Ordinameto

Luca Tagliavini

March 25-April 6, 2021

Contents

1 Ordinamento

1.1 Definizione formale

- Consideriamo un array di n numeri $v[1], v[2], \ldots, v[n]$.
- Vogliamo trovare una permutazione $p[1], p[2], \ldots, p[n]$ degli interi $1, \ldots, n$ tale che $v[p[1]] \leq v[p[2]] \leq \ldots \leq v[p[n]]$.

  Esempio con $v = \{7, 32, 88, 21, 92, -4\}$
  $p = \{6, 1, 4, 2, 3, 5\}$
  dove $p[i]$ indica la posizione nell'array di inputi dell'i-esimo elemento nella versione ordinata.
  $v[p[]] = \{-4, 7, 21, 32, 88, 92\}$

1.1.1 Generalizzazione

Possiamo astrarre dai numeri pensando a un array di n elmenti tali per cui ogni elemento ha una *chiave*, le quali sono confrontabili tra di loro, e un *contentuo* arbitrario che contiene in valore vero e proprio del dato.

Vogliamo permutare l'array in modo che le chiavi siano ordinate tra loro.

1.2 Syllabus

- Si definisce un algoritmo di ordinamento *in loco* quando esso muta direttamente l'array che gli viene dato in pasto, senza utilizzare un secondo array di appoggio. Possiamo dunque dire che **un algoritmo ordina in loco se usa una quantita' costante di memoria aggiuntiva**.
- Si parla di ordinamento *stabile* se l'algoritmo mantiene l'ordine di elementi con la stessa chiave. (i.e. Ho due elementi con chiave 2, un algoritmo si dice stabile sse in ouput ritrovo i due elementi con chiave 2 nello stesso ordine in cui li ho dati in pasto all'algoritmo).

1.2.1 Nota sulla stabilita'

Si noti che e' *sempre* possibile rendere un algoritmo stabile. Basta usare come chiave di ordinamento la coppia $(chiave, i_{unordered})$ e fare un confronto sia tra il primo valore della chiave che il secondo. (i.e. nell'esempio precedente, avremo come chiavi $(2, 0)$ e $(2, 1)$, e sara' quindi facile mantere l'ordinamento) Useremo: $(k_1, p_1) < (k_2, p_2)$

- $(k_1 < k2)$, oppure
- $(k_1 = k2)$ and $(p_1 < p_2)$

2 Algoritmi di ordinamento (incrementali)

Questi algoritmi partono da un prefisso $A[1..k]$ ordinato, e ne estendono la parte ordinata di un elemento: $A[1..k+1]$

- selection sort
  cerca il minimo in $A[k+1..n]$ e spostalo in posizione $k+1$
- insertion sort
  inserisce l'elemento $A[k+1]$ nella posizione corretta all'interno del prefisso gia' ordinato $A[1..k]$

2.1 Selection sort (NON stabile)

Nota: le noste posizioni vanno da $1..n$

- cerco il minimo in $A[1..n]$ e lo scambio con $A[1]$
- cerco il minimo in $A[2..n]$ e lo scambio con $A[2]$
- ...
- cerco il minimo in $A[k..n]$ e lo scambio con $A[k]$
- ...
- quando ho ordinato $n-1$ ho la garanzia che l'elemento in n sia ordinato

```
public static void selection_sort(Comparable A[]) {
  for(int k = 0; k < A.length - 1; k++) {
    int min = k;
    // troviamo il minimo
    for(int j = k + 1; j < A.length; j++)
      if(A[j].compareTo(A[m] < 0)
        min = j;

    // cambiamo il k-esimo elemento con il minimo
    if(min != k) {
      Comparable tmp = A[m];
      A[m] = A[k];
      A[k] = tmp;
    }
  }
}
```

Complessita': $O(n^2)$

2.2 Selection sort (stabile)

Nota: le noste posizioni vanno da $1..n$

- cerco la posizione di A in cui inserire $A[2]$
- cerco la posizione di A in cui inserire $A[3]$

- ...
- cerco la posizione di A in cui inserire $A[k]$
- ...
- fino alla fine

```
public static void insertion_sort(Comparable A[]) {
  for(int k = 1; k < A.length; k++) {
    int j;
    Comparable x = A[k];
    // trovo la posizione j in cui inserire A[k]
    for(int j = k + 1; j < k; j++)
      if(A[j].compareTo(x > 0) break;

    // spostiamo A[j..k-1] in A[j+1..k]
    if(j < k) {
      // shift
      for(int t = k; t > j; t--)
        A[t] = A[t-1];

      A[j] = x;
    }
  }
}
```

