Appunti di Algoritmi e Strutture Dati

Sofia Amarù

5 settembre 2018

Indice

Note

Il seguente testo può contenere errori. Nel caso in cui ne trovaste, siete pregati di notificarlo per procedere alla correzione.

Sono presenti link che rimandano a gif esplicative dell'algoritmo in questione. Se i link dovessero venire rimossi e non fossero più funzionanti, potete comunuqe trovare le immagini in questa raccolta.

1 Introduzione

1.1 Cos'è un algoritmo

Un **algoritmo** è una procedura di calcolo ben definita che prende un valore (input) e lo trasforma generando un altro valore (output). La sequenza di input è detta istanza del problema.

Un algoritmo si dice **corretto** se per, ogni istanza di input, termina con l'output corretto. In questo caso l'algoritmo **risolve** il problema computazionale. Un algoritmo **errato** potrebbe fornire un risultato errato. Gli algoritmi errati però possono essere utili quando si riesce a mantenere sotto controllo la percentuale di errore (ad es. gli algoritmi per calcolare i numeri primi grandi). Un algoritmo può essere specificato in lingua italiana, come un programma per computer o come un progetto hardware; unico requisito è che la specifica fornisca una descrizione esatta della procedura computazionale da eseguire.

1.2 Alcune nozioni

Struttura dati Una struttura dati è un modo per memorizzare e organizzare i dati semplificandone accesso e modifica.

Problemi NP-completi Problemi per i quali non si sa se esistano algoritmi efficienti.

Parallelismo Per molti anni le velocità di clock dei processori sono state costantemente in crescita. Tuttavia la potenza aumenta in modo superlineare alla velocità di clock, per tale motivo i chip corrono il rischio di fondere una volta raggiunte velocità sufficientemente elevate. Dunque, per poter eseguire più operazioni al secondo, vengono progettati chip multicore (con più "nuclei" di elaborazione all'interno). Dobbiamo quindi progettare algoritmi tenendo conto del parallelismo.

Segnale di clock In elettronica il termine clock indica un segnale periodico, generalmente un'onda quadra, utilizzato per sincronizzare il funzionamento dei dispositivi elettronici digitali. La velocità o frequenza di clock è il numero di commutazioni tra i due livelli logici "0" e "1" che circuiti all'interno di un'unità di calcolo o di un microprocessore sono in grado di eseguire nell'unità di tempo di un secondo, ed è espressa in cicli al secondo, o hertz, e suoi multipli; normalmente per eseguire un'istruzione o una semplice somma sono necessari più cicli di clock.

Efficienza Un algoritmo deve essere efficienti in termini di tempo (velocità del computer) e spazio (la memoria ha un suo costo).

Problema dell'ordinamento Data una sequenza $< a_1, a_2, \ldots, a_n >$, l'output $< a'_1, a'_2, \ldots, a'_n >$ è un riarrangiamento dell'input tale che $a'_1 \leq a'_2 \leq \ldots \leq a'_n$. I numeri da ordinare sono anche detti **chiavi**.

Pseudocodice Linguaggio simile ad un linguaggio di programmazione ma che impiega qualsiasi mezzo espressivo (anche la lingua parlata) per specificare in modo chiaro ed esaustivo un algoritmo.

2 INSERTION-SORT (ordine delle carte)

Prende come parametro un array A[1, ..., n] contenente una sequenza di lunghezza n ($A.length$) che deve essere ordinata. È un **algoritmo in place**, cioè ordina l'array utilizzando soltanto un piccolo e costante spazio di memoria extra, risparmiando memoria; ogni numero viene confrontato con i precedenti e inserito nel posto più opportuno: si assume che la sequenza da ordinare sia partizionata in una sottosequenza già ordinata, all'inizio composta da un solo elemento, e una ancora da ordinare. Alla k-esima iterazione, la sequenza già ordinata contiene k elementi. In ogni iterazione, viene rimosso un elemento dalla sottosequenza non ordinata (scelto, in generale, arbitrariamente) e inserito (da cui il nome dell'algoritmo) nella posizione corretta della sottosequenza ordinata, estendendola così di un elemento.

Per fare questo, un'implementazione tipica dell'algoritmo utilizza due indici: uno punta all'elemento da ordinare e l'altro all'elemento immediatamente precedente. Se l'elemento puntato dal secondo indice è maggiore di quello a cui punta il primo indice, i due elementi vengono scambiati di posto; altrimenti il primo indice avanza. Il procedimento è ripetuto finché si trova nel punto in cui il valore del primo indice deve essere inserito. Il primo indice punta inizialmente al secondo elemento dell'array, il secondo inizia dal primo. L'algoritmo così tende a spostare man mano gli elementi maggiori verso destra.

INSERTION-SORT(A[ ])

```
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
```

Note:
j = indice numero corrente
i = j-1 indice numero precedente

Invariante di ciclo di INSERTION-SORT All'inizio di ogni iterazione del ciclo for (righe 1-8) il sottoarray $A[1..j-1]$ è ordinato e formato dagli stessi elementi che originariamente erano in $A[1..j-1]$.

Inizializzazione l'invariante è vera prima della prima iterazione del ciclo

Conservazione se l'invariante è vera prima di un'iterazione del ciclo, rimane vera prima della successiva iterazione

Conclusione l'invariante favorisce un'utile proprietà per vedere se l'algoritmo è corretto.

Convenzioni di pseudocodifica

- Identazione con **for**, **while**, **if-else** ecc;
- Useremo le parole **while**, **for**, **repeat-until**, **to**, **downto** e **by** (quando il contatore del ciclo varia di una quantità >1);
- // introduce un commento;
- Possiamo fare assegnazioni multiple del tipo i = j = e (j = e; i = j;);
- Variabili come i, j e key sono locali;
- A[1..j] indica il sottoarray composto dagli elementi A[1], A[2], . . . , A[j];
- I dati sono organizzati in oggetti, che hanno attributi. Indicheremo un oggetto x e il suo attributo in questo modo x.f. Ricordiamo che una variabile che rappresenta un oggetto è trattata come un puntatore ai dati che costituiscono l'oggetto; pertanto l'assegnazione x=y tra due oggetti implica che x e y puntano allo stesso oggetto.
- Un puntatore può non fare riferimento ad alcun oggetto; daremo ad esso il valore NIL;
- Un'istruzione **return** può passare più valori;
- Gli operatori and e or sono cortocircuitati;
- La parola chiave **error** indica che si è verificato un errore perché le condizioni non erano corrette per la procedura chiamata.

2.1 Analisi degli algoritmi

Analizzare un algoritmo significa prevedere le risorse che esso richiede (memoria, larghezza di banda nelle comunicazioni e hardware sono poco rilevanti; è importante il tempo di elaborazione). Considereremo come tecnologia di implementazione un modello di calcolo a un processore che chiameremo modello **RAM (random-access machine)**. Supporremo che ci sia un limite alla dimensione di ogni word di dati: ad esempio, quando l'input ha dimensione n, supponiamo che i numeri interi siano rappresentati da $c \lg n$ bit per una costante $c \geq 1$.

2.1.1 Analisi di `INSERTION-SORT`

Il tempo richiesto dalla procedura di `INSERTION-SORT` dipende dal numero e dal tipo di input. In generale, il tempo richiesto da un algoritmo cresce con la dimensione dell'input, quindi si scrive il tempo di esecuzione di un programma come una **funzione della dimensione del suo input**. La **dimensione dell'input** dipende dal problema che si sta studiando; per la maggior parte dei problemi esso è il numero di elementi dell'input (per l'ordinamento è la dimensione n dell'array, per la moltiplicazione di due interi è il numero totale di bit richiesti per rappresentare l'input in notazione binaria). Il **tempo di esecuzione** è il **numero di operazioni primitive (passi)** che vengono eseguite.

Per eseguire una riga di pseudocodice occorre una quantità costante di tempo. Una riga può richiedere una quantità di tempo diversa da un'altra riga, ma supporremo che ogni esecuzione della i-esima riga richieda un tempo c_i, dove c_i è una costante.

righe di codice	costo*	numero di volte**
1	c_1	n
2	c_2	$n-1$
3	0	$n-1$
4	c_4	$n-1$
5	c_5	$\sum_{j=2}^{n} t_j$
6	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	c_8	$n-1$

* tempo impiegato da ogni istruzione
** numero di volte che vengono eseguite le singole istruzioni
$n = A.length$
t_j è il numero di volte che il test del **while** nella riga 5 viene eseguito per il valore di j.

Quando un **for** o un **while** termina nel modo corretto, il test viene eseguito una volta in più rispetto al corpo del ciclo. Il costo dei commenti è nullo.

Il **tempo di esecuzione dell'algoritmo** è la somma dei tempi di esecuzione per ogni istruzione eseguita. Un'istruzione che richiede c_i passi e viene eseguita n volte contribuirà al tempo di esecuzione totale con $c_i n$.

Per l'`INSERTION-SORT` avremo:

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

Caso migliore Il caso migliore si presenta quando l'array è già ordinato (righe 6 e 7 nulle):

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Questo tempo può essere espresso come $an + b$ (a e b dipendono dai costi c_i), quindi è una funzione lineare di n.

Caso peggiore Si presenta quando l'array è ordinato in senso inverso (in ordine decrescente): Il while viene eseguito ogni volta (j volte): $t_j = j$;

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

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

Quindi:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2})$$

$$+ c_8(n-1)$$

$$= (\frac{c_5 + c_6 + c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5 - c_6 - c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Questo tempo può essere espresso come $an^2 + bn + c$ (a, b e c dipendono dai costi c_i), quindi è una funzione quadratica di n.

Tasso di crescita Abbiamo espresso il caso peggiore con $an^2 + bn + c$, ignorando i costi effettivi delle istruzioni e quelli astratti c_i. Possiamo fare un'altra astrazione semplificativa: a noi interessa la velocità con cui cresce il tempo di esecuzione, ovvero il suo **tasso di crescita**. Di conseguenza, di una formula prendiamo solo il suo termine principale (nel caso peggiore è an^2), e ignoriamo anche il suo coefficiente (n^2). **Un algoritmo è considerato più efficiente di un altro se il suo tasso di crescita è inferiore**.

