Chapter 3

Algoritmi su sequenze

3.1 Ordinamento di sequenze

L'ordinamento di insiemi di dati è un processo di fondamentale importanza per molte applicazioni informatiche. In primis, ovviamente, ci sono quelle applicazioni, di tipo interattivo, il cui fine stesso è il recupero di informazioni da collezioni di dati, spesso di grandi dimensioni. Un esempio è la consultazione di elenchi telefonici. Tuttavia, ordinare i dati in input è sovente il primo passo per risolvere problemi che con l'ordinamento in se non hanno nulla a che vedere; in questo caso l'ordinamento è semplicemente un componente di un qualche altro algoritmo. Fra i molti esempi che potremmo citare, ricordiamo qui l'algoritmo di Kruskal per il calcolo di un minimo albero di copertura in un grafo pesato. Si capisce quindi perché l'ordinamento sia stato oggetto di moltissimi studi scientifici e, in ambito didattico, un caso esemplare per l'insegnamento di idee algoritmiche e di tecniche di analisi della complessità.

L'input per un algoritmo di ordinamento è costituito da una collezione di dati, che possono essere atomici, tipicamente numeri o stringhe, o strutturati. Nel primo caso, sul "tipo" di dato deve essere definita una relazione d'ordine totale; questo vuol dire che, presi comunque due elementi della collezione, poniamo a e b, deve sempre valere una e una sola delle seguenti relazioni: $a < b$, $a = b$ o $b < a$. Detto ancora più semplicemente, per poter ordinare dei dati questi devono poter essere confrontati; si deve cioè poter dire chi viene prima e chi viene dopo. Nel caso di dati strutturati, la proprietà di ordine totale deve valere per almeno un componente della struttura (i componenti

sono spesso detti *campi*). Ad esempio, ogni struttura relativa a studenti universitari includerà sempre campi atomici quali il nome, il cognome e la matricola, che è possibile utilizzare per l'ordinamento. Nel seguito chiameremo *chiave* il campo utilizzato per l'ordinamento, da non confondersi con il concetto di chiave per un database relazionale.

Da un punto di vista puramente algoritmico, anche in presenza di dati strutturati, quel che conta è sempre e solo la chiave. Gli altri campi non possono certamente essere ignorati quando i dati devono essere spostati per metterli in ordine, ma non giocano alcun ruolo nell'algoritmo che decide la posizione ordinata finale. A volte in letteratura si fa riferimento ai campi che non sono la chiave come ai *dati satellite*. La seguente definizione formale del problema dell'ordinamento esclude quindi, per il motivo illustrato sopra, ogni riferimento ai dati datellite.

ORDINAMENTO	
INPUT:	Sequenza $S = \langle x_1, x_2, \ldots, x_n \rangle$ di elementi distinti
OUTPUT:	Permutazione $\bar{S} = \langle \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n \rangle$ di S t.c. $i < j \Rightarrow \bar{x}_i < \bar{x}_j$

Se gli elementi non sono tutti distinti, allora la proprietà che caratterizza la permutazione di output deve essere sostituita con:

$$i < j \Rightarrow \bar{x}_i \leq \bar{x}_j$$

Nel resto di questa sezione 3.1 presenteremo alcuni algoritmi che rappresentano soluzioni "classiche" al problema dell'ordinamento. I vari algoritmi che vedremo sono interessanti da uno o più fra i seguenti punti di vista: semplicità, efficienza, tecnica algoritmica o struttura dati utilizzata.

3.1.1 Insertion sort

L'algoritmo di ordinamento per inserzione, INSERTION-SORT, è una procedura che viene a volte descritta come il metodo con cui un giocatore ordina una "mano" di carte. Il modo classico con cui il giocatore procede consiste nell'inserire "in mano" ogni nuova carta alzata dal tavolo mantenendo ordinato l'insieme di tutte le carte già in mano.

Il procedimento seguito dall'algoritmo INSERTION-SORT per ordinare una generica sequenza S di n elementi simula questo comportamento ed è composto di $n - 1$ passi: il j-esimo passo corrisponde alla sistemazione della j-esima carta ricevuta, $j = 2, \ldots, n$. Più precisamente, se all'inizio del passo

j-esimo vale la condizione che la sotto-sequenza $S[1 : j-1]$ è ordinata, allora la stessa condizione vale per la sotto-sequenza $S[1 : j]$ quando il passo termina[1]. Alla fine del passo n, e dunque al termine dell'algoritmo, tutta la sequenza S risulta pertanto ordinata. Si noti che, inizialmente, cioè all'inizio del passo con $j = 2$, la condizione è verificata in quanto, banalmente, una sequenza composta da un solo elemento è ordinata.

Il funzionamento del generico passo j, che simula l'inserimento di una nuova carta, è descritto di seguito. Si tenga presente che, in questa "analogia" che stiamo perseguendo, le carte già in mano sono rappresentate dalle chiavi alle posizioni da 1 a $j-1$ (che quindi sono già in ordine); la chiave in posizione j corrisponde invece alla nuova carta da inserire; infine, le chiavi alle posizioni da $j+1$ ad n rappresentano le carte ancora sul tavolo. Nella descrizione che segue diremo che la chiave $S[j]$ da inserire "vince" il confronto con una chiave $S[k]$ sistemata in precedenza (dunque $k < j$) se $S[j] < S[k]$, altrimenti diremo che lo "perde".

1. Si copia la chiave $S[j]$ in una variabile ausiliaria t. Questa operazione corrisponde ad alzare la carta (e quindi toglierla) dal tavolo. Dopo la copiatura, la posizione j può essere considerata libera e, dunque, potrà essere liberamente sovrascritta senza tema di perdere dati.

2. Per stabilire la posizione corretta di inserimento, si confronta la chiave t in successione con le chiavi $S[k]$, $k = j-1, j-2, \ldots$; se t vince il confronto, allora $S[k]$ viene spostata di una posizione a destra e si procede al confronto successivo (decrementando k). Ci sono due condizioni che pongono fine ai confronti: (i) se t è stata già confrontata con tutte le chiavi e ha sempre vinto i confronti (questo caso si rileva perché il cursore k arriva al valore 0); (ii) se, per un certo valore di k, t perde il confronto con $S[k]$. In entrambi i casi, la corretta posizione di inserimento è la $(k+1)$-esima della sequenza. Si noti che tutto il blocco di chiavi originalmente in $S[k+1 : j-1]$ è stato spostato di una posizione a destra, così che la posizione $S[k+1]$ risulta libera per l'inserimento. Si veda la figura 3.1.

L'algoritmo risultante è illustrato in Figura 3.2. La correttezza della procedura INSERTION-SORT è garantita dal fatto vale la condizione invariante di ciclo.

[1]Una condizione che è vera all'inizio e alla fine del passo viene anche detta *invariante* di ciclo.