Complessita': $\Theta(n^2)$

2.3 Bubble sort

Esegue una serie di scansioni sull'array. Ad ogni passata controlla una coppia di elementi dell'array e li inverte se la coppia non e' ordinata. Si continua con questa tecnica fino a quando non si giunge ad incontrare una coppia ordinata, il che significa che l'intero array e' stato ordinato.

```
public static void bubble_sort(Comparable A[]) {
  for(int i = 1; i < A.length; i++) {
    boolean swapped = false;
    for(int j = 1; j <= A.lenth - i; j++) {
      if(A[j-1].compareTo(A[j]) > 0) {
        Comparable tmp = A[j-1];
        A[j-1] = A[j];
        A[j] = tmp;
        swapped = true;
      }
    }
    // interrompiamo l'algoritmo quando non e' avvenuto uno scambio
       ,
    // il che indica che l'array e' ora ordinato.
    if(!swapped) break;
  }
}
```

Complessita': $\Theta(n^2)$

Il bubble sort ha la proprieta' di avere tempo di esecuzione "naturale", ossia *tende* a variare in base al disordine dell'array. Tuttavia, questo non e' sempre vero, prendendo ad esempio un input come $\{2, 3, 4, 5, 6, 7, 8, 9, 1\}$.

3 Algoritmi divide-et-impera

La tecnica divide et impera (dal latino "dividi e conquista") consistono in due fasi:

- *divide*: comporre in problema in sottoproblemi dello stesso tipo
- *impera*: combinare le soluzioni parziali dei sottoproblemi piu' semplici in una soluzione per il problema completo.

3.1 Quick sort

Algoritmo ricorsivo divide-et-impera: si sceglie un elemento x che chiamereo *pivot*, e partizioneremo il vettore in due parti considerando gli elementi $\leq x$ e quelli $> x$. Ordiniamo poi i due gruppi in modo ricorsivi, al che li uniamo mettendo in mezzo il pivot tra i due insiemi ordinati.

Implementazione: avremo in input un array $A[1..n]$, ed useremo indici i e f. L'indice i partira' dalla sinistra dell'array (pos. 0) e andra' avanti proseguendo per numeri $<$ *pivot*, mentre f partira' dalla destra (pos. n-1) e andra' avanti per numeir $>$ *pivot*. Quando entrambi gli iteratori si sono fermati possiamo scambiare le due cellette $A[i]$ con $A[f]$. Si ripete ricorsivamente fino a quando i e f non si toccano, al che avremo diviso l'array in due sezioni, una con tutti valori minoiri del pivot, e l'altra con tutti valori magigori del pivot. A questo punto si puo' svolgere l'operazione identica in modo ricorsivo sulle sottoparti per ottenere un array interamente ordinato e poi rinuire i frammenti e il pivot.

Vediamone il codice:

```
public static void quick_sort(Comparable A[]) {
  // impostiamo i e f ai valori di default
  quick_sort_rec(A, 0, A.length-1);
}

public static void quick_sort_rec(Comparable A[], int i, int f) {
  // caso base:
  // se i e f si toccano o si superano abbiamo ordinato le due
      sezioni
  if(i >= f) return;

  // caso ricorsivo:
  int m = partition(A, i, f); // m e' la posizione del pivot
  // ordino i sottosegmenti
  quick_sort_rec(A, i, m-1);
  quick_sort_rec(A, m+1, f);
}

public static int partition(Comparable A[], int i, int f) {
  // sup fuori dall'array poiche' usiamo do-while e quindi il --
```

```
  // viene fatto prima della guardia
  int inf = i, sup = f+1;
  Comparable tmp, x = A[i]; // x = pivot = primo elemento

  while(true) {
    do {
      inf++;
      // inf <= f per non far uscire inf dall'array
    } while(inf <= f && A[inf].compareTo(x) <= 0);
    do {
      sup--;
      // non controlliamo (sup > 0) poiche' siamo sicuri che quando
           arriveremo
      // in posizione 0 A[sup = 0] == x e quindi il ciclo si
          fermera'
      // insomma: non c'e' rischio che sup vada fuori dall'array(<
          0)
    } while(A[sup].compareTo(x) > 0);
    if(inf < sup) {
      tmp = A[inf];
      A[inf] = A[sup];
      A[sup] = tmp;
    } else break;
  }

  // sup e' la posizione in cui abbiamo messo il nostro pivot
  tmp = A[i];
  A[i] = A[sup];
  A[sup] = tmp;
  return sup;
}
```

