



NoiStudiamo

DISPENSA DI LABORATORIO



Laboratorio di Algoritmi

Algoritmi e Laboratorio, modulo Laboratorio: notazioni asintotiche, ricorsione, equazioni di ricorrenza, correttezza e dimostrazioni.

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

I capitoli della dispensa, nell'ordine in cui conviene studiarli.

1. Analizzare un algoritmo	3
2. Notazioni asintotiche	6
3. La ricorsione	12
4. Equazioni di ricorrenza	18
5. Analisi di algoritmi ricorsivi notevoli	25
6. Ottimizzazione con la ricorsione	28
7. Analisi dell'hashing	33
8. Alberi rosso neri: altezza e correttezza	37
9. Programmazione dinamica: sottostruttura ottima e LCS	41
10. Strategia greedy: le dimostrazioni	46
11. Grafi: correttezza di ordinamento topologico e cammini minimi	50

Analizzare un algoritmo

Che cosa si misura quando si dice che un algoritmo è efficiente, come si conta il costo di uno pseudocodice riga per riga e come si dimostra che un algoritmo è corretto con gli invarianti di ciclo.

Perché analizzare

Analizzare un algoritmo significa **prevedere** le risorse che userà, prima di eseguirlo. La risorsa più importante è il tempo di calcolo; a volte conta anche la memoria. Il tempo misurato con un cronometro però dipende da troppe cose che non riguardano l'algoritmo: il processore, il linguaggio, il compilatore, gli altri programmi in esecuzione. Serve una misura astratta, uguale su ogni macchina.

IL MODELLO DI CALCOLO

Si usa il modello **RAM** (Random Access Machine): un calcolatore ideale con un solo processore, in cui le istruzioni vengono eseguite una dopo l'altra e ogni **operazione elementare** costa un tempo costante. Sono elementari le operazioni aritmetiche (somma, sottrazione, moltiplicazione, divisione, resto), i confronti, gli assegnamenti, l'accesso a un elemento di un array, la chiamata e il ritorno da una funzione. Non sono elementari, ovviamente, operazioni come "ordina l'array" o "trova il massimo": vanno scomposte.

Il tempo di esecuzione su un ingresso è il **numero di operazioni elementari** eseguite. Poiché cresce con la dimensione dell'ingresso n , lo si descrive come una funzione $T(n)$.

CASO PEGGIORE, MIGLIORE E MEDIO

Anche a parità di n , ingressi diversi possono richiedere tempi diversi: Insertion Sort è velocissimo su un array già ordinato e lento su uno ordinato al contrario. Si distinguono quindi:

- il **caso peggiore**: il tempo massimo su tutti gli ingressi di dimensione n . È il più usato, perché è una **garanzia**: l'algoritmo non andrà mai peggio di così;
- il **caso migliore**: il tempo minimo. Serve poco, perché un algoritmo può essere veloce su un caso fortunato e pessimo su tutti gli altri;
- il **caso medio**: il tempo atteso, supponendo una distribuzione di probabilità sugli ingressi (per esempio tutti gli ordinamenti equiprobabili). È spesso il più realistico, ma richiede un'ipotesi sui dati.

DA RICORDARE

Quando non si specifica altro, "il tempo di esecuzione" di un algoritmo è quello nel **caso peggiore**.

Analisi riga per riga: Insertion Sort

Analizziamo Insertion Sort in dettaglio, assegnando a ogni riga un costo costante c_k e contando quante volte viene eseguita. Indichiamo con t_j il numero di volte che il test del ciclo while viene eseguito per un certo valore di j .

1	InsertionSort(A, n)	costo	volte
2	for j = 2 to n do	c1	n
3	chiave = A[j]	c2	n - 1
4	i = j - 1	c3	n - 1
5	while i > 0 and A[i] > chiave do	c4	somma di t _j
6	A[i+1] = A[i]	c5	somma di (t _j - 1)
7	i = i - 1	c6	somma di (t _j - 1)
8	A[i+1] = chiave	c7	n - 1

Il test del for viene eseguito n volte (una in più delle iterazioni, quella che fa uscire dal ciclo). Il test del while viene eseguito t_j volte per ogni j, e il corpo del while una volta in meno. Sommando:

$$T(n) = c_1 n + (c_2 + c_3 + c_7)(n - 1) + c_4 \sum t_j + (c_5 + c_6) \sum (t_j - 1)$$

dove le somme vanno da j = 2 a j = n. Tutto dipende da t_j, cioè dall'ingresso.

Caso migliore (array già ordinato). Per ogni j il primo test trova A[j - 1] ≤ chiave, quindi t_j = 1 e il corpo del while non viene mai eseguito. T(n) diventa della forma a·n + b: una funzione **lineare**.

Caso peggiore (array ordinato al contrario). La chiave deve superare tutti gli elementi precedenti, quindi t_j = j. Usando le somme note

$$\sum_{j=2..n} j = n(n + 1)/2 - 1 \quad \sum_{j=2..n} (j - 1) = n(n - 1)/2$$

si ottiene una funzione della forma a·n² + b·n + c: **quadratica**.

Caso medio. In media metà degli elementi di A[1..j - 1] è maggiore della chiave, quindi t_j ≈ j/2: le somme si dimezzano ma restano quadratiche.

DA SAPERE: ORDINE DI GRANDEZZA

Delle costanti c_k e dei termini di grado più basso non ci interessa il valore esatto: per n grande conta solo il termine che cresce più in fretta (n² nel caso peggiore). È esattamente l'idea delle notazioni asintotiche del prossimo capitolo: si dirà che Insertion Sort ha tempo Θ(n²) nel caso peggiore e Θ(n) nel caso migliore.

Correttezza: gli invarianti di ciclo

Provare un algoritmo su qualche esempio non dimostra che sia corretto: dimostra solo che funziona su quegli esempi. Per gli algoritmi iterativi lo strumento standard è l'**invariante di ciclo**: una proprietà che è vera prima di ogni iterazione del ciclo.

DEFINIZIONE: INVARIANTE DI CICLO

Per dimostrare che un invariante è utile bisogna provare tre cose:

1. **Inizializzazione**: è vero prima della prima iterazione;
2. **Conservazione**: se è vero prima di un'iterazione, resta vero prima della successiva;
3. **Conclusione**: quando il ciclo termina, l'invariante (insieme alla condizione di uscita) dice che l'algoritmo ha prodotto il risultato corretto.

Le prime due fasi sono come una dimostrazione per induzione: il caso base e il passo induttivo. La terza è la differenza: nell'induzione si va avanti all'infinito, qui ci si ferma quando il ciclo finisce e si usa l'invariante per concludere. È come una catena di domino: basta far cadere il primo (inizializzazione) e sapere che ogni pezzo fa cadere il successivo (conservazione) per essere sicuri che cadranno tutti fino all'ultimo (conclusione).

ESEMPIO: CORRETTEZZA DI INSERTION SORT

Invariante. All'inizio di ogni iterazione del for, il sottoarray $A[1..j - 1]$ contiene gli elementi che stavano originariamente in $A[1..j - 1]$, ma ordinati.

1. **Inizializzazione.** Prima della prima iterazione $j = 2$, e $A[1..1]$ contiene un solo elemento, che è banalmente ordinato e coincide con quello originale.
2. **Conservazione.** Il corpo del for sposta di una posizione verso destra $A[j - 1]$, $A[j - 2]$, ... finché trova il posto per la chiave $A[j]$, e la inserisce lì. Ora $A[1..j]$ contiene gli elementi originali di $A[1..j]$, in ordine. Incrementare j ripristina l'invariante per il passo successivo. (Una dimostrazione completa richiederebbe un invariante anche per il while interno.)
3. **Conclusione.** Il ciclo termina quando $j = n + 1$. Sostituendo nell'invariante, $A[1..n]$ contiene gli elementi originali in ordine: l'intero array è ordinato.

UN SECONDO ESEMPIO: RICERCA DEL MASSIMO

```
1 Massimo(A, n)
2   m = A[1]
3   for i = 2 to n do
4       if A[i] > m then m = A[i]
5   return m
```

Invariante: all'inizio dell'iterazione i , m è il massimo di $A[1..i - 1]$.

1. **Inizializzazione:** con $i = 2$, $m = A[1]$ è il massimo di $A[1..1]$.
2. **Conservazione:** se m è il massimo di $A[1..i - 1]$, dopo il confronto con $A[i]$ m diventa il massimo tra il vecchio massimo e $A[i]$, cioè il massimo di $A[1..i]$.
3. **Conclusione:** all'uscita $i = n + 1$, quindi m è il massimo di $A[1..n]$.

DA SAPERE: COME SI SCRIVE UNA DIMOSTRAZIONE DI CORRETTEZZA

Si enuncia l'invariante in modo preciso (che cosa vale, su quale porzione dei dati, in quale momento del ciclo), poi si verificano nell'ordine inizializzazione, conservazione e conclusione. Nella conclusione si sostituisce nell'invariante il valore che la variabile del ciclo ha all'uscita. Per gli algoritmi **ricorsivi** lo strumento analogo è l'**induzione sulla dimensione** del problema: si verifica il caso base e si dimostra che, se le chiamate ricorsive sono corrette, lo è anche la chiamata corrente.

Analisi di algoritmi divide et impera

Per un algoritmo ricorsivo il tempo $T(n)$ si esprime in funzione del tempo sulle istanze più piccole: si ottiene un'**equazione di ricorrenza**. Se il problema viene diviso in a sottoproblemi di dimensione n/b , con un costo $D(n)$ per dividere e $C(n)$ per combinare, e un costo costante sotto una certa soglia:

$$T(n) = \Theta(1) \text{ se } n \leq c; \quad T(n) = a \cdot T(n/b) + D(n) + C(n) \text{ altrimenti}$$

Per Merge Sort $a = b = 2$, dividere costa $\Theta(1)$ (calcolare il punto di mezzo) e combinare costa $\Theta(n)$ (il Merge):

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

Come si risolvono queste equazioni è l'argomento dei capitoli sulla ricorsione e sulle ricorrenze.

Notazioni asintotiche

Il linguaggio per confrontare la velocità di crescita delle funzioni di costo: le funzioni di crescita più comuni, le notazioni Θ , O , Ω , o e ω con le loro definizioni formali, il criterio del limite e molti esempi svolti passo per passo.

Perché conta la crescita

Immagina due offerte per un lavoro di un mese. La prima ti paga 1000 euro al giorno. La seconda ti paga 1 centesimo il primo giorno, 2 il secondo, 4 il terzo, raddoppiando ogni giorno. Dopo una settimana la prima offerta ti ha dato 7000 euro, la seconda appena 1,27 euro: sembra non esserci partita. Ma al trentesimo giorno la seconda offerta ti paga, solo quel giorno, 2^{29} centesimi, cioè più di 5 milioni di euro.

La morale è che il **modo in cui una quantità cresce** conta più del suo valore iniziale. La prima offerta cresce in modo **lineare** ($1000 \cdot n$), la seconda in modo **esponenziale** (2^n), e prima o poi l'esponenziale supera qualunque lineare, per quanto grande sia la costante. Con gli algoritmi è uguale: per ingressi grandi, ciò che decide è il ritmo con cui cresce $T(n)$, non le costanti.

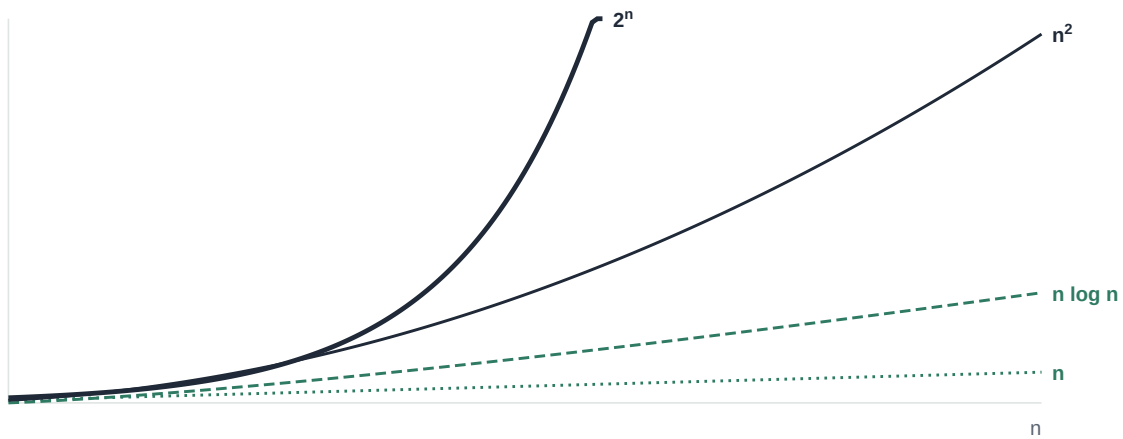
LE FUNZIONI DI CRESCITA PIÙ COMUNI

FUNZIONE	NOME	ESEMPIO TIPICO
1	costante	accedere a un elemento di un array
$\log n$	logaritmica	ricerca binaria
n	lineare	scorrere un array
$n \log n$	quasi lineare	Merge Sort, Heapsort
n^2	quadratica	Insertion Sort nel caso peggiore
n^3	cubica	Floyd Warshall
2^n	esponenziale	provare tutti i sottoinsiemi
$n!$	fattoriale	provare tutte le permutazioni

Ogni funzione della tabella cresce, per n grande, più lentamente di tutte quelle sotto di lei:

$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$$

Per avere un'idea: con $n = 1000$ si ha $\log_2 n \approx 10$, $n \log_2 n \approx 10^4$, $n^2 = 10^6$, mentre 2^n ha circa 300 cifre.



Andamento di alcune funzioni per n da 1 a 12: l'esponenziale parte piano ma esce presto dal grafico.

TERMINI DOMINANTI E COSTANTI

Se $T(n) = 3n^2 + 50n + 200$, per n piccolo pesano molto i termini $50n$ e 200 , ma già per $n = 100$ il termine $3n^2$ vale 30 000 contro $5000 + 200$. Al crescere di n i termini di grado più basso diventano trascurabili, e anche il fattore 3 non cambia la **forma** della crescita, solo la scala. Per descrivere la crescita di $T(n)$ si fa quindi così:

1. si tiene solo il termine che cresce più in fretta (il **termine dominante**);
2. si elimina il suo coefficiente costante.

Nell'esempio resta n^2 : $T(n)$ cresce in modo quadratico. Le notazioni asintotiche rendono rigoroso questo ragionamento.

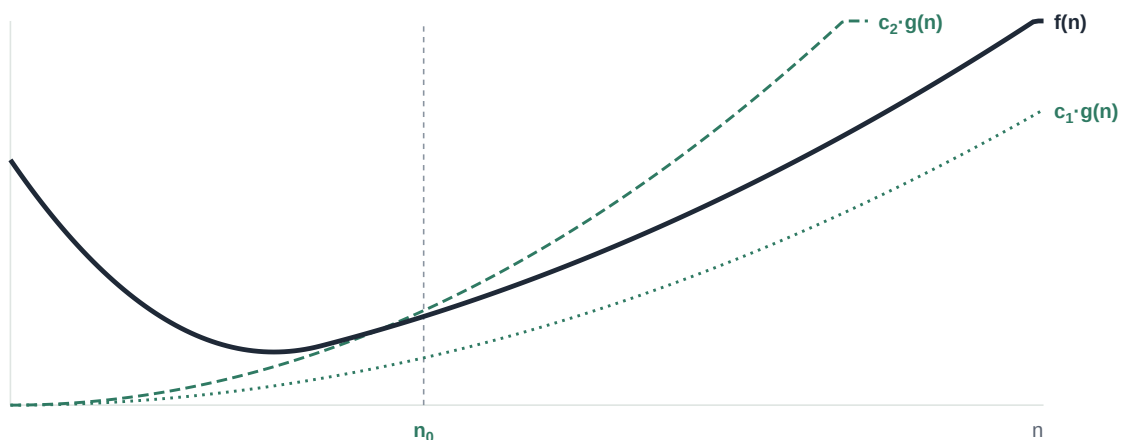
La notazione Θ

DEFINIZIONE: NOTAZIONE THETA

Data una funzione $g(n)$, $\Theta(g(n))$ è l'**insieme** delle funzioni $f(n)$ per cui esistono costanti positive c_1 , c_2 e n_0 tali che

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{per ogni } n \geq n_0$$

Da un certo punto n_0 in poi, $f(n)$ è "intrappolata" in una fascia tra due multipli di $g(n)$: cresce alla stessa velocità di $g(n)$, a meno di costanti. Si dice che $g(n)$ è un **limite asintoticamente stretto** per $f(n)$. Anche se $\Theta(g(n))$ è un insieme, si scrive per abitudine $f(n) = \Theta(g(n))$ invece di $f(n) \in \Theta(g(n))$.



Significato grafico di Θ : a destra di n_0 la funzione $f(n)$ resta sempre tra $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$.

ESEMPIO SVOLTO: $2N^2 + 6N + 4 = \Theta(N^2)$

Bisogna trovare tre costanti. Conviene trattare separatamente le due disuguaglianze.

Limite superiore. Vogliamo $2n^2 + 6n + 4 \leq c_2 \cdot n^2$. Per $n \geq 1$ vale $n \leq n^2$ e $1 \leq n^2$, quindi possiamo maggiorare ogni termine con un multiplo di n^2 :

$$2n^2 + 6n + 4 \leq 2n^2 + 6n^2 + 4n^2 = 12n^2$$

Basta quindi $c_2 = 12$, per ogni $n \geq 1$.

Limite inferiore. Vogliamo $c_1 \cdot n^2 \leq 2n^2 + 6n + 4$. I termini $6n$ e 4 sono positivi, quindi possiamo semplicemente toglierli:

$$2n^2 \leq 2n^2 + 6n + 4$$

Basta $c_1 = 2$, per ogni $n \geq 1$.

Conclusione. Con $c_1 = 2$, $c_2 = 12$ e $n_0 = 1$ la definizione è soddisfatta. Le costanti non sono uniche: ne basta una scelta qualsiasi che funzioni.

La notazione O

DEFINIZIONE: NOTAZIONE O GRANDE

$O(g(n))$ è l'insieme delle funzioni $f(n)$ per cui esistono costanti positive c e n_0 tali che

$$0 \leq f(n) \leq c \cdot g(n) \quad \text{per ogni } n \geq n_0$$

O dà solo un **limite superiore**: $f(n)$ cresce **al più** come $g(n)$. È la notazione usata più spesso per il caso peggiore, perché garantisce che l'algoritmo "non va peggio di così". Attenzione: è un limite che può essere largo. È vero che $n = O(n^2)$, anche se n cresce molto più lentamente di n^2 .

Esempio: $4n + 10 = O(n)$. Per $n \geq 1$ vale $10 \leq 10n$, quindi $4n + 10 \leq 4n + 10n = 14n$. Bastano $c = 14$ e $n_0 = 1$.

Esempio: n^2 non è $O(n)$. Supponiamo per assurdo che esistano c e n_0 con $n^2 \leq c \cdot n$ per ogni $n \geq n_0$. Dividendo per n (positivo) si ottiene $n \leq c$ per ogni $n \geq n_0$: impossibile, perché n cresce senza limiti e c è una costante fissata.

La notazione Ω

DEFINIZIONE: NOTAZIONE OMEGA GRANDE

$\Omega(g(n))$ è l'insieme delle funzioni $f(n)$ per cui esistono costanti positive c e n_0 tali che

$$0 \leq c \cdot g(n) \leq f(n) \quad \text{per ogni } n \geq n_0$$

Ω dà un **limite inferiore**: $f(n)$ cresce **almeno** come $g(n)$. Si usa per dire che un problema o un algoritmo richiede almeno un certo tempo, come il limite $\Omega(n \log n)$ degli ordinamenti per confronto.

Esempio: $n^2/2 - 3n = \Omega(n^2)$. Vogliamo $c \cdot n^2 \leq n^2/2 - 3n$. Dividiamo tutto per n^2 :

$$c \leq 1/2 - 3/n$$

Il termine $3/n$ diminuisce al crescere di n . Per $n \geq 12$ vale $3/n \leq 1/4$, quindi $1/2 - 3/n \geq 1/4$. Bastano $c = 1/4$ e $n_0 = 12$. Si noti il ruolo di n_0 : per n piccolo (per esempio $n = 3$) la funzione è addirittura negativa, ma da 12 in poi la disuguaglianza vale sempre.