3 Divide et impera

Molti algoritmi sono **ricorsivi**; generalmente questi adottano un approccio **divide et impera**: il problema viene suddiviso in sottoproblemi più piccoli, vengono risolti i sottoproblemi e combinate le varie soluzioni per costruire una soluzione del problema originale. Il divide et impera prevede tre passi:

Divide il problema viene diviso in sottoproblemi;

Impera i sottoproblemi vengono risolti in modo ricorsivo (se sono di dimensione sufficientemente piccola vengono risolti direttamente);

Combina le soluzioni vengono combinate.

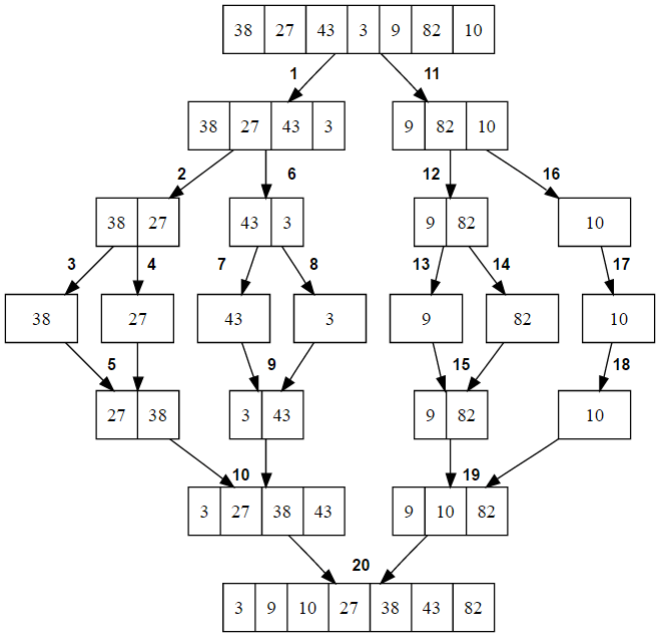
Questo tipo di approccio è tipicamente detto **top-down**. Il nome top down significa dall'alto verso il basso: in "alto" viene posto il problema e in "basso" i sottoproblemi che lo compongono. Al contrario, il **bottom-up** richiama un'immagine raffigurante una freccia in cui la coda è il bottom (la parte bassa) mentre up è la punta: dal punto di vista dinamico si parte dal bottom e si procede verso up. Il bottom up prende corpo dal punto di partenza (bottom) ovvero dalla situazione iniziale; considera l'obiettivo finale, induce a costruire un percorso sequenziale organizzato in passaggi successivi in cui l'ancoraggio tra traguardi intermedi e obiettivo finale è generalmente ricercato in modo intuitivo.

4 MERGE-SORT

Divide divide la sequenza di n elementi in due sottosequenze di $\frac{n}{2}$ elementi ciascuna;

Impera ordina le due sottosequenze in modo ricorsivo usando l'algoritmo merge-sort.

Combina fonde le sue sottosequenze ordinate per generare la sequenza ordinata.



La ricorsione "tocca il fondo" quando la sottosequenza ha lunghezza 1: ogni sequenza di lunghezza 1 è già ordinata. Fondamentale è l'operazione "combina", che fonde due sottosequenze già ordinate: utilizziamo la procedura `MERGE`(A, p, q, r) dove A è un array e p, q e r sono indici tali che $p \leq q \leq r$. La procedura assume che i sottoarray $A[p..q]$ e $A[q+1..r]$ siano già ordinati e li fonde dando vita a $A[p..r]$. Ogni sottoarray è ordinato, con i valori più piccoli all'inizio. Scegliamo il valore più piccolo tra i due valori che stanno all'inizio dei due array, e lo collochiamo nell'array di output. Ripetiamo questo passo fino a quando uno dei due sottoarray non sarà vuoto. Collochiamo dunque tutti i numeri rimasti nell'altro sottoarray nell'array di output. Al massimo svolgiamo n passi base, quindi la fusione dei mazzi impiega un tempo $\Theta(n)$.

4.1 Procedura `MERGE`

`MERGE`(A, p, q, r)

```
1  n1 = q-p+1
2  n2 = r-q
3  crea due nuovi array L[1..n1+1] e R[1..2+1]
4  for i = 1 to n1
5      L[i] = A[p+i-1]
6  for j=1 to n2
7      R[j] =  A[q+j]
8  L[n1+1] = ∞
9  R[n2+1] = ∞
10 i=1
11 j=1
12 for k = p to r
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
```

Ovvero:

```
1  calcola la lunghezza del sottoarray A[p..q]
2  calcola la lunghezza del sottoarray A[q+1..r]
3  creiamo gli array R e L (lunghezza n 1 +1 e n 2 +1)
4
5  copia il sottoarray A[p..q] in L
6
7  copia il sottoarray A[q+1..r] in R
8  pone il valore sentinella
9  pone il valore sentinella
10
11
12 combina
```

In fondo ai sottoarray R e L mettiamo un valore sentinella (∞): quando incontriamo questo valore in un array, esso non può essere più piccolo di quello del secondo array. Inoltre sappiamo che quando arriviamo alla sentinella, tutti i valori precedenti nell'array sono stati posti nell'array di output.

Invariante di ciclo a ogni iterazione del ciclo for (righe 12-17), il sottoarray $A[p..k]$ contiene ordinati i $k-p$ elementi più piccoli di L e R. $L[i]$ e $R[j]$ sono i più piccoli elementi dei propri array che non sono stati copiati in A.

4.1.1 Analisi di `MERGE`

righe di codice	tempo
1	tempo costante
2	tempo costante
3	tempo costante
4	n_1
5	
6	n_2
7	
8	tempo costante
9	tempo costante
10	tempo costante
11	tempo costante
12	n
13	
14	
15	
16	
17	

4.2 `MERGE-SORT`

Ordina gli elementi nel sottoarray $A[p..r]$. Se $p \geq r$, il sottoarray ha massimo un elemento (è già ordinato), altrimenti il passo "divide" calcola un indice q che separa $A[p..r]$ in due sottoarray:

`MERGE-SORT`(A, p, r)

```
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)
```

4.3 Analisi degli algoritmi divide et impera

Quando un algoritmo contiene una chiamata ricorsiva a se stesso, il suo tempo di esecuzione può essere descritto con un'**equazione di ricorrenza** (o **ricorrenza**). Essa esprime il tempo di esecuzione totale di un problema di dimensione n in funzione del tempo di esecuzione per input più piccoli. Una ricorrenza per il tempo di esecuzione di un algoritmo divide et impera si basa sui tre passi del paradigma di base. Supponiamo che n sia la dimensione del problema; $T(n)$ sarà il tempo di esecuzione. Se la dimensione del problema è sufficientemente piccola, ad esempio $n \leq c$, la soluzione richiede

un tempo costante $\Theta(1)$. Supponiamo che la suddivisione del problema generi a sottoproblemi e che la dimensione di ogni sottoproblema sia $\frac{1}{b}$ volte la dimensione dell'originale: serve un tempo $T(\frac{n}{b})$ e per an un tempo $aT(\frac{n}{b})$:

$$T(n) = \begin{cases} \Theta(1) & se\ n \leq c \\ aT(\frac{n}{b}) + D(n)^* + C(n)^{**} & negli\ altri\ casi \end{cases}$$

* tempo che serve per dividere i problemi in sottoproblemi
** tempo che serve per combinare le soluzioni dei sottoproblemi

4.4 Analisi di `MERGE-SORT`

Divide calcola il centro del sottoarray, richiede un tempo costante, $D(n) = \Theta(1)$;

Impera risolviamo in modo ricorsivo i due problemi, ognuno di dimensione $\frac{n}{2}$, quindi $2T(\frac{n}{2})$;

Combina la procedura `MERGE` con un sottoarray di n elementi richiede un tempo $\Theta(n)$.

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

Teorema dell'esperto $T(n)$ è $\Theta(n \lg n)$, dove lg sta per $\log_2 n$

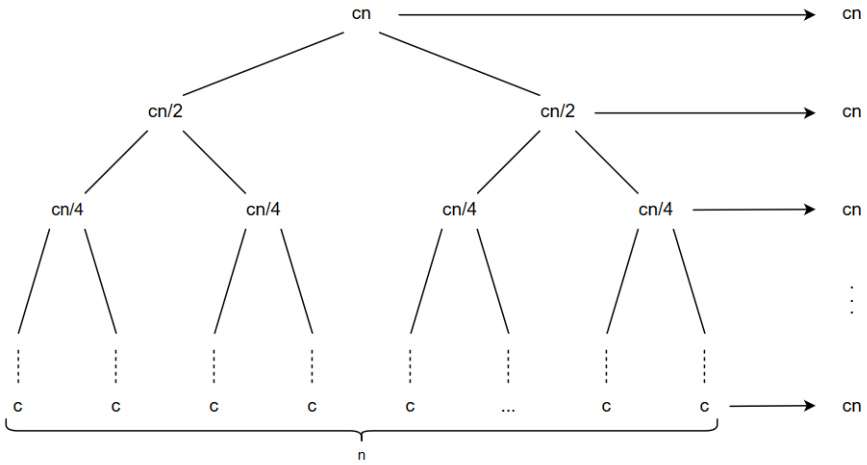
`MERGE-SORT` quindi supera le prestazioni di `INSERTION-SORT`, il cui tempo di esecuzione è $\Theta(n^2)$, nel caso peggiore.

Possiamo riscrivere la ricorrenza in questo modo:

$$T(n) = \begin{cases} c & se\ n = 1 \\ 2T(\frac{n}{2}) + cn & sen > 1 \end{cases}$$

c rappresenta il tempo richiesto per risolvere i problemi di dimensione 1, ma anche il tempo dei passi divide e combina per ogni elemento.

