Capitolo 5
Casualità e ammortamento

Lucidi tratti da
P. Crescenzi · G. Gambosi · R. Grossi · G. Rossi
Strutture di dati e algoritmi
Progettazione, analisi e visualizzazione
Addison-Wesley, 2012
`http://algoritmica.org`

1. I lucidi sono utilizzabili dai soli docenti e se ne sconsiglia la distribuzione agli studenti: oltre al rischio di violare una qualche forma di copyright, il problema principale è che gli studenti studino in modo superficiale la materia senza il necessario approfondimento e la dovuta riflessione che la lettura del libro fornisce
2. Il simbolo [alvie] nei lucidi indica l'uso di ALVIE per visualizzare il corrispettivo algoritmo: per un proficuo rendimento dello strumento, conviene esaminare in anticipo la visualizzazione per determinare i punti salienti da mostrare a lezione (l'intera visualizzazione potrebbe risultare altrimenti noiosa)

Tempo medio di quicksort

L'algoritmo di ordinamento per distribuzione (o `QuickSort`) ha una complessità che dipende dall'ordine iniziale degli elementi:

- distribuzione iniziale degli elementi bilanciata: costo $O(n \log n)$
- distribuzione iniziale sbilanciata: costo $O(n^2)$

Se si considerano tutti i possibili array di ingresso, il tempo medio è $O(n \log n)$: la distribuzione sbilanciata compare raramente, tra tutti i possibili input.

Quicksort randomizzato

- Con la sua versione randomizzata, il tempo di esecuzione di `QuickSort` su un array in ingresso è indipendente dalla distribuzione degli elementi nell'array.Il tempo di esecuzione risulta determinato da una distribuzione casuale degli elementi dell'array, e quindi, im media $O(n \log n)$.
- Modifica all'algoritmo:riguarda la scelta del `pivot` che deve avvenire in modo aleatorio, equiprobabile e uniforme nell'intervallo $[\texttt{sinistra} \ldots \texttt{destra}]$.

L'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).

Quicksort randomizzato

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

[alvie]

La primitiva random() genera un valore reale r pseudocasuale appartenente all'intervallo $0 \leq r \leq 1$, in modo uniforme ed equiprobabile

Quicksort randomizzato

Il valore di `rango` restituito da `Distribuzione` è uniformemente distribuito tra le (equiprobabili) posizioni in [`sinistra...destra`].

Consideriamo l'intervallo [`sinistra...destra`] suddiviso in quattro **zone** di stessa lunghezza. Due eventi equiprobabili:

- `rango` ricade nella prima o nell'ultima zona: `rango` *esterno*. La distribuzione è sbilanciata: il costo medio in questo caso può risultare

$$T(n) = T(n-1) + c_1 n$$

- `rango` ricade nella seconda o nella terza zona: `rango` *interno*. La distribuzione è bilanciata (al peggio, $n/4$ elementi da una parte e $3n/4$) dall'altra: il costo medio in questo caso risulta al più

$$T(n) = T(n/4) + T(3n/4) + c_2 n$$

Quicksort randomizzato

Il costo medio in generale è

$$\begin{aligned} T(n) &\leq \frac{1}{2}\left(T(n-1)+c_1 n\right)+\frac{1}{2}\left(T\left(\frac{n}{4}\right)+T\left(\frac{3n}{4}\right)+c_2 n\right) \\ &< \frac{1}{2}T(n)+\frac{c_1}{2}n+\frac{1}{2}T\left(\frac{n}{4}\right)+\frac{1}{2}T\left(\frac{3n}{4}\right)+\frac{c_2}{2}n \end{aligned}$$

e quindi

$$T(n) < T\left(\frac{n}{4}\right)+T\left(\frac{3n}{4}\right)+cn$$

con $c = c_1 + c_2$

Quicksort randomizzato

Osserviamo che

$$
\begin{aligned}
T(n) &< T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + cn \\
&< T\left(\frac{1}{4}\frac{n}{4}\right) + T\left(\frac{3}{4}\frac{n}{4}\right) + \frac{1}{4}cn + T\left(\frac{1}{4}\frac{3n}{4}\right) + T\left(\frac{3}{4}\frac{3n}{4}\right) + \frac{3}{4}cn + cn \\
&= T\left(\frac{n}{4^2}\right) + 2T\left(\frac{3n}{4^2}\right) + T\left(\frac{3^2 n}{4^2}\right) + 2cn \\
&< T\left(\frac{n}{4^3}\right) + T\left(\frac{3n}{4^3}\right) + \frac{1}{4^2}cn + 2T\left(\frac{3n}{4^3}\right) + 2T\left(\frac{3^2 n}{4^3}\right) + 2\frac{3}{4^2}cn + \\
&+ T\left(\frac{3^2 n}{4^3}\right) + T\left(\frac{3^3 n}{4^3}\right) + \frac{3^2}{4^2}cn + 2cn \\
&= T\left(\frac{n}{4^3}\right) + 3T\left(\frac{3n}{4^3}\right) + 3T\left(\frac{3^2 n}{4^3}\right) + T\left(\frac{3^3 n}{4^3}\right) + 3n
\end{aligned}
$$