DA SAPERE: TEOREMA DI CARATTERIZZAZIONE

Per ogni coppia di funzioni $f(n)$ e $g(n)$: **$f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$** . Per dimostrare un Θ si dimostrano quindi separatamente il limite superiore e quello inferiore, come nell'esempio svolto.

Le notazioni o e ω

O e Ω possono essere "stretti" ($2n^2 = O(n^2)$) oppure no ($2n = O(n^2)$). Le versioni minuscole descrivono i limiti che **non** sono stretti.

DEFINIZIONE: O PICCOLO E OMEGA PICCOLO

$f(n) = o(g(n))$ se per **ogni** costante $c > 0$ esiste n_0 tale che $0 \leq f(n) < c \cdot g(n)$ per ogni $n \geq n_0$. $f(n) = \omega(g(n))$ se per **ogni** costante $c > 0$ esiste n_0 tale che $0 \leq c \cdot g(n) < f(n)$ per ogni $n \geq n_0$.

La differenza con O e Ω è il quantificatore: in O basta **una** costante c che funzioni, in o deve funzionare **qualunque** c, anche piccolissima. Intuitivamente $f(n) = o(g(n))$ significa che f diventa trascurabile rispetto a g: $2n = o(n^2)$, ma $2n^2$ non è $o(n^2)$.

IL CRITERIO DEL LIMITE

In pratica il modo più comodo per confrontare due funzioni è calcolare il limite del loro rapporto.

SE IL LIMITE DI $F(N)/G(N)$ PER $N \rightarrow \infty$ VALE	ALLORA
0	$f(n) = o(g(n))$, quindi anche O
una costante k con $0 < k < \infty$	$f(n) = \Theta(g(n))$
∞	$f(n) = \omega(g(n))$, quindi anche Ω

Esempio: confrontiamo $n \log n$ e n^2 . Il rapporto è $(n \log n)/n^2 = (\log n)/n$, che tende a 0 perché il logaritmo cresce più lentamente di n. Quindi $n \log n = o(n^2)$.

ANALOGIA CON I CONFRONTI TRA NUMERI

NOTAZIONE	SOMIGLIA A	SIGNIFICATO
$f = O(g)$	$f \leq g$	f cresce al più come g
$f = \Omega(g)$	$f \geq g$	f cresce almeno come g
$f = \Theta(g)$	$f = g$	stessa crescita
$f = o(g)$	$f < g$	f cresce strettamente più piano
$f = \omega(g)$	$f > g$	f cresce strettamente più in fretta

Le notazioni hanno anche le proprietà dei confronti: sono **transitive** (se $f = O(g)$ e $g = O(h)$ allora $f = O(h)$), Θ , O e Ω sono **riflessive** ($f = \Theta(f)$), Θ è **simmetrica** ($f = \Theta(g)$ se e solo se $g = \Theta(f)$), e vale la **simmetria trasposta**: $f = O(g)$ se e solo se $g = \Omega(f)$. L'analogia però non è completa: con i numeri, dati a e b, vale sempre $a \leq b$ oppure $a \geq b$; con le funzioni no. Per esempio n e $n^{1 + \sin n}$ non sono confrontabili, perché l'esponente oscilla tra 0 e 2.

Regole pratiche

DA RICORDARE: FATTI DA USARE SPESSO

- La base del logaritmo non conta: $\log_a n = \log_b n / \log_b a$, e $\log_b a$ è una costante. Per questo si scrive solo $\log n$.
- Ogni potenza di logaritmo cresce più piano di ogni potenza di n : $\log^k n = o(n^\epsilon)$ per ogni k e ogni $\epsilon > 0$.
- Ogni polinomio cresce più piano di ogni esponenziale con base maggiore di 1: $n^k = o(a^n)$ per $a > 1$.
- Un polinomio di grado d con coefficiente principale positivo è $\Theta(n^d)$.
- Somma: $\Theta(f) + \Theta(g) = \Theta(\max(f, g))$. Prodotto: $\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g)$.

DALLE RIGHE DI CODICE ALLA NOTAZIONE

Le ultime due regole si applicano direttamente ai programmi. Blocchi eseguiti **uno dopo l'altro** si sommano, quindi conta il più costoso. Cicli **annidati** si moltiplicano.

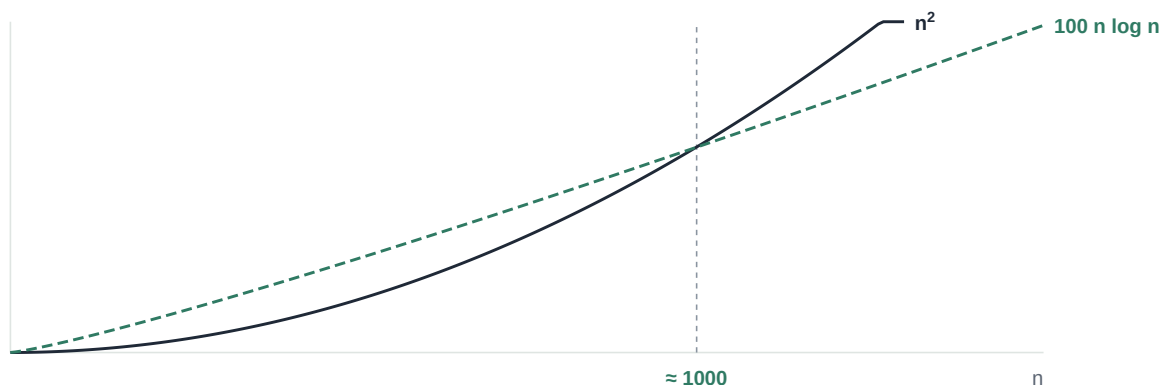
```
1 for i = 1 to n do           // ciclo A: Theta(n)
2   x = x + 1
3 for i = 1 to n do           // ciclo B: n iterazioni ...
4   for j = i to n do         // ... con n - i + 1 iterazioni interne
5     x = x + 1
6 i = 1
7 while i < n do               // ciclo C: i raddoppia ogni volta
8   i = 2 * i
```

- Il ciclo A esegue n operazioni: $\Theta(n)$.
- Il ciclo B esegue $n + (n - 1) + \dots + 1 = n(n + 1)/2$ operazioni: $\Theta(n^2)$.
- Nel ciclo C, i vale 1, 2, 4, 8, ...: dopo k giri vale 2^k , e il ciclo si ferma quando $2^k \geq n$, cioè dopo circa $\log_2 n$ giri: $\Theta(\log n)$.

Il totale è $\Theta(n) + \Theta(n^2) + \Theta(\log n) = \Theta(n^2)$.

Quando le costanti contano

Le notazioni asintotiche descrivono il comportamento per n **grande**. Per ingressi piccoli le costanti possono ribaltare il confronto. Supponiamo che un algoritmo costi $100 \cdot n \log_2 n$ operazioni e un altro n^2 . Asintoticamente il primo è migliore, ma i due costi sono uguali quando $n = 100 \log_2 n$, cioè circa per $n = 1000$: per tutti gli ingressi più piccoli di circa mille elementi l'algoritmo "quadratico" è più veloce.



Per n piccolo vince n^2 ; il sorpasso avviene intorno a $n = 1000$.

È per questo che molte librerie, quando ordinano, usano Merge Sort o Quicksort per gli array grandi ma passano a Insertion Sort quando i sottoarray diventano piccoli.

Esercizi svolti

ORDINARE PER CRESCITA

Ordinare dalla più lenta alla più veloce: $n^{1.5}$, $\log^2 n$, 3^n , $\sqrt{n}$, $n \log n$, 1 , $n/\log n$, 2^n , $n!$, $\log n$.

1. 1 è costante, la più lenta.
2. $\log n$, poi $\log^2 n = (\log n)^2$: il quadrato di qualcosa che tende a infinito cresce di più.
3. $\sqrt{n} = n^{0.5}$: una qualunque potenza di n batte qualunque potenza di logaritmo.
4. $n/\log n$: è più di $\sqrt{n}$ (il rapporto è $\sqrt{n}/\log n$, che tende a infinito) ma meno di n .
5. $n \log n$, poi $n^{1.5}$: il rapporto $n^{1.5}/(n \log n) = \sqrt{n}/\log n$ tende a infinito.
6. 2^n , poi 3^n : il rapporto $(3/2)^n$ tende a infinito.
7. $n!$ per ultima: per $n \geq 7$ vale $n! > 3^n$, e il rapporto $n!/3^n$ cresce senza limiti.

Risultato: 1 , $\log n$, $\log^2 n$, $\sqrt{n}$, $n/\log n$, $n \log n$, $n^{1.5}$, 2^n , 3^n , $n!$.

DUE DOMANDE A TRABOCCHETTO

$2^{n+1} = O(2^n)$? Sì: $2^{n+1} = 2 \cdot 2^n$, quindi basta $c = 2$. Moltiplicare per una costante non cambia la classe.

$2^{2n} = O(2^n)$? No: $2^{2n} = (2^n)^2 = 4^n$, e il rapporto $4^n/2^n = 2^n$ tende a infinito. Raddoppiare l'**esponente** non è moltiplicare per una costante.

DA SAPERE: ERRORI TIPICI

Non confondere O con Θ : dire che Merge Sort è $O(n^2)$ è vero ma poco informativo. Non dimenticare n_0 : le disuguaglianze devono valere solo da un certo punto in poi. Una costante nell'**esponente** o nella **base** di un esponenziale conta (2^n e 3^n sono classi diverse), una costante che **moltiplica** no.

La ricorsione

Che cos'è una funzione ricorsiva, come si progetta, che cosa succede in memoria quando si richiama e come si traduce il suo comportamento in un'equazione di ricorrenza. Con molti esempi in C++ su numeri e array.

Il concetto

Una definizione è **ricorsiva** quando definisce qualcosa in termini di sé stessa, applicata a un caso più semplice. Il fattoriale ne è l'esempio classico: $n! = n \cdot (n - 1)!$, e $0! = 1$. Per calcolare $4!$ ci serve $3!$, per $3!$ ci serve $2!$, e così via fino a $0!$, che conosciamo direttamente.

Una **funzione ricorsiva** è una funzione che, durante la sua esecuzione, chiama sé stessa su un'istanza più piccola del problema. Sembra un circolo vizioso, ma non lo è purché siano rispettate due condizioni:

- esiste un **caso base**, un'istanza così piccola che la si risolve direttamente, senza altre chiamate;
- ogni chiamata ricorsiva lavora su un'istanza **strettamente più piccola**, così che prima o poi si arrivi al caso base.

Pensa alle matrioske: per sapere quante bambole ci sono, apri la più grande, conti uno e ripeti la stessa domanda sulla bambola che c'è dentro. Quando trovi la bambolina piena, che non si apre (il caso base), rispondi "una" e torni indietro sommando.

DA SAPERE: IL SALTO DI FIDUCIA

Quando si scrive una funzione ricorsiva non bisogna cercare di seguire mentalmente tutte le chiamate. Si **assume** che la chiamata sull'istanza più piccola restituisca il risultato giusto, e ci si concentra solo su due domande: come risolvo il caso base? come costruisco la soluzione grande a partire da quella piccola? Se entrambe le risposte sono corrette, la funzione è corretta: è esattamente il principio di induzione.

STRUTTURE DATI RICORSIVE

Molte strutture dati sono ricorsive per natura, ed è per questo che la ricorsione è così naturale per lavorarci.

- Una **lista** è vuota, oppure è un nodo seguito da una lista.
- Un **albero binario** è vuoto, oppure è un nodo radice con due alberi binari come figli, il sinistro e il destro.
- Un **array** $A[l..r]$ è vuoto ($l > r$), oppure è il suo primo elemento seguito dall'array $A[l + 1..r]$, oppure ancora due metà $A[l..m]$ e $A[m + 1..r]$.

Struttura di una funzione ricorsiva

Quasi tutte le funzioni ricorsive hanno la stessa forma:

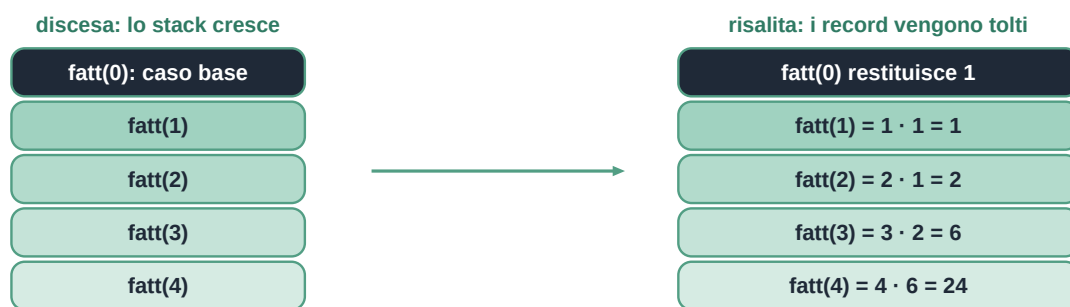
```
1 Ricorsiva(problema P)
2   if P e' abbastanza piccolo then
3       return soluzione diretta          // caso base
4   dividi P in sottoproblemi P1, ..., Pk piu' piccoli
5   S1 = Ricorsiva(P1) ; ... ; Sk = Ricorsiva(Pk)
6   return combina(S1, ..., Sk)           // ricombinazione
```

LA DIMENSIONE DEL PROBLEMA

Prima di scrivere il codice bisogna decidere che cosa misura la **dimensione** n del problema, cioè la quantità che diminuisce a ogni chiamata e che garantisce l'arrivo al caso base. Per il fattoriale è il numero stesso. Per un array può essere la lunghezza, ma nel codice si rappresenta spesso con due indici l e r che delimitano la parte ancora da esaminare: la dimensione è allora $r - l + 1$. La scelta di come ridurre la dimensione (togliere un elemento o dimezzare) cambia completamente il costo, come vedremo.

Che cosa succede in memoria: lo stack delle attivazioni

Ogni volta che una funzione viene chiamata, il sistema crea un **record di attivazione** (o frame) che contiene i suoi parametri, le variabili locali e il punto in cui tornare alla fine. I record sono impilati in una **pila** (stack): la chiamata più recente sta in cima, e quando termina il suo record viene tolto e il controllo torna alla chiamata sottostante. Con la ricorsione la stessa funzione ha contemporaneamente più record attivi, uno per ogni chiamata non ancora conclusa, ciascuno con i propri valori dei parametri.



Lo stack durante il calcolo di 4!: a sinistra la fase di discesa (cinque record attivi insieme), a destra la risalita, in cui ogni chiamata usa il risultato di quella sopra.

DA RICORDARE: MEMORIA DELLA RICORSIONE

Una ricorsione con profondità massima p occupa memoria $\Theta(p)$ sullo stack, anche se ogni chiamata usa poche variabili. Se la profondità è troppo grande (per esempio una ricorsione lineare su un array di milioni di elementi) lo stack si esaurisce: è lo **stack overflow**.

L'equazione di ricorrenza

Il tempo di una funzione ricorsiva si descrive con un'**equazione di ricorrenza**: il costo $T(n)$ su un'istanza di dimensione n è dato dalla somma dei costi delle chiamate ricorsive più il costo del lavoro "locale" (dividere e ricombinare), che indichiamo con $f(n)$:

$$T(n) = T(n_1) + T(n_2) + \dots + T(n_k) + f(n)$$

più il costo del caso base, di solito una costante. Per scrivere l'equazione basta leggere il codice e chiedersi: **quante** chiamate ricorsive fa la funzione, su istanze di **quale dimensione**, e **quanto lavoro** fa al di fuori di esse. Nei prossimi esempi la scriviamo sempre; come risolverla in generale è l'argomento del capitolo successivo.

Esempi su numeri

IL FATTORIALE

```
#include <iostream>
using namespace std;

// n! = n * (n-1)!, con 0! = 1
long long fattoriale(int n) {
    if (n == 0) return 1;           // caso base
    return n * fattoriale(n - 1);   // una chiamata su n - 1
}
```

Una chiamata su $n - 1$ più una moltiplicazione: $T(n) = T(n - 1) + \Theta(1)$. Sviluppando, $T(n) = T(n - 2) + 2 \cdot \Theta(1) = \dots = T(0) + n \cdot \Theta(1)$, quindi **$T(n) = \Theta(n)$** . Anche la profondità dello stack è n .

MOLTIPLICAZIONE COME SOMMA RIPETUTA

Supponiamo di saper fare solo somme. Il prodotto $a \cdot b$, con $b \geq 0$, si può definire così: $a \cdot 0 = 0$ e $a \cdot b = a + a \cdot (b - 1)$.

```
// a * b usando solo somme (b >= 0)
int moltiplica(int a, int b) {
    if (b == 0) return 0;           // caso base
    return a + moltiplica(a, b - 1); // b diminuisce di 1
}
```

La dimensione è b : $T(b) = T(b - 1) + \Theta(1) = \Theta(b)$.

LA POTENZA: LENTA E VELOCE

La definizione più diretta di x^n è $x^0 = 1$ e $x^n = x \cdot x^{n-1}$, che dà $T(n) = T(n - 1) + \Theta(1) = \Theta(n)$, esattamente come il fattoriale.

Si può fare molto meglio osservando che $x^n = (x^{n/2})^2$ se n è pari, e $x^n = x \cdot (x^{(n-1)/2})^2$ se n è dispari. Per esempio $x^{10} = (x^5)^2$ e $x^5 = x \cdot (x^2)^2$. A ogni passo n si dimezza.

```
// x^n in tempo lineare
long long potenzaLenta(long long x, int n) {
    if (n == 0) return 1;
    return x * potenzaLenta(x, n - 1);
}

// x^n per quadrati successivi
long long potenzaVeloce(long long x, int n) {
    if (n == 0) return 1;
    long long meta = potenzaVeloce(x, n / 2); // UNA sola chiamata
    if (n % 2 == 0) return meta * meta;
    else return x * meta * meta;
}
```

Per `potenzaVeloce`: $T(n) = T(n/2) + \Theta(1)$, quindi la dimensione si dimezza a ogni chiamata e si arriva a 0 dopo circa $\log_2 n$ passi: **$T(n) = \Theta(\log n)$** . Per x^{1000} servono una decina di chiamate invece di mille.

DA SAPERE: ATTENZIONE ALLE CHIAMATE RIPETUTE

Se in `potenzaVeloce` si scrivesse `return potenzaVeloce(x, n / 2) * potenzaVeloce(x, n / 2)`, il risultato sarebbe ancora giusto, ma le chiamate diventerebbero **due**: $T(n) = 2T(n/2) + \Theta(1)$, che vale $\Theta(n)$. Tutto il guadagno andrebbe perso. Un risultato ricorsivo che serve due volte va calcolato **una volta** e salvato in una variabile.

Esempi su array

SOMMA DEGLI ELEMENTI: DUE MODI DI RIDURRE

Il primo modo toglie un elemento alla volta: la somma di $A[l..r]$ è $A[l]$ più la somma di $A[l + 1..r]$.

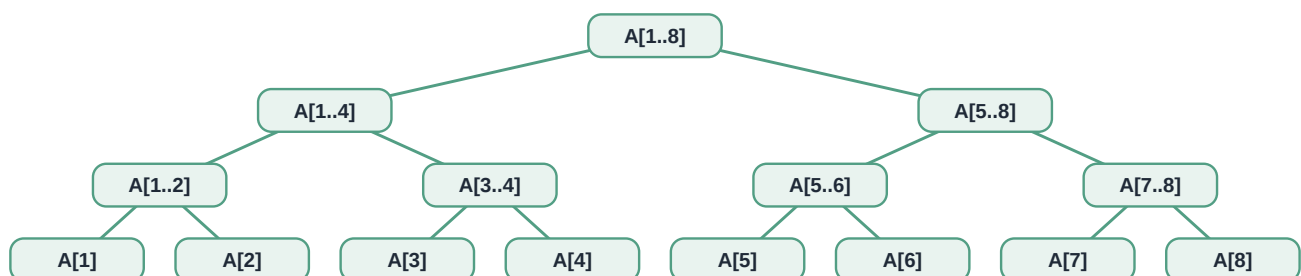
```
// Somma di A[l..r] togliendo un elemento alla volta
int sommaLineare(int A[], int l, int r) {
    if (l > r) return 0;                // array vuoto
    return A[l] + sommaLineare(A, l + 1, r);
}
```

Il secondo modo divide l'array a metà, somma ricorsivamente le due metà e somma i due risultati.

```
// Somma di A[l..r] dividendo a metà
int sommaDicotomica(int A[], int l, int r) {
    if (l > r) return 0;                // array vuoto
    if (l == r) return A[l];            // un solo elemento
    int m = (l + r) / 2;
    return sommaDicotomica(A, l, m) + sommaDicotomica(A, m + 1, r);
}
```

VERSIONE	RICORRENZA	TEMPO	PROFONDITÀ DELLO STACK
lineare	$T(n) = T(n - 1) + \Theta(1)$	$\Theta(n)$	n
dicotomica	$T(n) = 2T(n/2) + \Theta(1)$	$\Theta(n)$	$\log_2 n$

Entrambe fanno $n - 1$ somme, quindi lo stesso tempo: non si può sommare n numeri senza guardarli tutti. Ma la seconda usa molto meno stack. Il tempo $\Theta(n)$ della dicotomica si vede con l'albero delle chiamate: ha n foglie (una per elemento) e $n - 1$ nodi interni, e ognuno costa una costante.



