Algoritmi

Lezione 1

Correttezza e analisi delle performance:

Utilizzo delle risorse:

- Memoria;
- Tempo computazionale (runtime);
- Larghezza della banda di comunicazione;
- Energie consumate;

Random Access Machine (RAM)

Assumeremo che gli algoritmi siano formalizzati come programmi informatici scritti in pseudocodice.

- Le istruzioni di un algoritmo sono eseguite in maniera sequenziale.
- Ogni **istruzione elementare** e accesso alla memoria impiega **tempo costante**.

Istruzioni elementari RAM:

- Operazioni aritmetiche (+, -, ·, ÷, %, floor, ceiling);
- Spostamento dei dati (upload, save, copy);
- Controllo (branching condizionale, branching incondizionale, chiamata di subroutine e return);

Formato dei dati:

- Interi;
- Floating point;
- Caratteri;

Ogni stringa di dati è codificata da un **numero limitato di bit**.

Es. se devo codificare una stringa di lunghezza n, posso assumere che essa sia rappresentabile con $c \cdot \log_2 n, \exists\, c \in Q,\, c \geq 1.$

Il modello **RAM** non tiene conto delle gerarchie di memoria.

La misura dell'input dipende dal **tipo di dato** o **lunghezza**.

Calcolo del tempo computazionale

Il **tempo computazionale (routine)** rappresenta il numero di istruzioni elementari e di accessi ai dati eseguiti, in funzione della lunghezza dell'input n.

Prendiamo ad esempio l'algoritmo di ordinamento **Insertion Sort** in pseudo-codice:

```
1. for i = 2 to n
2.   key = A[i]
3.   j = i-1
4.   while j > 0 and A[j] > key
5.     A[j+1] = A[j]
6.       j = j-1
7.   A[j+1] = key
```

Analizziamo il costo di ogni operazione in funzione di n:

$$c_1 : n$$

$$c_2 : n-1$$

$$c_3 : n-1$$

$$c_4 : \sum_{i=2}^{n} t_i$$

$$c_5 : \sum_{i=2}^{n} (t_i - 1)$$

$$c_6 : \sum_{i=2}^{n} (t_i - 1)$$

$$c_7 : n-1$$

Best Case

Nell'esempio, il **best case** si verifica quando l'input è **già ordinato**, dunque il ciclo while (righe 4,5,6) non verrà mai eseguito:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \cdot 0 + c_6 \cdot 0 + c_7(n-1)$$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7)n + (-c_2 - c_3 - c_4 - c_5 - c_6 - c_7)$$

$$T(n) = an + b$$

Otteniamo dunque un tempo lineare.

Worst Case

In questo caso, il worst case si verifica quando l'input è ordinato al contrario:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} i + c_5 \sum_{i=2}^{n} (i-1) + c_6 \sum_{i=2}^{n} (i-1) + c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5(\frac{n(n+1)}{2}) + c_6(\frac{n(n+1)}{2}) + c_7(n-1)$$

$$T(n) = (\frac{c_4 + c_5 + c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7)n + (-c_2 - c_3 - c_4 - c_7)$$

$$T(n) = an^2 + bn + c$$

Otteniamo in questo caso un **tempo quadratico**.

In generale, un algoritmo si dirà **più efficiente** di un altro se il runtime sul caso peggiore cresce più lentamente al crescere della dimensione dell'input.

Notazione asintotica

Esistono varie notazioni per esplicitare la crescita asintotica di un algoritmo:

- **Big-O** notation: denota un limite superiore (upper bound) al comportamento asintotico di una funzione. Ad esempio la funzione:

$$f(n) = 7n^3 + 10n^2 - 20n + 6$$

$f(n)$ non crescerà più velocemente di $8n^3$, che dunque rappresenterà un limite superiore per $f(n)$, e quindi:

$$f(n) = 7n^3 + 10n^2 - 20n + g, O(n^3)$$

In generale, diremo che:

$$O(g(n)) = \{f(n) \mid \exists\, c \in Q^+,\ \exists\, n_0 \in \mathbb{N} : \forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n)\}$$

