6) Il problema dell'ordinamento

Sommario: In questo capitolo si affronta l'importante problema dell'ordinamento.

Dapprima sono descritti degli algoritmi concettualmente semplici, che hanno costo computazionale quadratico nella dimesnione dell'input.

Poi ci si chiede se si possa sperare di trovare un algoritmo più efficiente; a tale scopo, si fornisce un teprema per cui nessun algoritmo di ordinamento che operi per confronti possa avere costo computazionale inferiore ad un ordine di $n \log n$ (dove n è la dimensione dell'input).

Vengono quindi descritti tre algoritmi con caratteristiche diverse e che risultino, almeno in certi casi, più efficienti dei precedenti. Uno di questi ha bisogno dell'introduzione di una particolare struttura dati.

Infine, vengono forniti degli algoritmi linari nella dimensione dell'input; si osserva che essi non contraddicono il teorema precedentemente nominato poiché non operano per confronti.

Il problema dell'ordinamento degli elementi di un insieme è un problema molto ricorrente in informatica poiché ha un'importanza fondamentale per le applicazioni: lo si ritrova molto frequentemente come sottoproblema nell'ambito dei problemi reali. In effetti si stima che una parte rilevante del tempo di calcolo complessivo consumato nel mondo sia relativa all'esecuzione di algoritmi di ordinamento.

Un algoritmo di ordinamento è un algoritmo capace di ordinare gli elementi di un insieme sulla base di una certa relazione d'ordine, definita sull'insieme stesso (maggiore/minore, precede/segue, ecc.).

Tutti gli algoritmi che vedremo hanno alcuni aspetti in comune. Si basano su due tipi fondamentali di operazioni: il confronto fra due elementi e lo scambio di due elementi. Questa ipotesi di lavoro è necessaria per non costruire soluzioni

irrealistiche (ad esempio, definendo una operazione che in un sol passo ordina un numero di elementi non limitato da una costante). Inoltre, per semplicità di trattazione, si suppone che gli *n* elementi da ordinare siano numeri interi e siano contenuti in un array i cui indici vanno da 1 a *n*.

Tuttavia, nei problemi reali, i dati da ordinare sono ben più complessi: in generale essi sono strutturati in ***record***, cioè in gruppi di informazioni non sempre omogenee relative allo stesso soggetto (ad esempio, l'archivio studenti ha moltissimi record, ciascuno contenente molte informazioni relative ad un singolo studente), e si vuole ordinarli rispetto ad una di tali informazioni (ad esempio il cognome, oppure il numero di matricola, ecc.).

Nel seguito assumeremo che gli elementi da ordinare dati in input siano memorizzati in un ***array***, o ***vettore***, una struttura dati in grado di mantenere in memoria dei dati omogenei e l'accesso a ciascun dato è possibile indicandone la sua posizione (ad esempio, volendo recuperare il quinto elemento dell'insieme *A,* leggeremo la posizione di memoria indicata da *A[5]*).

Notiamo una certa somiglianza tra un array e la classe Python *list*, la quale è tuttavia molto più potente. Discuteremo nel seguito differenze e somiglianze.

6.1 Algoritmi semplici

Di algoritmi di ordinamento ne esitono diversi e di diversi tipi. In un'ottica di progetto di algoritmi efficienti, illustreremo dapprima algoritmi di ordinamento concettualmente semplici ma non molto efficienti, per renderci poi conto del fatto che –volendo migliorare l'efficienza- sarà necessario mettere in campo idee meno banali.

6.1.1 Insertion sort

L'algoritmo ***insertion sort*** (***ordinamento per inserzione***) si può intuitivamente assimilare al modo in cui una persona ordina un mazzo di carte. Partendo da un mazzo vuoto inseriamo una carta alla volta, cercando il punto giusto dove inserirla. Alla fine il mazzo di carte risulta ordinato.

Nel caso dell'algoritmo insertion sort gli elementi da ordinare sono inizialmente contenuti nell'array quindi non si parte da un insieme vuoto ma si deve trovare la posizione giusta di ogni elemento (è come se si volessero ordinare le carte tenendole in mano, anziché appoggiandole e prendendole poi una per una). Ciò si effettua "estraendo" l'elemento così da liberare la sua posizione corrente, spostando poi verso destra tutti gli elementi alla sua sinistra (già ordinati) che sono maggiori di esso ed, infine, inserendo l'elemento nella posizione che si è liberata. Alla fine del procedimento l'array risulterà ordinato.

La formulazione dell'algoritmo è la seguente:

	Pseudocodice	Costo	Numero di esecuzioni
	`def Insertion_Sort(A):`		
1	`    for j in range(1, len(A)):`	$c_1 = \Theta(1)$	n
2	`        x = A[j]`	$c_2 = \Theta(1)$	$n - 1$
3	`        i = j - 1`	$c_3 = \Theta(1)$	$n - 1$
4	`        while ((i >= 0)and(A[i]) > x):`	$c_4 = \Theta(1)$	$\sum_{j=2}^{n} t_j$
5	`            A[i+1] = A[i]`	$c_5 = \Theta(1)$	$\sum_{j=2}^{n} (t_j - 1)$
6	`            i = i - 1`	$c_6 = \Theta(1)$	$\sum_{j=2}^{n} (t_j - 1)$
7	`        A[i+1] = x`	$c_7 = \Theta(1)$	$n - 1$

Nella terza colonna, t_j denota il numero di volte che il test del `while` viene eseguito per quel valore di j.

Valutiamo il costo computazionale dell'algoritmo insertion sort.

Caso migliore

Esso si verifica quando il numero di spostamenti da effettuare è minimo e cioè quando l'array è già ordinato. In tal caso la condizione del `while` non è mai verificata e quindi $t_j = 1$ per tutti i valori di j, quindi il costo è:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) = \Theta(n)$$

Caso peggiore

Esso si verifica quando il numero di spostamenti da effettuare è massimo e cioè quando l'array in partenza è ordinato all'incontrario. In tal caso $t_j = j$ per tutti i valori di j, quindi il costo è:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=2}^{n} j + c_5\sum_{j=2}^{n}(j-1) + c_6\sum_{j=2}^{n}(j-1) + c_7(n-1) =$$

$$= \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Si noti che a causa della presenza dei vari fattori costanti e dei termini di ordine inferiore l'analisi potrebbe essere errata per piccoli valori di n, ma è accurata per n abbastanza grande il che, come già esposto, è quello che ci interessa veramente.

Inoltre, questo algoritmo ci permette di evidenziare un aspetto interessante: i costi asintotici nel caso migliore e nel caso peggiore non risultano uguali, e questo si indica dicendo che l'algoritmo è ***input sensitive***.

In casi come questo, poiché vogliamo avere una stima di quanto tempo serve per eseguire l'algoritmo in questione, ci concentriamo sul caso peggiore, certi che l'algoritmo non potrà impiegare un tempo maggiore.

6.1.2 Selection sort

L'algoritmo ***selection sort*** (***ordinamento per selezione***) funziona nel seguente modo: cerca il minimo dell'intero array e lo mette in prima posizione (con uno scambio, fatto anche se inutile); quindi cerca il nuovo minimo nell'array restante, cioè nelle posizioni dalla seconda all'ultima incluse, e lo mette in seconda posizione (con uno scambio), poi cerca il minimo nelle posizioni dalla terza all'ultima incluse e lo mette in terza posizione, e così via. In pratica, dopo l'iterazione *i*-esima, l'array risulterà suddiviso in due parti: quella di sinistra (di lunghezza *i*) contenente i primi *i* elementi della sequenza ordinata nel giusto ordine, e quella di destra (di lunghezza *n*-*i*) contenente gli altri elementi non ordinati.

Rispetto all'insertion sort, la principale differenza è che, per sistemare definitivamente un singolo elemento, si fa un solo scambio, ma si scorre sempre

l'intero sottoarray di destra (la parte non ordinata). Quindi, se lo scambio è una operazione costosa, selection sort è meglio di insertion sort.