Albero delle chiamate di `sommaDicotomica` su 8 elementi: 15 chiamate, profondità 3.

RICERCA BINARIA

In un array **ordinato** si confronta x con l'elemento centrale: se è uguale si è finito, se è più piccolo si cerca solo nella metà sinistra, altrimenti solo nella destra.

```
// Restituisce la posizione di x in A[l..r] ordinato, oppure -1
int ricercaBinaria(int A[], int l, int r, int x) {
    if (l > r) return -1; // intervallo vuoto: non c'e'
    int m = (l + r) / 2;
    if (A[m] == x) return m;
    if (x < A[m]) return ricercaBinaria(A, l, m - 1, x);
    else return ricercaBinaria(A, m + 1, r, x);
}
```

Una sola chiamata su metà array: $T(n) = T(n/2) + \Theta(1)$, quindi $\Theta(\log n)$. Si confronti con la somma dicotomica: stessa divisione a metà, ma **una** chiamata invece di due, e il tempo scende da lineare a logaritmico.

VERIFICARE UNA PROPRIETÀ: L'ARRAY È ORDINATO?

Molti esercizi chiedono di verificare una proprietà. Dividendo a metà, $A[l..r]$ è ordinato se lo sono entrambe le metà **e** se l'ultimo elemento della prima non supera il primo della seconda: è la condizione "di confine" da non dimenticare.

```
// true se A[l..r] e' ordinato in modo crescente
bool ordinato(int A[], int l, int r) {
    if (l >= r) return true; // 0 o 1 elementi
    int m = (l + r) / 2;
    if (A[m] > A[m + 1]) return false; // controllo sul confine
    return ordinato(A, l, m) && ordinato(A, m + 1, r);
}
```

Nel caso peggiore (array ordinato, si devono controllare entrambe le metà) $T(n) = 2T(n/2) + \Theta(1) = \Theta(n)$. L'operatore `&&` è "cortocircuitato": se la prima metà non è ordinata, la seconda chiamata non viene nemmeno fatta, quindi in molti casi si finisce prima.

MASSIMO DI UN ARRAY

```
// Massimo di A[l..r] (almeno un elemento)
int massimo(int A[], int l, int r) {
    if (l == r) return A[l];
    int m = (l + r) / 2;
    int a = massimo(A, l, m); // massimo della meta' sinistra
    int b = massimo(A, m + 1, r); // massimo della meta' destra
    if (a > b) return a;
    else return b;
}

int main() {
    int A[] = {3, 8, 1, 9, 4, 7, 2, 6};
    int B[] = {1, 3, 4, 6, 8, 9, 11, 15};
    cout << fattoriale(5) << " " << moltiplica(7, 6) << endl;
    cout << potenzaLenta(3, 10) << " " << potenzaVeloce(3, 10) << endl;
    cout << sommaLineare(A, 0, 7) << " " << sommaDicotomica(A, 0, 7) << endl;
    cout << ricercaBinaria(B, 0, 7, 8) << " " << ricercaBinaria(B, 0, 7, 5) << endl;
    cout << ordinato(A, 0, 7) << " " << ordinato(B, 0, 7) << endl;
    cout << massimo(A, 0, 7) << endl;
    return 0;
}
```

Due chiamate su metà array e un confronto: $T(n) = 2T(n/2) + \Theta(1) = \Theta(n)$.

Ricorsivo o iterativo?

Ogni funzione ricorsiva si può riscrivere in forma iterativa, con cicli ed eventualmente una pila esplicita, e viceversa. Le due forme hanno lo stesso costo asintotico in tempo, ma differiscono in altri aspetti.

ASPETTO	RICORSIONE	ITERAZIONE
Leggibilità	naturale per divide et impera, alberi, backtracking	naturale per scansioni semplici
Memoria	stack proporzionale alla profondità	di solito costante
Costo nascosto	creazione dei record di attivazione	nessuno
Rischi	stack overflow se troppo profonda	cicli infiniti

Un caso particolare è la **ricorsione in coda** (tail recursion): la chiamata ricorsiva è l'**ultima** operazione della funzione, e il suo risultato viene restituito così com'è. `sommaLineare` **non** è in coda, perché dopo la chiamata deve ancora sommare `A[l]`. Una ricorsione in coda si trasforma meccanicamente in un ciclo, e molti compilatori lo fanno da soli.

DA SAPERE: ERRORI TIPICI

- Dimenticare il caso base, o scriverne uno che non viene mai raggiunto: ricorsione infinita e stack overflow.
- Non ridurre davvero la dimensione: per esempio, con $m = (l + r)/2$, chiamare su `A[l..m]` e `A[m..r]` invece che su `A[l..m]` e `A[m + 1..r]` fa ripetere l'elemento `A[m]` e, con due elementi, può non terminare mai.
- Ricalcolare più volte lo stesso risultato ricorsivo invece di salvarlo in una variabile.
- Nel dividere un array, dimenticare la condizione "sul confine" tra le due metà.

Equazioni di ricorrenza

Come scrivere l'equazione di ricorrenza di un algoritmo leggendone il codice, e come risolverla con i tre metodi del corso: l'albero di ricorsione, la sostituzione e il Teorema Master. Con molti esempi risolti passo per passo.

Che cosa è un'equazione di ricorrenza

Un'equazione di ricorrenza descrive una funzione in termini dei suoi valori su argomenti più piccoli. Per gli algoritmi divide et impera ha quasi sempre questa forma:

$$T(n) = a \cdot T(n/b) + f(n)$$

DEFINIZIONE: I TRE INGREDIENTI

- $a \geq 1$ è il **numero** di sottoproblemi (chiamate ricorsive) generati a ogni passo;
- $b > 1$ è il **fattore di riduzione**: ogni sottoproblema ha dimensione n/b ;
- $f(n)$ è il costo del lavoro **non ricorsivo** fatto dalla chiamata su dimensione n : dividere il problema, combinare i risultati, cicli e operazioni varie.

Si aggiunge un caso base, di solito $T(n) = \Theta(1)$ per n piccolo, che quasi sempre non cambia il risultato asintotico e per questo spesso non si scrive. Si trascurano anche le parti intere: si scrive $T(n/2)$ invece di $T(\lfloor n/2 \rfloor)$, perché non cambiano la soluzione asintotica.

Primo passo: scrivere l'equazione

Per passare dal codice all'equazione si risponde a tre domande:

1. **Quante** chiamate ricorsive fa la procedura? Quel numero è a (o, se le chiamate hanno dimensioni diverse, servono più termini).
2. Su istanze di **quale dimensione**? Questo dà n/b , oppure $n - 1$, $n - 2$, ...
3. Quanto costa **tutto il resto**? Si sommano i costi di cicli, chiamate ad altre procedure non ricorsive e operazioni elementari: è $f(n)$.

ESEMPI CLASSICI

Ricerca binaria. Una sola chiamata su metà intervallo, più un confronto: $T(n) = T(n/2) + \Theta(1)$.

Merge Sort. Due chiamate su metà array, più la fusione lineare: $T(n) = 2T(n/2) + \Theta(n)$.

Strassen. Questo algoritmo moltiplica due matrici $n \times n$ dividendole in quattro blocchi di dimensione $n/2$ e calcolando **sette** prodotti di blocchi (invece degli otto ovvi), più un numero costante di somme di matrici, ciascuna $\Theta(n^2)$: $T(n) = 7T(n/2) + \Theta(n^2)$.

PROCEDURE CON CICLI E CHIAMATE AD ALTRE PROCEDURE

Spesso la parte non ricorsiva contiene cicli che chiamano procedure già note. Bisogna moltiplicare il numero di iterazioni per il costo di ogni iterazione. Consideriamo questa procedura che lavora su un max heap.

```

1 Dimezza(A) // A e' un max heap
2   n = heapSize(A)
3   if n <= 1 then return
4   for i = 1 to n/2 do
5       ExtractMax(A) // ogni estrazione costa O(log n)
6   Dimezza(A) // ora l'heap ha n/2 elementi

```

Il ciclo fa $n/2$ estrazioni da $O(\log n)$ ciascuna: $f(n) = (n/2) \cdot \log n = \Theta(n \log n)$. Poi c'è una sola chiamata, su un heap che ora ha $n/2$ elementi. Quindi $T(n) = T(n/2) + \Theta(n \log n)$. Attenzione: la dimensione della chiamata ricorsiva non si legge dai parametri (è sempre A), ma da **che cosa succede ai dati** prima della chiamata.

RICORSIONI SU ALBERI

Una procedura su un albero binario chiama sé stessa sui due sottoalberi, ma la loro dimensione dipende dalla **forma** dell'albero.

```

1 Visita(x)
2   if x = NIL then return
3   Visita(x.sinistro)
4   Conta(x) // conta i nodi del sottoalbero di x: costa Theta(dimensione)
5   Visita(x.destro)

```

- Se l'albero è **bilanciato**, i due sottoalberi hanno circa $n/2$ nodi: $T(n) = 2T(n/2) + \Theta(n)$.
- Se l'albero è **degenere** (una lista), un sottoalbero ha $n - 1$ nodi e l'altro è vuoto: $T(n) = T(n - 1) + \Theta(n)$.

Vedremo che la prima vale $\Theta(n \log n)$ e la seconda $\Theta(n^2)$: stesso codice, costo diverso a seconda della forma dei dati.

RICORRENZE NON UNIFORMI

Non sempre i sottoproblemi hanno la stessa dimensione. In Quicksort, se il pivot divide l'array in una parte con una frazione α degli elementi e una con il resto, $T(n) = T(\alpha n) + T((1 - \alpha)n) + \Theta(n)$; nel caso peggiore $T(n) = T(n - 1) + \Theta(n)$. Nelle **Torri di Hanoi**, per spostare n dischi si spostano $n - 1$ dischi sul piolo di appoggio, si sposta il disco più grande e si rispostano gli $n - 1$ dischi sopra di esso: $T(n) = 2T(n - 1) + 1$. Il Teorema Master non si applica a queste forme, ma l'albero di ricorsione sì.

Metodo dell'albero di ricorsione

È il metodo più intuitivo e quello che fa davvero capire la ricorrenza. Si disegnano le chiamate come un albero: la radice è la chiamata su n , i suoi figli sono le chiamate che genera, e così via. In ogni nodo si scrive il costo **non ricorsivo** di quella chiamata. Il costo totale è la somma di tutti i nodi, che si calcola **livello per livello**.

DA RICORDARE: LO SCHEMA DI CALCOLO PER $T(N) = AT(N/B) + F(N)$

1. Al livello i (la radice è il livello 0) ci sono a^i nodi, ciascuno su un problema di dimensione n/b^i .
2. Il costo del livello i è quindi $a^i \cdot f(n/b^i)$.
3. La dimensione arriva a 1 quando $n/b^i = 1$, cioè al livello $\log_b n$: è l'altezza dell'albero.
4. Le foglie sono $a^{\log_b n} = n^{\log_b a}$, e costano $\Theta(1)$ ciascuna.
5. Si sommano i costi dei livelli, spesso con una serie geometrica.

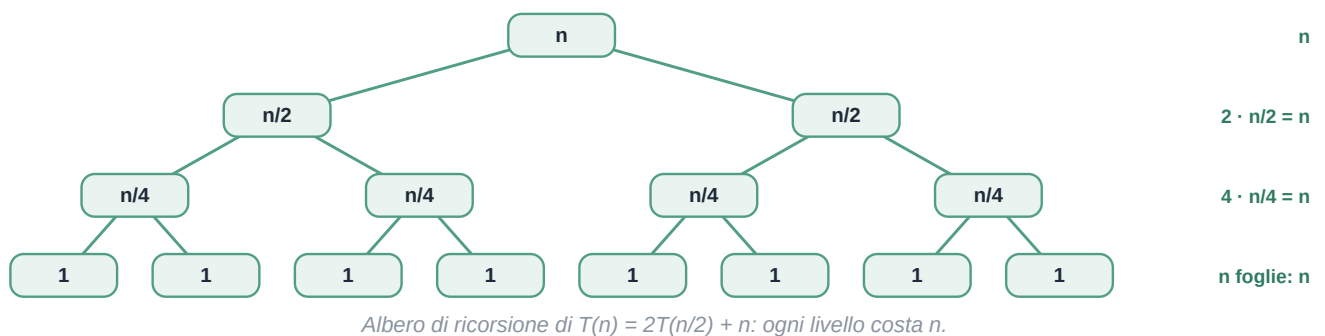
Il punto 4 usa l'identità $a^{\log_b n} = n^{\log_b a}$, che si verifica prendendo il logaritmo in base b di entrambi i lati: entrambi danno $\log_b n \cdot \log_b a$.

DA RICORDARE: LE SERIE CHE SERVONO

- Serie aritmetica: $1 + 2 + \dots + n = n(n+1)/2 = \Theta(n^2)$.
- Serie geometrica finita, con q diverso da 1: $1 + q + q^2 + \dots + q^k = (q^{k+1} - 1)/(q - 1)$.
- Se $q < 1$ la somma è sempre minore di $1/(1 - q)$, una costante: **domina il primo termine**.
- Se $q > 1$ la somma è $\Theta(q^k)$: **domina l'ultimo termine**.
- Se $q = 1$ tutti i termini sono uguali: la somma è $(k + 1)$ volte il termine.

Queste tre situazioni (domina la radice, dominano le foglie, tutti i livelli contano uguale) sono esattamente i tre casi del Teorema Master.

ESEMPIO 1: $T(N) = 2T(N/2) + N$



1. Al livello i ci sono 2^i nodi di costo $n/2^i$: il livello costa $2^i \cdot n/2^i = n$.
2. I livelli vanno da 0 a $\log_2 n$: sono $\log_2 n + 1$.
3. Totale: $n \cdot (\log_2 n + 1) = \Theta(n \log n)$.

ESEMPIO 2: $T(N) = 3T(N/2) + N$

1. Al livello i : 3^i nodi, ciascuno di costo $n/2^i$. Costo del livello: $3^i \cdot n/2^i = (3/2)^i \cdot n$.
2. Il costo **cresce** scendendo (rapporto $3/2 > 1$): domina l'ultimo livello.
3. Altezza $\log_2 n$. Le foglie sono $3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$.
4. Sommando la serie geometrica di ragione $3/2$:

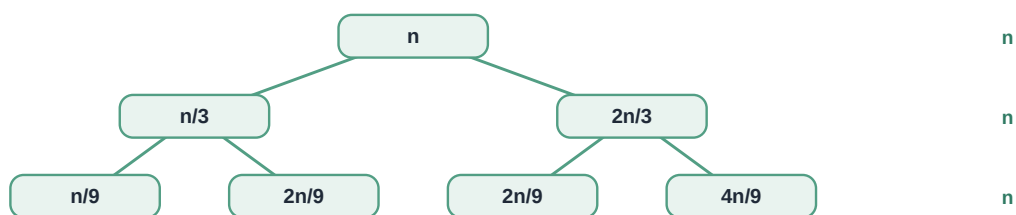
$$T(n) = n \cdot \sum_{i=0.. \log_2 n} (3/2)^i = \Theta(n \cdot (3/2)^{\log_2 n}) = \Theta(n \cdot n^{\log_2 3} / n) = \Theta(n^{\log_2 3})$$

dove si è usato $(3/2)^{\log_2 n} = 3^{\log_2 n} / 2^{\log_2 n} = n^{\log_2 3} / n$.

ESEMPIO 3: $T(N) = 2T(N/2) + N^2$

1. Livello i : 2^i nodi di costo $(n/2^i)^2 = n^2/4^i$. Costo del livello: $2^i \cdot n^2/4^i = n^2/2^i$.
2. I costi **diminuiscono** (rapporto $1/2$): domina la radice.
3. Somma: $n^2 \cdot (1 + 1/2 + 1/4 + \dots) \leq n^2 \cdot 2$.
4. Quindi $T(n) \leq 2n^2$, e d'altra parte $T(n) \geq n^2$ (la sola radice): $\Theta(n^2)$.

ESEMPIO 4: RICORRENZA NON UNIFORME $T(N) = T(N/3) + T(2N/3) + N$



Albero sbilanciato: ogni livello completo costa n , ma i rami hanno lunghezze diverse.

1. Ogni livello completo costa n : $n/3 + 2n/3 = n$, poi $n/9 + 2n/9 + 2n/9 + 4n/9 = n$, e così via.
2. Il ramo più corto (sempre a sinistra, divide per 3) arriva a 1 dopo $\log_3 n$ livelli; il più lungo (sempre a destra, moltiplica per $2/3$) dopo $\log_{3/2} n$ livelli.
3. Fino al livello $\log_3 n$ tutti i livelli sono completi: il costo è almeno $n \cdot \log_3 n$. In totale ci sono al più $\log_{3/2} n$ livelli, ciascuno di costo al più n : il costo è al più $n \cdot \log_{3/2} n$.
4. Entrambi i limiti sono $\Theta(n \log n)$, perché logaritmi in basi diverse differiscono per una costante: **$T(n) = \Theta(n \log n)$** .

È lo stesso ragionamento che mostra che Quicksort con partizioni sbilanciate in proporzione costante resta $\Theta(n \log n)$.

ESEMPIO 5: $T(N) = 2T(N - 1) + 1$ (TORRI DI HANOI)

1. Livello i : 2^i nodi, ciascuno di costo 1. La dimensione diminuisce di 1 per livello, quindi ci sono n livelli.
2. Somma: $1 + 2 + 4 + \dots + 2^{n-1} = 2^n - 1$.
3. **$T(n) = \Theta(2^n)$** : esponenziale. Ridurre la dimensione di una costante invece che di un fattore, con due chiamate, è catastrofico.

ESEMPIO 6: $T(N) = T(N - 1) + N$

Un solo nodo per livello, n livelli, costi $n, n - 1, \dots, 1$: è la serie aritmetica, **$\Theta(n^2)$** . È il caso peggiore di Quicksort e la visita dell'albero degenera vista prima.

Metodo della sostituzione

Il metodo della sostituzione serve a **dimostrare** rigorosamente una soluzione, e si usa spesso per verificare quella ottenuta con l'albero di ricorsione. Ha due passi:

1. **indovinare** la forma della soluzione (per esempio $T(n) \leq c \cdot n \log n$);
2. dimostrare per **induzione** che la stima vale, trovando costanti che funzionano.

ESEMPIO: $T(N) = 2T(N/2) + N$ È $O(N \log N)$

Ipotesi: $T(n) \leq c \cdot n \log_2 n$ per una costante c da scegliere. **Ipotesi induttiva:** supponiamo che valga per $n/2$, cioè $T(n/2) \leq c \cdot (n/2) \log_2(n/2)$. Sostituiamo nella ricorrenza:

$$\begin{aligned}
 T(n) &= 2T(n/2) + n \\
 &\leq 2 \cdot c \cdot (n/2) \log_2(n/2) + n \\
 &= c n (\log_2 n - 1) + n \\
 &= c n \log_2 n - c n + n \\
 &\leq c n \log_2 n \quad \text{se } c \geq 1
 \end{aligned}$$

Nel terzo passaggio si è usato $\log_2(n/2) = \log_2 n - 1$. L'ultima disuguaglianza vale perché, se $c \geq 1$, il termine $-cn + n$ è minore o uguale a zero.

Casi base. Per $n = 1$ la stima direbbe $T(1) \leq c \cdot 1 \cdot \log 1 = 0$, falso. Ma la notazione asintotica chiede che la disuguaglianza valga solo da un certo n_0 in poi: si prendono come casi base $n = 2$ e $n = 3$ (da cui discendono tutti gli altri valori), e si sceglie c abbastanza grande da soddisfarli, per esempio $c \geq 2$ se $T(1) = 1$.