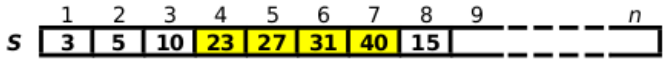

Figure 3.1: Il blocco di chiavi evidenziato in giallo deve essere spostato a destra di una posizione per consentire l'inserimento della chiave 15.

INSERTION-SORT(S)
1 $n = S.length$
2 **for** $j = 2$ **to** n
3 $t = S[j]$ // Estrazione del valore $S[j]$ da posizionare in ordine
// Calcolo della posizione k di inserimento e spostamento
4 $k = j - 1$
5 **while** $k > 0$ **and** $t < S[k]$
6 $S[k + 1] = S[k]$
7 $k = k - 1$
8 $S[k + 1] = t$ // Inserimento di t nella posizione corretta

Figure 3.2: Ordinamento per inserzione.

Per quanto riguarda l'efficienza, vale il seguente:

Teorema 3.31. Su input una sequenza composta da n chiavi, l'algoritmo INSERTION-SORT termina in tempo $O(n^2)$.

Dim. Indichiamo con $T_{iS}(n)$ la complessità dell'algoritmo illustrato in Figura 3.2 nel caso più sfavorevole. Ogni riga della procedura richiede un tempo di esecuzione costante, cioè indipendente da n; ogni riga può infatti essere "compilata" (su un qualsiasi computer e ipotizzando un qualsiasi linguaggio/compilatore concreto) in poche righe di codice macchina. Bisogna quindi solo chiedersi "quante volte" ogni data riga viene eseguita. Il valore è evidentemente diverso da riga a riga e anche in dipendenza della specifica sequenza in ingresso. Più precisamente (escludendo ovviamente i commenti):

1. La riga 1 viene eseguita una volta sola.

2. La riga 2 viene eseguita n volte.

3. Le righe 3, 4 e 8 vengono eseguite $n-1$ volte.

4. Per ogni valore di $j \in \{2, \ldots, n\}$, la riga 5 viene eseguita da un minimo di 1 volta (se il test fallisce subito) fino ad un massimo di j volte (se il test fallisce solo quando $k = 0$). Poiché siamo interessati allo studio di complessità nel caso più sfavorevole, dobbiamo considerare la situazione in cui questa riga venga eseguita esattamente j volte. Considerato l'intervallo di variazione di j, possiamo quindi concludere che la riga 5 viene eseguita

$$\sum_{j=2}^{n} j = -1 + \sum_{j=1}^{n} j = -1 + \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} - 1$$

volte.

5. Infine, le righe 6 e 7 vengono eseguite, per ogni valore di $j \in \{2, \ldots, n\}$, esattamente una volta meno rispetto a riga 5. Nel caso più sfavorevole esse vengono dunque eseguite

$$\sum_{j=2}^{n} (j-1) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

volte.

Sommando tutti i contributi possiamo concludere che, in dipendenza della tecnologia utilizzata (architettura, linguaggio e compilatore/interprete), esistono opportune costanti a, b e c tali che risulta $T_{iS}(n) = an^2+bn+c = \Theta(n^2)$. Questo dimostra che, in generale, il tempo di esecuzione di Insertion-Sort è $O(n^2)$. □

3.1.2 Quicksort

L'algoritmo che presentiamo in questa sezione rappresenta la scelta "tipica" per ordinare una sequenza di dati in memoria ad accesso diretto. L'algoritmo ha una complessità $O(n^2)$; tuttavia tempi di esecuzione quadratici nella lunghezza della sequenza non sono quasi mai raggiunti. Al contrario l'algoritmo è molto efficiente in pratica e questo giustifica il suo stesso nome: Quicksort, cioè (algoritmo di) ordinamento veloce.

Il QUICKSORT è un algoritmo basato sulla tecnica *divide-et-impera* e quindi implementabile in modo naturale mediante ricorsione. Di esso è tuttavia relativamente agevole fornire anche implementazioni non ricorsive, a tutto vantaggio dell'efficienza. Il QUICKSORT è anche, in un certo senso, "speculare" all'algoritmo MERGESORT già studiato. Entrambi gli algoritmi suddividono la sequenza in due sotto-sequenze, procedono in modo ricorsivo al loro ordinamento e infine ricombinano le sotto-sequenze ordinate in un'unica sequenza ordinata; tuttavia, mentre per il MERGESORT la componente dispendiosa è la ricombinazione delle sotto-sequenze, nel caso del QUICKSORT tutto il lavoro viene svolto in fase di suddivisione della sequenza in sotto-sequenze.

L'idea algoritmica alla base del QUICKSORT può essere descritta in modo molto semplice. Si considera un elemento qualsiasi della sequenza S da ordinare. Tale elemento prende il nome di *pivot*. Si riorganizza poi S in modo tale che valga la seguente proprietà, che chiameremo *di partizionamento*:

Proprietà 3.32. (Di partizionamento) Tutte le chiavi che, nella sequenza riordinata, si trovano alla sinistra (ovvero, destra) del pivot sono non maggiori (ovvero, non minori) del pivot stesso.

In Figura 3.3 mostriamo una sequenza prima e dopo il riordinamento. In questo caso il pivot scelto è il valore in prima posizione nella sequenza originale. È immediato rendersi conto che, in forza della proprietà di partizionamento, dopo la riorganizzazione il pivot è collocato nella posizione "giusta", cioè nella posizione in cui deve trovarsi nella sequenza ordinata. Si noti che la proprieà di partizionamento non dice nulla riguardo l'ordine relativo delle chiavi a sinistra e a destra del pivot, le quali non sono necessariamente già al loro posto (si veda ancora la Figura 3.3). Tuttavia, se ordiniamo anche le due sotto-sequenze dei numeri a sinistra e a destra del pivot, otteniamo come risultato che l'intera sequenza S risulta ordinata.

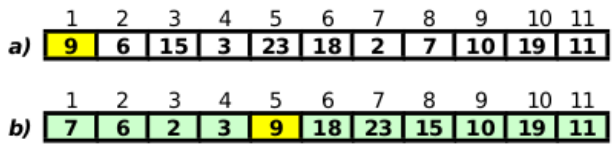

Figure 3.3: Riordinamento della sequenza iniziale a) intorno al pivot 9.

```
QUICKSORT(S, l, r)
1  if l < r
2        j = PARTITION(S, l, r)
3        QUICKSORT(S, l, j − 1)
4        QUICKSORT(S, j + 1, r)
```

Figure 3.4: Algoritmo Quicksort.

