Algoritmi e strutture dati

“Introduzione agli algoritmi e strutture dati”

Thomas H. Cormen, Charles E. Leiserson,

Ronald L. Rivest, Clifford Stein - 3.a edizione. Mc-Graw Hill, 2010

SOMMARIO

PROGRAMMA

- **Introduzione**
  - Nozioni fondamentali: Algoritmo, problema, istanza
  - Pseudocodice e macchina RAM
  - Problema dell'ordinamento di un vettore: Algoritmo **insertion sort**
  - Limiti asintotici inferiori e superiori
  - Il principio di induzione
  - Nozioni varie: base, tetto, sommatorie, approssimazioni, polinomi
- **Analisi di algoritmi**
  - Dimensione dei dati di un problema
  - Complessita' computazionale: caso pessimo, ottimo e medio
  - Valutazione dei tempi di esecuzione
- **Ricorsione**
  - Definizioni ricorsive di insiemi
  - Algoritmi ricorsivi
  - Induzione e ricorsione
- **Approccio Divide-et-Impera**
  - Definizione
  - Ordinamento di un vettore con algoritmo **Merge-Sort**
- **Valutazione del tempo di esecuzione di algoritmi ricorsivi: equazioni di ricorrenza**
  - Definizioni
  - Metodo Sostituzione
  - Metodo Iterativo e alberi di ricorsione
  - Metodo Principale
- **Ordinamento in loco**
  - Algoritmo **Quick-sort**
  - Quick-sort Randomizzato
- **Algoritmo di ordinamento Heapsort**
  - Grafi e Alberi: nozioni fondamentali
  - Nozione di Heap
  - **Heapsort**
- **Altri algoritmi di ordinamento**
  - **Selection sort**
  - **Insertion Sort**
  - Ordinamento in tempo lineare
  - Limite inferiore per ordinamenti basati su confronti
  - **Counting sort**
  - Concetto di stabilita'
  - **Radix sort**
- **Tipi astratti di dati e strutture dati - Introduzione**
  - La nozione di tipo astratto (ADT, Abstract Data Type)
  - Insiemi dinamici
  - Operazioni fondamentali su insiemi dinamici
- **Liste concatenate**
  - Liste semplici e doppie
  - Liste circolari
  - Uso di sentinelle

  - Operazioni fondamentali
  - Gestione di liste mediante array multipli
- **Pile (stack)**
  - Inserimento (push), cancellazione (pop)
  - Implementazioni: array, lista concatenata
- **Code (queue)**
  - Inserimento (enqueue), cancellazione (dequeue), attraversamento
  - Implementazioni: array, lista concatenata
- **Insiemi**
  - Definizioni e operazioni fondamentali
  - Realizzazione tramite liste
  - Realizzazione tramite vettori caratteristici
- **Alberi**
  - Nozione di grafo e proprieta'
  - Alberi generici: definizione ricorsiva
  - Nozioni generali: nodo, figlio, antenato, discendente, altezza.
  - Implementazione: record con puntatori; varianti
- **Alberi binari**
  - Attraversamento: ordine anticipato, simmetrico, posticipato
  - Alberi binari di ricerca e operazioni fondamentali: inserimento, cancellazione, ricerca, max, min, successore, predecessore
- **Tabelle di hash**
  - Indirizzamento diretto
  - Funzioni di hash
  - Risoluzione di collisioni tramite liste concatenate
  - Risoluzione di collisioni tramite indirizzamento aperto
  - Scansione lineare: inserimento, ricerca, cancellazione
  - Scansione quadratica
  - Doppio hashing: inserimento, ricerca, cancellazione

CAPITOLI

- Cap. 1 (tutto)
- Cap. 2 (tutto)
- Cap. 3 (tutto)
- Cap. 4: tutto tranne Sez. 4.1, 4.2 e 4.6
- Cap. 6 (tutto)
- Cap. 7 (tutto)
- Cap. 8: tutto tranne Sez. 8.4
- Cap. 9: tutto tranne Sez. 9.3
- Cap. 10 (tutto)
- Cap. 11: tutto tranne Sez. 11.5
- Cap. 12: tutto tranne Sez. 12.4

1 INTRODUZIONE: KHAN ACADEMY

Una definizione è “una sequenza di azioni per completare un compito”.

Un buon algoritmo è fatto da:

- **Correttezza**: porta sempre a termine correttamente il suo compito
- **Efficienza**: compie la sua funzione nel minor numero possibile di passi

Viene utilizzato il sistema di analisi asintotica per analizzarli indipendentemente dalla macchina sul quale vengono fatti funzionare.

1.1 PSEUDOCODICE

Lo pseudocodice è uno strumento per descrivere qualitativamente un algoritmo. Le istruzioni vengono descritte tramite la loro funzione e non tramite la loro implementazione. Ad esempio l’algoritmo descritto dopo è uno pseudocodice.

1.2 RICERCA BINARIA

La ricerca binaria è un algoritmo efficiente per ricercare un oggetto in una lista ordinata di oggetti. Funziona dividendo a metà la lista e procedendo ricorsivamente alla divisione della parte considerata fino a quando viene trovato l’oggetto cercato.

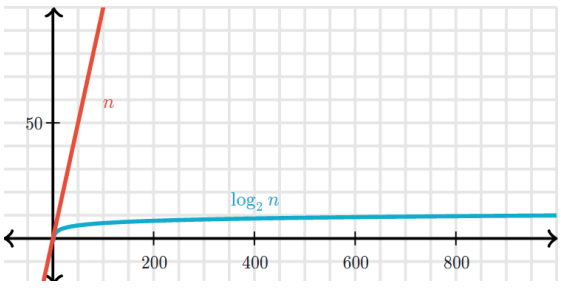
Ecco ad esempio un algoritmo per la ricerca di un numero in una lista ordinata di n numeri

1. Min = 1, max = n
2. Prova con la media tra min e max arrotondata per difetto
3. Se è stato trovato, stop
4. Se è troppo basso, posiziona min al numero selezionato
5. Se è troppo alto, posiziona max al numero selezionato
6. Torna allo step 2

1.2.1 Tempo di esecuzione per la ricerca binaria

Vediamo come analizzare il massimo numero di tentativi richiesti per la ricerca.

L’idea è che quando la ricerca non trova il risultato, il numero di possibilità si dimezza. Quindi, ad ogni errore, il numero di elementi dell’array è ridotto ad un sottoinsieme. Dimezzando di volta in volta, si descrive un logaritmo in base 2 di n. il numero massimo di tentativi è $(\log_2 n)+1$. Nel caso n non sia una potenza di due, si utilizza il valore che approssima per difetto n. ad esempio per 300, il numero massimo di volte sarà $(\log_2 300)+1$, ovvero 8.23+1 =9.23, quindi 9 tentativi massimi.



Ora, comparandolo ad esempio alla ricerca lineare (che ha come caso peggiore n tentativi), vediamo come la ricerca binaria sia più efficiente.

1.3 NOTAZIONE ASINTOTICA

Fino ad ora abbiamo cercato il numero massimo di tentativi richiesti per trovare il valore desiderato ma ciò che vogliamo veramente è il tempo richiesto.

Dobbiamo tenere da conto due cose:

- Dobbiamo determinare quanto il programma richieda in termini di grandezza dell'input. Infatti, intuitivamente, comprendiamo che maggiore è la grandezza dell'input, più tempo ci metterà l'algoritmo
- La seconda è cercare il tasso di crescita della funzione del tempo in pase all'input

Ad esempio, immaginiamo un algoritmo che gestisce un input di grandezza n e che ci mette $6n^2 + 100n + 300$ istruzioni. $6n^2$ diventa più grande con l'aumentare della grandezza dell'input rispetto agli altri due termini. Quindi diremo che l'algoritmo cresce con un ritmo di n^2.

Eliminando i termini meno significativi e i coefficienti numerici, possiamo concentrarci sulla parte importante del tempo di esecuzione: il tasso di crescita. Quando eliminiamo questi orpelli, stiamo utilizzando l'annotazione asintotica. Ci sono tre forme:

- Θ (theta) grande
- O grande
- Ω (Omega) grande

1.3.1 Θ grande

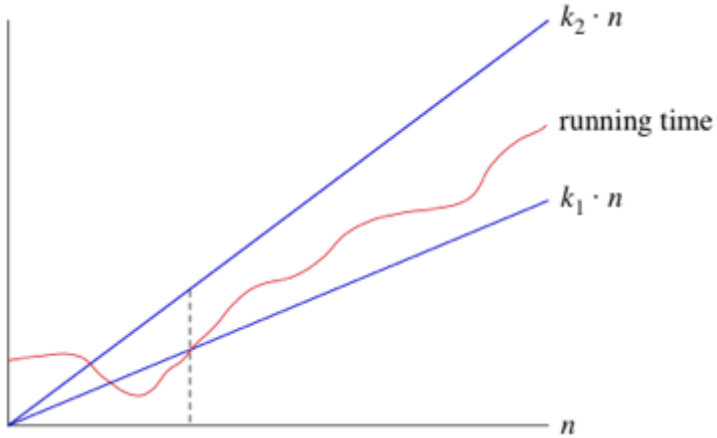
```
1  var doLinearSearch = function(array, targetValue) {
2    for (var guess = 0; guess < array.length; guess++) {
3      if (array[guess] === targetValue) {
4          return guess;  // found it!
5      }
6    }
7    return -1;  // didn't find it
8  };
```

Prendiamo `array.length` come n. il numero massimo di iterazioni è n e questo caso peggiore accade quando il numero non è nell'array.

Ogni iterazione, l'algoritmo deve:

- Comparare guess e array.length
- Comparare array[guess] con il valore obiettivo
- Dare guess come valore di ritorno eventualmente
- Incrementare guess

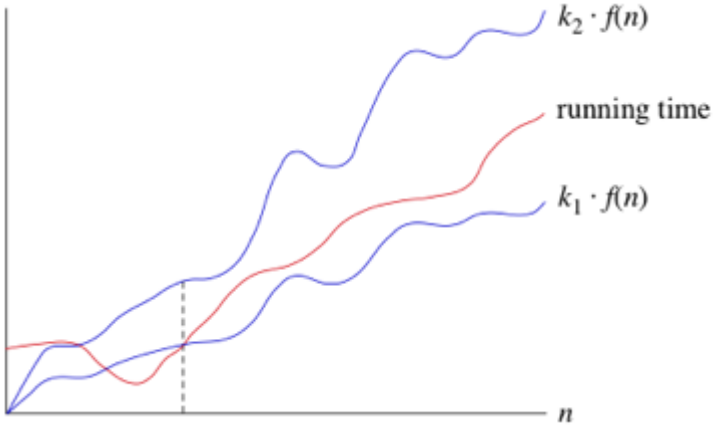
Ognuna di queste istruzioni richiede un tempo costante ogni volta che viene eseguita. Se il ciclo for itera n volta, il tempo richiesto per le n iterazioni è di $c_1 * n$ dove c1 è la somma dei tempi di computazione di un ciclo. C_1 dipende da varie cose: il linguaggio di programmazione, la macchina, il compilatore o l'interprete.



Questo codice ha anche una premessa e l'eventuale return -1, chiamiamo queste aggiunte c_2, quindi il tempo totale del caso peggiore è $c_1 * n + c_2$. Come abbiamo spiegato, il coefficiente c1 e il fattore costante c2 sono poco significativi per quanto riguarda il ritmo di crescita. La parte significativa è che nel caso peggiore la ricerca lineare cresce al ritmo di n. La notazione utilizzata è Θ(n).

Quando diciamo che un tempo di esecuzione è Θ(n), significa che quando n diventa abbastanza grande, il tempo di esecuzione è compreso tra $k_1 * n$ e $k_2 * n$ per alcune costanti k_1 e k_2.

Per piccoli valori di n, non ci interessa comparare il tempo di esecuzione con $k_1 * n$ e $k_2 * n$. Ma una volta che n diventa sufficientemente grande, il tempo di esecuzione diventa sempre compreso tra le due rette. Fino a quando k_1 e k_2 esistono, possiamo dire che il tempo di esecuzione è Θ(n).



Non siamo limitati a solo n nella notazione Θ grande. Possiamo anche usare funzioni come n^2, $nlog_2n$, o altre funzioni con n. ecco ad esempio il tempo di esecuzione di Θ(f(n)). Una volta che n diventa sufficientemente grande, verrà sempre compreso tra $k_1 * f(n)$ e $k_2 * f(n)$.

Nella pratica semplicemente togliamo costanti e coefficienti dei termini minori, ad esempio $6n^2 + 100n + 300$ diventa semplicemente $\Theta(n^2)$.

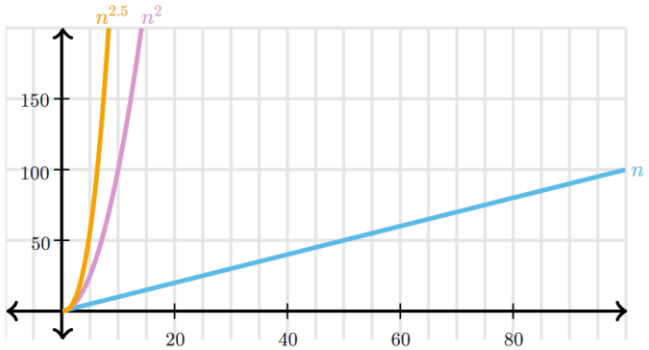
Quando usiamo la notazione Θ grande, diciamo che abbiamo uno stretto vincolo asintotico con il tempo di esecuzione, asintotico perché importa solo per grandi valori di n e stretto vincolo perché è stato confinato tra due fattori.

1.3.2 Funzioni nella notazione asintotica

È importante tenere a mente alcune nozioni importanti.

Supponiamo che l'algoritmo abbia un tempo di esecuzione costante, ad esempio la ricerca del minimo in un array già ordinato. Poiché in questa notazione preferiamo usare una funzione di n, potremmo dire che l'algoritmo impiega $\Theta(n^0)$ tempo, anche se in realtà scriveremmo Θ(1).

Ora, supponiamo che un algoritmo richieda $\Theta(\log_{10}n)$ tempo. Potremmo anche dire che richiede $\Theta(\log_2 n)$. quando la base di un algoritmo è una costante, non importa che base si utilizzi nella notazione asintotica. Perché no? Perché tramite la formula $\log_a n = \frac{\log_b n}{\log_b a}$ per tutti i numeri positivi a, b ed n. quindi, possiamo dire che nel caso peggiore il tempo richiesto da una ricerca binaria sia $\Theta(\log_a n)$ per ogni costante positiva a.



Nonostante ciò spesso scriviamo che il tempo della ricerca binaria è $\Theta(\log_2 n)$ perchè nella scienza dell'informatica si è abituati a pensare per potenze di 2.

C'è da tenere a mente che nella notazione, per ogni a < b, si ha che $\Theta(n^a)$ cresce più lentamente rispetto a $\Theta(n^b)$. a e b non devono essere per forza interi, ad esempio il grafico seguente compara la crescita di n, n^2 e $n^{2.5}$.

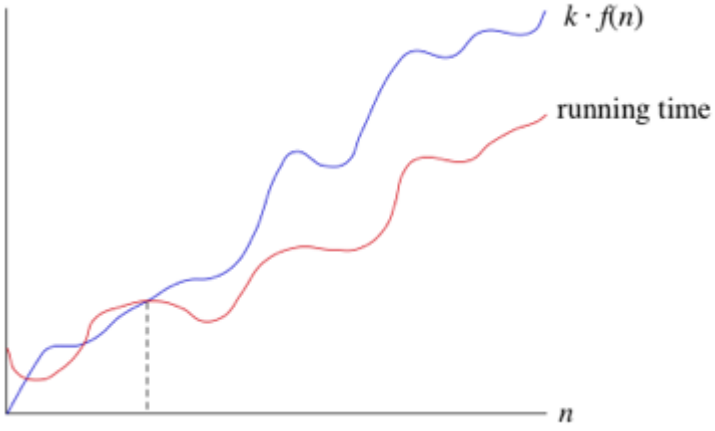
I logaritmi crescono più lentamente rispetto ai polinomi. Ecco una lista di funzioni asintotiche, ordinate dalla più lenta alla più veloce:

1. Θ(1)
2. $\Theta(\log_2 n)$
3. Θ(n)
4. $\Theta(n\log_2 n)$
5. $\Theta(n^2)$
6. $\Theta(n^2\log_2 n)$
7. $\Theta(n^3)$
8. $\Theta(a^n)$ con a > 1
9. Θ(n!)

Questa lista non è esaustiva. Attenzione: per ogni a > b, $\log_a n$ cresce più lentamente rispetto a $\log_b n$.

1.3.3 Notazione O grande

Usiamo la notazione Θ grande per legare asintoticamente la crescita del tempo di esecuzione a due costanti, una sopra e una sotto. In certi casi potremmo voler trovare solo il limite superiore.



Ad esempio, nonostante il caso peggiore di una ricerca binaria sia Θ($\log_2 n$), sarebbe scorretto dire che questo sia il suo tempo di esecuzione in tutti i casi. Il tempo di esecuzione della ricerca binaria non è mai peggiore di Θ($\log_2 n$). La notazione serve per dire ciò.

Se un tempo di esecuzione è $O(f(n))$, allora per n abbastanza grande, il tempo di esecuzione è massimo $k * f(n)$ per una costante k.

Diciamo che il tempo di esecuzione è O grande di $f(n)$, o semplicemente O di $f(n)$. Usiamo la notazione O grande per definire il vincolo asintotico superiore.

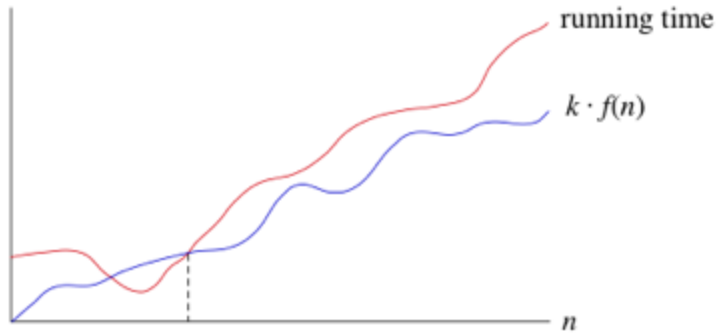
Ora dobbiamo trovare un modo per caratterizzare il tempo di esecuzione di una ricerca binaria per tutti i casi. Possiamo dire che il tempo di esecuzione della ricerca binaria è sempre $O(log2n)$. possiamo rendere ancora più forte l'affermazione dicendo che il caso peggiore è $\Theta(log2n)$ ma per un'affermazione che copra tutti i casi è meglio dire che la ricerca binaria ha tempo di esecuzione sempre $O(log2n)$.

Quindi O grande è il vincolo asintotico superiore di Θ grande. Quindi possiamo dire che se il caso peggiore è $\Theta(log2n)$, allora è anche $O(log2n)$, ma non sempre il contrario è vero. Possiamo dire che la ricerca binaria ha sempre tempo $O(log2n)$ ma che non sempre ha tempo Θ($\log_2 n$).

Poiché la notazione O grande conferisce solo il limite superiore e non entrambi i vincoli, possiamo fare dichiarazioni che possono sembrare scorrette ma che sono tecnicamente corrette. Ad eempio, possiamo dire che la ricerca binaria ha tempo O(n), questo perché il tempo di esecuzione non cresce più velocemente rispetto ad una costante n, è come dire, avendo 10€ in tasca, "non ho più di un miliardo di € in tasca".

1.3.4 Notazione Ω grande

Ci sono casi nei quali vogliamo dire che un algoritmo richiede almeno un certo ammontare di tempo, senza dare un limite superiore. In questo caso usiamo la notazione Ω grande.



Se il tempo di esecuzione è $\Omega(f(n))$, allora per un n sufficientemente grande, il tempo di esecuzione è almeno $k * f(n)$ per una costante k. Ecco un esempio di come immaginare $\Omega(f(n))$.

Usiamo la notazione $\Omega(f(n))$ per descrivere il limite asintotico inferiore.

