



NoiStudiamo

DISPENSA DI TEORIA



Algoritmi

Algoritmi e Laboratorio, modulo Algoritmi: ordinamento, hashing, alberi rosso neri, programmazione dinamica, greedy e cammini minimi, con implementazioni in C++.

A CURA DI

Damiano La Pusata

Ottobre 2026

Indice dei capitoli

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

1. Problemi, algoritmi e ordinamento	3
2. Heap, Heapsort e code di priorità	10
3. Quicksort	16
4. Ordinamento in tempo lineare	21
5. Mediane e statistiche d'ordine	27
6. Tabelle hash	31
7. Alberi rosso neri	38
8. Programmazione dinamica	50
9. Algoritmi greedy	58
10. Cammini minimi da sorgente unica	65
11. Cammini minimi tra tutte le coppie	73

Problemi, algoritmi e ordinamento

Che cosa significa risolvere un problema con un algoritmo, perché l'ordinamento è il banco di prova di tutto il corso e come nascono le due grandi strategie di progettazione: quella incrementale e quella ricorsiva.

Problemi computazionali e algoritmi

Un **problema computazionale** è una relazione tra dati di ingresso (input) e risultati attesi (output), descritta in modo preciso e senza ambiguità. Il problema non dice *come* ottenere il risultato, dice soltanto *quale* risultato è corretto per ciascun possibile ingresso. Un singolo ingresso concreto si chiama **istanza** del problema.

Un **algoritmo** è invece una procedura ben definita, composta da un numero finito di passi elementari, che prende un'istanza in ingresso e termina producendo l'uscita. Si dice che un algoritmo **risolve** un problema se, per ogni istanza, termina e restituisce l'uscita corretta. Basta una sola istanza sbagliata perché l'algoritmo non sia corretto.

Un'analogia utile è quella della ricetta. Il problema è "preparare una torta con questi ingredienti"; l'istanza è la specifica dispensa che hai in cucina oggi; l'algoritmo è la ricetta, cioè la sequenza di azioni che, eseguita da chiunque, porta alla torta. Due ricette diverse possono produrre la stessa torta, ma una può richiedere un'ora e l'altra tre: allo stesso modo due algoritmi possono risolvere lo stesso problema con costi molto diversi.

DEFINIZIONE: PROBLEMA, ISTANZA, ALGORITMO

Un **problema** specifica, per ogni ingresso ammissibile, quali uscite sono corrette. Un **istanza** è un ingresso particolare. Un **algoritmo** è una sequenza finita di passi elementari e non ambigui che, su ogni istanza, termina e produce un'uscita; è **corretto** se tale uscita è sempre quella richiesta dal problema.

DUE QUALITÀ DA MISURARE

Di un algoritmo ci interessano soprattutto due cose. La prima è la **correttezza**: deve funzionare su ogni istanza, non solo su quelle provate. La seconda è l'**efficienza**: quante risorse consuma, in particolare il tempo di esecuzione e la memoria, al crescere della dimensione dell'ingresso. La dimensione si indica di solito con n : per un array è il numero di elementi, per un grafo il numero di nodi e di archi.

Il tempo non si misura in secondi, perché dipenderebbe dal calcolatore. Si conta invece il numero di **operazioni elementari** (confronti, assegnamenti, somme, accessi a un elemento di array), ciascuna di costo costante. Il risultato è una funzione $T(n)$ che descrive come cresce il lavoro; il modo di confrontare queste funzioni (le notazioni Θ , O , Ω) è trattato nella dispensa di Laboratorio, qui lo usiamo in modo intuitivo.

PROBLEMI DI OTTIMIZZAZIONE

Alcuni problemi ammettono molte soluzioni ammissibili, ciascuna con un costo o un valore, e chiedono quella **migliore**: il percorso più corto tra due città, il carico più prezioso in uno zaino, il codice binario più compatto per un testo. Sono i **problemi di ottimizzazione**, e gran parte della seconda metà del corso (programmazione dinamica, greedy, cammini minimi) parla proprio di loro. Ordinare un array, al contrario, non è un problema di ottimizzazione: l'uscita corretta è una sola.

Il problema dell'ordinamento

L'ordinamento è il problema più studiato dell'informatica, ed è il terreno su cui il corso introduce quasi tutte le tecniche.

DEFINIZIONE: PROBLEMA DELL'ORDINAMENTO

Input: una sequenza di n numeri $\langle a_1, a_2, \dots, a_n \rangle$. **Output:** una permutazione $\langle a'_1, a'_2, \dots, a'_n \rangle$ della sequenza tale che $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

I valori da ordinare si chiamano **chiavi**. Nella pratica una chiave è spesso accompagnata da altri dati (un record con nome, matricola, voto) e si ordinano i record in base alla chiave; per semplicità negli esempi useremo solo numeri interi.

Due proprietà degli algoritmi di ordinamento torneranno spesso:

- un algoritmo ordina **in loco** (in place) se usa solo una quantità costante di memoria oltre all'array da ordinare;
- un algoritmo è **stabile** se elementi con chiave uguale compaiono nell'uscita nello stesso ordine relativo che avevano nell'ingresso. La stabilità conta quando si ordina per più criteri in successione, come vedremo con il Radix Sort.

DA RICORDARE: CONVENZIONE SUGLI INDICI

Lo pseudocodice del corso, come il libro di Cormen, numera gli array **da 1 a n**: $A[1]$ è il primo elemento. In C++ gli array partono invece da 0, quindi $A[0]$ è il primo e $A[n - 1]$ l'ultimo. Nelle implementazioni di questa dispensa la traduzione è sempre segnalata nei commenti: è la fonte più comune di errori di un'unità.

Strategia incrementale: Insertion Sort

La strategia **incrementale** costruisce la soluzione un pezzo alla volta: si mantiene una parte del problema già risolta e la si allarga di un elemento a ogni passo.

Insertion Sort è l'esempio tipico, ed è esattamente il modo in cui molte persone ordinano le carte da gioco in mano. Si tengono a sinistra le carte già in ordine; si prende la carta successiva e la si fa scorrere verso sinistra finché non trova il suo posto, spostando di una posizione le carte più grandi.

In termini di array: all'inizio del passo j il sottoarray $A[1..j - 1]$ è già ordinato. Si salva $A[j]$ in una variabile *chiave*, si scorrono verso sinistra gli elementi maggiori di *chiave* spostandoli di un posto a destra, e infine si scrive *chiave* nel buco rimasto libero.

```
1 InsertionSort(A, n)
2   for j = 2 to n do
3       chiave = A[j]
4       // inserisce A[j] nella parte già ordinata A[1..j-1]
5       i = j - 1
6       while i > 0 and A[i] > chiave do
7           A[i+1] = A[i]      // sposta a destra l'elemento più grande
8           i = i - 1
9       A[i+1] = chiave
```

ESEMPIO PASSO PER PASSO

Ordiniamo $A = \langle 6, 3, 8, 1, 5 \rangle$. In ogni riga la parte evidenziata è quella già ordinata al termine del passo; la cella scura è la chiave appena inserita.

inizio	6	3	8	1	5
j = 2	3	6	8	1	5
j = 3	3	6	8	1	5
j = 4	1	3	6	8	5
j = 5	1	3	5	6	8
	1	2	3	4	5

Insertion Sort su {6, 3, 8, 1, 5}: la zona ordinata cresce di un elemento per passo.

Seguiamo in dettaglio il passo $j = 4$, il più istruttivo.

1. La chiave è $A[4] = 1$; la parte ordinata è $\{3, 6, 8\}$.
2. Si confronta 1 con 8: 8 è maggiore, quindi 8 scivola in posizione 4.
3. Si confronta 1 con 6: anche 6 scivola, in posizione 3.
4. Si confronta 1 con 3: anche 3 scivola, in posizione 2.
5. Ora i vale 0, il ciclo si ferma e la chiave 1 viene scritta in $A[1]$.

QUANTO COSTA

Il costo dipende da quanto deve scorrere ogni chiave. Nel **caso migliore** l'array è già ordinato: il ciclo interno non sposta mai nulla e si fa un solo confronto per elemento, quindi il tempo è proporzionale a n , cioè $\Theta(n)$. Nel **caso peggiore** l'array è ordinato al contrario: la chiave j deve superare tutti i $j - 1$ elementi precedenti, e il totale dei confronti è

$$1 + 2 + \dots + (n - 1) = n(n - 1) / 2$$

che cresce come n^2 , cioè $\Theta(n^2)$. Anche in media, su ingressi casuali, ogni chiave scorre circa per metà della parte ordinata, e il tempo resta $\Theta(n^2)$.

DA SAPERE: INSERTION SORT

Insertion Sort è **in loco** e **stabile** (usa il confronto stretto $A[i] > \text{chiave}$, quindi non scavalca le chiavi uguali). Costa $\Theta(n)$ nel caso migliore e $\Theta(n^2)$ nel caso peggiore e medio. È **adattivo**: più l'ingresso è già quasi ordinato, meno lavora. Per questo è ottimo su array piccoli o quasi ordinati.

IMPLEMENTAZIONE IN C++

Gli indici sono traslati di uno: il ciclo esterno parte da 1 (il secondo elemento) e la condizione del ciclo interno diventa $i \geq 0$.

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

// Ordina A[0..n-1] in modo crescente con Insertion Sort
void insertionSort(int A[], int n) {
    for (int j = 1; j < n; j++) {          // j parte dal secondo elemento
        int chiave = A[j];                // elemento da inserire
        int i = j - 1;
        // sposta a destra gli elementi maggiori della chiave
        while (i >= 0 && A[i] > chiave) {
            A[i + 1] = A[i];
            i--;
        }
        A[i + 1] = chiave;                 // inserisce la chiave nel buco libero
    }
}
```

Strategia ricorsiva: divide et impera

La seconda grande strategia è il **divide et impera**. L'idea è che un problema grande sia difficile, ma che spesso si possa spezzare in problemi più piccoli *dello stesso tipo*, risolverli separatamente (con lo stesso metodo, cioè ricorsivamente) e poi mettere insieme le soluzioni. Si articola in tre fasi:

1. **Divide**: si suddivide il problema in sottoproblemi più piccoli.
2. **Impera**: si risolvono i sottoproblemi ricorsivamente; quando sono abbastanza piccoli (il **caso base**) si risolvono direttamente.
3. **Combina**: si uniscono le soluzioni dei sottoproblemi nella soluzione del problema originale.

Pensa a un professore che deve correggere 400 compiti. Li divide in due pile da 200 e le affida a due assistenti; ciascun assistente fa lo stesso con due collaboratori, e così via finché qualcuno riceve un solo compito e lo corregge. Poi i risultati risalgono e vengono riuniti. Nessuno fa tutto, ma il lavoro complessivo è organizzato in modo molto efficiente.

Merge Sort

Merge Sort applica il divide et impera all'ordinamento:

- **divide**: spezza l'array in due metà di dimensione circa $n/2$;
- **impera**: ordina ricorsivamente ciascuna metà; un array di un solo elemento è già ordinato (caso base);
- **combina**: fonde (merge) le due metà ordinate in un unico array ordinato.

Tutta l'intelligenza sta nella fusione. Immagina due mazzi di carte già ordinati, a faccia in su, con la carta più piccola in cima. Per ottenere un unico mazzo ordinato basta guardare le due carte in cima, prendere la più piccola e metterla nel mazzo d'uscita, e ripetere. Ogni carta viene toccata una sola volta, quindi la fusione di due mazzi con n carte in totale costa $\Theta(n)$.

```

1 MergeSort(A, p, r)
2   if p < r then
3     q = floor((p + r) / 2)    // punto di divisione
4     MergeSort(A, p, q)       // ordina la meta' sinistra
5     MergeSort(A, q + 1, r)   // ordina la meta' destra
6     Merge(A, p, q, r)       // fonde le due meta'

```

```

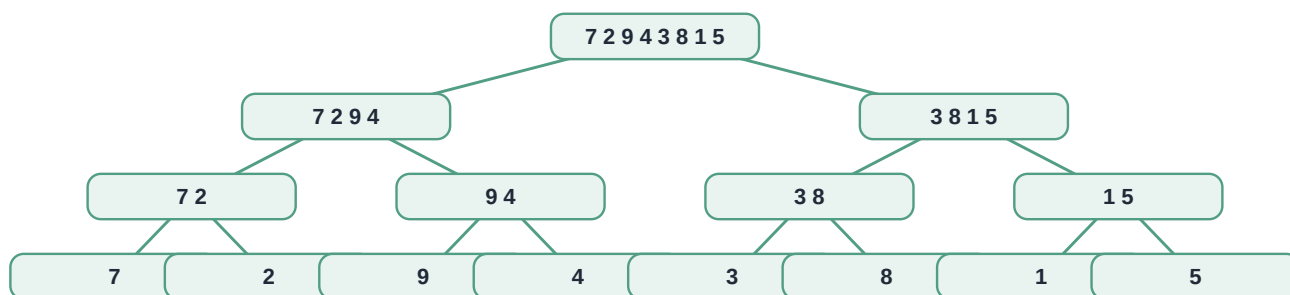
1 Merge(A, p, q, r)
2   n1 = q - p + 1 ; n2 = r - q
3   copia A[p..q] in L[1..n1] e A[q+1..r] in R[1..n2]
4   L[n1+1] = infinito ; R[n2+1] = infinito // sentinelle
5   i = 1 ; j = 1
6   for k = p to r do
7     if L[i] <= R[j] then
8       A[k] = L[i] ; i = i + 1
9     else
10      A[k] = R[j] ; j = j + 1

```

Le **sentinelle** (il valore infinito in fondo a L e R) evitano di controllare ogni volta se uno dei due mazzi è finito: quando un mazzo si esaurisce, in cima resta l'infinito, che perde sempre il confronto, e quindi si prende automaticamente dall'altro mazzo.

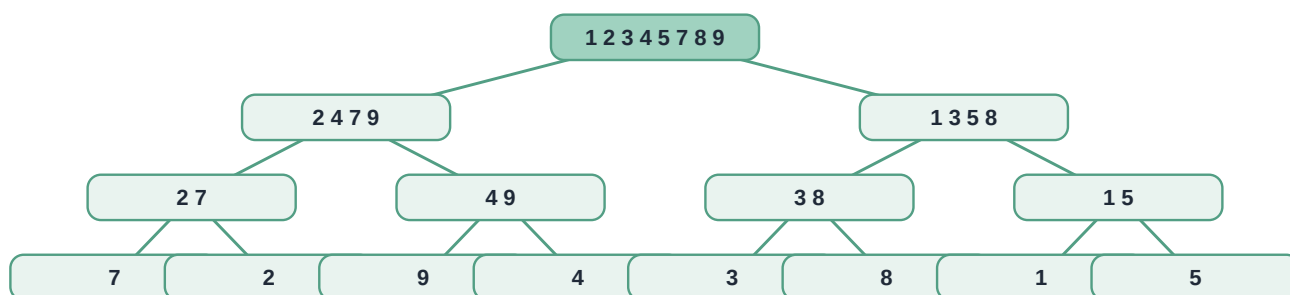
ESEMPIO: LA DISCESA E LA RISALITA

Ordiniamo {7, 2, 9, 4, 3, 8, 1, 5}. Nella fase di discesa l'array viene dimezzato finché restano singoli elementi.



Fase di divisione: ogni nodo è un sottoarray, le foglie sono già ordinate.

Nella fase di risalita ogni coppia di fratelli viene fusa: {7} e {2} diventano {2, 7}, {9} e {4} diventano {4, 9}, poi {2, 7} e {4, 9} diventano {2, 4, 7, 9}, e così via fino alla radice.



Fase di fusione: lo stesso albero letto dal basso verso l'alto; in ogni nodo c'è il risultato del Merge.

Vediamo l'ultima fusione passo per passo, con L = {2, 4, 7, 9} e R = {1, 3, 5, 8}:

PASSO	CONFRONTO	SI PRENDE	USCITA FINORA
1	2 contro 1	1 da R	1
2	2 contro 3	2 da L	1 2
3	4 contro 3	3 da R	1 2 3
4	4 contro 5	4 da L	1 2 3 4
5	7 contro 5	5 da R	1 2 3 4 5
6	7 contro 8	7 da L	1 2 3 4 5 7
7	9 contro 8	8 da R	1 2 3 4 5 7 8
8	9 contro ∞	9 da L	1 2 3 4 5 7 8 9

QUANTO COSTA

L'albero della figura ha circa $\log_2 n$ livelli, perché ogni livello dimezza la dimensione. A ogni livello, sommando tutti i Merge, si toccano esattamente n elementi. Quindi il costo totale è circa n per $\log_2 n$, cioè $\Theta(n \log n)$, in **ogni** caso: Merge Sort non è adattivo, lavora uguale su ingressi ordinati e disordinati. In forma di equazione di ricorrenza:

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

DA SAPERE: MERGE SORT

Merge Sort costa $\Theta(n \log n)$ nel caso migliore, medio e peggiore, ed è **stabile** (a parità si prende da L, con il confronto $\leq$). **Non** è in loco: la fusione richiede memoria ausiliaria $\Theta(n)$ per gli array L e R. La differenza tra n^2 e $n \log n$ è enorme: per $n = 1\,000\,000$, n^2 vale 10^{12} mentre $n \log_2 n$ vale circa $2 \cdot 10^7$.

IMPLEMENTAZIONE IN C++

In C++ non usiamo le sentinelle (non esiste un vero infinito per gli interi) ma controlliamo esplicitamente se una delle due metà è finita. Gli array ausiliari sono allocati con new e liberati con delete.

```
// Fonde A[p..q] e A[q+1..r], già ordinati, in A[p..r]
void merge(int A[], int p, int q, int r) {
    int n1 = q - p + 1;           // lunghezza meta' sinistra
    int n2 = r - q;               // lunghezza meta' destra
    int* L = new int[n1];
    int* R = new int[n2];
    for (int i = 0; i < n1; i++) L[i] = A[p + i];
    for (int j = 0; j < n2; j++) R[j] = A[q + 1 + j];

    int i = 0, j = 0;            // indici in L e in R
    for (int k = p; k <= r; k++) {
        // prendo da L se R e' finito oppure se L[i] <= R[j]
        if (j >= n2 || (i < n1 && L[i] <= R[j])) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
    delete[] L;
    delete[] R;
}

// Ordina A[p..r]; chiamata iniziale: mergeSort(A, 0, n - 1)
void mergeSort(int A[], int p, int r) {
    if (p < r) {
        int q = (p + r) / 2;      // punto di mezzo
        mergeSort(A, p, q);
        mergeSort(A, q + 1, r);
        merge(A, p, q, r);
    }
}

int main() {
    int A[] = {7, 2, 9, 4, 3, 8, 1, 5};
    int n = 8;
    mergeSort(A, 0, n - 1);
    for (int i = 0; i < n; i++) cout << A[i] << " ";
    cout << endl;
    int B[] = {6, 3, 8, 1, 5};
    insertionSort(B, 5);
    for (int i = 0; i < 5; i++) cout << B[i] << " ";
    cout << endl;
    return 0;
}
```

Incrementale o ricorsivo?

ALGORITMO	STRATEGIA	TEMPO	MEMORIA EXTRA
Insertion Sort	incrementale	da $\Theta(n)$ a $\Theta(n^2)$	$\Theta(1)$, in loco
Merge Sort	divide et impera	$\Theta(n \log n)$ sempre	$\Theta(n)$

DA RICORDARE

La strategia non decide da sola l'efficienza, ma il divide et impera, quando i sottoproblemi sono di dimensione proporzionale ($n/2$, $n/3$, ...) e la combinazione costa poco, porta spesso a tempi del tipo $n \log n$. Nei prossimi capitoli vedremo altri due ordinamenti $\Theta(n \log n)$: Heapsort, che a differenza di Merge Sort è in loco, e Quicksort, che è il più veloce in pratica.

Heap, Heapsort e code di priorità

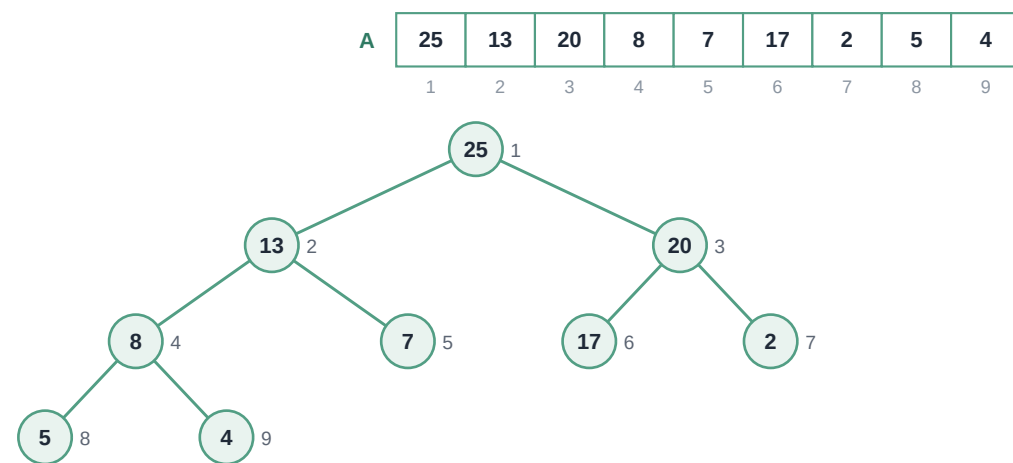
Una struttura dati semplice, un albero conservato dentro un array, che permette di trovare sempre il massimo in un istante e di ordinare in tempo $n \log n$ senza memoria aggiuntiva.

Che cos'è un heap

Un **heap binario** è un array che va letto come un **albero binario quasi completo**: tutti i livelli sono pieni tranne, eventualmente, l'ultimo, che è riempito da sinistra verso destra. Non ci sono puntatori: la forma dell'albero è implicita nella posizione degli elementi nell'array.

La regola di lettura è semplicissima. La radice è $A[1]$; i figli del nodo in posizione i stanno nelle posizioni $2i$ e $2i + 1$; il padre del nodo i sta in posizione $\lfloor i/2 \rfloor$.

FUNZIONE	INDICI DA 1 (PSEUDOCODICE)	INDICI DA 0 (C++)
Parent(i)	$\lfloor i/2 \rfloor$	$(i - 1) / 2$
Left(i)	$2i$	$2i + 1$
Right(i)	$2i + 1$	$2i + 2$



Lo stesso max heap visto come array e come albero; accanto a ogni nodo è indicato il suo indice nell'array.

LA PROPRIETÀ DI HEAP

La forma da sola non basta: serve una regola sull'ordine dei valori.

DEFINIZIONE: MAX HEAP E MIN HEAP

In un **max heap** ogni nodo diverso dalla radice soddisfa $A[\text{Parent}(i)] \geq A[i]$: ogni padre è maggiore o uguale ai suoi figli. Di conseguenza il **massimo** dell'intero heap si trova nella radice $A[1]$. In un **min heap** vale la disuguaglianza opposta e nella radice c'è il minimo.

Attenzione a non confondere l'heap con un albero binario di ricerca. In un heap tra fratelli non c'è alcun ordine (13 sta a sinistra di 20, ma potrebbe essere il contrario) e il fatto di essere ordinato riguarda solo le "linee verticali" dalla radice alle foglie. È un ordine parziale, più debole, e proprio per questo costa poco mantenerlo.

Due misure importanti: l'**altezza** di un nodo è il numero di archi del cammino più lungo da quel nodo a una foglia; l'altezza dell'heap è quella della radice. Poiché l'albero è quasi completo, un heap di n elementi ha altezza $\lfloor \log_2 n \rfloor$. Tutte le operazioni che vedremo percorrono al più un cammino dalla radice a una foglia, e per questo costano $O(\log n)$.

DA RICORDARE

Le foglie di un heap di n elementi sono i nodi di indice da $\lfloor n/2 \rfloor + 1$ a n . Lo si vede dal fatto che il nodo $\lfloor n/2 \rfloor + 1$ avrebbe il figlio sinistro in posizione maggiore di n , quindi non ha figli. Questo fatto serve a BuildMaxHeap.

MaxHeapify: riparare un heap

MaxHeapify(A, i) è il mattone di tutto. Si usa quando i due sottoalberi del nodo i sono già max heap, ma $A[i]$ potrebbe essere più piccolo di uno dei suoi figli, violando la proprietà. L'idea è far **affondare** il valore troppo piccolo: lo si scambia con il maggiore dei due figli e si ripete più in basso, finché il valore non trova un punto in cui è maggiore di entrambi i figli (o diventa una foglia).

Un'immagine utile è quella di una gerarchia aziendale in cui un nuovo capo meno esperto viene messo sopra due capi più bravi: il più bravo dei due prende il suo posto, e il nuovo arrivato scende di un livello, dove il confronto si ripete.

```
1 MaxHeapify(A, i)
2   l = Left(i) ; r = Right(i)
3   if l <= heapSize(A) and A[l] > A[i] then massimo = l
4   else massimo = i
5   if r <= heapSize(A) and A[r] > A[massimo] then massimo = r
6   if massimo != i then
7       scambia A[i] con A[massimo]
8       MaxHeapify(A, massimo) // il valore continua a scendere
```

DA SAPERE: COSTO DI MAXHEAPIFY

A ogni livello MaxHeapify fa un numero costante di confronti, e scende di un livello. Il costo è quindi proporzionale all'altezza del nodo: $O(h)$, cioè **$O(\log n)$** quando parte dalla radice. Nella dispensa di Laboratorio questo risultato viene dimostrato con l'equazione di ricorrenza $T(n) \leq T(2n/3) + \Theta(1)$.

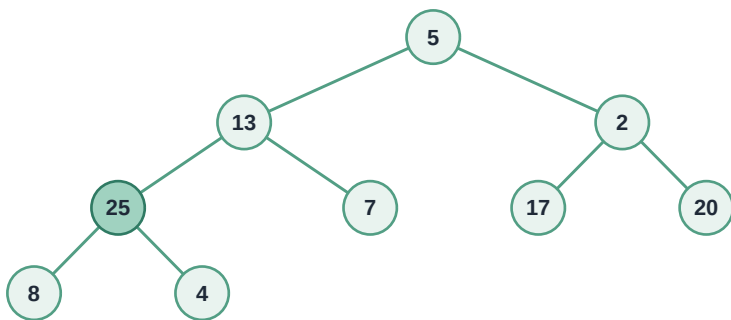
BuildMaxHeap: costruire un heap da un array qualsiasi

Per trasformare un array disordinato in un max heap si osserva che le foglie sono già, da sole, heap di un elemento. Basta allora chiamare MaxHeapify su tutti i nodi interni, **dal basso verso l'alto**, cioè da $\lfloor n/2 \rfloor$ fino a 1. Andare dal basso è essenziale: quando si ripara il nodo i , i suoi sottoalberi sono già stati sistemati, che è proprio la condizione richiesta da MaxHeapify.