La formulazione dell'algoritmo è la seguente:

Pseudocodice	Costo	Numero di esecuzioni
`def Selection_Sort(A):`		
`0.  n=len(A)`	$C_0 = \Theta(1)$	*1*
`1  for i in range(n-1)`	$c_1 = \Theta(1)$	*n*
`2  m = i`	$c_2 = \Theta(1)$	*n - 1*
`3  for j in range(i + 1, n):`	$c_3 = \Theta(1)$	$\sum_{i=1}^{n-1}(n-i+1)$
`4  if (A[j] < A[m]):`	$c_4 = \Theta(1)$	$\sum_{i=1}^{n-1}(n-i)$
`5  m = j`	$c_5 = O(1)$	$\sum_{i=1}^{n-1}(n-i)$
`6  A[i],A[m]=A[m],A[i]`	$c_6 = \Theta(1)$	*n – 1*

Il costo di questo algoritmo non dipende in alcun modo dalla distribuzione iniziale dei dati nell'array, cioè resta la stessa sia che l'array sia inizialmente ordinato o no. Essa è:

$$T(n) = c_0 + c_1\, n + (c_2 + c_6)(n-1) + c_3 \sum_{i=1}^{n-1}(n-i+1) + (c_4 + c_5)\sum_{i=1}^{n-1}(n-i) =$$

$$= c_0 + c_1\, n + (c_2 + c_6)(n-1) + c_3\,(n-1) + (c_3 + c_4 + c_5)\sum_{i=1}^{n-1}(n-i) =$$

$$= c_0 + c_1\, n + (c_2 + c_3 + c_6)(n-1) + (c_3 + c_4 + c_5)\sum_{i=1}^{n-1}(n-i)$$

Ora, ponendo $k = n - i$, abbiamo $\sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} = \Theta(n^2)$ per cui

$$T(n) = c_0 + c_1\, n + (c_2 + c_3 + c_6)(n-1) + (c_3 + c_4 + c_5)\Theta(n^2) =$$

$$= \Theta(n) + \Theta(n^2) = \Theta(n^2).$$

6.1.3 Bubble sort

L'algoritmo ***bubble sort*** (***ordinamento a bolle***) ha un funzionamento molto semplice: l'algoritmo ispeziona una dopo l'altra ogni coppia di elementi adiacenti

e, se l'ordine dei due elementi non è quello giusto, essi vengono scambiati. L'algoritmo prosegue fino a che non vi sono più coppie di elementi adiacenti nell'ordine sbagliato.

La formulazione dell'algoritmo è la seguente:

Pseudocodice	Costo	Numero di esecuzioni
`def Bubble_Sort(A):`		
`0  n=len(A)`	$c_0 = \Theta(1)$	*1*
`1  for i in range(n-1):`	$c_1 = \Theta(1)$	*n*
`2  for j in range (n-i-1):`	$c_2 = \Theta(1)$	$\sum_{i=1}^{n-1}(n-i+1)$
`3  if (A[j] > A[j+1]):`	$c_3 = \Theta(1)$	$\sum_{i=1}^{n-1}(n-i)$
`4  A[j],A[j+1]=A[j+1],A[j]`	$c_4 = O(1)$	$\sum_{i=1}^{n-1}(n-i)$

In questo caso abbiamo:

$$T(n) = c_0 + c_1\,(n+1) + c_2 \sum_{i=1}^{n-1}(n-i+1) + (c_3 + c_4)\sum_{i=1}^{n-1}(n-i) =$$

$$= c_0 + c_1\,(n+1) + c_2\,n + (c_2 + c_3 + c_4)\sum_{i=1}^{n-1}(n-i) =$$

$$= c_0 + c_1\,(n+1) + c_2\,n + (c_2 + c_3 + c_4)\sum_{k=1}^{n-1} k =$$

$$= c_0 + c_1\,(n+1) + c_2\,n + (c_2 + c_3 + c_4)\frac{(n-1)n}{2} = \Theta(n) + \Theta(n^2) = \Theta(n^2).$$

Si osservi che è possibile, con poche istruzioni, modificare l'algoritmo di Bubble Sort affinché diventi input sensitive.

6.2 La complessità del problema dell'ordinamento

Abbiamo visto che i tre algoritmi di ordinamento precedenti hanno tutti un costo computazionale asintotico che cresce come il quadrato del numero di elementi da ordinare.

Una domanda sorge spontanea: si può fare di meglio? E se si, quanto meglio si può fare?

Una risposta alla prima domanda viene data non appena si riesce a progettare un algoritmo di ordinamento che esibisca un costo computazionale inferiore a quella quadratica, ma rispondere alla seconda domanda è più difficile: chi ci assicura che non si possa fare meglio anche di questo ipotetico nuovo algoritmo?

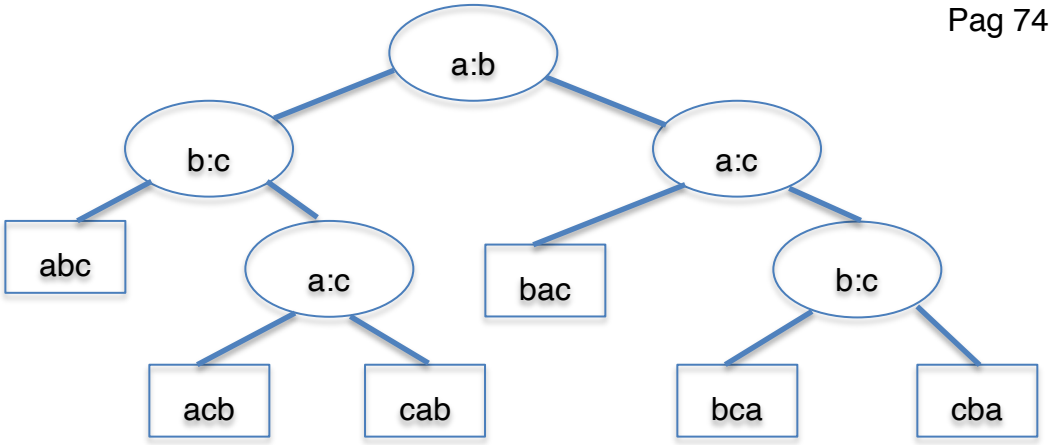
In altre parole, come si fa a stabilire un limite di costo computazionale ***al di sotto del quale nessun algoritmo di ordinamento basato su confronti fra coppie di elementi possa andare***?

Esiste uno strumento adatto allo scopo, l'***albero di decisione***. Esso permette di rappresentare tutte le strade che la computazione di uno specifico algoritmo può intraprendere, sulla base dei possibili esiti dei test previsti dall'algoritmo stesso.

Nel caso degli algoritmi di ordinamento basati su confronti, ogni test effettuato ha due soli possibili esiti (ad es.: minore o uguale, oppure no). Quindi l'albero di decisione relativo a un qualunque algoritmo di ordinamento basato su confronti ha queste proprietà:

- è un albero binario che rappresenta tutti i possibili confronti che vengono effettuati dall'algoritmo; ogni nodo interno (ossia un nodo che non è una foglia) ha esattamente due figli;
- ogni nodo interno rappresenta un singolo confronto, ed i due figli del nodo sono relativi ai due possibili esiti di tale confronto;
- ogni foglia rappresenta una possibile soluzione del problema, la quale è una specifica permutazione della sequenza in ingresso;
- gli altri aspetti dell'algoritmo (ad es. lo spostamento dei dati) vengono ignorati.