Così come $\Theta(f(n))$ implica $O(f(n))$, allo stesso modo implica $\Omega(f(n))$. Quindi possiamo dire che il caso migliore della ricerca binaria è $\Omega(log2n)$.

Così come per O grande, possiamo fare affermazioni tecnicamente corrette per Ω grande, ad esempio avendo un milione di € in tasca dire “ho almeno 10 € in tasca”.

1.3.5 Promemoria dagli esercizi

La funzione all’interno della notazione asintotica è quella che stiamo considerando come limite.

Dobbiamo ricordare che i logaritmi possono cambiare base senza problemi.

1.4 ORDINAMENTO

Ordinare una lista di oggetti in ordine crescente o decrescente può aiutare a trovare gli oggetti richiesti più velocemente.

1.4.1 Selection Sort

Pseudocodice:

1. Trova il valore minimo, sostituiscilo al primo posto
2. Trova il secondo valore più piccolo, sostituiscilo al secondo posto
3. Trova il terzo valore più piccolo, sostituiscilo al terzo posto
4. Ripetere fino all’ultimo numero

Si chiama selection sort perché continua a selezionale e sostituire gli elementi.

1.4.1.1 Trovare l’indice del numero più piccolo in un subarray

Dopo aver sostituito il primo valore di un array, diventa inutile iniziare la ricerca dal valore di indice 0, quindi si può farla iniziare dall’indice 1 e così via, in modo da ridurre il carico necessario di volta in volta.

1.4.1.2 Analisi del selection sort

```
1   public class SelectionSort{
2
3       public static void main(int [] array){
4
5           int indiceMinimo = 0;
6
7           for(int i = 0; i < array.length-1; i++) {
8               for (int j = i; j<array.length-1; j++){
9                   indiceMinimo = indexOfMinimum(i, array);
10                  swap(i, indiceMinimo, array);
11              }
12          }
13      }
14
15      public int indexOfMinimum(int i, int[] array){
16
17          int minimo = array[i];
18
19          for(i; i<array.length; i++)
20              if(array[i]<minimo)
21                  minimo = i;
22
23          return i;
24      }
25
26      public void swap(int posizione, int nuovoIndice, int[] array){
27
28          int temp = array[posizione];
29          array[posizione] = array[nuovoIndice];
30          array[nuovoIndice] = temp;
31      }
32  }
```

Il selection sort cicla tenendo in considerazione gli indici di un array. Per ogni indice, il selection sort invoca `indexOfMinimum` e `swap`. `Array.length() = n`.

L’esecuzione del primo for potrebbe indurre a pensare che il codice sia composto da solo due linee di codice ma non è corretto. Per ogni invocazione di `indexOfMinimum` e swap devono essere valutate anche le loro linee di codice.

Quante linee di codice corrispondono a `swap`? In questo caso (e nel caso più comune) tre.

Quante linee di codice comprende `indexOfMinimum`? Dipende dalla posizione di inizio del subarray perché bisogna ricordare che ad ogni passaggio di posizione il ciclo for deve ciclare una volta di meno.

Quindi quante volte ciclerà? Ecco una spiegazione rapida. Immaginiamo un array di 8 elementi.

Invece di sommare $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36$ possiamo sommarlo a coppie, ovvero $(1 + 8) + (2 + 7) + (3 + 6) + (4 + 5) = 4 * 9 = 36$. Questo è il caso nel quale il numero di elementi è pari, se fosse dispari? Prendiamo $1 + 2 + 3 + 4 + 5 = 15$. Potremmo considerare il valore centrale come una "somma a metà", quindi faremmo $(1 + 5) + (2 + 4) + (3 + 0) = 6 * 2{,}5 = 15$.

Ora abbiamo ragionato con n fissato, ma con un n generico? Si chiama somma aritmetica e la somma totale è $(1 + n) * (\frac{n}{2}) = (\frac{n^2}{2}) + \frac{n}{2}$. Ad esempio con 8 è $(1 + 8) * \frac{8}{2} = 9 * 4 = 36$.

1.4.1.3 Analisi asintotica per il selection sort

Il totale del tempo è composto da tre parti:

1. Il tempo per `indexOfMinimum`
2. Il tempo per `swap`
3. Il tempo per il ciclo del selection sort

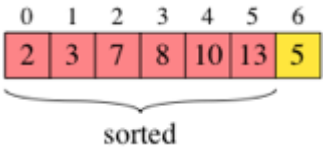
La seconda e la terza parte sono semplici. Sappiamo che ci saranno n chiamate per `swap` e che tutte queste chiamate chiedono tempo fisso. Quindi il tempo per tutte le chiamate di `swap` è θ(n). il ciclo del selection sort, similmente, esegue sempre le solite chiamate per i tutte le iterazioni, ottenendo così un altro θ(n).

Per la parte di `indexOfMinimum` ogni parte richiede un tempo costante. Il numero di iterazioni è $\sum_{i=1}^{n} i = (n + 1) * \left(\frac{n}{2}\right) = \frac{n^2}{2} + \frac{n}{2}$. In termini di θ non ci interessa altro se non n^2, quindi possiamo dire che `indexOfMinimum` è $\theta(n^2)$. Tra le tre parti, $\theta(n^2)$, θ(n), θ(n), possiamo dire che la più significativa è $\theta(n^2)$, quindi il tempo di esecuzione del selection sort è $\theta(n^2)$.

Notare che non c'è nemmeno un caso buono o cattivo per il selection sort, il tempo di `indexOfMinimum` è sempre $\theta(n^2)$, quindi possiamo dire che il selection sort è $\theta(n^2)$ in tutti i casi.

Θ	**θ**	**O**
$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$

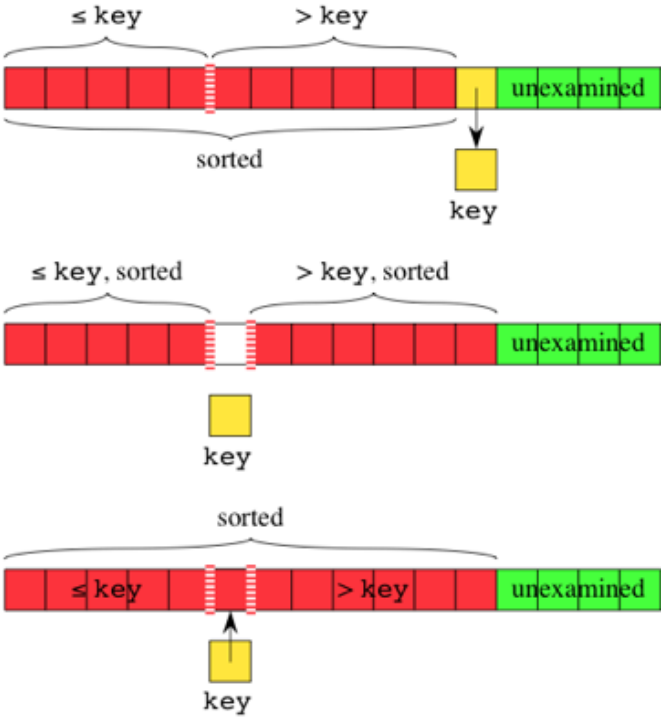
1.4.2 Insertion sort



Ecco un altro modo di pensare ad un algoritmo di ordinamento.

Immaginiamo di avere un array ordinato e che ci venga chiesto di inserire un altro numero. Inizieremmo a cercare dalla prima posizione fino a quando non incontriamo un numero maggiore di quello considerato.

chiamiamo l'elemento di indice 6 "chiave". L'operazione richiesta è quella di comparare la chiave con il valore precedente, nel caso esso sia maggiore, allora scambieremmo i due.



Questo strumento, però, richiede una precauzione: se il numero fosse più piccolo di tutti gli altri, a questo punto incorreremmo in un'eccezione nel caso provassimo a confrontare la posizione più a sinistra con la sua precedente.

Nel caso invece sia il più grande, al primo controllo non dovrebbe seguire alcuna azione.

1.4.2.1 Pseudocodice

Ecco lo pseudocodice del insertion sort:

1. Chiama `insert` per inserire l'elemento che inizia con l'indice 1 in un subarray ordinato di indice 0
2. Chiama `insert` per inserire l'elemento che inizia con l'indice 2 in un subarray ordinato con indice da 0 a 1
3. Chiama `insert` per inserire l'elemento che inizia con l'indice 3 in un subarray ordinato con indice da 0 a 2
4. …
5. Chiama `insert` per inserire l'elemento che inizia con l'indice n-1 in un subarray ordinato con indice da 0 a n-2

1.4.2.2 Analisi dell'insertion sort

```
1  import java.util.Scanner;
2
3  public class InsertionSort{
4      public static void main(int[] array){
5
6          System.out.print("Inserire grandezza array: ");
7          array = new int[tastiera.nextInt()];
8          Scanner tastiera = new Scanner(System.in);
9
10         for(int i=0; i<array.length; i++){
11             array[i] = tastiera.nextInt();
12             insert(array, array[i], i);
13         }
14     }
15
16     public void insert(int[] array, int nuovoValore, int indice){
17
18         int temp;
19
20         for(int i=indice-1; i>0; i--){
21             if(nuovoValore < array[i]){
22                 temp = array[i];
23                 array[i]=nuovoValore;
24                 array[i+1] = temp;
25             }
26             else
27                 break;
28         }
29     }
30 }
```

Così come `indexOfMinimum` richiedeva un ammontare di tempo dipendente dalla grandezza dell'array, anche `insert` segue lo stesso principio.

Prendiamo il caso nel quale l'elemento inserito sia il più piccolo di tutto l'array. In questo caso `insert` deve controllare l'intero array. Invece di dire quante linee di codice richieda l'implementazione di questo controllo, diciamo che è una costante c poiché è sempre uguale. Quindi per inserire un nuovo valore in un subarray ordinato di k elementi verrà richiesto $c * k$ tempo.

Immaginiamo ora che l'elemento inserito sia sempre il più piccolo, a questo punto avremmo $c * \sum_{k=1}^{n-1} k$ chiamate, ovvero $c * (n - 1 + 1)\left(\frac{n-1}{2}\right) = \frac{cn^2}{2} - \frac{cn}{2}$. Quindi avremmo una crescita con θ(n²).

Può l'insertion sort richiedere meno tempo? Sì, nel caso il numero inserito sia sempre più grande di tutti gli altri. In questo caso `insert` richiederebbe sempre tempo costante ma ci sarebbero comunque $n - 1$ chiamate a `insert`, quindi il tempo totale sarebbe $c(n - 1)$, che è θ(n).

Quindi nel caso peggiore (un array ordinato in maniera decrescente e con numeri da inserire sempre minori del resto) abbiamo θ(n²) mentre nel caso migliore (un array ordinato e con numeri da inserire sempre maggiori del resto) avremmo θ(n).

Nel caso intermedio invece, ovvero quando alcuni valori sono ordinati e altri no? Immaginiamo che un valore possa essere spostato massimo di 17 posizioni, in questo caso `insert` dovrebbe spostare massimo 17 valori, e quindi `insert` richiederebbe massimo $17 * c$. in tutti i n-1 casi di chiamate a `insert`, il tempo totale sarebbe $17 * c * (n - 1)$, ovvero θ(n).

Quindi si può dire che il caso medio segua θ(n²) poiché c'è sempre la possibilità del peggior scenario.

Θ	**θ**	**O**
Ω(n)	θ(n²)	O(n²)

1.4.3 Ricorsività

```
1   import java.util.Scanner;
2
3   public class Fattoriale{
4       public static void main (){
5
6           Scanner tastiera = new Scanner(System.in);
7           int n = tastiera.nextInt();
8           int private prodotto = 1;
9
10          if(n>0)
11              fattoriale(n);
12      }
13
14      public void fattoriale(int n){
15
16          if(n>0)
17              prodotto = fattoriale(n-1);
18      }
19  }
```

Prendiamo ad esempio la funzione fattoriale $n!$. Ricordiamo prima di tutto che $0! = 1$, ovvero il valore neutro della moltiplicazione. Immaginiamo ora di avere n vestiti ma di poterne ritirare solo k. In quanti modi possiamo ordinare questi vestiti?

$$\frac{n!}{k!*(n-k!)}$$

In questo caso la ricorsività è utile per calcolare i fattoriali poiché si tratta di calcolare la produttoria dei valori richiesti. Infatti partiamo da n, che verrà moltiplicato a $n-1$, che verrà moltiplicato a sua volta a $n-2$ e così via, fino a $0! = 1$.

Nelle funzioni ricorsive chiamiamo il caso di cui conosciamo già il risultato "caso base".

1.4.4 Divide et impera

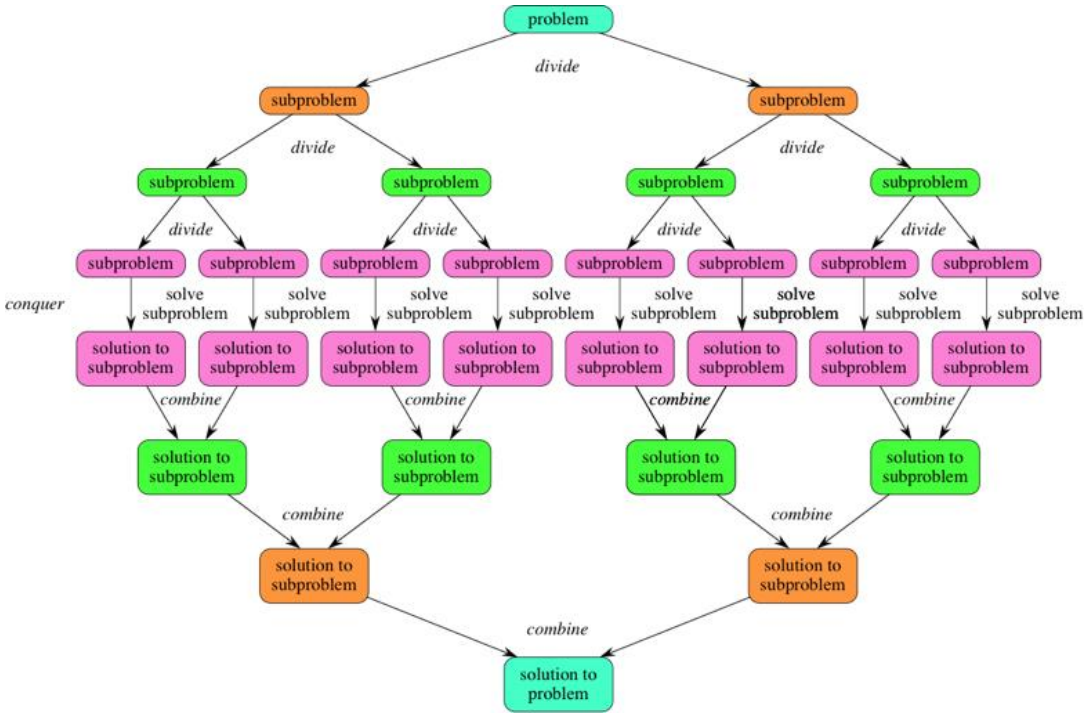
Sia il merge sort che il quicksort usano un paradigma comune basato sulla ricorsione. Il paradigma divide et impera divide il problema in sottoproblemi che sono simili al problema originale e li risolve ricorsivamente per terminare unendo i risultati.

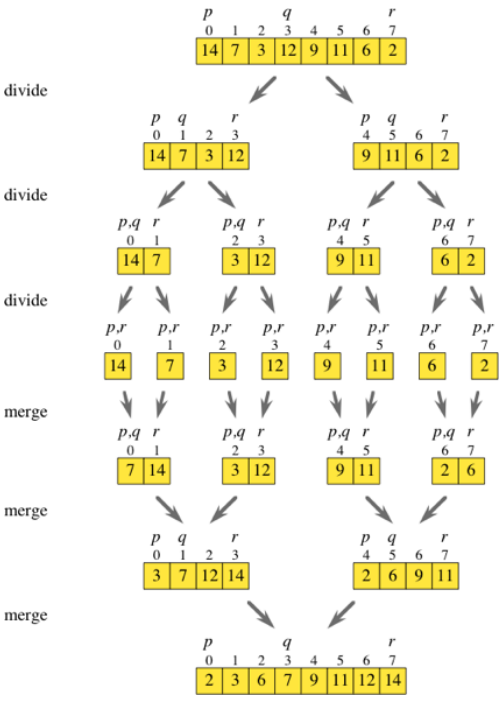
Poiché divide et impera risolve i problemi ricorsivamente, vuol dire che ogni problema sarà più piccolo del precedente fino al raggiungimento di un caso base.

L'idea del divide et impera è:

1. Divide il problema in sottoproblemi che sono istanze più piccole dello stesso problema
2. Impera sui sottoproblemi risolvendoli ricorsivamente. Se sono abbastanza piccoli li risolve come casi base

3. Combina le varie soluzioni






```java
public class EsempioMergeSort {

    public static void main(String[] args) {
        int[] unArray = {7, 5, 11, 2, 16, 4, 18, 14, 12, 30};
        System.out.println("Valori dell'array prima dell'ordinamento");
        for (int i = 0; i < unArray.length; i++)
            System.out.print(unArray [i] + " ");
        System.out.println();
        ordina(unArray);
        System.out.println("Valori dell'array dopo l'ordinamento");
        for (int i = 0; i < unArray.length; i++)
            System.out.print(unArray [i] + " ");
        System.out.println();
    }

    /**
    Precondizione: ogni posizione dell'array a contiene un valore.
    Postcondizione: a[0] <= a[1] <= ... <= a[a.length - 1].
     */
    public static void ordina(int[] a) {
        if (a.length >= 2) {
            int metaLunghezza = a.length / 2;
            int[] primaMeta = new int[metaLunghezza];
            int[] ultimaMeta = new int[a.length - metaLunghezza];
            dividi(a, primaMeta, ultimaMeta);
            ordina(primaMeta);
            ordina(ultimaMeta);
            merge(a, primaMeta, ultimaMeta);
        }
        //altrimenti non fare niente. a.length == 1, quindi a è
        //già ordinato.
    }

    /**
    Precondizione: a.length = primaMeta.length + ultimaMeta.length.
    Postcondizione: Gli elementi di a sono divisi
    tra gli arrays primaMeta e ultimaMeta.
     */
    public static void dividi(int[] a, int[] primaMeta, int[] ultimaMeta) {
        for (int i = 0; i < primaMeta.length; i++)
            primaMeta[i] = a[i];
        for (int i = 0; i < ultimaMeta.length; i++)
            ultimaMeta[i] = a[primaMeta.length + i];
    }
```

1.4.5 Merge sort

Poiché useremo il metodo divide et impera, dovremo decidere come rappresentare i sottoproblemi.

Immaginiamo che il problrma sia di ordinare un intero array. Allora il sottoproblema sarebbe di ordinare un subarray. In particolare, possiamo pensare di ordinare un subarray che parte da p e finisce in r.

Ecco come il merge sort lavora:

1. Divide l'array trovando la posizione centrale q facendo $\frac{p+r}{2}$ e arrotondando per difetto
2. Impera ordinando ricorsivamente il subarray in ognuno dei due subarray creati
3. Combina unendo (merge) i due subarray

Il caso base è un subarray che contiene meno di due elementi, ovvero quando p≥r, poiché un array di zero o un elemento è già ordinato, quindi eseguiremo il divide et impera solo quando p<r.

Immaginiamo di avere l'array $[14, 6, 3, 12, 9, 11, 6, 2]$, quindi il primo array è $array[0..7]$ $(p = 0, r = 7)$.

1. Nello step dividi troviamo $q = (0 + 7)/2 = 3.5 \approx 3$
2. Nello step conquista dobbiamo ordinare due subarray, $array[0..3]$ che contiene $[14, 7, 3, 12]$ e $array$ $[4..7]$ che contiene $[9, 11, 6, 2]$

Poiché $p < r$, dobbiamo nuovamente dividere, quindi gli array diventano $[14, 7]$, $[3, 12]$, $[9, 11]$, $[6, 2]$ per poi arrivare ai casi base [14], [7], [3], [12], [9], [11], [6], [2].