```
1 BuildMaxHeap(A, n)
2   heapSize(A) = n
3   for i = floor(n/2) downto 1 do
4       MaxHeapify(A, i)
```

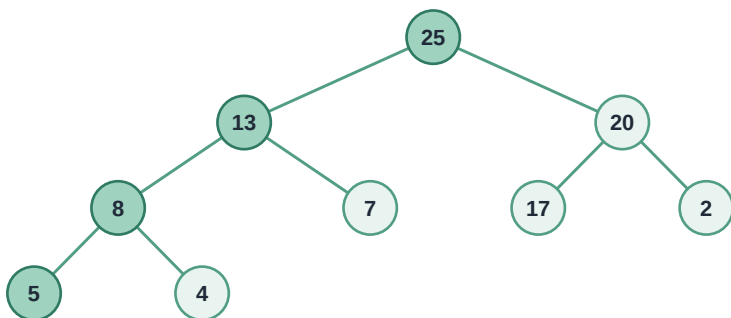
ESEMPIO PASSO PER PASSO

Costruiamo un max heap da $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$, con $n = 9$. I nodi interni sono quelli di indice da 4 a 1.



Situazione iniziale: si parte dal nodo di indice 4 (evidenziato).

1. $i = 4$, valore 25, figli 8 e 4: 25 è già il più grande, nulla da fare.
2. $i = 3$, valore 2, figli 17 e 20: il maggiore è 20, si scambia. Il 2 finisce in una foglia e si ferma.
3. $i = 2$, valore 13, figli 25 e 7: si scambia con 25. Il 13 scende in posizione 4, dove ha figli 8 e 4: è maggiore di entrambi e si ferma.
4. $i = 1$, valore 5, figli 25 e 20: si scambia con 25. Il 5 scende in posizione 2, dove ha figli 13 e 7: si scambia con 13. Scende in posizione 4, figli 8 e 4: si scambia con 8 e finisce in una foglia.



Risultato: evidenziato il percorso compiuto dal 5 durante l'ultimo MaxHeapify. L'array è ora (25, 13, 20, 8, 7, 17, 2, 5, 4).

DA SAPERE: COSTO DI BUILDMAXHEAP

Una stima frettolosa direbbe $n/2$ chiamate da $O(\log n)$, quindi $O(n \log n)$. È vero ma non stretto: la maggior parte dei nodi sta in basso, dove MaxHeapify costa pochissimo (metà dei nodi sono foglie, un quarto ha altezza 1, e così via). Facendo il conto preciso il costo totale è **$O(n)$** , lineare. La dimostrazione, che usa la serie $\sum h/2^h$, è nella dispensa di Laboratorio.

Heapsort

Ora che sappiamo costruire un heap, ordinare è facile. Il massimo è in $A[1]$: lo scambiamo con l'ultimo elemento $A[n]$, così il massimo finisce nella sua posizione definitiva in fondo all'array. Poi "dimentichiamo" l'ultima cella riducendo heapSize di uno, e ripariamo la radice con MaxHeapify. Ripetiamo finché l'heap ha un solo elemento.

```

1 HeapSort(A, n)
2   BuildMaxHeap(A, n)
3   for i = n downto 2 do
4       scambia A[1] con A[i]           // il massimo va in fondo
5       heapSize(A) = heapSize(A) - 1 // A[i] esce dall'heap
6       MaxHeapify(A, 1)                // ripara la radice

```

Seguiamo i primi due giri partendo dall'heap (25, 13, 20, 8, 7, 17, 2, 5, 4). Le celle scure sono già nella posizione finale.

heap	25	13	20	8	7	17	2	5	4
scambio	4	13	20	8	7	17	2	5	25
heapify	20	13	17	8	7	4	2	5	25
scambio	5	13	17	8	7	4	2	20	25
heapify	17	13	5	8	7	4	2	20	25
	1	2	3	4	5	6	7	8	9

Primi due giri di Heapsort: in verde le celle toccate da MaxHeapify.

Nel primo giro il 4, arrivato in radice, scende scambiandosi prima con 20 e poi con 17. Nel secondo giro il 5 scende scambiandosi con 17 e si ferma, perché i suoi nuovi figli 4 e 2 sono più piccoli. Continuando così, l'array finale è (2, 4, 5, 7, 8, 13, 17, 20, 25).

DA SAPERE: HEAPSORT

Heapsort costa $O(n)$ per BuildMaxHeap più $n - 1$ chiamate di MaxHeapify da $O(\log n)$: in totale **$O(n \log n)$** nel caso peggiore. È **in loco** (unisce i pregi di Merge Sort e di Insertion Sort) ma **non è stabile**: gli scambi a lunga distanza possono invertire chiavi uguali.

Code di priorità

Una **coda di priorità** è una struttura che conserva un insieme di elementi, ciascuno con una chiave che ne indica la priorità, e permette di estrarre sempre quello più importante. Il pronto soccorso è l'esempio perfetto: non si viene serviti in ordine di arrivo, ma in base al codice di gravità, e un nuovo paziente grave può passare davanti a tutti.

Una **coda di max priorità** offre queste operazioni:

- **Insert(S, x)**: inserisce l'elemento x;
- **Maximum(S)**: restituisce l'elemento con chiave massima, senza toglierlo;
- **ExtractMax(S)**: toglie e restituisce l'elemento con chiave massima;
- **IncreaseKey(S, x, k)**: aumenta la chiave di x al nuovo valore k, che non deve essere minore di quella attuale.

Il max heap è l'implementazione naturale. Esiste anche la versione con min heap (ExtractMin, DecreaseKey), che useremo nell'algoritmo di Dijkstra e nel codice di Huffman.

```

1  HeapMaximum(A)
2      return A[1]
3
4  HeapExtractMax(A)
5      if heapSize(A) < 1 then errore "heap vuoto"
6      massimo = A[1]
7      A[1] = A[heapSize(A)]      // l'ultimo elemento va in radice
8      heapSize(A) = heapSize(A) - 1
9      MaxHeapify(A, 1)          // e la radice viene riparata
10     return massimo

```

IncreaseKey fa il movimento opposto a MaxHeapify: una chiave aumentata può essere diventata più grande del padre, quindi deve **risalire**, scambiandosi con il padre finché questo non è maggiore o uguale. Insert si ottiene come caso particolare: si aggiunge in fondo una foglia con chiave meno infinito e poi si aumenta la sua chiave al valore voluto.

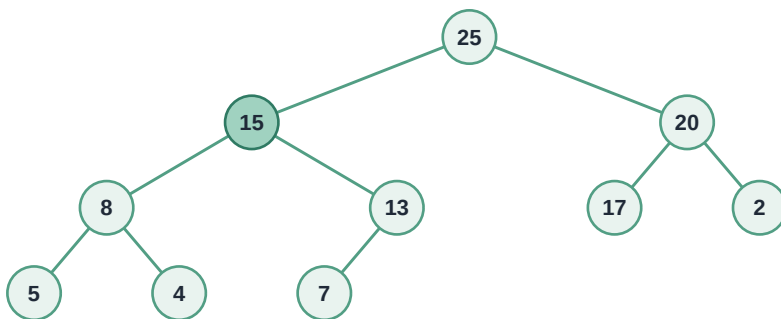
```

1  HeapIncreaseKey(A, i, chiave)
2      if chiave < A[i] then errore "la nuova chiave e' piu' piccola"
3      A[i] = chiave
4      while i > 1 and A[Parent(i)] < A[i] do
5          scambia A[i] con A[Parent(i)]    // la chiave risale
6          i = Parent(i)
7
8  MaxHeapInsert(A, chiave)
9      heapSize(A) = heapSize(A) + 1
10     A[heapSize(A)] = meno infinito
11     HeapIncreaseKey(A, heapSize(A), chiave)

```

ESEMPIO: INSERIRE 15

Inseriamo la chiave 15 nell'heap {25, 13, 20, 8, 7, 17, 2, 5, 4}. La nuova foglia occupa la posizione 10, figlia del nodo 5 (valore 7). Poiché $15 > 7$ si scambia e sale in posizione 5; il nuovo padre è il nodo 2 (valore 13), ancora più piccolo, e si scambia di nuovo; ora il padre è la radice 25, più grande, e la risalita si ferma.



Dopo l'inserimento di 15: evidenziata la posizione raggiunta. L'array diventa {25, 15, 20, 8, 13, 17, 2, 5, 4, 7}.

OPERAZIONE	COSTO
Maximum	$\Theta(1)$
ExtractMax	$O(\log n)$
IncreaseKey	$O(\log n)$
Insert	$O(\log n)$

Implementazione in C++

Con indici da 0 i figli di i sono $2i + 1$ e $2i + 2$. La dimensione dell'heap è passata come parametro; per la coda di priorità usiamo un array di capacità fissa.

```

#include <iostream>
using namespace std;

// Indici con array che parte da 0
int sinistro(int i) { return 2 * i + 1; }
int destro(int i)  { return 2 * i + 2; }
int padre(int i)   { return (i - 1) / 2; }

// Fa scendere A[i] finche' non rispetta la proprieta' di max heap
void maxHeapify(int A[], int dim, int i) {
    int l = sinistro(i);
    int r = destro(i);

```

```

    int massimo = i;
    if (l < dim && A[l] > A[massimo]) massimo = l;
    if (r < dim && A[r] > A[massimo]) massimo = r;
    if (massimo != i) {
        swap(A[i], A[massimo]);
        maxHeapify(A, dim, massimo);    // continua piu' in basso
    }
}

// Trasforma A[0..n-1] in un max heap
void buildMaxHeap(int A[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)    // dall'ultimo nodo interno alla radice
        maxHeapify(A, n, i);
}

void heapSort(int A[], int n) {
    buildMaxHeap(A, n);
    for (int i = n - 1; i >= 1; i--) {
        swap(A[0], A[i]);                // il massimo va in fondo
        maxHeapify(A, i, 0);            // l'heap ora ha i elementi
    }
}

// Coda di max priorit : restituisce e toglie il massimo
int extractMax(int A[], int& dim) {
    int massimo = A[0];
    A[0] = A[dim - 1];                // l'ultimo elemento va in radice
    dim--;
    maxHeapify(A, dim, 0);
    return massimo;
}

// Aumenta la chiave in posizione i e la fa risalire
void increaseKey(int A[], int i, int chiave) {
    A[i] = chiave;
    while (i > 0 && A[padre(i)] < A[i]) {
        swap(A[i], A[padre(i)]);
        i = padre(i);
    }
}

// Inserisce una chiave (l'array deve avere spazio libero)
void insert(int A[], int& dim, int chiave) {
    dim++;
    A[dim - 1] = chiave;
    increaseKey(A, dim - 1, chiave);    // risale fino al posto giusto
}

int main() {
    int A[20] = {5, 13, 2, 25, 7, 17, 20, 8, 4};
    int dim = 9;
    buildMaxHeap(A, dim);
    insert(A, dim, 15);
    for (int i = 0; i < dim; i++) cout << A[i] << " ";
    cout << endl << "massimo estratto: " << extractMax(A, dim) << endl;

    int B[] = {5, 13, 2, 25, 7, 17, 20, 8, 4};
    heapSort(B, 9);
    for (int i = 0; i < 9; i++) cout << B[i] << " ";
    cout << endl;
    return 0;
}

```

Quicksort

Il divide et impera rovesciato: tutto il lavoro si fa prima della ricorsione, nella partizione.

Nel caso peggiore è quadratico, ma in pratica è l'ordinamento più veloce, soprattutto nella sua versione randomizzata.

L'idea

Merge Sort divide l'array "a caso" (a metà) e fa tutto il lavoro dopo, nella fusione. Quicksort fa l'opposto: lavora **prima**, dividendo l'array in modo intelligente, così che dopo non serva più combinare nulla.

Immagina di dover mettere in ordine di altezza una classe di studenti. Scegli uno studente, il **pivot**, e chiedi a tutti quelli più bassi di lui di andare a sinistra e a quelli più alti di andare a destra. Il pivot è già al suo posto definitivo. Ora basta ripetere la stessa cosa separatamente nel gruppo di sinistra e in quello di destra: alla fine tutta la fila è ordinata, senza bisogno di "fondere" nulla.

Le tre fasi del divide et impera diventano:

1. **Divide**: si sceglie un pivot e si riorganizza $A[p..r]$ in due parti $A[p..q - 1]$ e $A[q + 1..r]$, tali che ogni elemento della prima sia $\leq A[q]$ e ogni elemento della seconda sia $> A[q]$ (questo è Partition).
2. **Impera**: si ordinano le due parti con due chiamate ricorsive.
3. **Combina**: non serve fare nulla, le parti sono già al posto giusto.

```
1 QuickSort(A, p, r)
2   if p < r then
3       q = Partition(A, p, r)    // il pivot finisce in posizione q
4       QuickSort(A, p, q - 1)
5       QuickSort(A, q + 1, r)
```

La procedura Partition

Usiamo la versione di Lomuto, quella del libro: il pivot è l'ultimo elemento $x = A[r]$. Durante la scansione l'array è diviso in quattro zone:

- $A[p..i]$: elementi già esaminati e $\leq x$;
- $A[i + 1..j - 1]$: elementi già esaminati e $> x$;
- $A[j..r - 1]$: elementi ancora da esaminare;
- $A[r]$: il pivot.

A ogni passo si guarda $A[j]$. Se è maggiore del pivot non si fa nulla: la zona dei "grandi" si allunga da sola, perché j avanza. Se invece è minore o uguale, si allarga di uno la zona dei "piccoli" (i aumenta) e si scambia $A[j]$ con il primo dei "grandi", che finisce così in coda alla sua zona. Alla fine il pivot viene scambiato con il primo dei grandi, $A[i + 1]$, e si ritrova esattamente tra le due zone.

```

1 Partition(A, p, r)
2     x = A[r]                // pivot
3     i = p - 1               // fine della zona dei piccoli
4     for j = p to r - 1 do
5         if A[j] <= x then
6             i = i + 1
7             scambia A[i] con A[j]
8     scambia A[i+1] con A[r]  // il pivot va tra le due zone
9     return i + 1

```

ESEMPIO PASSO PER PASSO

Partizioniamo $A = \{3, 9, 5, 1, 8, 2, 7, 4\}$. Il pivot è 4. Nella figura le celle chiare sono i "piccoli" (≤ 4), quelle grigie i "grandi", quella scura il pivot.

J	A[J]	AZIONE	ARRAY DOPO IL PASSO
1	3	$3 \leq 4$: $i = 1$, scambio $A[1]$ con sé stesso	3 9 5 1 8 2 7 4
2	9	$9 > 4$: nessuna azione	3 9 5 1 8 2 7 4
3	5	$5 > 4$: nessuna azione	3 9 5 1 8 2 7 4
4	1	$1 \leq 4$: $i = 2$, scambio $A[2]$ e $A[4]$	3 1 5 9 8 2 7 4
5	8	$8 > 4$: nessuna azione	3 1 5 9 8 2 7 4
6	2	$2 \leq 4$: $i = 3$, scambio $A[3]$ e $A[6]$	3 1 2 9 8 5 7 4
7	7	$7 > 4$: nessuna azione	3 1 2 9 8 5 7 4
fine	.	scambio $A[4]$ e $A[8]$, si restituisce 4	3 1 2 4 8 5 7 9

j = 4	3	1	5	9	8	2	7	4
j = 6	3	1	2	9	8	5	7	4
j = 7	3	1	2	9	8	5	7	4
fine	3	1	2	4	8	5	7	9
	1	2	3	4	5	6	7	8

Le zone di Partition: piccoli (verde chiaro), grandi (grigio), pivot (scuro).

Dopo la partizione il 4 è in posizione 4, la sua posizione definitiva. Quicksort prosegue ricorsivamente su $\{3, 1, 2\}$ e su $\{8, 5, 7, 9\}$.

DA SAPERE: CORRETTEZZA DI PARTITION

All'inizio di ogni iterazione del ciclo vale l'**invariante**: gli elementi in $A[p..i]$ sono $\leq x$, quelli in $A[i + 1..j - 1]$ sono $> x$ e $A[r] = x$. L'invariante è vero all'inizio (le due zone sono vuote), resta vero a ogni passo per come sono fatti i due casi, e alla fine ($j = r$) dice che tutto l'array, tranne il pivot, è diviso correttamente. Partition costa $\Theta(n)$, con $n = r - p + 1$: un confronto per elemento.

Prestazioni

Il tempo di Quicksort dipende da quanto è **bilanciata** la partizione, cioè da dove finisce il pivot.

CASO PEGGIORE

Il caso peggiore si ha quando la partizione è la più sbilanciata possibile: una parte vuota e l'altra con $n - 1$ elementi. Succede, con il pivot sull'ultimo elemento, proprio quando l'array è **già ordinato** (o ordinato al contrario). Ogni livello di ricorsione toglie un solo elemento e costa quanto la sua dimensione:

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

È lo stesso conto di Insertion Sort nel caso peggiore: $n + (n - 1) + \dots + 1$.

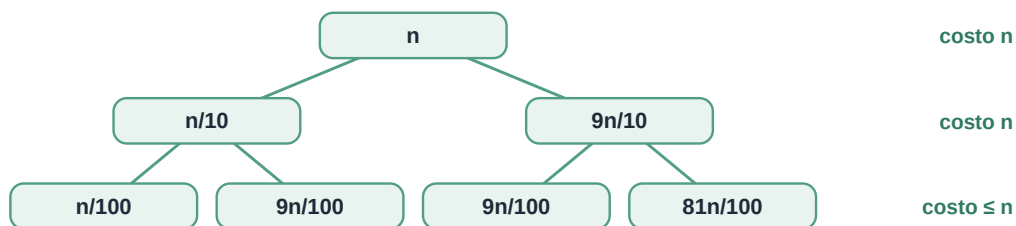
CASO MIGLIORE

Se il pivot cade sempre a metà, le due parti hanno dimensione circa $n/2$ e si ottiene la stessa ricorrenza di Merge Sort:

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

CASO MEDIO: ANCHE LO SBILANCIAMENTO COSTANTE VA BENE

La sorpresa è che non serve una divisione perfetta. Supponiamo che Partition produca sempre una divisione molto sbilanciata, ad esempio 9 a 1. L'albero di ricorsione ha ancora un costo di al più n per livello, e il ramo più lungo (quello che prende sempre i 9/10) arriva alla foglia dopo $\log_{10/9} n$ livelli, che è ancora $\Theta(\log n)$. Il totale resta $\Theta(n \log n)$. Qualunque divisione con proporzione costante, anche 99 a 1, dà $n \log n$: solo divisioni del tipo "tutto da una parte" portano al caso quadratico.



Partizione 9 a 1: ogni livello costa al più n e i livelli sono $\Theta(\log n)$.

DA SAPERE: PRESTAZIONI DI QUICKSORT

Caso peggiore $\Theta(n^2)$ (partizioni completamente sbilanciate, per esempio array già ordinato); caso migliore e caso medio $\Theta(n \log n)$. Quicksort è **in loco** ma **non stabile**. In pratica è il più veloce tra gli ordinamenti per confronto, perché le costanti nascoste nel suo $n \log n$ sono piccole.

Quicksort randomizzato

Il problema del caso peggiore è che dipende dall'ingresso: un utente che ci passa sempre array già ordinati ci farebbe lavorare in tempo quadratico ogni volta. La soluzione è introdurre il **caso**: invece di usare sempre $A[r]$ come pivot, si sceglie un indice a caso tra p e r e lo si scambia con $A[r]$ prima di partizionare.

```

1 RandomizedPartition(A, p, r)
2     i = Random(p, r)           // indice casuale uniforme tra p e r
3     scambia A[r] con A[i]
4     return Partition(A, p, r)
5
6 RandomizedQuickSort(A, p, r)
7     if p < r then
8         q = RandomizedPartition(A, p, r)
9         RandomizedQuickSort(A, p, q - 1)
10        RandomizedQuickSort(A, q + 1, r)

```

Ora nessun ingresso è "cattivo" di per sé: la sfortuna può capitare, ma dipende solo dalle scelte casuali, non da chi prepara i dati. È come mescolare il mazzo prima di giocare: nessuno può preparare un ordine delle carte che ti svantaggi apposta.

DA RICORDARE

Il tempo **atteso** di RandomizedQuickSort è **$O(n \log n)$** su qualunque ingresso. Il caso peggiore $\Theta(n^2)$ resta possibile in teoria, ma ha una probabilità trascurabile. La stessa idea di "pivot casuale" ritorna nel capitolo sulle statistiche d'ordine, con RandomizedSelect.

Implementazione in C++

Per la scelta casuale usiamo rand() della libreria standard: rand() % (r - p + 1) dà un numero tra 0 e r - p, a cui si somma p.

```

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

// Partition di Lomuto: pivot = A[r], restituisce la posizione finale del pivot
int partition(int A[], int p, int r) {
    int x = A[r];           // pivot
    int i = p - 1;          // ultimo indice della zona dei piccoli
    for (int j = p; j < r; j++) {
        if (A[j] <= x) {
            i++;
            swap(A[i], A[j]); // A[j] entra nella zona dei piccoli
        }
    }
    swap(A[i + 1], A[r]);    // il pivot va tra le due zone
    return i + 1;
}

// Sceglie un pivot casuale tra p e r e lo porta in fondo
int randomizedPartition(int A[], int p, int r) {
    int i = p + rand() % (r - p + 1);
    swap(A[i], A[r]);
    return partition(A, p, r);
}

// Ordina A[p..r]; chiamata iniziale: quickSort(A, 0, n - 1)
void quickSort(int A[], int p, int r) {
    if (p < r) {
        int q = randomizedPartition(A, p, r);
        quickSort(A, p, q - 1);
        quickSort(A, q + 1, r);
    }
}

```

```
    }  
}  
  
int main() {  
    srand(time(0));           // inizializza il generatore casuale  
    int A[] = {3, 9, 5, 1, 8, 2, 7, 4};  
    int n = 8;  
    quickSort(A, 0, n - 1);  
    for (int i = 0; i < n; i++) cout << A[i] << " ";  
    cout << endl;  
    return 0;  
}
```

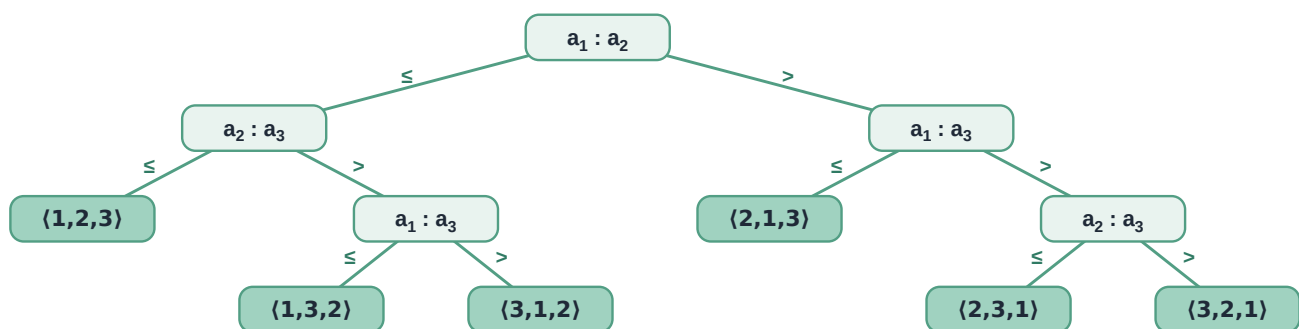
Ordinamento in tempo lineare

Perché nessun algoritmo basato sui confronti può battere $n \log n$, e come Counting Sort, Radix Sort e Bucket Sort riescono ad andare più veloci sfruttando informazioni sui valori delle chiavi.

Il limite inferiore degli ordinamenti per confronto

Insertion Sort, Merge Sort, Heapsort e Quicksort hanno una cosa in comune: l'unico modo in cui "guardano" le chiavi è **confrontarle** tra loro ($a_i \leq a_j?$). Si chiamano **ordinamenti per confronto**, e per loro esiste un limite invalicabile.

Lo si vede con l'**albero di decisione**. Ogni nodo interno rappresenta un confronto tra due elementi; a seconda dell'esito si scende a sinistra o a destra; ogni foglia rappresenta l'ordine finale prodotto dall'algoritmo. Un'esecuzione dell'algoritmo su un ingresso è un cammino dalla radice a una foglia, e il numero di confronti eseguiti è la lunghezza di quel cammino.



Albero di decisione di Insertion Sort su tre elementi: sei foglie, una per ogni permutazione possibile.

Il ragionamento è questo. Un algoritmo corretto deve poter produrre **ogni** permutazione dell'ingresso, perché ognuna può essere quella giusta: quindi l'albero ha almeno $n!$ foglie. Un albero binario di altezza h ha al più 2^h foglie. Quindi

$$2^h \geq n! \Rightarrow h \geq \log_2(n!) = \Omega(n \log n)$$

dove l'ultimo passaggio usa il fatto che $n!$ contiene almeno $n/2$ fattori maggiori o uguali a $n/2$, quindi $n! \geq (n/2)^{n/2}$. L'altezza h è il numero di confronti nel caso peggiore.

DA SAPERE: LIMITE INFERIORE

Qualunque ordinamento per confronto richiede **$\Omega(n \log n)$** confronti nel caso peggiore. Merge Sort e Heapsort sono quindi **asintoticamente ottimi** tra gli ordinamenti per confronto. Per scendere sotto $n \log n$ bisogna smettere di usare solo confronti e sfruttare la forma delle chiavi, come fanno gli algoritmi di questo capitolo.

Counting Sort

Counting Sort funziona quando le chiavi sono **interi in un intervallo piccolo**, da 0 a k . Invece di confrontare, **conta**. Se so che 5 elementi sono minori o uguali a un certo valore x , allora l'ultimo x va in posizione 5. Per gestire i valori ripetuti, ogni volta che si piazza un x si decrementa il contatore, così il prossimo x andrà una posizione più a sinistra.

L'analogia è quella dei posti a teatro assegnati per fasce di prezzo: se so quante persone hanno comprato biglietti di platea, so subito in quale fila iniziano quelli di galleria, senza bisogno di confrontare le persone a due a due.

Usa tre array: $A[1..n]$ l'ingresso, $B[1..n]$ l'uscita, $C[0..k]$ i contatori.