- Ω notation: denota un limite inferiore (lower bound) al comportamento asintotico di una funzione. Prendendo come esempio la funzione precedente, $f(n)$ non crescerà più lentamente di $7n^3$, che rappresenterà un limite inferiore per $f(n)$, e quindi:

$$f(n) = 7n^3 + 10n^2 - 20n + g, \Omega(n^3)$$

In generale, diremo che:

$$\Omega(g(n)) = \{f(n) \mid \exists\, c \in Q^+,\ \exists\, n_0 \in \mathbb{N} : \forall n \geq n_0, 0 \leq c \cdot g(n) \leq f(n)\}$$

- Θ notation: denota un limite superiore e inferiore (tight bound) al comportamento asintotico di una funzione. Nel caso della funzione precedente, possiamo notare che n^3 rappresenta sia un limite superiore che inferiore, e dunque:

$$f(n) = 7n^3 + 10n^2 - 20n + g, \Theta(n^3)$$

Dunque, se $\Omega(g(n)) = O(g(n)) \iff \Theta(g(n))$.

- **Little-o** notation: denota un limite superiore (upper bound) non asintoticamente stretto:

$$o(g(n)) = \{f(n) \mid \exists\, c \in Q^+,\ \exists\, n_0 \in \mathbb{N} : \forall n \geq n_0, 0 \leq f(n) < c \cdot g(n)\}$$

- ω-notation: denota un limite inferiore (lower bound) non asintoticamente stretto:

$$\omega(g(n)) = \{f(n) \mid \exists\, c \in Q^+,\ \exists\, n_0 \in \mathbb{N} : \forall n \geq n_0, 0 \leq c \cdot g(n) < f(n)\}$$

Proprietà relative alle notazioni asintotiche

- Proprietà **simmetrica**: $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$
- Proprietà riflessive:

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

$$f(n) = \Theta(f(n))$$

Algoritmi di ordinamento iterativi

- **Selection sort**: Scorro l'array n volte alla ricerca dell'elemento minimo (massimo) e ad ogni iterazione lo posiziono come primo (ultimo) elemento dell'array, continuo iterativamente con il sottoarray. Complessità nel worst case e nel best case $O(n^2)$: infatti, anche se l'array risulta già ordinato devo comunque eseguire tutte le iterazioni alla ricerca del minimo.
- **Bubble sort**: Controllo e confronto gli elementi dell'array a due a due scambiandoli di posto se non sono correttamente ordinati, itero n volte su tutto l'array. Complessità nel worst case $O(n^2)$ e nel best case $O(n)$: nel caso in cui l'array sia già ordinato infatti basterà una sola verifica per appurare che non sono necessari ulteriori scambi. Se l'array è ordinato al contrario dovrò invece eseguire n scansioni con n scambi in ognuna.
- **Insertion sort**: Divido l'array in sotto-array (destro e sinistro). Inizialmente, la parte ordinata contiene solo il primo elemento e la parte non ordinata contiene il resto degli elementi. L'algoritmo prende uno per uno gli elementi della parte non ordinata e li inserisce al posto corretto nella parte ordinata dell'array. Complessità nel worst case $O(n^2)$ (o $O(n \cdot k)$ dipendentemente dall'input) e nel best case $O(n)$: se l'array è già ordinato, basta un confronto per ogni elemento. Se l'array è ordinato al contrario, ogni elemento deve essere confrontato e spostato fino alla posizione corretta.

Gli algoritmi di ordinamento presentano alcune caratteristiche:

- **Adattività**: La performance dell'algoritmo dipende anche dall'input iniziale ed è possibile calcolarne la complessità in funzione di una variabile k.

- **Ordinamento in loco**: Non è richiesta una memoria ausiliaria.
- **Stabilità**: Mantiene l'ordine relativo degli elementi con chiavi uguali, ovvero, se due elementi hanno lo stesso valore, il loro ordine nell'array finale sarà lo stesso che avevano nell'array iniziale.

Selection sort: in loco; **Bubble sort**: in loco, stabile; **Insertion sort**: adattivo, in loco, stabile;

Ricorsione

La ricorsione è una tecnica di programmazione che permette di **chiamare una funzione all'interno della definizione della funzione stessa**. La ricorsione si presta molto bene alla risoluzione di problemi scomponibili in problemi più piccoli, è dunque **necessaria** la presenza di almeno un **sottoproblema** ma anche di una **soluzione** idealmente semplice al problema di dimensione minore, che chiameremo **caso base**.

Nonostante l'eleganza di una soluzione ricorsiva rispetto a una soluzione iterativa, la ricorsione richiede molte più risorse ad ogni chiamata ricorsiva.

