Riassunti

Indice

Ch 2 - Insertion Sort

Ch 3 - Notazione asintotica

Ch 4 - Divide et Impera

Ch7 - Quicksort

Ch8 - Limite ordinamento

Ch11 - Hashing

CGGR: Alberi binari

Capitolo 2 - Insertion Sort

Algoritmo

Sequenza di passi computazionali che trasformano l'input nell'output.

É **corretto** se per ogni istanza di input termina con l'output corretto.

2.1: Insertion sort

Risolve il problema dell'ordinamento:

> **Input**: una sequenza di n numeri $< a_1, a_2, ..., a_n >$
>
> **Output**: una permutazione (riordinamento) $< a'_1, a'_2, ..., a'_n >$ della sequenza di input tale che $a'_1 \leq a'_2 \leq ... \leq a'_n$
>
> I numeri da ordinare sono detti **chiavi**

Efficiente nell'ordinamento di un piccolo numero di elementi. I numeri sono ordinati in loco.

```
Insertion-sort(A)
1 for j = 2 to A.length do
2   key = A[j]
3   // Si inserisce A[j] nella sequenza ordinata A[1...j-1]
4   i = j-1
5   while i > 0 e A[i]>key
6     do  A[i+1] = A[i]
7         i = i-1
8   A[i+1] = key
```

È un algoritmo incrementale.

Invarianti di ciclo

Gli **invarianti di ciclo** ci aiutano a dire perché un algoritmo è corretto.

Invariante di insertion sort

> All'inizio di ogni iterazione gli elementi A[1…j-1] sono ordinati e son gli stessi elementi che originariamente erano in A[1…j-1].

Proprietà che un invariante deve rispettare

- **Inizializzazione**: è vera prima della prima iterazione del ciclo. [caso base]

- **Conservazione**: se è vera prima di un iterazione del ciclo, rimane vera prima della successiva iterazione. [passo induttivo]
- **Conclusione**: quando il ciclo termina, l'invariante fornisce un'utile proprietà che ci aiuta a dimostrare che l'algoritmo è corretto.

2.2: Analisi degli algoritmi

Analizzare un algoritmo significa prevedere le risorse che richiedo, spesso misurando il tempo di elaborazione.

Tecnologia di implementazione: **random access machine (RAM)**. Le istruzioni vengono eseguite sequenzialmente, non abbiamo operazioni contemporanee. Queste istruzioni sono quelle che si trovano comunemente nei computer reali, che avranno costo costante.

Il tempo richiesto da un'algoritmo cresce con la dimensione dell'input. La **dimensione dell'input** può essere la dimensione n dell'array, il numero totale di bit richiesti per rappresentare l'input binario, o a volte può essere più di un numero, dipende dal problema. Il **tempo di esecuzione** è rappresentato dal numero di operazioni primitive che vengono eseguite, o "passi", che noi rappresentiamo con una riga di codice. In generale, supponiamo che l'esecuzione dell'i-esima riga richieda un tempo c_i, dove c_i è una costante.

Analisi di Insertion Sort

Per ogni $j = 2, 3, ..., n$ dove $n = A.length$, indichiamo con t_j il numero di volte che il test del ciclo while nella riga 5 viene eseguito per quel valore di j.

Inoltre, il test di un ciclo viene eseguito una volta in più del corpo del ciclo.

Insertion-sort(A)	costo	numero di volte
1 for j = 2 to A.length do	c_1	n
2 key = A[j]	c_2	$n-1$
3 // Si inserisce A[j] nella sequenza ordinata A[1...j-1]	0	$n-1$
4 i = j-1	c_4	$n-1$
5 while i > 0 e A[i]>key	c_5	$\sum_{j=2}^{n} t_j$
6 do A[i+1] = A[i]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 i = i-1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 A[i+1] = key	c_8	$n-1$

> Il **tempo di esecuzione totale** è la somma dei tempi di esecuzione di ogni istruzione eseguita:
> $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$

Anche per input della stessa dimensione, il valore di $T(n)$ dipende da quale input viene scelto.

Per insertion sort, il **caso migliore** è dato da un array già ordinato; questo vuol dire che non entrerà mai nel while e il suo costo è:
$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$
$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$
$= an + b$
con le costanti a, b che dipendono dai c_i, quindi è una **funzione lineare** di n.

Se l'array è ordinato in senso inverso, allora si verifica il **caso peggiore**, in cui ogni elemento $A[j]$ viene confrontato con ogni elemento del sotto-array $A[1..j-1]$, e quindi $t_j = j$ per $j = 2...n$.
Poiché $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ e $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$:

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_8(n-1)$$
$$= (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$$
$$= an^2 + bn + c$$

con le costanti a, b, c che dipendono dai costi delle istruzioni c_i, quindi è una **funzione quadratica** di n.

In generale, consideriamo soltanto il **tempo di esecuzione al caso peggiore**, perché è un limite superiore per il tempo di esecuzione per qualsiasi input e spesso il caso medio è molto simile.

Inoltre, consideriamo il **tasso di crescita** del tempo di esecuzione: consideriamo i termini di grado maggiore e ignoriamo le costanti ad esse moltiplicate. Quindi il costo $an^2 + bn + c$ diventa $\Theta(n^n)$.

2.3: Progettare gli algoritmi

Divide et Impera

È un approccio usato da algoritmi iterativi.

Prevede tre passi ad ogni livello di ricorsione:

1. **Divide**: Il problema viene diviso in un certo numero di sotto-problemi, che sono istanze più piccole dello stesso problema.
2. **Impera**: i sotto-problemi vengono risolti in modo ricorsivo; quando hanno dimensione abbastanza piccola vengono risolti direttamente.
3. **Combina**: le soluzioni del sotto-problema vengono combinate per generare la soluzione del problema originale

L'algoritmo **merge sort** usa un approccio divide et impera; intuitivamente funziona nel modo seguente:

1. **Divide**: divide la sequenza degli n elementi da ordinare in due sotto-sequenza di $\frac{n}{2}$ elementi ciascuna.
2. **Impera**: ordina le due sotto-sequenza in modo ricorsivo utilizzando l'algoritmo merge sort.
3. **Combina**: fonde le due sotto-sequenze ordinate per generare la sequenza ordinata.

Il caso base di merge sort è la sequenza di lunghezza 1, che è già ordinata.

```
MERGE(A,p,q,r)
01      n1 = q - p + 1
02      n2 = r - q
03      crea due nuovi array L[1 ... n1+1] e R[1 ... n2+1]
04      for i = 1 to n1
05          L[i] = A[p + i - 1]
06      for j = 1 to n2
07          R[i] = A[q + j]
08      L[n1 + 1] = inf
09      R[n2 + 1] = inf
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
```

Il ciclo for alle righe 12-17 ha il seguente invariante:

> All'inizio di ogni iterazione del ciclo for il sotto-array $A[p...k-1]$ contiene, ordinati, i $k-p$ elementi più piccoli di $L[1...n_1+1]$ e $R[1...n_2+1]$. Inoltre $L[i]$ e $R[j]$ sono i più piccoli elementi dei loro array che non sono stati copiati in A.

La procedura `MERGE` viene eseguita nel tempo $\Theta(n)$, perché i cicli for alle righe 4-7 impiegano un tempo $\Theta(n_1 + n_2) = \Theta(n)$ e il ciclo alle righe 12-17 viene eseguito per n iterazioni, dove ciascuna di queste ha un tempo costante.

La procedura `MERGE` viene chiamata all'interno della funzione `MERGE-SORT` che ordina gli elementi nel sotto-array $A[p...r]$. Se $p \geq r$, il sotto-array ha al massimo un elemento, e quindi è già ordinato; altrimenti, il passo "divide" calcola semplicemente un indice q che separa $A[p...r]$ in due sotto-array: $A[p...q]$ e $A[q+1...r]$.

```
MERGE-SORT(A,p,r)
1   if p < r
2       q = parte intera inferiore di (p+r)/2
3       MERGE-SORT(A,p,q)
4       MERGE-SORT(A,q+1,r)
5       MERGE(A,p,q,r)
```

Analisi degli algoritmi divide et impera

Il tempo di esecuzione di un algoritmo rcorsivo può essere espresso con un'**equazione di ricorrenza**, che esprime il tempo di esecuzione in funzione del tempo di esecuzione per input più piccoli.

Se $T(n)$ è il tempo di esecuzione per un problema di dimensione n, possiamo dire che per n sufficientemente piccolo ($n \leq c$) la soluzione richiede un tempo costante $\Theta(1)$.
Se la nostra suddivisione del problema genera a sotto-problemi di dimensione $1/b$ volte la dimensione del problema originale, serve tempo $T(n/b)$ per risolvere un sotto-problema di dimensione n/b. Alternativamente, n/b è la dimensione di ognuno dei a sottoproblemi.

Se impieghiamo un tempo $D(n)$ per dividere in sotto-prblemi e $C(n)$ per combinare le soluzioni, otteniamo:

$$T(n) = \begin{cases} \Theta(1) & se\ n \leq c \\ aT(n/b) + D(n) + C(n) & altrimenti \end{cases}$$

Analisi di MERGE-SORT

L'algoritmo applicato ad un solo elemento impiega un tempo costante, lo usiamo come caso base.

- Divide: consiste nel calcolare il centro del sotto-array, quindi $D(n) = \Theta(1)$
- Impera: risolviamo due sottoproblemi di dimensione $n/2$, quindi $2T(n/2)$
- Combina: dato dalla procedura `MERGE` che costa $C(n) = \Theta(n)$

Sommando $D(n)$ e $C(n)$ otteniamo una funzione lineare in n, ovvero costa $\Theta(n)$.