4.5 Albero di ricorsione



Sommiamo i costi per ogni livello dell'albero: il livello i ha 2^i nodi, ciascuno dei quali ha un costo $c(\frac{n}{2^i})$, ogni livello ha dunque un costo totale di cn. Il numero totale di livelli è $\lg n + 1$, dove n è il numero delle foglie (uguale alla dimensione dell'input). Il costo totale è di $cn(\lg n + 1)$, ovvero $\Theta(n \lg n)$.

5 BUBBLESORT

È un noto, ma inefficiente, algoritmo di ordinamento che opera scambiando ripetutamente gli elementi adiacenti che non sono ordinati.

Bubblesort()

```
1  for i = 1 to A.length-1
2    for j = A.length downto i + 1
3          if A[j] < A[j-1]
4                  scambia A[j] con A[j-1]
```

6 Efficienza asintotica

Quando operiamo con dimensioni di input abbastanza grandi da rendere rilevante soltanto il tasso di crescita del tempo di esecuzione, studiamo l'**efficienza asintotica** degli algoritmi: ci interessa sapere come aumenta il tempo di esecuzione di un algoritmo al crescere della dimensione dell'input al limite. Di solito, un algoritmo che è asintoticamente più efficiente sarà il migliore con tutti gli input, ad eccezione di quelli molto piccoli.

6.1 Notazione asintotica, funzioni e tempi di esecuzione

La notazione asintotica si applica alle funzioni. Abbiamo definito il tempo di esecuzione di `INSERTION-SORT` nel caso peggiore come $\Theta(n^2)$, riferendoci all'equazione $an^2 + bn + c$, con a, b e c costanti. Scrivendo che il tempo di esecuzione $\Theta(n^2)$ abbiamo trascurato alcuni dettagli.

Una notazione asintotica può anche essere applicata a funzioni che caratterizzano qualche altro aspetto degli algoritmi (quantità di spazio utilizzato) o funzioni che non hanno nulla a che fare con gli algoritmi.

Quando usiamo la notazione asintotica per applicarla al tempo di esecuzione dobbiamo sapere a quale tempo di esecuzione ci riferiamo. A volte siamo interessati al caso peggiore, a volte vogliamo caratterizzare il tempo indipendentemente da un input.

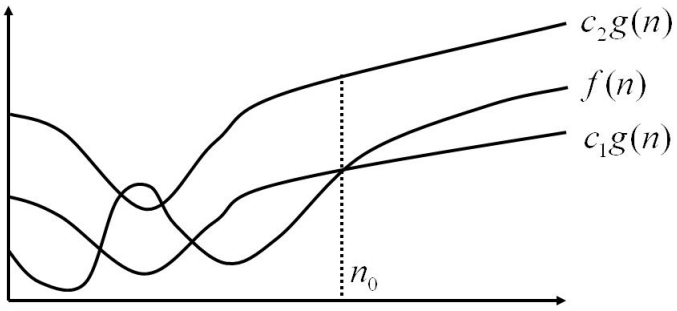
6.1.1 Notazione Θ

Per una funzione $g(n)$, indichiamo con $\Theta(g(n))$ l'insieme delle funzioni

$$\Theta(g(n)) = \{f(n) : \exists \textit{ delle costanti positive } c_1, c_2 \textit{ e } n_0 \textit{ t.c.}$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0\}$$

Una funzione $f(n)$ appartiene all'insieme $\Theta(g(n))$ se esistono delle costanti positive c_1 e c_2 tali che $f(n)$ possa essere racchiusa fra $c_1 g(n)$ e $c_2 g(n)$. Si dice che $g(n)$ è un **limite asintoticamente stretto** per $f(n)$. Ogni membro di $f(n)$ deve essere **asintoticamente non negativo**, ovvero $f(n)$ deve essere non negativa quando n è sufficientemente grande (una **funzione asintoticamente positiva** è positiva per qualsiasi valore sufficientemente grande di n). Di conseguenza anche $g(n)$ deve essere asintoticamente non negativa.

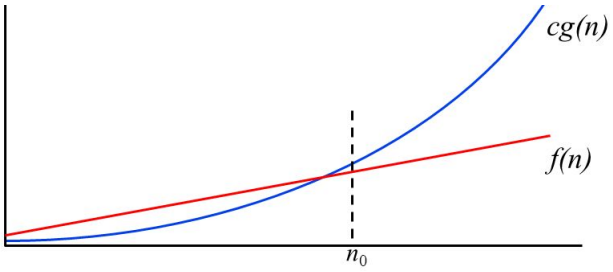


Informalmente, per trovare una notazione Θ, possiamo escludere i termini di ordine inferiore e ignorare il coefficiente del termine di ordine più elevato.

6.1.2 Notazione O

$$O(g(n)) = \{f(n) : \exists \textit{ delle costanti positive } c \textit{ e } n_0 \textit{ t.c.}$$
$$0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

La notazione Θ limita asintoticamente una funzione da sopra e da sotto. Quando abbiamo soltanto un **limite asintotico superiore**, utilizziamo una notazione O. Per qualsiasi valore n a destra di n_0, il valore di $f(n)$ coincide o sta sotto $cg(n)$.



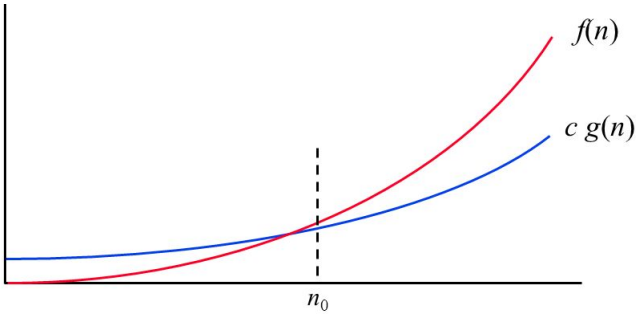
Si dice che $f(n)$ è O di $cg(n)$.

6.1.3 Notazione Ω

Fornisce un **limite asintotico inferiore**.

$$\Omega(g(n)) = \{f(n) : \exists \textit{ delle costanti positive } c \textit{ e } n_0 \textit{ t.c.}$$
$$0 \leq cg(n) \leq f(n) \forall n \geq n_0\}$$

Per qualsiasi valore n a destra di n_0 , il valore di $f(n)$ coincide o sta sopra $cg(n)$.



Si dice che $f(n)$ è Ω di $cg(n)$.

Teorema Per ogni coppia di fuzioni $f(n)$ e $g(n)$, si ha $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))ef(n) = \Omega(g(n))$.

6.1.4 Notazione asintotica nelle equazioni e nelle disequazioni

Quando una notazione asintotica appare in una formula, essa va trattata come se indicasse una funzione anonima, di cui non è importante fare il nome. In questo modo è possibilie eliminare i dettagli superflui e ingombranti da un'equazione. Il numero di funzioni anonime in un'espressione è sottinteso che sia uguale al numero di volte che appare la notazione asintotica; per esempio, in $\sum_{i=1}^{n} O(i)$ c'è una sola funzione anonima (una funzione di i). Questa espressione quindi è diversa da $O(1) + O(2) + ... + O(n)$. In alcuni casi la notazione asintotica si trova sul lato sinistro dell'equazione, ad esempio in $2n^2 + \Theta(n) = \Theta(n^2)$. Per qualsiasi funzione $f(n) \in \Theta(n)$, c'è qualche funzione $g(n) \in \Theta(n^2)$ t.c. $2n^2 + f(n) = g(n)$ per ogni n. Il lato destro di un'equazione fornisce un livello di dettaglio più grossolano del lato sinistro.

6.1.5 Notazione o

Il limite asintotico superiore (=) può essere stretto oppure no. $2n^2 = O(n^2)$ è asintoticamente stretto $2n = O(n^2)$ non è asintoticamente stretto. Questo lo definiamo con o. In O, il limite vale per qualche costante $c > 0$; in o, il limite vale per tutte le costanti $c > 0$.

6.1.6 Notazione ω

Analogamente, ω sta a Ω come o sta a O. Usiamo la notazione ω per indicare un limite non asintoticamente stretto.

6.1.7 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	$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$ $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$
Tricotomia	Se a e b sono due numeri reali qualsiasi deve valare solo una delle seguenti relazioni: $a < b$, $a = b$, $a > b$

6.1.8 Funzioni monotòne

monotona crescente	$m \leq n \Rightarrow f(m) \leq f(n)$
monotona decrescente	$m \leq n \Rightarrow f(m) \geq f(n)$
strettamente crescente	$m < n \Rightarrow f(m) < f(n)$
strettamente decrescente	$m < n \Rightarrow f(m) > f(n)$

6.2 Altro

6.2.1 Floor e ceiling

Floor	$\lfloor ... \rfloor$	difetto
Ceiling	$\lceil ... \rceil$	eccesso

7 Ricorrenze

Una ricorrenza è un'equazione o disequazione che descrive una funzione in termini del suo valore con input più piccoli. Abbiamo diversi metodi per risolvere le ricorrenze (cioè per ottenere dei limiti asintotici "Θ" o "O"):

- **Metodo della sostituzione**: ipotizziamo un limite e usiamo l'induzione matematica per dimostrare che la nostra ipotesi è corretta.
- **Metodo dell'albero della ricorsione**: converte la ricorrenza in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione.
- **Metodo dell'esperto**: fornisce i limiti per ricorrenze nella forma $T(n) = aT(\frac{n}{b}) + f(n)$ dove $a \geq 1$, $b > 1$ e $f(n)$ è una funzione data. Una ricorrenza come questa caratterizza un algoritmo che crea a sottoproblemi, ciascuno di dimensione $\frac{1}{b}$ e in cui i passi divide e combina insieme richiedono un tempo $f(n)$.

