore successivo va in seconda posizione e così via). Ipotizziamo che $n = m = 2^k + 1$ per qualche $k \geq 0$, che inizialmente la lista contenga gli elementi $e_0, e_1, ..., e_{m-1}$ e che la sequenza di richieste sia $e_0, e_1, ..., e_{m-1}$, in questo ordine (vengono cioè richiesti gli elementi nell'ordine in cui appaiono nella lista iniziale). Dimostrate che il costo di MTF risulta essere $\Theta(n^2)$ mentre quello di REV è $O(n \log n)$. Estendete la dimostrazione al caso $n > m$.

5.9 Calcolare un valore della costante c adoperata nell'analisi ammortizzata con i crediti per il ridimensionamento di un array di lunghezza variabile, dettagliando come gestire i crediti.

5.10 Fornire un'analisi ammortizzata basata sul potenziale per il problema del ridimensionamento di un array di lunghezza variabile.