Quindi `MERGE-SORT` al caso peggiore costa:

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

Con un po' di calcoli ottengo $T(n) = \Theta(n\ log_2 n)$

Capitolo 3 - Crescita delle funzioni

3.1 - Notazione asintotica

Quando vogliamo descrivere il tempo di esecuzione di un algoritmo, per input sufficientemente grandi le costanti moltiplicative e i termini di ordine inferiore sono dominati dagli effetti della dimensione dell'input. In questi casi, stiamo studiando l'efficienza **asintotica** degli algoritmi. Di solito un algoritmo che è asintoticamente più efficiente sarà il migliore con tutti gli input, tranne con quelli molto piccoli.

Useremo la notazione asintotica per descrivere i tempi di esecuzione degli algoritmi, ma questa si applica alle funzioni.

Notazione Θ

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

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \ t.c. \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \ \forall n \geq n_0\}$$

Una funzione $f(n)$ appartiene all'insieme $\Theta(g(n))$ se esistono delle costanti positive c_1 e c_2 tali che essa possa essere "racchiusa" fra $c_1 g(n)$ e $c_2 g(n)$, per valori sufficientemente grandi di n.
In altre parole, per ogni $n \geq n_0$, la funzione $f(n)$ è uguale a $g(n)$ a meno di un fattore costante. Si dice che $g(n)$ è un **limite asintoticamente stretto** per $f(n)$.

Supponiamo che ogni funzione utilizzata nella notazione asintotica sia **asintoticamente non negativa**, perchè la definizione di $\Theta(g(n))$ lo richiede di $f(n)$ per n sufficientemente grande, e questo implica he $g(n)$ stessa deve essere asintoticamente non negativa.

Intuitivamente, i termini di ordine inferiore di una funzione asintoticamente positiva possono essere ignorati nel determinare i limiti asintoticamente stretti, perchè sono insignificanti per grandi valori di n.

Notazione O

Fornisce un **limite asintotico superiore**.

$$O(g(n)) = \{f(n) : \exists c, n_0 > 0 \ t.c. \\ 0 \leq f(n) \leq cg(n) \ \forall n \geq n_0\}$$

La notazione O si usa per assegnare un limite superiore a una funzione, a meno di un fattore costante.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$, in quanto la notazione Θ è più forte della notazione O. Possiamo scrivere $\Theta(g(n)) \subseteq O(g(n))$.

Poiché la notazione O descrive un limite superiore, nell'esempio di `INSERTION-SORT`, il limite di $O(n^2)$ si applica anche al suo tempo di esecuzione per qualsiasi input. Il limite $\Theta(n^2)$ sul caso peggiore invece non si applica a qualsiasi input, per esempio se l'array è già ordinato sappiamo che `INSERTION-SORT` viene eseguito nel tempo $\Theta(n)$.

Notazione Ω

Fornisce un **limite asintotico inferiore**.

$$\Omega(g(n)) = \{f(n) : \ \exists c, n_0 > 0 \ t.c. \\ 0 \leq cg(b) \leq f(n) \ \forall n \geq n_0\}$$

> **Teorema 3.1**:
> Per ogni coppia di funzioni $f(n)$ e $g(n)$, si ha $f(n) = \Theta g(n)$, se e soltanto se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Notazione o

Denota un limite superiore che non è asintoticamente stretto.

$$o(g(n)) = \{f(n) : \forall c > 0, \exists n_0 > 0 \ t.c. \ 0 \leq f(n) < cg(n), \forall n \geq n_0\}$$

Per esempio, $2n = o(n^2)$, ma $2n^2 \neq o(n^2)$.

La grande differenza con la notazione O è che, nella notazione O, il limite vale per *qualche* costante $c > 0$; invece nella notazione *o* il limite $0 \leq f(n) < cg(n)$ deve valere per *tutte* le costanti $c > 0$.

Notazione ω

Denota un limite inferiore che non è asintoticamente stretto.

$$\omega(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0, \ t.c. \ 0 \leq cg(n) < f(n), \forall n \geq n_o\}$$

Un modo alternativo di definirla è:

"

Confronto di funzioni

Proprietà transitiva:

$f(n) = \Theta(g(n)), g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

$f(n) = O(g(n)), g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

$f(n) = \Omega(g(n)), g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

$f(n) = o(g(n)), g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$

$f(n) = \omega(g(n)), g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

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:

$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

$f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

Analogia con confronto di due numeri reali a, b

$f(n) = O(g(n)) \equiv a \leq b$

$f(n) = \Omega(g(n)) \equiv a \geq b$

$f(n) = \Theta(g(n)) \equiv a = b$

$f(n) = o(g(n)) \equiv a < b$, $f(n)$ è asintoticamente più piccola di $g(n)$

$f(n) = \omega(g(n)) \equiv a > b$, $f(n)$ è asintoticamente più grande di $g(n)$

Tricotomia: se a e b sono due numeri reali qualsiasi, dve essere valida una sola delle seguenti relazioni: $a < b, a = b, a > b$.

Non tutte le funzioni sono asintoticamente confrontabili, per esempio le funzioni n e $n^{1+sin\ n}$ non sono confrontabili.

3.2 - Notazioni Standard e funzioni comuni

Questo paragrafo contiene riferimenti a: Funzioni monotone, Floor e ceiling, Aritmetica modulare, Polinomi, Esponenziali, Logaritmi, Fattoriale, Iterazione di una funzione, La funzione logaritmo iterato, Numeri di Fibonacci

Capitolo 4 - Divide et Impera

4.2 - Algoritmo di Strassen

É un algoritmo usato per calcolare il prodotto di due matrici.

Metodo semplice

Il metodo semplice segue dalla definizione di prodotto di matrice $A = (a_{ij}), B = (b_{ij}), C = A \cdot B \rightarrow c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{ik}$ per $i, j = 1...n$.

Dobbiamo calcolare n^2 elementi della matrice, ciascuno dei quali è la somma di n valori; questo è dato da tre `for` annidati: uno scorre le righe di C, uno scorre le colonne di C e l'ultimo calcola la sommatoria.

Questo algoritmo ha costo $\Theta(n^3)$.

Metodo divide et impera

Possiamo semplificare il calcolo implementando un metodo divide et impera. Supponiamo che la dimensione delle matrici n sia una potenza esatta di 2 in modo da avere la certezza che, finché $n \geq 2$, $n/2$ sia un intero.

Dividiamo ciascuna delle matrici in 4:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Il prodotto $C = A \cdot B$ corrisponde alle seguenti equazioni:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Ciascuna di queste equazioni specifica il prodotto di due matrici $n/2 \times n/2$ e la somma dei loro prodotti $n/2 \times n/2$.

In questo modo possiamo scrivere un semplice algoritmo divide et impera, il cui caso base, per $n = 1$ è composto da $c_{11} = a_{11} \cdot b_{11}$. Il passo ricorsivo consiste nel suddividere le tre matrici come sopra e di creare gli elementi $C_{11}, C_{12}, C_{21}, C_{22}$ come dalle equazioni sopra, dove ogni prodotto sarà una chiamata ricorsiva.

La suddivisione delle matrici, se le copiamo in nuove matrici di dimensione $n/2 \times n/2$ ha costo $\Theta(n^2)$, ma si può realizzare in tempo costante usando il calcolo degli indici.

Utilizziamo ora una ricorrenza per descrivere il costo di questa procedura: il caso base ha tempo costante. In quello ricorsivo ho otto chiamate ricorsive, ognuna delle quali moltiplica due matrici $n/2 \times n/2$, che va a sommare a coppie per un costo $\Theta(n^2)$:

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ 8T(n/2) + \Theta(n^2) & se\ n > 1 \end{cases}$$

Questa ricorrenza ha soluzione $T(n) = \Theta(n^3)$.

Metodo di Strassen

Variazione del metodo ricorsivo, in cui ho 7 chiamate ricorsive invece di 8. É composto da 4 passi:

1. Divido le matrici A, B, C in dodici sottomatrici $n/2 \times n/2$, come descritto nel metodo precedente.
   Richiede un tempo $\Theta(1)$ per il calcolo degli indici.
2. Creo dieci matrici $S_1, S_2, ...S_{10}$, ciascuna di dimensione $n/2 \times n/2$ ed è la somma o differenza di due matrici create al passo 1.
   Richiede un tempo $\Theta(n^2)$ per creare tutte le sottomatrici.
3. Utilizzando tutte le sottomatrici create fino ad ora, calcolo sette matrici prodotto $P_1, P_2, ..., P_7$, ciascuna di dimensione $n/2 \times n/2$
4. Calcolo le matrici richieste $C_{11}, C_{12}, C_{21}, C_{22}$ sommando e/o sottraendo varie combinazioni delle matrici P_i.
   Possiamo calcolarle tutte in un tempo $\Theta(n^2)$.

Da queste informazioni possiamo già calcolare la ricorrenza per il tempo di esecuzione dell'algoritmo di Strassen:

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ 7T(n/2) + \Theta(n^2) & se\ n > 1 \end{cases}$$

Che ha soluzione $T(n) = \Theta(n^{lg7})$.

4.5 - Il metodo dell'esperto

Serve a risolvere le ricorrenze del tipo $T(n) = aT(n/b) + f(n)$, dove $a \geq 1$ e $b > 1$ sono costanti e $f(n)$ è una funzione asintoticamente positiva.