Siamo dunque perfettamente nella "logica" della tecnica divide-et-impera: abbiamo cioè ricondotto il problema originale (ordinamento di una sequenza di n chiavi) a due problemi di identica natura ma su input di dimensione più piccola. E dunque questi ultimi possono essere risolti ricorsivamente mediante lo stesso procedimento. Nell'ipotesi di disporre di una procedura che opera il riordinamento della sequenza in modo che soddisfi la proprietà di partizionamento, il codice per QUICKSORT risulta di immediata scrittura (Figura 3.4).

Nel codice per QUICKSORT abbiamo chiamato PARTITION la funzione che opera il riordinamento, perché questo è il nome ad essa attribuito in letteratura. La funzione restituisce l'indice della posizione in cui è stato collocato il pivot. La figura 3.5 illustra tutte le chiamate ricorsive di QUICKSORT generate su input la sequenza già considerata nella figura 3.3.

La procedura QUICKSORT riceve in ingresso non solo la sequenza da ordinare ma anche due indici l ed r che ne individuano una porzione. In effetti, le chiamate ricorsive che via via si susseguono dovono procedere all'ordinamento di porzioni diverse della sequenza che possono essere individuate indicando la posizione di inizio e di fine. Ovviamente la chiamata iniziale sarà QUICKSORT$(S, 1, n)$, dove $n = S.length$. Il test di riga 1 controlla la lunghezza della sotto-sequenza da ordinare. Se essa è composta di un solo elemento (caso $l = r$) allora non è necessario compiere alcuna operazione. È anche possibile che risulti $l > r$: questo accade quando il pivot è il più piccolo (o il più grande) elemento della sotto-sequenza, cioè quando $j = l$ (o $j = r$); in tal caso la successiva chiamata di riga 3 (riga 4) sarà effettuata con gli argomenti $S, l, l - 1$ (o $S, r + 1, r$). In altri termini, la condizione $l > r$ indica che una delle due sotto-sequenze da ordinare è vuota; e anche in questi casi non è necessario fare alcunché. Se invece ci sono almeno

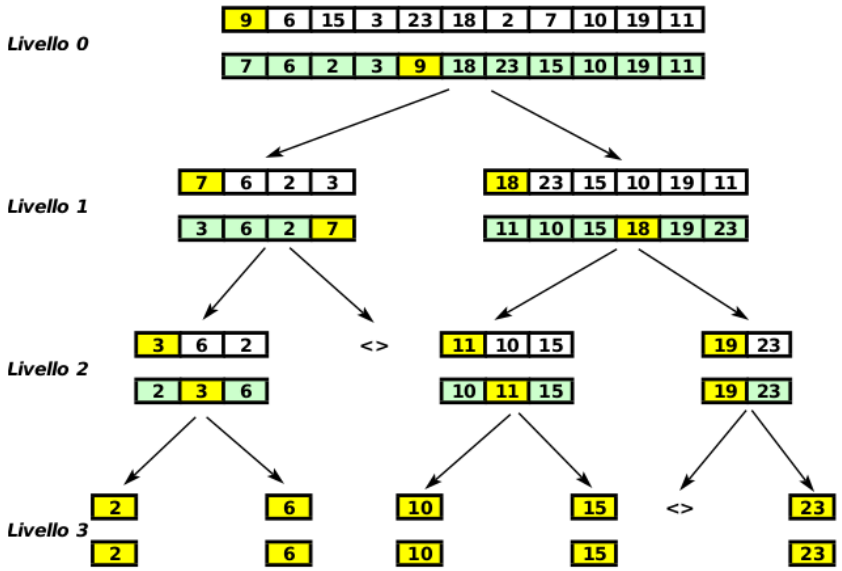

Figure 3.5: Chiamate ricorsive di QUICKSORT generate sulla sequenza già considerata nella figura 3.3. Ad ogni livello è illustrata la sotto-sequenza in input e la sotto-sequenza dopo il riordinamento, in modo che siano evidenti gli input per le successive chiamate ricorsive. Sotto-sequenze "vuote" sono indicate con $\langle\rangle$.

due elementi nella sequenza (caso $l < r$) si procede al riordinamento e alle chiamate ricorsive.

Rimane dunque da descrivere come può essere effettuato il riordinamento, cioè fornire una possibile implementazione per PARTITION. Di questa funzione esistono moltissime varianti in letteratura. La versione che viene presentata in questi appunti non è quella originale ne' la più diffusa. La scelta è stata fatta esclusivamente per considerazioni didattiche.

Iniziamo dalla scelta del pivot. L'ideale (per ragioni che vedremo quando parleremo della complessià del QUICKSORT) è che il pivot sia l'elemento mediano della sequenza, o quanto meno un elemento vicino al mediano. In questo modo, infatti, la sua collocazione nella sequenza ordinata sarebbe all'incirca a metà della sequenza e le due chiamate ricorsive verrebbero quindi effettuate su sotto-sequenze grosso modo della stessa lunghezza. Tuttavia cercare l'elemento mediano richiederebbe un algoritmo a parte che comprometterebbe la performance concreta (anche se non la complessità asintotica)

```
RANDOM-PARTITION(S, l, r)
1  q = ⌊RANDOM() · (r − l + 1) + l⌋
2  S[l] ↔ S[q] // Scambia le chiavi S[l] e S[q]
3  return PARTITION(S, l, r)
```

Figure 3.6: Procedura di riordinamento probabilistica. Si suppone che la procedura deterministica PARTITION prenda il pivot dalla prima posizione (cioè consideri $S[l]$ come pivot).

del QUICKSORT. Si preferisce quindi operare una scelta che sia efficiente in termini di tempo pur con la possibilità di incorrere in un forte sbilanciamento delle due sotto-sequenze in termini di lunghezza. Ci sono due possibilità.

Scelta *deterministica* Il pivot è sempre il primo (oppure l'ultimo) elemento della sequenza.

Scelta *probabilistica* Il pivot viene scelto "a caso" fra gli elementi della sequenza.

Dal punto di vista della complessità worst-case le due scelte sono equivalenti. Tuttavia l'opzione probabilistica è preferibile sia in pratica (quando esiste la possibilità che le sequenze da ordinare siano affette da qualche "polarizzazione" statistica) sia perché consente un agevole studio del comportamento "medio". Torneremo su questo punto in seguito.

L'opzione probabilistica può essere semplicemente realizzata utilizzando un generatore casuale per generare un numero intero compreso nell'insieme $\{l, l+1, \ldots, r\}$, da utilizzare come indice del pivot. La Figura 3.6 illustra questa soluzione, che ha il pregio di ricondurre la soluzione probabilistica a quella deterministica, che andremo subito dopo a sviluppare.