Ad esempio, l'albero di decisione relativo all'algoritmo di insertion sort applicato a un array di tre elementi [a, b, c] è il seguente (la notazione *a:b* rappresenta l'operazione di confronto fra *a* e *b,* la scelta di sinistra indica che a≤b mentre quella di destra che a>b):



Eseguire l'algoritmo corrisponde a scendere dalla radice dell'albero alla foglia che contiene la permutazione che costituisce la soluzione; la discesa è governata dagli esiti dei confronti che vengono via via effettuati. La lunghezza (ossia il numero degli archi) di tale cammino rappresenta il numero di confronti necessari per trovare la soluzione.

Dunque, la lunghezza del percorso più lungo dalla radice ad una foglia (ossia l'***altezza*** dell'albero binario) rappresenta il numero di confronti che l'algoritmo deve effettuare nel caso peggiore.

Questo significa che determinare una limitazione inferiore all'altezza dell'albero di decisione relativo a ***qualunque*** algoritmo di ordinamento basato su confronti equivale a trovare una limitazione inferiore al tempo di esecuzione nel caso peggiore di qualunque algoritmo di ordinamento basato su confronti.

Ora, dato che la sequenza di ingresso può avere una qualunque delle sue permutazioni come soluzione, l'albero di decisione deve contenere nelle foglie tutte le permutazioni della sequenza in ingresso, che sono $n!$ per un problema di dimensione n. D'altra parte, un albero binario di altezza h non può contenere più di 2^h foglie (per questo risultato, si consulti il capitolo sugli alberi). Quindi l'altezza h dell'albero di decisione di qualunque algoritmo di ordinamento basato su confronti deve essere tale per cui:

$$2^h \geq n! \text{ ossia } h \geq \log n!$$

Osservando che gli ultimi $n/2$ fattori di $n!$ sono grandi almeno $n/2$, si ha che

$n! \geq (n/2)^{n/2}$ per qualunque n, da cui:

$$\log n! \geq \log (n/2)^{n/2} = n/2 \log n/2 = \Theta(n \log n).$$

Dunque $h \geq \Theta(n \log n)$, per cui si deduce il seguente:

Teorema: *Il costo computazionale di qualunque algoritmo di ordinamento basato su confronti è $\Omega(n \log n)$.*

NOTA: In letteratura, per la dimostrazione di questo risultato sfrutta spesso l'approssimazione di Stirling:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right) \text{ per } n \text{ abbastanza grande}$$

da cui:

$$\log n! = \Theta\left(\log \sqrt{n}\left(\frac{n}{e}\right)^n\right) = \Theta(\log n^{n+\frac{1}{2}} - \log e^n) = \Theta((n+\tfrac{1}{2})\log n - n \log e) =$$

$$= \Theta(n \log n) - \Theta(n) = \Theta(n \log n).$$

6.3 Algoritmi efficienti

Studieremo nel seguito tre algoritmi di ordinamento molto più efficienti dei precedenti: essi infatti raggiungono il limite inferiore teorico. Per due degli algoritmi (mergesort ed heapsort) ciò avviene anche nel caso peggiore, per il terzo (quicksort) solo nel caso medio.

6.3.1 Mergesort

L'algoritmo ***mergesort*** (***ordinamento per fusione***) è un algoritmo ricorsivo che adotta una tecnica algoritmica detta ***divide et impera***. Essa può essere descritta come segue:

- il problema complessivo si suddivide in sottoproblemi di dimensione inferiore (***divide***);
- i sottoproblemi si risolvono ricorsivamente (***impera***);
- le soluzioni dei sottoproblemi si compongono per ottenere la soluzione al problema complessivo (***combina***).

Intuitivamente il mergesort funziona in questo modo:

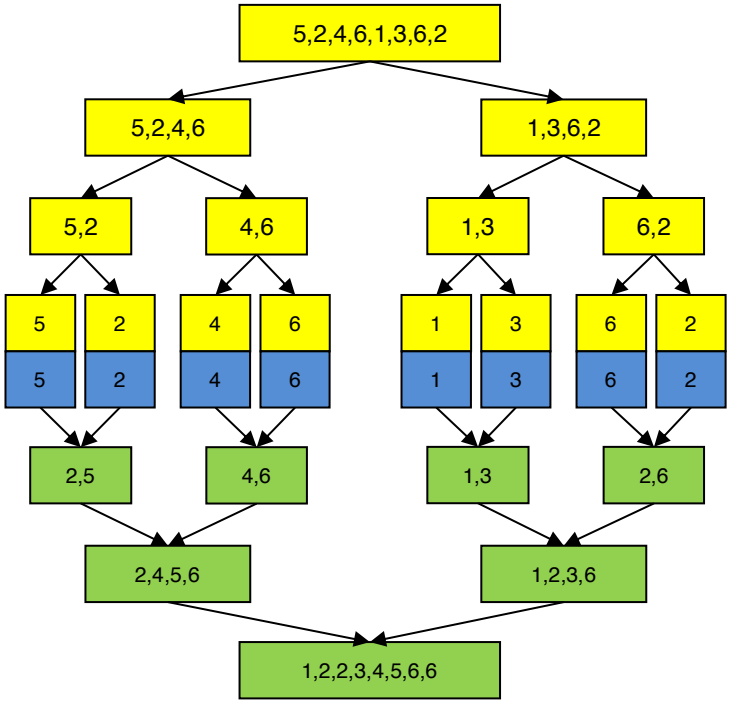
- ***divide***: la sequenza di *n* elementi viene divisa in due sottosequenze di *n/2* elementi ciascuna;
- ***impera***: le due sottosequenze di *n/2* elementi vengono ordinate ricorsivamente;
- ***passo base:*** la ricorsione termina quando la sottosequenza è costituita di un solo elemento, per cui è già ordinata;

- ***combina***: le due sottosequenze – ormai ordinate – di *n/2* elementi ciascuna vengono "fuse" in un'unica sequenza ordinata di *n* elementi.

Lo pseudocodice dell'algoritmo mergesort è il seguente:

```
def Merge_sort (A, indice_primo, indice_ultimo):
      if (indice_primo < indice_ultimo):
            indice_medio = (indice_primo+indice_ultimo)//2
            Merge_sort (A, indice_primo, indice_medio)
            Merge_sort (A, indice_medio + 1, indice_ultimo)
            Fondi (A, indice_primo, indice_medio, indice_ultimo)
```

Vediamone il funzionamento su un problema costituito di 8 elementi, dando per scontato (solo per il momento) il funzionamento della funzione `Fondi`:



Si osservi che le caselle gialle del disegno corripondono alle operazioni di suddivisione dell'array in sottoarray, cosa che avviene solo tramite le varie chiamate ricorsive. Le caselle azzurre del disegno invece indicano i casi base e quelle verdi corrispondono alle chiamate della funzione `Fondi`, le quali restituiscono porzioni di array ordinate sempre più grandi.

Illustriamo ora il funzionamento della fase di fusione. Essa sfrutta il fatto che le due sottosequenze sono ordinate, cosa che rende facile sfruttare una proprietà molto importante:

- il minimo della sequenza complessiva non può che essere il più piccolo fra i minimi delle due sottosequenze (se essi sono uguali, scegliere l'uno o l'altro non fa differenza);
- dopo aver eliminato da una delle due sottosequenze tale minimo, la proprietà rimane: il prossimo minimo non può che essere il più piccolo fra i minimi delle due parti rimanenti delle due sottosequenze.

L'ordinamento delle due porzioni di array permette di scandire ciascun elemento delle sequenze una volta sola, garantendo come vedremo un costo lineare. Lo pseudocodice è il seguente:

```
def Fondi (A; indice_primo, indice_medio, indice_ultimo):
1       i,j = indice_primo, indice_medio+1
2       B=[]
3       while ((i <= indice_medio) and (j <= indice_ultimo))
4               if (A[i] <= A[j]):
5                       B.append(A[i])
6                       i += 1
7               else:
8                       B.append(A[j])
9                       j += 1
10      while (i ≤ indice_medio) //il primo sottoarray non è terminato
11              B.append(A[i])
12              i += 1
13      while (j ≤ indice_ultimo) //il secondo sottoarray non è terminato
14              B.append(A[j])
15              j += 1
16      for i in range(len(B)):
17              A[primo+i] = B[i]
```

Analizziamo il costo dell'operazione di fusione e poi quella dell'algoritmo mergesort.