Algoritmi di ordinamento ricorsivi

- **Quick Sort**: Algoritmo non adattivo, in loco, non stabile.
- **Merge Sort**: Algoritmo non adattivo, non in loco, stabile.

Calcolo del fattoriale

Uno dei problemi ricorsivi più tipici è quello del calcolo del **fattoriale**:

$$f(n) = n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot (n-1) \cdot n$$

Ricorsivamente scriveremo:

$$f(n) = \begin{cases} 1 \; se \; n = 1 \\ n \cdot f(n-1) \end{cases}$$

```
1. fattoriale(n)
2.   if n = 1 then return 1
3.   return fattoriale(n-1) * n
```

La **forma iterativa** dello stesso algoritmo sarà la seguente:

```
1. fattoriale(n)
2.   for i = 2 to n do
3.     p = p * i
4.     return p
```

Moltiplicazione

```
1. mul(x,y)
2.   if y = 1 then return x
3.   if y = 0 then return 0
4.   return mul(x, y-1) + x
```

Serie di Fibonacci

$$f(n) = \begin{cases} 1 \; se \; n \leq 2 \\ f(n-1) + f(n-2) \; se \; n > 2 \end{cases}$$

```
1. fib(n)
2.   if n <= 2 then return 1
```

```
3.   if y = 0 return 0
4.   return fib(n-1) + fib(n-2)
```

Il problema della soluzione ricorsiva al calcolo della serie di Fibonacci sta nel fatto che i sottoproblemi vengono ricalcolati più volte, rendendo la complessità dell'operazione esponenziale.

Una possibile soluzione è quella di implementare una ricorsione con memorizzazione utilizzando un array.

Equazioni di ricorrenza

L'approccio *divide et impera* è un approccio di risoluzione ai problemi suddivisibili in sottoproblemi, ed è alla base di molte delle soluzioni che studieremo. Il sottoproblema più piccolo si dirà **caso base**, ed è risolvibile in **tempo costante**.

Introduzione alle equazioni di ricorrenza

Un'**equazione di ricorrenza** è una formula che descrive il valore di una funzione, avvalendosi della funzione stessa su argomenti presumibilmente più piccoli.

Indichiamo convenzionalmente un'equazione di ricorrenza con $T(n)$.

Supponiamo che l'algoritmo ricorsivo considerato divida il problema di dimensione n in a sottoproblemi di dimensione $\frac{n}{b}$. Quindi, a è il numero di sottoproblemi, b è il fattore di riduzione, ovvero ogni sottoproblema avrà dimensione $\frac{n}{b}$.

$$T(n) = \begin{cases} D(n) + aT(\frac{n}{b}) + C(n) \forall n \geq n_0 \\ \Theta(1) \forall n < n_0 \end{cases}$$

In cui: $D(n)$ è la complessità della fase di **divisione** o riduzione del problema, $aT(n)$ rappresenta la complessità di risolvere a sottoproblemi, ognuno di dimensione ridotta $\frac{n}{b}$, $C(n)$ è la complessità della **combinazione** delle soluzioni dei sottoproblemi. La seconda parte dell'equazione, $\Theta(1) \forall n < n_0$, indica invece il **caso base**, ovvero il sottoproblema minimo.

Calcolare un'equazione di ricorrenza

Esistono quattro approcci differenti per calcolare la complessità di un algoritmo ricorsivo:

- Metodo di sostituzione;
- Metodo dell'albero di ricorsione;
- Metodo Master;
- Metodo Akra-Bazzi (che non tratteremo).

Metodo dell'albero di ricorsione

Questo metodo rappresenta il problema come un albero di cui ogni nodo corrisponde ad una chiamata ricorsiva, e i figli di un nodo rappresentano i sottoproblemi in cui il problema è stato diviso. Si può dunque dire che ogni sottoalbero rappresenta un sottoproblema.

Esempio su Merge Sort

Prendiamo come caso di studio l'algoritmo Merge Sort.

Partiamo dall'equazione di base $T(n) = D(n) + aT(\frac{n}{b}) + C(n)$ e applichiamola al nostro algoritmo:

- $D(n) = \Theta(1)$ è il costo della divisione dell'array, che è sempre **costante**;
- $a = 2, b = 2$, poiché dividiamo l'array in due sotto-array di dimensione $n/2$ ad ogni chiamata;
- $C(n) = \Theta(n)$ è il costo di combinazione, che è **lineare**.