UNA TRAPPOLA: IL TERMINE DI ORDINE INFERIORE

Proviamo a dimostrare che $T(n) = 2T(n/2) + 1$ è $O(n)$ con l'ipotesi $T(n) \leq cn$:

$$T(n) \leq 2 \cdot c(n/2) + 1 = cn + 1$$

e $cn + 1$ **non** è $\leq cn$. L'ipotesi è giusta (la soluzione è davvero $\Theta(n)$), ma è troppo debole per far funzionare l'induzione. Il trucco è **sottrarre** un termine di ordine inferiore e provare $T(n) \leq cn - d$:

$$T(n) \leq 2(cn/2 - d) + 1 = cn - 2d + 1 \leq cn - d \quad \text{se } d \geq 1$$

DA SAPERE: L'ERRORE DA NON FARE

Non si può "dimostrare" che $T(n) = 2T(n/2) + n$ è $O(n)$ scrivendo $T(n) \leq 2cn/2 + n = cn + n = O(n)$. È sbagliato: bisognava dimostrare **esattamente** $T(n) \leq cn$ con la **stessa** costante c dell'ipotesi, e $cn + n$ non lo è. La costante non può crescere a ogni passo dell'induzione.

CAMBIAMENTO DI VARIABILE

A volte una ricorrenza strana diventa familiare con una sostituzione. Per $T(n) = 2T(\sqrt{n}) + \log_2 n$ si pone $m = \log_2 n$, cioè $n = 2^m$, così $\sqrt{n} = 2^{m/2}$. Chiamando $S(m) = T(2^m)$:

$$S(m) = 2S(m/2) + m \Rightarrow S(m) = \Theta(m \log m)$$

e tornando a n : $T(n) = \Theta(\log n \cdot \log \log n)$.

Il Teorema Master

Il Teorema Master è un "ricettario" che risolve immediatamente la maggior parte delle ricorrenze del tipo $T(n) = aT(n/b) + f(n)$. L'idea è confrontare $f(n)$, il costo della radice, con $n^{\log_b a}$, il numero di foglie dell'albero, e vedere chi vince.

DEFINIZIONE: TEOREMA MASTER

Siano $a \geq 1$ e $b > 1$ costanti, $f(n)$ una funzione e $T(n) = aT(n/b) + f(n)$. Allora:

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 **$T(n) = \Theta(n^{\log_b a} \log n)$** ;
3. se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche costante $\epsilon > 0$, e se vale la **condizione di regolarità** $a \cdot f(n/b) \leq k \cdot f(n)$ per qualche costante $k < 1$ e per n abbastanza grande, allora **$T(n) = \Theta(f(n))$** .

INTERPRETAZIONE

- **Caso 1, vincono le foglie.** $f(n)$ è polinomialmente più piccola di $n^{\log_b a}$: i costi dei livelli crescono scendendo e il totale è dominato dal costo delle foglie.
- **Caso 2, pareggio.** Ogni livello costa circa lo stesso; ci sono $\log n$ livelli, da cui il fattore $\log n$ in più.
- **Caso 3, vince la radice.** $f(n)$ è polinomialmente più grande: i costi diminuiscono scendendo e domina il lavoro della prima chiamata. La condizione di regolarità garantisce proprio che i costi diminuiscano.

Le parole "polinomialmente più piccola/grande" sono essenziali: non basta che $f(n)$ sia più piccola o più grande, deve esserlo di un fattore n^ϵ . Per esempio $n \log n$ è più grande di n , ma non polinomialmente (il rapporto è $\log n$, che cresce più piano di qualunque n^ϵ).

COME APPLICARLO, PASSO PER PASSO

1. Leggere a , b e $f(n)$ dalla ricorrenza.
2. Calcolare l'esponente $\log_b a$ e quindi $n^{\log_b a}$.
3. Confrontare $f(n)$ con $n^{\log_b a}$ e decidere il caso (o concludere che il teorema non si applica).
4. Nel caso 3, verificare la condizione di regolarità.
5. Scrivere la soluzione.

Esempi risolti con il Teorema Master

RICORRENZA	A	B	$N^{\log_B A}$	CONFRONTO CON $F(N)$	SOLUZIONE
$T(n) = 9T(n/3) + n$	9	3	n^2	$n = O(n^{2-\epsilon})$, caso 1	$\Theta(n^2)$
$T(n) = 2T(n/2) + 1$	2	2	n	$1 = O(n^{1-\epsilon})$, caso 1	$\Theta(n)$
$T(n) = 7T(n/2) + n^2$	7	2	$n^{2,81}$	$n^2 = O(n^{2,81-\epsilon})$, caso 1	$\Theta(n^{\log_2 7})$
$T(n) = T(2n/3) + 1$	1	3/2	$n^0 = 1$	$1 = \Theta(1)$, caso 2	$\Theta(\log n)$
$T(n) = 4T(n/2) + n^2$	4	2	n^2	uguali, caso 2	$\Theta(n^2 \log n)$
$T(n) = T(n/2) + n^2$	1	2	1	$n^2 = \Omega(n^{0+\epsilon})$, caso 3	$\Theta(n^2)$
$T(n) = 3T(n/4) + n \log n$	3	4	$n^{0,79}$	$n \log n = \Omega(n^{0,79+\epsilon})$, caso 3	$\Theta(n \log n)$
$T(n) = T(n/2) + \Theta(n \log n)$	1	2	1	caso 3	$\Theta(n \log n)$

Vediamo alcuni casi in dettaglio.

$T(n) = 9T(n/3) + n$. $a = 9$, $b = 3$, $\log_3 9 = 2$, quindi $n^{\log_b a} = n^2$. $f(n) = n = O(n^{2-\epsilon})$ con $\epsilon = 1$: caso 1, $T(n) = \Theta(n^2)$. Intuitivamente: 9 sottoproblemi grandi un terzo generano tantissime foglie, n^2 , e il lavoro lineare della radice è trascurabile.

$T(n) = T(2n/3) + 1$. Attenzione al valore di b : la dimensione passa da n a $2n/3 = n/(3/2)$, quindi $b = 3/2$. Con $a = 1$, $\log_{3/2} 1 = 0$ e $n^0 = 1$. $f(n) = 1 = \Theta(1)$: caso 2, $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$. È la ricorrenza di MaxHeapify.

$T(n) = 3T(n/4) + n \log n$. $\log_4 3 \approx 0,79$, e $f(n) = n \log n = \Omega(n^{0,79+\epsilon})$ con $\epsilon \approx 0,2$: candidato al caso 3. Regolarità: $a \cdot f(n/b) = 3 \cdot (n/4) \log(n/4) \leq (3/4) n \log n = (3/4) f(n)$, quindi vale con $k = 3/4 < 1$. Soluzione: $\Theta(n \log n)$.

La procedura Dimezza vista all'inizio: $T(n) = T(n/2) + \Theta(n \log n)$. Con $a = 1$ e $b = 2$, $n^{\log_b a} = 1$, e $n \log n$ è polinomialmente più grande. Regolarità: $f(n/2) = (n/2) \log(n/2) \leq (1/2) n \log n$. Caso 3: $T(n) = \Theta(n \log n)$.

Quando il Teorema Master non si applica

Ci sono tre "buchi" in cui il teorema tace:

- la ricorrenza non ha la forma $aT(n/b) + f(n)$ (per esempio $T(n-1)$, o sottoproblemi di dimensioni diverse come $T(n/3) + T(2n/3)$);
- $f(n)$ è più piccola o più grande di $n^{\log_b a}$, ma non **polinomialmente**;
- nel caso 3 non vale la condizione di regolarità.

ESEMPIO 7: $T(N) = 2T(N/2) + N \log N$

$n^{\log_b a} = n$, e $f(n) = n \log n$ è più grande di n solo di un fattore $\log n$, non di un n^ϵ : il caso 3 non si applica, e nemmeno il 2. Usiamo l'albero di ricorsione.

- Livello i : 2^i nodi su dimensione $n/2^i$, ciascuno di costo $(n/2^i) \log_2(n/2^i)$. Costo del livello: $n \cdot (\log_2 n - i)$.
- Sommando per i da 0 a $\log_2 n - 1$, con $L = \log_2 n$:

$$n \cdot (L + (L-1) + \dots + 1) = n \cdot L(L+1)/2 = \Theta(n \log^2 n)$$

ESEMPIO 8: $T(N) = T(N/2) + \log N$

Qui $n^{\log_b a} = 1$ e $f(n) = \log n$ non è $\Theta(1)$ né polinomialmente più grande. Con l'albero: un solo nodo per livello, al livello i di costo $\log_2(n/2^i) = L - i$. La somma $L + (L - 1) + \dots + 1 = \Theta(L^2)$, quindi $T(n) = \Theta(\log^2 n)$.

DA RICORDARE: ESTENSIONE DEL CASO 2

I due esempi precedenti rientrano in una forma più generale del caso 2: se $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ con $k \geq 0$, allora $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$. Con $k = 1$: $2T(n/2) + n \log n = \Theta(n \log^2 n)$ e $T(n/2) + \log n = \Theta(\log^2 n)$. Se non si ricorda questa estensione, l'albero di ricorsione dà sempre la risposta.

Riepilogo

RICORRENZA	SOLUZIONE	ESEMPIO
$T(n) = T(n/2) + 1$	$\Theta(\log n)$	ricerca binaria
$T(n) = T(n - 1) + 1$	$\Theta(n)$	fattoriale, scansione ricorsiva
$T(n) = 2T(n/2) + 1$	$\Theta(n)$	somma o massimo dicotomici
$T(n) = 2T(n/2) + n$	$\Theta(n \log n)$	Merge Sort
$T(n) = T(n - 1) + n$	$\Theta(n^2)$	Quicksort nel caso peggiore
$T(n) = 2T(n - 1) + 1$	$\Theta(2^n)$	Torri di Hanoi
$T(n) = T(n/2) + n$	$\Theta(n)$	selezione, se dimezza
$T(n) = 7T(n/2) + n^2$	$\Theta(n^{2.81})$	Strassen

DA SAPERE: STRATEGIA GENERALE

Prima si scrive con cura la ricorrenza dal codice (quante chiamate, di che dimensione, quanto lavoro extra). Poi si prova il **Teorema Master**, controllando che la forma sia quella giusta e che il confronto sia polinomiale. Se non si applica, si disegna l'**albero di ricorsione** e si sommano i livelli. Se serve una dimostrazione rigorosa, si conferma il risultato con la **sostituzione**.

Analisi di algoritmi ricorsivi notevoli

Le dimostrazioni di costo e di correttezza degli algoritmi visti nella dispensa di progettazione: MaxHeapify, BuildMaxHeap, Heapsort, Merge Sort e Quicksort, compresa l'analisi del caso medio di Quicksort randomizzato.

MaxHeapify: perché la ricorrenza è $T(2n/3)$

MaxHeapify(A, i) confronta A[i] con i figli, eventualmente scambia e si richiama su **uno** dei due sottoalberi. Il lavoro locale è $\Theta(1)$. Per scrivere la ricorrenza serve sapere quanto può essere grande, al massimo, il sottoalbero su cui si prosegue.

DA SAPERE: IL SOTTOALBERO PIÙ GRANDE

In un heap di n nodi, ciascuno dei due sottoalberi della radice ha al più $2n/3$ nodi.

Il caso peggiore si ha quando l'ultimo livello è pieno **esattamente a metà**: tutto il livello inferiore sta nel sottoalbero sinistro, che è completo di altezza h, mentre il destro è completo ma di altezza h - 1. Contiamo.

1. Un albero binario completo con k livelli ha $1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1$ nodi.
2. Nel caso peggiore il sottoalbero sinistro è completo con h livelli, quindi ha $2^h - 1$ nodi, e il destro è completo con h - 1 livelli, quindi ha $2^{h-1} - 1$ nodi.
3. Totale, contando la radice: $n = 1 + (2^h - 1) + (2^{h-1} - 1) = 3 \cdot 2^{h-1} - 1$.
4. Frazione del sinistro: $(2^h - 1)/(3 \cdot 2^{h-1} - 1) = (2 \cdot 2^{h-1} - 1)/(3 \cdot 2^{h-1} - 1) \leq 2/3$.

L'ultima disuguaglianza si verifica moltiplicando in croce: $3(2x - 1) \leq 2(3x - 1)$, cioè $6x - 3 \leq 6x - 2$, sempre vera (con $x = 2^{h-1}$). Quindi:

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

Con il Teorema Master: $a = 1$, $b = 3/2$, $n^{\log_{3/2} 1} = n^0 = 1$, $f(n) = \Theta(1)$: caso 2, **$T(n) = O(\log n)$** . In modo equivalente, MaxHeapify su un nodo di altezza h costa $O(h)$.

BuildMaxHeap costa $O(n)$

CORRETTEZZA

Invariante: all'inizio di ogni iterazione del ciclo for, ogni nodo $i + 1$, $i + 2$, ..., n è radice di un max heap.

1. **Inizializzazione:** prima della prima iterazione $i = \lfloor n/2 \rfloor$, e i nodi $\lfloor n/2 \rfloor + 1$, ..., n sono foglie, quindi radici di heap banali di un elemento.
2. **Conservazione:** i figli del nodo i hanno indici maggiori di i, quindi per l'invariante sono radici di max heap. È esattamente la condizione richiesta da MaxHeapify(A, i), che rende il nodo i radice di un max heap senza rovinare i nodi $i + 1$, ..., n. Decrementare i ristabilisce l'invariante.
3. **Conclusione:** all'uscita $i = 0$, quindi ogni nodo 1, 2, ..., n è radice di un max heap; in particolare lo è il nodo 1.

COSTO

Il conto ingenuo dà $n/2$ chiamate per $O(\log n)$, cioè $O(n \log n)$. Il conto preciso sfrutta il fatto che il costo di MaxHeapify dipende dall'altezza del nodo, e che i nodi alti sono pochi: in un heap di n elementi ci sono al più $\lceil n/2^{h+1} \rceil$ nodi di altezza h. Circa metà dei nodi ha altezza 0 (foglie, costo nullo), un quarto altezza 1, un ottavo altezza 2, e così via.

$$T(n) \leq \sum_{h=0..[\log n]} \lceil n/2^{h+1} \rceil \cdot O(h) = O(n \cdot \sum_{h=0.. \infty} h/2^h)$$

Resta da mostrare che la serie $\sum h/2^h$ converge a una costante. Si usa la serie nota, valida per $|x| < 1$:

$$\sum_{h=0.. \infty} h \cdot x^h = x/(1-x)^2$$

Con $x = 1/2$: $(1/2)/(1/2)^2 = (1/2)/(1/4) = 2$. Quindi $T(n) = O(2n) = \mathbf{O(n)}$.

DA RICORDARE

Il costo è dominato dai tanti nodi bassi, che costano poco, e non dai pochi nodi alti. È la stessa intuizione del caso "domina la radice" dell'albero di ricorsione, ma rovesciata: qui la somma è una serie che converge.

Heapsort

Invariante: all'inizio di ogni iterazione del ciclo for (con indice i), il sottoarray $A[1..i]$ è un max heap che contiene gli i elementi più piccoli di A , e $A[i+1..n]$ contiene gli $n-i$ elementi più grandi, ordinati.

Ogni iterazione sposta il massimo dell'heap, cioè il più grande tra gli i più piccoli, in posizione i , appena prima dei più grandi già ordinati, e ripara l'heap di dimensione $i-1$. All'uscita ($i=1$) $A[2..n]$ contiene gli $n-1$ elementi più grandi ordinati e $A[1]$ il più piccolo: tutto l'array è ordinato.

Costo: $O(n)$ per BuildMaxHeap più $n-1$ chiamate a MaxHeapify da $O(\log n)$, cioè $\mathbf{O(n \log n)}$.

Merge Sort

CORRETTEZZA DI MERGE

Invariante: all'inizio di ogni iterazione del ciclo for (con indice k), $A[p..k-1]$ contiene i $k-p$ elementi più piccoli di L e R , in ordine; inoltre $L[i]$ e $R[j]$ sono i più piccoli elementi dei rispettivi array non ancora copiati in A .

1. **Inizializzazione:** $k = p$, $A[p..p-1]$ è vuoto; $i = j = 1$ indicano i primi elementi di L e R , che sono i più piccoli perché L e R sono ordinati.
2. **Conservazione:** supponiamo $L[i] \leq R[j]$. Allora $L[i]$ è il più piccolo elemento non ancora copiato; copiandolo in $A[k]$, $A[p..k]$ contiene i $k-p+1$ elementi più piccoli in ordine. Incrementare i e k ristabilisce l'invariante. Il caso $R[j] < L[i]$ è simmetrico.
3. **Conclusione:** all'uscita $k = r+1$, quindi $A[p..r]$ contiene i $r-p+1$ elementi più piccoli di L e R , cioè tutti tranne le due sentinelle, in ordine.

CORRETTEZZA DI MERGE SORT

Per induzione sulla dimensione $n = r - p + 1$. Se $n \leq 1$ l'array è già ordinato e la procedura non fa nulla. Se $n > 1$, per ipotesi induttiva le due chiamate su metà array (di dimensione minore di n) ordinano correttamente le due metà, e Merge, come appena dimostrato, le fonde in un array ordinato.

COSTO

$T(n) = 2T(n/2) + \Theta(n)$: albero di ricorsione con $\log_2 n + 1$ livelli da $\Theta(n)$, oppure Teorema Master caso 2 ($n^{\log_2 2} = n = f(n)$). $\mathbf{\Theta(n \log n)}$ in ogni caso.

Quicksort

CASO PEGGIORE

Nel caso peggiore la partizione divide in q e $n-q-1$ elementi, con q scelto nel modo più sfavorevole:

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

Dimostriamo per sostituzione che $T(n) \leq cn^2$. Per ipotesi induttiva:

$$T(n) \leq \max_q (cq^2 + c(n-q-1)^2) + \Theta(n)$$

La funzione $q^2 + (n-q-1)^2$ è una parabola con la concavità verso l'alto, quindi sull'intervallo da 0 a $n-1$ ha il massimo agli **estremi**, $q=0$ oppure $q=n-1$, dove vale $(n-1)^2$. Quindi:

$$T(n) \leq c(n-1)^2 + \Theta(n) = cn^2 - c(2n-1) + \Theta(n) \leq cn^2$$

dove l'ultimo passaggio vale scegliendo c abbastanza grande perché $c(2n-1)$ superi il termine $\Theta(n)$. Quindi $T(n) = O(n^2)$, e poiché l'array ordinato richiede davvero $\Theta(n^2)$, il caso peggiore è $\Theta(n^2)$.

CASO MEDIO DI QUICKSORT RANDOMIZZATO

Il tempo di Quicksort è dominato dai **confronti** fatti dentro Partition: ogni altra operazione si fa un numero costante di volte per confronto o per chiamata. Contiamo quindi il numero atteso X di confronti.

Rinominiamo gli elementi in ordine crescente: $z_1 < z_2 < \dots < z_n$. Osservazioni chiave:

1. Due elementi vengono confrontati **solo** se uno dei due è il pivot del momento, e ogni coppia viene confrontata **al più una volta** (dopo la partizione il pivot non partecipa più).
2. Consideriamo l'insieme $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$. Finché nessun suo elemento è scelto come pivot, restano tutti nella stessa parte. Il primo elemento di Z_{ij} scelto come pivot decide: se è z_i o z_j , i due vengono confrontati; se è uno di quelli in mezzo, z_i e z_j finiscono in parti diverse e non si confronteranno mai.
3. Il pivot è casuale, quindi ogni elemento di Z_{ij} ha la stessa probabilità di essere il primo scelto, cioè $1/(j-i+1)$.

Ne segue:

$$\Pr\{z_i \text{ confrontato con } z_j\} = 2/(j-i+1)$$

Sommando su tutte le coppie (per la linearità del valore atteso) e ponendo $k = j - i$:

$$E[X] = \sum_{i=1..n-1} \sum_{j=i+1..n} 2/(j-i+1) = \sum_{i=1..n-1} \sum_{k=1..n-i} 2/(k+1) < \sum_{i=1..n-1} \sum_{k=1..n} 2/k$$

La somma interna $1 + 1/2 + 1/3 + \dots + 1/n$ è la **serie armonica**, che vale $\ln n + O(1) = O(\log n)$. Quindi:

$$E[X] < (n-1) \cdot O(\log n) = O(n \log n)$$

DA SAPERE: CHE COSA ABBIAMO DIMOSTRATO

Il numero atteso di confronti di RandomizedQuickSort, e quindi il suo tempo atteso, è **$O(n \log n)$** su **ogni** ingresso. L'idea portante è che due elementi "lontani" nell'ordine sono confrontati di rado, perché è probabile che venga scelto prima un pivot che li separa.