```
1 CountingSort(A, B, n, k)
2   for i = 0 to k do C[i] = 0
3   for j = 1 to n do           // 1) conta le occorrenze
4     C[A[j]] = C[A[j]] + 1
5   for i = 1 to k do           // 2) somme prefisse
6     C[i] = C[i] + C[i-1]      // ora C[i] = quanti elementi sono <= i
7   for j = n downto 1 do       // 3) piazza gli elementi, da destra
8     B[C[A[j]]] = A[j]
9     C[A[j]] = C[A[j]] - 1
```

ESEMPIO PASSO PER PASSO

Ordiniamo $A = \langle 4, 1, 3, 4, 0, 3, 1, 4 \rangle$, con $n = 8$ e $k = 4$.

A	4	1	3	4	0	3	1	4
	1	2	3	4	5	6	7	8
C conta	1	2	0	2	3			
C prefisse	1	3	3	5	8			
	0	1	2	3	4			

Ingresso, contatori dopo il primo ciclo e dopo le somme prefisse (indici di C da 0 a 4).

Dopo le somme prefisse, $C[3] = 5$ significa che 5 elementi sono ≤ 3 , quindi l'ultimo 3 va in $B[5]$. Ora si scorre A da destra:

J	A[J]	DOVE VA	C DOPO IL PASSO
8	4	$B[C[4]] = B[8]$	$C[4] = 7$
7	1	$B[C[1]] = B[3]$	$C[1] = 2$
6	3	$B[C[3]] = B[5]$	$C[3] = 4$
5	0	$B[C[0]] = B[1]$	$C[0] = 0$
4	4	$B[7]$	$C[4] = 6$
3	3	$B[4]$	$C[3] = 3$
2	1	$B[2]$	$C[1] = 1$
1	4	$B[6]$	$C[4] = 5$

B	0	1	1	3	3	4	4	4
	1	2	3	4	5	6	7	8

Uscita ordinata.

Perché scorrere da destra? Guardiamo i tre 4: quello in posizione 8 di A va in B[8], quello in posizione 4 in B[7], quello in posizione 1 in B[6]. L'ordine relativo dei tre 4 è rimasto quello di partenza: Counting Sort è **stabile**. Scorrendo da sinistra sarebbe invertito.

DA SAPERE: COUNTING SORT

Tempo $\Theta(n + k)$: un ciclo su k per i contatori e due cicli su n . Quando $k = O(n)$ il tempo è $\Theta(n)$. È **stabile** ma **non in loco** (servono B e C). Non fa nessun confronto tra elementi, per questo il limite $\Omega(n \log n)$ non si applica. Diventa inutile se k è enorme rispetto a n (per esempio ordinare 10 numeri tra 0 e un miliardo).

Radix Sort

E se le chiavi sono grandi, per esempio numeri di molte cifre? Radix Sort li ordina **una cifra alla volta, partendo dalla meno significativa**, usando a ogni passata un ordinamento stabile (tipicamente Counting Sort, con $k = 9$ per le cifre decimali).

Sembra controintuitivo partire dalle unità, ma è proprio la stabilità a farlo funzionare: quando si ordina per la cifra delle centinaia, due numeri con la stessa centinaia restano nell'ordine stabilito dalle passate precedenti, cioè ordinati per decine e unità. È il modo in cui si ordinano a mano le schede di un archivio per data: prima per giorno, poi per mese, poi per anno, ogni volta senza rimescolare chi ha lo stesso valore.

```

1 RadixSort(A, d)
2   for i = 1 to d do
3       ordina A in modo stabile secondo la cifra i  // i = 1: unità'

```

ESEMPIO

Ordiniamo sette numeri di tre cifre. Ogni colonna è il risultato di una passata; la cifra in grassetto è quella usata in quella passata.

INGRESSO	DOPO LE UNITÀ	DOPO LE DECINE	DOPO LE CENTINAIA
512	290	307	133
307	512	512	290
845	133	718	307
133	845	133	456
290	456	845	512
718	307	456	718
456	718	290	845

Nella passata delle decine, 512 e 718 hanno la stessa cifra 1: restano nell'ordine della passata precedente, in cui 512 veniva prima di 718 per via delle unità.

DA SAPERE: RADIX SORT

Con n numeri di d cifre, ciascuna tra 0 e $k - 1$, e Counting Sort come ordinamento stabile, Radix Sort costa $\Theta(d(n + k))$. Se d è costante e $k = O(n)$, il tempo è $\Theta(n)$. L'ordinamento usato per le singole cifre **deve essere stabile**, altrimenti le passate precedenti andrebbero perse.

Bucket Sort

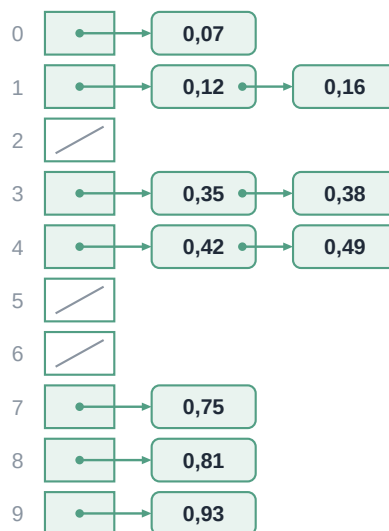
Bucket Sort si usa quando le chiavi sono numeri reali **distribuiti in modo uniforme** nell'intervallo $[0, 1)$. Si divide l'intervallo in n "secchi" (bucket) di uguale ampiezza: il secchio i raccoglie i valori tra i/n e $(i + 1)/n$. Si distribuiscono gli elementi nei secchi, si ordina ogni secchio (con Insertion Sort, visto che conterrà pochi elementi) e si concatenano i secchi in ordine.

È come smistare la posta per CAP: prima si divide in cassette per zona, poi si mette in ordine ogni cassetta, che è piccola, e infine si mettono in fila le cassette.

```
1 BucketSort(A, n)
2   crea n liste vuote B[0..n-1]
3   for i = 1 to n do
4     inserisci A[i] nella lista B[floor(n * A[i])]
5   for i = 0 to n-1 do
6     ordina la lista B[i] con InsertionSort
7   concatena B[0], B[1], ..., B[n-1] in ordine
```

ESEMPIO

Con $n = 10$ e $A = \{0,42; 0,07; 0,81; 0,35; 0,49; 0,93; 0,12; 0,38; 0,75; 0,16\}$, l'elemento x finisce nel secchio $\lfloor 10x \rfloor$: $0,42$ nel secchio 4 , $0,07$ nel secchio 0 , e così via.



I dieci secchi dopo la distribuzione e l'ordinamento interno; la concatenazione dà l'array ordinato.

DA SAPERE: BUCKET SORT

Se l'ingresso è uniforme, ogni secchio contiene in media un numero costante di elementi e il tempo **atteso** è $\Theta(n)$. Nel caso peggiore (tutti gli elementi nello stesso secchio) degenera nel costo di Insertion Sort, $\Theta(n^2)$. L'ipotesi sulla distribuzione è essenziale.

Riepilogo degli ordinamenti

ALGORITMO	CASO PEGGIORE	CASO MEDIO	IN LOCO	STABILE
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	sì	sì
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	no	sì
Heapsort	$O(n \log n)$	$O(n \log n)$	sì	no
Quicksort	$\Theta(n^2)$	$\Theta(n \log n)$	sì	no
Counting Sort	$\Theta(n + k)$	$\Theta(n + k)$	no	sì
Radix Sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	no	sì
Bucket Sort	$\Theta(n^2)$	$\Theta(n)$ atteso	no	sì

Implementazione in C++

Counting Sort riceve il valore massimo k ; per Radix Sort usiamo una variante di Counting Sort che ordina secondo la cifra selezionata da esp (1 per le unità, 10 per le decine, ...). Bucket Sort usa secchi a capacità fissa, sufficiente per l'esempio.

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

// Ordina A[0..n-1] con valori tra 0 e k; il risultato va in B
void countingSort(int A[], int B[], int n, int k) {
    int* C = new int[k + 1];
    for (int i = 0; i <= k; i++) C[i] = 0;
    for (int j = 0; j < n; j++) C[A[j]]++;           // conta
    for (int i = 1; i <= k; i++) C[i] += C[i - 1];   // somme prefisse
    for (int j = n - 1; j >= 0; j--) {               // da destra: stabile
        B[C[A[j]] - 1] = A[j];                       // -1 perche' B parte da 0
        C[A[j]]--;
    }
    delete[] C;
}

// Counting Sort stabile sulla cifra indicata da esp (1, 10, 100, ...)
void countingSortCifra(int A[], int n, int esp) {
    int* B = new int[n];
    int C[10] = {0};
    for (int j = 0; j < n; j++) C[(A[j] / esp) % 10]++;
    for (int i = 1; i < 10; i++) C[i] += C[i - 1];
    for (int j = n - 1; j >= 0; j--) {
        int cifra = (A[j] / esp) % 10;
        B[C[cifra] - 1] = A[j];
        C[cifra]--;
    }
    for (int j = 0; j < n; j++) A[j] = B[j];          // ricopia in A
    delete[] B;
}

// Radix Sort per numeri con al piu' d cifre
void radixSort(int A[], int n, int d) {
    int esp = 1;
    for (int i = 0; i < d; i++) {
        countingSortCifra(A, n, esp);
    }
}
```

```

        esp *= 10;
    }
}

// Bucket Sort per valori in [0, 1); al massimo 20 elementi
void bucketSort(double A[], int n) {
    double secchi[20][20];           // secchi[i] contiene cont[i] valori
    int cont[20] = {0};
    for (int i = 0; i < n; i++) {
        int s = (int)(n * A[i]);      // indice del secchio
        secchi[s][cont[s]] = A[i];
        cont[s]++;
    }
    int pos = 0;
    for (int s = 0; s < n; s++) {
        // Insertion Sort sul secchio s
        for (int j = 1; j < cont[s]; j++) {
            double chiave = secchi[s][j];
            int i = j - 1;
            while (i >= 0 && secchi[s][i] > chiave) {
                secchi[s][i + 1] = secchi[s][i];
                i--;
            }
            secchi[s][i + 1] = chiave;
        }
        for (int j = 0; j < cont[s]; j++) A[pos++] = secchi[s][j]; // concatena
    }
}

int main() {
    int A[] = {4, 1, 3, 4, 0, 3, 1, 4};
    int B[8];
    countingSort(A, B, 8, 4);
    for (int i = 0; i < 8; i++) cout << B[i] << " ";
    cout << endl;

    int R[] = {512, 307, 845, 133, 290, 718, 456};
    radixSort(R, 7, 3);
    for (int i = 0; i < 7; i++) cout << R[i] << " ";
    cout << endl;

    double D[] = {0.42, 0.07, 0.81, 0.35, 0.49, 0.93, 0.12, 0.38, 0.75, 0.16};
    bucketSort(D, 10);
    for (int i = 0; i < 10; i++) cout << D[i] << " ";
    cout << endl;
    return 0;
}

```

Mediane e statistiche d'ordine

Come trovare il minimo, il massimo, la mediana o in generale l' i -esimo elemento più piccolo di un insieme senza ordinarlo tutto: in tempo lineare atteso con la selezione randomizzata, e in tempo lineare garantito con la mediana delle mediane.

Il problema della selezione

La **i -esima statistica d'ordine** di un insieme di n elementi è l'elemento che occuperebbe la posizione i se l'insieme fosse ordinato. La prima statistica d'ordine è il **minimo**, la n -esima è il **massimo**. La **mediana** è l'elemento "di mezzo": per n dispari è unica e sta in posizione $(n + 1)/2$; per n pari ce ne sono due, la mediana inferiore in posizione $\lfloor (n + 1)/2 \rfloor$ e quella superiore in posizione $\lceil (n + 1)/2 \rceil$. Quando si dice "la mediana" senza altre precisazioni si intende quella inferiore.

DEFINIZIONE: PROBLEMA DELLA SELEZIONE

Input: un insieme A di n numeri distinti e un intero i con $1 \leq i \leq n$. **Output:** l'elemento x di A che è maggiore di esattamente $i - 1$ altri elementi di A .

Una soluzione ovvia esiste già: ordinare con Merge Sort o Heapsort in $\Theta(n \log n)$ e prendere $A[i]$. Ma ordinare tutto per trovare un solo elemento è come mettere in fila per altezza tutta la scuola solo per sapere chi è il terzo più basso: si fa molto più lavoro del necessario. Vedremo che si può fare in **tempo lineare**.

Minimo e massimo

Per trovare il minimo basta scorrere l'array tenendo il più piccolo visto finora: $n - 1$ confronti, ed è ottimo, perché ogni elemento tranne il vincitore deve "perdere" almeno un confronto.

```
1 Minimum(A, n)
2   min = A[1]
3   for i = 2 to n do
4       if A[i] < min then min = A[i]
5   return min
```

MINIMO E MASSIMO INSIEME

Se servono entrambi, cercarli separatamente costa $2(n - 1)$ confronti. Si può fare meglio lavorando **a coppie**: si prendono due elementi alla volta, li si confronta tra loro, e poi si confronta solo il più piccolo con il minimo corrente e solo il più grande con il massimo corrente. Sono 3 confronti ogni 2 elementi invece di 4.

Per esempio con $\langle 7, 2, 9, 4, 1, 8 \rangle$: la prima coppia $(7, 2)$ inizializza $\min = 2$ e $\max = 7$ con un confronto. La coppia $(9, 4)$: $4 < 9$, poi 4 contro $\min = 2$ (resta 2), 9 contro $\max = 7$ (diventa 9). La coppia $(1, 8)$: $1 < 8$, poi 1 contro 2 (diventa 1), 8 contro 9 (resta 9). Risultato: $\min = 1$, $\max = 9$ con 7 confronti invece di 10.

DA RICORDARE

Minimo e massimo simultanei richiedono al più $3\lfloor n/2 \rfloor$ confronti, invece di $2n - 2$.

Selezione randomizzata

L'idea viene direttamente da Quicksort. Dopo una Partition sappiamo esattamente in che posizione è finito il pivot: se il pivot è in posizione q , allora è la $(q - p + 1)$ esima statistica d'ordine del sottoarray $A[p..r]$, e chiamiamo questo numero k . Ora:

- se $i = k$, il pivot è proprio l'elemento cercato;
- se $i < k$, l'elemento cercato sta nella parte sinistra, ed è ancora la i esima di quella parte;
- se $i > k$, sta nella parte destra, e lì è la $(i - k)$ esima, perché i k elementi a sinistra (pivot compreso) sono tutti più piccoli.

La differenza cruciale rispetto a Quicksort è che si prosegue ricorsivamente su **una sola** delle due parti. È come cercare una parola nel dizionario: aperto a caso, capisci se la parola sta prima o dopo, e butti via l'altra metà.

```
1 RandomizedSelect(A, p, r, i)
2   if p = r then return A[p]           // un solo elemento
3   q = RandomizedPartition(A, p, r)
4   k = q - p + 1                       // posizione del pivot nel sottoarray
5   if i = k then return A[q]
6   else if i < k then
7       return RandomizedSelect(A, p, q - 1, i)
8   else
9       return RandomizedSelect(A, q + 1, r, i - k)
```

ESEMPIO

Cerchiamo la terza statistica d'ordine ($i = 3$) di $A = \{3, 9, 5, 1, 8, 2, 7, 4\}$. Per seguire il conto supponiamo che il pivot scelto a caso sia, ogni volta, l'ultimo elemento.

1. Partition sull'intero array, pivot 4: si ottiene $\{3, 1, 2, 4, 8, 5, 7, 9\}$ con $q = 4$, quindi $k = 4$. Poiché $i = 3 < 4$, si cerca la terza statistica nella parte sinistra $\{3, 1, 2\}$.
2. Partition su $\{3, 1, 2\}$, pivot 2: si ottiene $\{1, 2, 3\}$ con $k = 2$. Poiché $i = 3 > 2$, si cerca la statistica $i - k = 1$ nella parte destra $\{3\}$.
3. Il sottoarray ha un solo elemento: la risposta è **3**. Infatti l'array ordinato è $\{1, 2, 3, 4, 5, 7, 8, 9\}$.

DA SAPERE: RANDOMIZEDSELECT

Tempo **atteso** $\Theta(n)$: in media ogni partizione elimina una frazione costante degli elementi, e la somma $n + cn + c^2n + \dots$ con $c < 1$ è lineare. Nel **caso peggiore** (pivot sempre estremo) il tempo è $\Theta(n^2)$, come Quicksort.

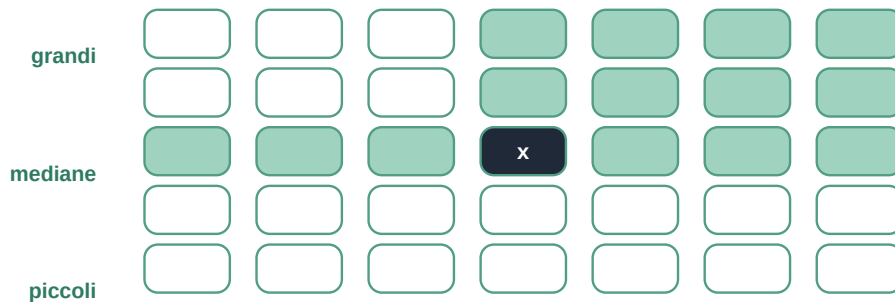
Selezione in tempo lineare nel caso peggiore

L'algoritmo Select garantisce tempo lineare anche nel caso peggiore. Il trucco è scegliere un pivot che sia sicuramente "abbastanza centrale", usando la **mediana delle mediane**.

1. Si dividono gli n elementi in $\lceil n/5 \rceil$ gruppi da 5 (l'ultimo può essere più corto).
2. Si trova la mediana di ogni gruppo, ordinandolo con Insertion Sort: costa una costante per gruppo.
3. Si trova ricorsivamente, con Select stesso, la mediana x delle $\lceil n/5 \rceil$ mediane.
4. Si partiziona l'array usando x come pivot; sia k la posizione di x .
5. Se $i = k$ si restituisce x , altrimenti si prosegue ricorsivamente con Select sulla parte giusta, come in RandomizedSelect.

PERCHÉ IL PIVOT È BUONO

Disponiamo i gruppi come colonne, ognuna ordinata dal basso (piccoli) verso l'alto (grandi), e le colonne ordinate per mediana. La mediana delle mediane x sta al centro.



Gruppi da cinque come colonne, mediane nella riga centrale e x in nero: le celle verdi nel blocco in alto a destra sono sicuramente maggiori o uguali a x .

Metà delle mediane, quelle delle colonne a destra di x , sono $\geq x$, e in ciascuna di quelle colonne anche i due elementi sopra la mediana sono $\geq x$. Quindi ci sono almeno circa $3n/10$ elementi maggiori di x e, per simmetria, altrettanti minori. La parte su cui si prosegue ha al più circa $7n/10$ elementi (esattamente al più $7n/10 + 6$). La ricorrenza è

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

e poiché $1/5 + 7/10 = 9/10 < 1$, il lavoro si riduce di una frazione costante a ogni livello e la soluzione è $T(n) = O(n)$.

DA SAPERE: SELECT

Select trova qualunque statistica d'ordine in tempo $O(n)$ nel caso peggiore. I gruppi devono avere almeno 5 elementi: con gruppi da 3 la somma delle frazioni non scende sotto 1 e il tempo non è più lineare. In pratica RandomizedSelect è più veloce, perché Select ha costanti nascoste alte.

Implementazione in C++

Riusiamo la Partition di Lomuto vista con Quicksort. Il parametro i resta "da 1" (prima, seconda, ... statistica), mentre gli indici dell'array partono da 0.

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

int partition(int A[], int p, int r) {
    int x = A[r];
    int i = p - 1;
    for (int j = p; j < r; j++) {
        if (A[j] <= x) {
            i++;
            swap(A[i], A[j]);
        }
    }
    swap(A[i + 1], A[r]);
    return i + 1;
}

int randomizedPartition(int A[], int p, int r) {
    int i = p + rand() % (r - p + 1);
    swap(A[i], A[r]);
```

```

    return partition(A, p, r);
}

// Restituisce la i-esima statistica d'ordine di A[p..r] (i parte da 1)
int randomizedSelect(int A[], int p, int r, int i) {
    if (p == r) return A[p];
    int q = randomizedPartition(A, p, r);
    int k = q - p + 1; // posizione del pivot nel sottoarray
    if (i == k) return A[q];
    else if (i < k) return randomizedSelect(A, p, q - 1, i);
    else return randomizedSelect(A, q + 1, r, i - k);
}

// Minimo e massimo con circa 3n/2 confronti
void minMax(int A[], int n, int& mn, int& mx) {
    int inizio;
    if (n % 2 == 1) { // n dispari: il primo inizializza entrambi
        mn = mx = A[0];
        inizio = 1;
    } else { // n pari: la prima coppia
        if (A[0] < A[1]) { mn = A[0]; mx = A[1]; }
        else { mn = A[1]; mx = A[0]; }
        inizio = 2;
    }
    for (int i = inizio; i + 1 < n; i += 2) {
        int piccolo = A[i], grande = A[i + 1];
        if (piccolo > grande) swap(piccolo, grande); // 1 confronto nella coppia
        if (piccolo < mn) mn = piccolo; // 1 confronto col minimo
        if (grande > mx) mx = grande; // 1 confronto col massimo
    }
}

int main() {
    srand(time(0));
    int A[] = {3, 9, 5, 1, 8, 2, 7, 4};
    cout << "terza statistica: " << randomizedSelect(A, 0, 7, 3) << endl;
    int B[] = {7, 2, 9, 4, 1, 8};
    int mn, mx;
    minMax(B, 6, mn, mx);
    cout << "min " << mn << " max " << mx << endl;
    return 0;
}

```

Tabelle hash

Come realizzare un dizionario in cui inserire, cercare e cancellare costano in media un tempo costante: indirizzamento diretto, funzioni hash, gestione delle collisioni con liste di trabocco e con indirizzamento aperto.

Il problema del dizionario

Molte applicazioni hanno bisogno di un **dizionario**: un insieme dinamico di elementi, ciascuno identificato da una **chiave**, su cui si eseguono tre operazioni.

- **Insert(T, x)**: inserisce l'elemento x .
- **Search(T, k)**: restituisce l'elemento con chiave k , oppure NIL se non c'è.
- **Delete(T, x)**: elimina l'elemento x .

Un compilatore che tiene la tabella dei simboli, una rubrica, la cache di un browser: sono tutti dizionari. Con una lista la ricerca costa $\Theta(n)$; con un albero bilanciato (come gli alberi rosso neri del prossimo capitolo) costa $\Theta(\log n)$. Le tabelle hash puntano più in alto: **$\Theta(1)$ in media**.

Tabelle a indirizzamento diretto

Se le chiavi sono interi presi da un universo piccolo $U = \{0, 1, \dots, m - 1\}$, la soluzione è banale: si usa un array $T[0..m - 1]$ e l'elemento con chiave k si mette nella cella $T[k]$. Ogni cella si chiama anche **slot**. Tutte e tre le operazioni costano $\Theta(1)$.

```

1 DirectAddressSearch(T, k)
2   return T[k]
3 DirectAddressInsert(T, x)
4   T[x.chiave] = x
5 DirectAddressDelete(T, x)
6   T[x.chiave] = NIL
```

È come un parcheggio con un posto riservato per ogni targa possibile: trovare un'auto è immediato, ma se le targhe possibili sono milioni e le auto sono cinquanta il parcheggio è quasi tutto vuoto. Quando l'universo U è enorme rispetto al numero di chiavi effettivamente usate, l'indirizzamento diretto spreca troppa memoria, o è proprio impossibile.

Tabelle hash

Una **tabella hash** usa un array di m slot, con m molto più piccolo di $|U|$, e una **funzione hash** h che "comprime" ogni chiave in un indice:

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

L'elemento con chiave k va nello slot $h(k)$, e si dice che k **viene mappata** in $h(k)$. Il prezzo da pagare è che due chiavi diverse possono finire nello stesso slot: è una **collisione**. Le collisioni sono inevitabili, perché ci sono più chiavi possibili che slot (è il principio dei cassetti). Bisogna quindi avere una funzione hash che le renda rare e un metodo per gestirle quando capitano.

DEFINIZIONE: FATTORE DI CARICO

Data una tabella con m slot che contiene n elementi, il **fattore di carico** è $\alpha = n/m$, il numero medio di elementi per slot. Con le liste di trabocco può essere maggiore di 1; con l'indirizzamento aperto è sempre al più 1.

Risoluzione delle collisioni per concatenazione

Il metodo più semplice: ogni slot contiene il puntatore a una **lista concatenata** (lista di trabocco) con tutti gli elementi che vi sono stati mappati.