7.1 Dettagli tecnici

Quando definiamo e risolviamo le ricorrenze, trascuriamo alcuni dettagli tecnici. Ad esempio, se chiamiamo `MERGE-SORT` su n elementi con n dispari, avremo sottoproblemi di dimensione $\lfloor \frac{n}{2} \rfloor$ e $\lceil \frac{n}{2} \rceil$. La ricorrenza è effettivamente

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & sen > 1 \end{cases}$$

Solitamente però omettiamo le condizioni al contorno e le operazioni di floor e ceiling: a noi interessa infatti il tasso di crescita, che rimane immutato e non dipende da fattori costanti.

7.2 Il metodo di sostituzione per risolvere le ricorrenze

- Ipotizzare la forma della soluzione
- Usare l'induzione matematica per trovare le costanti e dimostrare che la soluzione funziona

Questo metodo può essere utilizzato per determinare il limite superiore o inferiore di una ricorrenza. Ad esempio determiniamo il limite superiore per la ricorrenza $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$. Supponiamo che la soluzione sia $T(n) = O(n \lg n)$. Il metodo consiste nel dimostrare che $T(n) \leq cn \lg n$ per una scelta appropriata della costante $c > 0$. Supponiamo che questo limite sia valido per $\lfloor \frac{n}{2} \rfloor$, ovvero che $T(\lfloor \frac{n}{2} \rfloor) \leq c\lfloor \frac{n}{2} \rfloor lg(\lfloor \frac{n}{2} \rfloor)$.

Avremo:

$$T(n) \leq 2(c\lfloor \frac{n}{2} \rfloor \lg(\lfloor \frac{n}{2} \rfloor)) + n$$

$$\leq cn \lg(\frac{n}{2}) + n$$
$$= cn \lg n - cn \lg 2 + n$$
$$= cn \lg n - cn + n$$
$$\leq cn \lg n$$

L'ultimo passo è vero quando $c \geq 1$. 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 richiededi 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 $c \geq 2$.

7.2.1 Formulare una buona ipotesi

Se una ricorrenza è simile a una già vista, ha senso provare una soluzione analoga. Ad esempio, prendiamo $T(n) = 2T(\lfloor \frac{n}{2} \rfloor + 17) + n$. Il termine aggiuntivo 17 non influisce in modo sostanziale alla soluzione della ricorrenza. Quando n è grande, la differenza tra $T(n) = 2T(\lfloor \frac{n}{2} \rfloor + 17) + n$ e $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$ 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.

7.2.2 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.

7.2.3 Sostituzione di variabili

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

Poniamo $m = \lg n$

$$T(2^m) = 2T(2^{\frac{m}{2}}) + m$$

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

$$S(m) = 2S(\frac{m}{2}) + m$$

che ha soluzione

$$S(m) = O(m \lg m)$$

Ripristinando T, otteniamo

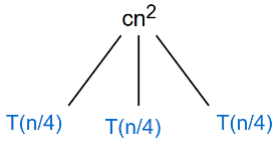
$$T(n) = O(m \lg m) = O(\lg n \lg(\lg n))$$

7.3 Il metodo dell'albero di ricorsione per risolvere le ricorrenze

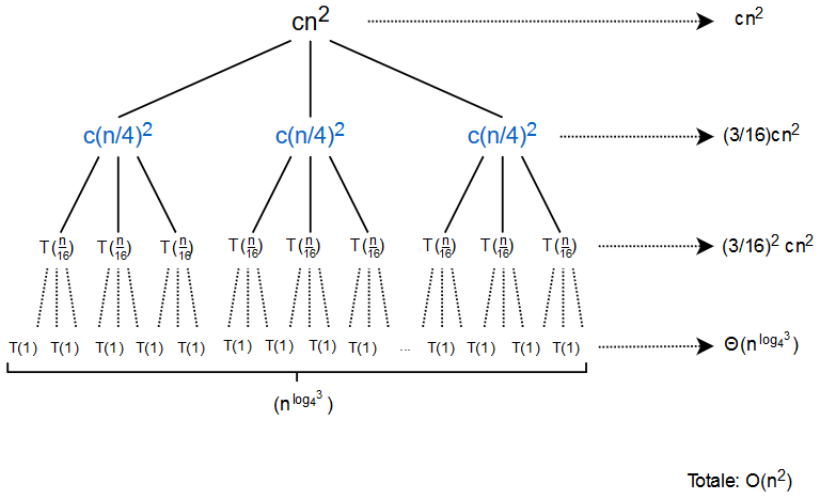
In un albero di ricorsione ogni nodo rappresenta il costo di un singolo sottoproblema. Sommiamo i costi all'interno di ogni livello per avere un insieme di costi per livello, poi sommiamo tutti i costi per livello per ottenere il costo totale. Un albero è un ottimo metodo per avere un'ipotesi, che poi verrà verificata con il metodo di sostituzione (ma possiamo anche usarlo come prova diretta di una soluzione di ricorrenza).

7.3.1 Esempio

Sia data la ricorrenza $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$ 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 $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + cn^2$, 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 $\frac{n}{4}^i$. La dimensione del sottoproblema diventa quindi $n = 1$ quando $\frac{n}{4}^i = 1$ 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 $c(\frac{n}{4}^i)^2$. Il costo totale di tutti i nodi alla profondità i è $(\frac{3}{16})^i cn^2$. 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 $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + \Theta(n^2)$. Dobbiamo dimostrare che $T(n) \leq dn^2$ per qualche costante $d > 0$. Otteniamo:

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

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

7.4 Il metodo dell'esperto per risolvere le ricorrenze

Si usa per risolvere ricorrenze della forma $T(n) = aT(\frac{n}{b}) + f(n)$, 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 $\frac{n}{b}$. I sottoproblemi vengono risolti in modo ricorsivo, ciascuno nel tempo $T(\frac{n}{b})$.

Teorema date le costanti $a \geq 1$ e $b > 1$ e la sua funzione $f(n)$, sia $T(n)$ una funzione definita sufli interi non negativi della ricorrenza $T(n) = aT(\frac{n}{b}) + f(n)$ dove $\frac{n}{b}$ rappresenta $\lfloor \frac{n}{b} \rfloor$ o $\lceil \frac{n}{b} \rceil$. Allora $T(n)$ può essere asintoticamente limitata nei seguenti modi:

1. se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche costante $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$
2. se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \lg n)$
3. se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche costante $\epsilon > 0$ e se $af(\frac{n}{b}) \leq cf(n)$ 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 $\epsilon > 0$. Nel terzo caso $f(n)$ deve essere polinomialmente più grande grande di $n^{\log_b a}$, e soddisfare le condizioni di regolarità $af(\frac{n}{b}) \leq cf(n)$.

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

7.4.1 Applicazione del metodo dell'esperto

Determiniamo quale caso del teorema dell'esperto possiamo applicare (se esiste) e scriviamo la soluzione

8 Heap

Un heap (binario) è una struttura dati composta da un array che possiamo considerare come un albero binario quasi completo. Ogni nodo è un elemento dell'array. Tutti i livelli sono riempiti, tranne eventualmente l'ultimo che da sinistra è riempito fino ad un certo punto. Anche se ci sono numeri memorizzati in tutto l'array, solo quelli $A[1..A.heap - size]$ sono validi nell'heap. La radice dell'albero è $A[1]$.

Se i è l'indice di un nodo possiamo calcolare gli indici di:

- `PARENT`(i) padre

  ```
  1 return ⌊i/2⌋
  ```

  Può calcolare $\lfloor i/2 \rfloor$ con uno scorrimento di una posizione a destra della rappresentazione di i.

- `LEFT`(i) figlio sx

  ```
  1 return 2 i
  ```

  La procedura può calcolare 2i con una sola istruzione, facendo scorrere di una posizione a sinistra la rappresentazione binaria di i.

- `RIGHT`(i) figlio dx

  ```
  1 return 2 i +1
  ```

  La procedura può calcolare 2i+1 facendo scorrere di una posizione a sinistra la rappresentazione binaria di i e aggiungendo 1 come bit meno significativo.

altezza di un nodo numero di archi nel cammino semplice più lungo che dal nodo scende fino a una foglia.

altezza di un heap altezza della radice. Poiché un heap di n elementi è basato su un albero binario completo, la sua altezza è $\Theta(\lg n)$.

8.1 Max-heap e Min-heap

Gli heap binari si dividono in

max-heap per ogni nodo diverso dalla radice si ha $A[\texttt{PARENT}(i) \geq A[i]$. Il valore di un nodo è al massimo quello di suo padre. L'elemento più grande è memorizzato nella radice.

min-heap per ogni nodo diverso dalla radice si ha $A[\texttt{PARENT}(i) \leq A[i]$. Il più piccolo elemento è nella radice.

8.2 Conservare la proprietà dell'heap

Useremo la procedura `MAX-HEAPIFY` . I suoi input sono un array A e un indice i. Quando viene chiamata, assume che gli alberi binari con radici in `LEFT`(i) e `RIGHT`(i) siano max-heap, ma che $A[i]$ possa essere più piccolo dei suoi figli (violando la proprietà del max-heap). La procedura fa scendere il valore $A[i]$ nel max-heap in modo che il sottoalbero con radice di indice i diventi un max-heap: a ogni passo, tra gli elementi $A[i]$, $A[$`LEFT`$(i)]$ e $A[$`RIGHT`$(i)]$ viene determinato il più grande. Il suo indice viene memorizzato in *massimo*. Se $A[i]$ è più grande, il sottoalbero con radice i è un max-heap e la procedura termina. Altrimenti, uno dei due figli ha l'elemento più grande e viene scambiato $A[i]$ con $A[massimo]$; il sottoalbero però potrebbe violare la proprietà del max-heap; viene chiamata ricorsivamente la subroutine `MAX-HEAPIFY`.