Selezione: perché il pivot buono dà tempo lineare

Per l'algoritmo Select (mediana delle mediane) si è visto che la chiamata ricorsiva finale lavora su al più $7n/10 + 6$ elementi, e quella per trovare la mediana delle mediane su $\lceil n/5 \rceil$. Mostriamo con la sostituzione che $T(n) \leq cn$. Per n abbastanza grande:

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an = 9cn/10 + 7c + an = cn + (-cn/10 + 7c + an) \end{aligned}$$

L'ultima parentesi è ≤ 0 se $cn/10 - 7c \geq an$, cioè $c \geq 10a \cdot n/(n-70)$. Per $n \geq 140$ il fattore $n/(n-70)$ è al più 2, quindi basta $c \geq 20a$. Il punto essenziale è che $1/5 + 7/10 = 9/10 < 1$.

Ottimizzazione con la ricorsione

Come si affronta un problema di ottimizzazione con una funzione ricorsiva che esplora tutte le scelte: il problema dello zaino nelle sue versioni, il costo esponenziale della ricerca esaustiva, la memoizzazione e una serie di problemi tipici su array, stringhe e alberi.

Lo schema generale

In un problema di ottimizzazione si cerca, tra tante soluzioni ammissibili, quella di valore massimo (o costo minimo). La ricorsione offre un metodo semplice e sempre applicabile: a ogni passo si prende **una decisione** (prendo questo oggetto o no? taglio qui o più avanti?), ogni decisione lascia un **problema residuo** dello stesso tipo ma più piccolo, lo si risolve ricorsivamente, e tra tutte le decisioni possibili si sceglie quella che porta al risultato migliore.

DA RICORDARE: LA RICETTA

1. Definire con precisione la funzione F : che cosa calcola e da quali parametri dipende (di solito un indice e una capacità o un intervallo).
2. Scrivere i **casi base**: le istanze così piccole che il valore ottimo è ovvio.
3. Elencare le **scelte** possibili al passo corrente e, per ciascuna, il sottoproblema residuo.
4. **Combinare**: il valore di F è il massimo (o minimo) tra le scelte, ciascuna valutata come "guadagno immediato più F sul residuo".

Questa ricetta funziona solo se il problema ha la **sottostruttura ottima**: una soluzione ottima, tolta la prima decisione, deve contenere una soluzione ottima del problema residuo. È la stessa proprietà su cui si basano la programmazione dinamica e il greedy.

Il problema dello zaino

Un escursionista prepara lo zaino per una gita. Lo zaino regge al massimo K chili, e davanti a lui ci sono n oggetti: l'oggetto i pesa w_i e ha per lui un'utilità v_i . Vuole scegliere gli oggetti in modo da massimizzare l'utilità totale senza superare K .

DEFINIZIONE: ZAINO 0/1

Dati n oggetti con pesi $w_i > 0$ e valori $v_i \geq 0$, e una capacità K , trovare un sottoinsieme S degli oggetti che massimizza $\sum_{i \in S} v_i$ con il vincolo $\sum_{i \in S} w_i \leq K$.

UN CASO FACILE: PESI TUTTI UGUALI

Se tutti gli oggetti pesano 1 chilo, lo zaino contiene al più K oggetti e la scelta è ovvia: si prendono i K oggetti di valore più alto. Se gli oggetti sono già ordinati per valore crescente, basta sommare gli ultimi K valori. Qui non serve esplorare alternative, perché sostituire un oggetto con uno più prezioso migliora sempre il totale senza cambiare il peso: la scelta locale migliore è anche quella globalmente ottima (è un esempio di strategia greedy). Anche scritta in forma ricorsiva, con $F(i, k) = v_i + F(i - 1, k - 1)$ e casi base $F(0, k) = F(i, 0) = 0$, fa una sola chiamata per livello e costa $O(K)$.

IL CASO GENERALE: PRENDERE O LASCIARE

Con pesi diversi l'oggetto più prezioso può essere anche pesantissimo e "rubare" lo spazio a due oggetti leggeri che insieme valgono di più. Bisogna davvero confrontare le alternative. Definiamo $F(i, k)$ come il valore massimo ottenibile usando solo i primi i oggetti, con capacità residua k . Per l'oggetto i ci sono due scelte:

- **lasciarlo**: il valore è $F(i - 1, k)$;
- **prenderlo**, se ci sta ($w_i \leq k$): il valore è $v_i + F(i - 1, k - w_i)$.

$$F(i, k) = 0 \quad \text{se } i = 0 \text{ oppure } k = 0$$

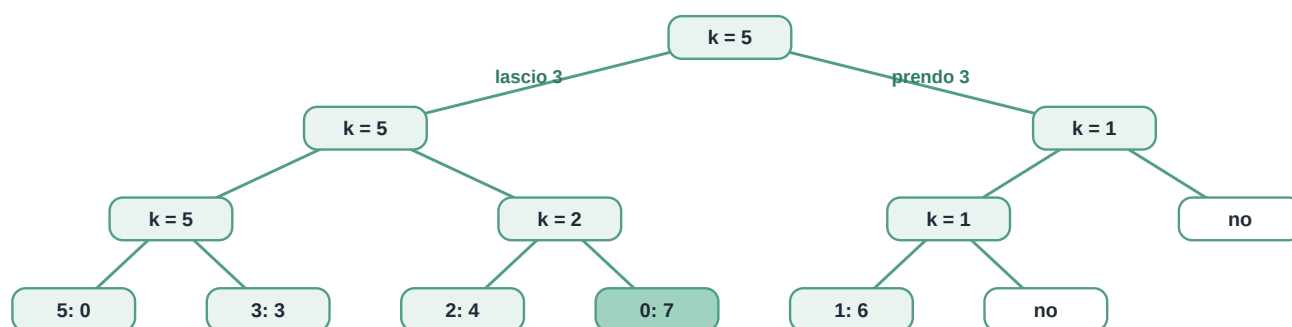
$$F(i, k) = F(i - 1, k) \quad \text{se } w_i > k$$

$$F(i, k) = \max(F(i - 1, k), \quad v_i + F(i - 1, k - w_i)) \quad \text{altrimenti}$$

Un modo equivalente, usato spesso nel codice, è non controllare se l'oggetto ci sta e restituire un valore "meno infinito" (un numero molto negativo) quando la capacità diventa negativa: così le scelte impossibili perdono sempre il confronto con il max.

L'ALBERO DELLE SCELTE

Con tre oggetti (peso, valore) = (2, 3), (3, 4), (4, 6) e capacità $K = 5$, la ricorsione esplora questo albero, in cui ogni livello decide un oggetto (a sinistra lo si lascia, a destra lo si prende) e ogni nodo è etichettato con la capacità residua.



Albero delle chiamate: il primo livello decide l'oggetto 3, il secondo l'oggetto 2, il terzo l'oggetto 1; a sinistra si lascia, a destra si prende. In ogni foglia la capacità rimasta e il valore raccolto; le caselle "no" sono scelte impossibili. Il massimo è 7, con gli oggetti 1 e 2 (foglia evidenziata).

DA SAPERE: COSTO DELLA RICERCA ESAUSTIVA

Ogni chiamata ne genera fino a due su $i - 1$: $T(n) = 2T(n - 1) + \Theta(1)$, cioè $\Theta(2^n)$, come le Torri di Hanoi. È inevitabile se si esplorano davvero tutti i 2^n sottoinsiemi. Ma le chiamate distinte sono solo quelle con parametri (i, k) diversi, al più $(n + 1)(K + 1)$: molte vengono ripetute. Salvando i risultati (**memoizzazione**) il costo scende a $O(n \cdot K)$, ed è la programmazione dinamica.

RICOSTRUIRE LA SOLUZIONE

La funzione restituisce il valore ottimo, non gli oggetti. Per sapere quali oggetti prendere si ripercorre la ricorsione: partendo da (n, K) , se $F(i, k) = F(i - 1, k)$ l'oggetto i non è servito e si passa a $(i - 1, k)$; altrimenti è stato preso e si passa a $(i - 1, k - w_i)$.

Implementazione: ricorsione semplice e memoizzata

```
#include <iostream>
using namespace std;

const int MENO_INF = -1000000;          // rappresenta "meno infinito"

// Valore massimo con i primi i oggetti e capacita' k (ricerca esaustiva)
int zaino(int w[], int v[], int i, int k) {
    if (k < 0) return MENO_INF;          // scelta impossibile
    if (i == 0 || k == 0) return 0;      // nessun oggetto o zaino pieno
    int lascio = zaino(w, v, i - 1, k);
    int prendo = v[i] + zaino(w, v, i - 1, k - w[i]);
    if (lascio > prendo) return lascio;
    else return prendo;
}

int memo[20][100];                      // -1 = non ancora calcolato

// Stessa funzione, ma ogni (i, k) viene calcolato una sola volta
int zainoMemo(int w[], int v[], int i, int k) {
    if (k < 0) return MENO_INF;
    if (i == 0 || k == 0) return 0;
    if (memo[i][k] != -1) return memo[i][k]; // gia' calcolato: lo riuso
    int lascio = zainoMemo(w, v, i - 1, k);
    int prendo = v[i] + zainoMemo(w, v, i - 1, k - w[i]);
    if (lascio > prendo) memo[i][k] = lascio;
    else memo[i][k] = prendo;
    return memo[i][k];
}
```

Nella versione memoizzata cambiano solo tre righe: il controllo iniziale sulla tabella e il salvataggio del risultato. È il modo più semplice di trasformare una ricorsione esponenziale in un algoritmo polinomiale, quando i sottoproblemi si ripetono.

Problemi tipici risolti

In tutti i problemi che seguono il ragionamento è lo stesso: si definisce F , si individuano le scelte e si scrive la ricorrenza. Per ciascuno è indicato il costo della ricorsione semplice e quello con memoizzazione.

SU ARRAY: IL TAGLIO DELL'ASTA

Un'asta lunga n metri si può vendere intera o tagliata in pezzi di lunghezza intera; un pezzo lungo i metri si vende al prezzo p_i . Qual è il ricavo massimo? La scelta è **la lunghezza del primo pezzo**: se è i , si incassa p_i e resta un'asta di $n - i$ metri da tagliare nel modo migliore.

$$R(0) = 0 \quad R(n) = \max_{1 \leq i \leq n} (p_i + R(n - i))$$

Con i prezzi $p_1 = 2$, $p_2 = 5$, $p_3 = 7$, $p_4 = 9$ e $n = 4$: $R(1) = 2$, $R(2) = \max(2 + 2, 5) = 5$, $R(3) = \max(2 + 5, 5 + 2, 7) = 7$, $R(4) = \max(2 + 7, 5 + 5, 7 + 2, 9) = 10$, tagliando in due pezzi da 2. La ricorsione semplice fa $T(n) = T(n - 1) + T(n - 2) + \dots + T(0) + \Theta(n)$ chiamate, che vale $\Theta(2^n)$; con memoizzazione i sottoproblemi distinti sono $n + 1$, ciascuno con $O(n)$ scelte: **$O(n^2)$** .

SU GRIGLIE: IL CAMMINO DI VALORE MASSIMO

In una griglia di m righe e n colonne ogni cella contiene delle monete. Si parte dalla cella in alto a sinistra e si arriva in basso a destra muovendosi solo verso destra o verso il basso. Quante monete si possono raccogliere al massimo? In ogni cella (i, j) si arriva o da sopra o da sinistra, quindi:

$$M(i, j) = G[i][j] + \max(M(i - 1, j), M(i, j - 1))$$

con M fuori dalla griglia pari a meno infinito e $M(1, 1) = G[1][1]$. Senza memoria il numero di cammini esplorati è esponenziale; con memoizzazione si calcolano $m \cdot n$ valori in tempo costante: $O(m \cdot n)$.

SU STRINGHE: LA SOTTOSEQUENZA PALINDROMA PIÙ LUNGA

Una **sottosequenza** si ottiene cancellando caratteri senza cambiare l'ordine dei restanti. Vogliamo la lunghezza della più lunga sottosequenza palindroma di $x[1..n]$. Sia $P(i, j)$ la risposta per la sottostringa $x[i..j]$. La scelta riguarda i due estremi:

- se $x[i] = x[j]$, possono fare da "guscio" esterno del palindromo: $P(i, j) = 2 + P(i + 1, j - 1)$;
- altrimenti almeno uno dei due non fa parte della soluzione: $P(i, j) = \max(P(i + 1, j), P(i, j - 1))$.

Casi base: $P(i, i) = 1$ (un carattere è un palindromo) e $P(i, j) = 0$ se $i > j$. Per $x = \text{"carta"}$, la risposta è 3 ("ara" oppure "ata"). Con memoizzazione: n^2 sottoproblemi, $O(n^2)$.

SU ALBERI: IL CAMMINO RADICE FOGLIA PIÙ PESANTE

In un albero binario con un valore in ogni nodo, vogliamo il cammino dalla radice a una foglia con somma massima. La scelta è **da che parte scendere**:

$$S(x) = x.\text{valore} + \max(S(x.\text{sinistro}), S(x.\text{destra}))$$

con l'attenzione che un figlio vuoto non è una foglia: se manca un figlio si scende per forza nell'altro, e S di una foglia è il suo valore. Qui i sottoproblemi non si ripetono (ogni nodo appartiene a un solo sottoalbero), quindi la ricorsione semplice è già ottima: $T(n) = T(n_{sx}) + T(n_{dx}) + O(1) = O(n)$.

```
// Taglio dell'asta: p[i] = prezzo di un pezzo lungo i
int asta(int p[], int n) {
    if (n == 0) return 0;
    int migliore = MENO_INF;
    for (int i = 1; i <= n; i++) {           // lunghezza del primo pezzo
        int r = p[i] + asta(p, n - i);
        if (r > migliore) migliore = r;
    }
    return migliore;
}

// Lunghezza della piu' lunga sottosequenza palindroma di x[i..j]
int palindroma(char x[], int i, int j) {
    if (i > j) return 0;
    if (i == j) return 1;
    if (x[i] == x[j]) return 2 + palindroma(x, i + 1, j - 1);
    int a = palindroma(x, i + 1, j);
    int b = palindroma(x, i, j - 1);
    if (a > b) return a;
    else return b;
}

struct Nodo {
    int valore;
    Nodo* sx;
    Nodo* dx;
};

// Somma massima di un cammino dalla radice x a una foglia
int camminoMax(Nodo* x) {
    if (x->sx == NULL && x->dx == NULL) return x->valore; // foglia
    if (x->sx == NULL) return x->valore + camminoMax(x->dx);
    if (x->dx == NULL) return x->valore + camminoMax(x->sx);
    int a = camminoMax(x->sx);
    int b = camminoMax(x->dx);
    if (a > b) return x->valore + a;
```

```

    else return x->valore + b;
}

int main() {
    int w[] = {0, 2, 3, 4}, v[] = {0, 3, 4, 6};    // oggetti 1..3
    for (int i = 0; i < 20; i++)
        for (int k = 0; k < 100; k++) memo[i][k] = -1;
    cout << zaino(w, v, 3, 5) << " " << zainoMemo(w, v, 3, 5) << endl;

    int p[] = {0, 2, 5, 7, 9};
    cout << asta(p, 4) << endl;

    char x[] = "carta";
    cout << palindroma(x, 0, 4) << endl;

    // albero: radice 5, figli 3 e 8, figlio sinistro di 3 vale 10
    Nodo a = {10, NULL, NULL}, b = {3, &a, NULL}, c = {8, NULL, NULL};
    Nodo r = {5, &b, &c};
    cout << camminoMax(&r) << endl;
    return 0;
}

```

DA SAPERE: RICONOSCERE IL TIPO DI PROBLEMA

- Se le scelte lasciano sottoproblemi che **si ripetono** (zaino, asta, griglia, stringhe), la ricorsione semplice è esponenziale e va memoizzata o trasformata in programmazione dinamica.
- Se una scelta locale è **sempre sicura** (zaino con pesi uguali, zaino frazionario), basta un algoritmo greedy, senza esplorare alternative.
- Se i sottoproblemi sono **disgiunti** (alberi, divide et impera), la ricorsione semplice è già efficiente.

Analisi dell'hashing

Le dimostrazioni dei tempi medi delle tabelle hash: ricerca con e senza successo nella concatenazione e nell'indirizzamento aperto, sotto le ipotesi di hashing uniforme, e il comportamento dell'hashing universale.

Le ipotesi

Nel caso peggiore una tabella hash è pessima: se tutte le n chiavi finiscono nello stesso slot, cercare costa $\Theta(n)$. L'analisi interessante è quindi quella del caso **medio**, che richiede un'ipotesi su come la funzione hash distribuisce le chiavi.

DEFINIZIONE: HASHING UNIFORME SEMPLICE

Ogni chiave ha la **stessa probabilità** di essere mappata in ciascuno degli m slot, **indipendentemente** da dove sono state mappate le altre chiavi. Si assume anche che $h(k)$ si calcoli in tempo $\Theta(1)$.

Indichiamo con n il numero di chiavi presenti, con m il numero di slot e con $\alpha = n/m$ il **fattore di carico**. Con la concatenazione, chiamiamo n_j la lunghezza della lista nello slot j : vale $n_0 + n_1 + \dots + n_{m-1} = n$, e sotto l'ipotesi di hashing uniforme semplice il valore atteso di ogni n_j è $n/m = \alpha$.

Concatenazione: ricerca senza successo

DA SAPERE: TEOREMA

Con la concatenazione e l'hashing uniforme semplice, una ricerca **senza successo** richiede in media tempo $\Theta(1 + \alpha)$.

Dimostrazione. Una chiave k non presente nella tabella ha la stessa probabilità di essere mappata in uno qualsiasi degli m slot. Per stabilire che non c'è, bisogna scorrere **tutta** la lista $T[h(k)]$, la cui lunghezza attesa è α . Si aggiunge il tempo $\Theta(1)$ per calcolare $h(k)$ e accedere allo slot. Totale: $\Theta(1 + \alpha)$.

Il termine 1 serve a non dimenticare il costo del calcolo dell'hash quando α è molto piccolo: con una tabella quasi vuota la ricerca costa comunque qualcosa.

Concatenazione: ricerca con successo

Qui la situazione è diversa: la chiave cercata x è nella tabella, e la ricerca si ferma quando la trova. Le liste di trabocco ricevono i nuovi elementi **in testa**, quindi davanti a x ci sono esattamente gli elementi finiti nella stessa lista **dopo** di lui. Supponiamo che x sia scelto a caso, con uguale probabilità, tra le n chiavi presenti.

DA SAPERE: TEOREMA

Con la concatenazione e l'hashing uniforme semplice, una ricerca **con successo** richiede in media tempo $\Theta(1 + \alpha)$; più precisamente il numero atteso di elementi esaminati è $1 + \alpha/2 - \alpha/(2n)$.

Dimostrazione passo per passo.

1. Numeriamo le chiavi nell'ordine di inserimento: $x_1, x_2, \dots, x_n$.
2. Se cerchiamo x_i , gli elementi esaminati sono x_i stesso più quelli tra $x_{i+1}, \dots, x_n$ finiti nella stessa lista.

3. Per $j > i$ definiamo la variabile indicatrice X_{ij} , che vale 1 se $h(x_i) = h(x_j)$ e 0 altrimenti. Per l'hashing uniforme semplice, $\Pr\{h(x_i) = h(x_j)\} = 1/m$, quindi $E[X_{ij}] = 1/m$.

4. Il numero atteso di elementi esaminati, facendo la media su tutte le n chiavi, è:

$$E = (1/n) \sum_{i=1..n} (1 + \sum_{j=i+1..n} E[X_{ij}]) = (1/n) \sum_{i=1..n} (1 + (n-i)/m)$$

5. Separiamo le due somme. La prima vale $n \cdot 1 = n$, divisa per n dà 1. Nella seconda $\sum (n-i)$ per i da 1 a n vale $(n-1) + (n-2) + \dots + 0 = n(n-1)/2$:

$$E = 1 + (1/(nm)) \cdot n(n-1)/2 = 1 + (n-1)/(2m)$$

6. Scriviamo il risultato in funzione di $\alpha = n/m$:

$$E = 1 + n/(2m) - 1/(2m) = 1 + \alpha/2 - \alpha/(2n)$$

Il tempo totale, includendo il calcolo dell'hash, è $\Theta(2 + \alpha/2 - \alpha/(2n)) = \Theta(1 + \alpha)$.