Si noti il modo con cui viene generato un numero intero compreso fra l e r. Si è ipotizzato che la funzione RANDOM (tipicamente disponibile nelle librerie di qualunque linguaggio di programmazione), invocata senza parametri restituisca un numero reale R uniformemente distribuito nell'intervallo $[0, 1)$. Moltiplicando R per la quantità $r - l + 1$ si ottiene un secondo numero $\bar{R}$ distribuito uniformemente nell'intervallo $[0, r - l + 1)$. Se ad $\bar{R}$ sommiamo il valore l e prendiamo poi la parte intera del risultato otteniamo con identica

probabilità uno qualsiasi dei numeri interi compresi nell'intervallo $[l, r+1)$ e dunque nell'insieme $\{l, l+1, \ldots, r\}$.

Veniamo ora al punto conclusivo, e cioè la definizione della procedura deterministica di riordinamento. Nel codice di figura 3.6 abbiamo già implicitamente stabilito, senza perdita di generalità, che il pivot x sia il primo elemento della sotto-sequenza, cioè $x = S[l]$. L'idea per effettuare il riordinamento consiste nel verificare alternativamente se le chiavi a destra del pivot sono maggiori di x (cioè del pivot stesso) e se le chiavi a sinistra del pivot sono minori di x.

Inizialmente, poiché abbiamo posto $x = S[l]$, non ci sono elementi alla sinistra del pivot, per cui si inizia "dalla parte destra". Supponiamo che gli ultimi k elementi della sequenza, cioè $S[r-k+1], \ldots, S[r]$, siano maggiori di x (naturalmente può anche risultare $k = 0$) mentre $S[r-k] \leq x$. Gli ultimi k elementi sono dunque già "dalla parte giusta" rispetto al pivot mentre $S[r-k]$ è dalla parte sbagliata. L'algoritmo procede quindi a scambiare di posto il pivot con l'elemento $S[r-k]$ il quale, passando a sinistra del pivot, si trova ora dalla parte giusta. Dopo lo scambio il pivot in generale non si troverà nell'ultima posizione della sequenza (salvo nel caso in cui $k = 0$). Tuttavia è come se lo fosse, perché tutti gli eventuali elementi che si trovano alla sua destra sono strettamente maggiori del pivot e dunque possono essere ignorati perché già a posto.

In questa situazione si procede a verificare quali elementi alla sinistra del pivot sono a posto e quali fuori posto. Questo viene fatto in modo "speculare" a quanto appena visto: le chiavi a sinistra che sono minori del pivot rimangono dove sono, perché dalla parte giusta; non appena se ne trova una non minore del pivot si procede ad un nuovo scambio. Il pivot ritorna così a sinistra e si ripete l'intero processo. La strategia è illustrata in Figura 3.7.

Tradurre in pseudocodice l'idea appena illustrata è relativamente semplice. Si utilizzano due indici, i e j, da utilizzare come cursori della sequenza, rispettivamente per i confronti con gli elementi a destra e a sinistra del pivot. L'indice i viene inizializzato al valore l e viene incrementato a seguito di un confronto superato[2]; specularmente, l'indice j viene inizializzato al valore r e decrementato a seguito di un confronto superato. Il pivot si troverà memorizzato in modo alternato nelle posizioni indicate da i e da j. La condizione di terminazione, che astrattamente richiede che "ogni chiave sia dalla parte

[2]Con questo intendiamo che la chiave confrontata con il pivot sta dalla parte giusta del pivot stesso, cioè a sinistra se minore e a destra se maggiore.

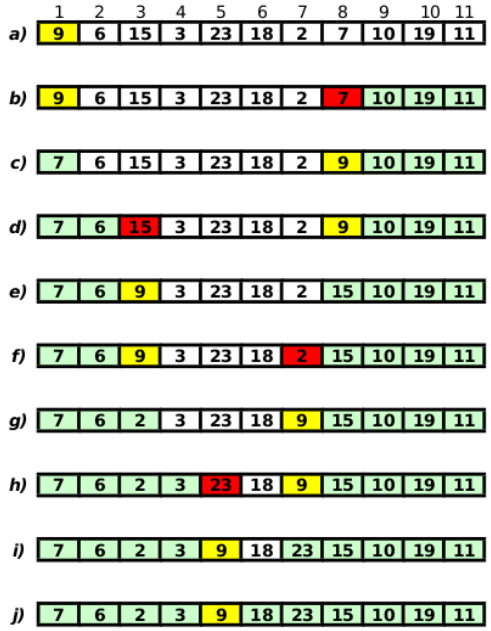

Figure 3.7: Funzionamento di PARTITION. Il pivot è evidenziato su sfondo giallo, le chiavi già dal lato giusto del pivot su sfondo verde, la chiave che deve essere scambiata con il pivot su sfondo rosso e le chiavi non ancora controllate su sfondo bianco. a) La sequenza iniziale con evidenziato il pivot; b) la prima fase di confronto (da destra) si arresta sulla chiave 7, che è minore del pivot e dunque dal lato sbagliato; c) dopo lo scambio la chiave 7 è a posto; d) la seconda fase di confronto (da sinistra) si arresta sulla chiave 15; e) dopo lo scambio la chiave 15 è a posto e ricomincia il confronto da destra; ...

giusta rispetto al pivot" , può essere verificata utilizzando i cursori. Infatti, fintanto che $i < j$ esiste almeno un elemento che non è stato confrontato con il pivot e dunque la procedura deve continuare. Lo pseudo-codice è illustrato in Figura 3.8

Esercizio 3.33. (Di comprensione) Perchè i confronti alle righe 4 e 9 sono effettuati con il minore stretto e il maggiore stretto? Dopo tutto, una chiave uguale al pivot potrebbe benissimo stare sia a destra che a sinistra del pivot stesso.

Per studiare la complessità dell'algoritmo QUICKSORT dobbiamo preliminarmente valutare quella della procedura PARTITION. Come vedremo, tale

```
PARTITION(S, l, r)
 1  i = l
 2  j = r
 3  while i < j
 4        while S[j] > S[i] // Il pivot è S[i]
 5              j = j − 1
 6        if i < j              // j potrebbe anche aver raggiunto i
 7              S[i] ↔ S[j] // Scambia le chiavi S[i] e S[j]
 8              i = i + 1     // Inutile riconsiderare S[i] dopo lo scambio
 9              while S[i] < S[j] // Il pivot è S[j]
10                    i = i + 1
11              if i < j            // i potrebbe anche aver raggiunto j
12                    S[i] ↔ S[j] // Scambia le chiavi S[i] e S[j]
13                    j = j − 1   // Inutile riconsiderare S[j] dopo lo scambio
        // Il pivot è nuovamente S[i]
14  return i // i == j
```

Figure 3.8: Procedura di riordinamento deterministica. Il pivot utilizzato è la chiave $S[l]$.

complessità è $\Theta(n)$. Tuttavia, la presenza di due cicli while annidati (riga 3 e riga 4, come pure riga 3 e riga 9) rende non banalissima questa conclusione; in effetti tale presenza potrebbe far sospettare che la complessità sia quadratica. Approfittiamo di questo caso, comunque non troppo complicato, per illustrare una tecnica generale per valutare la complessità di un algoritmo che funziona a grandi linee nel modo seguente: è data una condizione di terminazione per l'algoritmo e si studia quanto costa ogni singolo passo di avvicinamento a tale condizione.