`MAX-HEAPIFY`(A, i)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4        massimo = l
5  else massimo = i
6  if r ≤ A.heap-size and A[r] > A[massimo]
7        massimo = r
8  if massimo ≠ i
9        scambia A[i] con A[]
10       MAX-HEAPIFY(A, massimo)
```

Il tempo di esecuzione di `MAX-HEAPIFY` in un sottoalbero di dimensione n con radice in un nodo i è pari al tempo $\Theta(1)$ per sistemare le relazioni fra gli elementi $A[i]$, $A[$`LEFT`$(i)]$ e $A[$`RIGHT`$(i)]$, più il tempo per eseguire MAX-HEAPIFY in un sottoalbero con radice in uno dei figli del nodo i. La dimensione dei sottoalberi dei figli non supera $2\frac{n}{3}$. Il tempo di esecuzione può essere descritto dalla ricorrenza $T(n) \leq T(2\frac{n}{3}) + \Theta(1)$. La soluzione è $T(n) = O(\lg n)$, oppure $O(h)$ (nodo di altezza h).

8.3 Costruire un heap

Possiamo utilizzare la procedura `MAX-HEAPIFY` dal basso verso l'alto per convertire un array in un max-heap. La procedura `BUILD-MAX-HEAP` attraversa i restanti nodi dell'albero ed esegue `MAX-HEAPIFY` in ciascuno di essi.

`BUILD-MAX-HEAP`(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3  MAX-HEAPIFY(A, i)
```

Invariante di ciclo all'inizio di ogni ciclo for, ogni nodo i+1, i+2, ..., n è la radice di un heap

Possiamo calcolare un semplice limite superiore sul tempo di esecuzione di `BUILD-MAX-HEAP` nel seguente modo: ogni chiamata di `MAX-HEAPIFY` costa un tempo $O(\lg n)$ e ci sono $O(n)$ di queste chiamate, per un totale di $O(n \lg n)$. Questo limite però non è asintoticamente stretto. Osserviamo che il tempo per eseguire `MAX-HEAPIFY` in un nodo varia con l'altezza del nodo dell'albero. Un heap di n elementi ha un'altezza $\lfloor \lg n \rfloor$ e per ogni h, al massimo $\lceil \frac{n}{2}^{h+1} \rceil$ nodi di altezza h. Il tempo richiesto dalla procedura quando viene chiamata per un nodo di altezza h è $O(h)$, quindi il costo totale di `BUILD-MAX-HEAP` è limitato superiormente da

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

Possiamo costruire un min-heap utilizzando la procedura `BUILD-MIN-HEAP`, che è uguale a `BUILD-MAX- HEAP`, ma la chiamata di `MAX-HEAPIFY` alla riga 3 è sostituita dalla chiamata di `MIN-HEAPIFY`.

8.4 L'algoritmo HEAPSORT

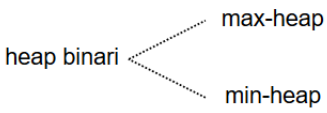
Inizia costruendo con `BUILD-MAX-HEAP` un max-heap nell'array di input $A[1..n]$ ($n = A.length$). L'elemento più grande è memorizzato nella radice $A[1]$, può essere scambiato con A[n] per andare nella sua posizione finale corretta. "Togliamo" il nodo n dall'heap: $A[n-1]$ può essere facilmente trasformato in un max-heap. I figli della radice sono un max-heap, ma la nuova radice potrebbe violare la proprietà del max-heap. Per ripristinare la proprietà chiamiamo di `MAX-HEAPIFY`$(A, 1)$. Esso lascia un max-heap in $A[1..(n-1)]$. L'algoritmo viene ripetuto per il max-heap di dimensione $n-1$ e così via fino a un heap di dimensione 2.

`HEAPSORT`(A)

```
1 BUILD-MAX-HEAP(A)
2 for i = A.length downto 2
3 scambia A[1] con A[i]
4 A.heap-size = A.heap-size -1
5 MAX-HEAPIFY(A, 1)
```

8.5 Code di priorità

Una buona implementazione di quicksort solitamente batte heapsort. Tuttavia la struttura dati heap ha molteplici usi, ad esempio le **code di priorità**. Una coda di priorità è una struttura dati che serve a mantenere un insieme S di elementi ognuno dei quali ha un valore associato detto **chiave**.



Una coda di max-priorità supporta le operazioni:

- `INSERT`(S, x) - Inserisce x nell'insieme S
- `MAXIMUM`(S) - Restituisce l'elemento di S con la chiave più grande
- `EXTRACT-MAX`(S) - Rimuove e restituisce l'elemento di S con la chiave più grande
- `INCREASE-KEY`(S, x, k) - Aumenta il valore della chiave dell'elemento x al nuovo valore k, che si suppone sia almeno grande quantoil valore corrente della chiave dell'elemento x

La coda di max-priority tiene traccia dei lavori da svolgere e delle loro relative proprietà. Quando un lavoro è ultimato o interrotto, lo scheduler seleziona il lavoro con priorità più alta fra quelli in attesa, chiamando `EXTRACT-MAX`. In ogni momento si può inserire un lavoro tramite INSERT . Una coda di min-priority supporta le operazioni `INSERT`, `MAXIMUM`, `EXTRACT-MIN`, `DECREASE-KEY`.

Quando un heap viene utilizzato per implementare una coda di priorità, spesso occorre memorizzare in ogni elemento dell'heap un **handle** (aggancio) al corrispondente oggetto dell'applicazione. Tipicamente l'handle è un indice dell'array.

- `HEAP-MAXIMUM`(A)

```
1  return A[1]
```

- `HEAP-EXTRACT-MAX`(A)

```
1  if A.heap-size < 1
2       error "underflow dell'heap"
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
```

Il tempo di esecuzione è $O(\lg n)$.

- `HEAP-INCREASE-KEY`(A, i, key)

```
1  if key < A[i]
2       error "nuova chiave piu' piccola di quella corrente"
3  A[i] = key
```

```
4  while i > 1 and A[PARENT(i)] < A[i]
5        scambia A[i] con A[PARENT(i)]
6        i = PARENT(i)
```

Il tempo di esecuzione con un heap di n elementi è $O(\lg n)$.

- `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)
```

Il tempo di esecuzione con un heap di n elementi è $O(\lg n)$.

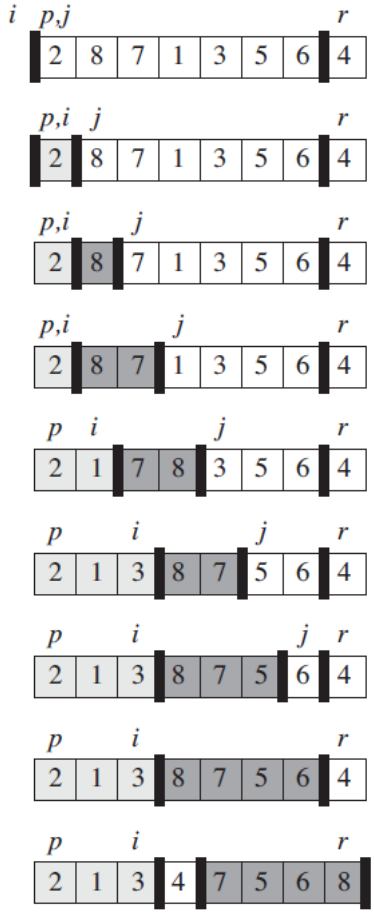
9 QUICKSORT

Quicksort è un algoritmo di ordinamento ricorsivo in place non stabile. Appartiene alla classe degli algoritmi divide et impera, dal momento che scompone ricorsivamente i dati da processare in sottoprocessi.

divide partiziona l'array $A[p..r]$ in due sottoarray $A[p..q-1]$ e $A[q+1..r]$ tali che ogni elemento di $A[p..q-1]$ sia minore o uguale ad $A[q]$ che, a sua volta è minore o uguale a ogni elemento di $A[q+1..r]$

impera ordina i due sottoarray chiamando ricorsivamente quicksort

combina poiché i sottoarray sono già ordinati, non occorre alcun lavoro per combinarli



Tale procedura ricorsiva viene comunemente detta `PARTITION`: preso un elemento chiamato **pivot** da una struttura dati (es. array) si pongono gli

elementi minori a sinistra rispetto al pivot e gli elementi maggiori a destra. L'operazione viene quindi reiterata sui due insiemi risultanti fino al completo ordinamento della struttura. Il `QUICKSORT` è l'algoritmo di ordinamento che ha, nel caso medio, prestazioni migliori tra quelli basati su confronto.

`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)
```