La ricorrenza sopra descrive il tempo di esecuzione di un algoritmo che divide un problema di dimensione n in a sotto-problemi di dimensione n/b. I sotto-problemi vengono risolti in modo ricorsivo, ciascuno nel tempo $T(n/b)$.

Teorema dell'esperto

Date le costanti $a \geq 1$ e $b > 1$ e la funzione $f(n)$, sia $T(n)$ una funzione definita sugli interi non negativi della ricorrenza:

$$T(n) = aT(n/b) + f(n)$$

dove n/b rappresenta $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. 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 $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$.

Intuitivamente, confrontiamo $f(n)$ con la funzione $n^{log_b a}$, e in base a quale delle due è "più grande" ci troviamo potenzialmente in uno dei tre casi (il terzo caso ha dei vincoli in più).

Dimostrazione del teorema dell'esperto per potenze esatte

Analizziamo la ricorrenza $T(n) = aT(n/b) + f(n)$ nell'ipotesi che n sia una potenza esatta di $b > 1$, dove b non deve necessariamente essere un intero. Questa analisi è suddivisa in tre lemmi:

1. **Lemma 4.2**

   Siano $a \geq 1$ e $b > 1$ due costanti e $f(n)$ una funzione non negativa definita sulle potenze esatte di b. Se $T(n)$ è definita sulle potenze esatte di b dalla ricorrenza

   $$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ aT(n/b) + f(n) & se\ n = b^i \end{cases}$$

   dove i è un intero positivo, allora

$$T(n) = \Theta(n^{log_b a}) + \sum_{j=0}^{log_b n - 1} a^j f(n/b^j) \quad (1)$$

Dimostrazione

Si basa su un albero di ricorsione. La radice dell'albero ha costo $f(n)$ e ha a figli, ciascuno di costo $f(n/b)$. Ciascuno di questi figli ha, a sua volta, a figli con un costo di $f(n/b^2)$; quindi ci sono a^2 nodi alla profondità 2.

In generale, ci sono a^j nodi alla profondità j, ciascuno dei quali ha un costo di $f(n/b^j)$.

Il costo di ogni foglia è $T(1) = \Theta(1)$ e ogni foglia si trova alla profondità $log_b n$, in quanto $n/b^{log_b n} = 1$.

L'albero ha $a^{log_b n} = n^{log_b a}$ foglie.

Sommiamo i costi di ogni livello dell'albero; il costo per un livello j di nodi interni è $a^j f(n/b^j)$, quindi il totale dei nodi interni è:

$\sum_{j=0}^{log_b n - 1} a^j f(n/b^j)$

Questa sommatoria rappresenta i costi per dividere i problemi in sottoproblemi e poi per ricombinare i sottoproblemi.

Il costo di tutte le foglie, che è il costo per svolgere $n^{log_b a}$ sottoproblemi di dimensione 1, p $\Theta(n^{log_b a})$

n/b è la dimensione di ognuno dei a sottoproblemi.

I tre casi del teorema dell'esperto corrispondono ai casi in cui il costo dell'albero è (1) dominato dai costi delle foglie, (2) equamente distribuito fra i livelli dell'albero o (3) dominato dal costo della radice.

2. **Lemma 4.3**

   Siamo $a \geq 1$ e $b > 1$ due costanti e $f(n)$ una funzione non negativa definita dalle potenze esatte di b. Una funzione $g(n)$ definita sulle potenze di b da

$$g(n) = \sum_{j=0}^{log_b n - 1} a^j f(n/b^j) \quad (2)$$

   ha i seguenti limiti asintotici per le potenze esatte di b.

   1. Se $f(n) = O(n^{log_b a - \epsilon})$ per qualche costante $\epsilon > 0$, allora $g(n) = O(n^{log_b a})$
   2. Se $f(n) = \Theta(n^{log_b a})$, allora $g(n) = \Theta(n^{log_b a}\ lg\ n)$.
   3. Se $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e per ogni n sufficientemente grande, allora $g(n) = \Theta(f(n))$.

   Dimostrazione

   Per il caso 1, abbiamo $f(n) = O(n^{log_b a - \epsilon})$, che implica che $f(n/b^i) = O((n/b^j)^{log_b a - \epsilon})$. Sostituendo nella (2), otteniamo

$$g(n) = O(\sum_{j=0}^{log_b n - 1} a^j (\frac{n}{b^j})^{log_b a - \epsilon}) \quad (3)$$

   Limitiamo la sommatoria all'interno della notazione O mettendo in evidenza i fattori comuni e semplificando; alla fine otteniamo una serie geometrica crescente:

$$\begin{aligned}\sum_{j=0}^{log_b n-1} a^j (\frac{n}{b^j})^{log_b a-\epsilon} &= n^{log_b a-\epsilon} \sum_{j=0}^{log_b n-1} (\frac{ab^\epsilon}{b^{log_b a}})^j \\ &= n^{log_b a-\epsilon} \sum_{j=0}^{log_b n-1} (b^\epsilon)^j \\ &= n^{log_b a-\epsilon} (\frac{b^{\epsilon log_b n}-1}{b^\epsilon-1}) \\ &= n^{log_b a-\epsilon} (\frac{n^\epsilon-1}{b^\epsilon-1})\end{aligned}$$

Poiché b e ϵ sono costanti, possiamo riscrivere l'ultima espressione così:

$n^{log_b a-\epsilon}\ O(n^\epsilon) = O(n^{log_b a})$

Sostituendo la sommatoria dell'equazione (3) con questa espressione, si ha $g(n) = O(n^{log_b a})$, il che prova il caso 1.

Poiché il caso 2 assume $f(n) = \Theta(n^{log_b a})$, si ha $f(n/b^j) = \Theta((n/b^j)^{log_b a})$. Sostituendo nell'equazione (2) si ha:

$$g(n) = \Theta\Big(\sum_{j=0}^{log_b n-1} a^j (\frac{n}{b^j})^{log_b a}\Big) \tag{4}$$

Limitiamo la sommatoria all'interno della notazione Θ come nel caso 1; stavolta, però, non otteniamo una serie geometrica. Scopriamo invece che i termini della sommatoria sono tutti uguali:

$$\begin{aligned}\sum_{j=0}^{log_b n-1} a^j (\frac{n}{b^j})^{log_b a} &= n^{log_b a-\epsilon} \sum_{j=0}^{log_b n-1} (\frac{a}{b^{log_b a}})^j \\ &= n^{log_b a} \sum_{j=0}^{log_b n-1} 1 \\ &= n^{log_b a}\ log_b n\end{aligned}$$

Sostituendo la sommatoria dell'equazione (4) con questa espressione, si ha

$$\begin{aligned}g(n) &= \Theta(n^{log_b a} log_b n) \\ &= \Theta(n^{log_b a} lg\ n)\end{aligned}$$

il che prova il caso 2.

Il **caso 3** si dimostra in modo simile. Poichè $f(n)$ appare nella definizione (2) di $g(n)$ e tutti i termini di $g(n)$ sono non negativi, possiamo concludere che $g(n) = \Omega(f(n))$ per potenze esatte di b.

Nell'enunciato del lemma abbiamo assunto $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e per ogni n sufficientemente grande. Riscriviamo questa assunzione nella forma

$f(n/b) \leq (c/a)f(n)$

e la iteriamo j volte ottenendo

$f(n/b^j) \leq (c/a)^j f(n)$, ovvero $a^j f(n/b^j) \leq c^j f(n)$, dove assumiamo che i valori su cui iteriamo siano sempre sufficientemente grandi. Siccome l'ultimo, e più piccolo, di questi valori è n/b^{j-1} basta assumere che n/b^{j-1} sia sufficientemente grande.

Sostituendo nell'equazione (2) e semplificando, si ottiene una serie geometrica che, diversamente da quella led caso 1, è decrescente. Useremo un termine $O(1)$ per catturare i termini che non soddisfano l'ipotesi che n sia sufficientemente grande.

$$
\begin{aligned}
g(n) &= \sum_{j=0}^{log_b n-1} a^j f(n/b^j) \\
&\leq \sum_{j=0}^{log_b n-1} c^j f(n) + O(1) \\
&\leq f(n) \sum_{j=0}^{\infty} c^j + O(1) \\
&= f(n)(\frac{1}{1-c}) + O(1) \\
&= O(f(n))
\end{aligned}
$$

L'ultimo passaggio è corretto in quanto c è costante. Pertanto possiamo concludere che $g(n) = \Theta(f(n))$ per le potenze esatte di b. Il caso 3 è dimostrato e questo completa la dimostrazione del lemma-

3. **Lemma 4.4**

   Date le costanti $a \geq 1$ e $b > 1$ e $f(n)$ una funzione non negativa sulle potenze esatte di b. Se $T(n)$ è definita sulle potenze esatte di b dalla ricorrenza

   $$
   T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ aT(n/b) + f(n) & se\ n = b^i \end{cases}
   $$

   dove i è un intero positivo, allora $T(n)$ ha i seguenti limiti asintotici per le potenze esatte di b.

   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 $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$.

   Dimostrazione

   Usiamo i limiti del lemma 4.3 per valutare la sommatoria (1).