Nel caso di PARTITION, possiamo illustrare la situazione in termini di "distanza" fra gli indici i e j che individuano la porzione di sequenza che deve essere ancora analizzata. Se indichiamo con δ tale distanza, cioè poniamo $\delta = j - i$, inizialmente abbiamo $\delta = r - l = n - 1$. La condizione di terminazione, $i = j$, equivale dunque ad avere distanza $\delta = 0$ e le operazioni che riducono tale distanza agiscono o decrementando j (istruzioni 5 e 13) oppure incrementando i (istruzioni 8 e 10). Il seguente Lemma prova che la

complessità di PARTITION è lineare facendo vedere che il costo per ridurre δ di un'unità è costante

Lemma 3.34. La complessità di PARTITION è $\Theta(n)$, dove $n = r - l + 1$ è la lunghezza della (sotto) sequenza in input.

Dim. Osserviamo preliminarmente che ogni singola riga di codice richiede tempo costante e dunque, come già visto nel caso INSERTION-SORT, la complessità dipende dal numero di volte in cui le varie istruzioni vengono eseguite. Chiaramente, le istruzioni 1, 2 e 14 vengono eseguita una sola volta, per cui ci concentriamo sulle istruzioni dell'intero ciclo while che inizia a riga 3 (e termina a riga 13). Consideriamo dunque una generica iterazione del ciclo, con $i < j$, e indichiamo con n_k il numero di volte che viene eseguita l'istruzione numero k ($k = 3, \ldots, 13$). Si noti che solo per semplicità di notazione non utilizziamo variabili a due indici, come sarebbe più corretto per poter fare riferimento al numero dell'operazione e a quello del ciclo.

Con la posizione $\delta = j - i$, il cui significato abbiamo illustrato sopra, andiamo dunque a valutare il costo, in termini di numero complessivo di istruzioni eseguite, necessario e sufficiente per ridurre δ di un'unità. Le istruzioni che più ci interessano sono quelle che decrementano δ, e che dunque avvicinano alla condizione di terminazione $\delta = 0$: si tratta delle istruzioni 8 e 13, che vengono eseguite al più una volta, e delle istruzioni 5 e 10. Ci interessano inoltre le istruzioni 4 e 9 che "controllano" rispettivamente l'esecuzione della 5 e della 10. Delle altre istruzioni, la 3 e la 6 vengono eseguite esattamente una volta mentre le altre, cioè 7, 11 e 12, al più una volta. Posto $\bar{n} = n_3 + n_6 + n_7 + n_{11} + n_{12}$, valgono dunque le diseguaglianze $2 \leq \bar{n} \leq 5$.

Osserviamo ora che risulta $n_4 = n_5 + 1$ (come pure $n_9 = n_{10} + 1$), questo perché quando il test di controllo di riga 4 è falso l'istruzione di riga 5 non viene eseguita. Possiamo quindi esprimere il numero N complessivo di istruzioni eseguite durante l'iterazione con la formula:

$$N = 1 + \bar{n} + 2n_5 + 1 + n_8 + 2n_{10} + 1 + n_{13} = \tilde{n} + 2(n_5 + n_{10}) + n_8 + n_{13}$$

dove ovviamente abbiamo posto $\tilde{n} = \bar{n} + 2$, con $4 \leq \tilde{n} \leq 7$.

Il decremento cui è soggetto δ durante l'iterazione è invece dato dalla somma $D = n_5 + n_8 + n_{10} + n_{13}$, per cui la quantità cercata, cioè il costo necessario e sufficiente per ridurre δ di un'unità è il rapporto $R = N/D$, che ora andiamo a studiare.

Si noti che $D \geq 1$, cioè ad ogni iterazione δ viene almeno ridotto di un'unità. Questa proprietà è di facile verifica. Infatti, se l'istruzione 5 viene

eseguita almeno una volta allora la conclusione è immediata; altrimenti basta osservare che il test di riga 6 (identico a quello di riga 3) è ancora soddisfatto e dunque che almeno l'istruzione 8 viene eseguita.

Per limitare inferiormente e superiormente R distinguiamo i due casi in cui $n_5 + n_{10}$ sia rispettivamente: (a) uguale a 0, o (b) maggiore di 0. Ricordiamo inoltre che $n_8 + n_{13} \leq 2$.

(a) In questo caso, poiché $D \geq 1$, deve risultare $n_8 + n_{13} \geq 1$, per cui:

$$R = \frac{\tilde{n} + 2(n_5 + n_{10}) + n_8 + n_{13}}{n_5 + n_8 + n_{10} + n_{13}} = \frac{\tilde{n} + n_8 + n_{13}}{n_8 + n_{13}}$$

e dunque $R \geq \frac{4+2}{2} = 3$ e $R \leq \frac{7+1}{1} = 8$.

(b) Se $n_5 + n_{10} > 0$, per la maggiorazione possiamo assumere senza problemi $n_8 + n_{13} = 0$ a denominatore e $n_8 + n_{13} = 2$ a numeratore, per cui risulta $R < \frac{\tilde{n}+2(n_5+n_{10})+2}{n_5+n_{10}} = 2 + \frac{\tilde{n}+2}{n_5+n_{10}} \leq 2 + 9 = 11$. Per la minorazione, osserviamo che se $x > y > 0$ e $c > 0$, allora risulta $\frac{x+c}{y+c} < \frac{x}{y}$ e dunque possiamo porre $n_8 + n_{13} = 2$, ottenendo $R \geq \frac{\tilde{n}+2(n_5+n_{10})+2}{n_5+n_{10}+2} = \frac{6+2(n_5+n_{10})}{n_5+n_{10}+2} = \frac{2(n_5+n_{10}+2)+2}{n_5+n_{10}+2} > 2$.

Possiamo dunque concludere che, in ogni caso, un numero di istruzioni maggiore di 2 ma non superiore a 11 è necessario e sufficiente per ridurre di una unità il valore di δ. Se ricordiamo che $\delta = n - 1$ e includiamo anche le istruzioni 1, 2 e 14, possiamo concludere che la procedura termina dopo l'esecuzione di un numero di istruzioni compreso fra $2(n-1)+3 = 2n+1$ e $11(n-1)+3 = 11n-8$. Questo dimostra che la complessità di PARTITION è $\Theta(n)$. □

I dettagli dell'analisi di complessità per la procedura QUICKSORT sono un poco elaborati; anche in questo caso, come la procedura PARTITION, ne approfittiamo per illustrare una tecnica generale per studiare il comportamento di un algoritmo ricorsivo.