9.1 Partizionare l'array

L'elemento chiave è la procedura `PARTITION`, che riarrangia il sottoarray $A[p..r]$ sul posto. `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            scambia A[i] con A[j]
7  scambia A[i+1] con A[r]
8  return i+1
```

Ovvero:
seleziona un elemento come pivot intorno al quale partizionare l'array
All'inizio di ogni iterazione del ciclo for, per qualsiasi indice k dell'array:
1. se $p \leq k \leq i$, allora $A[k] \leq x$
2. se $i + 1 \leq k \leq j - 1$, allora $A[k] > x$
3. se $k = r$, allora $A[k] = x$

Inserisce il pivot al suo posto in mezzo all'array, scambiandolo con l'elemento più a sx che è $> x$ e restituisce il nuovo indice del pivot.

Invariante di ciclo

inizializzazione prima della prima iterazione del ciclo, $i = p - 1$ e $j = p$.
conservazione quando $A[j] > x$, viene incrementata j. Quando $A[j] \leq x$, viene incrementato i e scambiati $A[i]$ e $A[j]$, poi viene incrementato j.
conclusione alla fine del ciclo, $j = r$.

Il tempo di esecuzione di `PARTITION` con il sottoarray $A[p..r]$ è $\Theta(n)$, dove $n = r - p + 1$.

9.2 Prestazioni di QUICKSORT

Il tempo di esecuzione di QUICKSORT dipende dal fatto che il partizionamento può essere bilanciato o meno, e questo a sua volta dipende da quali elementi vengono utilizzati per il partizionamento. Se esso è bilanciato, l'algoritmo ha la stessa velocità asintotica di merge-sort. Se è sbilanciao, può essere asintoticamente lento come INSERTION-SORT.

9.2.1 Partizionamento nel caso peggiore

Il comportamento nel caso peggiore si verifica quando il partizionamento produce un sottoproblema con $n-1$ elementi e uno con 0 elementi. Se questo sbilanciamento si verifica in ogni chiamata ricorsiva, poiché per una chiamata ricorsiva su un array vuoto $T(0) = \Theta(1)$, la ricorrenza può essere espressa così:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Se sommiamo i costi a ogni livello della ricorsione, otteniamo una serie aritmetica di valore $+\Theta(n^2)$. Il tempo di esecuzione $\Theta(n^2)$ si ha quando l'array di input è già completamente ordinato, situazione in cui insertion-sort è eseguito nel tempo $O(n)$.

9.2.2 Partizionamento nel caso migliore

Nel caso di bilanciamento massimo, vengono prodotti due sottoproblemi, ciasuno di dimensione non maggiore di $\frac{n}{2}$ (uno ha dimensione $\lfloor \frac{n}{2} \rfloor$ e l'altro $\lceil \frac{n}{2} \rceil - 1$). In questo caso viene eseguito molto più velocemente. La ricorrenza è:

$$T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$$

che ha soluzione $T(n) = O(n \lg n)$.

9.2.3 Partizionamento bilanciato

Il tempo di esecuzione di quicksort nel caso medio è vicino al caso migliore. Supponiamo una ripartizione 9 a 1, che potrebbe sembrare molto sbilanciata. Otteniamo la ricorrenza:

$$T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + cn$$

Ogni livello dell'albero ha un costo cn, finchè non viene raggiunta una condizione al contorno alla profondità $\log_1 0n$ dopo la quale i livelli hanno al massimo un costo cn. La ricorsione termina alla profondità $\log_{\frac{10}{9}} n = \Theta(\lg n)$. Il costo totale di quicksort è dunque $O(n \lg n)$. Ogni ripartizione con proporzionalità costante produce un albero di ricorsione di profondità $\Theta(\lg n)$, dove il costo di ogni livello è $O(n)$, e il tempo di esecuzione è $O(n \lg n)$.

9.3 Una versione randomizzata di quicksort

A volte è possibile aggiungere la randomizzazione ad un algoritmo per ottenere una buona prestazione con ogni input. Adottiamo il campionamento casuale: anziché usare sempre $A[r]$ come pivot, utilizzeremo un elemento scelto a caso nel sottoarray $A[p..r]$, scambiandoli. In questa nuova procedura, implementiamo semplicemente lo scambio prima dell'effettivo partizionamento:

`RANDOMIZED-PARTITION`(A, p, r)

```
1  i = RANDOM(p, r)
2  scambia A[r] con 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)
```

9.3.1 Analisi del caso peggiore

Una ripartizione nel caso peggiore a qualsiasi livello di ricorsione in quicksort produce un tempo di esecuzione $\Theta(n^2)$. Utilizzando il metodo di sostituzione possiamo dimostrare che il tempo di esecuzione di quicksort è $O(n^2)$. Quando il partizionamento è sbilanciato, impiega un tempo $\Omega(n^2)$.

9.4 Tempo di esecuzione e confronti

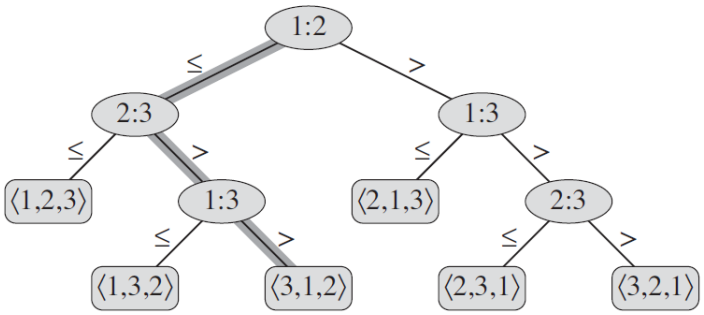
Il tempo di esecuzione di `QUICKSORT` è dominato dal tempo impiegato nella procedura `PARTITION`. Ogni volta che quest'ultima viene chiamata, viene selezionato un pivot che non sarà mai incluso nelle chiamate ricorsive successive. Quindi ci possono essere al massimo n chiamate di `PARTITION`. Una chiamata di `PARTITION` impiega tempo $O(1)$ più una quantità di tempo proporzionale al numero di iterazioni del **for** (righe 3-6). Ogni iterazione del **for** effettua un confronto tra il pivot e un altro elemento dell'array (riga 4). Se X è il numero di confronti svolti nella riga 4 di `PARTITION` nell'intera esecuzione di `QUICKSORT` su un array di n elementi, allora il tempo di esecuzione di `QUICKSORT` è $O(n + X)$.

10 Il modello dell'albero di decisione

Gli **ordinamenti per confronti** (ordinamento basato su confronti di input) possono essere visti come alberi di decisione. Un **albero di decisione** è un albero binario completo che rappresenta i confronti fra elementi che vengono effettuati da un particolare algoritmo.

Ogni nodo interno è rappresentato da $i : j$ per qualche i e j nell'intervallo $1 \leq i,\ j \leq n$ (n numero di elementi di input). Ogni foglia è rappresentata da una permutazione. Il sottoalber sx detta tutti i successivi confronti per $a_i \leq a_j$; quello dx per $a_i \geq a_j$. Quando raggiunge la foglia, l'algoritmo ha stabilito l'ordinamento. Per questo consideriamo solo gli alberi di decisione in cui ogni permutazione si presenta come una foglia raggiungibile.

Nel caso perggiore, il numero di confronti per un algoritmo di ordinamento per confronti è uguale all'altezza del suo albero di decisione.



11 Altri algoritmi

11.1 COUNTING-SORT

Suppone che ciascuno degli elementi di input sia un numero intero compreso tra 0 e k. Quando $k = O(n)$, l'ordinamento viene effettuato nel tempo $\Theta(n)$. Per ogni elemento x di input, determiniamo il numero di elementi minori di x: ad esempio, se ci sono 17 elementi minori di x, allora x è alla posizione 18. Attenzione agli elementi con lo stesso valore!

COUNTING-SORT(A, B, k)

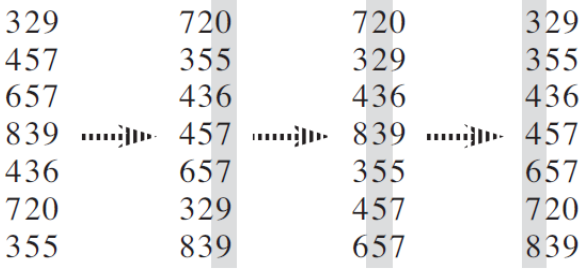
```
1  for i = 0 to k
2      C[i] = 0
3  for j = 1 to A.length
4      C[A[j]] = C[A[j]] +1
5  //C[i] ora contiene il numero di elementi uguali a i
6  for i = 1 to k
7      C[i] = C[i] + C[i-1]
8  //C[i] contiene il nuemro di elementi ≤ i
9  for j = A.length downto 1
10     B[C[A[j]]] = A[j]
11     C[A[j]] = C[A[j]]-1
```

riga 1. Dopo aver creato l'array C, inizializza tutti i valori a 0; riga 4.Conteggio del numero di elementi; riga 10. C contiene gli elementi di input ripetuti una sola volta

A[1..n] input;
B[1..n] array ordinato;
C[0..k] memoria temporanea

11.2 RADIX-SORT

È l'algoritmo usato per ordinare le schede perforate. Ordina prima in base alla cifra meno significativa, combina le schede in un unico mazzo, poi riordina in base alla seconda cifra meno significativa e così via. Se abbiamo un numero di cifre d, occorreranno d passaggi attraverso il mazzo per completare l'ordinamento.



RADIX-SORT(A, d)