3.1.1 Analisi del costo

Costo di partition: $\Theta(f - i)$
Costo di quicksort: strettamente dipendente dal partizionamento *partizionamento peggiore*: Dato un array di dimensione n, il caso peggiore avviene quando la funzioe di partizione lo separa in una array di dimensione 0 e uno di dimensione $n - 1$. Possiamo poi risolvere la seplice equazione con la tecnica dell'iterazione.

$$T(n) = T(n-1) + T(0) + n = T(n-1) + n = \Theta(n^2)$$

partizionamento migliore: Dato un array di dimensione n, il caso migliore avviene quando ogni sottoinsieme creato da partition ha grandezza $n/2$. Il che genera la seguente equazione semplificabile tramite il *master theorem*.

$$T(n) = 2T(n/2) + n = \Theta(n \log_2 n)$$

caso medio: Assumendo che tutti i partizionamenti siano equiprobabili ($/frac1n$) possiamo scrivere:

$$T(n) = \sum_{a=0}^{n-1} \frac{1}{n}(n-1-T(a)) + T(n-a-1)$$

Ricordando che $a+b = n-1$ (a, b sono le lunghezze delle due sottosequenze generate da partition) possiamo vedere che $T(a) = T(n-a-1)$, quindi riscriviamo la formula in modo semplificato:

$$T(n) = n - 1 + \frac{2}{n}\sum_{a=0}^{n-1} T(a)$$

Possiamo dimostrare questa formula per induzione.

3.1.2 Dimostrazione del caso medio

Theorem: la relazione

$$T(n) = n - 1 + \frac{2}{n}\sum_{a=0}^{n-1} T(a)$$

ha come soluzione $T(n) = O(n \log_2 n)$

Proof: mostriamo per induzione che $T(n) \leq \alpha(n \ln n)$

$$T(n) \leq \alpha(n \ln n)$$
$$n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \leq n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} \alpha i \ln i$$

porto fuori l'α e posso mettere $i = 2$ per i valori che assume il logaritmo

$$n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \leq n - 1 + \frac{2\alpha}{n}\sum_{i=0}^{n-1} i \ln i$$

3.1.3 Analisi del costo randomizzato

Usiamo un valore (pseudo)random per la scelta del pivot, modificando in modo appropriato le nostre formule: supponiamo di avere un insieme $\mathcal{R}$ di numeri random e un generico input, allora avremo:

$$T_{exp}(I) = \sum_{R \in \mathcal{R}} P(R) \cdot T(I, \underline{R})$$

Ora vogliamo calcolare il tempo atteso (expected) per un input di grandezza n, e la formula verra' dunque cambiata in:

$$T_{exp}(n) = \max_{I \in \mathcal{I}_n} T_{exp}(I)$$

Poiche' applichiamo sempre l'assunzione che ogni bilanciamento ha la stessa equiprobabilita', anche la versione randomizzata dell'algoritmo mantiene la stessa dimostrazione e calcolo e dunque complessita' del tempo di esecuzione del caso average: $O(n \log_2 n)$

3.2 Merge sort

Altro algoritmo *divide et impera* sviluppato da Neumann per coprire la falla del Quicksort nel caso di massimo sbilanciamento. Ha pensato di risolverlo dividendo sempre l'array in due parti larghe $n/2$, ordinarle ricorsivamente e e infine unirle.

L'algoritmo, sviluppato per chiamate ricorsive, si puo' dividere in due fasi:

- *split*: dividiamo progressivamete l'array e i sottoarray fino ad avere solo array di dimensione 1.
- *merge*: a questo punto le sottosequenze ottenute di dimensione 1 sono facili da unire, unite queste venogno unite quelle generate di dimensioni due, e cosi' via fino a ricomporre l'array finale.
  Un modo semplice per fondere due gruppi e' tramite il confronto del primo e ultimo elemento delle due sottosequenze. Chiamiamo la prima sottosequenza A e la econda B. Se $A.last \leq A.fist$ allora li uniamo come $A + B$, altrimenti come $B + A$.

3.3 Heap sort

Useremo la struttura dati *heap* per tenere traccia dei numeri maggiori nel nostro array, che ha un costo di ricerca: $\log_2 n$ ed e' quindi facile capire che facendo una passata su n elementi dell'array dove ad ogni chiamata chiediamo alla heap l'elemento piu' grande da mettere otterremo n volte la chiamata $log_2 n$. Questo ci da un tempo di ordinamento di $n \log_2 n$.

Analizziamo prima la struttura Heap, in particolare la *max heap* (heap che ordina per il massimo).