A questo punto inizieremo a riunire le parti risalendo a due a due, ordinandole contestualmente, fino ad ottenere l'array iniziale ordinato.

Molti dei passaggi nel merge sort sono semplici, è la parte del combina ad essere più complicata.

1.4.5.1 Unione in tempo lineare

Il merge sort funziona:

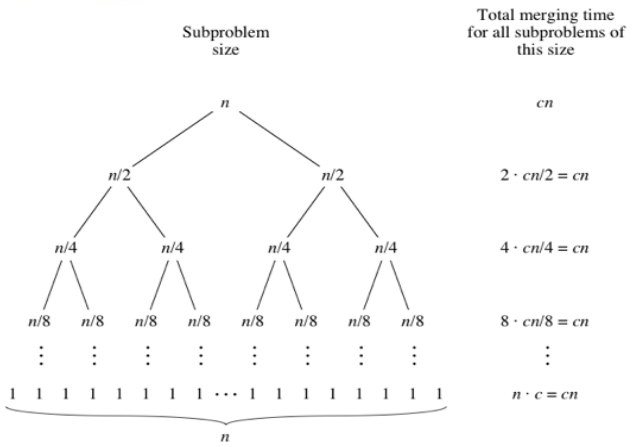
1. Copiando gli elementi dell'array in una metà inferiore e una superiore
2. Fino a quando ci sono elementi non esaminati vengono comparati i due valori nella stessa posizione relativa ai subarray e vengono copiati nell'array superiore in ordine

Poiché è una ricerca lineare e viene eseguito per un numero costante di volte, il tempo è θ(n).

1.4.5.2 Analisi del merge sort

```
47      /**
48          Precondizione: Gli array primaMeta e ultimaMeta sono ordinate in
49          senso crescente; a.length = primaMeta.length + ultimaMeta.length.
50          Postcondizione: L'array a contiene tutti gli elementi di primaMeta
51          e ultimaMeta ed è ordinato in senso crescente.
52       */
53      public static void merge(int[] a, int[] primaMeta, int[] ultimaMeta) {
54          int indicePrimaMeta = 0, indiceUltimaMeta = 0, indiceA = 0;
55          while ((indicePrimaMeta < primaMeta.length) &&
56                  (indiceUltimaMeta < ultimaMeta.length)) {
57              if (primaMeta [indicePrimaMeta] < ultimaMeta [indiceUltimaMeta]) {
58                  a[indiceA] = primaMeta[indicePrimaMeta];
59                  indicePrimaMeta++;
60              } else {
61                  a[indiceA] = ultimaMeta[indiceUltimaMeta];
62                  indiceUltimaMeta++;
63              }
64              indiceA ++;
65          }
66          /*
67          Almeno uno tra primaMeta e ultimaMeta è stato copiato
68          completamente in a.
69          Copiare il resto di primaMeta, se ne è rimasto.
70           */
71          while (indicePrimaMeta < primaMeta.length) {
72              a[indiceA] = primaMeta[indicePrimaMeta];
73              indiceA++;
74              indicePrimaMeta ++;
75          }
76          // Copiare il resto di ultimaMeta, se ne è rimasto.
77          while (indiceUltimaMeta < ultimaMeta.length) {
78              a[indiceA] = ultimaMeta[indiceUltimaMeta];
79              indiceA++;
80              indiceUltimaMeta++;
81          }
82      }
83  }
```

immaginiamo di avere n elementi in un array. Questi n elementi verranno divisi in due subarray di n/2 elementi, che verranno divisi a loro volta in quattro subarray di n/4 elementi ecc.



il tempo totale del merge sort è la somma dell'unione di tutti i livelli. Se ci sono l livelli in un albero, allora il tempo totale è $l * cn$. Quindi cos'è l? considerato che stiamo lavorando di metà in metà, possiamo dire che l sarà $l = \log_2 n + 1$. Quindi il totale di tempo per il merge sort è $\mathrm{cn}(\log_2 n + 1)$. Possiamo scartare il temine minore e i coefficienti, quindi possiamo dire: $\theta(n \log_2 n)$.

Θ	**θ**	**O**
$\Omega(\boldsymbol{n \log_2 n})$	$\theta(n \log_2 n)$	$O(n \log_2 n)$

1.4.6 Quicksort

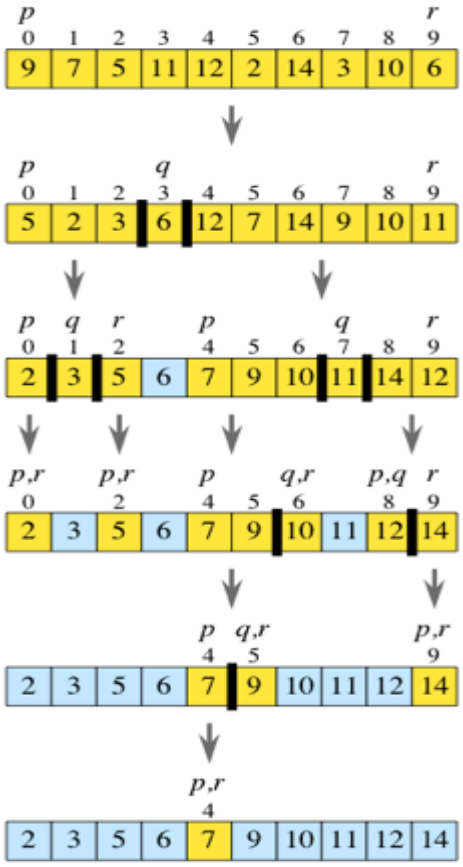
Così come il merge sort, anche il quicksort utilizza il sistema divide et impera, quindi è un algoritmo ricorsivo.

Nel merge sort il passo "divide" fa poco lavoro mentre tutto il lavoro viene svolto dal step combina. Nel quicksort è l'opposto.

Il quicksort ha anche un altro paio di differenze tra le quali che il caso peggiore è $O(n^2)$ come l'insertion sort ma il caso medio è $\theta(n \log_2 n)$ come nel merge sort. Quindi perché preferire il quicksort al merge sort? Perché il fattore costante nascosto nella notazione θ grande per il quicksort è abbastanza buono, nella pratica il quicksort è meglio anche del merge sort.

- Divide scegliendo un qualunque elemento nel subarray [p..r] (che chiameremo pivot). Sposta i vari elementi in modo che tutti gli elementi minori o uguali al pivot siano a sinistra e quelli maggiori a destra. Questa procedura si chiama partizionamento. A questo punto non è importante se sono ordinati gli elementi dei due sottogruppi. Per una questione di pratica sceglieremo sempre l'elemento più a destra come pivot.
- Conquista tramite un ordinamento ricorsivo del subarray [p..q-1] e del subarray [q+1..r] (dove q è il pivot). La ricorsione consiste nel prende un nuovo pivot e ripetere.

- Combina facendo niente poiché il passo conquista avrà già fatto tutto



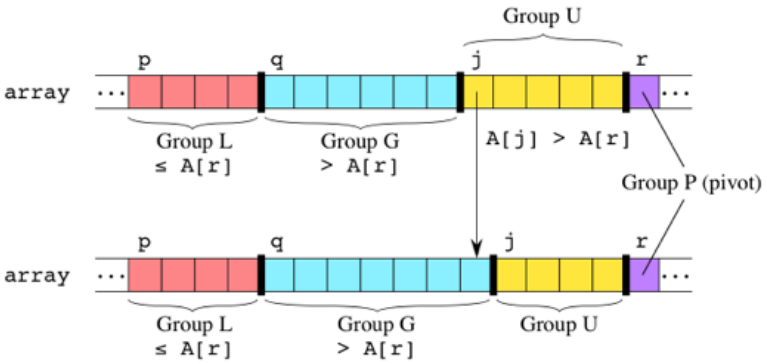
1.4.6.1 Partizionamento in tempo lineare

Il vero lavoro del quicksort avviene durante lo step di divisione attorno al pivot.

Dopo aver scelto un pivot, partizioniamo il subarray attraversandolo da sinistra a destra, comparando ogni elemento rispetto al pivot.

Manteniamo due indici q e j nel subarray che lo dividono in quattro gruppi:

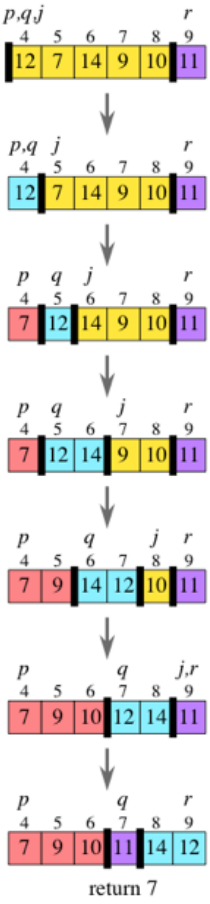
- Gruppo L [p..q-1], sono gli elementi minori del pivot
- Gruppo G[q..j-1], sono gli elementi più grandi del pivot
- Gruppo U[j..r-1], sono gli elementi non ancora valutati
- Gruppo P[r], il pivot



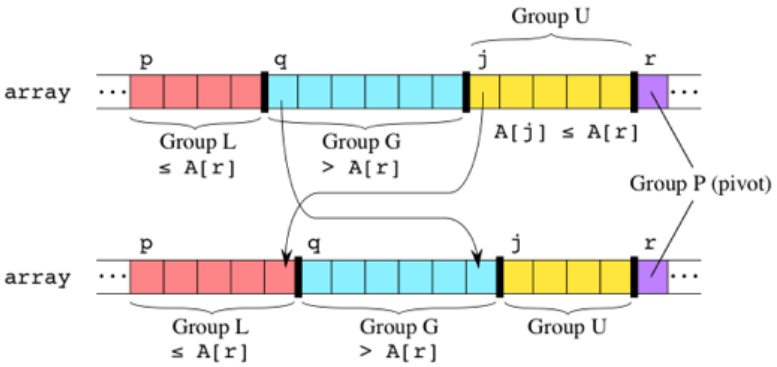
All'inizio, q, j e p sono uguali. Ad ogni step, compariamo A[j] (l'elemento più a sinistra di U) con il pivot. Se A[j] è maggiore del pivot, incrementare j in modo che la linea di divisione tra G e U si sposti a destra.

Se invece fosse minore del pivot, scambiamo di posto A[j] e A[q] (l'elemento più a sinistra del gruppo G), incrementiamo j e q, spostando le linee divisorie tra L e G e tra G e U a destra.

Una volta che raggiungiamo il pivot, il gruppo U sarà vuoto. Quindi scambiamo il pivot con l'elemento più a sinistra di G, scambiando A[r] con A[q].



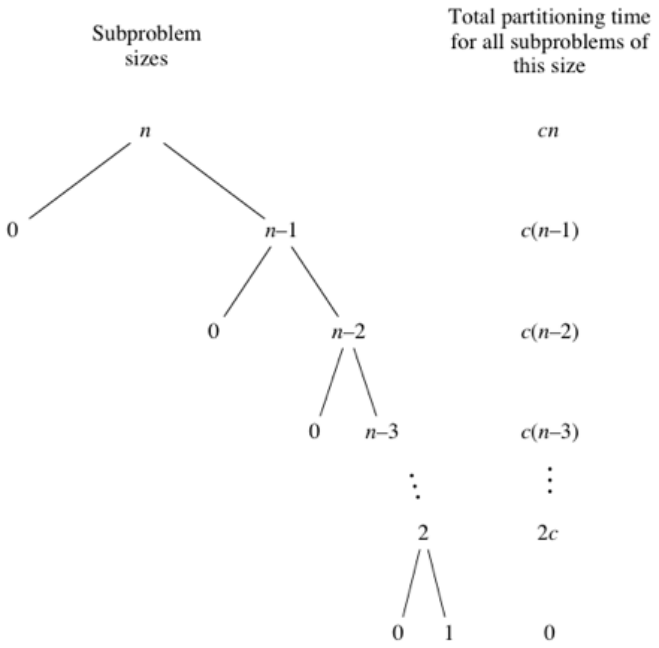
Partizionare un subarray con n elementi impiega θ(n) tempo, ovvero l'ordinamento è in tempo lineare.



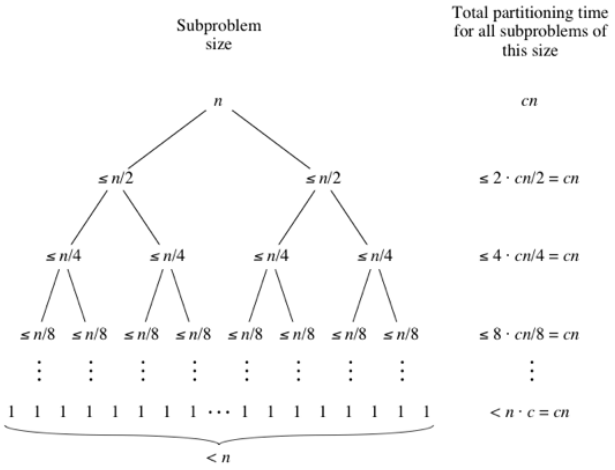
1.4.6.2 Analisi del quicksort

Com'è possibile che il caso peggiore e quello medio cambino? Iniziamo guardando il caso peggiore.

```
1 public class Quicksort{
2
3     public void quickSort(int arr[], int left, int right) {
4
5         int index = partition(arr, left, right);
6
7         if (left < index - 1)
8             quickSort(arr, left, index - 1);
9         if (index < right)
10            quickSort(arr, index, right);
11
12    }
13
14    public int partition(int arr[], int left, int right){
15
16        int i = left, j = right;
17        int tmp;
18        int pivot = arr[(left + right) / 2];
19
20        while (i <= j) {
21            while (arr[i] < pivot)
22                i++;
23            while (arr[j] > pivot)
24                j--;
25            if (i <= j) {
26                tmp = arr[i];
27                arr[i] = arr[j];
28                arr[j] = tmp;
29                i++;
30                j--;
31            }
32        };
33
34        return i;
35    }
36 }
```

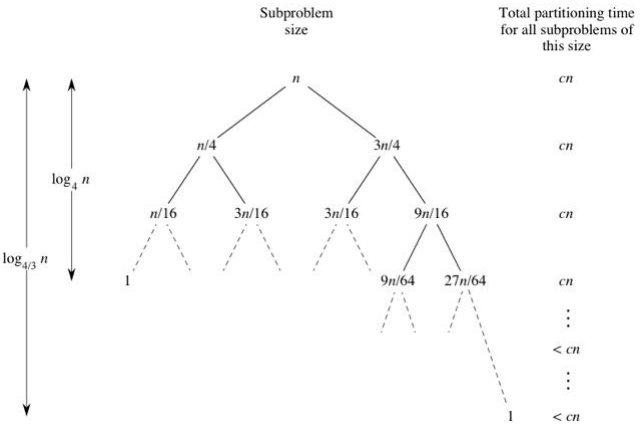


Quando il quicksort è fortemente sbilanciato, allora la chiamata iniziale richiede cn tempo per una qualche costante c, la seconda chiamata lavora su (n-1) quindi richiede c(n-1) ecc. essendo una serie aritmetica, questo caso peggiore lavora in $\theta(n^2)$.



Nel caso migliore le cose cambiano.

Il quicksort lavora nel caso migliore quando i rami sono più bilanciati possibile. Questo è il caso nel quale il numero di elementi sia dispari e il pivot sia l'elemento centrale. Ogni partizione avrebbe così $\frac{n-1}{2}$ elementi, quindi discendendo avremmo sempre meno elementi, assumendo così un tempo come quello del mergesort: $\theta(n \log_2 n)$.



Anche il caso medio lavora in $\theta(n \log_2 n)$ ma dimostrarlo è leggermente complicato.

Un modo per rendere meno comune il caso peggiore è l'utilizzo di una selezione casuale del pivot. L'ideale sarebbe selezionare n elementi casualmente, scegliere la mediana e utilizzare quella come pivot.

Θ	**θ**	**O**
$\Omega(\boldsymbol{n \log_2 n})$	$\theta(n \log_2 n)$	$O(n^2)$

“Introduzione agli algoritmi e strutture dati”

Thomas H. Cormen, Charles E. Leiserson,

Ronald L. Rivest, Clifford Stein - 3.a edizione. Mc-Graw Hill, 2010

2 IL RUOLO DEGLI ALGORITMI NELLA PROGRAMMAZIONE

2.1 ALGORITMI

Un algoritmo è una sequenza di passi che trasforma l'input dato in un output. È a che uno strumento per risolvere un problema ben definito.

Istanza di un problema: input necessario per computare la soluzione del problema.

Un algoritmo è **corretto** se, per ogni istanza del problema, termina con l'output corretto. Un algoritmo corretto risolve il dato problema computazionale. Un algoritmo scorretto potrebbe arrestarsi o restituire una soluzione errata.

There are some problems, however, for which no efficient solution is known which are known as NP-complete.

Why are NP-complete problems interesting?

1. although no efficient algorithm for an NP-complete problem has ever been found, **nobody has ever proven that an efficient algorithm for one cannot exist.** In other words, no one knows whether or not efficient algorithms exist for NP-complete problems.
2. the set of NP-complete problems has the remarkable property that i**f an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them**.
3. **several NP-complete problems are similar**, but not identical, **to problems for which we do know of efficient algorithms.**

2.1.1 Strutture dati

Una struttura dati è un modo per immagazzinare e organizzare i dati per facilitare l'accesso e le modifiche. Non esiste una struttura dati perfetta per tutte le necessità, quindi è importante conoscerne pro e contro in base all'utilizzo.

2.1.2 Parallelismo

We can liken these multicore computers to several sequential computers on a single chip; in other words, they are a type of "parallel computer." **In order to elicit the best performance from multicore computers, we need to design algorithms with parallelism in mind.**

2.2 ALGORITMI COME TECNOLOGIA

Immaginiamo che i computer siano infinitamente veloci e che la memoria sia gratuita, ci sarebbe comunque motivo di studiare gli algoritmi? Sì, perché comunque bisogna capire se un algoritmo è corretto.

2.2.1 Efficienza

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

For a concrete example, let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of 10 million numbers. Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more

dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $\boldsymbol{2n^2}$ instructions to sort n numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking $\boldsymbol{50n \log n}$ instructions. To sort 10 million numbers, computer A takes

$$\frac{2 * (10^7)^2 \; istruzioni}{10^{10} istruzioni \; al \; secondo} = 20{,}000 \; secondi \; (circa \; 5.5 \; ore)$$

while computer B takes

$$\frac{50 * 10^7 \log 10^7 \; istruzioni}{10^7 istruzioni \; al \; secondo} = 1163 \; secondi \; (circa \; 20 \; minuti)$$

3 INIZIAMO

3.1 INSERTION SORT

Our first algorithm, insertion sort, solves the ***sorting problem*** introduced in Chapter 1.

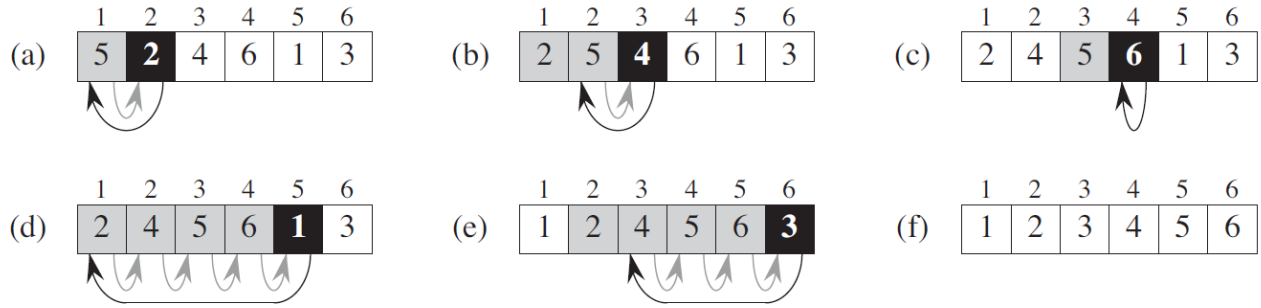
Input: A sequence of n numbers

Output: A permutation (reordering) della sequenza di input in modo che ogni numero sia maggiore uguale al precedente

The numbers that we wish to sort are also known as the **keys**.