   Per il caso 1 abbiamo

   $$
   \begin{aligned}
   T(n) &= \Theta(n^{log_b a}) + O(n^{log_b a}) \\
   &= \Theta(n^{log_b a})
   \end{aligned}
   $$

   Per il caso 2 abbiamo

   $$
   \begin{aligned}
   T(n) &= \Theta(n^{log_b a}) + \Theta(n^{log_b a} lg\ n) \\
   &= \Theta(n^{log_b a} lg\ n)
   \end{aligned}
   $$

   Per il caso 3

   $$
   \begin{aligned}
   T(n) &= \Theta(n^{log_b a}) + \Theta(f(n)) \\
   &= \Theta(f(n))
   \end{aligned}
   $$

   in quanto $f(n) = \Omega(n^{log_b a + \epsilon})$

Capitolo 7 - Quicksort

Nel caso peggiore è $\Theta(n^2)$, ma in generale è l'algoritmo utilizzato perché mediamente è efficiente. Il suo tempo di esecuzione atteso è $\Theta(n\ lg\ n)$. Ha anche il vantaggio di ordinare sul posto.

7.1 - Descrizione di quicksort

É basato su divide et impera:

- **Divide**: partiziono l'array $A[p...r]$ in due sottoarray $A[p...q-1]$ e $A[q+1...r]$, eventualmente vuoti, tali che ogni elemento di $A[p...q-1]$ sia minore o uguale a $A[q]$ che, a sua volta, è minore o uguale a ogni elemento di $A[q+1...r]$. Calcolare l'indice q come parte di questa procedura di partizionamento.
- **Impera**: ordinare i due sottoarray $A[p...q-1]$ e $A[q+1...r]$ chiamando ricorsivamente quicksort.
- **Combina**: poiché i sottoarray sono già ordinati, non occorre alcun lavoro per combinarli: l'inter array $A[p...r]$ è ordinato.

La seguente procedura implementa quicksort:

```
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)

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
```

La procedura `PARTITION` riordina il sottoarray sul posto: seleziona l'elemento $x = A[r]$ come **pivot** e durante la procedura l'array viene suddiviso in quattro regioni, eventualmente vuote:

- gli elementi minori o uguali a x
- gli elementi maggiori di x
- gli elementi ancora da visitare
- x

Queste proprietà formano un'invariante di ciclo per `PARTITION`:

> All'inizio di ogni iterazione del ciclo, righe 3-6, 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 $A[k] = x$.

Si dimostra che è vera prima della prima iterazione, che ogni iterazione mantiene queste proprietà e che è vera quando il ciclo termina.

1. **Inizializzazione**: prima della prima iterazione del ciclo, $i = p-1$ e $j = p$. Non ci sono valori fra p e i né fra $i+1$ e $j-1$. L'assegnazione nella riga 1 soddisfa la terza condizione.
2. **Conservazione**: ci sono due casi da considerare, a seconda se si entra nell'if della riga 4:
   Se $A[j] > x$, l'unica azione è incrementare j. Dopo questo incremento, la condizione 2 è soddisfatta per $A[j-1]$ e tutte le altre posizioni non cambiano.
   Se $A[j] \leq x$, viene incrementato l''indice i, vengono scambiati $A[i]$ e $A[j]$, poi, viene incrementato l'indice j. In seguito allo scambio, abbiamo $A[i] \leq x$ e la condizione 1 è soddisfatta. Analogamente, abbiamo anche

$A[j-1] > x$, in quanto l'elemento che è stato spostato in $A[j-1]$ è, per l'invariante di ciclo, più grande di x.

3. **Conclusione**: alla fine del ciclo, $j = r$. Pertanto, ogni posizione dell'array si trova in uno di tre insiemi: quelli minori o uguali a x, quelli maggiori di x e un insieme di un solo elemento contente x.

Le ultime due righe di `PARTITION` inseriscono il pivot al suo posto nel mezzo dell'array; l'output di partition adesso soddisfa le specifiche del passo decide.

$T(n) = \Theta(n)$

7.2 - Prestazioni di quicksort

Il tempo di esecuzione dipende dal fatto che il partizionamento sia bilanciato o sbilanciato e questo, a sua volta, dipende da quali elementi vengono selezionati per il partizionamento.
Se il partizionamento è bilanciato, va come `MERGE STORT` → $\Theta(n\ lg\ n)$, partizionamento sbilanciato, va come `INSERTION SORT` → $\Theta(n^2)$

Caso peggiore

`PARTITION` produce un sottoproblema con $n-1$ elementi e no con 0 elementi. Supponiamo che questo sbilanciamento accade ad ogni chiamata ricorsiva, `PARTITION` costa $\Theta(n)$ e la chiamata ricorsiva sull'array vuoto costa $T(0) = \Theta(1)$, quindi :

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

Intuitivamente, otteniamo una serie aritmetica il cui valore è $\Theta(n^2)$.

Caso migliore

Divido in due sottoproblemi, ciascuno di dimensione non maggiore di $n/2$. La ricorrenza è $T(n) \leq 2T(n/2) + \Theta(n)$, che per il caso 2 del teorema dell'esperto ha soluzione $T(n) = O(n\ lg\ n)$

Caso medio

Il tempo di esecuzione in questo caso è più vicino al caso migliore che al caso peggiore.

Se supponiamo che l'algoritmo produca sempre una ripartizione 9 a 1, otteniamo la ricorrenza $T(n) \leq T(9n/10) + T(n/10) + cn$ sul tempo di esecuzione di `QUICKSORT`. Se disegniamo l'albero di ricorsione, otteniamo che ogni livello ha costo cn fino a profondità $log_{10} n$, e da quel punto fino alla profondità dell'ultimo ramo, $log_{10/9} n$, avrà un costo minore o uguale di cn.

Pertanto il costo totale di `QUICKSORT` è $O(n\ lg\ n)$, per qualsiasi ripartizione con proporzionalità costante, perché produco un albero di profondità $\Theta(lg\ n)$, dove il costo di ogni livello è $O(n)$.

Se le partizioni non hanno proporzionalità costante, ad esempio ho ripartizioni "buone" e "cattive" alternate, mi trovo comunque nel caso medio, perchè il costo di una ripartizione sbilanciata seguita da una bilanciata è lo stesso di avere una singola ripartizione bilanciata.

7.3 - Una versione randomizzata di quicksort

Aggiungiamo la randomizzazione per ottenere una buona prestazione attesa con tutti gli input. Usiamo il **campionamento casuale**, ovvero randomizziamo la scelta del pivot; in questo modo ci aspettiamo che la ripartizione dell'array di input sia ben bilanciata.

Le modifiche che operiamo sono poche: `QUICKSORT` chiamerà `RANDOMIZED-PARITION`, che a sua volta chiamerà `PARTITION`:

```
RANDOMIZED-PARTITION(A,p,r)
    i = RANDOM(p,r)
```

```
    scambia A[r] con A[i]
    return PARTITION(A,p,r)
```

7.4 - Analisi di quicksort randomizzato

Tempo di esecuzione atteso

Se in ogni livello di ricorsione, la ripartizione indotta da `RANDOMIZED-PARTITION` pone una frazione costante qualsiasi degli elementi in un lato della partizione, allora l'albero di ricorsione ha profondità $\Theta(lg\ n)$ e in ogni livello viene svolto un lavoro $O(n)$, come abbiamo visto sopra.

Tempo di esecuzione e confronti

`QUICKSORT` e `RANDOMIZED-QUICKSORT` differiscono soltanto per il modo in cui scelgono il pivot; e il tempo di esecuzione di `QUICKSORT` è dominato dal tempo impiegato in `PARTITION`.

Poiché ad ogni chiamata di `PARTITION` viene selezionato un pivot che non sarà incluso nelle chiamate ricorsive, ci possono essere al massimo n chiamate di `PARTITION`.

Il costo di `PARTITION` dipende dal ciclo **for** nelle righe 3-6: ogni iterazione di questo **for** effettua un confronto tra il pivot e un'altro elemento dell'array (riga 4). Pertanto se contiamo quante volte viene eseguita la riga 4 possiamo limitare il tempo totale impiegato nel ciclo **for**.

> **Lemma 7.1**:
> 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 eseciuzione di `QUICKSORT` è $O(n + X)$.

Dimostrazione:

Come detto sopra, l'algoritmo effettua al massimo n chiamate di `PARTITION`, ciascuna delle quali svolge una quantità costante di lavoro e poi esegue il ciclo **for** un certo numero di volte. Ogni iterazione del ciclo **for** esegue la riga 4.

Il nostro obiettivo è quindi calcolare X, il numero di confronti svolti in tutte le chiamate di `PARTITION`. Cerchiam un limite globale sul numero totale di confronti.

Dobbiamo quindi capire quando l'algoritmo confronta due elementi e quando non lo fa, chiamiamo questi z_i, z_j, supponendo $z_i < z_j$ senza perdere di generalità.
Definiamo anche $Z_{ij} = \{z_i, z_{i+1}, ..., z_j\}$ l'insieme degli elementi dell'array ordinati conpresi tra z_i e z_j.
Osserviamo che ogni coppia di elementi viene confrontata al massimo una volta, perché confronto sempre un elemento con il pivot corrente che, terminata la chiamata di `PARTITION` corrente, non verrà più confrontato con nessun altro elemento.

Usiamo le variabili casuali indicatrici: definiamo $X_{ij} = I\{z_i\ è\ confrontato\ con\ z_j\}$

Poiché ogni coppia è confrontata una volta sola, rappresentiamo il numero totale di confronti con:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Prendendo i valori attesi da entrambi i lati e poi applicando la linearità del valore atteso, otteniamo:

$$
\begin{aligned}
E[X] &= E[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} Pr\{z_i \text{ è } confrontato \text{ } con \text{ } z_j\}
\end{aligned}
$$

Resta da calcolare $Pr\{z_i \text{ è } confrontato \text{ } con \text{ } z_j\}$, la nostra analisi suppone che ogni pivot sia scelto in modo casuale e indipendente.