Nella fusione si effettuano:

- linee 1-2: un numero costante di operazioni preliminari (assegnazione dei valori a *i, j, k*), $\Theta(1)$;
- linee 3-9: un ciclo while, nel quale per ogni elemento di ciascuna delle due sottosequenze si compie un numero costante di operazioni. Si osservi che le istruzioni all'interno del ciclo while sono tutte operazioni costanti, perciò il costo computazionale del while è individuato dal numero di iterazioni: poiché si scorre almeno un array di dimensione n/2 (nel caso in cui uno dei due array contenga elementi tutti minori del primo elemento dell'altro array) ed al più due array di dimensioni n/2 (ad esempio nel caso in cui gli elementi del primo e del secondo array si alternano nell'ordinamento), questo ciclo while verrà eseguito un numero di volte proporzionale ad n/2, cioè $\Theta(n)$.
- linee 10-12 o, in alternativa, linee 13-15: un altro ciclo while, nel quale si ricopia nell'array B l'eventuale "coda" di una delle due sottosequenze, $O(n)$;
- linee 16-17: la copia dell'array B nell'opportuna porzione dell'array A, $\Theta(n)$.

Dunque il costo computazionale della fusione è $\Theta(n)$.

L'equazione di ricorrenza del mergesort è di conseguenza:

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

Essa ricade nel caso 2 del teorema principale, e la soluzione è $T(n) = \Theta(n \log n)$ – cfr. anche Es. 5.6.

Un'ultima osservazione a proposito del mergesort è che l'operazione di fusione non si può fare "in loco", cioè aggiornando direttamente l'array A, senza incorrere in un aggravio del costo. Infatti, in A bisognerebbe fare spazio via via al minimo successivo, ma questo costringerebbe a spostare di una posizione tutta la sottosequenza rimanente per ogni nuovo minimo, il che costerebbe $\Theta(n)$ operazioni elementari per ciascun elemento da inserire facendo lievitare quindi il costo computazionale della fusione da $\Theta(n)$ a $\Theta(n^2)$.

Esempio 6.1

Nonostante il Merge Sort funzioni in tempo $O(n \log n)$ mentre l'Insertion Sort in $O(n^2)$, i fattori costanti sono tali che l'Insertion Sort è più veloce del Merge Sort

per valori piccoli di *n*. Quindi, ha senso usare l'Insertion Sort dentro il Merge Sort quando i sottoproblemi diventano sufficientemente piccoli.

Si consideri una modifica del Merge Sort in cui il caso base si applica ad una porzione dell'array di lunghezza *k*, ordinata usando l'Insertion Sort. Le porzioni vengono combinate usando il meccanismo standard di fusione, con *k* che deve essere determinato.

Si determini il valore di *k* come funzione di *n* per cui l'algoritmo modificato ha lo stesso tempo di esecuzione asintotico del Merge Sort.

Soluzione.

La funzione descritta nel testo può essere scritta come segue, e richiamata la prima volta con `primo`=1, `ultimo`=*n* e `dim`= `ultimo`-`primo`+1=*n*:

```
def Merge_Insertion (A,k, primo, ultimo, dim):
if dim>k:
       medio =(primo+ultimo)//2
       Merge_Insertion (A,k, primo, medio, medio-primo+1)
       Merge_Insertion (A,k, medio+1, ultimo, ultimo-primo)
       Fondi(primo, medio, ultimo)
else: InsertionSort(primo, ultimo)
```

L'equazione di ricorrenza associata a questa funzione è:

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

$$T(k)=\Theta(k^2)$$

che può essere risolta per iterazione al modo seguente:

$$T(n)=2T(n/2)+\Theta(n)=\ldots=2^iT(n/2^i)+\sum_{j=0}^{i-1}2^j\Theta(\frac{n}{2^j})$$

Procedendo fin quando $n/2^i=k$ cioè fin quando $i=\log n/k$ si ha:

$$T(n)=2^{\log n/k}\,\Theta(k^2)+\Theta(n)\sum_{j=0}^{\log\frac{n}{k}+1}1=\Theta(n/k\cdot k^2)+\Theta(n)\log n/k$$

che è dell'ordine di $n \log n$ se $k=O(\log n)$.

6.3.2 Quicksort

L'algoritmo ***quicksort*** (***ordinamento veloce***) è un algoritmo che ha una caratteristica molto interessante: nonostante abbia un costo che nel caso peggiore è $O(n^2)$, nella pratica si rivela spesso la soluzione migliore per grandi valori di n perché:

- il suo tempo di esecuzione atteso è $\Theta(n \log n)$;
- i fattori costanti nascosti sono molto piccoli;
- permette l'ordinamento "in loco".

Esso quindi riunisce i vantaggi del selection sort (ordinamento in loco) e del merge sort (ridotto tempo di esecuzione). Ha però lo svantaggio dell'elevato costo computazionale nel caso peggiore.

Anche il quicksort sfrutta la tecnica algoritmica del divide et impera e può convenientemente essere espresso in forma ricorsiva:

- ***divide***: nella sequenza di n elementi viene selezionato un elemento detto ***pivot***. La sequenza viene quindi divisa in due sottosequenze: la prima delle due contiene elementi minori o uguali del pivot, la seconda contiene elementi maggiori o uguali del pivot;
- ***passo base*****:** la ricorsione procede fino a quando le sottosequenze sono costituite da un solo elemento;
- ***impera***: le due sottosequenze vengono ordinate ricorsivamente;
- ***combina***: non occorre alcuna operazione poiché la sequenza {elementi minori/uguali del pivot, ordinati}{elementi maggiori/uguali del pivot, ordinati} è già ordinata.

Lo pseudocodice dell'algoritmo quicksort è il seguente:

```
def Quick_sort (A, indice_primo, indice_ultimo):
     if (indice_primo < indice_ultimo):
          indice_medio = Partiziona (A, indice_primo, indice_ultimo)
          Quick _sort (A, indice_primo, indice_medio-1)
          Quick _sort (A, indice_medio + 1, indice_ultimo)
```

Lo pseudocodice dell'operazione di partizionamento è il seguente:

```
def Partiziona (A, indice_primo, indice_ultimo):
1       pivot =indice_ultimo //scelta arbitraria
2       i=indice_primo - 1;
3       for j in range(indice_primo,indice_ultimo):
4               if A[j]<=pivot
5                       i+=1
6                       A[i],A[j]=A[j],A[i]
7       A[i+1],A[indice_ultimo]=A[indice_ultimo],A[i+1]
8       return i+1
```

`Partiziona` seleziona il valore dell'elemento più a destra e lo usa come pivot per partizionare la sequenza in tre regioni: alla fine del procedimento la regione di sinistra conterrà solo elementi minori o uguali al pivot, quella di destra conterrà solo elementi maggiori o uguali al pivot, e quella centrale, formata da un solo elemento, contenente il pivot.

Prima di analizzare il costo computazionale del quicksort, vediamo il suo funzionamento su un piccolo esempio.

Sia dato l'array A=[2, 8, 7, 1, 3, 5, 6, 4]; la funzione `Partiziona` modifica l'array in modo che esso divenga A=[2, 1, 3, 4, 7, 5, 6, 8] e restituisce l'indice 3 (corrispondente al pivot con valore 4). A questo punto, la funzione `Quick_sort` viene richiamata ricorsivamente sui due sottoarray [2, 1, 3] e [7, 5, 6, 8]. Proseguendo le chiamate sui sottoarray che via via scaturiscono dai vari valori che vengono restituiti dalla funzione `Partiziona`, si giunge, infine, all'array ordinato.