Costruiamo quindi l'equazione:

$$T(1) = \Theta(n) + 2T(\frac{n}{2}) + \Theta(n) = 2T(\frac{n}{2}) + \Theta(n)$$

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) \ \forall n \geq 2 \\ \Theta(1) \ n = 1 \end{cases}$$

Se vogliamo quindi costruire un albero di ricorrenza, procederemo in questo modo:

1. Al primo livello avremo il problema di costo $c(n)$, che si espanderà in due sottoproblemi $c(\frac{n}{2})$;
2. Al secondo livello avremo i due sottoproblemi di costo $c(\frac{n}{2})$ che si espanderanno a loro volta in quattro sottoproblemi di costo $c(\frac{n}{4})$.
3. La suddivisione procede fino a raggiungere il caso base.

Alla fine della procedura, l'albero avrà h livelli, e ogni livello i dell'albero avrà dimensione $T(\frac{n}{2^i})$.

Sappiamo dunque che il livello h avrà dimensione $T(\frac{n}{2^h})$, ma anche che l'ultimo livello dell'albero deve avere dimensione 1 (caso base), dunque possiamo impostare l'equazione:

$$h : T(\frac{n}{2^h}) = T(1) \rightarrow \frac{n}{2^h} = 1 \rightarrow n = 2^h \rightarrow h = \log_2 n$$

L'altezza dell'albero sarà pari a $\log_2 n$, poiché ad ogni livello il problema viene diviso in due sottoproblemi fino a raggiungere i problemi di dimensione 1.

Per ricavare dunque la complessità dell'algoritmo basterà tenere in considerazione l'altezza dell'albero h e la complessità di ogni livello $C(n)$ (che per il Merge Sort, corrisponde ad $O(n)$, ovvero il costo di unire (merge) i sotto-array ordinati):

$$T(n) = h \cdot c(n) + 1 = \log n \cdot O(n) = O(n \log n) = \Theta(n \log n)$$

Metodo di sostituzione

Il **metodo di sostituzione** si basa sull'idea di indovinare una soluzione (guess), in genere una funzione asintotica, e poi verificare che la soluzione proposta soddisfi l'equazione ricorsiva attraverso il procedimento di **induzione matematica**.

Esempio su Merge Sort

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$$

Un'ipotesi comune per le equazioni che sembrano avere valore logaritmico è:

$$T(n) = O(n \log n)$$

Quindi ipotizziamo la nostra soluzione.

Guess:

$$T(n) \leq cn \log n$$

Dove c è una costante da determinare, più precisamente:

$$\exists\, c > 0, \exists\, n_0 : \forall\, n \geq n_0$$

Caso base:

$$n < n_0, T(n) = \Theta(1)$$

Passo induttivo:

Assumiamo che

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

Adesso sostituiamo la nostra ipotesi all'equazione originale:

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + dn \leq$$

$$\leq 2(c \lfloor \frac{n}{2} \rfloor \log \lfloor \frac{n}{2} \rfloor) + dn \leq$$

$$\leq 2c\frac{n}{2}\log\frac{n}{2} + dn = cn(\log n - \log 2) + dn = cn\log n - cn + dn \leq$$

$$cn\log n$$

Metodo Master

Il metodo Master è applicabile a tutte le ricorrenze che si presentano nella forma Master, ovvero:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Con $a > 0$ (numero di sottoproblemi in cui viene suddiviso il problema ad ogni chiamata ricorsiva) e $b > 1$ (fattore di riduzione della dimensione del problema) **costanti**.

La formula può essere suddivisa in due funzioni:

- **Driving function**: $f(n)$, la parte non ricorsiva della formula;
- **Watershed function**: $\omega(n) = n^{\log_b a}$;

Master Theorem

Siano $a > 0, b > 1$ costanti, sia $f(n)$ una funzione definita e non negativa su tutti i numeri reali sufficientemente grandi.