```
1 ChainedHashInsert(T, x)
2   inserisci x in testa alla lista T[h(x.chiave)]
3 ChainedHashSearch(T, k)
4   cerca un elemento con chiave k nella lista T[h(k)]
5 ChainedHashDelete(T, x)
6   rimuovi x dalla lista T[h(x.chiave)]
```

ESEMPIO

Tabella con $m = 7$ e $h(k) = k \bmod 7$. Inseriamo nell'ordine 15, 8, 23, 4, 30, 11, 19. I resti sono: $15 \rightarrow 1$, $8 \rightarrow 1$, $23 \rightarrow 2$, $4 \rightarrow 4$, $30 \rightarrow 2$, $11 \rightarrow 4$, $19 \rightarrow 5$. Poiché si inserisce in testa, l'ultimo arrivato compare per primo.

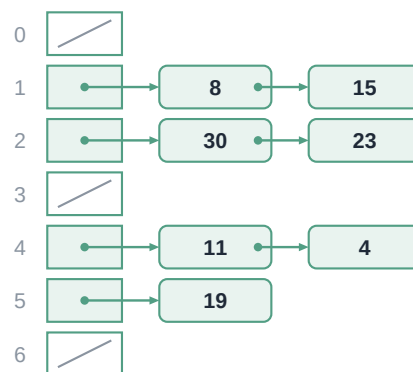


Tabella hash con concatenazione: gli slot 1, 2 e 4 hanno avuto una collisione.

DA SAPERE: COSTI DELLA CONCATENAZIONE

Insert costa $O(1)$ nel caso peggiore (inserimento in testa). **Delete** costa $O(1)$ se le liste sono doppiamente concatenate e si ha già il puntatore all'elemento. **Search** costa quanto la lista da scorrere: nel caso peggiore $\Theta(n)$ (tutte le chiavi nello stesso slot), ma sotto l'ipotesi di **hashing uniforme semplice** (ogni chiave ha la stessa probabilità di finire in ciascuno degli m slot, indipendentemente dalle altre) il tempo medio, sia con successo sia senza, è $\Theta(1 + \alpha)$. Se m è proporzionale a n , α è una costante e la ricerca costa $\Theta(1)$ in media. Le dimostrazioni sono nella dispensa di Laboratorio.

Funzioni hash

Una buona funzione hash soddisfa, almeno in modo approssimato, l'ipotesi di hashing uniforme semplice: distribuisce le chiavi in modo uniforme sugli slot, senza farsi influenzare da regolarità nei dati. Si assume che le chiavi siano numeri naturali; se non lo sono (per esempio stringhe) le si interpreta come numeri. Una stringa di caratteri ASCII si può vedere come un numero in base 128: "pt" diventa $112 \cdot 128 + 116 = 14\,452$, perché i codici di p e t sono 112 e 116.

METODO DELLA DIVISIONE

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

È velocissimo. La scelta di m è delicata: bisogna evitare le potenze di 2, perché se $m = 2^p$ allora $h(k)$ è fatto solo dai p bit meno significativi di k e ignora tutto il resto della chiave. Una buona scelta è un **numero primo non vicino a una potenza di 2**. Per esempio, per circa 2000 chiavi con una media accettabile di 3 elementi per lista, $m = 701$ va bene: è primo e lontano da 512 e da 1024.

METODO DELLA MOLTIPLICAZIONE

Si sceglie una costante A con $0 < A < 1$, si moltiplica k per A , si prende la parte frazionaria e la si moltiplica per m :

$$h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$$

dove " $k \cdot A \bmod 1$ " indica la parte frazionaria $k \cdot A - \lfloor k \cdot A \rfloor$. Qui il valore di m non è critico e anzi si sceglie di solito una potenza di 2, perché il calcolo diventa una moltiplicazione e uno spostamento di bit. Knuth suggerisce $A \approx (\sqrt{5} - 1)/2 \approx 0,618$.

Esempio con $m = 16$, $A = 0,618$ e $k = 21$: $k \cdot A = 12,978$, la parte frazionaria è 0,978, moltiplicata per 16 dà 15,6, quindi $h(21) = 15$.

HASHING UNIVERSALE

Qualunque funzione hash fissata ha un punto debole: un avversario che la conosce può scegliere n chiavi che vanno tutte nello stesso slot, e la ricerca diventa $\Theta(n)$. La difesa è la stessa di Quicksort randomizzato: scegliere la funzione **a caso**, all'inizio dell'esecuzione, da una famiglia opportuna.

DEFINIZIONE: FAMIGLIA UNIVERSALE

Una famiglia H di funzioni hash da U in $\{0, \dots, m - 1\}$ è **universale** se, per ogni coppia di chiavi distinte k e l , il numero di funzioni h di H per cui $h(k) = h(l)$ è al più $|H|/m$. In altre parole, scegliendo h a caso in H , la probabilità che due chiavi fissate collidano è al più $1/m$, la stessa che si avrebbe scegliendo i due slot completamente a caso.

Una famiglia universale classica si costruisce così: si sceglie un primo p più grande di ogni chiave e si prendono tutte le funzioni

$$h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$$

con a tra 1 e $p - 1$ e b tra 0 e $p - 1$. Per esempio con $p = 17$, $m = 6$, $a = 3$, $b = 4$ si ha $h(8) = ((24 + 4) \bmod 17) \bmod 6 = 11 \bmod 6 = 5$.

DA RICORDARE

Con l'hashing universale e la concatenazione, il tempo atteso di ogni sequenza di n operazioni con $O(m)$ inserimenti è $\Theta(n)$, cioè $\Theta(1)$ per operazione, **qualunque** sia l'ingresso: la casualità sta nella scelta della funzione, non nelle chiavi.

Indirizzamento aperto

Nell'indirizzamento aperto non ci sono liste: tutti gli elementi stanno **dentro la tabella**. Se lo slot calcolato è occupato, se ne prova un altro, e poi un altro ancora, secondo una sequenza di ispezione (probe sequence) che dipende dalla chiave. È come cercare posto in un parcheggio senza posti riservati: vai al tuo posto preferito, se è occupato provi il successivo previsto dalla tua regola, finché ne trovi uno libero.

La funzione hash prende quindi due argomenti, la chiave e il numero del tentativo:

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

e la sequenza $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$ deve essere una permutazione degli slot, così che, se c'è un posto libero, prima o poi lo si trovi. La tabella può riempirsi: α non supera mai 1.

```
1 HashInsert(T, k)
2   i = 0
3   repeat
4     j = h(k, i)
5     if T[j] = NIL then
6       T[j] = k
7       return j
8     else i = i + 1
9   until i = m
10  errore "overflow della tabella hash"
11
12 HashSearch(T, k)
13   i = 0
14   repeat
15     j = h(k, i)
16     if T[j] = k then return j
17     i = i + 1
18   until T[j] = NIL or i = m
19   return NIL
```

La ricerca segue esattamente la stessa sequenza dell'inserimento, e si ferma con insuccesso appena trova uno slot vuoto: se la chiave ci fosse, sarebbe stata messa lì.

DA SAPERE: LA CANCELLAZIONE

Non si può cancellare semplicemente scrivendo NIL in uno slot: una ricerca successiva si fermerebbe lì e non troverebbe chiavi inserite dopo, che avevano "saltato" quello slot perché era occupato. Si scrive invece un valore speciale **DELETED**: l'inserimento lo tratta come libero, la ricerca lo tratta come occupato e prosegue. Per questo, quando servono molte cancellazioni, si preferisce la concatenazione.

SCANSIONE LINEARE

$$h(k, i) = (h'(k) + i) \bmod m$$

dove h' è una funzione hash ordinaria. Si prova lo slot $h'(k)$, poi quello successivo, poi quello dopo ancora, ricominciando da 0 alla fine della tabella. È semplice ma soffre di **agglomerazione primaria** (primary clustering): si formano lunghe sequenze di slot occupati consecutivi, e ogni chiave che cade in un punto qualsiasi del gruppo lo allunga ancora. È come la coda che si forma davanti a un unico sportello: più è lunga, più attira gente.

SCANSIONE QUADRATICA

$$h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$$

con c_2 diverso da 0. I salti crescono con i , quindi i gruppi contigui non si formano. Resta però l'**agglomerazione secondaria**: due chiavi con lo stesso $h'(k)$ seguono esattamente la stessa sequenza. Le costanti e m vanno scelte in modo che la sequenza visiti tutti gli slot.

DOPPIO HASHING

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

Il passo $h_2(k)$ dipende dalla chiave, quindi due chiavi con lo stesso slot iniziale seguono in genere sequenze diverse. È il metodo migliore dei tre: si comporta quasi come l'hashing uniforme ideale. Perché la sequenza tocchi tutti gli slot, $h_2(k)$ deve essere **primo con m** : per esempio m primo e $h_2(k)$ tra 1 e $m - 1$, oppure m potenza di 2 e $h_2(k)$ sempre dispari.

ESEMPIO A CONFRONTO

Tabella con $m = 11$. Inseriamo nell'ordine 14, 25, 36, 7, 18, 3, 29 con $h'(k) = h_1(k) = k \bmod 11$ e, per il doppio hashing, $h_2(k) = 1 + (k \bmod 10)$. I valori iniziali sono $14 \rightarrow 3$, $25 \rightarrow 3$, $36 \rightarrow 3$, $7 \rightarrow 7$, $18 \rightarrow 7$, $3 \rightarrow 3$, $29 \rightarrow 7$: molte collisioni apposta.

Scansione lineare. 14 va in 3. 25 trova 3 occupato e va in 4. 36 trova occupati 3 e 4 e va in 5. 7 va in 7. 18 trova 7 occupato e va in 8. 3 prova 3, 4, 5 e va in 6. 29 prova 7 e 8 e va in 9.

lineare				14	25	36	3	7	18	29	
	0	1	2	3	4	5	6	7	8	9	10

Con la scansione lineare si forma un unico blocco di sette slot consecutivi.

Doppio hashing. 14 va in 3. Per 25: $h_1 = 3$ occupato, passo $h_2 = 6$, si prova $3 + 6 = 9$, libero. Per 36: passo 7, si prova 10, libero. 7 va in 7. Per 18: $h_1 = 7$ occupato, passo 9, si prova $16 \bmod 11 = 5$, libero. Per 3: $h_1 = 3$ occupato, passo 4, si prova 7 (occupato) e poi $11 \bmod 11 = 0$, libero. Per 29: $h_1 = 7$ occupato, passo 10, si prova $17 \bmod 11 = 6$, libero.

doppio	3			14		18	29	7		25	36
	0	1	2	3	4	5	6	7	8	9	10

Con il doppio hashing le stesse chiavi si spargono nella tabella.

DA SAPERE: COSTI DELL'INDIRIZZAMENTO APERTO

Sotto l'ipotesi di hashing uniforme e con $\alpha < 1$, il numero atteso di ispezioni è al più $1/(1 - \alpha)$ per una ricerca senza successo (e per un inserimento) e al più $(1/\alpha) \ln(1/(1 - \alpha))$ per una ricerca con successo. Con la tabella piena a metà ($\alpha = 0,5$) servono in media al più 2 ispezioni per una ricerca senza successo; piena al 90 per cento, al più 10. Le prestazioni crollano quando la tabella si avvicina al pieno.

ASPETTO	CONCATENAZIONE	INDIRIZZAMENTO APERTO
Dove stanno gli elementi	in liste esterne	dentro la tabella
Fattore di carico	anche maggiore di 1	sempre al più 1
Cancellazione	semplice	serve il marcatore DELETED
Memoria	puntatori extra	nessun puntatore

Implementazione in C++

Prima la concatenazione con $m = 7$ e liste semplici (struct con puntatore al successivo), poi l'indirizzamento aperto con scansione lineare e con doppio hashing, usando -1 come slot vuoto.

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

const int M = 7; // numero di slot per la concatenazione

struct Nodo {
    int chiave;
    Nodo* succ; // elemento successivo nella lista
};

int h(int k) { return k % M; } // metodo della divisione

// Inserisce in testa alla lista dello slot h(k)
```

```

void inserisciCatena(Nodo* T[], int k) {
    Nodo* nuovo = new Nodo;
    nuovo->chiave = k;
    nuovo->succ = T[h(k)];
    T[h(k)] = nuovo;
}

// Restituisce true se la chiave k e' presente
bool cercaCatena(Nodo* T[], int k) {
    Nodo* p = T[h(k)];
    while (p != NULL) {
        if (p->chiave == k) return true;
        p = p->succ;
    }
    return false;
}

const int M2 = 11;                // slot per l'indirizzamento aperto
const int VUOTO = -1;

int h1(int k) { return k % M2; }
int h2(int k) { return 1 + (k % 10); }

// Scansione lineare: h(k, i) = (h1(k) + i) mod M2
int inserisciLineare(int T[], int k) {
    for (int i = 0; i < M2; i++) {
        int j = (h1(k) + i) % M2;
        if (T[j] == VUOTO) { T[j] = k; return j; }
    }
    return -1;                    // tabella piena
}

// Doppio hashing: h(k, i) = (h1(k) + i * h2(k)) mod M2
int inserisciDoppio(int T[], int k) {
    for (int i = 0; i < M2; i++) {
        int j = (h1(k) + i * h2(k)) % M2;
        if (T[j] == VUOTO) { T[j] = k; return j; }
    }
    return -1;
}

// Ricerca con doppio hashing: si ferma al primo slot vuoto
int cercaDoppio(int T[], int k) {
    for (int i = 0; i < M2; i++) {
        int j = (h1(k) + i * h2(k)) % M2;
        if (T[j] == k) return j;
        if (T[j] == VUOTO) return -1;
    }
    return -1;
}

int main() {
    Nodo* T[M];
    for (int i = 0; i < M; i++) T[i] = NULL;
    int chiavi[] = {15, 8, 23, 4, 30, 11, 19};
    for (int i = 0; i < 7; i++) inserisciCatena(T, chiavi[i]);
    cout << "23 presente? " << cercaCatena(T, 23) << endl;

    int L[M2], D[M2];
    for (int i = 0; i < M2; i++) { L[i] = VUOTO; D[i] = VUOTO; }
    int k2[] = {14, 25, 36, 7, 18, 3, 29};
    for (int i = 0; i < 7; i++) {
        inserisciLineare(L, k2[i]);
    }
}

```

```

        inserisciDoppio(D, k2[i]);
    }
    for (int i = 0; i < M2; i++) cout << L[i] << " ";
    cout << endl;
    for (int i = 0; i < M2; i++) cout << D[i] << " ";
    cout << endl << "29 nello slot " << cercaDoppio(D, 29) << endl;
    // liberazione della memoria delle liste
    for (int i = 0; i < M; i++) {
        while (T[i] != NULL) {
            Nodo* p = T[i];
            T[i] = T[i]->succ;
            delete p;
        }
    }
    return 0;
}

```

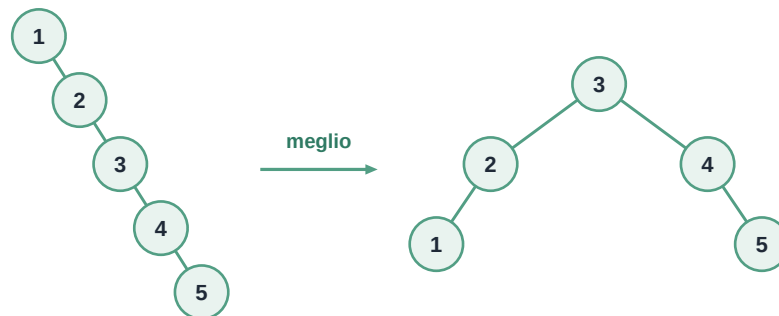
Alberi rosso neri

Alberi binari di ricerca che si mantengono bilanciati da soli grazie a un colore in più per ogni nodo: definizione, rotazioni, inserimento e cancellazione, con tutti i casi spiegati su un esempio completo.

Il problema: alberi di ricerca sbilanciati

Un **albero binario di ricerca** (BST) rispetta questa regola: per ogni nodo x , le chiavi nel sottoalbero sinistro sono $\leq x.chiave$ e quelle nel sottoalbero destro sono $\geq x.chiave$. Ricerca, minimo, massimo, successore, inserimento e cancellazione costano tutti $O(h)$, dove h è l'altezza dell'albero, perché seguono un solo cammino dalla radice verso il basso.

Il guaio è che h dipende dall'ordine degli inserimenti. Se inseriamo le chiavi 1, 2, 3, 4, 5 in quest'ordine, ogni nuova chiave va a destra della precedente e l'albero diventa una lista: $h = n - 1$ e tutte le operazioni costano $O(n)$.



Le stesse cinque chiavi: inserite in ordine crescente formano una lista (a sinistra), ben distribuite formano un albero basso (a destra).

Un albero **bilanciato** garantisce $h = O(\log n)$ qualunque sia l'ordine degli inserimenti. Gli alberi rosso neri sono uno dei modi per ottenerlo: ogni nodo ha un bit in più, il colore, e alcune regole sui colori impediscono a un ramo di diventare troppo più lungo degli altri.

Definizione

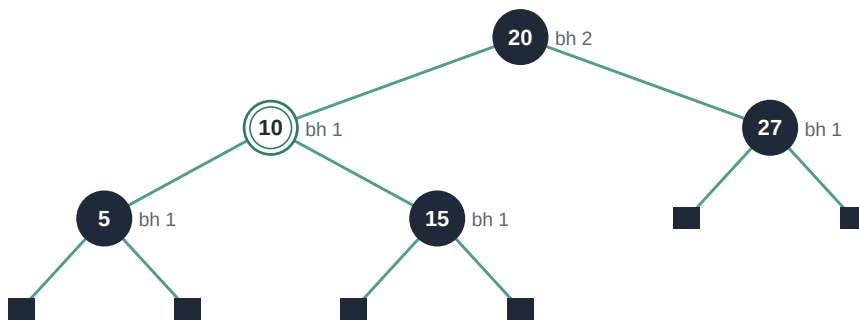
Ogni nodo ha i campi chiave, colore, sinistro, destro e padre. I figli mancanti non sono lasciati vuoti: puntano a una speciale **foglia NIL**, che si considera un nodo esterno di colore nero. In pratica si usa un unico nodo sentinella $T.nil$ condiviso da tutti; nelle figure lo disegniamo come un quadratino, e spesso lo omettiamo.

DEFINIZIONE: ALBERO ROSSO NERO

Un albero binario di ricerca è un **albero rosso nero** se soddisfa le cinque proprietà:

1. ogni nodo è rosso oppure nero;
2. la radice è nera;
3. ogni foglia (NIL) è nera;
4. se un nodo è rosso, entrambi i suoi figli sono neri (non ci sono due rossi consecutivi su un cammino);
5. per ogni nodo, tutti i cammini da quel nodo alle foglie discendenti contengono lo stesso numero di nodi neri.

Il numero di nodi neri incontrati scendendo da un nodo x fino a una foglia, **senza contare x** , si chiama **altezza nera** $bh(x)$; grazie alla proprietà 5 non dipende dal cammino scelto.



Un albero rosso nero con le foglie NIL. Nodi scuri = neri, nodi chiari con doppio bordo = rossi. Accanto a ogni nodo interno la sua altezza nera.

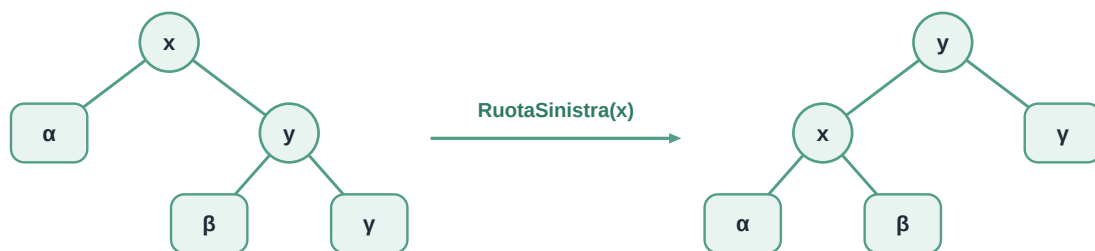
Perché queste regole tengono basso l'albero? Le proprietà 4 e 5 lavorano insieme. Su ogni cammino dalla radice a una foglia il numero di neri è sempre lo stesso, diciamo b ; i rossi non possono essere consecutivi, quindi al più si alternano con i neri. Il cammino più lungo è dunque al più il doppio del più corto. È come una regola di un'azienda: tutti i percorsi di carriera devono passare dallo stesso numero di "promozioni ufficiali" (i nodi neri), e tra due promozioni puoi fare al massimo un incarico "temporaneo" (un nodo rosso). Nessuno può fare una carriera lunghissima rispetto agli altri.

DA SAPERE: ALTEZZA DI UN ALBERO ROSSO NERO

Un albero rosso nero con n nodi interni ha altezza al più $2 \log_2(n + 1)$. Quindi ricerca, minimo, massimo, successore e predecessore costano $O(\log n)$. La dimostrazione (per induzione: un sottoalbero di radice x contiene almeno $2^{bh(x)} - 1$ nodi interni) è nella dispensa di Laboratorio.

Le rotazioni

Inserire o cancellare un nodo come in un normale BST può violare le proprietà. Per ripararle si usano due strumenti: **ricolorare** i nodi e **ruotare**. Una rotazione è una modifica locale dei puntatori che cambia la forma dell'albero ma **conserva la proprietà di albero di ricerca**.



Rotazione sinistra su x ; la rotazione destra su y è l'operazione inversa. α , β e γ sono sottoalberi qualsiasi.

Il modo più semplice di ricordarla: nella rotazione sinistra su x , il figlio destro y "sale" al posto di x , x "scende" a sinistra di y , e il sottoalbero β , che stava tra x e y , cambia padre passando da y a x . L'ordine in cui si leggono gli elementi da sinistra a destra ($\alpha, x, \beta, y, \gamma$) non cambia, ed è per questo che l'albero resta un albero di ricerca.

```

1 LeftRotate(T, x)
2   y = x. destro
3   x. destro = y. sinistro           // beta diventa figlio destro di x
4   if y. sinistro != T.nil then y. sinistro. padre = x
5   y. padre = x. padre               // y prende il posto di x
6   if x. padre = T.nil then T. radice = y
7   else if x = x. padre. sinistro then x. padre. sinistro = y
8   else x. padre. destro = y
9   y. sinistro = x                   // x diventa figlio sinistro di y
10  x. padre = y

```

RightRotate è identica con sinistro e destro scambiati. Entrambe costano **O(1)**: cambiano un numero costante di puntatori.

Inserimento

Si procede in due fasi:

1. si inserisce il nuovo nodo z come in un BST normale, come foglia, e lo si colora di **rosso**;
2. si chiama RBInsertFixup(T, z) per riparare le eventuali violazioni.

Perché rosso? Un nodo rosso non cambia il numero di neri su nessun cammino, quindi la proprietà 5, la più difficile da riparare, resta vera. Possono essere violate solo:

- la proprietà 2, se z è la radice (si risolve colorandola di nero);
- la proprietà 4, se il padre di z è anch'esso rosso: è il vero problema, il "doppio rosso".

```

1 RBInsert(T, z)
2   y = T.nil ; x = T. radice
3   while x != T.nil do           // discesa come in un BST
4       y = x
5       if z. chiave < x. chiave then x = x. sinistro
6       else x = x. destro
7   z. padre = y
8   if y = T.nil then T. radice = z
9   else if z. chiave < y. chiave then y. sinistro = z
10  else y. destro = z
11  z. sinistro = T.nil ; z. destro = T.nil
12  z. colore = ROSSO
13  RBInsertFixup(T, z)

```

I TRE CASI DELLA RIPARAZIONE

Finché il padre di z è rosso, c'è un doppio rosso tra z e suo padre. Il nonno di z esiste ed è nero (il padre è rosso, quindi non è la radice, e prima dell'inserimento valeva la proprietà 4). Tutto dipende dal colore dello **zio** y, il fratello del padre. Descriviamo il caso in cui il padre è figlio **sinistro** del nonno; l'altro è perfettamente simmetrico, scambiando sinistra e destra.

CASO 1, LO ZIO È ROSSO

Padre e zio sono rossi, il nonno è nero. Si **ricolora**: padre e zio diventano neri, il nonno diventa rosso. Il numero di neri su ogni cammino non cambia. Il problema però può essersi spostato due livelli più su (il nonno, ora rosso, potrebbe avere un padre rosso): si pone z = nonno e si ripete il ciclo.

CASO 2, ZIO NERO E Z FIGLIO INTERNO

z è figlio destro di un padre che è figlio sinistro (z sta "dentro", a zigzag). Si pone z = padre e si fa una **rotazione sinistra** su z: il zigzag diventa una linea retta e si ricade nel caso 3.