Calcoliamo ora il costo computazionale. Nel partizionamento si effettuano:

- linee 1-2: un numero costante di operazioni preliminari, $\Theta(1)$;
- linee 3-6: una scansione della sequenza, compiendo un numero di operazioni che è pari alla dimensione dell'array in input: $\Theta(n)$;
- linee 7-8: un'operazione elementare finale, che posiziona il pivot nella sua posizione definitiva e ne ritorna l'indice, $\Theta(1)$.

Dunque il costo computazionale del partizionamento è $\Theta(n)$.

Il partizionamento, a seconda del valore del pivot, suddivide la sequenza di n elementi in due sottosequenze di dimensioni k e $(n-1-k)$ rispettivamente, con $0 \leq k \leq n-1$. In particolare, una delle due sequenze è vuota ogniqualvolta il valore del pivot risulti strettamente minore o strettamente maggiore di tutti gli altri elementi della sequenza.

L'equazione di ricorrenza del quicksort va dunque espressa come segue:

$$T(n) = T(k) + T(n-1-k) + \Theta(n), \text{ dove } 0 \leq k \leq n-1 \text{ e } T(1) = \Theta(1)$$

che non sappiamo come risovere con alcuno dei metodi illustrati precedentemente. Possiamo però valutarla in casi specifici: nel caso peggiore, in quello migliore e in quello medio.

Caso peggiore

Il caso peggiore è quello in cui, ad ogni passo, la dimensione di uno dei due sottoproblemi da risolvere è nulla. In tale situazione l'equazione di ricorrenza è:

$$T(n) = T(n-1) + \Theta(n)$$

Applicando il metodo iterativo otteniamo:

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

$$= \sum_{i=0}^{n-1} \Theta(n-i) = \Theta\left(\sum_{i=0}^{n-1}(n-i)\right) = \Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2).$$

Possiamo risolvere la stessa equazione di ricorrenza applicando il metodo dell'albero: otteniamo un albero degenere di n livelli, in cui ogni nodo ha un solo figlio. Il costo dei nodi decresce linearmente da n (nella radice) ad 1 (nell'unica foglia). Quindi il costo computazionale è:

$$T(n) = \Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2).$$

Caso migliore

Si puo' dimostrare che il caso caso migliore è quello in cui `Partiziona` ottiene due sottoproblemi il piu' bilanciato possibile, e in questo caso la ricorrenza diviene:

$$T(n) = 2T\left(\frac{n-1}{2}\right) + \Theta(n)$$

che, similmente a come già visto per il mergesort, ha soluzione *T(n) = Θ(n log n)*.

Caso medio

Valutiamo il costo computazionale del caso medio, nell'ipotesi che il valore del pivot suddivida con uguale probabilità *1/n* la sequenza da ordinare in due sottosequenze di dimensioni *k* ed *(n –1– k)*, per tutti i valori di *k* compresi fra *0* ed *(n – 1)*.

Sotto tale ipotesi il costo computazionale diviene:

- $T(n) = \frac{1}{n}\left[\sum_{k=0}^{n-1}\left(T(k) + T(n-1-k)\right)\right] + \Theta(n)$
- $T(0)=T(1) = \Theta(1)$

Ora, per ogni valore di *k = 0, 1, …, n – 1* il termine *T(i)* compare due volte nella sommatoria, la prima quando *k = i* e la seconda quando *k = n –1– i*.

Valutiamo dunque il valore di:

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

Utilizziamo il metodo di sostituzione, e quindi eliminiamo innanzi tutto la notazione asintotica:

- $T(n) = \frac{2}{n}\left[\sum_{q=0}^{n-1} T(q)\right] + hn$
- $T(0)=T(1) = k$

Calcoliamo anche *T(2)=2k+2h*. Ipotizziamo ora la soluzione:

$$T(n) \leq a\, n \log n$$

Sostituiamo la soluzione innanzi tutto nel caso base; *log 0* non è definito e *log 1=0,* per cui non possiamo utilizzare T(0) né *T(1),* e sostituiamo quindi in *T(2),* ottenendo:

$$2k+2h=T(2) \leq 2a \log 2 = 2a$$

che è vera per *a* opportunamente grande ($a \geq k+h$).

Sostituendo, invece, nell'equazione generica, e ricordando che *log 0* non è definito, ma è noto il valore di *T(0),* possiamo scrivere:

$$T(n) = \frac{2}{n}\sum_{q=1}^{n-1} T(q) + hn + k \leq \frac{2}{n}\sum_{q=1}^{n-1}(aq\ log\ q) + hn + k =$$

(*) $$= \frac{2a}{n}\sum_{q=1}^{n-1} q \log q + hn + k.$$

Ora, la sommatoria $\sum_{q=1}^{n-1} q \log q$ può essere riscritta come:

$$\sum_{q=1}^{n-1} q \log q = \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q \log q + \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q \log q$$

Osserviamo che:

- nella prima delle due sommatorie a destra dell'uguale ogni termine *log q* è minore o uguale di *log* $\frac{n}{2}$ *= log n – 1*;
- nella seconda ogni termine *log q* è minore o uguale di *log n*.

Dunque possiamo scrivere:

$$\sum_{q=1}^{n-1} q \log q \leq \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q(\log n - 1) + \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q \log n = (\log n - 1)\sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q + \log n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q =$$

$$= \log n \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q + \log n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q =$$

$$= \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q \leq$$

$$\leq \log n \frac{(n-1)n}{2} - \frac{1}{2}\left(\frac{n}{2} - 1\right)\left(\frac{n}{2}\right) \leq \frac{1}{2} n(n-1) \log n - \frac{1}{4}\left(\frac{n}{2} - 1\right)(n-1)$$

Ritorniamo ora alla relazione (*) per inserirvi l'espressione trovata:

$$T(n) \leq \frac{2a}{n}\sum_{q=1}^{n-1} q \log q + cn + k \leq$$

$$\leq \frac{2a}{n}\left(\frac{1}{2} n(n-1) \log n - \frac{1}{4}\left(\frac{n}{2} - 1\right)(n-1)\right) + cn + k =$$

$$= an \log n - \frac{an}{4} + \frac{a}{2} + cn + k$$

scegliendo *a* sufficientemente grande, si ha $\left[cn - \frac{an}{4} + \frac{a}{2} + k\right] \leq 0$ e per tale *a* si avrà: $T(n) \leq an \log n + \left[cn - \frac{an}{4} + \frac{a}{2}\right] \leq an \log n$

deducendo che $T(n) = O(n \log n)$ nel caso medio.

Poiché il costo computazionale del caso migliore del Quicksort rappresenta una limitazione inferiore al costo computazionale nel caso medio, si ha che *T(n)=Ω(n log n)*. Dalle due relazioni trovate su *T(n)* si deduce che nel caso medio *T(n)=Θ(n log n).*

Si noti che, adottando un approccio più "naïf" non riusciremmo a dimostrare la relazione *T(n)=O(n log n)*. Infatti, riconsiderando la sommatoria $\sum_{q=0}^{n-1} q \log q$, potremmo basarci sul fatto che *log q ≤ log n* per derivare quest'altra disuguaglianza, più semplice della precedente:

$$\sum_{q=1}^{n-1} q \log q \leq \log n \sum_{q=1}^{n-1} q = \log n \frac{(n-1)n}{2}$$

Sostituendo questa disuguaglianza nella relazione (*) otteniamo:

$$T(n) \leq \frac{2a}{n}\left(\log n \frac{n(n-1)}{2}\right) + cn + k = a(n-1)\log n + cn + k$$

che, ovviamente, non sarà mai $\leq an \log n$, visto che *a log n* non sarà definitivamente abbastanza grande per compensare *cn.*

Un'ultima osservazione a proposito del caso medio. L'analisi fin qui sviluppata è valida nell'ipotesi che il valore del pivot sia equiprobabile e, quando questo è il caso, il quicksort è considerato l'algoritmo ideale per input di grandi dimensioni.