```
1  for i = 1 to d
2      do usa ordinamento stabile per ordinare array A su cifra i
```

L'algoritmo ordina nel tempo $\Theta(d(n+k))$ dove n è il numero degli input, d è il numero di cifre e k il valore massimo che una cifra può assumere.
L'algoritmo ordina nel tempo $\Theta((b/r)(n+2^r))$ dove b è il numero di bit e $r \leq b$

12 Mediane e statistiche d'ordine

L'i-esima **statistica d'ordine** di un insieme di n elementi è l'i-esimo numero più piccolo. Il **minimo** è la prima statistica d'ordine, il **massimo** l'n-esima. La **mediana** è il "punto di mezzo" dell'insieme. Se n è dispari essa è unica, se n è pari avremo una **mediana inferiore** $i = \lfloor (n+1)/2 \rfloor$ e una **mediana superiore** $i = \lceil (n+1)/2 \rceil$.

Per determinare il minimo (o il massimo) di un insieme di n elementi, servono al massimo $n-1$ confronti. Per trovare minimo e massimo simultaneamente, sono sufficienti al massimo $3\lfloor \frac{n}{2} \rfloor$ confronti: bisogna conservare gli ultimi elementi minimo e massimo che si sono trovati. Confrontiamo due elementi di input l'uno con l'altro; il più piccolo lo confrontiamo col minimo corrente e il più grande con il massimo corrente. Se n è dispari, assegniamo al minimo e al massimo il valore del primo input, se n è pari confrontiamoi primi due elementi e diventeranno uno il minimo e uno il massimo. Se n è dispari svolgiamo $3\lfloor \frac{n}{2} \rfloor$ confronti, se n è pari invece $3\frac{n}{2} - 2$.

13 RANDOMIZED-SELECT

RANDOMIZED-SELECT è un algoritmo divide et impera modellato sull'algoritmo QUICK-SORT. Opera solo su un lato della partizione ed è randomizzato. Il tempo di esecuzione atteso è $\Theta(n)$, mentre nel caso peggiore $\Theta(n^2)$.

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 //il valore del pivot e' la soluzione
6      return A[q]
7  else if i < k
8      return RANDOMIZED-SELECT(A, p, q - 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

riga 6. A[q] è il pivot
riga 8. opera ricorsivamente sul lato sx della partizione
riga 9. opera ricorsivamente sul lato dx della partizione

Alla fine di PARTITION abbiamo il pivot al centro, alla sua sx gli elementi (non necessariamente ordinati) più piccoli, a dx quelli (non necessariamente ordinati) più grandi.

14 Insiemi dinamici

Gli insiemi manipolati dagli algoritmi possono cambiare nel tempo: sono detti **dinamici**. Gli algoritmi possono richiedere vari tipi di operazioni da svolgere sugli insiemi. Un insieme dinamico che supporta queste operazioni è detto **dizionario**. In un insieme dinamico, un elemento è rappresentato di un oggetto i cui campi possono essere manipolati se vi è un puntatore all'oggetto. Per alcuni tipi di insiemi dinamici si suppone che uno dei campi dell'oggetto sia un campo chiave di identificazione. L'oggetto può contenere dati satelliti.

14.1 Operazioni sugli insiemi dinamici

Le operazioni si dividono in **query** (interrogazioni, che restituiscono informazioni sull'insieme) e **operazioni di modifica**. Esse sono:

- `SEARCH`(S, k)
- `INSERT`(S, x)
- `DELETE`(S, x)
- `MINIMUM`(S)
- `MAXIMUM`(S)
- `SUCCESSOR`(S, x)
- `PREDECESSOR`(S, x).

15 Strutture dati elementari

Stack e code - Sono insiemi dinamici dove l'elemento che viene rimosso dall'operazione delete è predeterminato. Lo stack (pila) implementa lo schema LIFO, la coda FIFO.

15.1 Stack

- `INSERT` = **`PUSH`**
- `DELETE` = **`POP`**
- top[S] = indice dell'ultimo elemento inserito

Se si tenta di estarre un elemento da uno stack vuoto (top[S] = 0), si ha un **underflow** dello stack; se top[S] supera n, si ha un **overflow**.

```
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]
```

15.2 Code

- `INSERT` = **`ENQUEUE`** */enkju:/*
- `DELETE` = **`DEQUEUE`** */dekju:/*
- head[Q] = inizio della coda (testa)
- tail[Q] = fine della coda (coda)

Se head[Q] = tail[Q] la coda è vuota. All'inizio head[Q] = tail[Q] = 1. Se si tenta di rimuovere un elemento da una coda vuota, avremo un underflow; se head[Q] = tail[Q] +1, la coda è piena e il tentetivo di inserire un elemento provocherà un overflow.

```
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
```

15.3 Liste concatenate

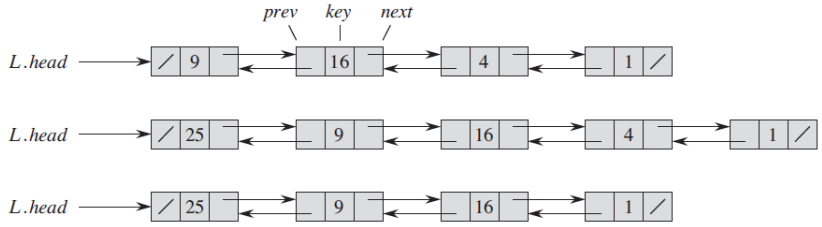
È una struttura dati i cui oggetti sono disposti in ordine lineare (ordine determinato da un puntatore in ogni oggetto).

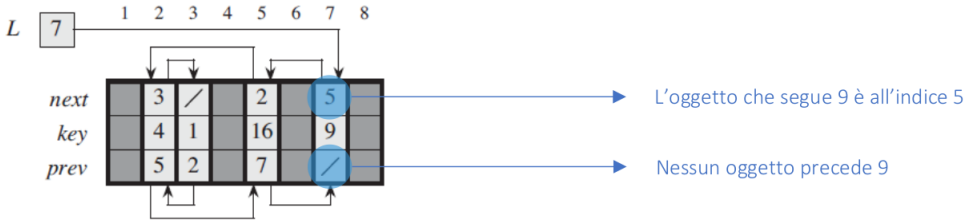
In una **lista doppiamente concatenata** l'oggetto ha un campo chiave **key** e due campi puntatori **next** e **prev**. L'oggetto può anche contenere dati satelliti.

• head[L] (testa) : prev[x] = NIL

• head[L] = NIL – lista vuota

Una **lista singolarmente concatenata** ha solo il puntatore next.
Una lista può essere non ordinata.
In una**lista circolare**, prev[head[L]] == tail[L] e next[tail[L]] == head[L].





L'array *key* contiene i valori che si trovano nell'insieme dinamico; i puntatori son memorizzati negli array *next* e *prev*.

```
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
```

Una **sentinella** è un oggetto fittizio che consente di semplificare le condizioni al contorno. Se sostiuiamo la sentinella nil[L] ad ogni riferimento a NIL, avremo una lista doppiamente concatenata con sentinella. Una lista vuota è formata solo da una sentinella. Le sentinelle vanno usate con attenzione: talvolta possono essere uno spreco di memoria.

15.4 Allocare e liberare gli oggetti

Supponiamo che gli array abbiano lunghezza m e contengano n elementi. Se $n < m$, $m - n$ elementi sono liberi e possono essere utilizzati per rappresentare gli elementi da inserire in futuro. Manteniamo gli oggetti liberi nella lista concatenata **free list** (è uno stack). La testa della free list è nella variabile globale **free**. Quando l'insieme dinamico rappresentato dalla

lista concatenata L non è vuoto, L può essere intrecciato alla free list. Ogni oggetto nella rappresentazione può trovarsi in una sola delle due liste.

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

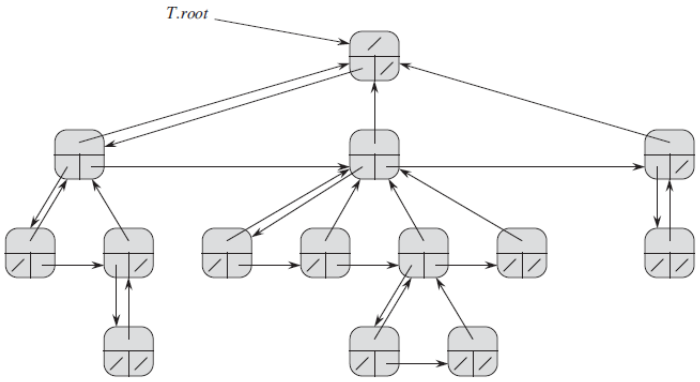
```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

Le due procedure sono eseguite nel tempo $O(1)$.

15.5 Alberi binari

Utilizziamo i campi p, $left$ e $rigth$ per memorizzare rispettivamente padre, figlio sx e figlio dx. L'attributo root[T] punta alla radice dell'albero; se root[T] = NIL allora l'albero è vuoto.

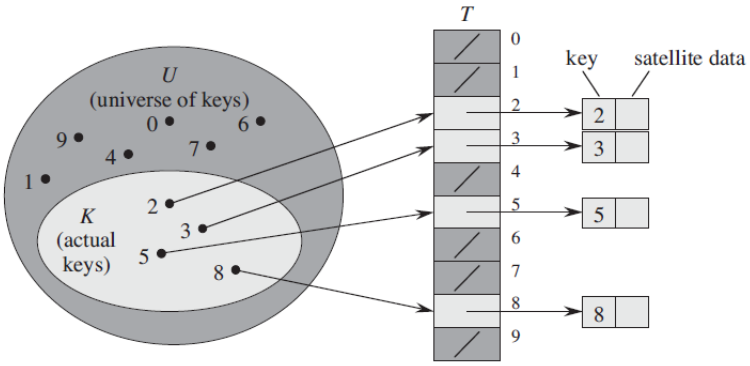
15.5.1 Rappresentazione figlio-sinistro fratello-destro



Prevede che un nodo x abbia solo due puntatori:
- left-child[x] punta al figlio di x più a sx
- right-sibling[x] punta al fratello di x immediatamente a destra

15.6 Tabella a indirizzamento diretto

L'indirizzamento diretto funziona bene quando l'universo U delle chiavi è ragionevolmente piccolo. Supponiamo che un'applicazione abbia bisogno di un insieme dinamico in cui ogni elemento ha una chiave estratta da U e che due elementi non possano avere la stessa chiave. Possiamo usare un array o tabella a indirizzamento diretto $T[0..m-1]$, dove ogni **slot** corrisponde a una chiave in U.



15.6.1 Operazioni del dizionario

```
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
```

In alcune applicazioni, possiamo risparmiare spazio memorizzando l'oggetto nello slot stesso (e non in un elemento esterno alla tabella con un puntatore dalla tabella all'oggetto).

15.7 Tabelle hash

Quando l'insieme K delle chiavi è molto più piccolo di U, una tabella hash richiede molto meno spazio di una tabella a indirizzamento diretto (lo spazio sarà $\Theta(|K|)$). Con l'indirizzamento diretto, un elemento con chiave k è memorizzato nello slot k; con **l'hashing** è memorizzato nello slot $h(k)$; ovvero utilizziamo una **funzione hash** per calcolare lo slot della chiave. h associa U agli slot di una **tabella hash**.

$$h : U \to 0, 1, ..., m - 1$$

$h(k)$ è il valore hash della chiave k.

15.7.1 Risoluzione delle collisioni mediante concatenamento

C'è un problema: due chiavi possono essere associate allo stesso slot (**collisione**). Evitare le collisioni è impossibile. Nel **concatenamento** poniamo tutti gli elementi che sono associati allo stesso slot in una lista concatenata. Lo slot in questione contiene un puntatore alla testa della lista di tutti gli elementi che corrispondono allo slot; se questi elementi non esistono, lo slot conterrà NIL.