CASO 3, ZIO NERO E Z FIGLIO ESTERNO

z è figlio sinistro di un padre che è figlio sinistro (in linea retta). Si colora il padre di **nero**, il nonno di **rosso** e si fa una **rotazione destra** sul nonno. Il padre, ora nero, sale al posto del nonno e il ciclo termina.

```
1  RBInsertFixup(T, z)
2      while z.padre.colore = ROSSO do
3          if z.padre = z.padre.padre.sinistro then
4              y = z.padre.padre.destro          // zio
5              if y.colore = ROSSO then          // caso 1
6                  z.padre.colore = NERO
7                  y.colore = NERO
8                  z.padre.padre.colore = ROSSO
9                  z = z.padre.padre
10             else
11                 if z = z.padre.destro then    // caso 2
12                     z = z.padre
13                     LeftRotate(T, z)
14                     z.padre.colore = NERO    // caso 3
15                     z.padre.padre.colore = ROSSO
16                     RightRotate(T, z.padre.padre)
17             else
18                 (come sopra, scambiando "sinistro" e "destro")
19         T.radice.colore = NERO
```

DA RICORDARE: COME RICORDARE I CASI

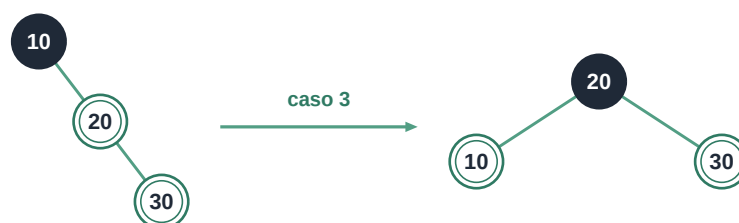
Lo zio rosso si risolve **ricolorando** e spostando il problema verso l'alto. Lo zio nero si risolve **ruotando**: se la forma è a zigzag (caso 2) una prima rotazione la raddrizza, poi la forma in linea retta (caso 3) si chiude con una rotazione sul nonno e uno scambio di colori. Dopo il caso 3 il ciclo finisce sempre: un inserimento fa al più **due** rotazioni.

ESEMPIO COMPLETO DI INSERIMENTO

Inseriamo, partendo da un albero vuoto, le chiavi 10, 20, 30, 15, 25, 5, 12, 27.

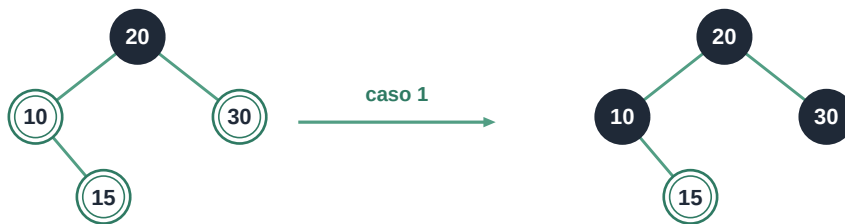
10 diventa la radice: è rossa, ma la riga finale del Fixup la colora di nero. **20** va a destra di 10; il padre è nero e non c'è nulla da fare.

30 va a destra di 20: doppio rosso tra 20 e 30. Il padre 20 è figlio *destro*, quindi siamo nei casi simmetrici. Lo zio (figlio sinistro di 10) è NIL, nero, e 30 è esterno (destro di un destro): è il caso 3 simmetrico. Si colora 20 di nero, 10 di rosso e si fa una rotazione sinistra su 10.



Inserimento di 30: zio nero, figlio esterno, una rotazione.

15 va a sinistra di **20** e poi a destra di **10**: doppio rosso tra **10** e **15**. Lo zio **30** è **rosso**: caso 1. Si colorano **10** e **30** di nero e **20** di rosso; ora $z = 20$, che è la radice, il ciclo termina e la radice torna nera.



Inserimento di 15: zio rosso, si ricolora.

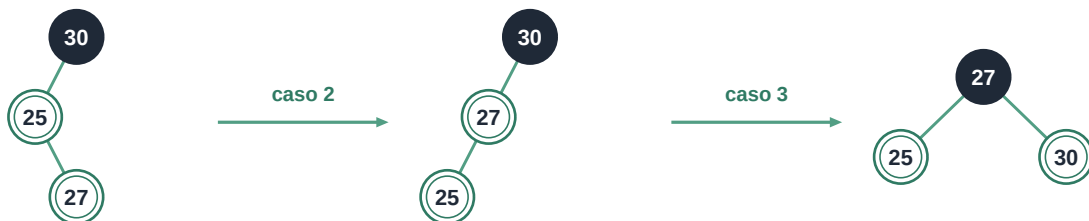
25 va a sinistra di **30**, **5** a sinistra di **10**: in entrambi i casi il padre è nero e non serve nulla.

12 scende **20**, **10**, **15** e diventa figlio sinistro di **15**: doppio rosso tra **15** e **12**. Il padre **15** è figlio destro di **10**, lo zio è **5**, **rosso**: caso 1. Si colorano **15** e **5** di nero e **10** di rosso. Ora $z = 10$, il cui padre **20** è nero: fine.

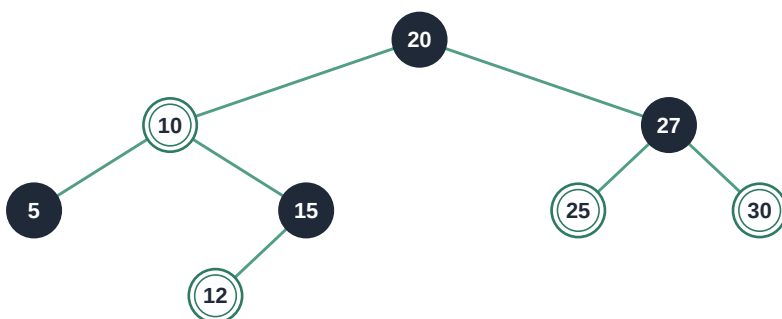


Inserimento di 12: un altro caso 1, che stavolta ricolora un livello intermedio.

27 scende **20**, **30**, **25** e diventa figlio destro di **25**: doppio rosso tra **25** e **27**. Il padre **25** è figlio sinistro di **30**, lo zio (destro di **30**) è NIL, nero, e **27** è figlio **destro** di un figlio sinistro: zigzag, caso 2. Si pone $z = 25$ e si ruota a sinistra su **25**: ora **27** è figlio sinistro di **30** e **25** è figlio sinistro di **27**, in linea retta. Caso 3: **27** diventa nero, **30** rosso, rotazione destra su **30**.



Inserimento di 27, sottoalbero di 30: il zigzag viene raddrizzato e poi chiuso con una seconda rotazione.



Albero finale dopo tutti gli inserimenti.

Cancellazione

La cancellazione è la parte più delicata. Si parte dalla cancellazione da BST, con una procedura di supporto RBTransplant che mette il sottoalbero radicato in v al posto di quello radicato in u.

Si distinguono il nodo z da eliminare e il nodo y che **esce davvero** dall'albero:

- se z ha al più un figlio, $y = z$, e il suo unico figlio (o NIL) x prende il suo posto;
- se z ha due figli, y è il **successore** di z (il minimo del sottoalbero destro, che non ha figlio sinistro): y prende il posto di z e il **colore di z**, e il figlio destro x di y prende il vecchio posto di y.

In entrambi i casi conta il colore originale di y. Se y era **rosso** non succede nulla di grave: nessuna altezza nera cambia e non si creano doppi rossi. Se y era **nero**, tutti i cammini che passavano da y hanno perso un nero, e la proprietà 5 è violata.

```
1  RBTransplant(T, u, v)
2      if u.padre = T.nil then T.radice = v
3      else if u = u.padre.sinistro then u.padre.sinistro = v
4      else u.padre.destro = v
5      v.padre = u.padre
6
7  RBDelete(T, z)
8      y = z ; coloreOriginale = y.colore
9      if z.sinistro = T.nil then
10         x = z.destro ; RBTransplant(T, z, z.destro)
11     else if z.destro = T.nil then
12         x = z.sinistro ; RBTransplant(T, z, z.sinistro)
13     else
14         y = TreeMinimum(z.destro)          // successore
15         coloreOriginale = y.colore
16         x = y.destro
17         if y.padre = z then x.padre = y
18         else
19             RBTransplant(T, y, y.destro)
20             y.destro = z.destro ; y.destro.padre = y
21         RBTransplant(T, z, y)
22         y.sinistro = z.sinistro ; y.sinistro.padre = y
23         y.colore = z.colore
24     if coloreOriginale = NERO then RBDeleteFixup(T, x)
```

L'IDEA DEL "DOPPIO NERO"

Per riparare la proprietà 5 si immagina di "regalare" il nero perso al nodo x che ha preso il posto di y. Se x è rosso, diventa semplicemente nero e il problema è risolto (si dice che x era "rosso e nero"). Se x è già nero, ora porta un **nero in più**: è un "doppio nero", che conta per due. Il Fixup sposta questo nero in eccesso verso l'alto, oppure lo elimina con rotazioni e ricolorazioni, finché x diventa rosso (e lo si colora di nero) o arriva alla radice (dove il nero in più si può semplicemente buttare).

Tutto dipende dal **fratello** w di x. Descriviamo il caso in cui x è figlio sinistro; l'altro è simmetrico.

CASO 1, IL FRATELLO W È ROSSO

Si scambiano i colori di w e del padre (w nero, padre rosso) e si fa una **rotazione sinistra sul padre**. Il nuovo fratello di x è uno dei figli di w, che è nero: ci si è ricondotti a uno dei casi 2, 3 o 4.

CASO 2, W NERO CON ENTRAMBI I FIGLI NERI

Si toglie un nero sia a x sia a w: w diventa **rosso** e il nero in più passa al **padre**, che diventa il nuovo x. Se il padre era rosso, diventa nero e si finisce; altrimenti il ciclo continua un livello più su.

CASO 3, W NERO, FIGLIO SINISTRO DI W ROSSO E FIGLIO DESTRO NERO

Si scambiano i colori di w e del suo figlio sinistro e si fa una **rotazione destra su w**. Ora il nuovo fratello è nero con figlio destro rosso: caso 4.

CASO 4, W NERO CON FIGLIO DESTRO ROSSO

w prende il colore del padre, il padre diventa nero, il figlio destro di w diventa nero, e si fa una **rotazione sinistra sul padre**. Il nero in più viene assorbito; si pone x = radice e il ciclo termina.

```
1  RBDeleteFixup(T, x)
2      while x != T.radice and x.colore = NERO do
3          if x = x.padre.sinistro then
4              w = x.padre.destro                // fratello
5              if w.colore = ROSSO then          // caso 1
6                  w.colore = NERO
7                  x.padre.colore = ROSSO
8                  LeftRotate(T, x.padre)
9                  w = x.padre.destro
10             if w.sinistro.colore = NERO and w.destro.colore = NERO then
11                 w.colore = ROSSO              // caso 2
12                 x = x.padre
13             else
14                 if w.destro.colore = NERO then // caso 3
15                     w.sinistro.colore = NERO
16                     w.colore = ROSSO
17                     RightRotate(T, w)
18                     w = x.padre.destro
19                     w.colore = x.padre.colore // caso 4
20                     x.padre.colore = NERO
21                     w.destro.colore = NERO
22                     LeftRotate(T, x.padre)
23                     x = T.radice
24             else
25                 (come sopra, scambiando "sinistro" e "destro")
26             x.colore = NERO
```

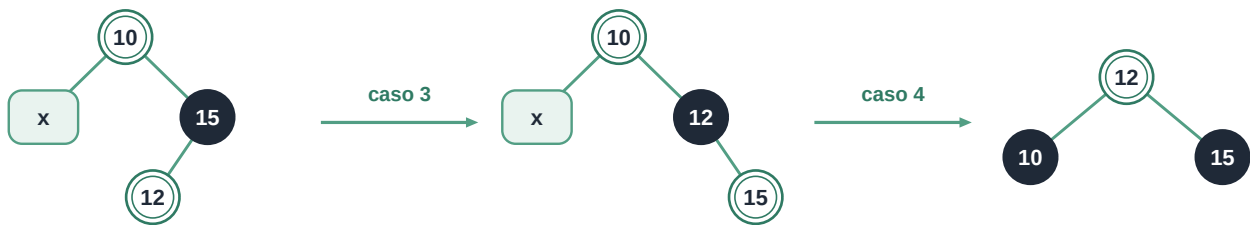
DA RICORDARE: COME RICORDARE I CASI

Il caso 1 (fratello rosso) è solo una preparazione: trasforma il fratello in un nero. Il caso 2 (fratello nero "povero", con figli neri) sposta il problema verso l'alto. I casi 3 e 4 (fratello nero con almeno un figlio rosso) lo risolvono definitivamente con al più due rotazioni. In totale una cancellazione fa al più **tre rotazioni**.

ESEMPIO COMPLETO DI CANCELLAZIONE

Partiamo dall'albero finale dell'esempio precedente e cancelliamo nell'ordine 5, 27, 30, 25.

Cancellare 5. Il nodo 5 è nero e non ha figli: y = 5, x = NIL al suo posto, colore originale nero, quindi serve il Fixup. x è figlio sinistro di 10; il fratello w = 15 è nero, con figlio sinistro 12 rosso e figlio destro NIL nero: **caso 3**. Si colora 12 di nero, 15 di rosso e si ruota a destra su 15. Il nuovo fratello è 12, nero, con figlio destro 15 rosso: **caso 4**. 12 prende il colore del padre 10 (rosso), 10 diventa nero, 15 diventa nero, rotazione sinistra su 10.



Cancellazione di 5, sottoalbero di 10: il quadrato x indica la posizione con il nero in più.

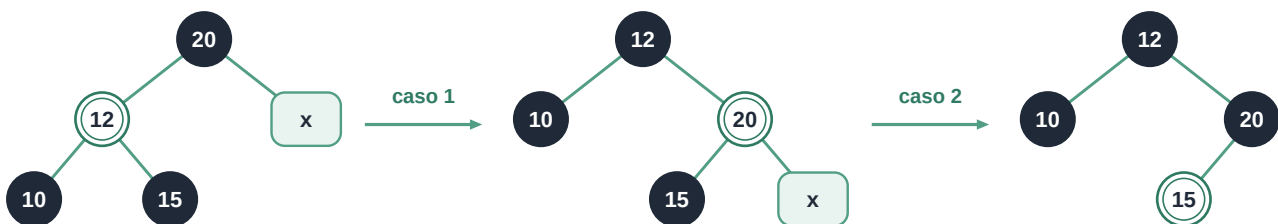
Cancellare 27. Il nodo 27 ha due figli: il successore è $y = 30$, figlio destro diretto di 27. Il colore originale di y è **rosso**: 30 prende il posto di 27 e il suo colore (nero), 25 diventa suo figlio sinistro. Nessun Fixup.

Cancellare 30. Ha solo il figlio sinistro 25, che prende il suo posto: $x = 25$. Il colore originale era nero, ma x è rosso: il ciclo non parte e la riga finale colora 25 di nero. È il caso "rosso e nero".



A sinistra l'albero dopo aver cancellato 27, a destra dopo aver cancellato 30.

Cancellare 25. È nero e senza figli: $x = \text{NIL}$, figlio **destro** di 20, quindi siamo nei casi simmetrici e il fratello è $w = 12$, **rosso**: caso 1. Si colora 12 di nero, 20 di rosso e si ruota a **destra** su 20. Il nuovo fratello di x è 15, nero con figli NIL neri: **caso 2**. 15 diventa rosso e il nero in più sale al padre 20, che è rosso: lo si colora di nero e si termina.



Cancellazione di 25: il fratello rosso viene trasformato (caso 1), poi il nero in più sale e viene assorbito dal padre rosso (caso 2).

DA SAPERE: COSTI

Ricerca, inserimento e cancellazione costano $O(\log n)$ nel caso peggiore. Il Fixup dell'inserimento può risalire solo con il caso 1 (due livelli alla volta) e fa al più 2 rotazioni; quello della cancellazione può risalire solo con il caso 2 e fa al più 3 rotazioni. Le rotazioni costano $O(1)$ ciascuna.

Implementazione in C++

L'implementazione segue esattamente lo pseudocodice, con un unico nodo sentinella NIL nero condiviso da tutte le foglie. Il main ripete l'esempio del capitolo e stampa l'albero in preordine (chiave seguita da R o N) dopo ogni operazione: l'uscita coincide con le figure.

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

enum Colore { ROSSO, NERO };

struct Nodo {
    int chiave;
    Colore colore;
```

```

    Nodo* sinistro;
    Nodo* destro;
    Nodo* padre;
};

Nodo* NIL; // sentinella unica: e' nera
Nodo* radice;

void inizializza() {
    NIL = new Nodo;
    NIL->colore = NERO;
    NIL->sinistro = NIL->destro = NIL->padre = NIL;
    radice = NIL;
}

// Rotazione sinistra attorno a x (x->destro deve esistere)
void ruotaSinistra(Nodo* x) {
    Nodo* y = x->destro;
    x->destro = y->sinistro; // il sottoalbero beta passa a x
    if (y->sinistro != NIL) y->sinistro->padre = x;
    y->padre = x->padre; // y prende il posto di x
    if (x->padre == NIL) radice = y;
    else if (x == x->padre->sinistro) x->padre->sinistro = y;
    else x->padre->destro = y;
    y->sinistro = x; // x diventa figlio sinistro di y
    x->padre = y;
}

// Rotazione destra: simmetrica della precedente
void ruotaDestra(Nodo* y) {
    Nodo* x = y->sinistro;
    y->sinistro = x->destro;
    if (x->destro != NIL) x->destro->padre = y;
    x->padre = y->padre;
    if (y->padre == NIL) radice = x;
    else if (y == y->padre->sinistro) y->padre->sinistro = x;
    else y->padre->destro = x;
    x->destro = y;
    y->padre = x;
}

// Ripristina le proprieta' dopo l'inserimento del nodo rosso z
void inserisciFixup(Nodo* z) {
    while (z->padre->colore == ROSSO) {
        if (z->padre == z->padre->padre->sinistro) {
            Nodo* zio = z->padre->padre->destro;
            if (zio->colore == ROSSO) { // caso 1
                z->padre->colore = NERO;
                zio->colore = NERO;
                z->padre->padre->colore = ROSSO;
                z = z->padre->padre;
            } else {
                if (z == z->padre->destro) { // caso 2
                    z = z->padre;
                    ruotaSinistra(z);
                }
                z->padre->colore = NERO; // caso 3
                z->padre->padre->colore = ROSSO;
                ruotaDestra(z->padre->padre);
            }
        } else { // casi simmetrici
            Nodo* zio = z->padre->padre->sinistro;

```

```

        if (zio->colore == ROSSO) {
            z->padre->colore = NERO;
            zio->colore = NERO;
            z->padre->padre->colore = ROSSO;
            z = z->padre->padre;
        } else {
            if (z == z->padre->sinistro) {
                z = z->padre;
                ruotaDestra(z);
            }
            z->padre->colore = NERO;
            z->padre->padre->colore = ROSSO;
            ruotaSinistra(z->padre->padre);
        }
    }
}
radice->colore = NERO;
}

// Inserimento come in un albero di ricerca, poi colora di rosso e ripara
void inserisci(int k) {
    Nodo* z = new Nodo;
    z->chiave = k;
    Nodo* y = NIL;                // futuro padre di z
    Nodo* x = radice;
    while (x != NIL) {
        y = x;
        if (k < x->chiave) x = x->sinistro;
        else x = x->destro;
    }
    z->padre = y;
    if (y == NIL) radice = z;
    else if (k < y->chiave) y->sinistro = z;
    else y->destro = z;
    z->sinistro = z->destro = NIL;
    z->colore = ROSSO;
    inserisciFixup(z);
}

// Sostituisce il sottoalbero di radice u con quello di radice v
void trapianta(Nodo* u, Nodo* v) {
    if (u->padre == NIL) radice = v;
    else if (u == u->padre->sinistro) u->padre->sinistro = v;
    else u->padre->destro = v;
    v->padre = u->padre;          // lecito anche se v e' NIL
}

Nodo* minimo(Nodo* x) {
    while (x->sinistro != NIL) x = x->sinistro;
    return x;
}

// Elimina il "doppio nero" portato da x
void cancellaFixup(Nodo* x) {
    while (x != radice && x->colore == NERO) {
        if (x == x->padre->sinistro) {
            Nodo* w = x->padre->destro;          // fratello
            if (w->colore == ROSSO) {            // caso 1
                w->colore = NERO;
                x->padre->colore = ROSSO;
                ruotaSinistra(x->padre);
            }
        }
    }
}

```

```

        w = x->padre->destro;
    }
    if (w->sinistro->colore == NERO && w->destro->colore == NERO) {
        w->colore = ROSSO;          // caso 2
        x = x->padre;
    } else {
        if (w->destro->colore == NERO) {          // caso 3
            w->sinistro->colore = NERO;
            w->colore = ROSSO;
            ruotaDestra(w);
            w = x->padre->destro;
        }
        w->colore = x->padre->colore;          // caso 4
        x->padre->colore = NERO;
        w->destro->colore = NERO;
        ruotaSinistra(x->padre);
        x = radice;
    }
} else {          // casi simmetrici
    Nodo* w = x->padre->sinistro;
    if (w->colore == ROSSO) {
        w->colore = NERO;
        x->padre->colore = ROSSO;
        ruotaDestra(x->padre);
        w = x->padre->sinistro;
    }
    if (w->destro->colore == NERO && w->sinistro->colore == NERO) {
        w->colore = ROSSO;
        x = x->padre;
    } else {
        if (w->sinistro->colore == NERO) {
            w->destro->colore = NERO;
            w->colore = ROSSO;
            ruotaSinistra(w);
            w = x->padre->sinistro;
        }
        w->colore = x->padre->colore;
        x->padre->colore = NERO;
        w->sinistro->colore = NERO;
        ruotaDestra(x->padre);
        x = radice;
    }
}
}
x->colore = NERO;
}

// Elimina il nodo z
void cancella(Nodo* z) {
    Nodo* y = z;          // nodo che esce davvero dall'albero
    Nodo* x;              // nodo che prende il posto di y
    Colore coloreOriginale = y->colore;
    if (z->sinistro == NIL) {
        x = z->destro;
        trapianta(z, z->destro);
    } else if (z->destro == NIL) {
        x = z->sinistro;
        trapianta(z, z->sinistro);
    } else {              // due figli: si usa il successore
        y = minimo(z->destro);

```

```

        coloreOriginale = y->colore;
        x = y->destro;
        if (y->padre == z) x->padre = y;
        else {
            trapianta(y, y->destro);
            y->destro = z->destro;
            y->destro->padre = y;
        }
        trapianta(z, y);
        y->sinistro = z->sinistro;
        y->sinistro->padre = y;
        y->colore = z->colore;
    }
    delete z;
    if (coloreOriginale == NERO) cancellaFixup(x);
}

Nodo* cerca(int k) {
    Nodo* x = radice;
    while (x != NIL && x->chiave != k) {
        if (k < x->chiave) x = x->sinistro;
        else x = x->destro;
    }
    return x;
}

// Stampa in preordine: chiave seguita da R o N
void stampa(Nodo* x) {
    if (x == NIL) return;
    cout << x->chiave << (x->colore == ROSSO ? "R " : "N ");
    stampa(x->sinistro);
    stampa(x->destro);
}

int main() {
    inizializza();
    int chiavi[] = {10, 20, 30, 15, 25, 5, 12, 27};
    for (int i = 0; i < 8; i++) {
        inserisci(chiavi[i]);
        stampa(radice);
        cout << endl;
    }
    int via[] = {5, 27, 30, 25};
    for (int i = 0; i < 4; i++) {
        cancella(cerca(via[i]));
        cout << "dopo " << via[i] << ": ";
        stampa(radice);
        cout << endl;
    }
    return 0;
}

```

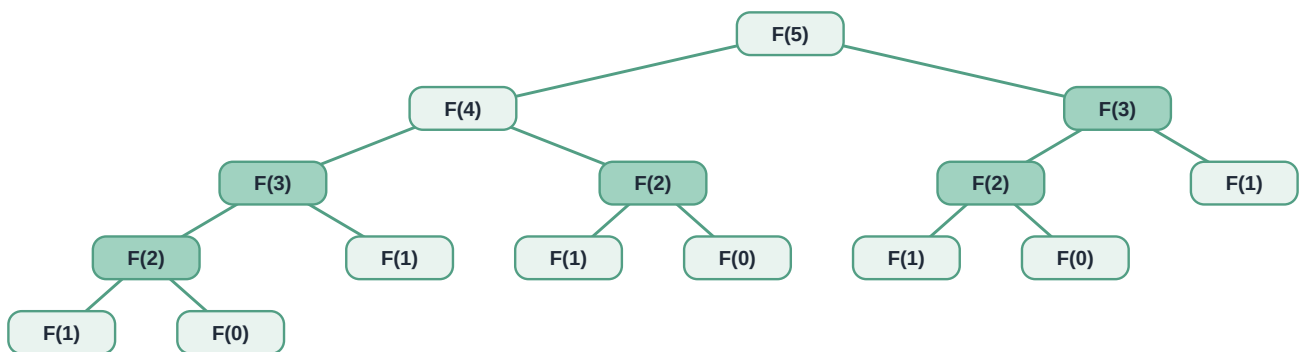
Programmazione dinamica

Una tecnica per i problemi di ottimizzazione in cui gli stessi sottoproblemi si ripresentano molte volte: si risolve ciascuno una sola volta, si salva il risultato in una tabella e si ricostruisce la soluzione ottima. Casi di studio: catena di montaggio, moltiplicazione di matrici, zaino e resto.

L'idea: non risolvere due volte la stessa cosa

Il divide et impera spezza un problema in sottoproblemi **indipendenti**. Ci sono però problemi in cui i sottoproblemi si **sovrappongono**: risolvendo il problema grande, lo stesso sottoproblema viene richiesto molte volte. Una ricorsione ingenua lo ricalcola ogni volta, e il costo esplode.

L'esempio più semplice sono i numeri di Fibonacci: $F(0) = 0$, $F(1) = 1$ e $F(n) = F(n-1) + F(n-2)$. La funzione ricorsiva diretta fa due chiamate per ogni livello.



Chiamate ricorsive per F(5): F(3) viene calcolato due volte, F(2) tre volte. Per n grande il numero di chiamate cresce in modo esponenziale.

La soluzione è banale a dirsi: **ricordare** i risultati già calcolati. Basta un array in cui $F[i]$ viene riempito una volta sola, dal più piccolo al più grande, e il tempo diventa lineare. È come fare i compiti di matematica tenendo un quaderno dei risultati intermedi: se un conto ti serve di nuovo lo copi, non lo rifai.