DA RICORDARE: CONSEGUENZA PRATICA

Se il numero di slot è proporzionale al numero di elementi, $n = O(m)$, allora $\alpha = n/m = O(1)$, e tutte le operazioni del dizionario (inserimento, cancellazione con liste doppie, ricerca) costano **$O(1)$ in media**.

Indirizzamento aperto

Nell'indirizzamento aperto l'ipotesi è più forte, perché riguarda l'intera sequenza di ispezione.

DEFINIZIONE: HASHING UNIFORME

La sequenza di ispezione $\{h(k, 0), \dots, h(k, m-1)\}$ di ogni chiave è, con uguale probabilità, una qualunque delle $m!$ permutazioni di $\{0, 1, \dots, m-1\}$.

Il doppio hashing si avvicina a questa ipotesi; scansione lineare e quadratica no (generano solo m sequenze diverse). Ora $\alpha = n/m \leq 1$, e supponiamo $\alpha < 1$, cioè che ci sia almeno uno slot libero.

RICERCA SENZA SUCCESSO

DA SAPERE: TEOREMA

Con l'indirizzamento aperto e l'hashing uniforme, con $\alpha < 1$, il numero atteso di ispezioni in una ricerca **senza successo** è al più **$1/(1-\alpha)$** .

Intuizione. La prima ispezione si fa sempre. Con probabilità circa α lo slot è occupato e serve una seconda ispezione. Con probabilità circa α^2 anche il secondo è occupato e ne serve una terza. E così via: il numero atteso è circa $1 + \alpha + \alpha^2 + \alpha^3 + \dots$, una serie geometrica che vale $1/(1-\alpha)$.

Dimostrazione.

1. Sia X il numero di ispezioni. Una ricerca senza successo si ferma al primo slot vuoto, quindi $X \geq i$ se e solo se le prime $i-1$ ispezioni hanno trovato slot occupati.
2. La prima ispezione trova uno slot occupato con probabilità n/m . Dato che lo era, la seconda ispeziona uno degli $m-1$ slot rimasti, di cui $n-1$ occupati: probabilità $(n-1)/(m-1)$. E così via:

$$\Pr\{X \geq i\} = (n/m) \cdot ((n-1)/(m-1)) \cdot \dots \cdot ((n-i+2)/(m-i+2)) \leq (n/m)^{i-1} = \alpha^{i-1}$$

perché ogni fattore $(n-j)/(m-j)$ è minore o uguale a n/m quando $n < m$.

3. Per una variabile che assume valori interi positivi, $E[X] = \sum_{i \geq 1} \Pr\{X \geq i\}$. Quindi:

$$E[X] \leq \sum_{i=1.. \infty} \alpha^{i-1} = \sum_{i=0.. \infty} \alpha^i = 1/(1-\alpha)$$

Poiché **inserire** una chiave significa fare una ricerca senza successo e poi scrivere nel primo slot vuoto, anche l'inserimento richiede in media al più $1/(1 - \alpha)$ ispezioni.

RICERCA CON SUCCESSO

DA SAPERE: TEOREMA

Con l'indirizzamento aperto e l'hashing uniforme, con $\alpha < 1$, il numero atteso di ispezioni in una ricerca **con successo** è al più $(1/\alpha) \ln(1/(1 - \alpha))$.

Dimostrazione passo per passo.

1. Cercare una chiave k presente segue esattamente la stessa sequenza di ispezioni fatta quando k è stata inserita.
2. Se k è stata la $(i + 1)$ esima chiave inserita, al momento dell'inserimento la tabella conteneva i chiavi, con fattore di carico i/m . Per il teorema precedente l'inserimento ha richiesto al più $1/(1 - i/m) = m/(m - i)$ ispezioni.
3. Facendo la media sulle n chiavi presenti:

$$(1/n) \sum_{i=0..n-1} m/(m - i) = (m/n) \sum_{i=0..n-1} 1/(m - i) = (1/\alpha) \sum_{k=m-n+1..m} 1/k$$

dove si è posto $k = m - i$ (al variare di i da 0 a $n - 1$, k va da m fino a $m - n + 1$).

4. Una somma di $1/k$ su interi consecutivi si maggiora con un integrale, perché $1/x$ è decrescente:

$$\sum_{k=m-n+1..m} 1/k \leq \int_{m-n}^m (1/x) dx = \ln(m/(m - n)) = \ln(1/(1 - \alpha))$$

5. Moltiplicando per $1/\alpha$ si ottiene il risultato.

A	SENZA SUCCESSO: $1/(1 - A)$	CON SUCCESSO: $(1/A) \ln(1/(1 - A))$
0,5	2	1,39
0,75	4	1,85
0,9	10	2,56
0,99	100	4,65

DA RICORDARE

Una ricerca con successo è sempre più veloce di una senza successo, perché si ferma prima. Ma entrambe peggiorano molto quando la tabella si riempie: in pratica si tiene α sotto 0,7 circa, e quando lo si supera si ricostruisce una tabella più grande (rehashing).

Hashing universale

Con l'hashing universale la casualità non sta più nelle chiavi ma nella **scelta della funzione**, presa a caso da una famiglia universale H (per ogni coppia di chiavi distinte, la probabilità di collisione è al più $1/m$).

DA SAPERE: TEOREMA

Sia h scelta a caso da una famiglia universale e usata con la concatenazione in una tabella di m slot con n chiavi. Se la chiave k **non** è nella tabella, la lunghezza attesa della lista in cui viene mappata è al più α . Se k è nella tabella, è al più $1 + \alpha$.

Dimostrazione. Per ogni coppia di chiavi distinte k e l sia X_{kl} l'indicatrice dell'evento $h(k) = h(l)$: per l'universalità $E[X_{kl}] \leq 1/m$. Il numero di chiavi diverse da k che stanno nella lista di k è $Y_k = \sum_{l \neq k} X_{kl}$, quindi per la linearità del valore atteso:

- se k non è nella tabella, le chiavi l sono n : $E[Y_k] \leq n/m = \alpha$;
- se k è nella tabella, le altre chiavi sono $n - 1$ e la lista contiene anche k stessa: la lunghezza attesa è $1 + E[Y_k] \leq 1 + (n - 1)/m < 1 + \alpha$.

Ne segue che, con $n = O(m)$, ogni operazione costa $O(1)$ in media **per qualunque insieme di chiavi**: nessun avversario può scegliere chiavi "cattive", perché non sa quale funzione verrà estratta.

Riepilogo

SCHEMA	RICERCA SENZA SUCCESSO	RICERCA CON SUCCESSO
Concatenazione, hashing uniforme semplice	$\Theta(1 + \alpha)$	$\Theta(1 + \alpha)$
Indirizzamento aperto, hashing uniforme	al più $1/(1 - \alpha)$ ispezioni	al più $(1/\alpha) \ln(1/(1 - \alpha))$
Concatenazione, hashing universale	lista attesa $\leq \alpha$	lista attesa $\leq 1 + \alpha$

Alberi rosso neri: altezza e correttezza

Perché un albero rosso nero è sempre basso, perché la riparazione dopo un inserimento o una cancellazione ristabilisce davvero tutte le proprietà, e come verificare con una funzione ricorsiva che un albero sia rosso nero.

Richiamo delle proprietà

Un albero binario di ricerca è **rosso nero** se: (1) ogni nodo è rosso o nero; (2) la radice è nera; (3) ogni foglia NIL è nera; (4) i figli di un nodo rosso sono neri; (5) per ogni nodo, tutti i cammini da esso alle foglie discendenti contengono lo stesso numero di nodi neri. L'**altezza nera** $bh(x)$ è il numero di nodi neri su un cammino da x a una foglia, x escluso (le foglie NIL contano). Le procedure di inserimento e cancellazione sono descritte nella dispensa di progettazione; qui ne dimostriamo le proprietà.

L'altezza è logaritmica

DA SAPERE: LEMMA

Un albero rosso nero con n nodi interni ha altezza al più $2 \log_2(n + 1)$.

La dimostrazione ha due passi: prima si mostra che un sottoalbero con altezza nera grande deve avere molti nodi, poi che l'altezza nera è almeno metà dell'altezza.

PASSO 1: UN SOTTOALBERO DI RADICE x CONTIENE ALMENO $2^{bh(x)} - 1$ NODI INTERNI

Per induzione sull'altezza di x .

- Base:** se x ha altezza 0, è una foglia NIL. Allora $bh(x) = 0$ e il sottoalbero contiene $2^0 - 1 = 0$ nodi interni: vero.
- Passo:** sia x un nodo interno di altezza positiva; ha due figli. Ogni figlio ha altezza nera $bh(x)$ se è rosso, $bh(x) - 1$ se è nero: in ogni caso **almeno** $bh(x) - 1$.
- I figli hanno altezza minore di x , quindi per ipotesi induttiva ciascuno dei due sottoalberi contiene almeno $2^{bh(x)-1} - 1$ nodi interni.
- Sommando i due sottoalberi e x stesso:

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2 \cdot 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$$

PASSO 2: L'ALTEZZA NERA DELLA RADICE È ALMENO $h/2$

Sia h l'altezza dell'albero. Per la proprietà 4 un nodo rosso ha sempre figli neri, quindi su un cammino dalla radice a una foglia non ci sono due rossi consecutivi: **almeno metà** dei nodi del cammino, radice esclusa, sono neri. Prendendo il cammino più lungo, $bh(\text{radice}) \geq h/2$.

CONCLUSIONE

Applicando il passo 1 alla radice e poi il passo 2:

$$n \geq 2^{bh(\text{radice})} - 1 \geq 2^{h/2} - 1$$

Si porta l'1 a sinistra e si prende il logaritmo in base 2 di entrambi i lati:

$$n + 1 \geq 2^{h/2} \Rightarrow \log_2(n + 1) \geq h/2 \Rightarrow h \leq 2 \log_2(n + 1)$$

DA RICORDARE: CONSEGUENZA

Ricerca, minimo, massimo, successore e predecessore, che in un albero di ricerca costano $O(h)$, in un albero rosso nero costano $O(\log n)$. Il fattore 2 è il prezzo della flessibilità: un albero rosso nero non è perfettamente bilanciato, ma il ramo più lungo è al più il doppio del più corto.

Correttezza dell'inserimento

Ricordiamo che il nuovo nodo z viene inserito come in un BST e colorato di rosso; poi `RBInsertFixup` ripara le violazioni.

QUALI PROPRIETÀ POSSONO ESSERE VIOLATE

- Le proprietà 1 e 3 restano vere: z è rosso e i suoi figli sono la sentinella NIL, nera.
- La proprietà 5 resta vera: z è rosso e ha preso il posto di una foglia nera, sostituendola con un nodo rosso con due figli NIL neri. Nessun cammino cambia il numero di neri.
- La proprietà 2 è violata se z è la radice (albero vuoto prima dell'inserimento).
- La proprietà 4 è violata se il padre di z è rosso.

Le due violazioni non possono presentarsi insieme: se z è la radice non ha padre.

L'INVARIANTE DEL CICLO

DEFINIZIONE: INVARIANTE DI `RBINSERTFIXUP`

All'inizio di ogni iterazione del ciclo `while`:

1. il nodo z è rosso;
2. se $z.padre$ è la radice, allora $z.padre$ è nero;
3. c'è al più **una** violazione delle proprietà, ed è o della proprietà 2 (z è la radice ed è rosso) o della proprietà 4 (z e $z.padre$ sono entrambi rossi).

- **Inizializzazione.** Prima della prima iterazione z è il nodo appena inserito, rosso; la radice non è stata toccata ed era nera; le violazioni sono al più una, come appena visto.
- **Conclusione.** Il ciclo termina quando $z.padre$ è nero. Allora la proprietà 4 non è violata, e l'unica violazione possibile è della proprietà 2, che l'ultima riga (`T.radice.colore = NERO`) ripara.
- **Conservazione.** Si esaminano i tre casi. Nel ciclo $z.padre$ è rosso, quindi non è la radice (per il punto 2) e il nonno esiste ed è nero.

Caso 1, zio rosso. Padre e zio diventano neri, il nonno rosso. Sui cammini che passano per il nonno, prima c'era un nero (il nonno) e dopo c'è un nero (padre o zio) a quel livello: le altezze nere non cambiano. Ora il nonno, nuovo z , è rosso (punto 1 dell'invariante) e l'unica violazione possibile è tra lui e suo padre, oppure, se il nonno è la radice, della proprietà 2. Il ciclo prosegue **due livelli più in alto**.

Casi 2 e 3, zio nero. La rotazione del caso 2 coinvolge due nodi rossi (z e suo padre), quindi non cambia le altezze nere, e trasforma la situazione nel caso 3. Nel caso 3 il padre diventa nero e sale al posto del nonno, che diventa rosso e scende dal lato dello zio nero. Contando i neri su ciascun sottoalbero coinvolto, ogni cammino passa ancora per esattamente un nero in quella zona, quindi la proprietà 5 è conservata; non ci sono più due rossi consecutivi, e il nuovo nodo in cima è nero, quindi il ciclo termina.

COSTO

Il ciclo si ripete solo nel caso 1, e ogni volta z sale di due livelli: al più $O(h) = O(\log n)$ iterazioni. I casi 2 e 3 si eseguono al più una volta, con una rotazione ciascuno. Quindi l'inserimento costa $O(\log n)$ e fa **al più 2 rotazioni**.

Correttezza della cancellazione

QUALE PROPRIETÀ SI ROMPE

Sia y il nodo che esce davvero dall'albero (z stesso se ha al più un figlio, altrimenti il suo successore) e x il nodo che ne prende il posto. Se y era **rosso** nulla si rompe: nessuna altezza nera cambia (si è tolto un rosso), non si creano due rossi adiacenti (x prende il posto di un rosso, quindi suo padre era nero o è la radice) e la radice resta nera (un rosso non poteva essere la radice). Se y era **nero** si possono violare:

- la proprietà 5: ogni cammino che passava per y ha ora un nero in meno;
- la proprietà 4: se x e il nuovo padre di x sono entrambi rossi;
- la proprietà 2: se y era la radice e x , rosso, diventa la nuova radice.

L'IDEA DEL NERO IN PIÙ

Per sistemare la proprietà 5 si "attribuisce" a x un **nero extra**: contando così, ogni cammino ha di nuovo il numero giusto di neri. Il nodo x diventa "doppio nero" (se era nero) o "rosso e nero" (se era rosso). Questo viola solo la proprietà 1 (un nodo deve avere un solo colore), e `RBDeleteFixup` ha il compito di **eliminare il nero extra**:

- se x è rosso e nero, basta colorarlo di nero (il ciclo non parte nemmeno);
- se x è la radice, il nero extra si elimina e basta: togliere un nero da tutti i cammini insieme non viola nulla;
- altrimenti si applicano i quattro casi, che spostano il nero extra verso l'alto o lo assorbono.

PERCHÉ I QUATTRO CASI FUNZIONANO

In ogni caso si può verificare, contando i neri dalla radice della zona modificata fino a ciascuno dei sottoalberi appesi sotto (tenendo conto del nero extra su x), che **il conteggio per ogni sottoalbero è lo stesso prima e dopo** la trasformazione. Le rotazioni e le ricolorazioni sono scelte apposta per questo.

- **Caso 1** (fratello w rosso): scambiare i colori di w e del padre e ruotare non cambia i conteggi; il nuovo fratello è nero. Si passa ai casi 2, 3 o 4.
- **Caso 2** (w nero con figli neri): si toglie un nero sia da x sia da w (w diventa rosso) e lo si aggiunge al padre. Il nero extra **sale di un livello**. Se si arriva qui dal caso 1 il padre è rosso, quindi il ciclo termina subito.
- **Caso 3** (w nero, figlio vicino rosso, figlio lontano nero): una rotazione su w e uno scambio di colori portano al caso 4 senza cambiare i conteggi.
- **Caso 4** (w nero con figlio lontano rosso): la rotazione sul padre e le ricolorazioni aggiungono un nero sul lato di x , che assorbe il nero extra. Il ciclo termina.

COSTO

Il ciclo si ripete solo nel caso 2, salendo ogni volta di un livello: $O(\log n)$ iterazioni. Gli altri casi portano alla terminazione dopo al più tre rotazioni (caso 1, poi 3, poi 4). La cancellazione costa quindi **$O(\log n)$ con al più 3 rotazioni**.

Verificare che un albero sia rosso nero

Un esercizio tipico chiede di scrivere una procedura che verifichi le proprietà. La proprietà 5 è la più delicata, ma si controlla bene con una funzione ricorsiva che restituisce il **numero di nodi neri** su un cammino dal nodo x (compreso) fino a una foglia, oppure il valore speciale -1 se il sottoalbero di x non è valido. Il trucco è far restituire alla funzione un'informazione (il conteggio) che il padre può confrontare tra i due figli.

```

1  ContaNeri(x)                                // neri da x (compreso) alle foglie, o -1
2      if x = T.nil then return 1              // la foglia NIL e' nera e conta 1
3      s = ContaNeri(x.sinistro)
4      d = ContaNeri(x.destro)
5      if s = -1 or d = -1 then return -1 // un sottoalbero non e' valido
6      if s != d then return -1                // violata la proprieta' 5
7      if x.colore = ROSSO and
8          (x.sinistro.colore = ROSSO or x.destro.colore = ROSSO) then
9          return -1                            // violata la proprieta' 4
10     if x.colore = NERO then return s + 1
11     else return s                            // un nodo rosso non si conta
12
13 VerificaRB(T)
14     return T.radice.colore = NERO and ContaNeri(T.radice) != -1

```

Seguiamo la logica. Una foglia NIL restituisce 1. Un nodo confronta i conteggi dei due figli: se sono diversi, i cammini a sinistra e a destra hanno un numero diverso di neri e la proprietà 5 è violata. Se un nodo rosso ha un figlio rosso, è violata la proprietà 4. Altrimenti il nodo restituisce il conteggio dei figli, aumentato di uno se è nero. Il valore restituito dalla radice, meno uno (la radice stessa, che è nera), è $bh(radice)$.

Il costo è $\Theta(n)$: è una visita in cui ogni nodo fa un lavoro costante, $T(n) = T(n_{sx}) + T(n_{dx}) + \Theta(1)$. Restano da controllare a parte la proprietà di albero di ricerca (con una visita simmetrica che verifichi che le chiavi escano in ordine crescente) e il colore della radice, già incluso in VerificaRB.

Programmazione dinamica: sottostruttura ottima e LCS

Come si dimostra la sottostruttura ottima, come si passa dalla ricorrenza alla tabella e alla ricostruzione della soluzione, applicato alla più lunga sottosequenza comune e alla distanza di editing tra due stringhe.

La sottostruttura ottima e come si dimostra

Un problema ha **sottostruttura ottima** se una soluzione ottima contiene al suo interno soluzioni ottime dei sottoproblemi. È la condizione che permette di scrivere una ricorrenza: il valore ottimo del problema si esprime tramite i valori ottimi dei sottoproblemi.

La dimostrazione segue quasi sempre lo stesso schema, detto **taglia e incolla** (cut and paste):

1. si mostra che una soluzione consiste nel fare una **scelta** (il punto di taglio, l'oggetto da prendere, l'ultimo carattere) che lascia uno o più sottoproblemi;
2. si suppone di conoscere la scelta fatta da una soluzione ottima;
3. si mostra che le parti della soluzione relative ai sottoproblemi devono essere ottime: per **assurdo**, se una non lo fosse, la si potrebbe "tagliare via" e "incollare" al suo posto una soluzione migliore del sottoproblema, ottenendo una soluzione del problema originale migliore di quella ottima. Contraddizione.

DA RICORDARE: QUANDO NON C'È SOTTOSTRUTTURA OTTIMA

Non tutti i problemi l'hanno. Il **cammino minimo** in un grafo sì: un sottocammino di un cammino minimo è minimo. Il **cammino semplice più lungo** no: il cammino più lungo da u a v passando per w non è in generale formato dal più lungo da u a w e dal più lungo da w a v , perché i due potrebbero condividere nodi e il risultato non sarebbe più semplice. I sottoproblemi non sono **indipendenti**, e la programmazione dinamica non si applica.

QUANTO COSTA UNA SOLUZIONE DI PROGRAMMAZIONE DINAMICA

Il tempo è, di solito, il **numero di sottoproblemi distinti** per il **numero di scelte** da valutare in ciascuno. Per la moltiplicazione di matrici: $\Theta(n^2)$ sottoproblemi per $O(n)$ scelte, $\Theta(n^3)$. Per lo zaino: $\Theta(nW)$ sottoproblemi per 2 scelte, $\Theta(nW)$. Per la LCS che vediamo ora: $\Theta(mn)$ sottoproblemi per una scelta costante, $\Theta(mn)$.