Sia

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Allora:

1. $\omega(n)$ cresce in maniera asintoticamente più veloce rispetto a $f(n)$, ed esse differiscono per un polinomio $\Theta(n^\epsilon)$. In altre parole esiste un ϵ per cui $\omega(n)$ è più grande di $f(n)$. In questo caso, la complessità totale sarà dominata da $\omega(n)$ e la soluzione sarà $\Theta(n^{\log_b a})$:

$$\exists\, \epsilon > 0 : f(n) = O(n^{\log_b a - \epsilon}) \implies T(n) = \Theta(n^{\log_b a})$$

2. $\omega(n)$ cresce in maniera asintoticamente simile a $f(n)$ o esse differiscono al massimo per un fattore polilogaritmico $\Theta(\log^k n)$. In questo caso dunque la complessità finale sarà $\Theta(n^{\log_b a} \cdot \log^{k+1} n)$.

$$\exists\, k \geq 0 : f(n) = \Theta(n^{\log_b a}\log^k n) \implies T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$$

3. $\omega(n)$ cresce in maniera asintoticamente più lenta di $f(n)$ ed esse differiscono per un fattore polinomiale $\Theta(n^\epsilon)$. In questo caso la complessità finale sarà $T(n) = \Theta(f(n))$.

$$\exists\, \epsilon > 0 : f(n) = \Omega(n^{\log_b a + \epsilon}) + f(n)$$

e soddisfa la condizione di regolarità:

$$\exists\, c < 1 : af(\frac{n}{b}) \leq cf(n)$$

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

Esempio

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

Questa funzione si presenta in forma **Master**, possiamo dunque applicare il metodo Master:

- $a = 1$ (numero di sottoproblemi);
- $b = 2$ (fattore di riduzione: ogni sottoproblema avrà dimensione $\frac{n}{2}$ del problema di origine);
- $f(n) = \log n$;

- $\omega(n) = n^{\log_2 1} = 1$

La differenza asintotica tra le due funzioni non è polinomiale ma **polilogaritmica**, ci troviamo quindi nel caso 2. Dunque:

$$f(n) = \Theta(n^{\log_2 1} \cdot \log^k n) = \Theta(1 \cdot \log^k n)$$

Dunque:

$$f(n) = \log n = 1 \cdot \log^k n \implies k = 1$$

E quindi procediamo in questo modo:

$$T(n) = \Theta(\log^{k+1} n) = \Theta(\log^{1+1} n) = \Theta(\log^2 n)$$

a. $T(n) = O(\log^2 n)$? La risposta è sì, se $T(n) = \Theta(\log^2 n)$ allora è anche uguale a $O(\log^2 n)$.
b. $T(n) = O(\log^3 n)$? La risposta è sì, infatti la notazione **big-**O è un upper bound.
c. $T(n) = o(\log^2 n)$? La risposta è no, poiché la notazione **little-**o si riferisce invece ad un upper bound **non asintoticamente** stretto.
d. $T(n) = o(\log^3 n)$? La risposta è sì, per lo stesso motivo della domanda precedente.

Se volessimo rappresentare l'algoritmo come albero, ogni nodo avrebbe un (a) solo figlio di dimensione $\frac{n}{2}$ ($\frac{n}{b}$). Il primo figlio avrà dunque dimensione $c(\log n)$, il secondo $c(\frac{\log n}{2})$ e così via. Ma ogni livello avrà costo = $\Theta(\log n)$, questo perché anche se la dimensione dei sottoproblemi si riduce, il **costo per risolvere ogni sottoproblema** a ogni livello è dato da $\Theta(\log n)$, e non dalla dimensione del sottoproblema. Questo accade perché il lavoro esterno alla ricorsione, $f(n)$, dipende dalla dimensione iniziale del problema.

Per trovare invece l'altezza h dell'albero, dobbiamo porre l'equazione:

$$\frac{n}{2^h} = 1 \implies h = \log n$$

Heap

L'**heap** è una struttura dati simile alla coda e serve ad implementare una coda con priorità.

Possiamo immaginare l'heap come un albero binario con le seguenti caratteristiche:

- Completo: in ogni livello i dell'albero sono presenti 2 nodi. Ricordiamo che il primo livello è sempre 0. Tuttavia nel caso specifico dell'heap, l'ultimo livello può essere non completo, ma gli elementi devono essere distribuiti da sinistra a destra;
- Parzialmente ordinato: la priorità di un nodo sarà sempre maggiore o uguale a quella dei suoi figli.

Implementazioni dell'heap

- **min-heap**: priorità più alta a valori più piccoli;
- **max-heap**: priorità più alta a valori più grandi.