```

1 Fibonacci(n)
2   F[0] = 0 ; F[1] = 1
3   for i = 2 to n do
4       F[i] = F[i-1] + F[i-2]    // ogni valore calcolato una sola volta
5   return F[n]
  
```

DEFINIZIONE: PROGRAMMAZIONE DINAMICA

La **programmazione dinamica** risolve un problema combinando le soluzioni di sottoproblemi che si sovrappongono, calcolando ciascun sottoproblema **una sola volta** e salvandone il risultato in una tabella. Si applica tipicamente ai **problemi di ottimizzazione**. Il nome è storico: "programmazione" significa qui "uso di una tabella", non scrittura di codice.

LE DUE PROPRIETÀ RICHIESTE

1. **Sottostruttura ottima:** una soluzione ottima del problema contiene al suo interno soluzioni ottime dei sottoproblemi. Esempio intuitivo: se il percorso più breve da Catania a Milano passa per Roma, allora il suo tratto da Catania a Roma è il percorso più breve tra Catania e Roma; se non lo fosse, potremmo sostituirlo con uno più corto e accorciare tutto il percorso.
2. **Sottoproblemi ripetuti** (sovrapposti): lo spazio dei sottoproblemi distinti è "piccolo" (tipicamente polinomiale), anche se una ricorsione ingenua li visita un numero esponenziale di volte.

IL METODO IN QUATTRO PASSI

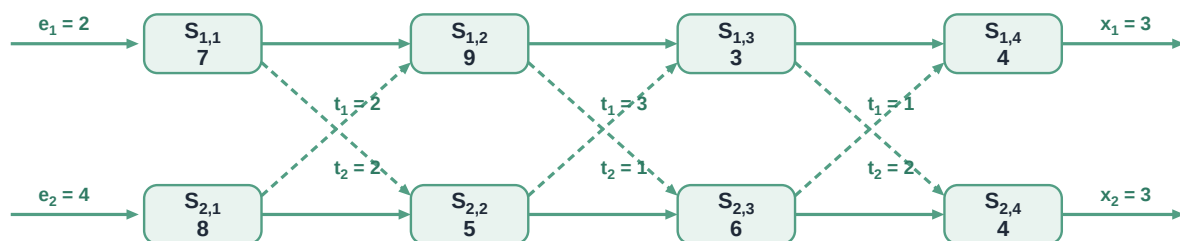
DA RICORDARE: I QUATTRO PASSI

1. Caratterizzare la **struttura** di una soluzione ottima (sottostruttura ottima).
2. Definire **ricorsivamente** il valore di una soluzione ottima, con un'equazione.
3. Calcolare il valore ottimo in modo **bottom up**, riempiendo una tabella dai sottoproblemi più piccoli ai più grandi.
4. **Ricostruire** una soluzione ottima usando informazioni salvate durante il passo 3.

Esiste anche una versione **top down**, detta **memoizzazione**: si scrive la ricorsione naturale, ma prima di calcolare un sottoproblema si controlla se il risultato è già nella tabella. Ha la stessa complessità asintotica e risolve solo i sottoproblemi davvero necessari; la versione bottom up evita il costo delle chiamate ricorsive.

Catena di montaggio

Una fabbrica di automobili ha **due linee** di montaggio, ciascuna con n stazioni. La stazione j della linea i si chiama $S_{i,j}$ e richiede un tempo $a_{i,j}$; le stazioni $S_{1,j}$ e $S_{2,j}$ fanno lo stesso lavoro. Entrare nella linea i costa e_i , uscirne costa x_i . Un telaio può restare sulla stessa linea gratis, oppure, dopo la stazione j , passare all'altra linea pagando un tempo di trasferimento $t_{i,j}$. Vogliamo scegliere le stazioni che minimizzano il tempo totale.



Istanza con $n = 4$: dentro ogni stazione il tempo di lavorazione; le frecce tratteggiate sono i trasferimenti, con il costo scritto accanto.

Provare tutte le combinazioni significa considerare 2^n percorsi: impraticabile. Applichiamo i quattro passi.

Passo 1, sottostruttura ottima. Consideriamo il modo più veloce per uscire dalla stazione $S_{1,j}$. Il telaio arriva o da $S_{1,j-1}$ (stessa linea) o da $S_{2,j-1}$ (con trasferimento). In entrambi i casi, il percorso fino alla stazione precedente deve essere a sua volta il più veloce possibile per arrivare lì.

Passo 2, ricorrenza. Sia $f_i[j]$ il tempo minimo per completare la stazione $S_{i,j}$ partendo dall'ingresso. Allora

$$f_1[1] = e_1 + a_{1,1} \quad \text{e} \quad f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$$

e simmetricamente per f_2 . Il tempo ottimo totale è $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$.

Passo 3, calcolo. Si riempiono le due righe della tabella da sinistra a destra. Insieme a f si salva $l[j]$, la linea da cui conviene arrivare in $S_{i,j}$.

J	1	2	3	4
$f_1[j]$	$2 + 7 = 9$	$\min(9 + 9, 12 + 2 + 9) = 18$	$\min(18 + 3, 16 + 1 + 3) = 20$	$\min(20 + 4, 22 + 2 + 4) = 24$
$l_1[j]$	.	1	2	1
$f_2[j]$	$4 + 8 = 12$	$\min(12 + 5, 9 + 2 + 5) = 16$	$\min(16 + 6, 18 + 3 + 6) = 22$	$\min(22 + 4, 20 + 1 + 4) = 25$
$l_2[j]$	.	1	2	1

Infine $f^* = \min(24 + 3, 25 + 3) = 27$, con uscita dalla linea $l^* = 1$.

Passo 4, ricostruzione. Si parte dalla fine e si segue l all'indietro: stazione 4 sulla linea 1; $l_1[4] = 1$, quindi stazione 3 sulla linea 1; $l_1[3] = 2$, quindi stazione 2 sulla linea 2; $l_2[2] = 1$, quindi stazione 1 sulla linea 1. Percorso: $S_{1,1}, S_{2,2}, S_{1,3}, S_{1,4}$. Verifica: $2 + 7 + 2 + 5 + 1 + 3 + 4 + 3 = 27$.

```

1 FastestWay(a, t, e, x, n)
2   f1[1] = e1 + a[1][1] ; f2[1] = e2 + a[2][1]
3   for j = 2 to n do
4     if f1[j-1] + a[1][j] <= f2[j-1] + t[2][j-1] + a[1][j] then
5       f1[j] = f1[j-1] + a[1][j] ; l1[j] = 1
6     else
7       f1[j] = f2[j-1] + t[2][j-1] + a[1][j] ; l1[j] = 2
8     (analogo per f2[j] e l2[j])
9   if f1[n] + x1 <= f2[n] + x2 then f* = f1[n] + x1 ; l* = 1
10  else f* = f2[n] + x2 ; l* = 2

```

DA SAPERE: CATENA DI MONTAGGIO

Tempo $\Theta(n)$ invece di $\Theta(2^n)$: ogni casella della tabella si calcola in tempo costante guardando solo la colonna precedente.

Moltiplicazione di una sequenza di matrici

Moltiplicare una matrice $p \times q$ per una $q \times r$ costa $p \cdot q \cdot r$ moltiplicazioni scalari e produce una matrice $p \times r$. Il prodotto di matrici è associativo, quindi una sequenza $A_1 A_2 \dots A_n$ si può calcolare con **parentesizzazioni** diverse, che danno lo stesso risultato ma con costi molto diversi.

Esempio: A_1 è 10×30 , A_2 è 30×5 , A_3 è 5×60 , A_4 è 60×10 . Le dimensioni si scrivono in un unico vettore $p = (10, 30, 5, 60, 10)$: la matrice A_i ha dimensioni $p_{i-1} \times p_i$.

- $((A_1 A_2) A_3) A_4$ costa $1500 + 3000 + 6000 = 10\,500$;
- $A_1 (A_2 (A_3 A_4))$ costa $3000 + 1500 + 3000 = 7500$;
- $(A_1 A_2) (A_3 A_4)$ costa $1500 + 3000 + 500 = 5000$.

Il numero di parentesizzazioni possibili cresce in modo esponenziale con n , quindi provarle tutte non è un'opzione.

Sottostruttura ottima. In una parentesizzazione ottima di $A_1 \dots A_j$ c'è un'ultima moltiplicazione, che divide la sequenza in $A_1 \dots A_k$ e $A_{k+1} \dots A_j$. Le due parti devono essere a loro volta parentesizzate in modo ottimo.

Ricorrenza. Sia $m[i, j]$ il costo minimo per calcolare $A_i \dots A_j$. Si provano tutti i punti di divisione k :

$$m[i, i] = 0 \quad m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_{i-1} \cdot p_k \cdot p_j)$$

Il terzo addendo è il costo dell'ultima moltiplicazione, tra una matrice $p_{i-1} \times p_k$ e una $p_k \times p_j$. In $s[i, j]$ si salva il k migliore.

Ordine di calcolo. $m[i, j]$ dipende da sottosequenze più corte, quindi si riempie la tabella per **lunghezza** crescente $l = j - i + 1$: prima tutte le coppie, poi le terne, e così via.

LUNGHEZZA	CALCOLO	RISULTATO
2	$m[1,2] = 10 \cdot 30 \cdot 5$; $m[2,3] = 30 \cdot 5 \cdot 60$; $m[3,4] = 5 \cdot 60 \cdot 10$	1500, 9000, 3000
3	$m[1,3] = \min(0 + 9000 + 18\,000, 1500 + 0 + 3000)$	4500 ($k = 2$)
3	$m[2,4] = \min(0 + 3000 + 1500, 9000 + 0 + 18\,000)$	4500 ($k = 2$)
4	$m[1,4] = \min(0 + 4500 + 3000, 1500 + 3000 + 500, 4500 + 0 + 6000)$	5000 ($k = 2$)

Il costo ottimo è $m[1, 4] = 5000$ con $s[1, 4] = 2$: l'ultima moltiplicazione separa A_1A_2 da A_3A_4 , cioè $(A_1A_2)(A_3A_4)$.

```

1  MatrixChainOrder(p, n)
2      for i = 1 to n do m[i][i] = 0
3      for l = 2 to n do                // lunghezza della sottosequenza
4          for i = 1 to n - l + 1 do
5              j = i + l - 1
6              m[i][j] = infinito
7              for k = i to j - 1 do
8                  q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
9                  if q < m[i][j] then
10                     m[i][j] = q ; s[i][j] = k
11      return m e s
12
13 PrintOptimalParens(s, i, j)
14     if i = j then stampa "A" i
15     else
16         stampa "("
17         PrintOptimalParens(s, i, s[i][j])
18         PrintOptimalParens(s, s[i][j] + 1, j)
19     stampa ")"

```

DA SAPERE: MOLTIPLICAZIONE DI MATRICI

I sottoproblemi distinti sono le coppie (i, j) , cioè $\Theta(n^2)$, e ciascuno prova $O(n)$ valori di k : tempo $\Theta(n^3)$, spazio $\Theta(n^2)$. L'algoritmo non moltiplica le matrici: trova solo l'ordine migliore in cui farlo.

Problema dello zaino 0/1

Un ladro ha uno zaino che regge al massimo un peso W e davanti a sé n oggetti; l'oggetto i ha peso w_i e valore v_i (interi). Ogni oggetto si prende **tutto o niente** (da qui "0/1"). Si vuole massimizzare il valore totale senza superare W .

Sottostruttura ottima. Guardiamo l'ultimo oggetto i . Le possibilità sono due: o non lo prendiamo, e allora resta il problema con i primi $i - 1$ oggetti e la stessa capacità; oppure lo prendiamo (se ci sta), guadagniamo v_i e resta il problema con i primi $i - 1$ oggetti e capacità ridotta di w_i .

Ricorrenza. Sia $K[i, c]$ il valore massimo ottenibile con i primi i oggetti e capacità c :

$$K[0, c] = 0 \quad K[i, c] = K[i-1, c] \quad \text{se } w_i > c$$

$$K[i, c] = \max(K[i-1, c], K[i-1, c - w_i] + v_i) \quad \text{altrimenti}$$

ESEMPIO

Capacità $W = 7$ e quattro oggetti con (peso, valore): 1 = (1, 1), 2 = (3, 4), 3 = (4, 5), 4 = (5, 7). Ogni riga si calcola dalla precedente.

OGGETTI \ C	0	1	2	3	4	5	6	7
nessuno	0	0	0	0	0	0	0	0
1 (1, 1)	0	1	1	1	1	1	1	1
2 (3, 4)	0	1	1	4	5	5	5	5
3 (4, 5)	0	1	1	4	5	6	6	9
4 (5, 7)	0	1	1	4	5	7	8	9

Per esempio $K[3, 7] = \max(K[2, 7], K[2, 3] + 5) = \max(5, 4 + 5) = 9$: conviene prendere l'oggetto 3 e riempire i 3 chili rimasti nel modo migliore con i primi due oggetti, cioè con l'oggetto 2.

Ricostruzione. Si parte da $K[4, 7] = 9$. È uguale a $K[3, 7]$, quindi l'oggetto 4 non è preso. $K[3, 7] = 9$ è diverso da $K[2, 7] = 5$, quindi l'oggetto 3 è preso e la capacità scende a 3. $K[2, 3] = 4$ è diverso da $K[1, 3] = 1$, quindi l'oggetto 2 è preso e la capacità scende a 0. Soluzione: oggetti 2 e 3, peso 7, valore 9.

DA SAPERE: ZAINO 0/1

Tempo e spazio $\Theta(n \cdot W)$. Attenzione: è polinomiale in n e nel **valore** W , non nella lunghezza dell'ingresso (W si scrive con $\log W$ cifre). Si dice **pseudopolinomiale**. Nel prossimo capitolo vedremo che la versione frazionaria si risolve con una strategia greedy, mentre quella 0/1 no.

Problema del resto

Dato un insieme di tagli di monete $d_1, \dots, d_k$ (con $d_1 = 1$, così il resto è sempre possibile) e un importo R , si vuole dare il resto con il **minimo numero di monete**, avendo a disposizione quante monete si vogliono di ogni taglio.

Ricorrenza. Sia $C[v]$ il minimo numero di monete per formare l'importo v . Se la soluzione ottima per v usa una moneta d_i , il resto $v - d_i$ deve essere formato in modo ottimo. Si prova ogni taglio:

$$C[0] = 0 \quad C[v] = 1 + \min_{i: d_i \leq v} C[v - d_i]$$

ESEMPIO

Tagli {1, 4, 6} e importo $R = 8$. In $S[v]$ si salva la moneta scelta.

v	0	1	2	3	4	5	6	7	8
$C[v]$	0	1	2	3	1	2	1	2	2
$S[v]$	.	1	1	1	4	1	6	1	4

Per esempio $C[8] = 1 + \min(C[7], C[4], C[2]) = 1 + \min(2, 1, 2) = 2$, con la moneta 4; poi $C[4] = 1$ con la moneta 4. Il resto ottimo è $4 + 4$. La strategia "prendo sempre la moneta più grande che ci sta" darebbe $6 + 1 + 1$, cioè 3 monete: con questi tagli il greedy sbaglia, la programmazione dinamica no.

DA SAPERE: RESTO

Tempo $\Theta(k \cdot R)$, spazio $\Theta(R)$. La ricostruzione segue S all'indietro: da R si toglie $S[R]$, e così via fino a 0.

Implementazione in C++

Tutte le tabelle sono array di dimensione massima fissa. Gli indici sono tenuti "da 1" come nello pseudocodice, lasciando inutilizzata la posizione 0 dove serve: così le formule coincidono con quelle del capitolo.

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

const int MAXN = 20;
const int INF = 1000000000;          // "infinito"

// Catena di montaggio con stazioni 1..n; restituisce il tempo minimo
int catenaMontaggio(int a[3][MAXN], int t[3][MAXN], int e[3], int x[3], int n) {
    int f[3][MAXN], l[3][MAXN];
    f[1][1] = e[1] + a[1][1];
    f[2][1] = e[2] + a[2][1];
    for (int j = 2; j <= n; j++) {
        // linea 1: resto sulla 1 oppure arrivo dalla 2
        if (f[1][j - 1] + a[1][j] <= f[2][j - 1] + t[2][j - 1] + a[1][j]) {
            f[1][j] = f[1][j - 1] + a[1][j];    l[1][j] = 1;
        } else {
            f[1][j] = f[2][j - 1] + t[2][j - 1] + a[1][j];    l[1][j] = 2;
        }
        // linea 2: resto sulla 2 oppure arrivo dalla 1
        if (f[2][j - 1] + a[2][j] <= f[1][j - 1] + t[1][j - 1] + a[2][j]) {
            f[2][j] = f[2][j - 1] + a[2][j];    l[2][j] = 2;
        } else {
            f[2][j] = f[1][j - 1] + t[1][j - 1] + a[2][j];    l[2][j] = 1;
        }
    }
    int linea, fstar;
    if (f[1][n] + x[1] <= f[2][n] + x[2]) { fstar = f[1][n] + x[1]; linea = 1; }
    else { fstar = f[2][n] + x[2]; linea = 2; }
    // ricostruzione all'indietro (stampa dall'ultima stazione alla prima)
    cout << "percorso al contrario: ";
    for (int j = n; j >= 1; j--) {
        cout << "S" << linea << ", " << j << " ";
        if (j > 1) linea = l[linea][j];
    }
    cout << endl;
    return fstar;
}

int m[MAXN][MAXN], s[MAXN][MAXN];    // tabelle per le matrici

// p[0..n] sono le dimensioni: la matrice i e' p[i-1] x p[i]
int ordineMatrici(int p[], int n) {
    for (int i = 1; i <= n; i++) m[i][i] = 0;
    for (int l = 2; l <= n; l++) {    // lunghezza della sottosequenza
        for (int i = 1; i <= n - l + 1; i++) {
            int j = i + l - 1;
            m[i][j] = INF;
            for (int k = i; k < j; k++) {    // punto di divisione
                int q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j]) { m[i][j] = q; s[i][j] = k; }
            }
        }
    }
    return m[1][n];
}
```

```

void stampaParentesi(int i, int j) {
    if (i == j) cout << "A" << i;
    else {
        cout << "(";
        stampaParentesi(i, s[i][j]);
        stampaParentesi(s[i][j] + 1, j);
        cout << ")";
    }
}

// Zaino 0/1: oggetti 1..n con peso w[i] e valore v[i], capacita' W
int zaino(int w[], int v[], int n, int W) {
    int K[MAXN][MAXN];
    for (int c = 0; c <= W; c++) K[0][c] = 0;
    for (int i = 1; i <= n; i++) {
        for (int c = 0; c <= W; c++) {
            K[i][c] = K[i - 1][c]; // non prendo l'oggetto i
            if (w[i] <= c && K[i - 1][c - w[i]] + v[i] > K[i][c])
                K[i][c] = K[i - 1][c - w[i]] + v[i]; // prendo l'oggetto i
        }
    }
    // ricostruzione
    cout << "oggetti presi: ";
    int c = W;
    for (int i = n; i >= 1; i--) {
        if (K[i][c] != K[i - 1][c]) {
            cout << i << " ";
            c = c - w[i];
        }
    }
    cout << endl;
    return K[n][W];
}

// Resto con il minimo numero di monete; tagli d[0..k-1], importo R
int resto(int d[], int k, int R) {
    int C[100], S[100];
    C[0] = 0;
    for (int v = 1; v <= R; v++) {
        C[v] = INF;
        for (int i = 0; i < k; i++) {
            if (d[i] <= v && C[v - d[i]] + 1 < C[v]) {
                C[v] = C[v - d[i]] + 1;
                S[v] = d[i]; // moneta scelta per l'importo v
            }
        }
    }
    cout << "monete: ";
    for (int v = R; v > 0; v = v - S[v]) cout << S[v] << " ";
    cout << endl;
    return C[R];
}

int main() {
    int a[3][MAXN], t[3][MAXN];
    int a1[] = {0, 7, 9, 3, 4}, a2[] = {0, 8, 5, 6, 4};
    int t1[] = {0, 2, 3, 1}, t2[] = {0, 2, 1, 2};
    for (int j = 1; j <= 4; j++) { a[1][j] = a1[j]; a[2][j] = a2[j]; }
    for (int j = 1; j <= 3; j++) { t[1][j] = t1[j]; t[2][j] = t2[j]; }
    int e[3] = {0, 2, 4}, x[3] = {0, 3, 3};
    int tempo = catenaMontaggio(a, t, e, x, 4);
}

```

```
cout << "tempo minimo: " << tempo << endl;

int p[] = {10, 30, 5, 60, 10};
cout << "costo minimo: " << ordineMatrici(p, 4) << " con ";
stampaParentesi(1, 4);
cout << endl;

int w[] = {0, 1, 3, 4, 5}, v[] = {0, 1, 4, 5, 7};
int valore = zaino(w, v, 4, 7);
cout << "valore massimo: " << valore << endl;

int d[] = {1, 4, 6};
int monete = resto(d, 3, 8);
cout << "numero di monete: " << monete << endl;
return 0;
}
```

Algoritmi greedy

La strategia golosa: a ogni passo si fa la scelta che sembra migliore in quel momento, senza mai tornare indietro. Quando funziona è semplicissima e velocissima; il difficile è capire quando funziona. Casi di studio: selezione di attività, codici di Huffman, zaino frazionario e resto.

L'idea

La programmazione dinamica, a ogni passo, valuta **tutte** le scelte possibili (prendo l'oggetto o no, divido in $k = 1, 2, \dots$) e sceglie dopo aver risolto i sottoproblemi. Un algoritmo **greedy** (goloso) è molto più impulsivo: fa subito la scelta che **localmente** sembra la migliore, e poi risolve l'unico sottoproblema che rimane. Non torna mai sui suoi passi.

È il comportamento di chi, a un buffet con un piatto piccolo, prende per primo il cibo che preferisce, poi il secondo preferito, e così via finché il piatto è pieno. A volte questa strategia porta al pranzo migliore possibile, a volte no (magari il piatto preferito occupava tutto lo spazio). La teoria serve a riconoscere quando la scelta golosa è sicura.

DEFINIZIONE: LE DUE PROPRIETÀ

Un problema si risolve correttamente con una strategia greedy se ha:

1. la **proprietà della scelta greedy**: esiste sempre una soluzione ottima che contiene la scelta golosa fatta al primo passo; quindi fare quella scelta non preclude l'ottimo;
2. la **sottostruttura ottima**: dopo la scelta golosa, ciò che resta è un sottoproblema dello stesso tipo, e una soluzione ottima del problema è formata dalla scelta golosa più una soluzione ottima del sottoproblema.

DA RICORDARE: GREEDY E PROGRAMMAZIONE DINAMICA

Entrambe richiedono la sottostruttura ottima. La differenza è **quando** si sceglie: la programmazione dinamica sceglie **dopo** aver risolto i sottoproblemi (dal basso verso l'alto), il greedy sceglie **prima**, e lascia un solo sottoproblema (dall'alto verso il basso). Per questo il greedy è più veloce, ma funziona su meno problemi.

Selezione di attività

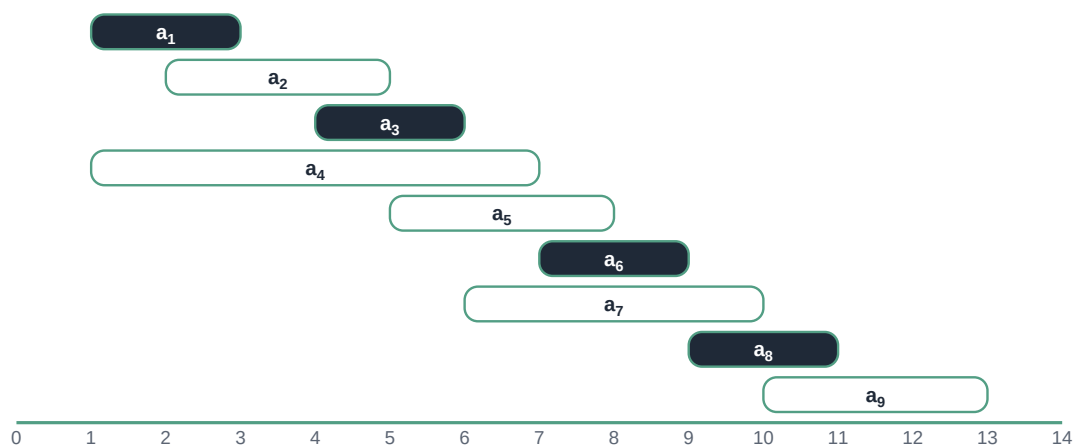
Abbiamo n attività che vogliono usare la stessa risorsa (per esempio un'aula), e una sola attività alla volta può usarla. L'attività a_i ha un tempo di inizio s_i e uno di fine f_i , e occupa l'intervallo $[s_i, f_i)$. Due attività sono **compatibili** se i loro intervalli non si sovrappongono. Vogliamo un insieme di attività mutuamente compatibili di **cardinalità massima**.

Quale scelta golosa? Si potrebbe pensare di prendere l'attività più corta, o quella che inizia prima: entrambe le idee falliscono su esempi semplici. Quella giusta è: **prendi l'attività che finisce per prima**. L'intuizione è che lascia libera la risorsa il prima possibile, cioè lascia il massimo spazio per le attività successive.

ESEMPIO

Nove attività, già ordinate per tempo di fine.

I	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	7	6	9	10
f_i	3	5	6	7	8	9	10	11	13



Le attività sull'asse dei tempi: in scuro quelle scelte dall'algoritmo.