La più lunga sottosequenza comune (LCS)

Confrontare due sequenze è un problema frequentissimo: due versioni di un file (il comando diff), due filamenti di DNA, due parole simili. Una misura della somiglianza è la lunghezza della più lunga sottosequenza che hanno in comune.

DEFINIZIONE: SOTTOSEQUENZA E LCS

Una **sottosequenza** di $X = \langle x_1, \dots, x_m \rangle$ si ottiene cancellando zero o più elementi di X , senza cambiare l'ordine dei restanti (non devono essere consecutivi). Una **sottosequenza comune** di X e Y è sottosequenza di entrambe. Il problema della **LCS** (longest common subsequence) chiede una sottosequenza comune di lunghezza massima.

Per esempio, con $X = \text{PIANTA}$ e $Y = \text{PATATA}$, PTA è una sottosequenza comune, ma non la più lunga: PATA lo è, con lunghezza 4. Provare tutte le sottosequenze di X sarebbe $\Theta(2^m)$: serve un'idea migliore.

SOTTOSTRUTTURA OTTIMA

Indichiamo con X_i il **prefisso** $\langle x_1, \dots, x_i \rangle$ (X_0 è vuoto). Tutto si decide guardando gli **ultimi caratteri**.

DA SAPERE: TEOREMA DELLA SOTTOSTRUTTURA OTTIMA DELLA LCS

Sia $Z = \langle z_1, \dots, z_k \rangle$ una LCS di X_m e Y_n .

1. Se $x_m = y_n$, allora $z_k = x_m = y_n$ e Z_{k-1} è una LCS di X_{m-1} e Y_{n-1} .
2. Se x_m è diverso da y_n e z_k è diverso da x_m , allora Z è una LCS di X_{m-1} e Y_n .
3. Se x_m è diverso da y_n e z_k è diverso da y_n , allora Z è una LCS di X_m e Y_{n-1} .

Dimostrazione del punto 1. Se z_k fosse diverso da x_m , potremmo aggiungere $x_m = y_n$ in fondo a Z e ottenere una sottosequenza comune di lunghezza $k + 1$, contro l'ipotesi che Z sia la più lunga. Quindi $z_k = x_m$. Poi Z_{k-1} è sottosequenza comune di X_{m-1} e Y_{n-1} ; se esistesse una sottosequenza comune W più lunga, aggiungendo x_m a W otterremmo una sottosequenza comune di X e Y più lunga di Z (taglia e incolla). Contraddizione.

Dimostrazione del punto 2. Poiché z_k è diverso da x_m , Z non usa l'ultimo carattere di X : è sottosequenza comune di X_{m-1} e Y . Se ce ne fosse una più lunga, sarebbe anche sottosequenza comune di X e Y , più lunga di Z . Contraddizione. Il punto 3 è simmetrico.

LA RICORRENZA

Sia $c[i, j]$ la lunghezza di una LCS di X_i e Y_j . Dal teorema:

$$c[i, j] = 0 \quad \text{se } i = 0 \text{ oppure } j = 0$$

$$c[i, j] = c[i - 1, j - 1] + 1 \quad \text{se } i, j > 0 \text{ e } x_i = y_j$$

$$c[i, j] = \max(c[i - 1, j], c[i, j - 1]) \quad \text{se } i, j > 0 \text{ e } x_i \text{ diverso da } y_j$$

In parole: se gli ultimi caratteri coincidono, fanno parte della LCS e si prosegue sui prefissi più corti di uno; altrimenti almeno uno dei due ultimi caratteri non serve, e si prova a toglierlo da X o da Y , tenendo il meglio.

IL CALCOLO CON LA TABELLA

Ci sono solo $(m + 1)(n + 1)$ sottoproblemi distinti, le coppie (i, j) . Si riempie la tabella c riga per riga, da sinistra a destra: ogni casella dipende solo da quella in alto, da quella a sinistra e da quella in diagonale in alto a sinistra, già calcolate. In una seconda tabella b si memorizza da quale casella viene il valore: ↖ (caratteri uguali), ↑ (dal valore sopra), ← (dal valore a sinistra).

```

1  LCSLength(X, Y, m, n)
2      for i = 0 to m do c[i][0] = 0
3      for j = 0 to n do c[0][j] = 0
4      for i = 1 to m do
5          for j = 1 to n do
6              if X[i] = Y[j] then
7                  c[i][j] = c[i-1][j-1] + 1 ; b[i][j] = "diagonale"
8              else if c[i-1][j] >= c[i][j-1] then
9                  c[i][j] = c[i-1][j] ; b[i][j] = "su"
10             else
11                 c[i][j] = c[i][j-1] ; b[i][j] = "sinistra"
12     return c e b

```

ESEMPIO: X = PIANTA, Y = PATATA

Le righe corrispondono ai caratteri di X , le colonne a quelli di Y . In ogni casella il valore $c[i, j]$ e la freccia di $b[i, j]$.

		P	A	T	A	T	A
	0	0	0	0	0	0	0
P	0	1 ↖	1 ←	1 ←	1 ←	1 ←	1 ←
I	0	1 ↑	1 ↑	1 ↑	1 ↑	1 ↑	1 ↑
A	0	1 ↑	2 ↖	2 ←	2 ↖	2 ←	2 ↖
N	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑
T	0	1 ↑	2 ↑	3 ↖	3 ←	3 ↖	3 ←
A	0	1 ↑	2 ↖	3 ↑	4 ↖	4 ←	4 ↖

Vediamo come si calcolano due caselle. Per (A, T) nella riga 3, colonna 3: A e T sono diversi, quindi $c = \max(c \text{ sopra} = 1, c \text{ a sinistra} = 2) = 2$, con freccia $\leftarrow$. Per (T, T) nella riga 5, colonna 5: $T = T$, quindi $c = c \text{ in diagonale (riga 4, colonna 4, valore 2)} + 1 = 3$, con freccia $\nwarrow$.

RICOSTRUIRE LA LCS

Si parte dall'angolo in basso a destra e si seguono le frecce: ogni $\nwarrow$ indica un carattere della LCS. Le caselle in grassetto sono il percorso.

1. (A, A) $\nwarrow$: si prende A, si va in diagonale.
2. (T, T) $\nwarrow$: si prende T.
3. (N, A) $\uparrow$: si sale.
4. (A, A) $\nwarrow$: si prende A.
5. (I, T) $\uparrow$, poi (P, T) $\leftarrow$, (P, A) $\leftarrow$: ci si sposta senza prendere nulla.
6. (P, P) $\nwarrow$: si prende P.

I caratteri sono stati raccolti dalla fine all'inizio: la LCS è **PATA**, di lunghezza $c[6, 6] = 4$.

```

1 PrintLCS(b, X, i, j)
2   if i = 0 or j = 0 then return
3   if b[i][j] = "diagonale" then
4     PrintLCS(b, X, i-1, j-1)
5     stampa X[i] // stampata dopo la ricorsione: ordine giusto
6   else if b[i][j] = "su" then PrintLCS(b, X, i-1, j)
7   else PrintLCS(b, X, i, j-1)
```

DA SAPERE: COSTI DELLA LCS

LCSLength costa $\Theta(mn)$ in tempo e in spazio; PrintLCS costa $O(m + n)$, perché a ogni passo diminuisce i oppure j . La tabella b non è indispensabile: in ogni casella si può capire da dove si viene guardando i tre vicini di c . Se serve solo la **lunghezza** e non la sequenza, bastano due righe della tabella alla volta, cioè spazio $O(\min(m, n))$.

Distanza di editing

Un problema strettamente collegato: quante operazioni servono, al minimo, per trasformare una stringa in un'altra? È ciò che fa un correttore ortografico quando propone la parola "più vicina" a quella sbagliata.

DEFINIZIONE: DISTANZA DI EDITING

Date due stringhe $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$, la **distanza di editing** (o di Levenshtein) è il minimo numero di operazioni per trasformare X in Y , dove le operazioni ammesse sono: **inserire** un carattere, **cancellare** un carattere, **sostituire** un carattere con un altro, ciascuna di costo 1.

RICORRENZA

Sia $d[i, j]$ la distanza tra i prefissi X_i e Y_j . Ancora una volta si guarda l'ultimo carattere. Per trasformare X_i in Y_j , l'ultima operazione di una sequenza ottima è una di queste:

- x_i viene **cancellato**: prima si trasforma X_{i-1} in Y_j , costo $d[i-1, j] + 1$;
- y_j viene **inserito** in fondo: prima si trasforma X_i in Y_{j-1} , costo $d[i, j-1] + 1$;
- x_i viene **sostituito** con y_j , o lasciato com'è se sono già uguali: prima si trasforma X_{i-1} in Y_{j-1} , costo $d[i-1, j-1] + (0 \text{ se } x_i = y_j, 1 \text{ altrimenti})$.

$$d[i, 0] = i \quad d[0, j] = j$$

$$d[i, j] = \min(d[i-1, j] + 1, \quad d[i, j-1] + 1, \quad d[i-1, j-1] + \delta(x_i, y_j))$$

dove δ vale 0 se i caratteri sono uguali e 1 altrimenti. I casi base dicono che per passare da un prefisso di lunghezza i alla stringa vuota servono i cancellazioni, e dalla stringa vuota a un prefisso di lunghezza j servono j inserimenti.

ESEMPIO: DA GATTO A GRATO

		G	R	A	T	O
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	1	1	2	3
T	3	2	2	2	1	2
T	4	3	3	3	2	2
O	5	4	4	4	3	2

La distanza è $d[5, 5] = 2$. Ricostruendo all'indietro: $O = O$ (nessun costo), poi si sale cancellando una T , poi $T = T$, $A = A$, poi a sinistra si inserisce la R , infine $G = G$. Quindi $GATTO \rightarrow GRATTO$ (inserisco R) $\rightarrow$ $GRATO$ (cancello una T).

DA SAPERE: DISTANZA DI EDITING

Tempo e spazio $\Theta(mn)$, come la LCS: $(m+1)(n+1)$ sottoproblemi, tre scelte ciascuno. Se si ammettono solo inserimenti e cancellazioni (niente sostituzioni), la distanza vale $m + n - 2 \cdot \text{LCS}(X, Y)$: i caratteri della LCS restano, tutti gli altri di X si cancellano e quelli di Y si inseriscono.

Implementazione in C++

Le stringhe sono indicizzate da 0, quindi il carattere i -esimo del prefisso X_i è $x[i - 1]$.

```
#include <iostream>
#include <cstring>
using namespace std;

int c[50][50]; // tabella della LCS
int d[50][50]; // tabella della distanza di editing

// Riempie c e restituisce la lunghezza della LCS
int lcs(char x[], char y[]) {
    int m = strlen(x), n = strlen(y);
    for (int i = 0; i <= m; i++) c[i][0] = 0;
    for (int j = 0; j <= n; j++) c[0][j] = 0;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++) {
            if (x[i - 1] == y[j - 1]) c[i][j] = c[i - 1][j - 1] + 1; // diagonale
            else if (c[i - 1][j] >= c[i][j - 1]) c[i][j] = c[i - 1][j]; // su
            else c[i][j] = c[i][j - 1]; // sinistra
        }
    return c[m][n];
}

// Stampa la LCS risalendo la tabella c (senza la tabella delle frecce)
void stampaLCS(char x[], char y[], int i, int j) {
    if (i == 0 || j == 0) return;
    if (x[i - 1] == y[j - 1]) {
        stampaLCS(x, y, i - 1, j - 1);
        cout << x[i - 1];
    } else if (c[i - 1][j] >= c[i][j - 1]) stampaLCS(x, y, i - 1, j);
    else stampaLCS(x, y, i, j - 1);
}

// Distanza di editing tra x e y
int editing(char x[], char y[]) {
    int m = strlen(x), n = strlen(y);
    for (int i = 0; i <= m; i++) d[i][0] = i; // i cancellazioni
    for (int j = 0; j <= n; j++) d[0][j] = j; // j inserimenti
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++) {
            int diverso = (x[i - 1] == y[j - 1]) ? 0 : 1;
            int canc = d[i - 1][j] + 1; // cancello x[i]
            int ins = d[i][j - 1] + 1; // inserisco y[j]
            int sost = d[i - 1][j - 1] + diverso; // sostituisco o tengo
            d[i][j] = min(canc, min(ins, sost));
        }
    return d[m][n];
}

int main() {
    char x[] = "PIANTA", y[] = "PATATA";
    cout << "LCS di lunghezza " << lcs(x, y) << ": ";
    stampaLCS(x, y, strlen(x), strlen(y));
    cout << endl;
    char a[] = "GATTO", b[] = "GRATO";
    cout << "distanza di editing: " << editing(a, b) << endl;
    return 0;
}
```

Strategia greedy: le dimostrazioni

Come si dimostra che una scelta golosa porta all'ottimo: l'argomento dello scambio, applicato alla selezione di attività, ai codici di Huffman, allo zaino frazionario e al problema del resto, con i controesempi che mostrano dove il greedy fallisce.

Le due proprietà da dimostrare

Un algoritmo greedy è corretto quando il problema ha:

1. la **proprietà della scelta greedy**: esiste una soluzione ottima che contiene la scelta golosa;
2. la **sottostruttura ottima**: fatta la scelta golosa, una soluzione ottima del sottoproblema rimasto, unita alla scelta, dà una soluzione ottima del problema.

Messe insieme, permettono un'induzione: la prima scelta non preclude l'ottimo, il resto è un problema dello stesso tipo, e su di esso l'algoritmo procede allo stesso modo.

DA RICORDARE: L'ARGOMENTO DELLO SCAMBIO

La proprietà della scelta greedy si dimostra quasi sempre così: si prende una soluzione ottima **qualsiasi**; se non contiene la scelta golosa, la si **modifica** scambiando uno dei suoi elementi con quello scelto dal greedy; si mostra che la soluzione modificata è ancora ammissibile e **non peggiore**. Quindi esiste una soluzione ottima che contiene la scelta golosa.

Selezione di attività

Sia $S = \{a_1, \dots, a_n\}$ un insieme di attività ordinate per tempo di fine crescente, $f_1 \leq f_2 \leq \dots \leq f_n$. Due attività sono compatibili se i loro intervalli $[s_i, f_i]$ non si sovrappongono.

DA SAPERE: TEOREMA DELLA SCELTA GREEDY

Sia S_k un insieme non vuoto di attività e sia a_m quella di S_k che **finisce per prima**. Allora a_m appartiene ad almeno un sottoinsieme di S_k di attività mutuamente compatibili di cardinalità massima.

Dimostrazione.

1. Sia A_k un sottoinsieme ottimo (di cardinalità massima) di attività compatibili di S_k , e sia a_j l'attività di A_k che finisce per prima.
2. Se $a_j = a_m$ abbiamo finito: a_m sta in una soluzione ottima.
3. Altrimenti costruiamo A'_k togliendo a_j da A_k e aggiungendo a_m .
4. Le attività di A'_k sono compatibili: tutte quelle di A_k diverse da a_j iniziano dopo la fine di a_j , e poiché a_m è la prima a finire in tutto S_k , vale $f_m \leq f_j$; quindi iniziano anche dopo la fine di a_m .
5. A'_k ha lo stesso numero di attività di A_k , quindi è anch'essa ottima, e contiene a_m .

Sottostruttura ottima. Scelta a_m , il sottoproblema rimasto è l'insieme delle attività che iniziano dopo f_m . Se una soluzione ottima A contiene a_m , allora A senza a_m è ottima per il sottoproblema: se ce ne fosse una più grande, aggiungendovi a_m si otterrebbe una soluzione del problema originale più grande di A (taglia e incolla).

Unendo le due proprietà, per induzione sul numero di attività, l'algoritmo che sceglie ripetutamente l'attività compatibile che finisce per prima produce una soluzione ottima.

DA RICORDARE: PERCHÉ ALTRE SCELTE GOLOSE FALLISCONO

- **La più corta:** con [1, 5), [4, 7), [6, 10) la più corta è [4, 7), che si sovrappone alle altre due; scegliendola si ottiene 1 attività invece di 2.
- **Quella che inizia prima:** con [0, 10), [1, 2), [3, 4) si sceglie [0, 10) e si ottiene 1 attività invece di 2.
- **Quella con meno conflitti:** esistono controesempi più elaborati anche per questa.

Il fatto che una regola sembri sensata non basta: serve la dimostrazione.

Codici di Huffman

Sia C un alfabeto in cui ogni carattere c ha frequenza $f(c)$. Un codice prefisso è rappresentato da un albero binario pieno T , e il suo costo è

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$$

dove $d_T(c)$ è la profondità della foglia c , cioè la lunghezza della sua parola di codice.

LEMMA 1: LA SCELTA GREEDY

DA SAPERE: LEMMA DELLA SCELTA GREEDY

Siano x e y i due caratteri di C con frequenza **minima**. Esiste un codice prefisso ottimo in cui le parole di x e y hanno la stessa lunghezza e differiscono solo nell'ultimo bit, cioè x e y sono **foglie sorelle alla massima profondità**.

Dimostrazione con lo scambio.

1. Sia T un albero ottimo qualsiasi. Poiché è pieno, alla profondità massima ci sono almeno due foglie sorelle: chiamiamole a e b , e supponiamo $f(a) \leq f(b)$. Supponiamo anche $f(x) \leq f(y)$.
2. Poiché x e y hanno le frequenze minime, $f(x) \leq f(a)$ e $f(y) \leq f(b)$. Poiché a e b sono alla massima profondità, $d_T(x) \leq d_T(a)$ e $d_T(y) \leq d_T(b)$.
3. Costruiamo T' scambiando di posto le foglie x e a . Nel costo cambiano solo i due termini di x e a :

$$\begin{aligned} B(T) - B(T') &= f(x)d_T(x) + f(a)d_T(a) - f(x)d_T(a) - f(a)d_T(x) \\ &= (f(a) - f(x)) \cdot (d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

perché entrambi i fattori sono ≥ 0 per il punto 2. Quindi $B(T') \leq B(T)$.

4. Allo stesso modo si costruisce T'' scambiando y e b in T' , e si ottiene $B(T'') \leq B(T') \leq B(T)$.
5. Poiché T è ottimo, anche T'' lo è; e in T'' le foglie x e y sono sorelle alla profondità massima.

L'intuizione è semplice: i caratteri più rari devono stare più in basso, perché le parole lunghe "costano" poco se usate di rado. Scambiare un carattere raro in alto con uno più frequente in basso non può mai peggiorare il costo.

LEMMA 2: LA SOTTOSTRUTTURA OTTIMA

DA SAPERE: LEMMA DELLA SOTTOSTRUTTURA OTTIMA

Siano x e y i caratteri di frequenza minima. Sia C' l'alfabeto ottenuto togliendo x e y e aggiungendo un nuovo carattere z con $f(z) = f(x) + f(y)$. Se T' è un albero ottimo per C' , allora l'albero T ottenuto da T' sostituendo la foglia z con un nodo interno che ha x e y come figli è un albero ottimo per C .

Dimostrazione.

1. In T le foglie x e y hanno profondità $d_{T'}(z) + 1$; tutti gli altri caratteri hanno la stessa profondità che in T' . Quindi:

$$f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_{T'}(z) + 1) = f(z)d_{T'}(z) + f(x) + f(y)$$

e di conseguenza $B(T) = B(T') + f(x) + f(y)$.

2. Supponiamo per assurdo che T non sia ottimo: esiste T^* con $B(T^*) < B(T)$. Per il Lemma 1 possiamo supporre che in T^* x e y siano sorelle. Sostituendo in T^* le foglie x e y e il loro padre con una foglia z di frequenza $f(x) + f(y)$, otteniamo un albero T''' per C' con

$$B(T''') = B(T^*) - f(x) - f(y) < B(T) - f(x) - f(y) = B(T')$$

contro l'ottimalità di T' . Quindi T è ottimo.

DA SAPERE: TEOREMA

L'algoritmo di Huffman produce un codice prefisso ottimo. Il Lemma 1 dice che fondere per primi i due caratteri meno frequenti è sicuro; il Lemma 2 dice che, dopo la fusione, basta risolvere in modo ottimo il problema con un carattere in meno, ed è esattamente ciò che l'algoritmo fa nelle iterazioni successive.

Zaino frazionario

Ordiniamo gli oggetti per valore unitario decrescente: $v_1/w_1 \geq v_2/w_2 \geq \dots$. Il greedy prende quanto più possibile dell'oggetto 1, poi del 2, e così via.