We start with insertion sort, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand.

The algorithm sorts the input numbers in place: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time.




```
1 for j = 2 to A.length
2       key = A[j]
3       // inserisce A[j] nella sequenza ordinata A[1 … j-1]
4       i = j-1
5       while i > 0 and A[i] > key
6               A[i+1] = A[i]
7               i = i -1
8       A[i+1] = key
```

j = indice numero corrente

i = j-1 = indice numero precedente

The figure shows how this algorithm works for = <5, 2, 4, 6, 1, 3> The index j indicates the “current card” being inserted into the hand. At the beginning of each iteration of the for loop, which is indexed by j , the subarray consisting of elements A[1..j-1] constitutes the currently sorted hand, and the remaining subarray A[j+1..n] corresponds to the pile of cards still on the table.

3.1.1 Invarianti di ciclo e correttezza dell'insertion sort

3.1.2 Convezioni dello pseudocodice

- identation indicates block structure
- The looping constructs while, for, and repeat-until and the if-else conditional construct have interpretations similar to those in C, C++, Java, Python, and Pascal
- The symbol "//" indicates that the remainder of the line is a comment.
- A multiple assignment of the form i = j = e assigns to both variables i and j the value of expression e;
- Variables (such as i , j, and key) are local to the given procedure
- We access array elements by specifying the array name followed by the index in square brackets.
- We typically organize compound data into objects, which are composed of attributes.
- We pass parameters to a procedure by value
- A return statement immediately transfers control back to the point of call in the calling procedure.
- The boolean operators "and" and "or" are short circuiting.
- The keyword error indicates that an error occurred because conditions were wrong for the procedure to have been called

3.2 ANALIZZARE GLI ALGORITMI

Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.

Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic one processor, **random-access machine (RAM)** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

The data types in the RAM model are integer and floating point (for storing real numbers). Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial**. We also assume a limit on the size of each word of data**. For example, when working with inputs of size n, we typically assume that integers are represented by $c\ lg\ n$ bits for some constant c ≥1.

We require c ≥1 so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily.

In the RAM model, we do not attempt to model the memory hierarchy that is common in contemporary computers. That is, we do not model caches or virtual memory.

Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

3.2.1 Analisi dell'insertion sort

The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, **INSERTION-SORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are**. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input.

The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.

When a for or while loop exits in the usual way (i.e., due to the test in the loop header), **the test is executed one time more than the loop body.** We assume that comments are not executable statements, and so they take no time.

		*costo**	*numero di volte***
1	**for** j = 2 **to** A.length	*c1*	*n*
2	key = A[j]	*c2*	*n-1*
3	// inserisce A[j] nella sequenza ordinata A[1 … j-1]	*0*	*n-1*
4	i = j-1	*c4*	*n-1*
5	**while** i > 0 and A[i] > key	*c5*	$\sum_{j=2}^{n} t_j$
6	A[i+1] = A[i]	*c6*	$\sum_{j=2}^{n}(t_j - 1)$
7	i = i -1	*c7*	$\sum_{j=2}^{n}(t_j - 1)$
8	A[i+1] = key	*c8*	*n-1*

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes ci steps to execute and executes n times will contribute $c_i n$ to the total running time. To compute T(n) the running time of INSERTION-SORT on an input of n values, we sum the products of the cost and times columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Caso migliore: quando l'array è già ordinato. Quindi non viene mai eseguito il while, permettendoci di ridurre il costo ad una semplice funzione lineare in n.

Caso peggiore: quando l'array è ordinato al contrario. Il while viene eseguito ogni volta, portandoci ad una funzione quadratica in n^2.

Θ	**θ**	**O**
$\Omega(\boldsymbol{n})$	$\theta(n^2)$	$O(n^2)$

3.2.2 Analisi del caso peggiore e medio

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input. For some algorithms, the worst case occurs fairly often.
- The "average case" is often roughly as bad as the worst case.

In some particular cases, we shall be interested in the average-case running time of an algorithm; we shall see the technique of probabilistic analysis applied to various algorithms throughout this book.

3.2.3 Ordine di crescita

We shall now make one more simplifying abstraction: it is the rate of growth, or order of growth, of the running time that really interests us. Per farlo vengono eliminate le costanti, I coefficienti e vengono considerati solo i valori di n che crescono più velocemente.

Ad esempio $an^2 + bn + c$ diventa semplicemente θ(n^2).

3.3 DESIGN DI ALGORITMI

We can choose from a wide range of algorithm design techniques. **For insertion sort, we used an incremental approach**: having sorted the subarray A[1..j-1] we inserted the single element A[j] into its proper place, yielding the sorted subarray a[1..j].

In this section, we examine an alternative design approach, known as "**divide-and- conquer**," which we shall explore in more detail in Chapter 4. We'll use divide-and- conquer to design a sorting algorithm whose worst-case running time is much less than that of insertion sort.

3.3.1 L'approccio divide-et-impera

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach:

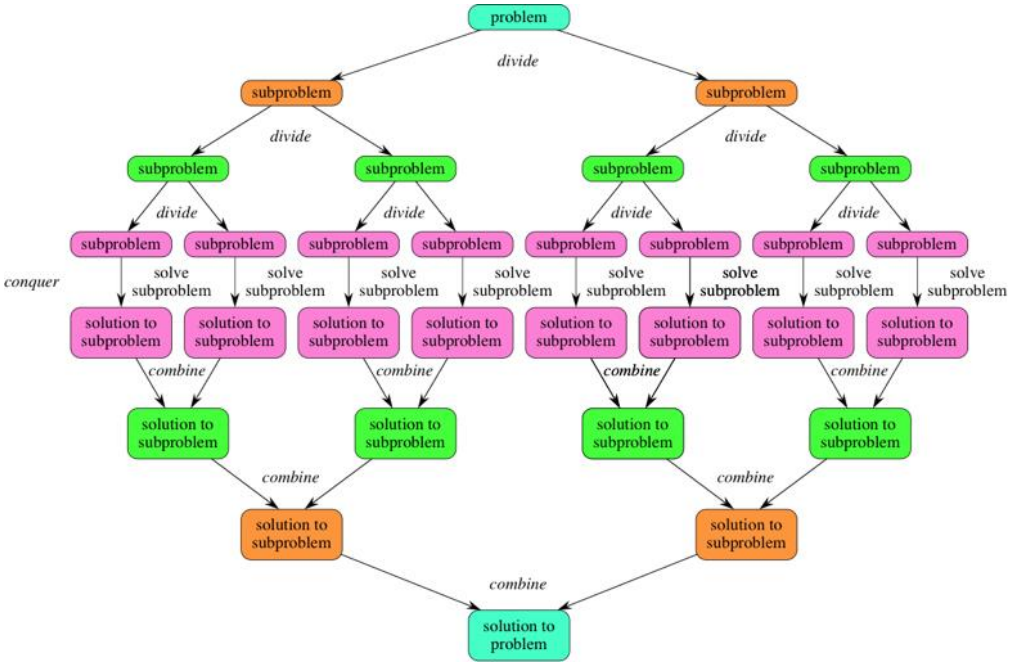
1. **Divide**: divide il problema in sottoproblemi più piccoli
2. **Conquista**: risolve i problemi ricorsivamente. Se il sottoproblema è abbastanza piccolo lo risolve direttamente
3. **Combina**: unisce tutte le soluzioni dei sottoproblemi per generare la soluzione completa

The **merge sort algorithm** closely follows the divide-and-conquer paradigm.

1. **Divide**: divide la sequenza di n elementi in due sotto sequenze di n/2 elementi
2. **Conquista**: ordina ricorsivamente le sotto sequenze
3. **Combina**: unisce tutte le soluzioni dei sottoproblemi per generare la soluzione completa

The recursion "bottoms out" when the sequence to be sorted has length 1.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step.



We merge by calling an auxiliary procedure MERGE(A, p, q, r) where A is an array and p, q, and r are indices into the array such that p ≤q < r. The procedure assumes that the subarrays a[p..q] and a[q+1..r] are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray a[p..r].

Our MERGE procedure takes time θ(n), where $n = r - p + 1$ is the total number of elements being merged.

```
MERGE(A, p, q, r)
1     n1 = q-p+1                          calcola la lunghezza del sottoarray
                                          A[p..q]
2     n2 = r-q                            calcola la lunghezza del sottoarray
                                          A[q+1..r]
3     crea due nuovi array L[1..n1+1] e   creiamo gli array R e L (lunghezza
R[1…n2+1]                                 n1+1 e n2+1)
4     for i = 1 to n1
5           L[i] = A[p+i-1]               copia il sottoarray A[p..q] in L
6     for j=1 to n2
7           R[j] =  A[q+j]                copia il sottoarray A[q+1..r] in R
8     L[n1+1] = ∞                         pone il valore sentinella
9     R[n2+1] = ∞                         pone il valore sentinella
10    i=1
11    j=1
12    for k = p to r                      combina
13          if L[i] ≤ R[j]
14                A[k] = L[i]
15                i = i+1
16          else A[k] = R[j]
17                j = j+1
```

```
1     n1 = q-p+1                                    tempo costante
2     n2 = r-q                                      tempo costante
3     crea due nuovi array L[1..n1+1] e R[1…n2+1]   tempo costante
4     for i = 1 to n1                               n1
5           L[i] = A[p+i-1]
6     for j=1 to n2                                 n2
7           R[j] =  A[q+j]
8     L[n1+1] = ∞                                   tempo costante
9     R[n2+1] = ∞                                   tempo costante
10    i=1                                           tempo costante
11    j=1                                           tempo costante
12    for k = p to r                                n
13          if L[i] ≤ R[j]
14                A[k] = L[i]
15                i = i+1
16          else A[k] = R[j]
17                j = j+1
```

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a sentinel card, which contains a special value that we use to simplify our code. Here, we use infinite as the sentinel value, so that whenever a card with infinite is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed.

Loop invariant: At the start of each iteration of the for loop of lines 12–17, the subarray $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n1+1]$ and $R[1..n2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

- **Inizio**: Prior to the first iteration of the loop, we have $k = p$, so that the subarray $A[p..k-1]$ is empty. This empty subarray contains the $k - p = 0$ smallest elements of L and R, and since $i =$

$j = 1$, both $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

- **Mantenimento**: To see that each iteration maintains the loop invariant, let us first suppose $that\ L[i] \leq R[j]$ Then $L[i]$ is the smallest element not yet copied back into A. Because $A[p..k-1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p..k]$ will contain the $k - p + 1$ smallest elements. Incrementing k (in the for loop update) and i (in line 15) reestablishes the loop invariant for the next iteration. If instead $L[i] > R[j]$ then lines 16–17 perform the appropriate action to maintain the loop invariant
- **Terminazione**: At termination, k = r + 1. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1..n1 + 1]$ and $R[1..n2 + 1]$, in sorted order. The arrays L and R together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into A, and these two largest elements are the sentinels.

To see that the MERGE procedure runs in θ(n) time, where = 0 r – p + 1, observe that each of lines 1–3 and 8–11 takes constant time, the for loops of lines 4–7 take $\theta(n_1 + n_2) = \theta(n)$ time, and there are n iterations of the for loop of lines 12–17, each of which takes constant time.

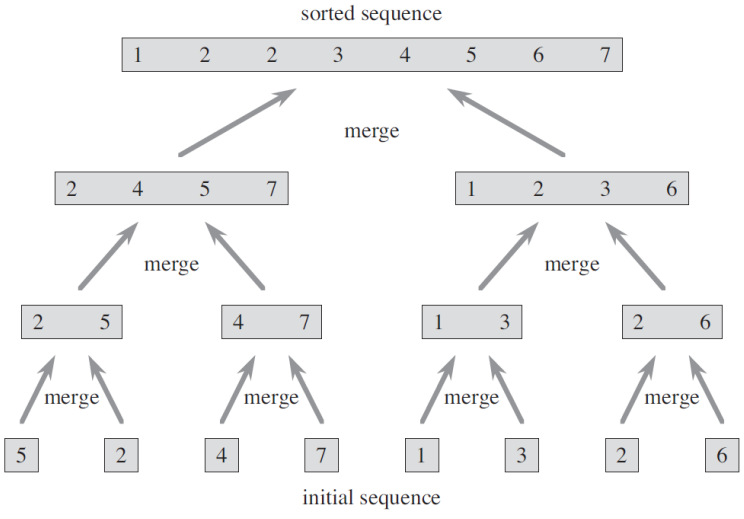
```
1    if p < r
2        q = ⌊(p+r)/2⌋
3        MERGE-SORT(A, p, q)
4        MERGE-SORT(A, q+1, r)
5        MERGE(A, p, q, r)
```

We can now use the MERGE procedure as a subroutine in the merge sort algorithm.

The procedure MERGE-SORT(A, p, r) sorts the elements in the subarray $A[p..r]$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two subarrays: $A[p..q]$, containing $[n/2]$ elements, and $A[q+1..r]$, containing $[n/2]$ elements.

3.3.2 Analisi degli algoritmi divide-et-impera

When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence.



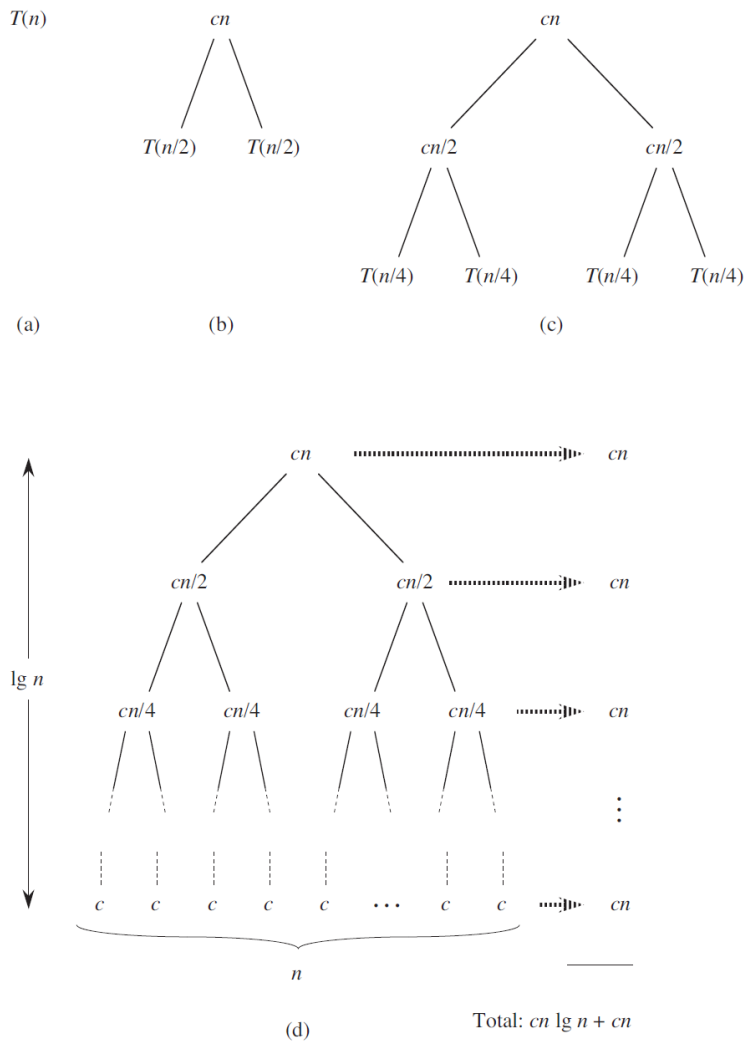
3.3.2.1 Analisi del merge-sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly n=2.

- **Divide**: The divide step just computes the middle of the subarray, which takes constant time. Thus D(n) = θ(1).
- **Impera**: We recursively solve two subproblems, each of size n=2, which contributes 2T(n/2) to the running time.
- **Combina**: We have already noted that the MERGE procedure on an n-element subarray takes time θ(n) and so C(n) = θ(n).

Quindi:

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ 2T(n/2) + \Theta(n) + \Theta(1) & se\ n > 1 \end{cases}$$



We add the costs across each level of the tree. The top level has total cost cn, the next level down has total cost $c\left(\frac{n}{2}\right) + c\left(\frac{n}{2}\right) = cn$, the level after that has total cost $c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) = cn$, and so on. In general, the level i below the top has $2i$ nodes, each contributing a cost of$c(\frac{n}{2^i})$, so that the i th level below the top has total cost $2^i c\left(\frac{n}{2^i}\right) = cn$. The bottom level has n nodes, each contributing a cost of c, for a total cost of cn. The total number of levels of the recursion tree in Figure 2.5 is $\log n + 1$, where n is the number of leaves, corresponding to the input size.

4 CRESCITA DELLE FUNZIONI

Once the input size n becomes large enough, merge sort, with its $\theta(n \log n)$ worst-case running time, beats insertion sort, whose worst-case running time is $\theta(n^2)$.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the asymptotic efficiency of algorithms.

4.1 NOTAZIONE ASINTOTICA

4.1.1 Notazione asintotica, funzioni e tempi di esecuzione

Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand which running time we mean. Sometimes we are interested in the worst-case running time. Often, however, we wish to characterize the running time no matter what the input. In other words, we often wish to make a blanket statement that covers all inputs, not just the worst case. We shall see asymptotic notations that are well suited to characterizing running times no matter what the input.

Eliminando i termini meno significativi e i coefficienti numerici, possiamo concentrarci sulla parte importante del tempo di esecuzione: il tasso di crescita. Quando eliminiamo questi orpelli, stiamo utilizzando l'annotazione asintotica. Ci sono tre forme:

- Θ (theta) grande
- O grande
- Ω (Omega) grande

4.1.2 Θ grande

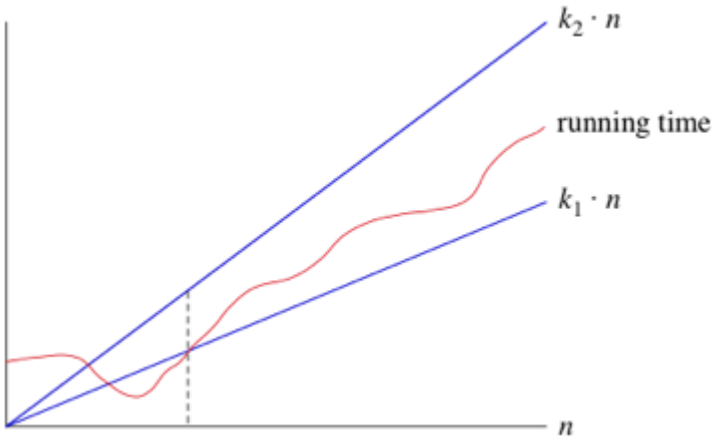
```
1  var doLinearSearch = function(array, targetValue) {
2    for (var guess = 0; guess < array.length; guess++) {
3      if (array[guess] === targetValue) {
4          return guess;  // found it!
5      }
6    }
7    return -1;  // didn't find it
8  };
```

Prendiamo `array.length` come n. il numero massimo di iterazioni è n e questo caso peggiore accade quando il numero non è nell'array.