Quicksort randomizzato

In generale, si può verificare che alla profondità k di ricorsione si ha

$$T(n) < \sum_{i=0}^{k} \binom{k}{i} T\left(\frac{3^i n}{4^k}\right) + kcn$$

La profondità della ricorsione è limitata superiormente da

$$s = \lceil \log_{4/3} n \rceil = O(\log n)$$

per cui esistono $O(\log n)$ livelli, ognuno dei quali comporta un costo aggiuntivo $O(n)$. Da ciò deriva che

$$T(n) < \sum_{i=0}^{s} \binom{s}{i} T\left(\frac{3^i n}{4^s}\right) + scn$$

Quicksort randomizzato

Per definizione di s, si ha $\frac{4^s}{3^i} = 3^{s-i}n \geq n$, per cui $\frac{3^i n}{4^s} \leq 1$ e possiamo considerare una costante c' tale che

$$T\left(\frac{3^i n}{4^s}\right) = c'$$

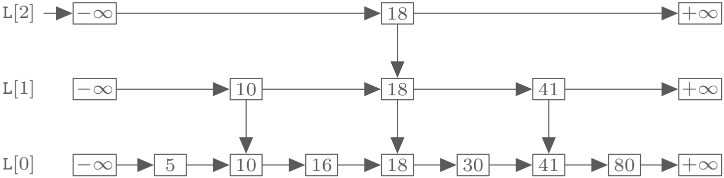
e quindi

$$T(n) < c' \sum_{i=0}^{s} \binom{s}{i} + scn = c'2^s + scn = O(n) + O(n \log n) = O(n \log n)$$

Liste a salti (**skip list**)

- Base: lista ordinata di $n+2$ elementi, $L_0 = e_0, e_1, \ldots, e_{n+1}$
  - livello 0 della lista a salti
  - primo e l'ultimo elemento hanno valori speciali, $-\infty$ e $+\infty$
  - sia ha sempre $-\infty < e_i < +\infty$, per $1 \leq i \leq n$
- per ogni elemento e_i di L_0 ($1 \leq i \leq n$, sia r_i la massima potenza di 2 che divide i ($i = 2^{r_i}$)
- r_i copie di e_i se $r_i > 0$, a livelli $\ell = 1, 2, \ldots, r_i$
- a livello ℓ, lista ordinata delle copie degli elementi e_i aventi $r_i \geq \ell$
- la copia di e_i a livello ℓ punta alla copia di livello inferiore $\ell - 1$
- il massimo livello (*altezza*) h della lista a salti è dato dal massimo valore di r_i incrementato di 1 e, quindi, $h = O(\log n)$.

Liste randomizzate e dizionari



Chiaramente, $L_\ell \subseteq L_{\ell-1} \subseteq \cdots \subseteq L_0$.

- L_0, contiene $n+2$ elementi
- L_1 contiene al più $2+n/2$ ordinati
- L_2 contiene al più $2+n/4$ ordinati
- L_ℓ contiene al più $2+n/2^\ell$ elementi

Il numero totale di copie presenti nella lista a salti è al più

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

Quindi, il numero totale di copie è $O(n)$.

Predecessori in liste randomizzate

- lista $L_\ell = e'_0, e'_1, \ldots, e'_{m-1}$ di elementi ordinati
- elemento x
- $e'_j \in L_\ell$ $(0 \leq j < m-1)$ è il **predecessore** di x (in L_ℓ) se e'_j e è il massimo tra gli elementi minori di x
- quindi, $e'_j \leq x < e'_{j+1}$

Il precedessore è sempre ben definito perché il primo elemento di L_ℓ è $-\infty$ e l'ultimo elemento è $+\infty$

Ricerca in liste randomizzate

```
ScansioneSkipList( k ):
  p = L[h];
  WHILE (p != null) {
    WHILE (p.succ.key <= k)
      p = p.succ;
    predecessore = p;
    p = p.inf;
  }
  RETURN predecessore;
```

⟨*pre: gli elementi* $-\infty$ *e* $+\infty$ *fungono da sentinelle*⟩

[alvie]

Simile alla ricerca binaria in array.