Una volta che viene scelto un pivot x con $x_i < x < z_j$, sappiamo che z_i e z_j non saranno mai confrontati. Se z_i o z_j sono scelti come pivot, saranno confrontati con tutti gli altri elementi. Quindi saranno confrontati solo se vengono scelti come pivot prima di ogni altro elemento in Z_{ij}.

Calcoliamo la probabilità che si verifichi questo evento.
Prima del punto in cui un elemento di Z_{ij} viene scelto come pivot, qualsiasi elemento in Z_{ij} ha la stessa probabilità di essere scelto come primo pivot. Poiché Z_{ij} ha $j - i + 1$ elementi e poiché i pivot vengono scelti in maniera casuale e indipendente, la probabilità che qualsiasi elemento sia il primo ad essere scelto è $1/(j - i + 1)$. Quindi abbiamo:

$$
\begin{aligned}
&Pr\{z_i \text{ è } confrontato \text{ } con \text{ } z_j\} \\
&= Pr\{z_i \text{ } o \text{ } z_j \text{ è } il \text{ } primo \text{ } pivot \text{ } scelto \text{ } in \text{ } Z_{ij}\} \\
&= Pr\{z_i \text{ è } il \text{ } primo \text{ } pivot \text{ } scelto \text{ } in \text{ } Z_{ij}\} + Pr\{z_j \text{ è } il \text{ } primo \text{ } pivot \text{ } scelto \text{ } in \text{ } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1}
\end{aligned}
$$

Combinando con l'equazione precedente otteniamo

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-1+1} \\
&= \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(lg\ n) \\
&= O(n\ lg\ n)
\end{aligned}
$$

Quindi concludiamo che utilizzando `RANDOMIZED-PARTITION`, il tempo di esecuzione atteso di quicksort è $O(n\ lg\ n)$ quando i valori degli elementi sono distinti.

Capitolo 8 - Limiti inferiori per l'ordinamento

In un ordinamento per confronti, usiamo una delle operazioni di confronto: $<, \leq, =, \geq, >$, non possiamo esaminare gli elementi in altro modo per ottenere il loro ordinamento relativo. Supponiamo che tutti gli elementi siano distinti.

Possiamo vedere gli ordinamenti per confronto come un albero di decisione, ovvero un albero binario pieno dove ogni confronto è rappresentato da un nodo. Ogni foglia contiene una permutazioni di tutti gli elementi, e l'esecuzione dell'algoritmo è un cammino da radice a foglia.

Un limite inferiore per il caso peggiore

La lunghezza del cammino semplice più lungo da radice a foglia rappresenta il numero di confronti che si svolgono al caso peggiore.

Quindi un limite inferiore sull'altezza di tutti gli alberi di decisione è un limite inferiore sul tempo di esecuzione di qualunque algoritmo di ordinamento per confronti. Questo limite è stabilito dal seguente teorema:

> **Teorema 8.1**:
> Qualsiasi algoritmo di ordinamento per confronti richiede $\Omega(n\ lg\ n)$ confronti nel caso peggiore.

Dimostrazione:

Per quanto detto sopra, è sufficiente determinare l'altezza di un albero di decisione dove ogni permutazione appare come una foglia raggiungibile.

Consideriamo un albero di altezza h con l foglie raggiungibili che corrisponde a un ordinamento per confronti di n elementi.

Poiché ciascuna delle $n!$ permutazioni dell'input compare in una foglia, si ha $n! \leq l$. Dal momento che un albero binario di altezza h non ha più di 2^h foglie, si ha

$n! \leq l \leq 2^h$

Prendendo i logaritmi, questa relazione implica che

$h \geq lg(n!)$
$\ \ = \Omega(n\ lg\ n)$

> **Corollario 8.2**:
> Heapsort e merge sort sono algoritmi di ordinamento per confronti asintoticamente ottimali.

Capitolo 11 - Hashing [senza dimostrazioni]

Una tavola hash è una struttura dati efficace per implementare dizionari.

La ricerca nel caso peggiore è $\Theta(n)$, ma sotto ipotesi ragionevoli il tempo medio è $O(1)$.

Nella struttura array ho indirizzamento diretto: posso accede a qualunque elemento in tempo $O(1)$ perchè ho una chiave memorizzata per ogni elemento dell'array.

Quando voglio memorizzare un numero minore di chiavi, la tabella hash è molto utile. Tipicamente usa un array di dimensione proporzionale al numero di chiavi effettivamente memorizzate, e usa la chiave per *calcolare* l'indice dell'elemento.

L'hashing è una tecnica molto pratica ed efficace, tipicamente richiede un tempo $O(1)$ per eseguire le operazioni fondamentali sui dizionari (insert, search, delete).

11.1 - Tavole a indirizzamento diretto

L'indirizzamento diretto è una tecnica semplice che funziona bene quando l'universo U delle chiavi è ragionevolmente piccolo.

Se ho bisogno di un insieme dinamico in cui ogni elemento ha una chiave esatta dall'universo $U = \{0, 1, ..., m-1\}$, e in cui due elementi non possono avere la stessa chiave, posso usare un array o **tavola a indirizzamento diretto**, che indichiamo con $T[0...m-1]$. Ogni posizione o **cella** corrisponde a una chiave dell'universo U.

Ogni cella k punta ad un elemento dell'insieme con chiave k. Se l'insieme non contiene l'elemento con chiave k, allora $T[k] = NIL$.

Le operazioni sono semplici da implementare:

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

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

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

Ciascuna delle operazioni richiede tempo $O(1)$.

In certi casi possiamo memorizzare i dati direttamente nelle celle, invece di avere un puntatore ad un oggetto esterno alla tabella. In questo caso dobbiamo prevedere un modo per indicare che una cella è vuota.

11.2 - Tavole Hash con concatenamento

Quando l'insieme K delle chiavi memorizzate in un dizionario è molto più piccolo dell'universo U di tutte le chiavi possibili, una tavola hash richiede molto meno spazio di una tavola a indirizzamento diretto.

Con l'indirizzamento diretto, un elemento di chiave k è memorizzato nella cella k.
Con l'hashing, questo elemento è memorizzato nella cella $h(k)$, cioè utilizziamo una **funzione hash** h per calcolare la cella della chiave k.

Qui h associa l'universo U delle chiavi alle celle di una **tavola hash** $T[0...m-1]$:

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

dove la dimensione m della tavola hash è generalmente molto più piccola di $|U|$.

Diciamo che un elemento con chiave k viene mappato nella cella $h(k)$, o anche che $h(k)$ è il **valore hash** della chiave k.

Un problema che sorge è la **collisione**, quando due chiavi vengono mappate nella stessa cella.

La soluzione ideale è evitare le collisioni, scegliendo un'opportuna funzione h. Si potrebbe scegliere in modo che sembri casuale, ma poiché $|U| > m$, ci saranno almeno due chiavi che hanno steso valore hash, quindi evitare completamente le collisioni è impossibile.

Risoluzione delle collisioni mediante concatenamento

Nel **concatenamento** poniamo tutti gli elementi che sono associati alla stessa cella in una lista concatenata.

Le operazioni per concatenamento sono facili da implementare:

```
DIRECT-ADDRESS-SEARCH(T,k)
    ricerca un elemento con chiave k nella lista T[h(k)]

DIRECT-ADDRESS-INSERT(T,x)
    inserisce  in testa alla lista T[h(x,key)]

DIRECT-ADDRESS-DELETE(T,x)
    cancella x dalla lista T[h(x,key)]
```

L'inserimento nel caso peggiore è $O(1)$; si suppone che l'elemento da inserire non sia già presente nella tavola, altrimenti bisognerebbe cercare un'elemento di chiave $x.key$ prima di inserire.

Per la ricerca, il tempo di esecuzione al caso peggiore è proporzionale alla lunghezza della lista.

Per l’eliminazione, se usiamo liste doppiamente concatenate l’eliminazione è più rapida; se la lista è singolarmente concatenata ho bisogno di più puntatori per poter aggiornare l’attributo $next$ dell’elemento precedente.

Analisi dell’hashing con concatenamento

Data una tavola hash T, con m celle, dove sono memorizzati n elementi, definiamo il **fattore di carico** α della tavola T il rapporto n/m, ossia il numero medo di elementi memorizzati in una lista.

Il comportamento nel **caso peggiore** è pessimo: tutte le chiavi n sono associate alla stessa cella, creando una lista di lunghezza n. Il tempo è quindi $\Theta(n)$ più il tempo per calcolare la funzione hash.

Le prestazioni nel **caso medio** dipendono dal modo in cui la funzione hash h distribuisce mediamente l’insieme delle chiavi da memorizzare.
Supponiamo che qualsiasi elemento abbia la stessa probabilità d essere mandato in una qualsiasi delle m celle; questa ipotesi è detta **hashing uniforme semplice**.

Per $j = 0, 1, ...m-1$, indicando con n_j la lunghezza della lista $T[j]$, avremo
$n = n_0 + n_1 + ... + n_{m-1}$
e il valore atteso di n_j sarà $E[n_j] = \alpha = n/m$.