1. Si prende a_1 , che finisce prima di tutte: la risorsa è libera dal tempo 3.
2. a_2 inizia a 2, prima di 3: scartata. a_3 inizia a 4: si prende, la risorsa è libera dal tempo 6.
3. a_4 (inizia a 1) e a_5 (inizia a 5) iniziano prima di 6: scartate. a_6 inizia a 7: si prende, libera dal tempo 9.
4. a_7 inizia a 6: scartata. a_8 inizia a 9: si prende, libera dal tempo 11.
5. a_9 inizia a 10, prima di 11: scartata.

Il risultato è $\{a_1, a_3, a_6, a_8\}$: quattro attività, ed è il massimo possibile.

```

1 GreedyActivitySelector(s, f, n)           // attività ordinate per f crescente
2   A = {a1}
3   k = 1                                   // ultima attività scelta
4   for m = 2 to n do
5       if s[m] >= f[k] then                 // a_m è compatibile con l'ultima scelta
6           A = A unione {a_m}
7           k = m
8   return A

```

DA SAPERE: SELEZIONE DI ATTIVITÀ

Con le attività già ordinate per tempo di fine l'algoritmo costa $\Theta(n)$; se vanno ordinate, $\Theta(n \log n)$ in tutto. La correttezza si dimostra con la proprietà della scelta greedy: in qualunque soluzione ottima si può sostituire la prima attività con quella che finisce per prima senza creare conflitti (la dimostrazione è nella dispensa di Laboratorio).

Codici di Huffman

Vogliamo comprimere un file di testo. Ogni carattere va rappresentato con una sequenza di bit, la sua **parola di codice**. Con un codice a **lunghezza fissa** per 6 caratteri servono 3 bit ciascuno. Ma se alcuni caratteri sono molto più frequenti di altri conviene un codice a **lunghezza variabile**: parole corte per i caratteri frequenti e lunghe per quelli rari. È la stessa idea dell'alfabeto Morse, in cui la lettera E, la più comune in inglese, è un solo punto.

CODICI PREFISSI

Con lunghezze variabili nasce un problema: come capire dove finisce una parola e inizia la successiva? Si usano i **codici prefissi**: nessuna parola di codice è prefisso di un'altra. Così la decodifica è immediata: si leggono bit finché si forma una parola valida, e quella non può essere l'inizio di nient'altro.

Un codice prefisso si rappresenta con un **albero binario**: i caratteri sono le foglie, e la parola di un carattere è il cammino dalla radice alla sua foglia, scrivendo 0 per ogni passo a sinistra e 1 per ogni passo a destra. La lunghezza della parola di c è la profondità $d_T(c)$ della sua foglia. Se $f(c)$ è la frequenza di c , il numero di bit necessari per codificare il file è il **costo** dell'albero:

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

Un codice ottimo corrisponde sempre a un albero **pieno** (ogni nodo interno ha due figli): se un nodo avesse un solo figlio, si potrebbe accorciare il codice togliendolo.

L'ALGORITMO

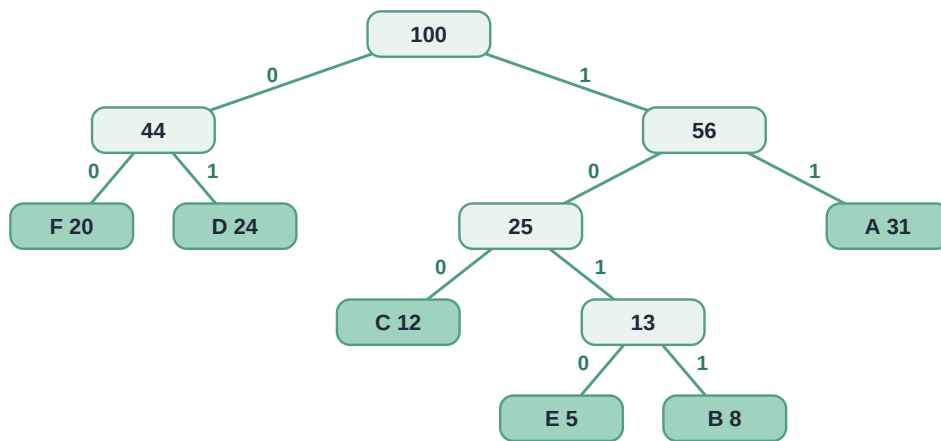
Huffman costruisce l'albero dal basso. Inizia con una foglia per carattere e ripete $n - 1$ volte: **estrae i due nodi con frequenza minore e li fonde** in un nuovo nodo, che li ha come figli e ha come frequenza la somma. I caratteri rari vengono fusi per primi e finiscono quindi più in profondità, con parole più lunghe.

```
1 Huffman(C)
2   n = |C|
3   Q = coda di min priorita' con i caratteri di C (chiave = frequenza)
4   for i = 1 to n - 1 do
5       alloca un nuovo nodo z
6       z.sinistro = x = ExtractMin(Q)
7       z.destro = y = ExtractMin(Q)
8       z.freq = x.freq + y.freq
9       Insert(Q, z)
10  return ExtractMin(Q)           // la radice dell'albero
```

ESEMPIO PASSO PER PASSO

Un file di 100 caratteri, con frequenze A 31, B 8, C 12, D 24, E 5, F 20.

PASSO	SI FONDONO	CODA DOPO IL PASSO
1	E 5 e B 8 → 13	C 12, 13, F 20, D 24, A 31
2	C 12 e 13 → 25	F 20, D 24, 25, A 31
3	F 20 e D 24 → 44	25, A 31, 44
4	25 e A 31 → 56	44, 56
5	44 e 56 → 100	100



Albero di Huffman: le foglie (evidenziate) sono i caratteri; sugli archi i bit 0 e 1.

CARATTERE	A	B	C	D	E	F
Frequenza	31	8	12	24	5	20
Codice	11	1011	100	01	1010	00
Bit totali	62	32	36	48	20	40

Il costo è $62 + 32 + 36 + 48 + 20 + 40 = \mathbf{238 \text{ bit}}$, contro i 300 bit del codice fisso a 3 bit: un risparmio di circa il 21 per cento. Per decodificare 1011001 si legge 1, 10, 101, 1011 = B; poi 0, 00 = F; poi 1, che non è ancora una parola: in un file completo seguirebbero altri bit.

DA SAPERE: HUFFMAN

Con la coda realizzata come min heap, le $n - 1$ iterazioni fanno ciascuna un numero costante di operazioni da $O(\log n)$: tempo $O(n \log n)$. Il codice prodotto è **ottimo** tra tutti i codici prefissi: la scelta greedy è sicura perché esiste sempre un albero ottimo in cui i due caratteri meno frequenti sono foglie sorelle alla massima profondità (dimostrazione nella dispensa di Laboratorio).

Zaino frazionario e zaino 0/1

Riprendiamo lo zaino, ma ora il ladro può prendere anche una **frazione** di un oggetto (polvere d'oro invece che lingotti). La strategia golosa giusta è: calcolare per ogni oggetto il **valore per unità di peso** v_i/w_i , e riempire lo zaino partendo dall'oggetto con il rapporto più alto, prendendone il più possibile, poi il successivo, finché lo zaino è pieno; l'ultimo oggetto può essere preso solo in parte.

Con gli oggetti del capitolo precedente, (peso, valore) = (1, 1), (3, 4), (4, 5), (5, 7) e capacità 7, i rapporti sono 1, 1,33, 1,25 e 1,4.

1. Si prende tutto l'oggetto 4 (rapporto 1,4): peso 5, valore 7. Restano 2 kg.
2. Il successivo è l'oggetto 2 (rapporto 1,33): ne entrano solo 2 kg su 3, cioè $2/3$, che valgono $4 \cdot 2/3 \approx 2,67$.
3. Lo zaino è pieno: valore totale $\approx \mathbf{9,67}$.

Nella versione **0/1** la stessa strategia sbaglia: preso l'oggetto 4, nei 2 kg rimasti entra solo l'oggetto 1, per un totale di 8, mentre l'ottimo (trovato con la programmazione dinamica) è 9, con gli oggetti 2 e 3. Il motivo è che nella versione 0/1 un oggetto "efficiente" può lasciare uno spazio vuoto che non si riesce a sfruttare, e la scelta locale non tiene conto di questo spreco.

```

1 FractionalKnapsack(w, v, n, W)
2     ordina gli oggetti per v[i]/w[i] decrescente
3     valore = 0 ; capacita = W
4     for i = 1 to n do
5         if w[i] <= capacita then          // l'oggetto entra tutto
6             valore = valore + v[i] ; capacita = capacita - w[i]
7         else                             // ne entra solo una parte
8             valore = valore + v[i] * capacita / w[i]
9     return valore
10    return valore

```

DA SAPERE: ZAINO

Lo zaino **frazionario** si risolve con il greedy in **$O(n \log n)$** (domina l'ordinamento). Lo zaino **0/1** ha la sottostruttura ottima ma **non** la proprietà della scelta greedy: serve la programmazione dinamica.

Il problema del resto

Per dare il resto con il minimo numero di monete, la strategia golosa è: **usa sempre la moneta più grande che non supera l'importo rimanente**. Con le monete in euro, in centesimi {1, 2, 5, 10, 20, 50, 100, 200}, funziona sempre. Per esempio per 2,87 euro, cioè 287 centesimi: 200 (restano 87), 50 (restano 37), 20 (restano 17), 10 (restano 7), 5 (restano 2), 2. Sei monete, ed è l'ottimo.

```

1 GreedyChange(d, k, R)                // tagli d[1] > d[2] > ... > d[k]
2     for i = 1 to k do
3         quante = floor(R / d[i])      // quante monete di questo taglio
4         R = R - quante * d[i]
5         dai "quante" monete da d[i]

```

Ma il greedy non funziona con tutti i sistemi di monete. Con i tagli {1, 4, 6} e importo 8, il greedy dà 6 + 1 + 1 (tre monete), mentre l'ottimo è 4 + 4 (due monete), come visto con la programmazione dinamica. I sistemi per cui il greedy è sempre ottimo si chiamano **canonici**; quelli usati per le monete reali sono progettati per esserlo.

DA SAPERE: RESTO

Il greedy costa **$O(k)$** con i tagli già ordinati ed è ottimo per i sistemi **canonici** come quello dell'euro. Per un sistema qualsiasi serve la programmazione dinamica, in $\Theta(k \cdot R)$.

Implementazione in C++

Per Huffman, invece di un heap, scegliamo i due nodi minimi scorrendo un array: il codice è più corto e il tempo diventa $O(n^2)$, irrilevante per pochi caratteri; con un min heap si torna a $O(n \log n)$. Ogni nodo dell'albero è un elemento di un array, e i figli sono indicati dal loro indice.

```

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

// Selezione di attivita' gia' ordinate per tempo di fine; stampa le scelte
int selezionaAttivita(int s[], int f[], int n) {
    int k = 0;                                // ultima attivita' scelta
    int quante = 1;
    cout << "attivita' scelte: 1 ";
    for (int m = 1; m < n; m++) {
        if (s[m] >= f[k]) {                    // compatibile con l'ultima scelta

```

```

        cout << m + 1 << " ";
        k = m;
        quante++;
    }
}
cout << endl;
return quante;
}

// Nodo dell'albero di Huffman
struct NodoH {
    char car;           // carattere (solo per le foglie)
    int freq;
    int sx, dx;         // indici dei figli, -1 se foglia
    bool usato;         // gia' fuso in un nodo padre
};

NodoH nodi[50];
int numNodi = 0;

// Restituisce l'indice del nodo non usato con frequenza minima
int estraiMinimo() {
    int min = -1;
    for (int i = 0; i < numNodi; i++)
        if (!nodi[i].usato && (min == -1 || nodi[i].freq < nodi[min].freq))
            min = i;
    nodi[min].usato = true;
    return min;
}

// Costruisce l'albero e restituisce l'indice della radice
int huffman(char c[], int f[], int n) {
    for (int i = 0; i < n; i++) {
        nodi[i].car = c[i]; nodi[i].freq = f[i];
        nodi[i].sx = nodi[i].dx = -1; nodi[i].usato = false;
    }
    numNodi = n;
    for (int i = 1; i < n; i++) {           // n - 1 fusioni
        int x = estraiMinimo();
        int y = estraiMinimo();
        nodi[numNodi].freq = nodi[x].freq + nodi[y].freq;
        nodi[numNodi].sx = x;
        nodi[numNodi].dx = y;
        nodi[numNodi].usato = false;
        numNodi++;
    }
    return numNodi - 1;                    // l'ultimo creato e' la radice
}

// Visita l'albero e stampa il codice di ogni foglia
void stampaCodici(int v, string codice) {
    if (nodi[v].sx == -1) {                 // foglia
        cout << nodi[v].car << " = " << codice << endl;
        return;
    }
    stampaCodici(nodi[v].sx, codice + "0"); // a sinistra si aggiunge 0
    stampaCodici(nodi[v].dx, codice + "1"); // a destra si aggiunge 1
}

// Zaino frazionario: ordina per valore/peso decrescente e riempie
double zainoFrazionario(double w[], double v[], int n, double W) {
    // ordinamento per rapporto decrescente (selection sort)

```

```

    for (int i = 0; i < n - 1; i++) {
        int best = i;
        for (int j = i + 1; j < n; j++)
            if (v[j] / w[j] > v[best] / w[best]) best = j;
        swap(w[i], w[best]);
        swap(v[i], v[best]);
    }
    double valore = 0, capacita = W;
    for (int i = 0; i < n; i++) {
        if (w[i] <= capacita) { // entra tutto
            valore += v[i];
            capacita -= w[i];
        } else { // ne entra solo una parte
            valore += v[i] * capacita / w[i];
            break;
        }
    }
    return valore;
}

// Resto greedy con tagli d[0] > d[1] > ... ; restituisce il numero di monete
int restoGreedy(int d[], int k, int R) {
    int monete = 0;
    for (int i = 0; i < k; i++) {
        int quante = R / d[i]; // monete di questo taglio
        R -= quante * d[i];
        monete += quante;
    }
    return monete;
}

int main() {
    int s[] = {1, 2, 4, 1, 5, 7, 6, 9, 10};
    int f[] = {3, 5, 6, 7, 8, 9, 10, 11, 13};
    int q = selezionaAttivita(s, f, 9);
    cout << "totale: " << q << endl;

    char c[] = {'A', 'B', 'C', 'D', 'E', 'F'};
    int fr[] = {31, 8, 12, 24, 5, 20};
    int radice = huffman(c, fr, 6);
    stampaCodici(radice, "");

    double w[] = {1, 3, 4, 5}, v[] = {1, 4, 5, 7};
    cout << "zaino frazionario: " << zainoFrazionario(w, v, 4, 7) << endl;

    int euro[] = {200, 100, 50, 20, 10, 5, 2, 1};
    cout << "monete per 287: " << restoGreedy(euro, 8, 287) << endl;
    return 0;
}

```

Cammini minimi da sorgente unica

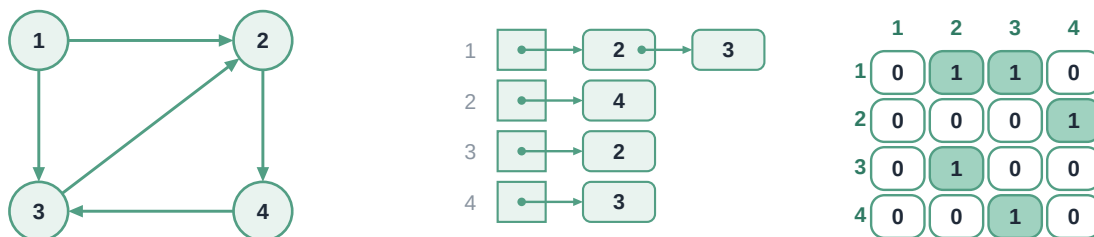
Grafi pesati, rilassamento degli archi e i tre algoritmi classici per trovare i cammini più brevi da un nodo a tutti gli altri: Bellman Ford, l'algoritmo per i grafi aciclici e Dijkstra.

Richiami sui grafi

Un **grafo** $G = (V, E)$ è formato da un insieme V di nodi (o vertici) e da un insieme E di archi, coppie di nodi. Nei grafi **orientati** l'arco (u, v) va da u a v e non viceversa; nei grafi **non orientati** vale in entrambi i sensi. In un grafo **pesato** ogni arco ha un peso $w(u, v)$, che può rappresentare una distanza, un tempo, un costo. Nelle complessità scriveremo V ed E per indicare il numero di nodi e di archi.

DUE RAPPRESENTAZIONI

- **Liste di adiacenza:** un array di V liste; la lista di u contiene i nodi v tali che (u, v) è un arco (con il peso, se il grafo è pesato). Occupa $\Theta(V + E)$ memoria ed è la scelta naturale per grafi **sparsi**, con pochi archi.
- **Matrice di adiacenza:** una matrice $V \times V$ in cui la cella (u, v) vale 1 (o il peso) se l'arco c'è e 0 (o ∞) se non c'è. Occupa $\Theta(V^2)$ memoria, ma dice in tempo costante se un arco esiste: va bene per grafi **densi**.



Lo stesso grafo orientato, con le sue liste di adiacenza e con la matrice di adiacenza (riga u , colonna v).

VISITA IN PROFONDITÀ E ORDINAMENTO TOPOLOGICO

Due strumenti del corso precedente ci servono qui. La **visita in profondità** (DFS) esplora il grafo scendendo il più possibile lungo ogni ramo prima di tornare indietro, e assegna a ogni nodo un tempo di **fine visita** (quando tutti i suoi discendenti sono stati esplorati). In un grafo orientato **aciclico** (DAG, directed acyclic graph) un **ordinamento topologico** è un ordine lineare dei nodi in cui ogni arco va da sinistra verso destra. Si ottiene facendo una DFS e mettendo ogni nodo in testa a una lista nel momento in cui finisce la sua visita, cioè ordinando i nodi per tempo di fine **decrescente**. Costa $\Theta(V + E)$. La sua correttezza è dimostrata nella dispensa di Laboratorio.

Il problema dei cammini minimi

Il **peso di un cammino** $p = \langle v_0, v_1, \dots, v_k \rangle$ è la somma dei pesi dei suoi archi. Il **peso del cammino minimo** da u a v è

$$\delta(u, v) = \min\{w(p) : p \text{ è un cammino da } u \text{ a } v\} \quad \text{oppure} \quad \infty \text{ se } v \text{ non è raggiungibile da } u$$

Nel problema dei **cammini minimi da sorgente unica** (SSSP, single source shortest paths) si fissa una sorgente s e si vuole $\delta(s, v)$ per **ogni** nodo v , insieme ai cammini stessi. È ciò che fa un navigatore quando calcola i tempi da casa tua verso tutte le destinazioni.

ARCHI NEGATIVI E CICLI NEGATIVI

I pesi possono essere negativi (un tratto in cui si guadagna invece di spendere). Questo non è un problema di per sé, ma se dalla sorgente si può raggiungere un **ciclo di peso negativo**, allora girando nel ciclo il peso scende all'infinito e il cammino minimo **non è definito**: si pone $\delta(s, v) = -\infty$. Bellman Ford sa riconoscere questa situazione; Dijkstra richiede invece che tutti i pesi siano non negativi.

DA RICORDARE: PROPRIETÀ UTILI

Un cammino minimo non contiene mai cicli: un ciclo positivo si può togliere accorciandolo, uno di peso zero si può togliere senza danni, uno negativo renderebbe il problema indefinito. Quindi un cammino minimo ha al più $V - 1$ archi. Inoltre ogni **sottocammino** di un cammino minimo è a sua volta minimo (sottostruttura ottima).

RAPPRESENTARE I CAMMINI: STIME E PREDECESSORI

Tutti gli algoritmi mantengono per ogni nodo v due attributi:

- $v.d$, la **stima** del peso del cammino minimo: parte da ∞ e scende man mano, senza mai andare sotto $\delta(s, v)$;
- $v.\pi$, il **predecessore** di v nel miglior cammino trovato finora. Seguendo i predecessori all'indietro da v si arriva a s ; alla fine i puntatori π formano l'**albero dei cammini minimi**.

```
1 InitializeSingleSource(G, s)
2   for each v in G.V do
3     v.d = infinito
4     v.pi = NIL
5   s.d = 0
```

IL RILASSAMENTO

L'operazione fondamentale si chiama **rilassamento** dell'arco (u, v) : ci si chiede se si può migliorare il cammino verso v passando per u . Se la stima di u più il peso dell'arco batte la stima attuale di v , si aggiorna v .

```
1 Relax(u, v, w)
2   if v.d > u.d + w(u, v) then
3     v.d = u.d + w(u, v)
4     v.pi = u
```

Un'immagine: $v.d$ è il miglior prezzo che conosci per arrivare a v . Un amico ti dice "io arrivo a u spendendo $u.d$, e da u a v il biglietto costa $w(u, v)$ ". Se la somma è più bassa del tuo prezzo, cambi piano e passi da u .

DA SAPERE: IL RILASSAMENTO

I tre algoritmi del capitolo differiscono **solo** nell'ordine in cui rilassano gli archi e nel numero di volte in cui lo fanno. Le proprietà che garantiscono la loro correttezza (limite superiore, convergenza, rilassamento del cammino) sono dimostrate nella dispensa di Laboratorio.

Algoritmo di Bellman Ford

Bellman Ford risolve il caso generale, con pesi anche negativi. L'idea è semplice e un po' "a forza bruta": si rilassano **tutti gli archi**, e si ripete questo giro $V - 1$ volte. Poiché un cammino minimo ha al più $V - 1$ archi, dopo il giro i è sicuramente corretto ogni cammino minimo di al più i archi. Alla fine si fa un ultimo controllo: se qualche arco si può ancora rilassare, c'è un ciclo negativo raggiungibile.

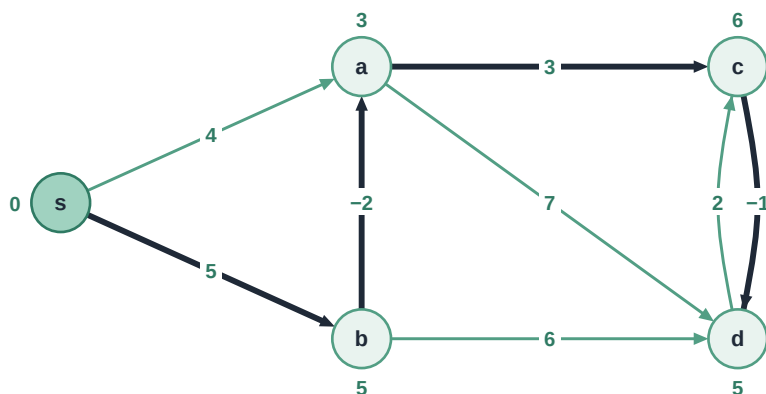
```

1 BellmanFord(G, w, s)
2   InitializeSingleSource(G, s)
3   for i = 1 to |G.V| - 1 do
4     for each arco (u, v) in G.E do
5       Relax(u, v, w)
6   for each arco (u, v) in G.E do      // controllo dei cicli negativi
7     if v.d > u.d + w(u, v) then return FALSE
8   return TRUE

```

ESEMPIO PASSO PER PASSO

Sorgente s. Gli archi vengono rilassati sempre in questo ordine: (s,a), (s,b), (a,c), (a,d), (b,a), (b,d), (c,d), (d,c).



Grafo con archi negativi e senza cicli negativi (il ciclo c, d pesa $-1 + 2 = 1$). Accanto ai nodi i valori finali di d; in grassetto l'albero dei cammini minimi.

Primo giro. Partendo da $s.d = 0$ e tutte le altre stime a ∞ :

1. (s,a): $a.d = 4$. (s,b): $b.d = 5$.
2. (a,c): $c.d = 4 + 3 = 7$. (a,d): $d.d = 4 + 7 = 11$.
3. (b,a): $5 - 2 = 3$ è minore di 4, quindi $a.d = 3$ con $a.\pi = b$. (b,d): $5 + 6 = 11$, non migliora.
4. (c,d): $7 - 1 = 6$, quindi $d.d = 6$ con $d.\pi = c$. (d,c): $6 + 2 = 8$, non migliora.

Secondo giro. Ora $a.d = 3$ è più basso di quando è stato usato per c: (a,c) dà $3 + 3 = 6$, quindi $c.d = 6$; poi (c,d) dà $6 - 1 = 5$, quindi $d.d = 5$. Nessun altro arco migliora.

Terzo e quarto giro. Nessun arco cambia più nulla. Il controllo finale non trova archi rilassabili: si restituisce TRUE.

STIME	S	A	B	C	D
iniziali	0	∞	∞	∞	∞
dopo il giro 1	0	3	5	7	6
dopo il giro 2	0	3	5	6	5
finali	0	3	5	6	5

Seguendo i predecessori: $d.\pi = c$, $c.\pi = a$, $a.\pi = b$, $b.\pi = s$. Il cammino minimo verso d è s, b, a, c, d con peso $5 - 2 + 3 - 1 = 5$.

DA SAPERE: BELLMAN FORD

Tempo $\Theta(V \cdot E)$: $V - 1$ giri, ciascuno su tutti gli E archi, più un giro di controllo. Funziona con **pesi negativi** e restituisce FALSE se esiste un **ciclo negativo raggiungibile** dalla sorgente. Spesso converge prima del giro $V - 1$: si può fermare in anticipo se un giro intero non cambia nulla.

Cammini minimi in un grafo aciclico

Se il grafo è un **DAG** non ci sono cicli, quindi nemmeno cicli negativi, e si può fare molto meglio. Si ordinano i nodi topologicamente e si rilassano gli archi uscenti di ciascun nodo **seguendo quell'ordine**. Quando si arriva a un nodo u , tutti i nodi che possono precederlo in un cammino sono già stati trattati, quindi $u.d$ è già definitivo: basta un solo passaggio.