Ogni iterazione, l'algoritmo deve:

- Comparare guess e array.length
- Comparare array[guess] con il valore obiettivo
- Dare guess come valore di ritorno eventualmente
- Incrementare guess

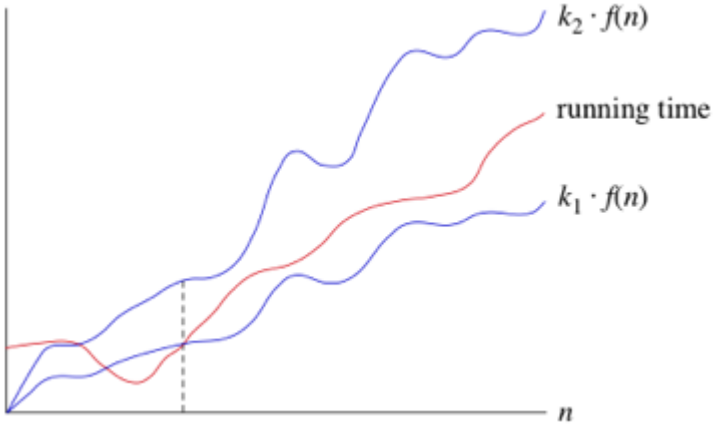
Ognuna di queste istruzioni richiede un tempo costante ogni volta che viene eseguita. Se il ciclo for itera n volta, il tempo richiesto per le n iterazioni è di $c_1 * n$ dove c1 è la somma dei tempi di computazione di un ciclo. C_1 dipende da varie cose: il linguaggio di programmazione, la macchina, il compilatore o l'interprete.



Questo codice ha anche una premessa e l'eventuale return -1, chiamiamo queste aggiunte c_2, quindi il tempo totale del caso peggiore è $c_1 * n + c_2$. Come abbiamo spiegato, il coefficiente c1 e il fattore costante c2 sono poco significativi per quanto riguarda il ritmo di crescita. La parte significativa è che nel caso peggiore la ricerca lineare cresce al ritmo di n. La notazione utilizzata è Θ(n).

Quando diciamo che un tempo di esecuzione è Θ(n), significa che quando n diventa abbastanza grande, il tempo di esecuzione è compreso tra $k_1 * n$ e $k_2 * n$ per alcune costanti k_1 e k_2.

Per piccoli valori di n, non ci interessa comparare il tempo di esecuzione con $k_1 * n$ e $k_2 * n$. Ma una volta che n diventa sufficientemente grande, il tempo di esecuzione diventa sempre compreso tra le due rette. Fino a quando k_1 e k_2 esistono, possiamo dire che il tempo di esecuzione è Θ(n).



Non siamo limitati a solo n nella notazione Θ grande. Possiamo anche usare funzioni come n^2, $nlog_2 n$, o altre funzioni con n. ecco ad esempio il tempo di esecuzione di Θ(f(n)). Una volta che n diventa sufficientemente grande, verrà sempre compreso tra $k_1 * f(n)$ e $k_2 * f(n)$.

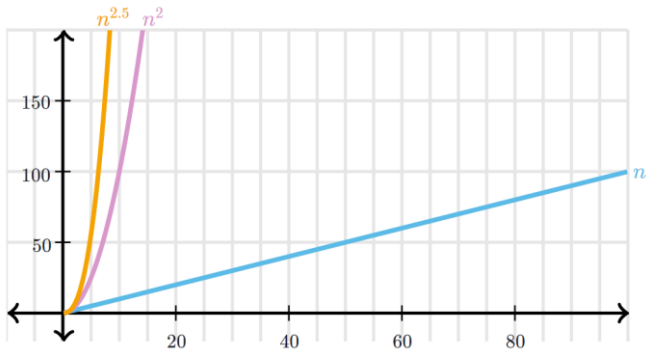
Nella pratica semplicemente togliamo costanti e coefficienti dei termini minori, ad esempio $6n^2 + 100n + 300$ diventa semplicemente $\Theta(n^2)$.

Quando usiamo la notazione Θ grande, diciamo che abbiamo uno stretto vincolo asintotico con il tempo di esecuzione, asintotico perché importa solo per grandi valori di n e stretto vincolo perché è stato confinato tra due fattori.

4.1.3 Funzioni nella notazione asintotica

È importante tenere a mente alcune nozioni importanti.

Supponiamo che l'algoritmo abbia un tempo di esecuzione costante, ad esempio la ricerca del minimo in un array già ordinato. Poiché in questa notazione preferiamo usare una funzione di n, potremmo dire che l'algoritmo impiega $\Theta(n^0)$ tempo, anche se in realtà scriveremmo Θ(1).



Ora, supponiamo che un algoritmo richieda $\Theta(\log_{10} n)$ tempo. Potremmo anche dire che richiede $\Theta(\log_2 n)$. quando la base di un algoritmo è una costante, non importa che base si utilizzi

nella notazione asintotica. Perché no? Perché tramite la formula $\log_a n = \frac{\log_b n}{\log_b a}$ per tutti i numeri positivi a, b ed n. quindi, possiamo dire che nel caso peggiore il tempo richiesto da una ricerca binaria sia Θ($\log_a$n) per ogni costante positiva a. Nonostante ciò spesso scriviamo che il tempo della ricerca binaria è Θ($\log_2$n) perchè nella scienza dell'informatica si è abituati a pensare per potenze di 2.

C'è da tenere a mente che nella notazione, per ogni a < b, si ha che Θ(na) cresce più lentamente rispetto a Θ(nb). a e b non devono essere per forza interi, ad esempio il grafico seguente compara la crescita di n, n2 e n2.5.

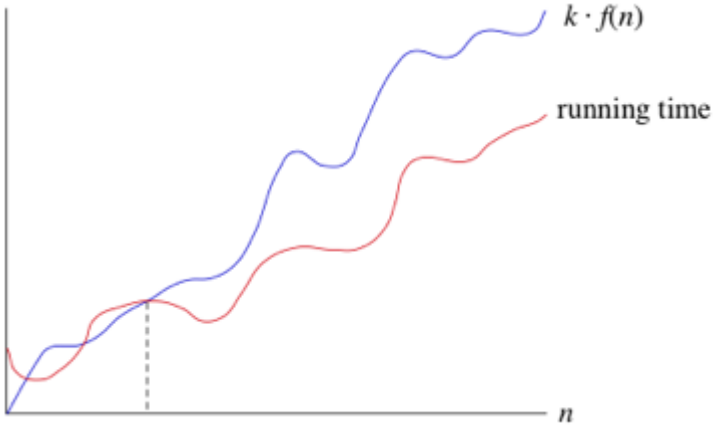
I logaritmi crescono più lentamente rispetto ai polinomi. Ecco una lista di funzioni asintotiche, ordinate dalla più lenta alla più veloce:

1	Θ(1)	6	Θ(n2log2n)
2	Θ(log2n)	7	Θ(n3)
3	Θ(n)	8	Θ(an) con a > 1
4	Θ(nlog2n)	9	Θ(n!)
5	Θ(n2)		

Questa lista non è esaustiva. Attenzione: per ogni a > b, logan cresce più lentamente rispetto a logbn.

4.1.4 Notazione O grande

Usiamo la notazione Θ grande per legare asintoticamente la crescita del tempo di esecuzione a due costanti, una sopra e una sotto. In certi casi potremmo voler trovare solo il limite superiore.



Ad esempio, nonostante il caso peggiore di una ricerca binaria sia Θ($\log_2$n), sarebbe scorretto dire che questo sia il suo tempo di esecuzione in tutti i casi. Il tempo di esecuzione della ricerca binaria non è mai peggiore di Θ($\log_2$n). La notazione serve per dire ciò.

Se un tempo di esecuzione è $O(f(n))$, allora per n abbastanza grande, il tempo di esecuzione è massimo $k * f(n)$ per una costante k.

Diciamo che il tempo di esecuzione è O grande di $f(n)$, o semplicemente O di $f(n)$. Usiamo la notazione O grande per definire il vincolo asintotico superiore.

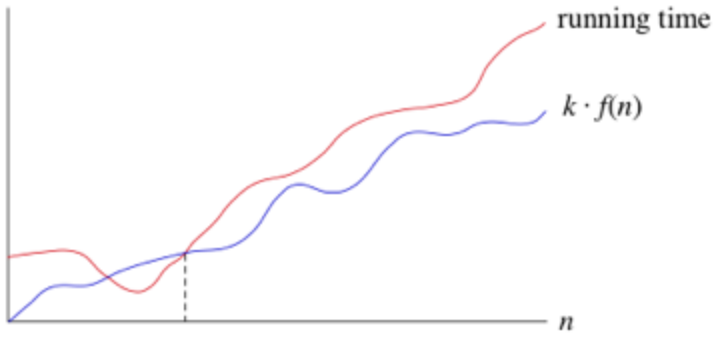
Ora dobbiamo trovare un modo per caratterizzare il tempo di esecuzione di una ricerca binaria per tutti i casi. Possiamo dire che il tempo di esecuzione della ricerca binaria è sempre $O(log2n)$. possiamo rendere ancora più forte l'affermazione dicendo che il caso peggiore è $\Theta(log2n)$ ma per un'affermazione che copra tutti i casi è meglio dire che la ricerca binaria ha tempo di esecuzione sempre $O(log2n)$.

Quindi O grande è il vincolo asintotico superiore di Θ grande. Quindi possiamo dire che se il caso peggiore è $\Theta(log2n)$, allora è anche $O(log2n)$, ma non sempre il contrario è vero. Possiamo dire che la ricerca binaria ha sempre tempo $O(log2n)$ ma che non sempre ha tempo Θ($\log_2$n).

Poiché la notazione O grande conferisce solo il limite superiore e non entrambi i vincoli, possiamo fare dichiarazioni che possono sembrare scorrette ma che sono tecnicamente corrette. Ad eempio, possiamo dire che la ricerca binaria ha tempo O(n), questo perché il tempo di esecuzione non cresce più velocemente rispetto ad una costante n, è come dire, avendo 10€ in tasca, "non ho più di un miliardo di € in tasca".

4.1.5 Notazione Ω grande

Ci sono casi nei quali vogliamo dire che un algoritmo richiede almeno un certo ammontare di tempo, senza dare un limite superiore. In questo caso usiamo la notazione Ω grande.



Se il tempo di esecuzione è $\Omega(f(n))$, allora per un n sufficientemente grande, il tempo di esecuzione è almeno $k * f(n)$ per una costante k. Ecco un esempio di come immaginare $\Omega(f(n))$.

Usiamo la notazione $\Omega(f(n))$ per descrivere il limite asintotico inferiore.

Così come $\Theta(f(n))$ implica $O(f(n))$, allo stesso modo implica $\Omega(f(n))$. Quindi possiamo dire che il caso migliore della ricerca binaria è $\Omega(log2n)$.

Così come per O grande, possiamo fare affermazioni tecnicamente corrette per Ω grande, ad esempio avendo un milione di € in tasca dire "ho almeno 10 € in tasca".

4.1.6 Confronto di funzioni

Proprietà transitiva	$f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$ $f(n) = O(g(n))$ e $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$ $f(n) = o(g(n))$ e $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$ $f(n) = \Omega(g(n))$ e $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$ $f(n) = \omega(g(n))$ e $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$
Proprietà riflessiva	$f(n) = \Theta(f(n))$ $f(n) = O(f(n))$ $f(n) = \Omega(f(n))$
Proprietà simmetrica	$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
Simmetria trasposta	$f(n) = \mathrm{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$ $f(n) = \mathrm{o}(g(n)) \Leftrightarrow g(n) = \omega(f(n))$
Tricotomia	Se a e b sono due numeri reali qualsiasi deve valare solo una delle seguenti relazioni: $\mathrm{a} < \mathrm{b}, \mathrm{a} = \mathrm{b}, \mathrm{a} > \mathrm{b}$

4.1.7 Promemoria dagli esercizi

La funzione all'interno della notazione asintotica è quella che stiamo considerando come limite.

Dobbiamo ricordare che i logaritmi possono cambiare base senza problemi.

4.2 NOTAZIONE STANDARD E FUNZIONI COMUNI

4.2.1 Monotonicità

A function $f(n)$ is monotonically increasing if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is monotonically decreasing if $m \leq n$ implies $f(m) \geq f(n)$. A function $f(n)$ is strictly increasing if $m < n$ implies $f(m) < f(n)$ and strictly decreasing if $m < n$ implies $f(m) > f(n)$.

4.2.2 Floor e ceiling

Floor ⌊…⌋ difetto

Ceiling ⌈…⌉ eccesso

4.2.3 Aritmetica modulare

Per qualsiasi intero a e intero positivo n, a mod n è il resto (o residuo) del quoziente a/n. Se a mod n = b mod n, a è congruo a b.

4.2.4 Polinomi

Dato un intero non negativo *d*, un polinomio n di grado *d* è una funzione p(n) della forma

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

dove le costanti a_0, a_1, …, a_d sono i coefficienti del polinomio e $a_d \neq 0$. Un polinomio è **asintoticamente positivo** se e solo se $a_d > 0$.

4.2.5 Esponenziali

$a^0 = 1$	$a^{-1} = 1/a$	$(a^m)^n = (a^n)^m$
$a^1 = a$	$(a^m)^n = a^{mn}$	$a^m a^n = a^{m+n}$

4.2.6 Logaritmi

$\lg n = \log_2 n$	logaritmo binario
$\ln n = \log_e n$	logaritmo naturale
$\lg^k n = (\lg n)^k$	elevamento a potenza
$\lg \lg n = \lg(\lg n)$	composizione

$\lg n + k$ significa $(\lg n) + k$.

$$a = b^{\log_b a}$$
$$\log_c(ab) = \log_c a + \log_c b$$
$$\log_b a^n = n \log_b a$$
$$\log_b a = \frac{\log_c a}{\log_c b}$$
$$\log_b(1/a) = -\log_b a$$
$$\log_b a = \frac{1}{\log_a b}$$
$$a^{\log_b c} = c^{\log_b a}$$

4.2.7 Fattoriali

La notazione n! (n fattoriale) è definita per i numeri interi n ≥ 0:

$$n! = \begin{cases} 1 & se\ n = 0 \\ n \cdot (n-1)! & se\ n > 0 \end{cases}$$

4.2.8 Iterazione di una funzione

Usiamo la notazione $f^{(i)}(n)$ per per denotare la funzione $f(n)$ applicata iterativamente *i* volte a un valore iniziale di n. Sia $f(n)$ una funzione definita sui reali. Per interi non negativi i, definiamo:

$$f^{(i)}(n) = \begin{cases} n & se\ i = 0 \\ f(f^{(i-1)}(n)) & se\ i > 0 \end{cases}$$

4.2.9 La funzione logaritmo iterato

Utilizziamo la notazione lg*n ("log stella di n") per denotare il logaritmo iterato. Non è $lg^{(i)}n$.

È definita così:

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

È una funzione che cresce molto lentamente:

*lg*2 = 1*

*lg*4 = 2*

*lg*65536 = 4*

lg$(2^{65536}) = 5$*

4.2.10 Numeri di Fibonacci

Ogni numero di fibonacci è la somma dei due numeri precedenti:

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$

5 DIVIDE-ET-IMPERA

Dal capitolo 3.3.1:

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach:

4. **Divide**: divide il problema in sottoproblemi più piccoli
5. **Conquista**: risolve i problemi ricorsivamente. Se il sottoproblema è abbastanza piccolo lo risolve direttamente
6. **Combina**: unisce tutte le soluzioni dei sottoproblemi per generare la soluzione completa

The **merge sort algorithm** closely follows the divide-and-conquer paradigm.

4. **Divide**: divide la sequenza di n elementi in due sotto sequenze di n/2 elementi
5. **Conquista**: ordina ricorsivamente le sotto sequenze
6. **Combina**: unisce tutte le soluzioni dei sottoproblemi per generare la soluzione completa

The recursion "bottoms out" when the sequence to be sorted has length 1.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step.

Recurrences go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the running times of divide-and-conquer algorithms. A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, in Section 3.3.2 we described the worst-case running time T(n) of the MERGE-SORT procedure by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

whose solution we claimed to be $T(n) = \theta(n \log n)$.

Recurrences can take many forms. For example, a recursive algorithm might divide subproblems into unequal sizes, such as a 2/3 to 1/3 split. If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence $T(n) = T(2n/3) + T(n/3) + \theta(n)$.

This chapter offers three methods for solving recurrences—that is, for obtaining asymptotic "θ" or "O" bounds on the solution:

- **In the substitution method**, we guess a bound and then use mathematical induction to prove our guess correct.
- **The recursion-tree method** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
- **The master method** provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$, and $f(n)$ is a given function. Such recurrences arise frequently. A recurrence of the form in equation (4.2) characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps

together take $f(n)$ time. To use the master method, you will need to memorize three cases, but once you do that, you will easily be able to determine asymptotic bounds for many simple recurrences.

5.1 IL METODO DI SOSTIUZIONE

We start in this section with the "substitution" method. The substitution method for solving recurrences comprises two steps:

- Indovinare la forma della soluzione
- Usare l'induzione matematica per trovare le costanti che dimostrano che la soluzione funziona

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence the name "substitution method." This method is powerful, but we must be able to guess the form of the answer in order to apply it.

We can use the substitution method to establish either upper or lower bounds on a recurrence.

ESEMPIO

let us determine an upper bound on the recurrence $\mathrm{T(n)} = 2\mathrm{T}\left(\left\lfloor\frac{\mathrm{n}}{2}\right\rfloor\right) + \mathrm{n}$. We guess that the solution is $T(n) = O(n \log n)$. The substitution method requires us to prove that $T(n) < cn \log n$ for an appropriate choice of the constant $c > 0$.

Avremo:

$$\begin{aligned} T(n) &\leq 2(c\lfloor \mathrm{n}/2 \rfloor \lg(\lfloor \mathrm{n}/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \end{aligned}$$

L'ultimo passo è vero quando c ≥ 1.

Dobbiamo ora dimostrare che la nostra soluzione vale per le condizioni al contorno.

Dobbiamo dimostrare che è possibile scegliere una costante c sufficientemente grande in modo che il limite $T(n) \leq cn \lg n$ sia valido anche per le condizioni al contorno. Sfruttiamo la notazione asintotica che ci richiede di provare che $T(n) \leq cn \lg n$ per $n > n_0$ dove n_0 è una costante *arbitrariamente scelta*. In questo caso è sufficiente scegliere c ≥ 2.

5.1.1 Formulare una buona ipotesi

Se una ricorrenza è simile a una già vista, ha senso provare una soluzione analoga.

Ad esempio, prendiamo $T(n) = 2T(\lfloor \mathrm{n}/2 \rfloor + 17) + n$.

Il termine aggiuntivo 17 non influisce in modo sostanziale alla soluzione della ricorrenza. Quando *n* è grande, la differenza tra $T(n) = 2T(\lfloor \mathrm{n}/2 \rfloor + 17) + n$ e $T(n) = 2T(\lfloor \mathrm{n}/2 \rfloor) + n$ non è così grande: *n* viene diviso circa a metà in entrambi i casi.

Un altro metodo è quello di dimostrare dei limiti superiori e inferiori laschi (non stretti) per la ricorrenza e poi ridurre man a mano il grado di incertezza.

5.1.2 Finezze

Ci sono casi in cui, pur avendo ipotizzato correttamente un limite asintotico per la soluzione della ricorrenza, i conti non tornano. Spesso ciò accade perché l'ipotesi induttiva non è abbastanza forte per dimostrare il limite esatto. A volte basta correggere l'ipotesi sottraendo un termine di ordine inferiore.

5.1.3 Cambio delle variabili

Si possono anche rappresentare le variabili tramite variabili ausiliare.

$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \lg n$$

Poniamo m = lg n

$T(2^m) = 2T(2^{m/2}) + m$

Poniamo S(m) = $T(2^m)$

$S(m) = 2S(m/2) + m$ che ha soluzione $S(m) = O(m \lg m)$

Ripristinansdo T, otteniamo $T(n) = O(m \lg m) = O(\lg n \lg \lg n)$

5.2 METODO DELL'ALBERO RICORSIVO PER RISOLVERE LE RICORRENZE

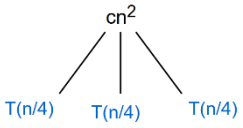
In un albero di ricorsione ogni nodo rappresenta il costo di un singolo sottoproblema. Sommiamo i costi all'interno di ogni livello per avere un insieme di costi per livello, poi sommiamo tutti i costi per livello per ottenere il costo totale.

Un albero è un ottimo metodo per avere un'ipotesi, che poi verrà verificata con il metodo di sostituzione (ma possiamo anche usarlo come prova diretta di una soluzione di ricorrenza).