A volte però l'ipotesi di equiprobabilità del valore del pivot non è soddisfatta (ad esempio quando i valori in input sono "poco disordinati") e le prestazioni dell'algoritmo degradano.

Per ovviare a tale inconveniente si possono adottare delle tecniche volte a ***randomizzare*** la sequenza da ordinare, cioè volte a disgregarne l'eventuale regolarità interna. Tali tecniche mirano a rendere l'algoritmo indipendente dall'input, e quindi consentono di ricadere nel caso medio con la stessa probabilità che nel caso di ipotesi di equiprobabilità del pivot:

1. prima di avviare l'algoritmo, alla sequenza da ordinare viene applicata una permutazione degli elementi generata casualmente;
2. l'operazione di partizionamento sceglie casualmente come pivot il valore di uno qualunque degli elementi della sequenza anziché sistematicamente il valore di quello più a sinistra.

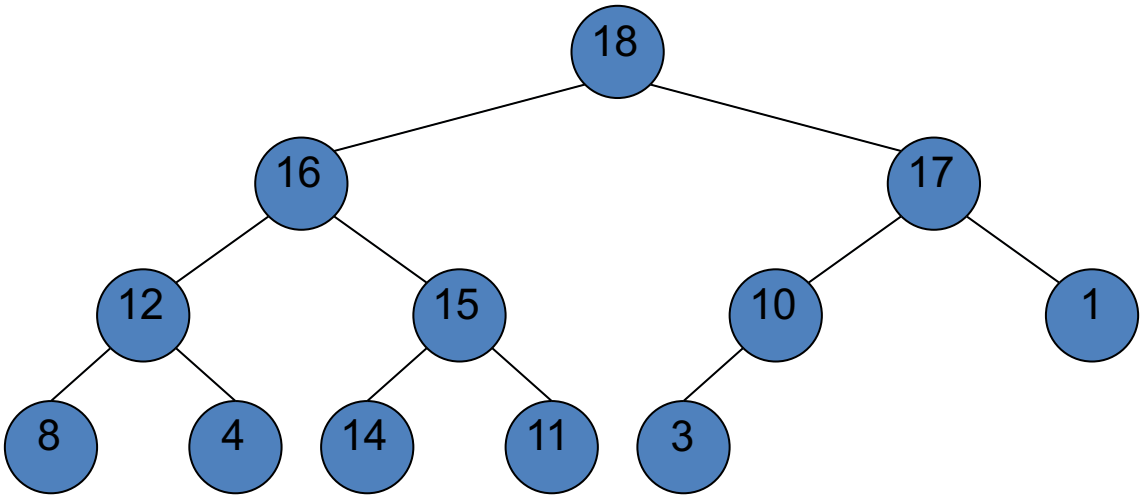
6.3.3 Heapsort

L'algoritmo ***heapsort*** è un algoritmo di ordinamento piuttosto complesso che esibisce ottime caratteristiche: come mergesort ha un costo computazionale di *O(n log n)* anche nel caso peggiore, e come selection sort ordina in loco.

Inoltre, questo algoritmo è l'occasione per iniziare a illustrare un concetto molto importante, quello di ***struttura dati***. Sfrutta, infatti, una opportuna organizzazione dei dati che garantisce una o più specifiche proprietà, il cui mantenimento è essenziale per il corretto funzionamento dell'algoritmo.

In questo caso la struttura dati utilizzata si chiama ***heap*** (o ***heap binario***). Esso è un albero binario *completo o quasi completo*, ossia un albero binario in cui tutti i livelli sono pieni, tranne l'ultimo, i cui nodi sono addensati a sinistra, con la proprietà che la chiave su ogni nodo è maggiore o uguale alla chiave dei suoi figli (proprietà di ordinamento verticale).

In figura è mostrato un esempio di heap.



Il modo più naturale per memorizzare questa struttura dati è utilizzare un array *A*, con indici che vanno da *1* fino al numero di nodi dell'heap, *heap_size*, i cui elementi possono essere messi in corrispondenza con i nodi dell'heap:

- l'array è riempito a partire da sinistra; se contiene più elementi del numero *heap_size* di nodi dell'albero, allora i suoi elementi di indice > *heap_size* non fanno parte dell'heap;
- ogni nodo dell'albero binario corrisponde a uno e un solo elemento dell'array *A*;

- la radice dell'albero corrisponde ad *A[0]*;
- il figlio sinistro del nodo che corrisponde all'elemento *A[i],* se esiste, corrisponde all'elemento A[2i+1]: ***left(i) = 2i+1***;
- il figlio destro del nodo che corrisponde all'elemento *A[i],* se esiste, corrisponde all'elemento *A[2i + 2]*: ***right(i) = 2i+2***;
- il padre del nodo che corrisponde all'elemento A[i] corrisponde all'elemento A[$\left\lfloor \frac{i-1}{2} \right\rfloor$]: ***parent(i) =*** $\left\lfloor \frac{i-1}{2} \right\rfloor$.

In figura è mostrato l'array corrispondente all'heap rappresentato in precedenza.

18	16	17	12	15	10	1	8	4	14	11	3

Con questa implementazione, la proprietà di ordinamento verticale implica che per tutti gli elementi tranne *A[0]* (poiché esso corrisponde alla radice dell'albero e quindi non ha genitore) vale:

$$A[i] \leq A[parent(i)].$$

Poiché lo heap ha tutti i livelli completamente pieni tranne al più l'ultimo, la sua altezza è $\Theta(\log n)$: questa proprietà, come vedremo, è fondamentale ai fini del costo computazionale dell'heapsort.

Si noti che l'elemento massimo risiede nella radice, quindi può essere trovato in tempo $O(1)$.

L'algoritmo heapsort si basa su tre componenti distinte:

- ***Heapify***: ripristina la proprietà di heap, posto che i due sottoalberi della radice siano ciascuno uno heap; richiede un tempo $O(\log n)$;
- ***Build_heap***: trasforma un array disordinato in uno heap, sfruttando Heapify; richiede un tempo $O(n)$;
- ***Heapsort***: l'algoritmo di ordinamento vero e proprio; ordina in loco un array disordinato, sfruttando Build_heap ; richiede un tempo $O(n \log n)$.

Heapify

Questa funzione ha lo scopo di mantenere la proprietà di heap, sotto l'ipotesi che nell'albero su cui viene fatta lavorare sia garantita la proprietà di heap per entrambi i sottoalberi (sinistro e destro) della radice. Di conseguenza l'unico nodo

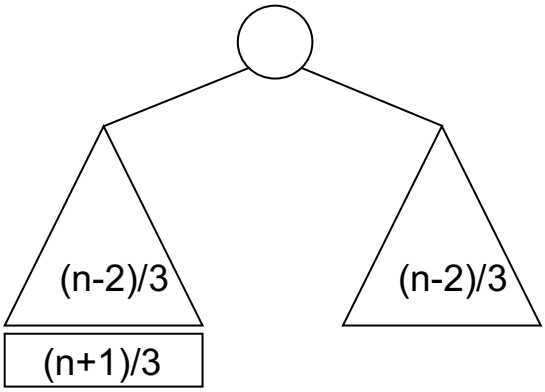
che può violare la proprietà di heap è la radice dell'albero, che può essere minore di uno o di entrambi i figli.