Sia P un programma ricorsivo e sia I un generico input per P. Definiamo *livello* di una chiamata ricorsiva il numero di chiamate di P che risultano "aperte" (cioè iniziate ma non concluse) al momento dell'invocazione della chiamata in questione. Consideriamo, ad esempio, un programma per il

calcolo del fattoriale di un numero che implementi in maniera "letterale" la definizione, ovvero:

$$\text{FATT}(n) = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot \text{FATT}(n-1) & \text{altrimenti} \end{cases}$$

Se vogliamo calcolare il fattoriale di 5 utilizzando tale programma, dobbiamo semplicemente invocare FATT(5). Questa invocazione è chiaramente di livello 0 perchè è la prima. La chiamata FATT(5) eseguirà poi la chiamata FATT(4), che in questo caso è di livello 1, e così via.

La *profondità* della ricorsione $d(P, I)$, di P su input I, è invece definita come il massimo valore k per cui $P(I)$ dà origine a chiamate di livello k. Nel caso del fattoriale la profondità su input n è chiaramente n.

Consideriamo ora l'algoritmo QUICKSORT, con riordinamento deterministico. È immediato rendersi conto che, se la sequenza di input è già ordinata, allora PARTITION non esegue nessuno scambio e restitiusce in output proprio il valore $l = 1$, dove si trova il pivot. In tale situazione, il partizionamento degli elementi della sequenza è tale per cui la prima sotto-sequenza (riga 3 di QUICKSORT) risulta "vuota" perché la sua lunghezza è addirittura negativa, mentre la seconda sotto-sequenza (riga 4) contiene $n-1$ elementi. La prima chiamata di livello 1 restituisce dunque subito il controllo al programma chiamante; la seconda invece inizia a lavorare su una sotto-sequenza di $n-1$ elementi e si comporterà esattamente come la prima. Anche in questo caso, come per il fattoriale, la profondità è lineare rispetto ad n (per la precisione è $n-1$). In questo caso si parla di *partizionamento sbilanciato* perché sbilanciato è l'albero che rappresenta la struttura delle chiamate ricorsive, come si può vedere dall'esempio in Figura 3.9, relativo al caso $n = 5$.

All'estremo opposto supponiamo che la sequenza di input sia tale per cui il pivot di ogni sotto-sequenza considerata è sempre l'elemento mediano. In tale circostanza, le due chiamate ricorsive di livello 1 lavorano su sotto-sequenze di lunghezza che è circa metà di n. A loro volta le quattro chiamate ricorsive di livello 2 (generate dalle due chiamate di livello 1) lavorano su sotto-sequenze di lunghezza che è circa un quarto di n. In generale, le chiamate di livello k sono in numero di 2^k e lavorano su sotto-sequenze di lunghezza circa $n/2^k$. Per $k = \log n$ risulta $n/2^k = 1$ e dunque la ricorsione termina. Ne consegue che la profondità è proprio $\log n$ circa[3]. In tal caso si parta di partizionamento

[3]Poiché k è un numero intero, scrivendo $k = \log n$ dovremo in realtà intendere, di volta in volta, $k = \lfloor \log n \rfloor$ oppure $k = \lceil \log n \rceil$. In caso di partizionamento perfettamente bilan-

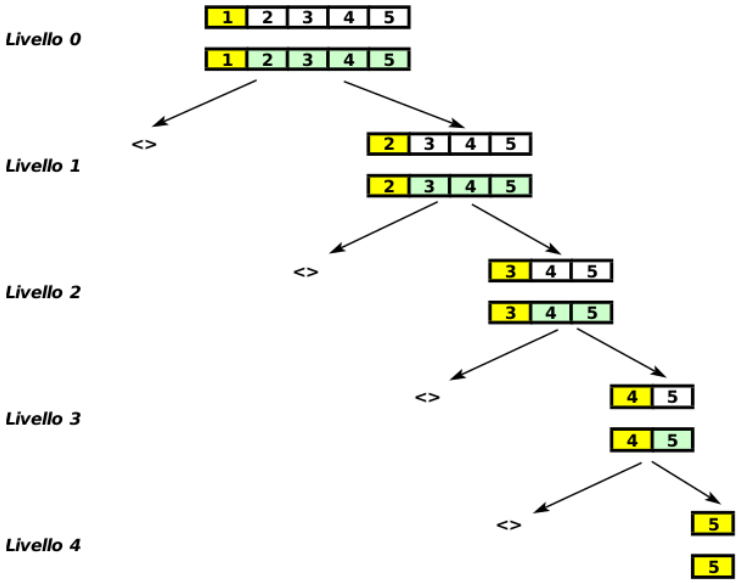

Figure 3.9: Caso di partizionamento sbilanciato per il QUICKSORT.

perfettamente bilanciato. Si veda al riguardo la Figura 3.10, relativo ad un particolare input di dimensione 7.

Esercizio 3.35. Usando il principio di induzione completa, si dimostri che in caso di partizionamento perfettamente bilanciato la profondità dell'albero di ricorsione della procedura QUICKSORT è $\lfloor \log n \rfloor$, dove n è la lunghezza della sequenza in input.

Il caso di partizionamento perfettamente bilanciato si verifica solo se $n + 1$ è una potenza di 2. In caso contrario, necessariamente qualche foglia dell'albero di ricorsione si troverà ad una profondità inferiore rispetto alla profondità massima. Si può anche facilmente dimostrare che $\lfloor \log n \rfloor$ è la minima profondità che può essere raggiunta su input di lunghezza n. L'albero di ricorsione originato dalla sequenza $\langle 9, 6, 15, 3, 23, 18, 2, 7, 10, 19, 11 \rangle$, illustrato in Figura 3.5, ha altezza $\lfloor \log n \rfloor = \lfloor \log 11 \rfloor = 3$; pur non essendo perfettamente bilanciato, perché 11+1 non è una potenza di 2, tale albero ha altezza minima e possiamo quindi affermare che esso descrive una situa-

ciato, qualunque sia n, la profondità dell'albero di ricorsione è $\lfloor \log n \rfloor$. Si veda l'esercizio 3.35.

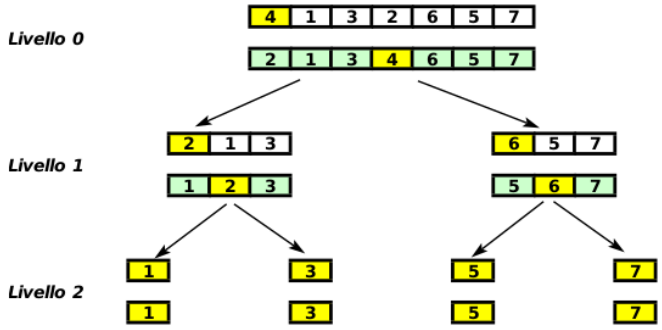

Figure 3.10: Caso di partizionamento perfettamente bilanciato per il QUICKSORT.

zione di partizionamento *bilanciato*. In realtà di parla più in generale di partizionamento bilanciato quando l'altezza dell'albero è $O(\log n)$.