Supponiamo che basti un tempo $O(1)$ per calcolare il valore hash $h(k)$, in modo che il tempo richiesto per cercare un elemento con chiave k dipenda linearmente della lunghezza $n_{h(k)}$ della lista $T[h(k)]$.
Mettendo assieme in un tempo $O(1)$ il tempo richiesto per calcolare la funzione hash e accedere alla cella $h(k)$, consideriamo il numero atteso di elementi esaminati dall’algoritmo di ricerca, ovvero il numero di elementi nella lista $T[h(k)]$ che vengono controllati per vedere se le loro chiavi sono uguali a k.

Considereremo due casi: nel primo la ricerca non ha successo (nessun elemento nella tavola ha chiave k), nel secondo la ricerca ha successo e viene trovato un elemento con chiave k.

Ricerca senza successo

> **Teorema 11.1**
> In una tavola hash le cui collisioni sono risolte con il concatenamento, una ricerca senza successo richiede un tempo $\Theta(1+\alpha)$ nel caso medio, nell’ipotesi di hashing uniforme semplice.

Dimostrazione

Nell’ipotesi di hashing uniforme semplice, qualsiasi chiave k non ancora memorizzata nella tavola ha la stessa probabilità di essere associata a una qualsiasi delle m celle.

Il tempo atteso per ricercare senza successo una chiave k è il tempo atteso per svolgere le ricerche fino alla fine della lista $T[h(k)]$, che ha una lunghezza attesa pari a $E[n_{h(k)}] = \alpha$. Quindi, il numero atteso di elementi esaminato in una ricerca senza successo è α e il tempo totale richiesto (incluso quello per calcolare $h(k)$) è $\Theta(1+\alpha)$

Ricerca con successo

> **Teorema 11.2**
> In una tavola hash in cui le collisioni sono risolte con il concatenamento, una ricerca con successo richiede un tempo $\Theta(1+\alpha)$ nel caso medio, nell’ipotesi di hashing uniforme semplice

Dimostrazione

Supponiamo che l’elemento da ricercare abbia la stessa probabilità di essere uno qualsiasi degli n elementi inseriti nella tavola.

Il numero di elementi esaminati durante una ricerca con successo di un elemento x è uno in più del numero di elementi che si trovano prima di x nella lista di x.

Gli elementi che precedono x nella lista sono stati inseriti tutti dopo di x, perchè i nuovi elementi vengono posti all'inizio della lista.
Per trovare il numero atteso di elementi esaminati, prendiamo la media, sugli n elementi della tavola, di 1 più il numero atteso di elementi aggiunti alla lista di x dopo che x è stato aggiunto alla lista.

Indichiamo con x_i l'i-esimo elemento inserito nella tavola, per $1 = 1, 2, ..., n$ e sia $k_i = x_i.key$.
Per le chiavi k_i e k_j, definiamo la variabile casuale indicatrice $X_{ij} = I\{h(k_i) = h(k_j)\}$.

Nell'ipotesi di hashing uniforme semplice, abbiamo $Pr\{h(k_i) = h(k_j)\} = 1/m$ e, quindi, $E[X_{ij}] = 1/m$.

Dunque, il numero atteso di elementi esaminati in una ricerca con successo è:

$$E[\frac{1}{n}\sum_{i=n}^{n}(1 + \sum_{j=i+1}^{n} X_{ij})]$$

Dove noi stiamo calcolando la media, su ogni elemento inserito nella tabella hash, del numero di elementi visitati prima di trovare l'elemento x_i. Il numero di elementi visitati prima di trovare x_i lo calcoliamo sommando 1 a tuti gli elementi che sono stati inseriti nella tabella hash dopo x_i, ovvero guardiamo su tutti gli elementi inseriti dopo x_i quanti hanno probabilità di essere mappati sulla stessa cella di x_i, ovvero $Pr\{h(k_i) = h(k_j)\}$.

Calcoliamo:

$$\begin{aligned}
&E[\frac{1}{n}\sum_{i=n}^{n}(1 + \sum_{j=i+1}^{n} X_{ij})] \\
&\qquad = \frac{1}{n}\sum_{i=n}^{n}(1 + \sum_{j=i+1}^{n} E[X_{ij}]) \\
&\qquad = \frac{1}{n}\sum_{i=n}^{n}(1 + \sum_{j=i+1}^{n} \frac{1}{m}) \\
&\qquad = \frac{1}{n}n + \frac{1}{n}\sum_{i=n}^{n}(\sum_{j=1+i}^{n} \frac{1}{m}) \\
&\qquad = 1 + \frac{1}{n}\sum_{i=1}^{n}\frac{n-i}{m} \\
&\qquad = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&\qquad = 1 + \frac{1}{nm}(\sum_{i=1}^{n}n - \sum_{i=1}^{n}i) \\
&\qquad = 1 + \frac{1}{nm}(n^2 - \frac{n(n+1)}{2}) \\
&\qquad = 1 + \frac{n-1}{2m} \\
&\qquad = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}
\end{aligned}$$

In conclusione, il tempo totale richiesto per una ricerca con successo (incluso il tempo per calcolare la funzione hash) è $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$

Questa analisi ci aiuta ad osservare:

Se il numero di celle nella tavola hash è almeno proporzionale al numero di elementi della tavola, abbiamo $n = O(m)$ e, di conseguenza, $\alpha = n/m = O(m)/m = O(1)$.

Pertanto l'inserimento richiede il tempo $O(1)$ nel caso peggiore quando le liste sono doppiamente concatenate, tutte le operazioni di dizionario possono essere svolte, in media, nel tempo $O(1)$.

11.3 - Funzioni hash

Un buona funzione hash soddisfa (approssimativamente) l'ipotesi dell'hashing uniforme semplice: ogni chiave a la stessa probabilità di essere mandata in una qualsiasi delle m celle, indipendentemente dalla cella in cui viene mandata qualsiasi altra chiave.
Questa condizione di solito non si può verificare, perchè raramente è nota la distribuzione delle probabilità.

Nella pratica si usano delle tecniche euristiche per realizzare funzioni hash con buone prestazioni.

Per esempio, se come chiavi abbiamo valori simili, come `pt` e `pts`, vogliamo ridurre al minimo la probabilità che queste due chiavi siano mappate sulla stessa cella.
In questo caso cerchiamo di derivare il valore hash in modo che sia indipendente da qualsiasi regolarità che ci possa essere nei dati.

Il **metodo della divisione** calcola il valore hash come il resto della divisione fra la chiave e un determinato numero primo.

Notiamo che alcune applicazioni delle funzioni hash possano richiedere proprietà più vincolanti dell'hashing uniforme semplice.

La maggior parte delle funzioni hash richiede che l'universo delle chiavi si l'insieme dei numeri naturali $\mathbb{N}$. Quindi, se le chiavi non sono numeri naturali conviene interpretarle come tali; da qui in avanti supponiamo che le chiavi siano numeri naturali.

Il metodo della divisione

Quando si applica il **metodo della divisione** per creare una funzione hash, una chiave k viene associata a una delle m celle prendendo il resto della divisione fra k e m; cioè la funzine hash è:

$h(k) = k \ mod \ m$

É un metodo veloce perchè richiede solo un'operazione.

Di solito per questo metodo evitiamo certi valori di m, come $m = 2^p$ o $m = 2^p - 1$ perchè non distribuiscono bene le chiavi sulle celle. Vogliamo che la funzione hash sia dipendente da tutti i bit della chiave.

Una numero primo non troppo vicino a una potenza di due di solito è una buona scelta.

Se, per esempio, vogliamo una tabella hash che contiene circa $n = 2000$ elementi, e vogliamo visitare in media 3 elementi in una ricerca senza successo, possiamo scegliere $m = 701$ perchè è un numero primo vicino a $2000/3$, ma non a una potenza di due.

Il metodo della moltiplicazione

Il **metodo della moltiplicazione** per creare funzioni hash si svolge in due passi:

1. Moltiplichiamo la chiave k per un valore A nell'intervallo $0 < A < 1$ ed estraiamo la parte frazionaria di kA, ovvero calcoliamo $kA - \lfloor kA \rfloor$
2. Moltiplichiamo questo valore per m e prendiamo la parte intera inferiore del risultato.

In sintesi la funzione hash è $h(k) = \lfloor m(kA \ mod \ 1) \rfloor$ dove "$k \ A \ mod \ 1$" rappresenta la parte frazionario di A.

Il valore di m non è critico, tipicamente lo scegliamo come una potenza di 2.

Supponiamo che la dimensione della parola della macchina sia w bit e che k entri in una sola parola. Come A prendiamo la frazione della forma $s/2^w$, dove s è un intero nell'intervallo $0 < s < 2^w$.

Moltiplichiamo k per $s = A\,2^w$, il risultato sarà un numero di $2w$ bit che rappresentiamo come $r_1 2^w + r_0$. Il valore hash desiderato di p bit è formato dai p bit più significativi di r_0.

Questo metodo funziona con qualunque valore A, tuttavia con alcuni valori funziona meglio che con altri, per esempio Kuth propone $A \approx (\sqrt{5} - 1)/2 = 0,618033...$

11.4 - Indirizzamento aperto

Tutti gli elementi sono memorizzati nella tavola hash stessa; ovvero ogni cella contiene un elemento (nel nostro esempio elemento = chiave) o la costante `NIL`.
Diversamente dal concatenamento, non ci sono liste nè elementi memorizzati all'interno della tavola.

Nell'indirizzamento aperto la tabella può riempirsi senza poter accettare altri più inserimenti, quindi il fattore di carico α è al massimo 1.
Anziché seguire i puntatori, calcoliamo la sequenza delle celle da esaminare; la memoria extra liberata per non aver memorizzato i puntatori offre alla tavola hash un maggior numero di celle a parità di memoria occupata, in modo da potenzialmente ridurre le collisioni.