3.3.1 (max) Heap

Una heap e' un albero binario con due importanti proprieta':

- e' un *albero binrio perfetto*, ossia tutte le foglie hanno la stessa altezza, il che gli conferisce un'altezza $h = \log_2 n$. Il numero di nodi e' invece $n = 2^{h+1} - 1$.

- e' un *albero binario completo*, ossia tutte le foglie hanno profondita' h o al piu' $h-1$, dove tutti i nodi al livello h sono spostati a sinistra.
  Oltretutto, tutti i nodi hanno grado 2 (sia figlio destro che sinistro) tranne al massimo un nodo che non rispetta questa regola.

Grazie a queste proprieta' queste heap possono essere rappresentate tramite un array dove una serie di posizioni continue corrispondono a un livello dell'albero.

Un albero *max heap* e' un particolare albero heap dove a ogni nodo i viene associato un valore $A[i]$ dove A e' l'array in input da ordinare. L'albero fa poi valere la seguente legge $A[parent(i)] \geq A[i]$, ossia che il padre ammette valori solo maggiori o uguali a quelli dei figli. Dunque, il valore *massimo* dell'array si trovera' alla radice dell'albero.

Le funzionalita' di maggiore rilevanza nell'implementazione di un max heap saranno essenzialmente due:

- *fix_heap*: Problema: abbiamo un albero heap corretto, ma la radice contiene un valore errato. Bisogna riposizionarla.
  Soluzione: Si confronta il nodo bacato con i figli. Se il nodo errato risulta minore del figlio con valore maggiore, si esegue uno scambio tra i due e si svolge la chiamata ricorsiva sul nodo fino a quando non si raggiunge un caso base (niente figli, la radice e' diventata una foglia) o fino a quando il nodo figlio maggiore e' minore del nodo bacato. A questo punto l'albero risultera' riordinato. Il numero massimo di scambi da esegiure e' limitato alla profondita' dell'albero, ovver $O(\log_2 n)$.

- *heapify*: Problema: trasformare un array in input in una struttura dati arborescente del tipo max_heap.
  Soluzione: scompone il problema in due sottochiamate ricorsive in cui si creano i due figli del "nodo" (stiamo sempre lavorando con heap tramite array) e poi si ordina la radice che si trovera' in posizione $2^i - 1$ in modo da avere un alberocorretto.

4 Limite inferiore alla complessita' del problema dell'ordinamento

Dobbiamo anzitutto fare alcune assunzioni su alcuni algoritmi astratti:

- prendiamo un algoritmo X basato su confronti che itera su un array in input di valori distinti (una volta dimostrato varra' per il caso generale).

- l'algoritmo X puo' essere rappresentato tramite un *albero di decisione*, un albero binario che rappresenta i confronti tra gli elementi.

Ecco un albero di decisione d'esempio con $n = 3$. Ogni *foglia* rappresenta una permutazione data dall'algoritmo dopo che ha fatto una serie di confronti

Proprieta':

- il *cammino radice-foglia* in un albero di decisione rappresenta la sequenza di confronti eseguiti dall'algoritmo.
- useremo dunque un albero di decisione associato all'algoritmo per capirne massima profondita' e il cammino piu' breve per determinarne il limite minore.

4.0.1 Lemma 1

Teorema: Un albero di decisione per l'ordinamento di n elementi contiene *almeno* $n!$ numero di foglie, ovvero $\#foglie \geq n!$.

NOTA: si usa *almeno* poiche' algoritmi non ottimizzati varanno percorsi duplicati.

Dimostrazione: Ogni foglia corrisponde a una possibile soluzione del problema. Ogni soluzione consiste in una possibile permutazione, che varia in grandezza da $n, n-1, n-2, \ldots, 1$. Il numero di permutazioni e' dunque $n \cdot (n-1) \cdot \ldots \cdot 1$.

4.0.2 Lemma 2

Teorema: Sia T un albero binario in cui ogni nodo interno ha esattamente 2 figli e sia k il numero delle sue foglie. L'altezza dell'alberno e' *almeno* $log_2 k$.