È come una catena di montaggio a più fasi: ogni fase dipende solo da quelle precedenti, quindi elaborandole in ordine ognuna si fa una volta sola.

```
1 DagShortestPaths(G, w, s)
2   ordina topologicamente i nodi di G
3   InitializeSingleSource(G, s)
4   for each u nell'ordine topologico do
5       for each v in Adj[u] do
6           Relax(u, v, w)
```

ESEMPIO

Nodi già in ordine topologico p, q, r, s, t, u ; la sorgente è q .



DAG con sorgente q : sotto ogni nodo il valore finale di d . Il nodo p precede la sorgente e resta irraggiungibile.

1. **p**: $p.d = \infty$, rilassare i suoi archi non cambia nulla.
2. **q**: $r.d = 2$, $s.d = 7$.
3. **r** ($d = 2$): $s.d = \min(7, 2 - 1) = 1$; $t.d = 2 + 4 = 6$.
4. **s** ($d = 1$): $t.d = \min(6, 1 + 1) = 2$; $u.d = 1 + 5 = 6$.
5. **t** ($d = 2$): $u.d = \min(6, 2 - 2) = 0$.
6. **u**: non ha archi uscenti.

DA SAPERE: DAG SHORTEST PATHS

Tempo $\Theta(V + E)$: ordinamento topologico più un solo rilassamento per arco. Funziona anche con **pesi negativi**, perché in un DAG non possono esserci cicli. Un'applicazione classica è il calcolo dei **cammini critici** nella pianificazione dei progetti (usando pesi negati per trovare il cammino più lungo).

Algoritmo di Dijkstra

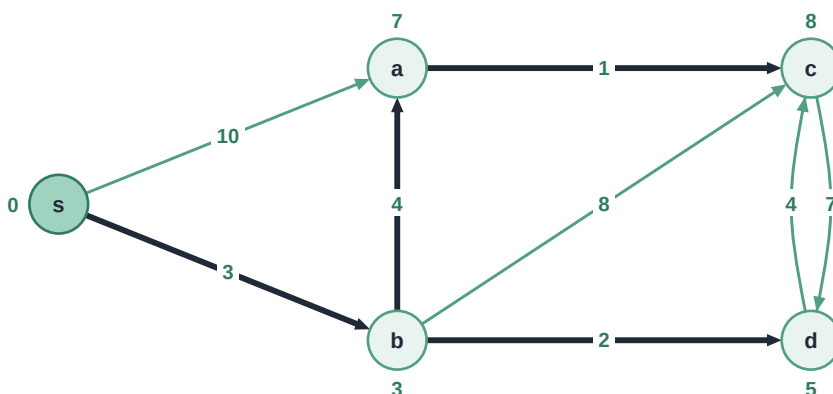
Dijkstra risolve il problema quando **tutti i pesi sono non negativi**, ed è più veloce di Bellman Ford. Mantiene un insieme S di nodi il cui cammino minimo è già **definitivo**. A ogni passo estrae, tra i nodi non ancora in S , quello con la stima d **più piccola** (scelta greedy), lo aggiunge a S e rilassa i suoi archi uscenti. Per estrarre velocemente il minimo usa una coda di min priorità Q con chiave d .

Immagina di versare acqua nella sorgente di una rete di canali: l'acqua raggiunge i punti nell'ordine della loro distanza, e quando arriva in un punto, quel punto non verrà mai raggiunto prima da un'altra strada. Dijkstra simula proprio questo fronte che avanza: il nodo estratto è il prossimo che l'acqua tocca.

```
1 Dijkstra(G, w, s)
2   InitializeSingleSource(G, s)
3   S = insieme vuoto
4   Q = G.V // coda di min priorit  con chiave d
5   while Q non   vuota do
6     u = ExtractMin(Q)
7     S = S unione {u} // la stima di u   definitiva
8     for each v in Adj[u] do
9       Relax(u, v, w) // se v.d cala, DecreaseKey su Q
```

ESEMPIO PASSO PER PASSO

Sorgente s , pesi tutti positivi.



Grafo per Dijkstra con i valori finali di d ; in grassetto l'albero dei cammini minimi.

PASSO	S	A	B	C	D	NODO ESTRATTO
inizio	0	∞	∞	∞	∞	.
1	0	10	3	∞	∞	s
2	0	7	3	11	5	b
3	0	7	3	9	5	d
4	0	7	3	8	5	a
5	0	7	3	8	5	c

- Si estrae s ($d = 0$) e si rilassano $s \rightarrow a$ e $s \rightarrow b$: $a.d = 10$, $b.d = 3$.
- Il minimo tra i non definitivi   b (3). Si rilassano i suoi archi: $a.d = \min(10, 3 + 4) = 7$, $c.d = 3 + 8 = 11$, $d.d = 3 + 2 = 5$.
- Il minimo   d (5): $d \rightarrow c$ d  $5 + 4 = 9 < 11$, quindi $c.d = 9$.

4. Il minimo è a (7): $a \rightarrow c$ dà $7 + 1 = 8 < 9$, quindi $c.d = 8$.
5. Resta c (8): $c \rightarrow d$ dà 15, ma d è già definitivo e la sua stima è 5. Fine.

DA SAPERE: DIJKSTRA

Funziona **solo con pesi non negativi**: con un arco negativo un nodo già estratto potrebbe in seguito essere raggiunto con un cammino più corto, e la scelta greedy non sarebbe più sicura. Il tempo dipende dalla coda di priorità: con un **array** semplice è $O(V^2)$, adatto ai grafi densi; con un **min heap** è $O((V + E) \log V)$, adatto ai grafi sparsi.

ALGORITMO	IPOTESI SUL GRAFO	PESI NEGATIVI	TEMPO
Bellman Ford	nessuna	sì, rileva cicli negativi	$O(V \cdot E)$
DAG shortest paths	aciclico	sì	$O(V + E)$
Dijkstra	pesi non negativi	no	$O((V + E) \log V)$ con heap

Implementazione in C++

I nodi sono numerati da 0. Bellman Ford lavora su una lista di archi; per il DAG e per Dijkstra usiamo una matrice dei pesi, con INF dove l'arco non c'è. Dijkstra è nella versione con array, $O(V^2)$, la più semplice da scrivere.

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

const int MAXV = 20;
const int INF = 1000000000;

int d[MAXV], pi[MAXV];           // stime e predecessori

void inizializza(int n, int s) {
    for (int v = 0; v < n; v++) { d[v] = INF; pi[v] = -1; }
    d[s] = 0;
}

// Rilassamento dell'arco (u, v) di peso w
void relax(int u, int v, int w) {
    if (d[u] != INF && d[v] > d[u] + w) {
        d[v] = d[u] + w;
        pi[v] = u;
    }
}

// Bellman Ford su una lista di m archi (da[i], a[i], peso[i])
bool bellmanFord(int n, int m, int da[], int a[], int peso[], int s) {
    inizializza(n, s);
    for (int i = 1; i <= n - 1; i++)           // n - 1 giri
        for (int j = 0; j < m; j++)
            relax(da[j], a[j], peso[j]);
    for (int j = 0; j < m; j++)               // controllo finale
        if (d[da[j]] != INF && d[a[j]] > d[da[j]] + peso[j])
            return false;                     // ciclo negativo
    return true;
}

int W[MAXV][MAXV];               // matrice dei pesi (INF = nessun arco)
bool visitato[MAXV];
int ordine[MAXV], posizione;
```

```

// DFS: a fine visita il nodo va in testa all'ordine
void dfs(int n, int u) {
    visitato[u] = true;
    for (int v = 0; v < n; v++)
        if (W[u][v] != INF && !visitato[v]) dfs(n, v);
    posizione--;
    ordine[posizione] = u;
}

void dagShortestPaths(int n, int s) {
    for (int v = 0; v < n; v++) visitato[v] = false;
    posizione = n;
    for (int v = 0; v < n; v++)
        if (!visitato[v]) dfs(n, v);
    inizializza(n, s);
    for (int i = 0; i < n; i++) {
        int u = ordine[i];
        for (int v = 0; v < n; v++)
            if (W[u][v] != INF) relax(u, v, W[u][v]);
    }
}

// Dijkstra con array: a ogni passo cerca il minimo tra i non definitivi
void dijkstra(int n, int s) {
    bool definitivo[MAXV];
    inizializza(n, s);
    for (int v = 0; v < n; v++) definitivo[v] = false;
    for (int passo = 0; passo < n; passo++) {
        int u = -1;
        for (int v = 0; v < n; v++)
            if (!definitivo[v] && (u == -1 || d[v] < d[u])) u = v;
        if (d[u] == INF) break;
        definitivo[u] = true;
        for (int v = 0; v < n; v++)
            if (W[u][v] != INF && !definitivo[v]) relax(u, v, W[u][v]);
    }
}

// Stampa il cammino dalla sorgente a v seguendo i predecessori
void stampaCammino(int v, char nome[]) {
    if (pi[v] != -1) stampaCammino(pi[v], nome);
    cout << nome[v] << " ";
}

void azzerMatrice(int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) W[i][j] = INF;
}

int main() {
    // Bellman Ford: s=0 a=1 b=2 c=3 d=4
    char nomil[] = {'s', 'a', 'b', 'c', 'd'};
    int da[] = {0, 0, 1, 1, 2, 2, 3, 4};
    int a[] = {1, 2, 3, 4, 1, 4, 4, 3};
    int peso[] = {4, 5, 3, 7, -2, 6, -1, 2};
    bool ok = bellmanFord(5, 8, da, a, peso, 0);
    cout << "nessun ciclo negativo: " << ok << endl;
    for (int v = 0; v < 5; v++) cout << nomil[v] << "=" << d[v] << " ";
    cout << endl << "cammino verso d: ";
    stampaCammino(4, nomil);
    cout << endl;
}

```

```

// DAG: p=0 q=1 r=2 s=3 t=4 u=5, sorgente q
char nomi2[] = {'p', 'q', 'r', 's', 't', 'u'};
azzerMatrice(6);
W[0][1] = 3; W[0][2] = 6; W[1][2] = 2; W[1][3] = 7; W[2][3] = -1;
W[2][4] = 4; W[3][4] = 1; W[3][5] = 5; W[4][5] = -2;
dagShortestPaths(6, 1);
for (int v = 0; v < 6; v++) {
    cout << nomi2[v] << "=";
    if (d[v] == INF) cout << "inf "; else cout << d[v] << " ";
}
cout << endl;

// Dijkstra: s=0 a=1 b=2 c=3 d=4
azzerMatrice(5);
W[0][1] = 10; W[0][2] = 3; W[1][3] = 1; W[2][1] = 4;
W[2][3] = 8; W[2][4] = 2; W[4][3] = 4; W[3][4] = 7;
dijkstra(5, 0);
for (int v = 0; v < 5; v++) cout << nomi1[v] << "=" << d[v] << " ";
cout << endl;
return 0;
}

```

Cammini minimi tra tutte le coppie

Come calcolare in un colpo solo le distanze tra ogni coppia di nodi: il metodo basato sulla moltiplicazione di matrici e l'algoritmo di Floyd Warshall, entrambi esempi di programmazione dinamica.

Il problema

Ora vogliamo $\delta(u, v)$ per **ogni** coppia di nodi u e v : è la tabella delle distanze chilometriche stampata in fondo agli atlanti stradali. Una prima soluzione è eseguire un algoritmo a sorgente unica da ciascuno dei V nodi:

- V volte Dijkstra, se i pesi sono non negativi: $O(V^3)$ con l'array, $O(V(V + E) \log V)$ con l'heap;
- V volte Bellman Ford, se ci sono pesi negativi: $O(V^2 E)$, che per grafi densi diventa $O(V^4)$.

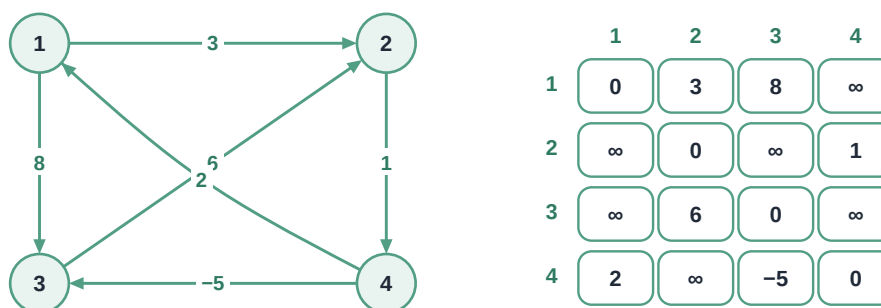
Gli algoritmi di questo capitolo lavorano direttamente su **matrici** e ammettono archi negativi (ma non cicli negativi). I nodi sono numerati da 1 a n , con $n = V$, e il grafo è dato con la matrice dei pesi $W = (w_{ij})$:

$$w_{ij} = 0 \text{ se } i = j; \quad \text{peso dell'arco } (i, j) \text{ se esiste}; \quad \infty \text{ altrimenti}$$

L'uscita è una matrice $D = (d_{ij})$ con $d_{ij} = \delta(i, j)$, e una matrice dei **predecessori** $\Pi = (\pi_{ij})$, dove π_{ij} è il nodo che precede j nel cammino minimo da i (NIL se $i = j$ o se j non è raggiungibile). La riga i di Π è l'albero dei cammini minimi con sorgente i .

ESEMPIO DI RIFERIMENTO

Useremo sempre questo grafo con quattro nodi, che ha due archi negativi ma nessun ciclo negativo.



Il grafo di riferimento e la sua matrice dei pesi W .

Metodo basato sulla moltiplicazione di matrici

È un'applicazione diretta della programmazione dinamica. Sia $l_{ij}^{(m)}$ il peso minimo di un cammino da i a j che usa **al più m archi**. Con zero archi si arriva solo a sé stessi:

$$l_{ij}^{(0)} = 0 \text{ se } i = j, \quad \infty \text{ altrimenti}$$

Un cammino di al più m archi è un cammino di al più $m - 1$ archi fino a un certo nodo k , seguito dall'arco (k, j) . Si prova ogni k :

$$l_{ij}^{(m)} = \min_{1 \leq k \leq n} (l_{ik}^{(m-1)} + w_{kj})$$

(il caso $k = j$, con $w_{jj} = 0$, copre la possibilità di non aggiungere archi). Poiché un cammino minimo ha al più $n - 1$ archi, la risposta è $L_{ij}^{(n-1)}$, e $L^{(1)} = W$.

PERCHÉ "MULTIPLICAZIONE"

Si confronti con il prodotto di matrici $C = A \cdot B$, dove $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$. La formula è identica se si sostituisce la somma con il **minimo** e il prodotto con la **somma**. Il calcolo di $L^{(m)}$ da $L^{(m-1)}$ è quindi una "moltiplicazione" $L^{(m-1)} \otimes W$, e si scrive con la stessa struttura a tre cicli.

```
1 ExtendShortestPaths(L, W, n)
2   sia L' una nuova matrice n x n
3   for i = 1 to n do
4     for j = 1 to n do
5       L'[i][j] = infinito
6       for k = 1 to n do
7         L'[i][j] = min(L'[i][j], L[i][k] + W[k][j])
8   return L'
9
10 SlowAllPairsShortestPaths(W, n)
11   L = W
12   for m = 2 to n - 1 do
13     L = ExtendShortestPaths(L, W, n)
14   return L
```

Nell'esempio, $L^{(2)}$ ammette cammini di due archi. Per esempio $l_{23}^{(2)} = \min(\infty + 8, 0 + \infty, \infty + 0, 1 - 5) = -4$, grazie al cammino 2, 4, 3; e $l_{42}^{(2)} = \min(2 + 3, \infty + 0, -5 + 6, 0 + \infty) = 1$, grazie a 4, 3, 2. Con $L^{(3)}$ compaiono i cammini di tre archi, come 1, 2, 4, 3 di peso -1 e 3, 2, 4, 1 di peso 9.

L(2)	1	2	3	4	L(3)	1	2	3	4
1	0	3	8	4	1	0	3	-1	4
2	3	0	-4	1	2	3	0	-4	1
3	∞	6	0	7	3	9	6	0	7
4	2	1	-5	0	4	2	1	-5	0

VERSIONE VELOCE: QUADRATI RIPETUTI

Ogni estensione costa $\Theta(n^3)$ e ne servono $n - 2$: in tutto $\Theta(n^4)$. Ma non serve passare per tutti i valori di m . Il prodotto $\otimes$ è associativo, quindi $L^{(2)} = W \otimes W$, $L^{(4)} = L^{(2)} \otimes L^{(2)}$, $L^{(8)} = L^{(4)} \otimes L^{(4)}$, e così via, raddoppiando ogni volta. Superare $n - 1$ non fa danni, perché senza cicli negativi aggiungere archi non abbassa più i pesi. Bastano $\lceil \log_2(n - 1) \rceil$ "quadrati": è la stessa idea della potenza veloce $x^8 = ((x^2)^2)^2$.

```
1 FasterAllPairsShortestPaths(W, n)
2   L = W ; m = 1
3   while m < n - 1 do
4     L = ExtendShortestPaths(L, L, n) // da L(m) a L(2m)
5     m = 2m
6   return L
```

DA SAPERE: METODO DELLE MATRICI

Versione lenta $\Theta(n^4)$, versione con quadrati ripetuti $\Theta(n^3 \log n)$. Il metodo è utile soprattutto per capire la struttura del problema: Floyd Warshall fa meglio.

Algoritmo di Floyd Warshall

Floyd Warshall usa un'altra scomposizione in sottoproblemi. Invece di limitare il **numero** di archi, limita **quali nodi** il cammino può attraversare. Si chiamano **intermedi** i nodi di un cammino diversi dal primo e dall'ultimo.

DEFINIZIONE: SOTTOPROBLEMA DI FLOYD WARSHALL

$d_{ij}^{(k)}$ è il peso minimo di un cammino da i a j i cui nodi intermedi appartengono tutti all'insieme $\{1, 2, \dots, k\}$.

Con $k = 0$ non sono ammessi intermedi, quindi conta solo l'arco diretto: $D^{(0)} = W$. Con $k = n$ tutti i nodi sono ammessi e si ottiene la risposta. Il passaggio da $k - 1$ a k è il cuore dell'algoritmo. Il cammino minimo da i a j con intermedi in $\{1, \dots, k\}$ può:

- **non passare per k** : allora è lo stesso di prima, $d_{ij}^{(k-1)}$;
- **passare per k** : allora si spezza in un tratto da i a k e uno da k a j , ciascuno con intermedi in $\{1, \dots, k - 1\}$ (k compare una volta sola, il cammino non ha cicli), di peso $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

Un'analogia: stai organizzando voli con scalo. All'inizio consideri solo voli diretti. Poi ammetti lo scalo nell'aeroporto 1 e controlli, per ogni coppia di città, se passare da lì conviene. Poi ammetti anche l'aeroporto 2, e così via: a ogni passo una sola nuova possibilità di scalo, provata su tutte le coppie.

I PREDECESSORI

Insieme a D si aggiorna Π . All'inizio $\pi_{ij} = i$ se l'arco (i, j) esiste e i è diverso da j , altrimenti NIL. Quando conviene passare per k , il predecessore di j diventa quello che j ha nel cammino da k : $\pi_{ij} = \pi_{kj}$.

```
1 FloydWarshall(W, n)
2   D = W
3   inizializza P come descritto sopra
4   for k = 1 to n do                      // nuovo nodo intermedio ammesso
5       for i = 1 to n do
6           for j = 1 to n do
7               if D[i][k] + D[k][j] < D[i][j] then
8                   D[i][j] = D[i][k] + D[k][j]
9                   P[i][j] = P[k][j]
10  return D, P
```

Si può usare un'unica matrice D aggiornata sul posto: durante il giro k la riga k e la colonna k non cambiano (passare per k per andare da k a j non conviene mai, perché $d_{kk} = 0$), quindi non si leggono mai valori "sbagliati".

ESEMPIO PASSO PER PASSO

Partiamo da $D^{(0)} = W$. In ogni giro sono in grassetto le celle che migliorano.

$k = 1$. Si cerca chi migliora passando per 1. Al nodo 1 arriva solo il 4 ($d_{41} = 2$), quindi può cambiare solo la riga 4: $d_{42} = \min(\infty, 2 + 3) = 5$, mentre $d_{43} = \min(-5, 2 + 8)$ resta -5 .

$k = 2$. $d_{14} = \min(\infty, 3 + 1) = 4$ e $d_{34} = \min(\infty, 6 + 1) = 7$.

$k = 3$. Dal nodo 3 si va in 2 con peso 6. Migliora $d_{42} = \min(5, -5 + 6) = 1$.

$k = 4$. Il nodo 4 permette di raggiungere 1 e 3: $d_{13} = \min(8, 4 - 5) = -1$, $d_{21} = \min(\infty, 1 + 2) = 3$, $d_{23} = \min(\infty, 1 - 5) = -4$, $d_{31} = \min(\infty, 7 + 2) = 9$.

D(2)	1	2	3	4	D(4)	1	2	3	4
1	0	3	8	4	1	0	3	-1	4
2	∞	0	∞	1	2	3	0	-4	1
3	∞	6	0	7	3	9	6	0	7
4	2	5	-5	0	4	2	1	-5	0

(In $D^{(2)}$ la cella $d_{42} = 5$ era già migliorata nel giro $k = 1$; in $D^{(4)}$ $d_{42} = 1$ viene dal giro $k = 3$.) La matrice finale coincide con $L^{(3)}$ del metodo precedente.

Ricostruire un cammino. La matrice finale dei predecessori ha riga 3 uguale a $\langle 4, 3, \text{NIL}, 2 \rangle$. Per il cammino da 3 a 1: $\pi_{31} = 4, \pi_{34} = 2, \pi_{32} = 3$. Leggendo al contrario: 3, 2, 4, 1, di peso $6 + 1 + 2 = 9$.

```

1 PrintAllPairsShortestPath(P, i, j)
2   if i = j then stampa i
3   else if P[i][j] = NIL then stampa "nessun cammino"
4   else
5       PrintAllPairsShortestPath(P, i, P[i][j])
6       stampa j

```

DA SAPERE: FLOYD WARSHALL

Tre cicli annidati da n : tempo $\Theta(n^3)$, con costanti molto piccole; spazio $\Theta(n^2)$. Ammette archi negativi. Se il grafo ha un ciclo negativo, alla fine qualche elemento della diagonale d_{ii} risulta negativo: è il modo per accorgersene.

METODO	TEMPO	PESI NEGATIVI
n volte Dijkstra (array)	$O(n^3)$	no
n volte Bellman Ford	$O(n^2 E)$	sì
Matrici, versione lenta	$\Theta(n^4)$	sì
Matrici, quadrati ripetuti	$\Theta(n^3 \log n)$	sì
Floyd Warshall	$\Theta(n^3)$	sì

Implementazione in C++

Nel codice i nodi vanno da 0 a $n - 1$ (nella stampa si aggiunge 1 per tornare alla numerazione del capitolo). La funzione somma evita che $\infty +$ un numero negativo diventi un valore finito sbagliato. Il main ricalcola l'esempio con il metodo lento e con Floyd Warshall, e ricostruisce il cammino da 3 a 1.

```

#include <iostream>
using namespace std;

const int MAXV = 10;
const int INF = 1000000000;

// Somma che rispetta l'infinito: inf + qualcosa = inf
int somma(int a, int b) {
    if (a == INF || b == INF) return INF;
    return a + b;
}

```

```

// Un passo del metodo con moltiplicazione: C = A "per" W con (min, +)
void estendi(int n, int A[MAXV][MAXV], int W[MAXV][MAXV], int C[MAXV][MAXV]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = INF;
            for (int k = 0; k < n; k++)
                if (somma(A[i][k], W[k][j]) < C[i][j])
                    C[i][j] = somma(A[i][k], W[k][j]);
        }
}

// Versione lenta: n - 2 estensioni a partire da L = W
void apsplento(int n, int W[MAXV][MAXV], int L[MAXV][MAXV]) {
    int T[MAXV][MAXV];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) L[i][j] = W[i][j];
    for (int m = 2; m <= n - 1; m++) {
        estendi(n, L, W, T);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++) L[i][j] = T[i][j];
    }
}

// Floyd Warshall: D = distanze, P = predecessori (-1 = nessuno)
void floydWarshall(int n, int W[MAXV][MAXV], int D[MAXV][MAXV], int P[MAXV][MAXV]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            D[i][j] = W[i][j];
            if (i != j && W[i][j] != INF) P[i][j] = i;
            else P[i][j] = -1;
        }
    for (int k = 0; k < n; k++) // nodo intermedio ammesso
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (somma(D[i][k], D[k][j]) < D[i][j]) {
                    D[i][j] = somma(D[i][k], D[k][j]);
                    P[i][j] = P[k][j]; // si passa per k
                }
}

// Stampa il cammino da i a j usando i predecessori (nodi numerati da 1)
void stampaCammino(int P[MAXV][MAXV], int i, int j) {
    if (i == j) cout << i + 1 << " ";
    else if (P[i][j] == -1) cout << "nessun cammino ";
    else {
        stampaCammino(P, i, P[i][j]);
        cout << j + 1 << " ";
    }
}

void stampaMatrice(int n, int M[MAXV][MAXV]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (M[i][j] == INF) cout << "inf ";
            else cout << M[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int n = 4;

```

```

int W[MAXV][MAXV] = {
    {0, 3, 8, INF},
    {INF, 0, INF, 1},
    {INF, 6, 0, INF},
    {2, INF, -5, 0}
};
int L[MAXV][MAXV], D[MAXV][MAXV], P[MAXV][MAXV];
apspLento(n, W, L);
cout << "metodo lento:" << endl;
stampaMatrice(n, L);
floydWarshall(n, W, D, P);
cout << "Floyd Warshall:" << endl;
stampaMatrice(n, D);
cout << "predecessori (0 = nessuno):" << endl;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) cout << P[i][j] + 1 << " ";
    cout << endl;
}
cout << "cammino da 3 a 1: ";
stampaCammino(P, 2, 0);
cout << endl;
return 0;
}

```