Abbiamo ora quasi tutti gli "strumenti" per fare qualche affermazione quantitativa circa la complessità di QUICKSORT. L'ultima nozione preliminare riguarda il numero di istruzioni eseguite dalle chiamate di uno stesso livello di ricorsione. Definiamo dunque *lavoro* svolto da QUICKSORT a livello k su input I, denotato con $L(I,k)$, come la somma delle istruzioni eseguite dalle chiamate di livello k (incluse ovviamente le corrispondenti invocazioni di PARTITION), $k = 0, ..., d(\text{QUICKSORT}, I)$. Con questa posizione, indicata con $C_{QS}(I)$ la complessità di QUICKSORT su input I, risulta chiaramente

$$C_{QS}(I) = \sum_{k=0}^{d(\text{QUICKSORT},I)} L(I,k)$$

cioè la complessità è la somma di tutto il lavoro svolto, ad ogni livello di ricorsione.

Teorema 3.36. Nel caso in cui l'input I sia una sequenza ordinata, risulta $C_{QS}(I) = \Theta(n^2)$.

Dim. La dimostrazione è veramente semplice. Basta osservare che al livello k vengono effettuate due sole chiamate (si veda la Figura 3.9 per il caso particolare $n = 5$): la prima chiamata (ricevendo in input una sequenza "vuota") esegue solo un numero costante di operazioni; la seconda lavora su una sequenza di lunghezza di $n - k$ e dunque, per quanto dimostrato nel Lemma 3.34, esegue un numero di istruzioni che è $\Theta(n-k)$, $k = 0, 1, \ldots n -$

1. Considerato il significato della notazione asintotica, possiamo dunque affermare che esistono costanti positive $c_0, c_1, \ldots, c_{n-1}$ tali che il lavoro svolto dalle chiamate di livello k è $c_k \cdot (n-k)$, $k = 0, 1, \ldots n-1$. Per la complessità $C_{QS}(I)$ di QUICKSORT valgono quindi le seguenti disuguaglianze:

$$\begin{aligned} C_{QS}(I) &= \sum_{k=0}^{n-1} c_k \cdot (n-k) \\ &\geq \left(\min_{k=0,\ldots,n-1} c_k \right) \cdot \sum_{k=0}^{n-1} n-k = \left(\min_{k=0,\ldots,n-1} c_k \right) \sum_{k=1}^{n} k = c(n^2-n) \end{aligned}$$

dove si è posto $c = \frac{1}{2} \min_{k=0,\ldots,n-1} c_k$, e

$$\begin{aligned} C_{QS}(I) &= \sum_{k=0}^{n-1} c_k \cdot (n-k) \\ &\leq \left(\max_{k=0,\ldots,n-1} c_k \right) \cdot \sum_{k=0}^{n-1} n-k = \left(\max_{k=0,\ldots,n-1} c_k \right) \sum_{k=1}^{n} k = C(n^2-n) \end{aligned}$$

dove, analogamente, si è posto $C = \frac{1}{2} \max_{k=0,\ldots,n-1} c_k$. Le due disuguaglianze combinate dimostrano che la complessità è proprio $\Theta(n^2)$. □

Chiamiamo *nodo attivo* ogni nodo dell'albero della ricorsione del QUICKSORT che corrisponde ad una chiamata con input non vuoto[4]. Possiamo enunciare questa semplice

Proprietà 3.37. Il numero di nodi attivi nell'albero di ricorsione della procedura QUICKSORT coincide con la lunghezza della sequenza in input.

La veridicità della 3.37 è immediatamente evidente quando si rifletta sul fatto che ogni chiamata ricorsiva "garantisce" (tramite la chiamata a PARTITION) che uno e un solo elemento della sotto-sequenza sia "sistemato" nella corretta posizione finale, e precisamente il pivot. Per la stessa ragione la proprietà vale non solo per l'albero completo ma per ogni suo sotto-albero. Si vedano, come esempi, gli alberi illustrati nelle figure 3.5, 3.9 e 3.10.

Possiamo ora valutare la complessità dell'algoritmo QUICKSORT nel caso di perfetto bilanciamento.

[4]Ricordiamo che si ha input vuoto quando $l > r$.

Teorema 3.38. Nel caso in cui l'input I induca un albero di ricorsione perfettamente bilanciato risulta $C_{QS}(I) = \Theta(n \log n)$.

Dim. La dimostrazione è solo leggermente più complessa rispetto al caso esaminato di sequenza ordinata esaminato in precedenza. Si faccia riferimento alla Figura 3.10, relativa al valore $n = 7$. In un albero perfettamente bilanciato, indipendentemente dal fatto che esso corrisponda all'albero delle chiamate di un algoritmo ricorsivo, il numero di nodi al livello k è esattamente 2^k, $k = 0, \ldots, d$, dove d è l'altezza dell'albero, ed inoltre il numero complessivo dei nodi (foglie incluse) è $n = 2^{d+1} - 1$. Se poi consideriamo un qualsiasi nodo a livello k, esso è a sua volta la radice di un sotto-albero perfettamente bilanciato in cui il numero totale di nodi è $2^{d+1-k} - 1$.

Dalla proprietà 3.37 possiamo quindi conludere che ogni chiamata di livello k riceve in input una sotto-sequenza di $2^{d+1-k} - 1$ elementi da ordinare. Ma allora, per il Lemma 3.34, il numero di istruzioni eseguite da una singola chiamata di PARTITION a livello k è $\Theta(2^{d+1-k})$. Esisteranno dunque opportune costanti c_{jk}, $j = 1, \ldots, 2^k$ tali che il lavoro $L(I, k)$ eseguito dalle chiamate di livello k sia esprimibile nel modo seguente:

$$L(I, k) = \sum_{j=1}^{2^k} c_{jk} 2^{d+1-k}$$

Utilizzando la stessa tecnica impiegata nella dimostrazione del Teorema 3.36, poniamo ora $c_k = \min_{j=1,\ldots,2^k} c_{jk}$ e $C_k = \max_{j=1,\ldots,2^k} c_{jk}$. Ne consegue che $L(I, k)$ è limitabile inferiormente e superiormente nel modo seguente:

$$\begin{aligned} L(I, k) &= \sum_{j=1}^{2^k} c_{jk} 2^{d+1-k} \\ &\geq c_k \cdot \sum_{j=1}^{2^k} 2^{d+1-k} = c_k \cdot 2^k \cdot 2^{d+1-k} = c_k \cdot 2^{d+1} \approx c_k \cdot n \end{aligned}$$

e, analogamente,

$$L(I, k) \leq C_k \cdot 2^{d+1} \approx C_k \cdot n$$

Data l'arbitrarietà di k, le due disuguaglianze combinate dimostrano che ad ogni livello il lavoro complessivo è $\Theta(n)$. Poiché i livelli sono $d + 1$ e $d = \log(n + 1) - 1 \approx \log n$, ne consegue che $C_{QS}(I) = \sum_{k=1}^{d} L(I, k) = \Theta(n \log n)$. □

Anche se relativi a casi molto particolari (sequenze ordinate e sequenze che inducono un perfetto bilanciamento nell'albero della ricorsione), i due risultati che abbiamo dimostrati caratterizzano più in generale i comportamenti worst-case e best-case di QUICKSORT. Vale infatti il seguente risultato, la cui dimostrazione lasciamo per esercizio.