Per effettuare un inserimento, esaminiamo in successione le posizioni della tavola hash (**ispezione**), finché non troviamo una cella vuota in cui inserire la chiave.
Anziché seguire sempre lo stesso ordine $0, 1, ..., m-1$ (che richiede un tempo di ricerca $\Theta(n)$), la sequenza delle posizioni esaminate *dipende dalla chiave da inserire*.

Estendiamo la funzione hash in modo da includere l'ordine di ispezione, a partire da 0, come secondo input.

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

Con l'indirizzamento aperto si richiede che, per ogni chiave k, la **sequenza di ispezione**

$\langle h(k,0), h(k,1), .., h(k, m-1) \rangle$

sia una permutazione di $\langle 0, 1, ..., m-1 \rangle$, in modo che ogni posizione della tavola hash possa essere considerata come possibile cella in cui inserire una chiave mentre la tavola si riempie.

Supponiamo che gli elementi della tavola hash T siano chiavi senza dati satelliti.

```
HASH-INSERT(T,k)
1     i = 0
2     repeat
3         j=h(k,i)
4         if T[j] == NIL
5             T[j] = k
6             return j
7         else i = i + 1
8     until i == m
9     error "overflow della tavola hash"
```

L'algoritmo che ricerca la chiave k esemina la stessa sequenza di celle che ha esaminato l'algoritmo di inserimento quando ha inserito la chiave k, quindi la ricerca può terminare (senza successo) quando trovo una cella vuota.
Presuppone che le chiavi non vengano cancellate dalla tabella hash.

```
HASH-SEARCH(T,k)
1     i = 0
2     repeat
3         j = h(k,i)
4         if T[j] == k
5             return j
6         i = i + 1
```

```
7     until T[j] == NIL or i == m
8     return NIL
```

La cancellazione è più complicata; quando eliminiamo un elemento non possiamo scrivere nella tabella `NIL`, perchè potrebbe portare al fallimento di operazioni di ricerca per altre chiavi.
Una soluzione è marcare la cella con un valore speciale `DELETED`, anzichè `NIL`.

Dovremmo dunque modificare `HASH-INSERT` in modo da poter scrivere sulle celle `DELETED`; nessuna modifica è richiesta per `HASH-SEARCH`.

Quando usiamo il valore `DELETED`, i tempi di ricerca non dipendono più dal fattore di carico α; per questo motivo se vogliamo effettuare delle operazioni di cancellazione si preferisce il concatenamento.

Nella nostra analisi facciamo l'ipotesi di **hashing uniforme ,**che estende l'hashing uniforme semplice a funzioni hash che non producono un solo numero, ma una sequenza di valori.
Supponiamo che ogni chiave abbia la stessa probabilità di avere come sequenza di ispezione una delle $m!$ permutazioni di $\langle 0, 1, ..., m-1 \rangle$.

É difficile implementare il vero hashing uniforme; in pratica si usano delle approssimazioni accettabili.

Ispezione lineare

Data una funzione hash ordinaria $h' : U \to \{0, 1, ..., m-1\}$, che chiamiamo **funzione hash ausiliaria**,il metodo dell'**ispezione lineare** usa la funzione hash

$h(k, i) = (h'(k) + i) \ mod \ m$ per $i = 0...m-1$

La prima cella esaminata sarà $T[h'(k)]$, la seconda $T[h'(k) + 1]$ e così via.

Poiché l prima cella determina l'intera sequenza di ispezione, ci sono soltanto m sequenze distinte.

É facile da implementare ma presenta il problema dell'**addensamento primario**, dove si formano lunghe file di celle occupate che aumentano il tempo medio di ricerca.

Ispezione quadratica

Usa una funzione hash della forma

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \ mod \ m$ dove h' è una funzione hash usiliaria, c_1 e $c_2 \neq 0$ sono costanti ausiliarie e $i = 0, 1, ..., m-1$.

La posizione iniziale esaminata è $T[h'(k)]$; le posizioni successivamente esaminate sono distanziate da quantità che dipendono in modo quadratico dal numero d'ordine di ispezione i.

Per usare tutta la tavola T i valori di c_1, c_2 e m non si possono scegliere arbitrariamente.

Inoltre, se due chiavi hanno stessa posizione iniziale, anche le loro sequenze sono identiche, perchè $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$. Questa proprietà porta ad una forma più lieve di addensamento, che chiameremo **addensamento secondario**.

Doppio hashing

Usa una funzione della forma

$h(k, i) = (h_1(k) + i h_2(k)) \ mod \ m$ dove h_1 e h_2 sono funzioni ausiliarie.

L'ispezione inizia in $T[h_1(k)]$; le successive posizioni sono distanziate dalle precedenti posizioni di una quantità $h_2(k)$, modulo m.
Quindi la sequenza di ispezione dipende in due modi dalla chiave k.

Per poter visitare tutta la tavola hash, i valori generati da h_1 e h_2 devono essere indipendenti.

Quando m è primo o una potenza di due, il doppio hashing è migliore delle ispezioni lineari e quadratiche, in quanto usa $\Theta(n^2)$ sequenze di ispezione, anziché $\Theta(m)$.

Analisi dell'hashing a indirizzamento aperto

Esprimiamo questa analisi in termini del fattore di carico $\alpha = n/m$. Visto che utilizziamo l'indirizzamento aperto, abbiamo al massimo un elemento per cella, quindi $n \leq m$ che implica $\alpha \leq 1$.
Supponiamo di applicare l'hashing uniforme, da cui abbiamo che la sequenza di ispezione relativa ad una chiave k ha la stessa probabilità di essere una qualunque permutazione di $\langle 0, 1, ..., m-1 \rangle$; oltretutto ogni possibile sequenza di ispezione è ugualmente probabile.

Analizziamo il numero di ispezioni atteso in una ricerca senza successo:

Teorema 11.6
Nell'ipotesi di hashing uniforme, data una tavola hash a indirizzamento aperto con un fattore di carico $\alpha = n/m < 1$, il numero atteso di ispezioni in una ricerca senza successo è al massimo $\frac{1}{1-\alpha}$.

Dimostrazione

In una ricerca senza successo, ogni ispezione, tranne l'ultima, accede ad una cella occupata che non contiene la chiave desiderata, e l'ultima cella esaminata è vuota.

Definiamo una variabile casuale X come il numero di ispezioni fatte in una ricerca senza successo; definiamo inoltre A_i, con $i = 1, 2...$ come l'evento in cui l'i-esima ispezione viene eseguita e trova una cella occupata.

Allora l'evento $\{X \geq 1\}$ è l'intersezione degli eventi $A_1 \cap A_2 \cap ... \cap A_{i-1}$.

$Pr\{A_1 \cap A_2 \cap ... \cap A_{i-1}\} = Pr\{A_1\} \cdot Pr\{A_2|A_1\} \cdot ... \cdot Pr\{A_{i-1}|A_1 \cap ... \cap A_{i-2}\}$

Poichè ci sono n elementi ed m celle, $Pr\{A_1\} = n/m$.
Per $j > 1$, la probabilità che la j-esima ispezione trovi una cella occupata, dopo che le prime $j-1$ ispezioni hanno trovato celle occupate, è $\frac{n-j+1}{m-j+1}$.
Questa probabilià deriva dal fatto che dovremmo trovare uno dei restanti $n-(j-1)$ elementi in una delle $m-(j-1)$ celle non ancora esaminate e, per l'ipotesi di hashing uniforme, la probabilità è il rapporto di queste quantità.

Ossevando che $n < m$ implica che $\frac{n-j}{m-j} \leq \frac{n}{m}$ per ogni j $t.c.$ $0 \leq j < m$, allora per ogni i $t.c.$ $1 \leq i \leq m$ si ha:

$$\begin{aligned} Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot ... \cdot \frac{n-i+2}{m-i+2} \\ &\leq (\frac{n}{m})^{i-1} \\ &= \alpha^{i-1} \end{aligned}$$

Adesso limitiamo il numero atteso di ispezioni:

$$\begin{aligned} E[X] &= \sum_{i=1}^{\infty} Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} \end{aligned}$$

Questo limite per $\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + ...$ ha un'interpretazione intuitiva.
Una prima ispezione viene sempre fatta. Con una probabilità approssimativamente pari ad α, la prima ispezione trova la cella occupata, quindi occorre effettuare una seconda ispezione e così via.

Se α è una costante, il teorema indica che una ricerca senza successo viene eseguita nel tempo $O(1)$.

Analizziamo il numero di ispezioni atteso per l'inserimento:

Corollario 11.7
L'inserimento di un elemento in una tavola hash a indirizzamento aperto con un fattore di carico α richiede in media non più di $\frac{1}{1-\alpha}$ ispezioni, nell'ipotesi di hashing uniforme.

Dimostrazione

Un elemento viene inserito soltanto de c'è spazio nella tavola e, quindi, $\alpha < 1$. L'inserimento di una chiave richiede una ricerca senza successo seguita dalla sisteazione della chiave nella prima cella vuota che vien trovata. Quindi, il numero atteso di ispezioni è al massimo $\frac{1}{1-\alpha}$.