Inserimento in liste randomizzate

- troppo costoso se volessimo continuare a mantenere le proprietà della lista a salti
- potrebbe voler dire modificare le copie di tutti gli elementi che seguono la chiave appena inserita

Uso della casualità: 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.

- non garantisce la struttura perfettamente bilanciata della lista
- con alta probabilità mantiene un'altezza media logaritmica
- e quindi un tempo medio di esecuzione di una ricerca logaritmico

Inserimento in liste randomizzate

- la casualità può essere vista come l'esito di una sequenza di lanci indipendenti di una moneta equiprobabile
- sequenza di b lanci: generazione di una sequenza random di lunghezza b

```
InserimentoSkipList( k ):
  pred = [p_0, p_1, ..., p_h];
  FOR (r = 1; r <= h && random() < 0.5; r = r + 1)
    ;
  IF (r > h) {
    piu.chiave = +∞; piu.succ = piu.inf = null;
    meno.chiave = −∞; meno.succ = piu; meno.inf = L[h];
    pred[h+1] = L[h+1] = meno; h = h + 1;
  }
  FOR (i = 0, ultimo = null; i <= r; i = i + 1) {
    nuovo.chiave = k; nuovo.succ = pred[i].succ; nuovo.inf = ultimo;
    ultimo = pred[i].succ = nuovo;
  }
```

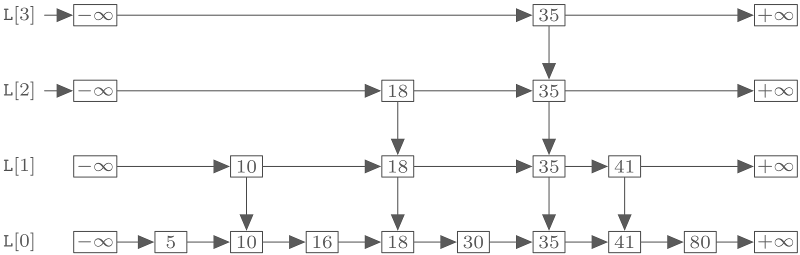
[alvie]

Inserimento in liste randomizzate

- identificazione dei predecessori $p_0, p_1, \ldots, p_h$ di k
- memorizzazione dei predecessori in un vettore `pred`
- sequenza di lanci di moneta: stop se otteniamo 1 oppure dopo h lanci; r è il numero di lanci eseguiti
- se abbiamo eseguito $r = h + 1$ lanci, incremento dell'altezza: nuova lista L_{h+1} composta dalle chiavi $-\infty$, k e $+\infty$
- r copie di k inserite nelle liste $L_0, L_1, L_2, \ldots, L_r$ dopo i predecessori $p_0, p_1, \ldots, p_r$

Inserimento in liste randomizzate

Inserimento di $k = 35$



Ricerca e inserimento in liste randomizzate

La complessità media delle operazioni di ricerca e inserimento su una lista a salti è $O(\log n)$.

Liste disgiunte

Utili per rappresentare partizioni degli elementi di un insieme di m elementi: una lista corrisponde ad un sottoinsieme nella partizione.

- si vuole gestire una sequenza arbitraria S di operazioni di unione e appartenenza sull'insieme di liste
- le liste sono *disgiunte*: l'intersezione di due liste qualunque è vuota
- inizialmente, m liste, ciascuna formata da un solo elemento
- operazione di unione: prende due delle liste attualmente disponibili e le unisce
- operazione di appartenenza: stabilisce se due elementi appartengono alla stessa lista

Problema noto come **union-find**.

Liste disgiunte: soluzione banale

Si mantengono i riferimenti al primo e all'ultimo elemento di ogni lista

- operazione di unione in tempo costante (concatenazione delle due liste)
- appartenenza richiede tempo $O(m)$ nel caso pessimo (scansione delle liste)
- tempo $O(nm)$ per eseguire una sequenza di n operazioni (unione e/o appartenenza)

Liste disgiunte: soluzione alternativa

Sequenza arbitraria S di n operazioni delle quali n_1 sono operazioni di unione e n_2 sono operazioni di appartenenza.

Tempo totale $O(n_1 \log n_1 + n_2) = O(n \log n)$: molto minore di $O(nm)$.