In tal caso, la funzione scambia la radice col maggiore dei suoi due figli. Questo scambio può rendere non più soddisfatta la proprietà di heap nel sottoalbero coinvolto nello scambio (ma non nell'altro), per cui è necessario riapplicare ricorsivamente la funzione a tale sottoalbero. In sostanza, la funzione muove il valore della radice lungo un opportuno cammino verso il basso finché tutti i sottoalberi le cui radici si trovano lungo quel cammino risultano avere la proprietà di heap. Lo pseudocodice di Heapify, che lavora su array, è il seguente.

```
def Heapify (A, i, heap_size)
      L=2*i+1; R=2*i+2
      indice_massimo=i
      if ((L < heap_size) and (A[L] > A[i])):
            indice_massimo=L
      if ((R ≤ heap_size) and (A[R] > A[indice_massimo])):
            indice_massimo=R
      if (indice_massimo!=i):
            A[i], A[indice_massimo]= A[indice_massimo], A[i]
            Heapify (A, indice_massimo, heap_size)
```

Il costo computazionale di `Heapify` è $\Theta(1)$ per la sistemazione della radice dell'albero, più il costo della sistemazione (ricorsiva) di uno dei suoi due sottoalberi.

Ora, è facile convincersi che i sottoalberi della radice non possono avere più di $2n/3$ nodi, situazione che accade quando l'ultimo livello è pieno esattamente a metà:



Dunque, nel caso peggiore, l'equazione di ricorrenza risulta essere (all'incirca):

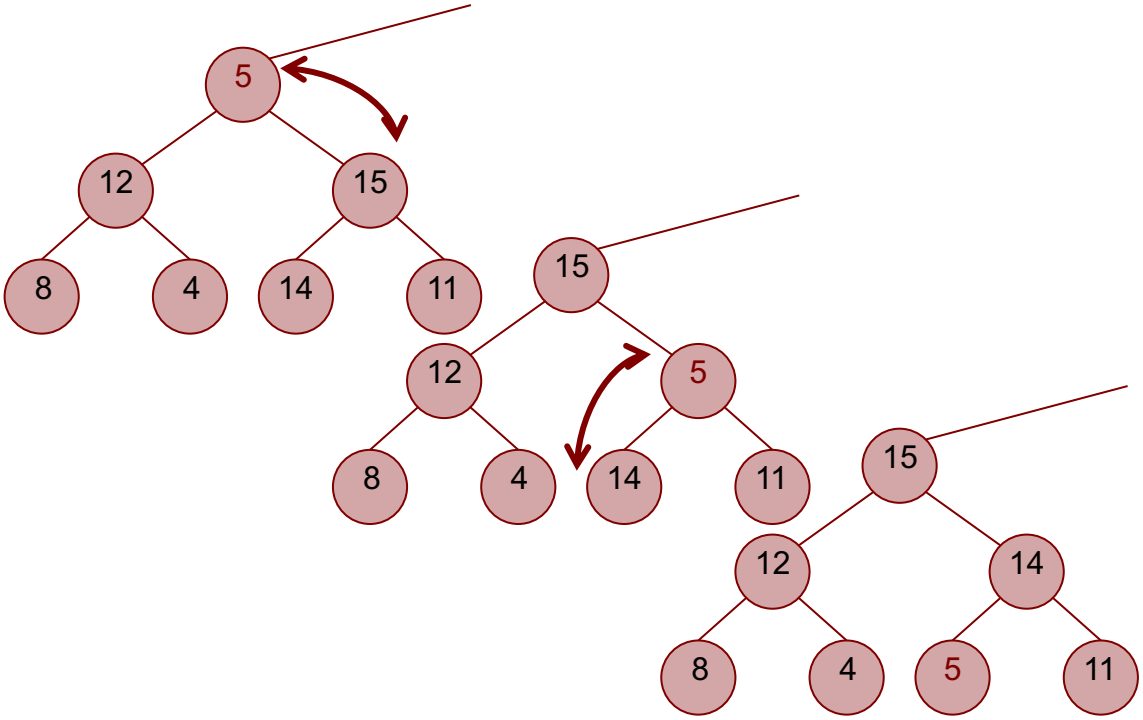
$$T(n) = T(2n/3) + \Theta(1)$$

che ricade nel caso 2 del teorema principale ed ha soluzione:

$$T(n) = O(\log n).$$

In alternativa, si può considerare che Heapify effettua un lavoro $\Theta(1)$ su ogni nodo lungo un cammino la cui lunghezza è limitata da $O(\log n)$, per cui l'equazione di ricorrenza diviene: $T(h) = T(h-1) + \Theta(1)$

da cui si riottiene lo stesso risultato.



La figura precedente illustra un esempio del funzionamento di Heapify.

Build_heap

Possiamo usare Heapify per trasformare qualunque array contenente *n* elementi in uno heap.

Poiché tutti gli elementi dell'array dalla posizione di indice $\left\lfloor \left\lfloor \frac{n}{2} \right\rfloor + 1 \right\rfloor$ in poi sono le foglie dell'albero, e quindi ciascuno di essi è uno heap di 1 solo nodo, su di essi non vi è alcuna operazione da fare.

Build_heap lavora quindi solo sugli altri elementi dell'albero, operando su di essi in un ordine tale (da destra verso sinistra e dal basso verso l'alto, cioè bottom-up) da garantire che i sottoalberi di ogni nodo *i* siano heap prima che si lavori sul nodo stesso. In termini di corrispondente lavoro sull'array, Build_heap lavora con un approccio da destra verso sinistra.

Lo pseudocodice di Build_heap è il seguente:

```
def Build_heap(A):
     for i in reversed(range(len(A)//2)):
          Heapify (A, i, heap_size)
```

Il tempo richiesto da Heapify applicata ad un nodo che è radice di un albero di altezza *h* (**nodo di altezza *h***) è *O(h)* per quanto già detto; inoltre, il numero di nodi che sono radice di un sottoalbero di altezza *h* è al massimo $\left\lceil \frac{n}{2^{h+1}} \right\rceil$, e quindi il costo totale di Build_heap è:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

Ora, ricordando che se |x| < 1 allora $\sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2}$ e ponendo in tale formula *x = ½*, otteniamo:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$$

e dunque possiamo scrivere:

$$T(n) = O\left(\frac{n}{2}\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) < O\left(\frac{n}{2}\sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O\left(\frac{n}{2}2\right) = O(n).$$

Concludiamo con l'osservare che un'analisi meno accurata, ma comunque corretta, del costo computazionale di questo algoritmo può essere condotta come segue: il ciclo viene eseguito *n/2* volte, e al suo interno vi è unicamente la chiamata ricorsiva a Heapify che, nel caso peggiore, richiede tempo *O(h) = O(log n)*. Ne segue che Build_heap ha costo nel caso peggiore di *O(n log n)*. Ovviamente, questo calcolo non conduce ad un valore stretto, ma sarebbe comunque sufficiente per garantire costo *O(n log n)* all'algoritmo di heap sort, come vedremo nel seguito.

Heapsort

L'algoritmo Heapsort inizia trasformando un array di dimensione *n* in un heap (di *n* nodi), mediante Build_heap.

Fatto questo, il massimo dell'array è in *A[0]* e, per metterlo nella corretta posizione, basta scambiarlo con *A[n]*.

Dopo lo scambio, la dimensione dello heap viene ridotta da *n* ad *(n – 1)*, e si può osservare che:

- i due sottoalberi della radice sono ancora degli heap: infatti hanno mantenuto la proprietà dell'ordinamento, sono entrambi alberi binari completi o quasi completi ed è stata tolta la foglia più a destra;
- solo la nuova radice (ex foglia più a destra) può violare la proprietà del nuovo heap di dimensione (n – 1).

E' quindi sufficiente:

- ripristinare la proprietà di heap sui residui *(n – 1)* elementi con Heapify;
- scambiare il nuovo massimo *A[0]* col penultimo elemento;
- riapplicare il procedimento riducendo via via la dimensione dell'heap a *(n – 2), (n –3),* ecc., fino ad arrivare a 2.

Lo pseudocodice di Heapsort è il seguente:

```
def Heapsort(A):
      Build_heap(A)
      for x in reversed(range(1,len(A))):
            A[0], A[x]= A[x],A[0]
            Heapify(A, 0, x)
```

Il costo computazionale di `Heapsort` è *O(n log n)*, poiché `Build_heap` ha costo *O(n)* e ciascuna delle *(n – 1)* chiamate di `Heapify` ha costo *O(log n)*. Quindi:

O(n) + O((n – 1)log n) = O(n log n).