Operazioni su min-heap

- Valore minimo (operazione in tempo costante, il minimo è sempre la radice);
- Estrazione del minimo;
- Insert;
- Decrease-key;
- Delete;
- Heapify;

Heapify

Operazioni che, dato in input un albero e la sua radice, scambia la posizione delle chiavi in modo da far rispettare l'ordine parziale.

```
1. heapify(H,x)
2.   y = left(x)
3.   z = righ(X)
4.   min = x
5.   if(y != NULL AND key(y) < key(x))
6.     then min = y
7.   if(y |= NULL AND key(z) < key(x))
8.     then min = z
9.   if min != x then
10.    swap(x, min)
11.    heapify(H, min)
```

La procedura di Heapify ha complessità $O(\log n)$. Un heap binario con n elementi ha infatti altezza $h = O(\log n)$. Nel caso peggiore, heapify deve attraversare $O(\log n)$ llivelli e effettuare uno scambio ad ogni livello. Ogni scambio e confronto tra elementi ha tempo costante $O(1)$, dunque la complessità totale è appunto $O(\log n)$.

Insert

```
1. insert(H, k)
2.   x = new_node(H) // x sarà il nuovo nodo
3.   key(x) = k // assegno il valore k al nodo x
4.   p = parent(x) // assegno a P il parent di x
5.   while(p != NULL AND key(p) < key(x)) do // fino alla radice
6.     swap(x,p) // faccio risalire il nodo alla posizione corretta
7.     x = p
8.     p = parent(x)
```

Complessità di $O(\log n)$, poiché nel caso peggiore devo attraversare ogni livello dell'albero.

Decrease-key

```
1. decrease_key(H, x, k)
2.   key(x) = k // assegno al nodo il nuovo valore
3.   p = parent(x) // trovo il parent di x
4.   while(p != NULL AND key(p) < key(x)) do // trovo la posizione corretta
5.     swap(x,p)
6.     x = p
7.     p = parent(x)
```

Complessità di $O(\log n)$.

Implementazione heap

Nonostante l'heap possa essere rappresentato tramite questo particolare tipo di albero binario, la sua vera implementazione avviene tramite un array.

Funzioni left, right, parent

Nel caso di heap implementato come array, possiamo notare che la posizione del figlio sinistro di ogni nodo corrisponderà sempre al doppio della posizione del parent e, di conseguenza, la posizione del figlio destro corrisponderà alla posizione successiva a quella del figlio sinistro.

Sarà dunque sufficiente eseguire una moltiplicazione per due per trovare il figlio sinistro di ogni nodo, e sommare uno per trovare invece la posizione del figlio destro. Per trovare il parent basterà invece eseguire una divisione intera.

```
1. left(i)
2.   return 2i
```

```
1. right(i)
2.   return 2i+1
```

```
1. parent(i)
2.   return floor(i/2)
```

Tuttavia questo non è il modo più efficiente di implementare queste funzioni.

Esiste infatti la possibilità di utilizzare un left-shift per la moltiplicazione per 2, un bitmasking con OR logico per la somma (+1) e un right-shift per la divisione intera per 2. A differenza delle operazioni classiche infatti, questo tipo di operazioni richiedono meno cicli di clock per essere eseguite.

```
1. left(i)
2.   return (i << 1)
```

```
1. right(i)
2.   return (i << 1) || 1
```

```
1. parent(i)
2.   return (i >> 1)
```

Heapify

Vediamo adesso l'implementazione della funzione Heapify per heap come array:

```
1. heapify(A, i)
2.   l = left(i)
3.   r = right(i)
4.   min = i
5.   if(l <= heapsize AND A[l] < A[min]) then
6.     min = l
7.   if(r <= heapsize AND A[r] < A[min]) then
8.     min = r
9.   if(min != i) then
10.    swap(A, i, min)
11.    heapify(A, min)
```

Estrazione del minimo/massimo

Invece di rimuovere la radice e ricollegare l'albero, scambio la radice (elemento minimo/massimo) con l'ultimo nodo dell'albero (da sinistra) e rimuovo il nodo. Chiamo poi Heapify sulla radice per ordinare l'albero. La complessità è dunque $O(\log n)$, ovvero $O(1)$ per lo scambio e $O(\log n)$ per la procedura di Heapify.

```
1. extractmin(A)
2.   swap(A, 1, heapsize)
3.   heapsize = heapsize - 1
4.   heapify(A, 1)
5.   return A[heapsize + 1]
```