Scelta greedy, con lo scambio. Sia O una soluzione ottima che prende dell'oggetto 1 una quantità q_1 minore di quella del greedy, cioè $\min(w_1, W)$. Allora O usa almeno una quantità $\delta > 0$ di capacità per oggetti con rapporto più basso (altrimenti lo zaino non sarebbe pieno e si potrebbe aggiungere altro oggetto 1, migliorando). Togliamo da O una quantità δ di quegli oggetti e mettiamo δ in più di oggetto 1: il peso non cambia, e il valore cambia di

$$\delta \cdot (v_1/w_1 - v_j/w_j) \geq 0$$

Ripetendo, si arriva a una soluzione ottima che prende tanto oggetto 1 quanto il greedy. La sottostruttura ottima è immediata: il resto dello zaino deve essere riempito in modo ottimo con gli oggetti rimanenti.

Perché non vale per lo 0/1. Lo scambio richiede di spostare una quantità δ **arbitraria** di peso, cosa impossibile se gli oggetti non si possono dividere. Controesempio: capacità 40, oggetti (peso, valore) = (10, 50), (15, 60), (25, 90), con rapporti 5, 4 e 3,6. Il greedy prende il primo e il secondo (peso 25, valore 110) e nei 15 chili rimasti il terzo non entra; l'ottimo è secondo più terzo, peso 40 e valore 150.

Il problema del resto

IL GREEDY È OTTIMO PER TAGLI COME QUELLI DELL'EURO

Dimostriamo per i tagli {1, 2, 5}; per 10, 20, 50 e per 100, 200, 500 il ragionamento si ripete identico moltiplicando per 10 e per 100. Consideriamo una soluzione **ottima** (minimo numero di monete) e vediamo che cosa non può contenere:

1. Due monete da 1: si potrebbero sostituire con una da 2, risparmiando una moneta. Quindi al più **una** moneta da 1.
2. Tre monete da 2: valgono 6 e si potrebbero sostituire con 5 + 1, risparmiando una moneta. Quindi al più **due** monete da 2.
3. Due monete da 2 e una da 1: valgono 5 e si sostituiscono con una sola moneta da 5.

Ne segue che, in una soluzione ottima, le monete da 1 e da 2 insieme valgono **al più 4** (i casi possibili sono 0, 1, 2, $3 = 2 + 1$, $4 = 2 + 2$). Quindi la parte dell'importo pagata con monete da 5 deve essere la più grande possibile: se $R \geq 5$ e si usassero meno monete da 5 del massimo, la parte restante sarebbe almeno 5, impossibile. È esattamente quello che fa il greedy: prende $\lfloor R/5 \rfloor$ monete da 5. Lo stesso ragionamento, applicato al resto $R \bmod 5$ con le monete da 2 e da 1, conclude.

IL GREEDY NON È OTTIMO IN GENERALE

Con i tagli $\{1, 4, 6\}$ e $R = 8$ il greedy dà $6 + 1 + 1$, tre monete, mentre $4 + 4$ ne usa due. Lo scambio fallisce perché non esiste un modo di "convertire" sempre più monete piccole in una grande senza perdere: $4 + 4 = 8$ non si può riscrivere con una moneta da 6 e lo stesso numero di monete o meno.

DA SAPERE: SCHEMA DELLE DIMOSTRAZIONI GREEDY

Per dimostrare che un greedy è corretto: (1) enunciare la scelta golosa; (2) dimostrare con lo **scambio** che una soluzione ottima qualsiasi si può trasformare, senza peggiorarla, in una che contiene quella scelta; (3) dimostrare la **sottostruttura ottima** con il taglia e incolla; (4) concludere per induzione. Per dimostrare che un greedy **non** è corretto basta un **controesempio** piccolo.

Grafi: correttezza di ordinamento topologico e cammini minimi

Le dimostrazioni che stanno dietro agli algoritmi sui grafi: la visita in profondità e l'ordinamento topologico, le proprietà del rilassamento, la correttezza di Bellman Ford, dell'algoritmo per i DAG e di Dijkstra, con esempi di esecuzione.

Visita in profondità

La DFS esplora il grafo andando sempre più in profondità finché può, e torna indietro solo quando un nodo non ha più vicini da scoprire. Usa tre colori per i nodi: **bianco** (non ancora scoperto), **grigio** (scoperto, visita in corso), **nero** (visita terminata). Registra per ogni nodo due istanti, dati da un contatore globale: $u.d$, quando u viene scoperto, e $u.f$, quando la sua visita finisce.

```

1 DFS(G)
2   for each u in G.V do u.colore = BIANCO ; u.pi = NIL
3   tempo = 0
4   for each u in G.V do
5       if u.colore = BIANCO then DFSVisit(G, u)
6
7 DFSVisit(G, u)
8   tempo = tempo + 1 ; u.d = tempo ; u.colore = GRIGIO
9   for each v in Adj[u] do
10      if v.colore = BIANCO then
11          v.pi = u
12          DFSVisit(G, v)
13   u.colore = NERO
14   tempo = tempo + 1 ; u.f = tempo

```

La DFS costa $\Theta(V + E)$: ogni nodo viene visitato una volta e ogni lista di adiacenza scorsa una volta.

GLI INTERVALLI E LA CLASSIFICAZIONE DEGLI ARCHI

Per ogni nodo l'intervallo $[u.d, u.f]$ descrive il periodo in cui u è grigio. Vale il **teorema delle parentesi**: per due nodi u e v , gli intervalli o sono disgiunti, oppure uno è contenuto nell'altro, e in questo caso il nodo con l'intervallo interno è un discendente dell'altro nella foresta della DFS. Come parentesi ben annidate: " $([])$ " è possibile, " $([])$ " no.

Gli archi si classificano guardando il colore di v quando si esamina l'arco (u, v) :

- **v bianco**: arco dell'albero (v viene scoperto ora);
- **v grigio**: arco **all'indietro** (back edge), v è un antenato di u ancora in corso;
- **v nero**: arco in avanti (verso un discendente già finito) o di attraversamento (verso un nodo in un altro ramo).

DA SAPERE: LEMMA

Un grafo orientato è **aciclico** se e solo se la DFS non trova **archi all'indietro**.

Dimostrazione. Se c'è un arco all'indietro (u, v) , v è un antenato di u : il cammino da v a u nell'albero più l'arco (u, v) forma un ciclo. Viceversa, se c'è un ciclo, sia v il primo nodo del ciclo scoperto dalla DFS e u il nodo che lo precede nel ciclo. Quando v viene scoperto, tutti gli altri nodi del ciclo sono bianchi e raggiungibili da v , quindi

diventeranno discendenti di v ; in particolare u , e l'arco (u, v) sarà esaminato mentre v è ancora grigio: è un arco all'indietro.

Ordinamento topologico

DEFINIZIONE: ORDINAMENTO TOPOLOGICO

Un **ordinamento topologico** di un grafo orientato aciclico $G = (V, E)$ è un ordinamento lineare di tutti i suoi nodi tale che, per ogni arco (u, v) di E , u compare prima di v . Disponendo i nodi su una retta in quell'ordine, tutti gli archi vanno da sinistra verso destra.

Esiste se e solo se il grafo è aciclico: in un ciclo ogni nodo dovrebbe precedere sé stesso. L'esempio classico è l'ordine in cui vestirsi: le calze prima delle scarpe, i pantaloni prima della cintura e delle scarpe, e così via. Un ordinamento topologico è una sequenza che rispetta tutte le precedenze.

```
1 TopologicalSort(G)
2   esegui DFS(G) per calcolare i tempi di fine v.f
3   quando la visita di un nodo finisce, inseriscilo in testa a una lista
4   return la lista
```

L'ordine prodotto è quello dei tempi di fine **decrescenti**. Costa $\Theta(V + E)$.

DA SAPERE: TEOREMA

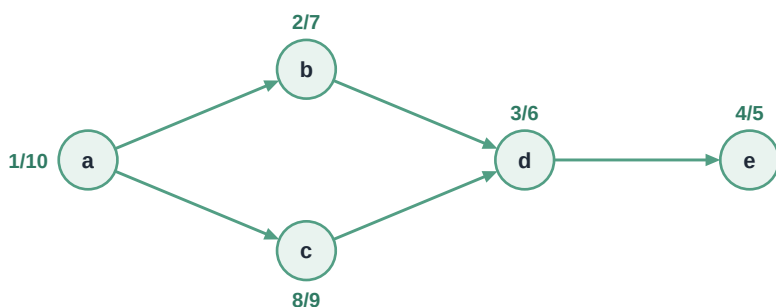
Su un grafo orientato aciclico, TopologicalSort produce un ordinamento topologico.

Dimostrazione. Basta mostrare che per ogni arco (u, v) vale $v.f < u.f$: allora v finisce prima, viene inserito in testa prima, e quindi nella lista finale sta **dopo** u . Quando la DFS esamina l'arco (u, v) , u è grigio. Vediamo il colore di v :

1. v non può essere **grigio**: sarebbe un arco all'indietro, impossibile in un grafo aciclico per il lemma.
2. Se v è **bianco**, diventa discendente di u e la sua visita finisce prima di quella di u : $v.f < u.f$.
3. Se v è **nero**, la sua visita è già finita mentre quella di u è in corso: $v.f < u.f$.

ESEMPIO

DAG con nodi a, b, c, d, e e archi $a \rightarrow b, a \rightarrow c, b \rightarrow d, c \rightarrow d, d \rightarrow e$. Visitando i nodi in ordine alfabetico, la DFS parte da a , scende in b , poi in d , poi in e ; e finisce, poi d , poi b ; da a scende in c , che trova d già nero e finisce; infine finisce a .



Il DAG dell'esempio: accanto a ogni nodo i tempi di scoperta e di fine, nella forma d/f .

NODO	A	B	C	D	E
scoperta d	1	2	8	3	4
fine f	10	7	9	6	5

Ordinando per f decrescente: a (10), c (9), b (7), d (6), e (5). Tutti gli archi vanno da sinistra a destra: $a \rightarrow b$, $a \rightarrow c$, $b \rightarrow d$, $c \rightarrow d$, $d \rightarrow e$. Ma anche a, b, c, d, e è un ordinamento topologico valido: b e c non sono collegati, quindi il loro ordine è libero.

DA RICORDARE: QUANDO L'ORDINAMENTO TOPOLOGICO È UNICO

Un DAG ha un **unico** ordinamento topologico se e solo se, in quell'ordinamento, ogni coppia di nodi **consecutivi** è collegata da un arco, cioè se esiste un cammino che tocca tutti i nodi. Infatti, se due nodi consecutivi u e v non fossero collegati da un arco, nessun vincolo imporrebbe u prima di v (un cammino da u a v dovrebbe passare per nodi intermedi, che invece non ci sono tra due consecutivi), e scambiandoli si otterrebbe un secondo ordinamento valido. Nell'esempio sopra b e c sono "liberi", quindi l'ordinamento non è unico; aggiungendo l'arco $b \rightarrow c$ diventerebbe unico: a, b, c, d, e .

Proprietà del rilassamento

Tutti gli algoritmi di cammino minimo partono con `InitializeSingleSource` e modificano le stime solo con `Relax`. Su questa base valgono alcune proprietà, che sono gli attrezzi per le dimostrazioni. Indichiamo con $\delta(s, v)$ il peso del cammino minimo.

DEFINIZIONE: LE PROPRIETÀ

1. **Disuguaglianza triangolare.** Per ogni arco (u, v) : $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
2. **Limite superiore.** Vale sempre $v.d \geq \delta(s, v)$, e quando $v.d$ raggiunge $\delta(s, v)$ non cambia più.
3. **Assenza di cammino.** Se v non è raggiungibile da s , allora $v.d = \delta(s, v) = \infty$ sempre.
4. **Convergenza.** Se $s \rightsquigarrow u \rightarrow v$ è un cammino minimo e $u.d = \delta(s, u)$ prima di rilassare (u, v) , dopo il rilassamento $v.d = \delta(s, v)$.
5. **Rilassamento del cammino.** Se $p = (s = v_0, v_1, \dots, v_k)$ è un cammino minimo e gli archi $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$ vengono rilassati **in quest'ordine** (anche con altri rilassamenti in mezzo), allora alla fine $v_k.d = \delta(s, v_k)$.

Le giustificazioni sono brevi. La (1) dice che il cammino minimo verso v non è peggiore di quello che passa per u e poi usa l'arco (u, v) . La (2) vale perché ogni valore assegnato a $v.d$ è il peso di un cammino reale da s a v , che non può essere minore del minimo. La (4) segue dalla (2): dopo `Relax`, $v.d \leq u.d + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v)$, e non può scendere sotto. La (5) segue applicando la (4) lungo il cammino, per induzione: dopo il rilassamento dell'arco i esimo, $v_i.d = \delta(s, v_i)$.

Correttezza di Bellman Ford

DA SAPERE: LEMMA

Se G non contiene cicli negativi raggiungibili da s , dopo le $|V| - 1$ iterazioni del ciclo principale vale $v.d = \delta(s, v)$ per ogni nodo v raggiungibile da s .

Dimostrazione. Sia $p = (v_0, \dots, v_k)$ un cammino minimo da s a v che non contiene cicli: allora ha al più $|V| - 1$ archi, cioè $k \leq |V| - 1$. Ogni iterazione rilassa **tutti** gli archi, quindi l'iterazione i rilassa anche (v_{i-1}, v_i) . Gli archi di p vengono così rilassati nell'ordine giusto, e per la proprietà del rilassamento del cammino $v.d = \delta(s, v)$.

DA SAPERE: TEOREMA

Se G non ha cicli negativi raggiungibili da s , Bellman Ford restituisce TRUE e calcola i cammini minimi. Se invece ne ha, restituisce FALSE.

Primo caso. Per il lemma, alla fine ogni $v.d = \delta(s, v)$. Nel controllo finale, per ogni arco (u, v) , la disuguaglianza triangolare dà $v.d = \delta(s, v) \leq \delta(s, u) + w(u, v) = u.d + w(u, v)$: nessun test fallisce e si restituisce TRUE.

Secondo caso. Sia $c = \langle v_0, v_1, \dots, v_k = v_0 \rangle$ un ciclo negativo raggiungibile: $\sum w(v_{i-1}, v_i) < 0$. Supponiamo per assurdo che si restituisca TRUE: allora per ogni arco del ciclo $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$. Sommando lungo tutto il ciclo:

$$\sum_{i=1..k} v_i.d \leq \sum_{i=1..k} v_{i-1}.d + \sum_{i=1..k} w(v_{i-1}, v_i)$$

Le prime due somme contengono gli stessi nodi (il ciclo si chiude, $v_0 = v_k$) e sono finite (i nodi sono raggiungibili), quindi si semplificano: resta $0 \leq \sum w(v_{i-1}, v_i)$, contro l'ipotesi che il ciclo sia negativo. Quindi almeno un test fallisce e si restituisce FALSE.

ESEMPIO: UN CICLO NEGATIVO

Archi $s \rightarrow a$ con peso 1, $a \rightarrow b$ con peso -2, $b \rightarrow a$ con peso 1: il ciclo a, b, a pesa -1. Ordine di rilassamento: (s, a) , (a, b) , (b, a) . Ci sono 3 nodi, quindi 2 iterazioni.

MOMENTO	S	A	B
inizio	0	∞	∞
dopo l'iterazione 1	0	0	-1
dopo l'iterazione 2	0	-1	-2

Nell'iterazione 1: $a.d = 1$, poi $b.d = 1 - 2 = -1$, poi (b, a) dà $-1 + 1 = 0 < 1$, quindi $a.d = 0$. Nell'iterazione 2 le stime scendono ancora. Nel controllo finale l'arco (a, b) dà $-1 - 2 = -3 < -2$: il test fallisce e si restituisce FALSE. Le stime continuerebbero a scendere all'infinito.

Correttezza per i DAG

Se i nodi vengono elaborati in ordine topologico, ogni cammino $p = \langle v_0, \dots, v_k \rangle$ ha i nodi in quell'ordine, quindi i suoi archi vengono rilassati nell'ordine (v_0, v_1) , (v_1, v_2) , ...: per la proprietà del rilassamento del cammino, alla fine $v_k.d = \delta(s, v_k)$. Basta un solo rilassamento per arco: $\Theta(V + E)$.

Correttezza di Dijkstra

DA SAPERE: TEOREMA

Se tutti i pesi sono **non negativi**, alla fine di Dijkstra vale $u.d = \delta(s, u)$ per ogni nodo u .

Invariante: all'inizio di ogni iterazione del ciclo while, $v.d = \delta(s, v)$ per ogni nodo v già in S . Basta mostrare che quando un nodo u viene estratto e aggiunto a S , vale già $u.d = \delta(s, u)$; poi, per la proprietà del limite superiore, non cambierà più.

Dimostrazione per assurdo.

1. Sia u il **primo** nodo per cui, al momento dell'inserimento in S , $u.d$ è diverso da $\delta(s, u)$. Non è s , per cui $s.d = 0 = \delta(s, s)$. Deve esistere un cammino da s a u , altrimenti $u.d = \delta(s, u) = \infty$ per la proprietà di assenza di cammino.
2. Consideriamo un cammino minimo p da s a u , nel momento in cui u viene estratto. Esso parte da s , che è in S , e arriva a u , che non lo è ancora. Sia y il **primo** nodo di p non in S , e x il nodo che lo precede su p (x è in S).
3. Quando x è stato aggiunto a S , per la scelta di u valeva $x.d = \delta(s, x)$, e l'arco (x, y) è stato rilassato. Per la proprietà di convergenza, ora $y.d = \delta(s, y)$.
4. y precede u sul cammino minimo e tutti i pesi sono non negativi, quindi $\delta(s, y) \leq \delta(s, u)$. Mettendo insieme:

$$y.d = \delta(s, y) \leq \delta(s, u) \leq u.d$$

dove l'ultima disuguaglianza è la proprietà del limite superiore.

5. Ma u è stato estratto **prima** di y , perché aveva la stima minima tra i nodi non in S : $u.d \leq y.d$. Allora tutte le disuguaglianze sono uguaglianze: $\delta(s, u) = u.d$. Contraddizione con la scelta di u .

DA RICORDARE: DOVE SERVONO I PESI NON NEGATIVI

Il passo 4 usa che, prolungando un cammino da y a u , il peso non può diminuire. Con un arco negativo questo è falso. Esempio: archi $s \rightarrow a$ con peso 2, $s \rightarrow b$ con peso 4, $b \rightarrow a$ con peso -3 , $a \rightarrow c$ con peso 1. Dijkstra estrae s , poi a ($d = 2$), che rilassando $a \rightarrow c$ pone $c.d = 3$; poi estrae c (3) e infine b (4). Rilassando $b \rightarrow a$ la stima di a scende a 1, ma a e c sono già stati elaborati: c resta con $d = 3$, mentre il cammino s, b, a, c pesa $4 - 3 + 1 = 2$.

Lavorare con l'albero dei predecessori

Al termine di qualunque algoritmo di cammino minimo, l'array π dei predecessori descrive l'albero dei cammini minimi. Molte domande si risolvono con una piccola funzione **ricorsiva** che risale i predecessori fino alla sorgente, l'unico nodo con $\pi = \text{NIL}$.

```

1 PrintPath(pi, s, v)                // stampa il cammino da s a v
2   if v = s then stampa s
3   else if pi[v] = NIL then stampa "nessun cammino"
4   else
5       PrintPath(pi, s, pi[v])      // prima il cammino fino al predecessore
6       stampa v                    // poi v
7
8 NumeroArchi(pi, s, v)              // quanti archi ha il cammino da s a v
9   if v = s then return 0
10  return 1 + NumeroArchi(pi, s, pi[v])
11
12 PesoCammino(pi, w, s, v)          // peso del cammino, sommando gli archi
13   if v = s then return 0
14   return PesoCammino(pi, w, s, pi[v]) + w(pi[v], v)

```

Ogni chiamata risale di un arco, quindi il costo è proporzionale alla lunghezza del cammino, al più $O(V)$. La ricorrenza è $T(k) = T(k - 1) + \Theta(1) = \Theta(k)$, con k numero di archi. Se v non è raggiungibile ($\pi[v] = \text{NIL}$ e v diverso da s) le ultime due funzioni andrebbero protette con lo stesso controllo di `PrintPath`. Il peso calcolato da `PesoCammino` coincide con $v.d$, e lo si può usare per verificare il risultato.