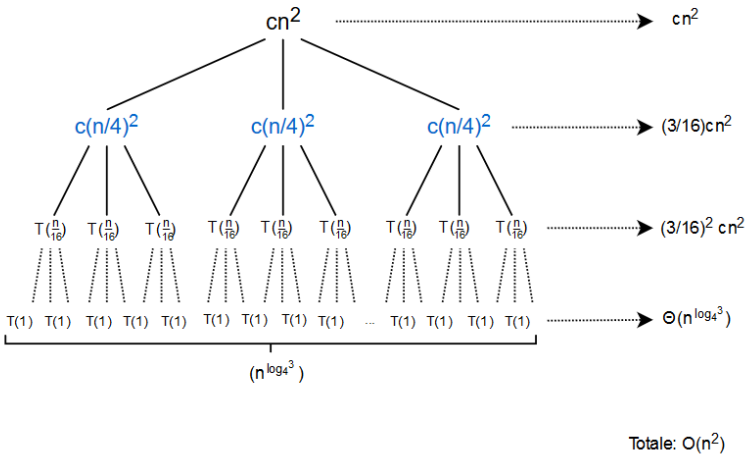
ESEMPIO

Sia data la ricorrenza $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

Iniziamo cercando un limite superiore per la soluzione.



Sappiamo che floor e ceiling non influiscono sulla risoluzione delle ricorrenze; creiamo un albero di ricorsione per la ricorrenza $T(n) = 3T(\lfloor n/4 \rfloor) + cn^2$, ricordando che *c* è costante.



Le dimensioni dei sottoproblemi diminuiscono di un fattore 4 ogni volta che scendiamo di livello. Alla fine dovemmo giungere a una condizione al contorno.

La dimensione del sottoproblema per un nodo è alla profondità *i* è $n/4^i$. La dimensione del sottoproblema diventa quindi n=1 quando $n/4^i$ =1 ovvero quando i = $\log_4 n$.

L'albero ha $\log_4 n$ +1 livelli.

Determiniamo ora il costo a ogni livello dell'albero. Ogni livello ha 3 volte i nodi del livello precedente, quindi il numero di nodi alla profondità i è 3^i. Poiché le dimensioni di un sottoproblema diminuiscono di 4 volte ogni volta, ogni nodo alla profondità i ha un costo di $c(n/4^i)^2$. Il costo totale di tutti i nodi alla profondità i è $(3/16)^i cn^2$. Nell'ultimo livello ogni nodo ha costo T(1), per un costo totale di $n^{log_4 3}\ T(1)$, che è $\Theta\ n^{log_4 3}$, perché supponiamo che T(1) sia una costante.

Sommiamo ora il costo di tutti i livelli. La formula si presenta complicata. Possiamo però approssimare ancora e usare come limite superiore una serie geometrica decrescente infinita.

Il costo totale dell'albero è dominato dal costo della radice.

Possiamo ora usare il metodo di sostituzione per dimostrare che la nostra ipotesi era corretta, ovvero $T(n)=O(n^2)$ è un limite superiore per la ricorrenza $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.

Dobbiamo dimostrare che $T(n) \leq dn^2$ per qualche costante $d > 0$.

Otteniamo:
$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \\ &\leq dn^2 \end{aligned}$$

L'ultimo passaggio è vero quando $d \geq (16/13)c$.

5.3 IL METODO DELL'ESPERTO PER RISOLVERE LE RICORRENZE

Si usa per risolvere ricorrenze della forma ***T(n) = aT(n/b)+f(n)***, dove $a \geq 1$ e $b > 1$ sono costanti e f(n) è asintoticamente positiva.

Questa ricorrenza descrive il tempo di esecuzione di un algoritmo che divide un problema di dimensione n in a sottoproblemi, ciascuno di dimensione n/b. I sottoproblemi vengono risolti in modo ricorsivo, ciascuno nel tempo T(n/b).

Teorema: date le costanti a ≥ 1 e b > 1 e la sua funzione f(n), sia T(n) una funzione definita sufli interi non negativi della ricorrenza ***T(n) = aT(n/b)+f(n)*** dove n/b rappresenta ⌊n/b⌋ o ⌈n/b⌉. Allora T(n) può essere asintoticamente limitata nei seguenti modi:

- se f(n) = $\boldsymbol{O}(n^{log_b\, a-\varepsilon})$ per qualche costante $\varepsilon > 0$, allora T(n) = $\Theta(n^{log_b\, a})$
- se f(n) = $\Theta(n^{log_b\, a})$, allora T(n) = $\Theta(n^{log_b\, a} \lg n)$
- se f(n) = $\Omega(n^{log_b\, a+\varepsilon})$ per qualche costante $\varepsilon > 0$ e se af(n/b) ≤ cf(n) per qualche costante c < 1 e per ogni n sufficientemente grande, allora T(n) = $\Theta(f(n))$.

Nel primo caso f(n) deve essere polinomialmente più piccola di $n^{log_b\, a}$, ovvero f(n) deve essere asintoticamente più piccola di $n^{log_b\, a}$ per un fattore n^{ε} per qualche costante $\varepsilon > 0$.

Nel terzo caso f(n) deve essere polinomialmente più grande grande di $n^{log_b\, a}$, e soddisfare le condizioni di regolarità af(n/b) ≤ cf(n).

I tre casi non coprono tutte le funzioni possibili di f(n).

ESEMPIO

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

For this recurrence, we have $a = 9, b = 3, f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\varepsilon})$, where $\varepsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \theta(n^2)$.

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

For this recurrence, we have $a = 1, b = \frac{3}{2}, f(n) = 1$, and thus we have that $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Since $f(n) = \theta\left(n^{\log_b a}\right) = \theta(1)$, where $\varepsilon = 1$, we can apply case 2 of the master theorem and conclude that the solution is $T(n) = \theta(\log n)$.

$$T(n) = 3T\left(\frac{n}{4}\right) + \mathrm{n}\log n$$

For this recurrence, we have $a = 3, b = 4, f(n) = \mathrm{n}\log n$, and thus we have that $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$. Since $f(n) = \Omega(n^{\log_4 3+\varepsilon})$, where $\varepsilon = 1$, we can case 3 applies if we can show that the regularity condition holds for f(n). For sufficiently large n, we have that $af\left(\frac{n}{b}\right) = 3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq \left(\frac{3}{4}\right) n\log n = cf(n)$ for $c = \frac{3}{4}$. Consequently, by case 3, the solution to the recurrence is $T(n) = \theta(n\log n)$.

6 ORDINAMENTO E STATISTICHE DI ORDINE

La struttura dei dati

In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a record. Each record contains a key, which is the value to be sorted. The remainder of the record consists of satellite data, which are usually carried around with the key.

In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

A sorting algorithm describes the method by which we determine the sorted order, regardless of whether we are sorting individual numbers or large records containing many bytes of satellite data. Thus, when focusing on the problem of sorting, we typically assume that the input consists only of numbers.

Perchè ordinare?

Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms. There are several reasons:

- Certe volte un'applicazione richiede implicitamente informazioni ordinate.
- Spesso gli algoritmi utilizzano l'ordinamento come una subroutine chiave
- Possiamo scegliere tra una grande varietà di algoritmi di ordinamento, tutti con un ricco insieme di tecniche
- Possiamo provare un limite inferiore per l'ordinamento
- Molti problemi di ingegneria arrivano ad una soluzione quando viene implementato un algoritmo di ordinamento

Algoritmi di ordinamento

Insertion sort takes n^2 time in the worst case. Because its inner loops are tight, however, it is a fast in-place sorting algorithm for small input sizes.

Merge sort has a better asymptotic running time, $n \log n$, but theMERGE procedure it uses does not operate in place.

In this part, we shall introduce two more algorithms that sort arbitrary real numbers:

- Heapsort sorts n numbers in place in $O(n \log n)$ time.
- Quicksort, in Chapter 7, also sorts n numbers in place, but its worst-case running time is $\theta(n^2)$. Its expected running time is $O(n \log n)$, however, and it generally outperforms heapsort in practice.

Insertion sort, merge sort, heapsort, and quicksort are all comparison sorts: they determine the sorted order of an input array by comparing elements.

We prove a lower bound of $\Omega(n \log n)$ in the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)

We'll show that we can beat this lower bound of $\Omega(n \log n)$ if we can gather information about the sorted order of the input by means other than comparing elements.

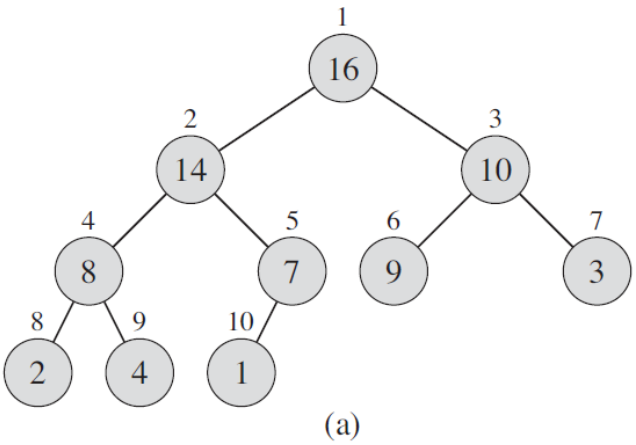
6.1 HEAPSORT

In this chapter, we introduce another sorting algorithm: heapsort. Like merge sort, but unlike insertion sort, heapsort's running time is $O(n \log n)$. Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

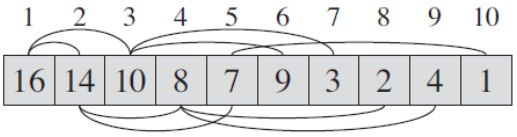
Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a "heap," to manage information.

6.1.1 Heaps

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree.



(a)



(b)

The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

An array A that represents a heap is an object with two attributes:

- A.length, which (as usual) gives the number of elements in the array

- A.heap-size, which represents how many elements in the heap are stored within array A.

The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

- Parent(i)
  - Return ⌊i/2⌋
- Left(i)
  - Return 2i
- Right(i)
  - Return 2i+1

There are two kinds of binary heaps:

- In a max-heap, the max-heap property is that for every node I other than the root, $A[Parent(i)] \geq A[i]$
- the min-heap property is that for every node i other than the root $A[Parent(i)] \leq A[i]$. The smallest element in a min-heap is at the root.

For the heapsort algorithm, we use max-heaps.

Viewing a heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is $\theta(\log n)$.

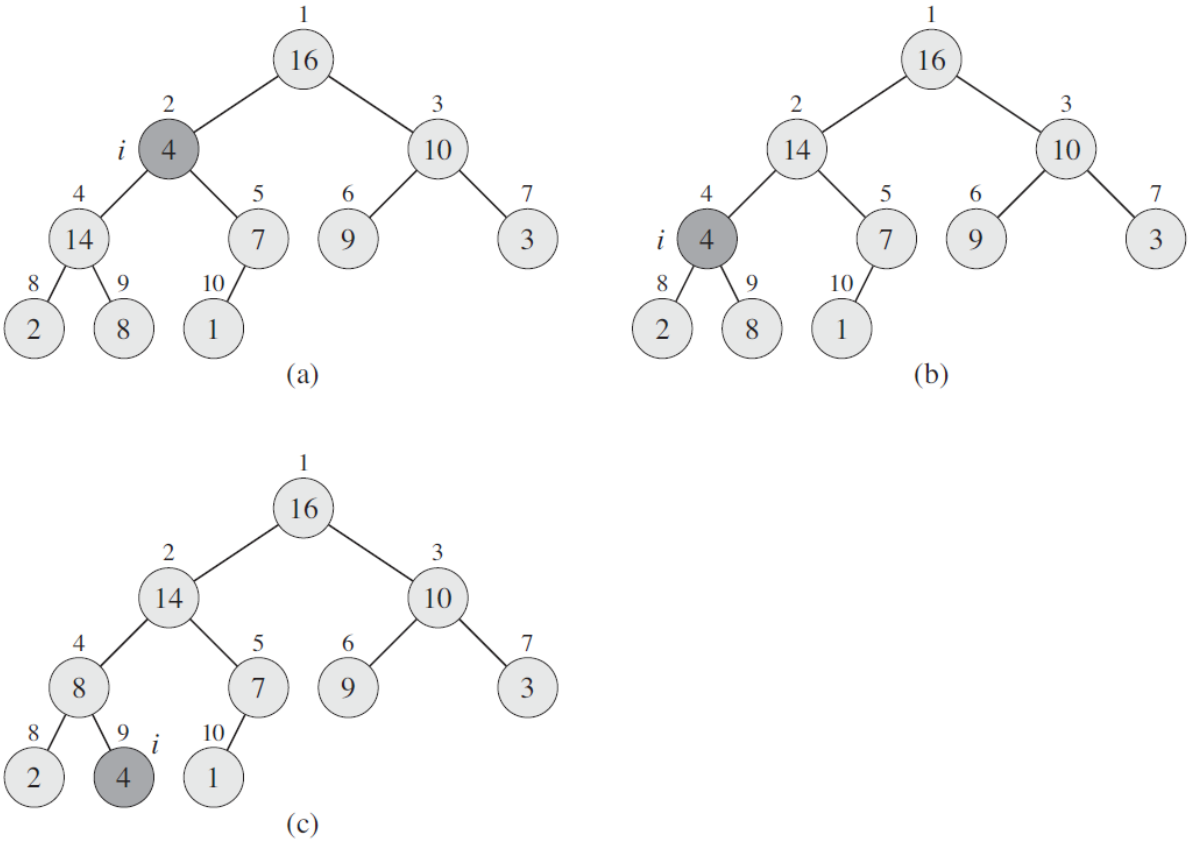
- The MAX-HEAPIFY procedure, which runs in $O(\log n)$ time, is the key to maintaining the max-heap property.
- The BUILD-MAX-HEAP procedure, which runs in linear time, produces a maxheap from an unordered input array.
- The HEAPSORT procedure, which runs in $O(\log n)$ time, sorts an array in place.
- The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM procedures, which run in $O(\log n)$ time, allow the heap data structure to implement a priority queue.

6.2 MANTENERE LA PROPRIETÀ DELL'HEAP

MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

MAX-HEAPIFY(A, I)

1 $l = Left(i)$
2 $r = Right(i)$
3 $\boldsymbol{If}\ l \leq A, heap - size\ and\ A[l] > A[i]$
4 $\quad largest = l$
5 $\boldsymbol{else}\ largest = i$
6 $\boldsymbol{if}\ r \leq A.heap.size\ and\ A[r] > A[largest]$
7 $\quad largest = r$
8 $\boldsymbol{if}\ largest \neq i$
9 $\quad exchange\ A[i]\ with\ A[largest]$
10 $\quad MAX - HEAPIFY\ (A, largest)$



The action of MAX-HEAPIFY.A; 2/, where A.heap-size = 10.

a) The initial configuration, with A[2] at node i = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in
b) by exchanging A[2] with A[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY(A, 4) now has i = 4. After swapping A[4] with A[9], as shown in
c) node 4 is fixed up, and the recursive call MAX-HEAPIFY(A, 9) yields no further change to the data structure.

The solution to this recurrence, by the master theorem is$T(n) = O(\log n)$. Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

6.3 COSTRUIRE UN HEAP

We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A[1..n] where $n = A.length$, into a max-heap.

BUILD-MAX-HEAP(A)
1 $A.heap - size = A.length$
2 $\boldsymbol{for}\ i = \left\lfloor \mathrm{A}.\frac{\mathrm{length}}{2} \right\rfloor \boldsymbol{downto}\ 1$
3 $MAX - HEAPIFY(A, i)$

To show why BUILD-MAX-HEAP works correctly, we use the following loop invariant:

At the start of each iteration of the for loop of lines 2–3, each node $i+1, i+2..n$ is the root of a max-heap.

Inizializzazione: Prior to the first iteration of the loop, $i = \left\lfloor \frac{n}{2} \right\rfloor$. Each node $\left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor + 2..n$ is a leaf and is thus the root of a trivial max-heap.

Mantenimento: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i . By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call $MAX - HEAPIFY(A, i)$ to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2..n$ are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.

Terminazione: At termination, $i = 0$. By the loop invariant, each node $1, 2..n$ is the root of a max-heap. In particular, node 1 is.

Each call to MAX-HEAPIFY costs $O(\log n)$ time, and BUILDMAX- HEAP makes $O(n)$ such calls. Thus, the running time is $O(\mathrm{n}\log n)$. This upper bound, though correct, is not asymptotically tight.

We can bound the running time of BUILD-MAX-HEAP as $O\left(n\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = \left(n\sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$

6.4 L'ALGORITMO HEAPSORT

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input $A[1..n]$ where $n = A.length$. Since the maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position by exchanging it with $A[n]$. If we now discard node n from the heap - and we can do so by simply decrementing $A.heap - size$ - we observe that the children of the root remain max-heaps, but the new root element might violate the max-heap property.

HEAPSORT (A)
1 $A.heap - size = A.length$
2 ***for*** $i =$ A. length ***downto*** 1
3 $exchange\ A[1]\ with\ A[i]$
4 $A.heap - size = A.heap - size - 1$
5 $MAX - HEAPIFY(A, 1)$

The HEAPSORT procedure takes time $O(n \log n)$, since the call to BUILD-MAXHEAP takes time $O(n)$ and each of the $n = 1$calls to MAX-HEAPIFY takes time $O(\log n)$.

Θ	θ	O
$\Omega(\boldsymbol{n\, log_2\, n})$	$\theta(n\, log_2\, n)$	$O(n\, log_2\, n)$

6.5 CODE DI PRIORITÀ

Heapsort is an excellent algorithm, but a good implementation of quicksort usually beats it in practice.

One of the most popular applications of a heap is as an efficient priority queue. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues.

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:

- INSERT(S, x) inserts the element x into the set S, which is equivalent to the operation S = S U {x}

- MAXIMUM(S) returns the element of S with the largest key.
- EXTRACT-MAX(S) removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Among their other applications, we can use max-priority queues to schedule jobs on a shared computer.

Alternatively, a min-priority queue supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY.

A min-priority queue can be used in an event-driven simulator.

When we use a heap to implement a priority queue, therefore, we often need to store a handle to the corresponding application object in each heap element. The exact makeup of the handle (such as a pointer or an integer) depends on the application. Similarly, we need to store a handle to the corresponding heap element in each application object.

The procedure HEAP-MAXIMUM implements the MAXIMUM operation in $\theta(1)$ time.

```
HEAP-MAXIMUM(A)
1   return A[1]
```

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation. It is similar to the for loop body (lines 3–5) of the HEAPSORT procedure.