15.7.2 Operazioni del dizionario

```
CHAINED-HASH-INSERT(T, x)
1  insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
1  search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
1  delete x from the list T[h(x.key)]
```

15.7.3 Analisi dell'hashing con concatenamento

Dati una tabella hash T con m slot in cui sono memorizzati n elementi, definiamo fattore di carico α della tabella T il rapporto $\frac{n}{m}$ (numero medio di elementi memorizzati in una catena). Se tutte le n chiavi sono associate allo stesso slot (caso peggiore), il tempo di esecuzione della ricerca sarà $\Theta(n)$.

Hashing uniforme semplice se qualsiasi elemento ha le stesse probabilità di essere associato a uno qualsiasi degli slot, indipendentemente dallo slot cui sarà associato qualsiasi altro elemento.

- (nell'ipotesi di hashing uniforme semplice) in una tabella le cui collisioni sono risolte con concatenamento, una ricerca senza successo richiede un tempo atteso $\Theta(1 + \alpha)$.
- (nell'ipotesi di hashing uniforme semplice) in una tabella le cui collisioni sono risolte con concatenamento, una ricerca con successo richiede in media un tempo $\Theta(1 + \alpha)$.

15.7.4 Funzioni hash

Una buona funzione hash soddisfa l'ipotesi di hashing uniforme semplice. Di solito però non è possibile verificare questa condizione. La maggior parte delle funzioni hash suppone che l'universo U delle chiavi sia l'insieme dei numeri naturali: quindi se le chiavi non sono numeri naturali occorre un metodo per interpretarli come tali.
Metodo della divisione - $h(k) = kmod$ m
Metodo della moltiplicazione - $h(k) = \lfloor m(kAmod1) \rfloor$

16 Alberi binari di ricerca

Sono strutture dati che supportano molte operazioni sugli insiemi dinamici (`SEARCH`, `MAXIMUM`, `MINIMUM`, `PREDECESSOR`, `SUCCESSOR`, `INSERT`, `DELETE`). Un albero binario può quindi essere usato sia come dizionario che come coda di priorità. È organizzato come un albero binario in cui ogni nodo è un oggetto che contiene i campi key, p, left e right (con il campo root che è la radice dell'albero) il cui **figlio sx è minore** di esso, mentre il **figlio dx è maggiore**.

Questa proprietà consente di visualizzare ordinatamente tutte le chiavi con un semplice algoritmo ricorsivo di **attraversamento simmetrico** dell'albero (inorder, SND). Possiamo usare un algoritmo di **attraversamento anticipato** dell'albero (preorder, NSD) oppure un algoritmo di **attraversamento posticipato** dell'albero (postorder, SDN).

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

`TREE-PREDECESSOR`(x) analogo

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4       y = x
 5       if z.key < x.key
 6            x = x.left
 7       else x = x.right
 8  z.p = y
 9  if y == NIL
10       T.root = z          // tree T was empty
11  elseif z.key < y.key
12       y.left = z
13  else y.right = z
```

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2       T.root = v
3  elseif u == u.p.left
4       u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7       v.p = u.p
```

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2       TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4       TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6       if y.p ≠ z
 7            TRANSPLANT(T, y, y.right)
 8            y.right = z.right
 9            y.right.p = y
10       TRANSPLANT(T, z, y)
11       y.left = z.left
12       y.left.p = y
```

17 Riepilogo

Tempi di esecuzione

Algoritmo	**Riassunto**	**caso migliore**	**caso peggiore**	**caso medio**
Insertion sort	In place. Per un array abbiamo due indici, uno punta a l'elemento da controllare, uno a quello immediatamente precedente. Se l'elemento precedente è più grande, i valori vengono scambiati. Confronta l'elemento selezionato con tutti i suoi precedenti.	$\Theta(n)$ - $an + b$	$\Theta(n^2)$ - $an^2 + bn + c$	$\Theta(n^2)$
Bubble sort	Seleziona due elementi e li confronta. Se quello precedente è più grande, vengono scambiati. Entrambi gli indici aumentano di 1, l'algoritmo si ripete. Attraversato tutto l'array, si ricomincia.			
Merge (Procedura)	Crea due array L e R in cui inserisce la copia dei due sottoarray di A più un valore sentinella (∞). Combina (prende i primi elemento in L e R e li confronta, inserisce il più grande tra i due nell'array di output.)		$\Theta(n)$	

		Tempi di esecuzione		
Algoritmo	**Riassunto**	**caso migliore**	**caso peggiore**	**caso medio**
Merge sort	Divide et impera. La sequenza di n elementi viene divisa in due sottosequenze di n/2 elementi ciascuna; le due sottosequenze vengono risolte in modo ricorsivo tramite `MERGE-SORT`; le sottosequenze ordinate vengono unite con `MERGE` per generare la sequenza ordinata (risoluzione del problema)		$\Theta(n \lg n)$	$\Theta(n \lg n)$
Max-heapify (Procedura)	Viene determinato il più grande elemento tra A[i] e i figli sx e dx. L'indice viene memorizzato in massimo. Se A[i] è il più grande, il sottoalbero con radice al nodo i è un max- heap, altrimenti A[i] viene scambiato con A[massimo]. La procedura viene chiamata ricorsivamente per tutto l'albero. NB. h è l'altezza del nodo.		$O(\lg n)$ - $O(h)$	

		Tempi di esecuzione		
Algoritmo	**Riassunto**	**caso migliore**	**caso peggiore**	**caso medio**
Build-max-heap (Procedura)	Da un array A[1...n] viene creato un albero binario, che viene trasformato in un max-heap chiamando la procedura `MAX-HEAPIFY`.		$O(n)$	
Heapsort	Su un array A[1...n] opera `BUILD-MAX- HEAP`. L'elemento alla radice è dunque il maggiore e può essere inserito in A[n]. Viene rimosso il nodo n dall'heap (posizionandolo al posto della radice). Questo nuovo albero viene convertito in un max-heap e si ripete il processo.		$O(n \lg n)$	
Heap-extract-max	La radice viene memorizzata in max. Si opera come in `HEAPSORT` spostando l'ultima foglia alla radice e chiamando `MAX- HEAPIFY`.		$O(\lg n)$	

		Tempi di esecuzione		
Algoritmo	**Riassunto**	**caso migliore**	**caso peggiore**	**caso medio**
Heap-increase-key	Ad A[i] viene assegnato il nuovo valore (maggiore o uguale al precedente). A[i] viene (ripetutamente) confrontato con il padre perché potrebbe essere più grande di esso. In tal caso le due chiavi vengono tra di loro scambiate.		$O(\lg n)$	
Max-heap-insert	Si aggiunge una foglia al max-heap (con chiave $-\infty$). Chiama `HEAP-INCREASE-KEY` per impostare la nuova chiave e mantenere le proprietà del max-heap.		$O(\lg n)$	
Partition (procedura)	In place. Seleziona x = A[r] come pivot su cui partizionare A[p..r] (elementi minori a sx del pivot ed elementi maggiori a dx). Viene posto un "muro" tra le posizioni 0 e 1 dell'array. Si scorre l'array: se l'elemento è minore del pivot, viene posto prima del muro, altrimenti rimane dopo di esso. Alla fine il pivot verrà inserito tra i due gruppi dividi dal muro. Divide et impera		$\Theta(n)$	

		Tempi di esecuzione		
Algoritmo	**Riassunto**	**caso migliore**	**caso peggiore**	**caso medio**
Quicksort	Divide et impera. In place. Usa ricorsivamente `PARTITION`.		$\Theta(n^2)$	$\Theta(n \lg n)$ - atteso
Counting sort	Partiamo da un array A. Ogni input è un numero intero compreso tra 0 e k. Per ogni input x viene determinato il numero di elementi minori di x (se vi sono 5 elementi minori di x, x è alla posizione 6). Creiamo un array C di k+1 elementi (la posizione inizia da 0), inizializzati a 0. Scorriamo l'array A: ogni volta che incontriamo un numero y, incrementiamo di 1 l'elemento alla posizione y nell'array C (stiamo contando per ogni x quanti numeri sono uguali a x). Ora vogliamo sapere per ogni x quanti numeri sono più piccoli o uguali a x: scorriamo l'array C e per ogni posizione y sostituiamo la somma del valore alla posizione y + il valore alla posizione y-1. Creiamo un array B della stessa dimensione di A. Scorriamo l'array A: inseriamo x nella posizione data dall'elemento di C in posizione x; decrementiamo di 1 l'elemento in questione di C.		$\Theta(k+n)$	$\Theta(k+n)$

		Tempi di esecuzione		
Algoritmo	**Riassunto**	**caso migliore**	**caso peggiore**	**caso medio**
Radix sort	Ordina tramite un algoritmo stabile ausiliario partendo dalla cifra meno significativa a quella più significativa.		$\Theta(d(n+k))$	
Randomized-select	Divide et impera. Opera come quick-sort, ma su un solo lato della partizione.		$\Theta(n^2)$	$\Theta(n)$
List-search	Semplice ricerca lineare che restituisce un puntatore all'oggetto.		$\Theta(n)$	
List-insert	Inserisce x davanti alla lista (all'inizio) concatenata.			$O(1)$
List-delete	Riceve un puntatore a x (dopo aver chiamato `LIST-SEARCH`), elimina x e aggiorna i puntatori.		$\Theta(n)$	$O(1)$
Direct-address-search Direct-address-insert Direct-address-delete Chained-hash-insert Chained-hash-search Chained-hash-delete				$O(1)$
Inorder-tree-walk				$\Theta(n) - nnodi$

		Tempi di esecuzione		
Algoritmo	**Riassunto**	**caso migliore**	**caso peggiore**	**caso medio**
Tree-search				
Tree-minimum				
Tree-maximum				
Tree-successor				$O(h)$ - h altezza albero
Tree-predecessor				
Tree-insert				
Tree-delete				