Analizziamo il numero di ispezioni atteso in una ricerca con successo:

Teorema 11.8
Data una tavola hash a indirizzamento aperto con un fattore di carico $\alpha < 1$, il numero atteso di ispezioni di una ricerca con successo è al massimo
$\frac{1}{\alpha} ln \frac{1}{1-\alpha}$
supponendo che l'hashing sia uniforme e che ogni chiave nella tavola abbiamo la stessa probabilità di essere cercata.

Dimostrazione

La ricerca di una chiave k segue la stessa sequenza di ispezione che è stata seguita quando è stato inserito l'elemento con chiave k.

Per il corollario precedente, se k era la $(i+1)$-esima chiave inserita nella tavola hash, il numero atteso di ispezioni fatte in una dricerca di k è al massimo $\frac{1}{1-i/m} = \frac{m}{m-i}$.

Calcolando la media su tutte le n chiavi della tavola hash, si ottiene un numero medio di ispezioni durante una ricerca con successo:

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\leq \frac{1}{\alpha}\int_{m-n}^{m}(1/x)dx \\
&= \frac{1}{\alpha} ln \frac{m}{m-n} \\
&= \frac{1}{\alpha} ln \frac{1}{1-\alpha}
\end{aligned}$$

CGGR: Alberi binari

Algoritmi ricorsivi su alberi binari

Gli alberi binari sono definiti in maniera ricorsiva, quindi si adattano bene al paradigma divide et impera.

Sono caratterizzati dalla **dimensione** n, che corrisponde al numero di nodi contenuti nell'albero; un albero con n nodi ha esattamente $n-1$ archi.

La dimensione può essere definita in maniera ricorsiva: un albero vuoto ha dimensione 0, mentre la dimensione di un albero non vuoto è pari alla somma dei suoi sottoalberi, incrementata di 1, per includere la radice.

```
Dimensione(u)
1   if u == NULL
2       return 0
3   else
4       dimSX = Dimensione(u.sx)
5       dimDX = Dimensione(u.dx)
6       return dimSX + dimDX + 1
```

Teorema
La dimensione n di un albero binario può essere calcolata in tempo $O(n)$.

Dimostrazione

Osserviamo che il problema della dimensione è un *problema decomponibile*, ovvero può essere risolto con uno schema di tipo divide et impera.

```
Decomponibile(u)
1   if u == NULL
2       return Decomponibile(NULL)
3   else
4       risultatoSX = Decomponibile(u.sx)
5       risultatoDX = Decomponibile(u.dx)
6       return Ricombina(risultatoSX, risultatoDX)
```

Il fatto di avere un algoritmo di tipo divide et impera ci permette di poter scrivere la relazione di ricorrenza:

ipotizziamo che il costo di divisione e ricombinazione sia una costante c. Preso un nodo u e il suo sottoalbero con n nodi (incluso u), ipotizziamo di avere $r-1$ nodi che discendono dal figlio sinistro e, quindi, $n-1$ che discendono da quello destro, dove $1 \leq r \leq n$ e c_0, c sono costanti positive:

$$T(n) \leq \begin{cases} c_0 & se\ n \leq 1 \\ T(r-1) + T(n-r) + c & altrimenti \end{cases}$$

Questa relazione ha soluzione $T(n) = O(n)$; dimostriamolo.

Non potendo applicare il teorema dell'esperto, osserviamo che vale $T(0) \leq c_0 \leq c'$ e dimostriamo per induzione che $T(n) \leq 3c'n$ per ogni $n \geq 1$, dove $c' = max\{c_0, c\}$.

Se $n = 1$, ovvero l'albero contiene un nodo solo, abbiamo $r = 1$ e, quindi:

$$T(1) \leq 2T(0) + c \leq 2c_0 + c \leq 3c' = 3c'n$$

Supponiamo che l'affermazione sia vera per $1 \leq n' < n$. Allora:

$$T(n) \leq T(r-1) + T(n-r) + c$$

e, se $1 < r < n$, per ipotesi induttiva abbiamo che

$T(n) \leq 3c'(r-1) + 3c'(n-r) + c \leq 3c'n - 2c < 3c'n$

Se invece $r = 1$, utilizziamo il fatto che $T(0) \leq c'$ e applichiamo l'ipotesi induttiva su $T(n-1) \leq 3c'(n-1)$, ottenendo:

$T(n) \leq c' + 3c'(n-1) + c \leq 3c'n - c < 3c'n$

Lo stesso ragionamento vale se $r = n$.

In cnclusione, $T(n) = O(n)$ e il teorema risulta essere dimostrato.

Altezza di un albero

Ricordiamo che l'altezza di un albero misura la massima distanza di una foglia dalla radice dell'albero, in termini di numero di archi attraversati.

```
Altezza(u)
    if u == NULL
        return -1
    else
        altezzaSX = Altezza(u.sx)
        altezzaDx = Altezza(u.dx)
        return max(altezzaSX, altezzaDX) + 1
```

Sia questo algoritmo sia quello per la dimensione, hanno come caso base l'albero vuoto e come passo induttivo l'albero non vuoto.

Poichè usano un approccio divide et impera, ogni nodo viene attraversato un numero costante di volte, quindi l'esecuzione richiede un tempo di $O(n)$.

Visite di Alberi

Lo schema ricorsivo permette di effettuare la **visita** a partire dalla sua radice, ovvero esaminare tutti i nodi in modo sistematico.

- **Visita anticipata**(*preorder*): stampa l'elemento contenuto nel nodo, visita ricorsivamente il sottoalbero sinistro, visita ricorsivamente il sottoalbero destro.
- **Visita simmetrica**(*inorder*): visita ricorsivamente il sottoalbero sinistro, stampa l'elemento contenuto nel nodo, visita ricorsivamente il sottoalbero destro.
- **Visita posticipata**(*postorder*): visita ricorsivamente il sottoalbero sinistro, visita ricorsivamente il sottoalbero destro, stampa l'elemento contenuto nel nodo.

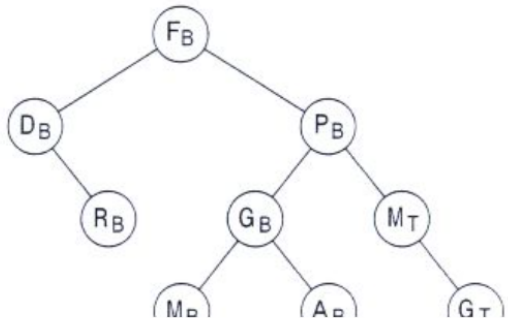
Possiamo dimostrare che il costo di ciascuna delle tre visite è $O(n)$ per un albero di dimensione n, in modo analogo alla dimostrazione del costo del calcolo della dimensione.

Notiamo in oltre che non abbiamo return in quanto non abbiamo bisogno di valori che si propagano tra le chiamate ricorsive.

```
Anticipata(u)
    if u != NULL
        print u.dato
        Anticipata(u.sx)
        Anticipata(u.dx)
```

Per ottenere le visite simmetrica e posticipata, spostiamo l'istruzione di stampa in una delle sue righe successive.

Per apprezzare la differenza delle tre visite, consideriamo l'esempio mostrato nella parte sinistra della seguente figura.



anticipata:
F_B D_B R_B P_B G_B M_B A_B M_T G_T
simmetrica:
D_B R_B F_B M_B G_B A_B P_B M_T G_T
posticipata:
R_B D_B M_B A_B G_B G_T M_T P_B F_B
ampiezza:

Alberi completamente bilanciati

Un albero binario è **completo** se ogni nodo interno ha esattamente due figli non vuoti.
L'albero è **completamente bilanciato** se, oltre ad essere completo, tutte le foglie hanno la stessa profondità.

Un albero completamente bilanciato di altezza h ha quindi $2^h - 1$ nodi interni e 2^h foglie: ne deriva che la relazione tra altezza h e numero di nodi $n = 2^{h+1} - 1$ è:
$h = log(n + 1) - 1$

In un albero binario **bilanciato** vale la relazione $h = O(log\ n)$.

Notiamo che un albero binario completamente bilanciato è bilanciato, ma non è sempre vero il viceversa.

Indicati come u_S e u_D i figli di u, $T(u)$ è completamente bilanciato se e solo se $T(u_S)$ e $T(u_D)$ sono completamente bilanciati e hanno stessa altezza.

Nel codice restituiamo una coppia di valori, un booleano che ci indica se l'albero è bilanciato, e la sua altezza.

```
CompletamenteBilanciato(u)
    if u == NULL
        return <TRUE, -1>
    else
        <bilSX, altSX> = CompletamenteBilanciato(u.sx)
        <bilDX, altDX> = CompletamenteBilanciato(u.dx)
        bilCorr = bilSX && bilDX && (altSX == altDX)
        altCorr = max(altSX, altDX) + 1
        return <bilCorr, altCorr>
```

Nodi cardine di un albero binario

Dato un nodo u, sia p_u la sua profondità e h_u la sua altezza. Diciamo che u è un nodo **cardine** se e solo se $p_u = h_u$.

Progettiamo un algoritmo ricorsivo che stampa il contenuto di tutti i nodi cardine; scegliamo come valore restituito dalla chiamata h_u e passiamo p_u come parametro (insieme al nodo).

Inizialmente i parametri saranno la radice r e la sua profondità $p_r = 0$

```
Cardine(u,p)
    if (u == NULL)
        return -1
    else
        altSX = Cardine(u.sx, p+1)
```

```
        altDX = Cardine(u.dx, p+1)
        altezza = max(altSX, altDX) + 1
        if p == altezza
            print u.dato
        return altezza
```

La complessità resta $O(n)$ perchè si tratta di una variazione della visita posticipata.