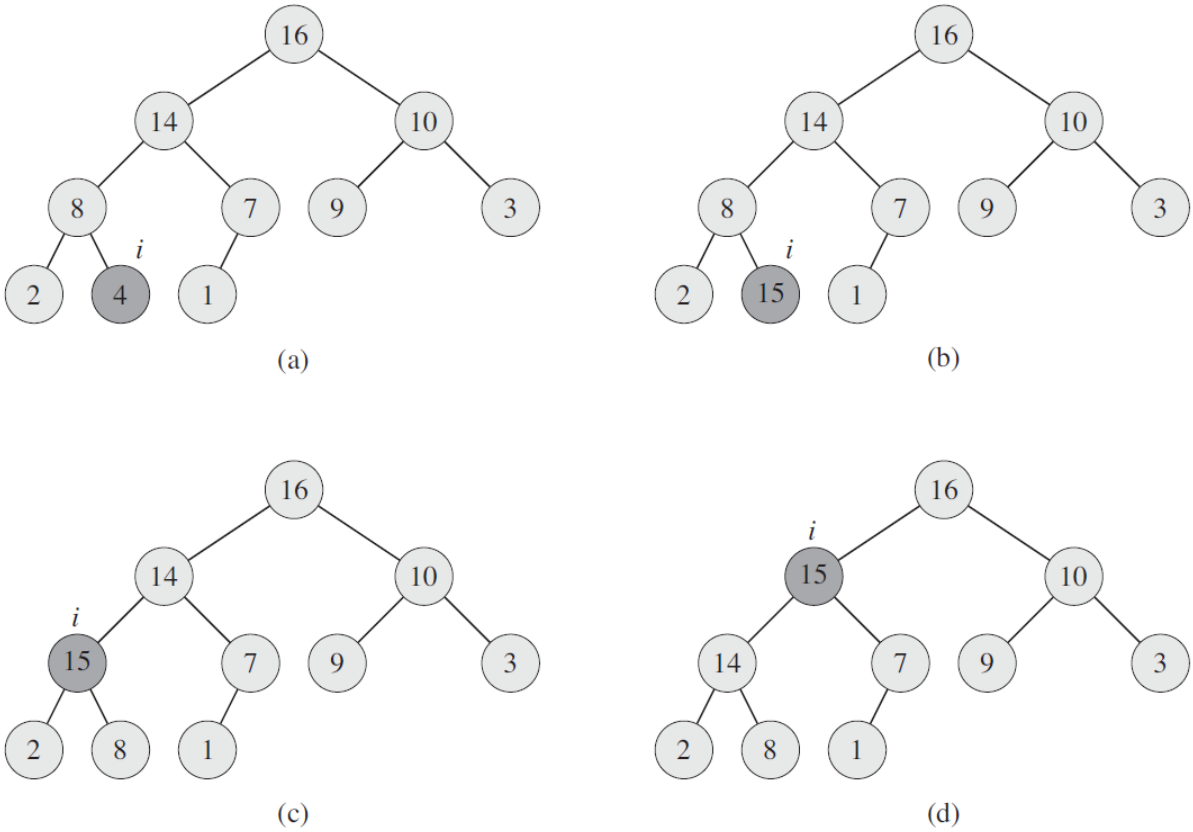
```
HEAP-EXTRACT-MAX(A)
1   if A.heap − size < 1
2         error "heap underflow"
3   max = A[1]
4   A[1] = A[A.heap − size]
5   A.heap − size = A.heap − size – 1
6   MAX − HEAPIFY(A, 1)
7   return max
```

The running time of HEAP-EXTRACT-MAX is $O(\log n)$, since it performs only a constant amount of work on top of the $O(\log n)$ time for MAX-HEAPIFY.

```
HEAP-INCREASE-KEY(A, i, key)
1   if key < A[i]
2         error "new key is smaller than current key"
3   A[i] = key
4   while i > 1 and A[Parent(i)] < A[i]
5         exchange A[i]with A[Parent(i)]
6         i = Parent(i)
```

As HEAP-INCREASEKEY traverses this path, it repeatedly compares an element to its parent, exchanging their keys and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds

The running time of HEAP-INCREASE-KEY on an n-element heap is $O(\log n)$, since the path traced from the node updated in line 3 to the root has length $O(\log n)$.



The operation of HEAP-INCREASE-KEY:

a) The max-heap of the figure (a) with a node whose index is i heavily shaded.
b) This node has its key increased to 15.
c) After one iteration of the while loop of lines 4–6, the node and its parent have exchanged keys, and the index I moves up to the parent.
d) The max-heap after one more iteration of the while loop. At this point, $A[Parent(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new element to be inserted into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.

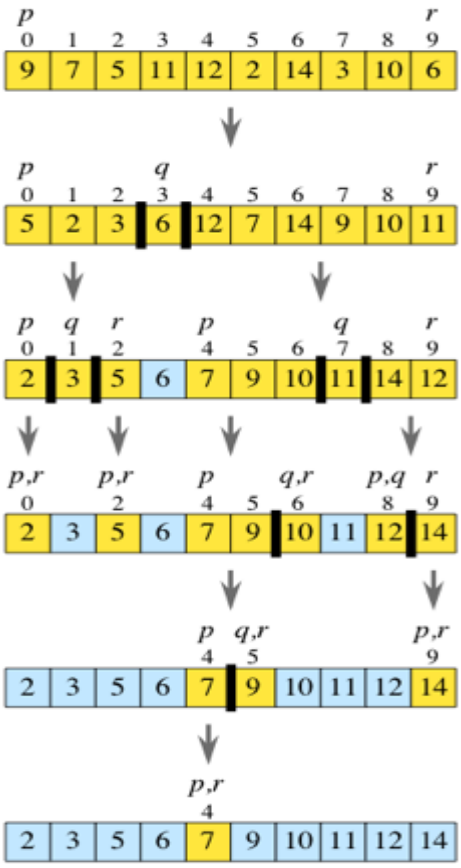
```
MAX-HEAP-INSERT(A, key)
1   A.heap − size = A.heap − size + 1
2   A[A.heap − size] = −∞
3   HEAP − INCREASE − KEY(A, A.heap − size, key)
```

The running time of MAX-HEAP-INSERT on an n-element heap is $O(\log n)$.

7 QUICKSORT

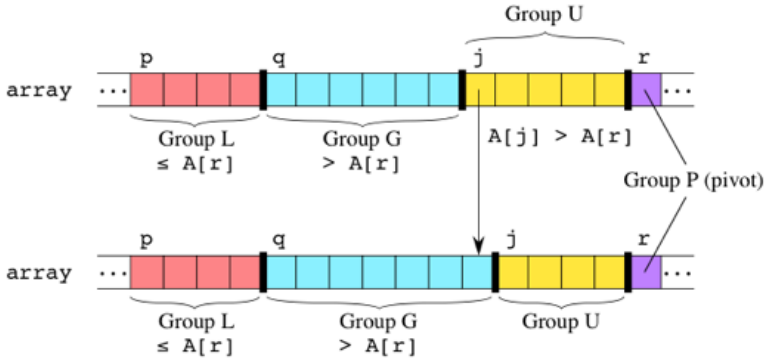
The quicksort algorithm has a worst-case running time of $\theta(n^2)$ on an input array of n numbers. Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is $\theta(n \log n)$, and the constant factors hidden in the $\theta(n \log n)$ notation are quite small. It also has the advantage of sorting in place and it works well even in virtual-memory environments.

7.1 DESCRIZIONE DI QUICKSORT



Quicksort, like merge sort, applies the divide-and-conquer paradigm.

- **Divide**: Partition (rearrange) the array A[p..r] into two (possibly empty) subarrays A[p..q-1] and A[q+1..r] such that each element of A[p..q-1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.
- **Conquista**: Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort.
- **Combina**: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p..r] is now sorted.

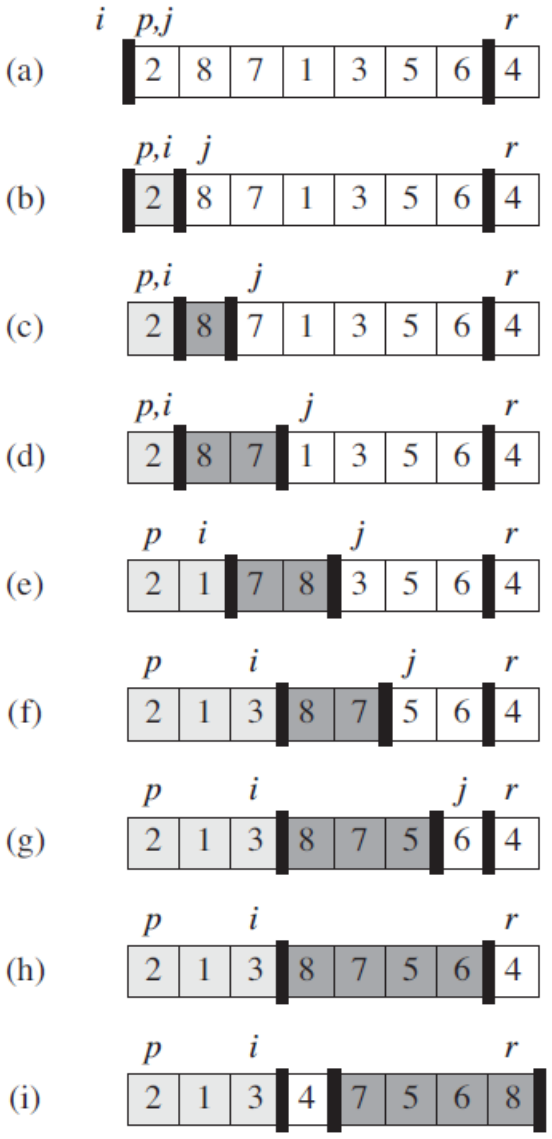



```
QUICKSORT(A, p, r)
1   if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, A.length).

7.1.1 Partizionamento dell'array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p..r]$ in place.




```
PARTITION(A, p, r)
1   x = A[r]
2   i = p − 1
3   for j = p to r − 1
4       if A[j] ≤ x
5           i = i + 1
6           exchange A[i] with A[j]
7   exchange A[i + 1] with A[r]
8   return i + 1
```

PARTITION always selects an element $x = A[r]$ as a pivot element around which to partition the subarray $A[p..r]$. As the procedure runs, it partitions the array into four (possibly empty) regions.

We state these properties as a loop invariant:

At the beginning of each iteration of the loop of lines 3–6, for any array index k,

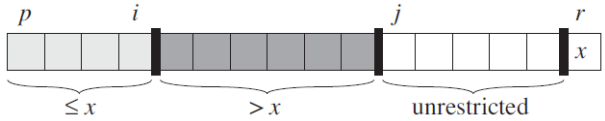
- **if** $p \le k \le i$, then $A[k] \le x$
- **if** $i + 1 \le k \le j - 1$, then $A[k] > x$
- **if** $k = r$, then $A[k] = x$

- Inizializzazione: Prior to the first iteration of the loop, I =p-1 and j = p. Because no values lie between p and i and no values lie between i+1 and j+1, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition
- Mantenimento: As the figure shows, we consider two cases, depending on the outcome of the test in line 4. Figure(a) shows what happens when $A[j] > x$; the only action in the loop is to increment j . After j is incremented, condition 2 holds for $A[j-1]$ and all other entries remain unchanged. Figure(b) shows what happens when $A[j] \le x$; the loop increments i, swaps $A[i]$ and $A[j]$, and then increments j . Because of the swap, we now have that $A[i] \le x$, and condition 1 is satisfied. Similarly, we also have that $A[j-1] > x$, since the item that was swapped into $A[j-1]$ is, by the loop invariant, greater than x.
- Terminazione: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets: those less than or equal to x, those greater than x, and a singleton set containing x.

The operation of PARTITION on a sample array. Array entry A[r] becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements

are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x.

a) The initial array and variable settings. None of the elements have been placed in either of the first two partitions.
b) The value 2 is "swapped with itself" and put in the partition of smaller values.
c) The values 8 and 7 are added to the partition of larger values
d) The values 8 and 7 are added to the partition of larger values.
e) The values 1 and 8 are swapped, and the smaller partition grows.
f) The values 3 and 7 are swapped, and the smaller partition grows.
g) The larger partition grows to include 5 and 6, and the loop terminates.
h) The larger partition grows to include 5 and 6, and the loop terminates.
i) In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

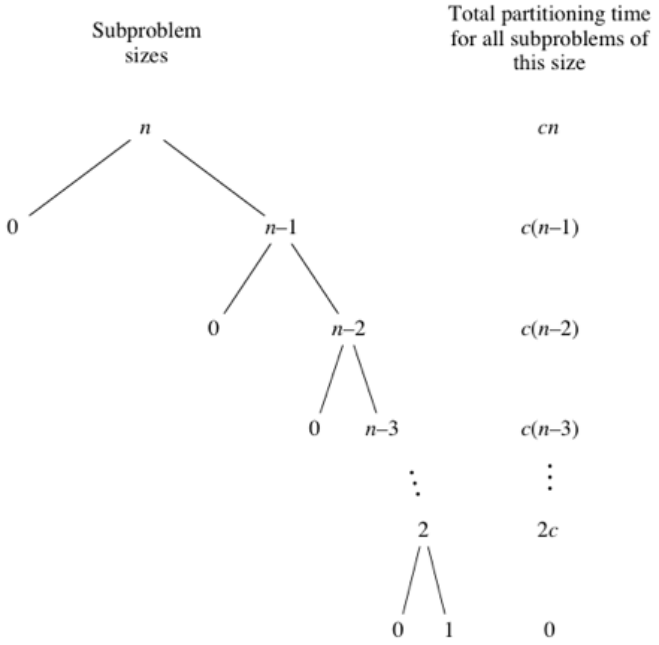


The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$.

The subarray $A[j..r-1]$ can take on any values.

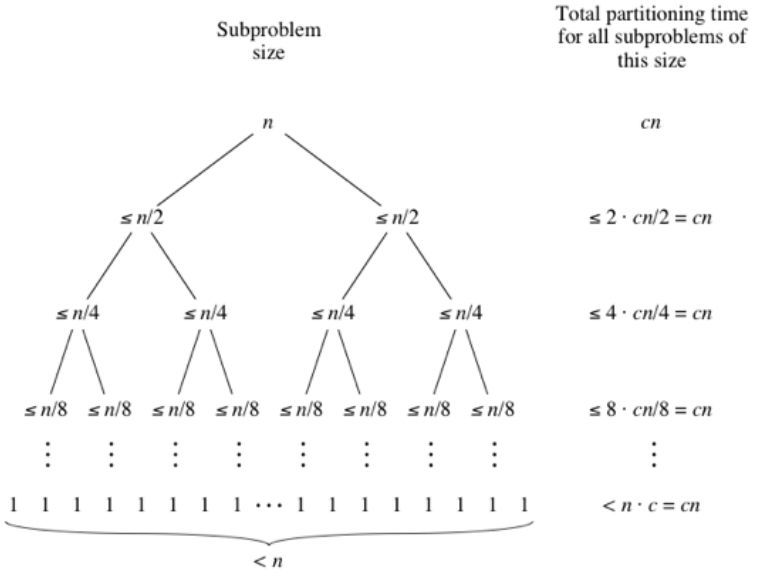
7.2 PERFORMANCE DEL QUICKSORT

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort.



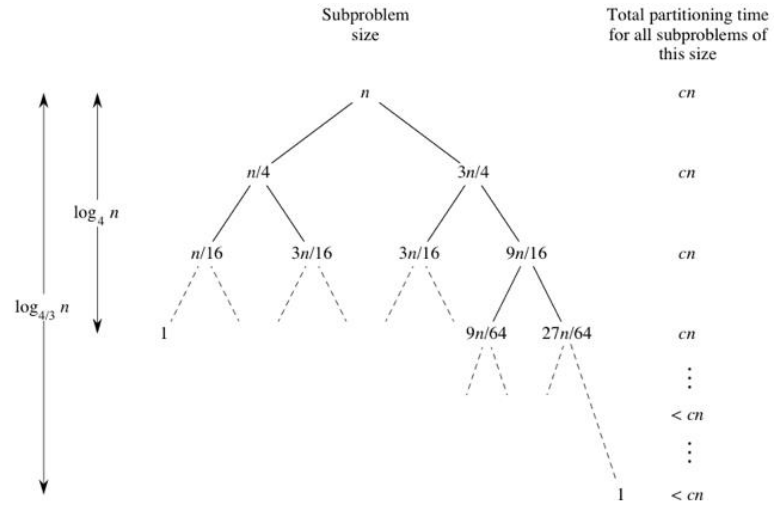
7.2.1 Caso peggiore

The worst-case behaviour for quicksort occurs when the partitioning routine produces one subproblem with $n - 1$ elements and one with 0 elements. The partitioning costs $\theta(n)$ time. If we sum the costs incurred at each level of the recursion we get $\theta(n^2)$.



7.2.2 Caso migliore

In the most even possible split, PARTITION produces two subproblems, each of size no more than $\frac{n}{2}$, since one is of size $\lfloor\frac{n}{2}\rfloor$ and one of size $\left\lfloor\frac{n}{2}\right\rfloor - 1$. This recurrence has the solution $T(n) = \theta(n \log n)$.



7.2.3 Caso bilanciato

The average-case running time of quicksort is much closer to the best case than to the worst case.

Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the

Recurrence $T(n) = T\left(\frac{9n}{20}\right) + T\left(\frac{n}{10}\right) + cn$.

INTUIZIONE PER IL CASO BILANCIATO

In the average case, partition produces a mix of “good” and “bad” splits. In a recursion tree for an average-case execution of partition, the good and bad splits are distributed randomly throughout the tree.

Intuitively, the $\theta(n-1)$ cost of the bad split can be absorbed into the $\theta(n)$ cost of the good split, and the resulting split is good. Thus, the running time of quicksort, when levels alternate between good and bad splits, is like the running time for good splits alone: still $O(n \log n)$, but with a slightly larger constant hidden by the O-notation.

Θ	θ	O
$\Omega(\boldsymbol{n \log n})$	$\boldsymbol{\theta(n \log n)}$	$O(n2)$

7.3 UNA VERSIONE CASUALE DI QUICKSORT

We could do so for quicksort also, but a different randomization technique, called random sampling, yields a simpler analysis. Instead of always using $A[r]$ as the pivot, we will select a randomly chosen element from

the subarray $A[p..r]$. We do so by first exchanging element $A[r]$ with an element chosen at random from $A[p..r]$. By randomly sampling the range $p,..,r$, we ensure that the pivot element $x = A[r]$ is equally likely to be any of the $r - p + 1$ elements in the subarray.

```
RANDOMIZED-PARTITION(A, p, r)
1   i = Random(p,r)
2   exchange A[r]with A[i]
3   return Partition(A,p,r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1   if p < r
2   q = RANDOMIZED − PARTITION(A,p,r)
3           RANDOMIZED − QUICKSORT(A,p,q − 1)
4           RANDOMIZED − QUICKSORT(A,q + 1,r)
```

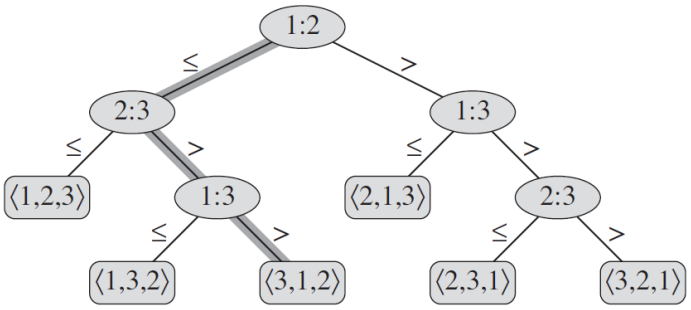
8 ORDINAMENTO IN TEMPO LINEARE

We have now introduced several algorithms that can sort n numbers in $O(n \log n)$ time. Merge sort and heapsort achieve this upper bound in the worst case; quicksort achieves it on average. Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \log n)$ time. These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms comparison sorts.

8.1 LIMITE INFERIORE PER L'ORDINAMENTO

In this section, we assume without loss of generality that all the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless, so we can assume that no comparisons of this form are made.

8.1.1 Il modello ad albero decisionale



We can view comparison sorts abstractly in terms of decision trees. A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

The figure shows the decision tree corresponding to the insertion sort algorithm from Section 2.1 operating on an input sequence of three elements.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the n! permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct.

8.1.1.1 Limite inferiore per il caso peggiore

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.

TEOREMA: Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

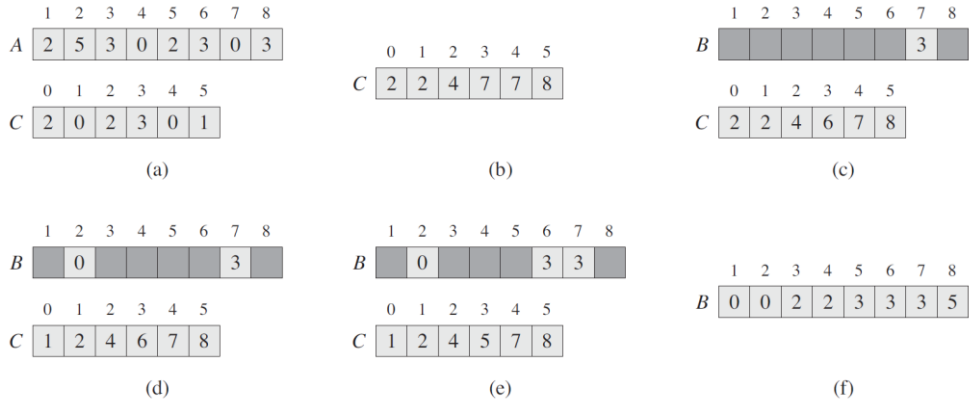
COROLLARIO: Heapsort and merge sort are asymptotically optimal comparison sorts.

8.2 COUNTING SORT

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When $k = O(n)$, the sort runs in $\theta(n)$ time.

Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array.

In the code for counting sort, we assume that the input is an array $A[1..n]$, and thus $A.length = n$. We require two other arrays: the array $B[1..n]$ holds the sorted output, and the array $C[0..k]$ provides temporary working storage.