Osserviamo che, in Python, il comando ***sort*** che si può applicare ad un oggetto *list* non è un comando semplice e non ha costo costante: esso viene implementato con il Quicksort, ed ha pertanto costo computazionale *O(n log n)* nel caso medio.

6.3.4 Ordinamento in tempo lineare: counting sort e bucket sort

E' possibile ordinare *n* elementi in un tempo inferiore a *Ω(n log n)*?

Il teorema che stabilisce una limitazione inferiore al costo degli algoritmi di ordinamento sembrerebbe dire di no; in realtà la risposta è si, purchè l'algoritmo di ordinamento non sia basato sui confronti fra elementi poiché, in tal caso, come abbiamo visto nessun algoritmo può scendere sotto tale limite.

E' possibile risolvere il problema dell'ordinamento mediante soluzioni che non siano basate sul confronto di elementi, e quindi non soggette alla limitazione inferiore sopra citata.

Un esempio è il ***counting sort*** (***algoritmo basato sul conteggio***). Esso opera sotto l'assunzione che ciascuno degli *n* elementi da ordinare sia un intero di valore compreso in un intervallo *[0..k]* ed ha un costo di *Θ(n+k)*. Se *k = O(n)* allora l'algoritmo ordina *n* elementi in tempo lineare, cioè *Θ(n)*.

L'idea è quella di fare in modo che il valore di ogni elemento della sequenza determini direttamente la sua posizione nella sequenza ordinata.

Questo semplice schema (banale se gli elementi costituiscono una permutazione di *0..n*) deve essere leggermente complicato per gestire l'eventualità che nella sequenza da ordinare vi siano più elementi con lo stesso valore o manchi un qualche valore.

Oltre all'array A che contiene gli *n* elementi da ordinare, usiamo un array C di appoggio, contenente *k* celle.

```
def Counting_sort (A)
       k=max(A)
       n=len(A)
       C[0]*(k+1) //crea un array di k+1 locazioni, tutte inizializzate a 0
       for j in range(n):
              C[A[j]]+= 1  //C[i] ora contiene il numero di elementi uguali a i
       j = 0
       for j in range (k)
             while (C[i] > 0)
                    A[j]=i; j+= 1
                    C[i]-= 1
```

La prima istruzione calcola il massimo dell'array *A* e costa, quindi, $\Theta(n)$.

L' inizializzazione a 0 dell'array *C* costa $\Theta(k)$.

Il primo ciclo conta, per ciascun indice di *C*, il numero di elementi di A aventi tale valore.

Il secondo ciclo scandisce l'array *C* e, al suo interno, per ogni valore di *i*, un ciclo `while` scrive *C[i]* copie di *i* sull'array *A*. Il numero complessivo di iterazioni del ciclo `while` è pari alla somma dei valori contenuti in C, che è esattamente *n*, quindi il costo computazionale è:

$$\Theta(n) + \Theta(k) + \Theta(n) + \Theta(k+n) = \Theta(max(k,n)) = \Theta(n) \text{ se } k = O(n).$$

Segue un piccolo esempio del funzionamento dell'algoritmo di Counting Sort.

Sia dato l'array *A = [2, 5, 1, 2, 3, 3, 5, 0, 3, 1].* In questo caso *n = 10* e *k = 5.*

Dopo l'esecuzione del ciclo `for j in range(n)`, l'array *C* contiene i seguenti valori:

$$C = [1, 2, 2, 3, 0, 2]$$

ed, ovviamente, risulta che la somma dei valori contenuti in *C* è proprio pari ad *n*.

A questo punto, è facile ricostruire l'array *A* ordinato, scrivendo dapprima 3 occorrenze del valore 1, poi 2 occorrenze del valore 2, e così via, fino alla fine dell'array *C*:

$$A = [0, 1, 1, 2, 2, 3, 3, 3, 5, 5].$$

Si noti che lo pseudocodice sopra illustrato è adeguato solamente se non vi sono "dati satellite", ossia ulteriori dati collegati a ciascun elemento da ordinare. In tal caso si deve ricorrere ad una soluzione un po' più complicata, nella quale, oltre all'array A che contiene gli *n* elementi da ordinare, sono necessarie due strutture d'appoggio:

- un array ausiliario *B* di *n* elementi, che alla fine contiene la sequenza ordinata;
- un array *C* di appoggio, contenente *k* elementi, per effettuare i conteggi.

```
Funzione Counting_sort_con_Dati_Satellite (A)
        k=max(A); n=len(A)
        C=[0]*(k+1); B=[0]*n
        for j in range(n)
                C[A[j]]+=1  //C[i] ora contiene il numero di elementi = i
        for i in range(1,k)
                C[i]+=C[i-1] //C[i] ora contiene il numero di elementi ≤ i
        for j in range(n,-1)
                B[C[A[j]]]=A[j]]
                C[A[j]]-=1
        return B
```

Il ciclo finale colloca ogni elemento *A[j]* dell'array originale nella giusta posizione dell'array *B*, in funzione di quanti elementi minori o uguali dell'elemento stesso sono stati conteggiati in *C*. Dopo aver collocato l'elemento, si decrementa il contatore degli elementi minori o uguali ad esso, in quanto l'elemento appena piazzato deve ovviamente essere levato dal conteggio.

Si noti che questo algoritmo ha una proprietà che in determinati contesti può essere importante, la ***stabilità***: gli elementi di uguale valore sono piazzati in B nello stesso ordine relativo in cui appaiono in A.

L'algoritmo consiste di una successione di quattro cicli, due di *k* iterazioni e due di *n* iterazioni. Dunque il costo computazionale è:

$$2\Theta(k) + 2\Theta(n) = \Theta(max(k,n)) = \Theta(n) \text{ se } k = O(n).$$

Un altro algoritmo interessante, che ha un costo computazionale medio lineare, è il **bucket sort.** Come il counting sort, il bucket sort è efficiente perché fa delle ipotesi sull'input. In particolare, assume che gli *n* valori dati in input da ordinare siano equamente distribuiti in un intervallo *[1..k]*, dove non c'è alcuna ipotesi su *k*.

L'idea del bucket sort consiste nel dividere l'intervallo *[1..k]* in *n* sottointervalli di uguale ampiezza *k/n*, detti bucket. A causa dell'ipotesi di equidistribuzione, non

ci si aspetta che molti valori dell'input cadano in ciascun intervallo, anzi -in media- essi saranno in numero costante.

Volutamente lasciamo lo pseudocodice del bucket sort ad un livello più alto di astrazione rispetto al resto, poiché introduce un array ausiliario di lunghezza *n* di liste (struttura dati che non abbiamo ancora incontrato e, che per il momento, possiamo vedere come una sequenza di lunghezza variabile di oggetti), ciascuna delle quali conterrà tutti gli elementi dati in input che cadono in un certo sottointervallo; più in dettaglio, la lista *i* conterrà tutti i valori compresi tra *(i-1)k/n* (escluso) *e ik/n* (compreso).

```
Funzione Bucket_sort (A; k, n: intero)
      for i = 1 to n
            inserisci A[i] nella lista B[⌈A[i]n/k⌉]
      for i=1 to n
            ordina la lista B[i] con Insertion Sort
      concatena insieme le liste B[1], …, B[n] in quest'ordine
      copia la lista unificata in A
      return
```

Per quanto riguarda il costo computazionale dell'algoritmo di bucket sort, è chiaro che esso dipende fortemente dalla lunghezza delle liste B[i]. Poiché, come abbiamo già evidenziato, tale lunghezza è *mediamente* costante, è facile vedere che il costo computazionale medio di questo algoritmo è lineare. Invece, nel caso peggiore, esso sarà quadratico.