Teorema 3.39. La complessità dell'algoritmo QUICKSORT nei casi più sfavorevole e più favorevole è, rispettivamente, $\Theta(n^2)$ e $\Theta(n \log n)$, dove n indica la lunghezza delle sequenze in input.

La domanda che emerge spontanea riguarda il comportamento "tipico" di QUICKSORT. Si noti che, parlando di comportamento tipico, evitiamo accuratamente ogni riferimento a quella che in letteratura è nota come analisi del *caso medio* degli algoritmi (*average-case analysis*). Quest'ultima presuppone infatti la conoscenza di una distribuzione di probabilità sui possibili input per l'algoritmo oggetto di indagine; realisticamente, però, tale distribuzione di probabilità non è quasi mai nota e non è azzardato affermare che in molti casi l'analisi del caso medio risulta solo un bell'esercizio matematico.

Fortunatamente, però, se si ricorre alla versione probabilistica della procedura PARTITION, risulta possibile fare affermazioni rigorose che si applicano a tutti i contesti applicativi, indipendentemente da come si presenta l'input (cioè dalla probabilità di avere determinate sequenze in input).

Teorema 3.40. Il numero atteso di operazioni eseguite dall'algoritmo QUICKSORT (con partizionamento probabilistico) è $\Theta(n \log n)$.

Dim. Dimostriamo che il numero atteso di confronti fra coppie di elementi distinti eseguiti dall'insieme di tutte le chiamate di RANDOM-PARTITION, e dunque da QUICKSORT, è proprio $\Theta(n \log n)$. Questo è sufficiente per dimostrare l'asserto in quanto il numero di operazioni complessive (confronti, assegnamenti, incremento e test sulle variabili indice) sono chiaramente dello stesso ordine di grandezza dei soli confronti.

Indichiamo dunque con C_i il numero atteso di confronti eseguiti da QUICKSORT su input una sequenza composta da i elementi. Vogliamo ovviamente stimare il valore di C_n. È immediato osservare che $C_0 = C_1 = 0$. Per $n > 1$, dall'analisi del codice della funzione PARTITION (che viene chiamata da RANDOM-PARTITION dopo la scelta casuale dell'elemento pivot) si evince inoltre facilmente che tutti gli elementi vengono confrontati una e una sola

volta col pivot. Dopodiché QUICKSORT esegue due chiamate ricorsive su sequenze di lunghezza $p-1$ e $n-p$, dove p è la posizione ordinata del pivot. In altri termini, per un valore di p fissato, potremmo scrivere:

$$C_n = n - 1 + C_{p-1} + C_{n-p} \tag{3.1}$$

Si noti, in particolare, che la formula implica correttamente che, quale che sia il valore di p, risulta $C_2 = 1$. Ora, poiché il pivot, e di conseguenza il valore di p, sono scelti con uguale probabilità fra tutti gli elementi della sequenza, ne consegue che il valore atteso di C_n è esattamente la media aritmetica delle quantità (3.1) per tutti i possibili valori di p. Dunque:

$$\begin{aligned} C_n &= n - 1 + \frac{1}{n}\sum_{p=1}^{n}(C_{p-1} + C_{n-p}) \\ &= n - 1 + \frac{2}{n}\sum_{p=0}^{n-1} C_p \end{aligned} \tag{3.2}$$

dove l'uguaglianza (3.2) dipende dal fatto che ogni valore di C_p compare esattamente due volte nella prima sommatoria. Se moltiplichiamo ora entrambi i membri per n otteniamo:

$$nC_n = n(n-1) + 2\sum_{p=0}^{n-1} C_p \tag{3.3}$$

Ora, nell'ipotesi $n > 2$, riscriviamo la (3.3) per C_{n-1}; l'espressione che otteniamo avrà ovviamente la stessa forma:

$$(n-1)C_n = (n-1)(n-2) + 2\sum_{p=0}^{n-2} C_p \tag{3.4}$$

Sottraiamo adesso la (3.4) dalla (3.3) e, con un ulteriore semplice passaggio, otteniamo:

$$nC_n = 2(n-1) + (n+1)C_{n-1}$$

da cui facilmente consegue:

$$n + (n+1)C_{n-1} \leq nC_n \leq 2n + (n+1)C_{n-1}$$

e, con un'ulteriore divisione per $n(n+1)$,

$$\frac{1}{n+1} + \frac{C_{n-1}}{n} \leq \frac{C_n}{n+1} \leq \frac{2}{n+1} + \frac{C_{n-1}}{n} \tag{3.5}$$

Queste sono le disiguaglianze cercate. Se infatti definiamo $F_n = \frac{C_n}{n+1}$ possiamo riscrivere le (3.5) come

$$\frac{1}{n+1} + F_{n-1} \leq F_n \leq \frac{2}{n+1} + F_{n-1}$$

per cui possiamo facilmente trovare forme chiuse per limitare F_n. Ad esempio, per il limite superiore abbiamo:

$$\begin{aligned} F_n &\leq \frac{2}{n+1} + F_{n-1} \\ &\leq \frac{2}{n+1} + \frac{2}{n} + F_{n-2} \\ &\leq \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + F_{n-3} \\ &\cdots \\ &\leq 2\sum_{k=4}^{n+1} \frac{1}{k} + F_2 \end{aligned}$$

e dunqe F_n è limitata superiormente da una funzione che, a meno di costanti additive, si comporta come $2 \ln n$. Ricordiamo infatti che per la serie armonica vale

$$\lim_{n\to\infty} \frac{\sum_{k=1}^{n} \frac{1}{k}}{\ln n} = 1$$

Analogamente, possiamo concludere che

$$F_n \geq \sum_{k=4}^{n+1} \frac{1}{k} + F_2$$

cioè che F_n è limitata inferiormente da una funzione che cresce come $\ln n$. Di conseguenza $F_n = \Theta(\ln n) = \Theta(\log n)$ e dunque $C_n = \Theta(n \log n)$. □