- Ogni lista rappresentata mediante:
  - riferimento all'inizio
  - riferimento alla fine
  - lunghezza
- Associato ad ogni elemento: riferimento alla lista di appartenenza

Liste disgiunte: soluzione alternativa

Appartenenza: si confrontano i riferimenti associati ai due elementi

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

[alvie]

Liste disgiunte: soluzione alternativa

Unione: viene cambiato il riferimento associato a tutti gli elementi della lista più corta, in modo da puntare alla lista più lunga

```
Unisci( x, y ):                                   ⟨pre: x, y non vuoti e x.lista ≠ y.lista⟩
  IF (x.lista.lunghezza <= y.lista.lunghezza) {
    corta = x.lista;
    lunga = y.lista;
  } ELSE {
    corta = y.lista;
    lunga = x.lista;
  }
  z = corta.inizio;
  WHILE (z != null) {
    z.lista = lunga;
    z = z.succ;
  }
  lunga.fine.succ = corta.inizio;
  lunga.fine = corta.fine;
  lunga.lunghezza = corta.lunghezza + lunga.lunghezza;
```

[alvie]

Liste disgiunte: soluzione alternativa

Inizializzazione: si crea una lista per ogni elemento

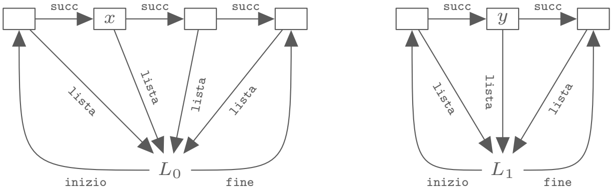
```
Crea( x ):                                   ⟨pre: x non vuoto⟩
  lista.inizio = lista.fine = x;
  lista.lunghezza = 1;
  x.lista = lista;
  x.succ = null;
```

[alvie]

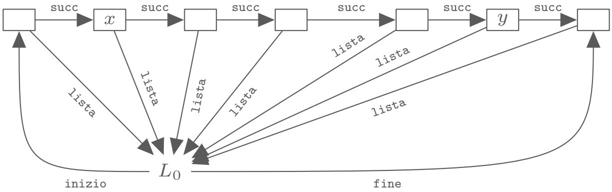
Liste disgiunte: soluzione alternativa

- Il costo di ciascuna operazione `Crea` e `Appartieni` è chiaramente $O(1)$ nel caso pessimo
- L'esecuzione di una sequenza di $n_1 < m$ operazioni `Unisci` richiede tempo $O(n_1 \log m)$.
  - Caso pessimo: il tempo di esecuzione di `Unisci` è proporzionale al numero di riferimenti modificati
  - Quanti riferimenti sono modificati nella sequenza? Equivale a stimare quante volte un elemento può cambiare lista, nel corso della sequenza.
  - In una operazione di `Unisci`, se un elemento cambia lista va a finire in una lista almeno doppia.
  - Inizialmente, un elemento si trova in una lista di lunghezza 1. Alla fine in una di lunghezza al più $m > n_1$.
  - Ogni elemento cambia lista al più $O(\log m)$ volte, per cui i cambiamenti totali di lista sono $O(n_1 \log m)$
- Il costo ammortizzato per operazione delle `Unisci` è $O(\log m)$

Liste disgiunte: soluzione alternativa



Unione tra L_0 e L_1



Liste ad auto-organizzazione

- Ricerca di una chiave k in una lista mediante scansione sequenziale
- La lista non è mantenuta ordinate in base alle chiavi di ricerca: si vogliono sfruttare proprietà ulteriori (località di accessi)
- Operazioni di modifica della struttura della lista eseguite in corrispondenza alle scansioni sequenziali

Liste ad auto-organizzazione

- **Principio di località temporale**: se si accede a un elemento in un dato istante, è molto probabile che si accederà allo stesso elemento in istanti immediatamente (o quasi) successivi.
- Sembra naturale riorganizzare la lista in modo da tenere nelle prime posizioni gli elementi cui si è acceduto di recente. che possiamo *riorganizzare* proficuamente gli elementi della lista dopo aver eseguito la loro scansione.
- Struttura di dati ad **auto-organizzazione**
- **Move-to-front** (MTF): la più diffusa ed efficace strategia di auto-organizzazione. L'elemento acceduto viene spostato alla prima posizione della lista
- MTF effettua ogni ricerca senza conoscere le ricerche che dovrà effettuare in seguito: algoritmo **in linea** (on-line)

Ricerca in MTF

```
MoveToFront( a, k ):
  p = a;
  IF (p == null || p.dato == k) RETURN p;
  WHILE (p.succ != null && p.succ.dato != k)
    p = p.succ;
  IF (p.succ == null) RETURN null;
  tmp = p.succ;
  p.succ = p.succ.succ;
  tmp.succ = a;
  a = tmp;
  RETURN a;
```

[alvie]