```
COUNTING-SORT(A, B, k)
1   let C[0..k]be a new array
2   for i = 0 to k
3           C[i] = 0
4   for j = 1 to A.length
5           C[A[j]] = C[A[j]] + 1
6   //C[i] now contains the number of elements equal to i
7   for i = 1 to k
8           C[i] = C[i] + C[i − 1]
9   //C[i] now contains the number of elements less than or equal to i
10  for j = A.length downto 1
11          B[C[A[j]]] = A[j]
12          C[A[j]] = C[A[j]] − 1
```

The figure illustrates counting sort. After the for loop of lines 2–3 initializes the array C to all zeros, the for loop of lines 4–5 inspects each input element. If the value of an input element is i , we increment $C[i]$. Thus, after line 5, $C[i]$ holds the number of input elements equal to i for each integer $i = 0, 1, \dots, k$. Lines 7–8 determine for each $i = 0, 1, \dots, k$ how many input elements are less than or equal to i by keeping a running sum of the array C. Finally, the for loop of lines 10–12 places each element $A[j]$ into its correct sorted position in the output array B.

An important property of counting sort is that it is stable: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array.

How much time does counting sort require? The for loop of lines 2–3 takes time $\theta(k)$, the for loop of lines 4–5 takes time $\theta(n)$, the for loop of lines 7–8 takes time $\theta(k)$, and the for loop of lines 10–12 takes time $\theta(n)$. Thus, the overall time is $\theta(k + n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\theta(n)$.

θ
$\boldsymbol{\theta(n)}$

8.3 RADIX SORT

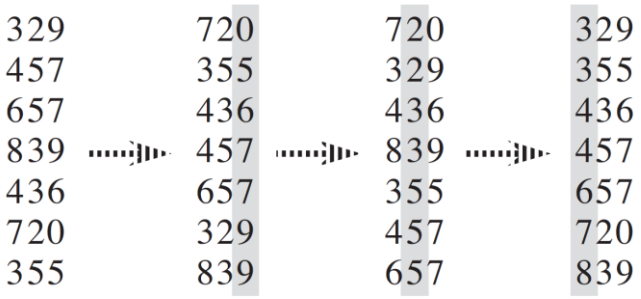
Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched.

For decimal digits, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.)

Radix sort solves the problem of card sorting—counterintuitively—by sorting on the least significant digit first.

We use counting sort as a subroutine for radix sort.

```
RADIX-SORT(A, B, k)
1   for i = 1 to d
2           use a stable sort to sort array A on digit i
```



The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

LEMMA: Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\theta\big(d(n+k)\big)$ time if the stable sort it uses takes $\theta(n+k)$ time.

Lemma: Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT correctly sorts these numbers in $\theta\left(\left(\frac{b}{r}\right)(n+2^r)\right)$ time if the stable sort it uses takes $\theta(n+k)$time for inputs in the range 0 to k.

Θ	θ	O
$\Omega(\boldsymbol{nk})$	$\boldsymbol{\theta}(nk)$	$O(nk)$

9 MEDIANS AND ORDER STATISTICS

The i^{th} order statistic of a set of n elements is the i^{th} smallest element. For example, the minimum of a set of elements is the first order statistic $i = 1$, and the maximum is the nth order statistic ($i = n$).

median, informally, is the "halfway point" of the set. When n is odd, the median is unique, occurring at $i = \frac{n+1}{2}$. When n is even, there are two medians, occurring at $i = \frac{n}{2}$ and $i = \frac{n}{2} + 1$.

9.1 SELECTION IN EXPECTER LINEAR TIME

The general selection problem appears more difficult than the simple problem of finding a minimum. Yet, surprisingly, the asymptotic running time for both problems is the same: $\theta(n)$. In this section, we present a divide-and-conquer algorithm for the selection problem. The algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm of Chapter 7.

```
RANDOMIZED-SELECT(A, p, r, i)
1   if p == r
2         return A[p]
3   q = RANDOMIZED − PARTITION(A, p, r)
4   K = q − p + 1
5   If i == k//the pivot value is the answer
6         return A[q]
7   elseif i < k
8         return RANDOMIZED − SELECT(A, p, q − 1, i)
9   else return RANDOMIZED − SELECT(a, q + 1, r, i − k)
```

The RANDOMIZED-SELECT procedure works as follows: Line 1 checks for the base case of the recursion, in which the subarray $A[p..r]$ consists of just one element. In this case, i must equal 1, and we simply return A[p] in line 2 as the i^{th} smallest element. Otherwise, the call to RANDOMIZED-PARTITION in line 3 partitions the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ Is less than or equal to $A[q]$, which in turn is less than each element of $A[q+1..r]$. As in quicksort, we will refer to $A[q]$ as the pivot element. Line 4 computes the number k of elements in the subarray $A[p..q]$, that is, the number of elements in the low side of the partition, plus one for the pivot element. Line 5 then checks whether $A[q]$ is the i^{th} smallest element. If it is, then line 6 returns $A[q]$. Otherwise, the algorithm determines in which of the two subarrays $A[p..q-1]$ and $A[q+1..r]$ the i^{th} smallest element lies. If $i < k$, then the desired element lies on the low side of the partition, and line 8 recursively selects it from the subarray. If $i > k$, however, then the desired element lies on the high side of the partition. Since we already know k values that are smaller than the i^{th} smallest element of $A[p..r]$—namely, the elements of $A[p..q]$—the desired element is the $(i-k)$ smallest element of $A[q+1..r]$, which line 9 finds recursively.

10 STRUTTURE DATI ELEMENTARI

10.1 STACK AND QUEUES

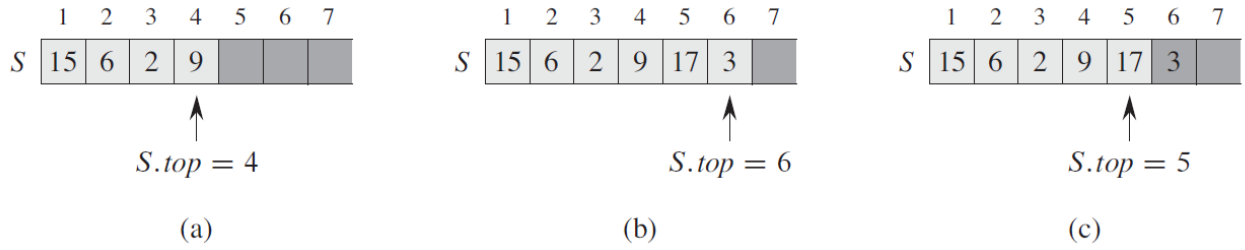
Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified:

- Stack: the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO, policy.
- Queue: the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy.

There are several efficient ways to implement stacks and queues on a computer.

10.1.1 Stacks

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP.



a) Stack S has 4 elements. The top element is 9.
b) Stack S after the calls $PUSH(S, 17)$ and $PUSH(S, 3)$
c) Stack S after the call $POP(S)$ has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

```
STACK-EMPTY(S)
1  if S.top == 0
2        return True
3  else return False
```

```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

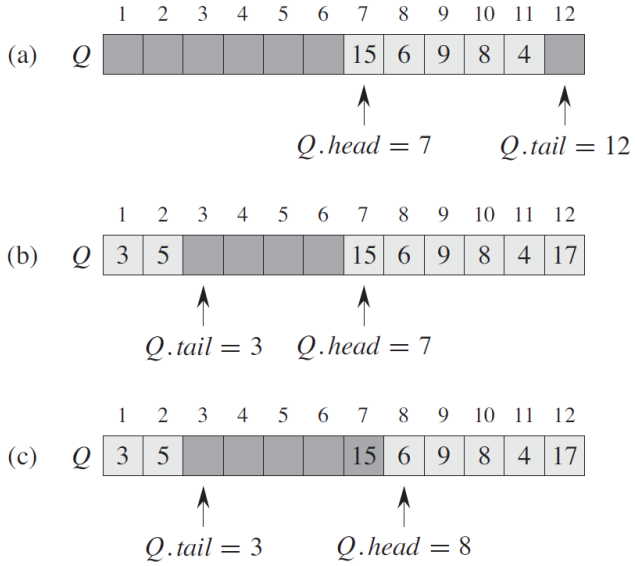
```
POP(S)
1  if STACK − EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

10.1.2 Queues

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting to pay a cashier. The queue has a head and a tail.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```



A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions.

a. The queue has 5 elements, in locations $Q[7..11]$.

b. The configuration of the queue after the calls $ENQUEUE(Q, 17)$, $ENQUEUE(Q, 3)$ and $ENQUEUE(Q, 5)$.

c. The configuration of the queue after the call $DEQUEUE(Q)$ returns the key value 15 formerly at the head of the queue. The new head has key 6.

10.2 LINKED LIST

A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Each element of a doubly linked list L is an object with an attribute key and two other pointer attributes: next and prev.

A list has a head and a tail. If a list is singly linked, we omit the prev pointer in each element. If a list is sorted, the linear order of the list corresponds to the linear order of keys stored in elements of the list; If the list is unsorted, the elements can appear in any order. In a circular list, the prev pointer of the head of the list points to the tail, and the next pointer of the tail of the list points to the head.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3          x = x.next
4  return x
```

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3          L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

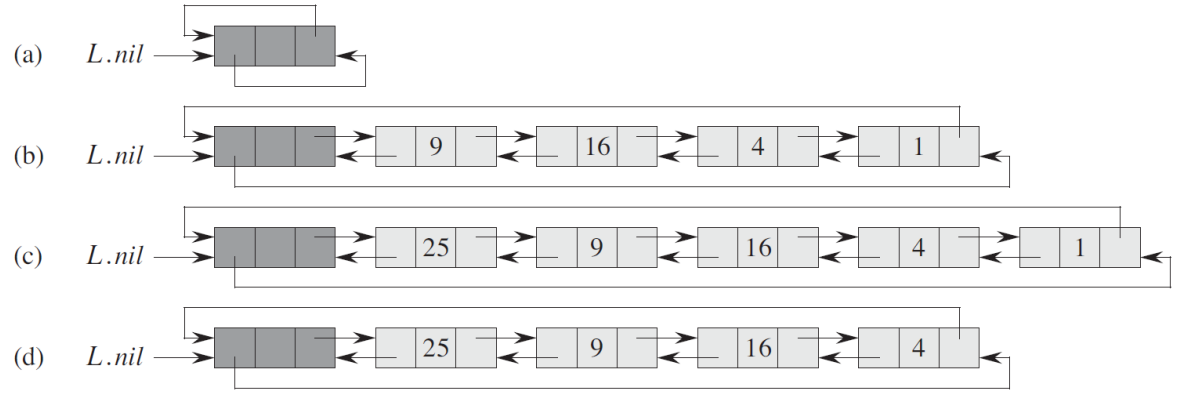
```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2          x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5          x.next.prev = x.prev
```

```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

```
LIST-SEARCH'(L, k)
1  x = L.nil.head
2  while x ≠ L.nil and x.key ≠ k
3          x = x.next
4  return x
```

```
LIST-INSERT'(L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
5  x.prev = NIL
```

The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and tail of the list as shown in LIST-DELETE'.



10.2.1 Sentinelle

A sentinel is a dummy object that allows us to simplify boundary conditions. For example, suppose that we provide with list L an object L.nil that represents NIL but has all the attributes of the other objects in the list. Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel L.nil. this change turns a regular doubly linked list into a circular, doubly linked list with a sentinel, in which the sentinel L.nil lies between the head and tail.

10.3 Implementare puntatori e oggetti

How do we implement pointers and objects in languages that do not provide them?

10.3.1 A multiple-array representation of objects

We can represent a collection of objects that have the same attributes by using an array for each attribute.

For a given array index x, the array entries $key[x]$, $next[x]$, and $prev[x]$ represent an object in the linked list. Under this interpretation, a pointer x is simply a common index into the key, next, and pre———————————— arrays.

10.3.2 A single-array representation of objects

The words in a computer memory are typically addressed by integers from 0 to M - 1, where M is a suitably large integer. In many programming languages, an object occupies a contiguous set of locations in the computer memory. A pointer is simply the address of the first memory location of the object, and we can address other memory locations within the object by adding an offset to the pointer.

10.3.3 Allocating and freeing objects

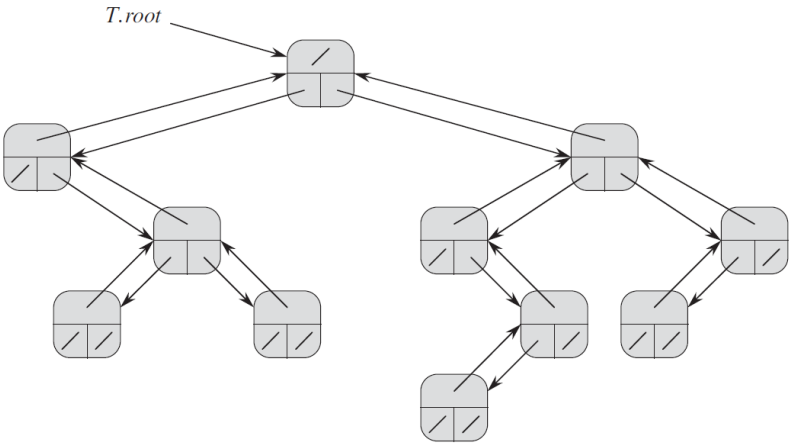
To insert a key into a dynamic set represented by a doubly linked list, we must allocate a pointer to a currently unused object in the linked-list representation. Thus, it is useful to manage the storage of objects not currently used in the linked-list representation so that one can be allocated. In some systems, a garbage collector is responsible for determining which objects are unused.

```
ALLOCATE-OBJECT()
1  if free == nill
2  error "out of space"
3  else x = free
4          free = x.next
```

```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

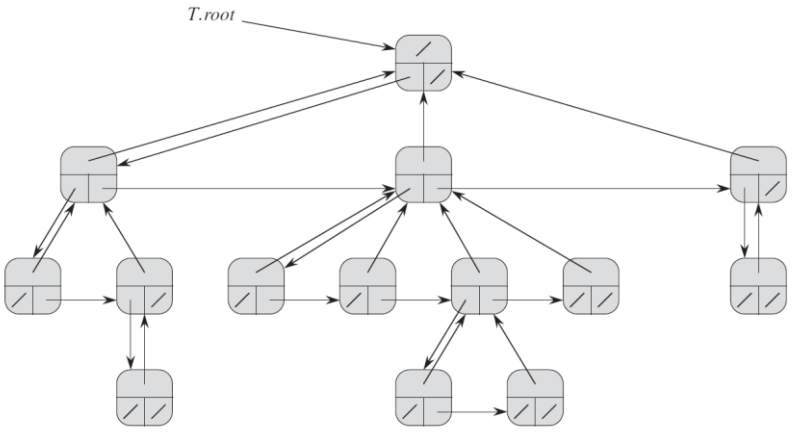
5 $\boldsymbol{return}\ x$

10.4 RAPPRESENTARE ALBERI RADICATI



10.4.1 Alberi binary

we use the attributes p, left, and right to store pointers to the parent, left child, and right child of each node in a binary tree T. if x.p = NIL, then x is the root.



10.4.2 Rooted trees with unbounded branching

We can extend the scheme for representing a binary tree to any class of trees in which the number of children of each node is at most some constant k: we replace the left and right attributes by $child_1$, $child_2$,…, $child_k$.

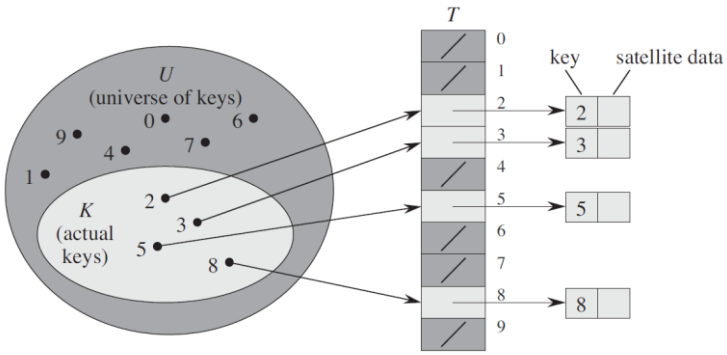
11 HASH TABLES

11.1 DIRECT-ADDRESS TABLES

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small. We shall assume that no two elements have the same key.

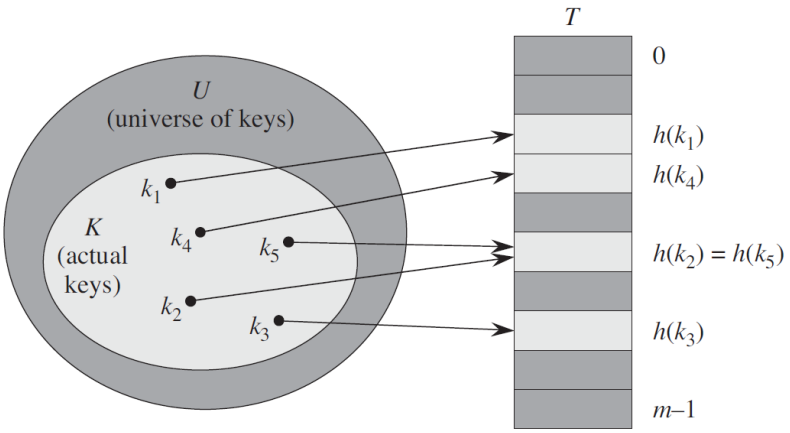
To represent the dynamic set, we use an array, or direct-address table, denoted by $T[0..m-1]$, in which each position, or slot, corresponds to a key in the universe U.

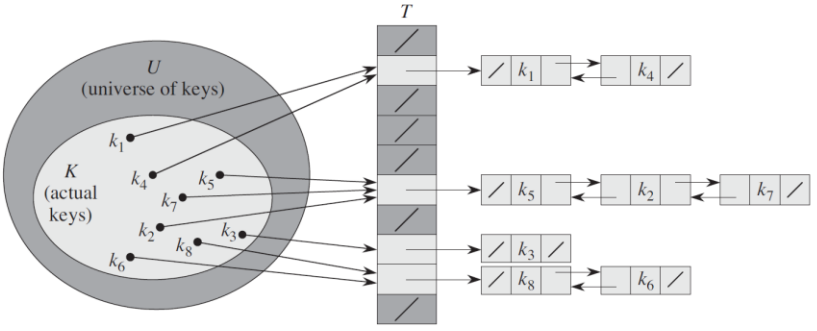
```
DIRECT-ADDRESS-SEARCH(T, k)
1    return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1    T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1    T[x.key] = NIL
```



Each of these operations takes only O(1) time.

11.2 HASH TABLES





With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot h(k); that is, we use a hash function h to compute the slot from the key k, we also say that h(k) is the hash value of key k.

There is one hitch: two keys may hash to the same slot. We call this situation a collision. Fortunately, we have effective techniques for resolving the conflict created by collisions.

TEOREMA

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\theta(1+\propto)$, under the assumption of simple uniform hashing.

TEOREMA

In a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\theta(1+\propto)$, under the assumption of simple uniform hashing.

APPENDICE

Algorithm	Time Complexity		
	Best	**Average**	**Worst**
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)