Dimostrazione (per induzione su n, numero di nodi):

- Albero con un solo nodo: $n = 1$. L'altezza e' $h(1) = 0 \geq \log_2 1 = 0$
- Albero con due sottoalberi: k_1 numero nodi in h_1 e k_2 numero nodi in h_2. Supponiamo $k_1 > k_2$. La profondita' dell'albero e' data da:

$$\begin{aligned} h(k_1 + k_2) = 1 + \max\{h(k_1), h(k_2)\} &\geq \log_2(k_1 + k_2) \\ 1 + h(k_1) &\geq \log_2(k_1 + k_2) \\ \text{per l'ipotesi induttiva} & \\ \log_2(2) + \log_2(k_1) &\geq \log_2(k_1 + k_2) \\ \log_2(2 \cdot k_1) &\geq \log_2(k_1 + k_2) \end{aligned}$$

ovvio poiche' ho assunto $k_1 > k_2$. □

4.1 Teorema del limite inferiore per l'ordinamento

Teorema: il numero di confronti necessari per ordinare n eleenti nel caso peggiore e' $\Omega(n \log_2 n)$

Sfrutteremo l'*approssimazione di Stirling*:

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$$

Diomstrazione: Per lemma 1 e lemma 2 abbiamo che un albero n ha $n!$ foglie e dunque altezza $\Theta(\log_2 n!)$. Ora con Sterling possiamo fare:

$$\begin{aligned}\Omega(\log_2 n!) &= \Omega(\log_2(\sqrt{2\pi n}(\frac{n}{e})^n)) \\ &= \Omega(\log_2(\sqrt{2\pi})) + \Omega(\log_2(n^{1/2})) + \Omega(\log_2(\frac{1}{e})^n) + \Omega(\log_2(n^n))\end{aligned}$$

il primo e' costante, lo buttiamo via

$$= \Omega(\frac{1}{2}\log_2(n)) + \Omega(n\log_2\frac{1}{e}) + \Omega(n\log_2 n)$$

prendiamo l'Ω piu' grande

$$= \Omega(n\log_2 n)$$

5 Ordinamento lineare

In casi particolari siamo in grado di eseguire gli ordinamenti in tempo lineare. Vedremo tre esempi di algoritmi che hanno questa proprieta'

5.1 Counting Sort

Funziona su un array $A[0..n-1]$ con valori che vanno ad $[0..k-1]$. Dove ogni valore puo' comparire zero o piu' volte. L'algoritmo costruisce un array $Y[0..k-1]$ $Y[i]$ conta il numero di volte in cui il valore i compare in A. Infine ricolloco i valori cosi' ottenuti in A. Eccone una possibile implementazione:

```
public static void counting_sort<T = number>(T[] A, int k) {
  T[] Y = new T[k];
  int j = 0;
  for(int i = 0; i < k; i++) Y[i] = 0;
  for(int i = 0; i < A.length; i++) Y[A[i]]++;
  for(int i = 0; i < k; i++) {
    while(Y[i] > 0) {
      A[j++] = i;
      Y[i]--;
    }
  }
}
```

Costo: $O(\max\{n,k\}) = O(n+k)$. Se poi $k = O(n)$, allora il costo e' $O(n)$.

5.2 Bucket Sort

Se i dati in input non sono numeri, ma valori dai quali si puo' estrarre una chiave, posso usare questo algoritmo, che funziona in modo simile a counting. Per ogni possibile valore dentro al nostro array di chiavi abbiamo una lista con tutte le occorrenze di oggetti con quella chiave. Queste strutture a forma di lista vengono denominate "bucket"s. Per ordinare la lista leggo poi l'array, leggendo ogni lista dalla testa alla coda. NOTA: facendo gli inserimenti in coda si mantiene l'ordine logico originale e l'algoritmo diventa stabile.

```
ALGORITMO bucket_sort(arrray X[], int k)
  Y <- bucket[k]
  for i := 0 until k do
    Y[i] = null
  endfor
  for i := 0 until n do
    append bucket(X[i]) to Y[chiave(i)]
  endfor
  j := 0
  for i := 0 until k do
    for it := Y[i]; it = it.next; until it = nullptr
      cpy unbucket(it) in X[j]
    endfor
  endfor
END
}
```

5.3 Radix Sort

Il bucket sort e' geniale ma a volte il valore di k e' troppo grande e la memoria necessaria ne rende inpratico l'utilizzo. Ad esempio se dovessimo ordinare numeri con 4 cifre decimali avremo $k = n + 1000$. Appena $n + 1000 \leq n \log n$ si ha che bucket sort e' sconveniente. L'idea do radix sort e' di sfruttare algoritmi convenienti in certi casi come bucket sort e farli lavorare proprio in questi ambienti. Per risolvere il problema dell'esempio precedente potremo usare ogni cifra decimale come chiave di bucket sort. Prendendo una serie di cifre decimali alla volta, da quelle meno significative a quelle piu' significative si ha un sistema ordinato con tempo lineare. Perche' questo sia fattibile bucket sort deve essere *stabile*, e sappiamo che lo e' quando ai bucket si aggiunge in coda.