Valutazione di MTF

- Termine di paragone: algoritmo OPT
  - **fuori linea** (*offline*)
  - prende le sue decisioni conoscendo tutte le richieste che perverranno
- Le prestazioni dei due algoritmi sono confrontate rispetto al loro costo, definito come la somma dei costi delle singole operazioni
- Costo operazione = numero di elementi della lista attraversati. Accedere all'elemento in posizione i-esima ha costo i

Valutazione di MTF

Regole di azione di OPT

- Inizialmente, esamina tutte le richieste: permuta gli elementi della lista in modo da minimizzare il costo futuro
- Successivamente, per ogni richiesta, accede agli elementi scandendo la lista, senza modificarla

Assumiamo che OPT e MTF partano con gli elementi nella lista ordinati allo stesso modo, e che gli elementi della lista non cambino.

Valutazione di MTF

- Sequenza arbitraria di n operazioni di ricerca su una lista di m elementi
- Operazioni enumerate da 0 a $n-1$ in base al loro ordine di esecuzione
- Sia c_j la posizione dell'elemento acceduto da MTF alla j-esima operazione j $(0 \leq j \leq n-1)$, e quindi il costo di tale operazione per MTF
- Sia c'_j la posizione dell'elemento acceduto da OPT alla j-esima operazione j $(0 \leq j \leq n-1)$, e quindi il costo di tale operazione per OPT

I costi dei due algoritmi sono quindi

$$\text{costo}(\mathsf{MTF}) = \sum_{j=0}^{n-1} c_j$$

$$\text{costo}(\mathsf{OPT}) = \sum_{j=0}^{n-1} c'_j$$

Valutazione di MTF

Se MTF e OPT operano su liste uguali, allora

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

quando le liste di partenza sono uguali.

- Se due elementi $\{x, y\}$ compaiono in ordine diverso nella lista di MTF e in quella di OPT si ha una **inversione**
- Φ_j: numero di inversioni tra le due liste dopo che è stata eseguita l'operazione j. Si ha
$$0 \leq \Phi_j \leq \frac{m(m-1)}{2}$$
per $0 \leq j \leq n-1$

Valutazione di MTF

Mostriamo che

$$c_j + \Phi_j - \Phi_{j-1} \leq 2c'_j$$

e quindi, ponendo $\Phi_{-1} = 0$, che

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

Sia k l'elemento acceduto in seguito alla j-esima operazione.

Valutazione di MTF

Lista di MTF:

- Se $c_j = 0$, MTF lascia la lista invariata, quindi $\Phi_j = \Phi_{j-1}$
- Altrimenti, sia k' un elemento che precede k nella lista di MTF. Dopo l'operazione, $\{k, k'\}$ è un'inversione se e solo se non lo era prima
- Sia $f \leq c_j$ il numero di elementi (incluso k) corrispondenti a inversioni e sia $g = c_j - f$ il numero di quelli che non danno inversioni
- Dopo l'operazione, il numero di inversioni diventa $\Phi_j = \Phi_{j-1} - f + g$
- Quindi, $c_j + \Phi_j - \Phi_{j-1} = i - f + g = (f + g) - f + g = 2g$

Valutazione di MTF

Lista di OPT:

- Nella lista di MTF ci sono g elementi che precedeno k senza inversioni
- Quindi, nella lista di OPT ci sono almeno g elementi che precedono k: $c'_j \geq g$
- Quindi, $c_j + \Phi_j - \Phi_{j-1} = 2g \leq 2c'_j$

Tecniche di analisi ammortizzata: aggregazione

Conteggio del numero totale $T(n)$ di passi elementari eseguiti e divisione per il numero n di operazioni effettuate.

Tecniche di analisi ammortizzata: crediti

- Fondo comune, in cui si depositano o si prelevano crediti
- Il numero di crediti depositati deve essere sempre non negativo
- Le operazioni possono sia depositare crediti nel fondo che prelevarne, per coprire il proprio costo computazionale
- il costo ammortizzato per ciascuna operazione è il numero di crediti depositati da essa

Tecniche di analisi ammortizzata: potenziale

- Definizione di una opportuna funzione **potenziale**
- Φ_{-1}, potenziale iniziale, e $\Phi_j \geq 0$ potenziale dopo l'operazione j, dove $0 \leq j \leq n-1$
- Il costo ammortizzato $\hat{c}_j$ della j-esima operazione è definito in termini di costo effettivo c_j e di differenza di potenziale:

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

- Per il costo ammortizzato totale:

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

- Il costo effettivo totale può essere espresso in termini del costo ammortizzato come:

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

se $\Phi_{n-1} \leq \Phi_{-1}$