Inserimento

```
1. insert(A, k)
2.   heapsize = heapsize + 1
3.   A[heapsize] = k
4.   i = heapsize
5.   p = parent(i)
```

```
6.   while(p != 0 AND A[p] > A[1]) do
7.     swap(A, i, p)
8.     i = p
9.     p = parent(i)
```

Costruzione di un heap

L'heap è una struttura dati di tipo ricorsivo costituita da sotto-alberi, dunque anche in un albero binario che non rispetta l'ordine di un heap ogni foglia è da considerarsi un heap. Sapendo questo, è possibile usare la funzione Heapify dai parent delle foglie fino alla radice per rendere qualunque albero binario un heap. Creiamo quindi la funzione build-min-heap:

```
1. build-min-heap
2.   for(i = n/2 down to 1) do
3.   heapify(A, i)
```

La complessità della procedura di build-heap è proporzionale a $O(n)$: questo perché la maggior parte dei nodi si trovano vicino alle foglie dell'albero e quindi l'operazione di Heapify è mediamente più veloce.

Heapsort

```
1. heapsort(A,n)
2.   build-max-heap(A,n)
3.   for(i = 1 to n) do
4.     extract-max(A)
```

La complessità della procedura di Heapsort invece è di $O(n \log n)$: inizialmente dobbiamo utilizzare infatti la funzione build-heap che abbiamo visto avere complessità $O(n)$ e successivamente eseguire n estrazioni (seguite da Heapify) e dunque n volte una procedura con complessità di $O(\log n)$.

Tabelle Hash

Le **tabelle Hash** sono strutture dati estremamente **veloci** ed **efficienti**. Le principali operazioni che vengono eseguite su queste particolari strutture dati sono la ricerca, l'inserimento e la rimozione.

La ricerca viene generalmente suddivisa in due casi:

- **Ricerca con successo**: stiamo cercando un elemento che sappiamo essere presente nella tabella;
- **Ricerca senza successo**: non troviamo l'elemento che cerchiamo, dunque sicuramente non è presente.

Immaginiamo dunque una struttura dati che ci comunica immediatamente se un elemento è presente o meno.

Una tabella Hash può essere implementata in modi diversi:

- Come **array ordinato**: in questo caso, inserimento e cancellazione avranno complessità $O(n)$ perché una volta inserito o eliminato l'elemento, tutti gli elementi vanno spostati mantenendo l'ordine, mentre la ricerca avrà complessità $O(\log n)$, sfruttando la ricerca dicotomica;
- Come **array non ordinato**: inserimento, così come la cancellazione, avrà tempo costante $O(1)$, in quanto non sarà necessario spostare gli altri elementi. La ricerca manterrà una complessità di $O(n)$;
- Come **lista ordinata**: inserimento e ricerca $O(n)$, cancellazione $O(1)$;
- Come **lista non ordinata**: inserimento e cancellazione $O(1)$, ricerca $O(n)$;
- Come **BST bilanciato**: in questo caso, tutte le operazioni avranno complessità $O(\log n)$;

Tuttavia, è anche possibile utilizzare una **tabella a indirizzamento diretto.**

Tabella a indirizzamento diretto

Sia U l'insieme di **tutte le chiavi rappresentabili** ed S l'insieme delle **chiavi effettivamente presenti** nella tabella. Creiamo una tabella T, di cardinalità m, in cui ogni posizione corrisponde ad una chiave in U. Se una chiave $k \in S$ è presente, allora poniamo $T[k] = 1$, altrimenti poniamo $T[k] = 0$.

Operazioni di base:

```
1. insert(T, k)
2.   T[k] = 1
```

```
1. delete(T, k)
2.   T[k] = 0
```

```
1. search(T, k)
2.   if T(k) = 1 then
3.     return true
4.   return false
```

In questo tipo di tabella, dunque, inserimento, cancellazione e ricerca sono operazioni eseguite in **tempo costante** $O(1)$.

La tabella a indirizzamento diretto presenta tuttavia un'importante limitazione:

Se l'insieme delle chiavi U è molto grande, la tabella T richiederà molta memoria, poiché verrà allocato spazio anche per le chiavi non effettivamente presenti.

Per risolvere questo problema, utilizziamo una **funzione di hashing** che mappa le chiavi da uno spazio grande come U ad uno più piccolo:

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

Le nostre funzioni diventano di conseguenza:

```
1. insert(T, k)
2.   T[h(k)] = 1
```

```
1. delete(T, k)
2.   T[h(k)] = 0
```

```
1. search(T, k)
2.   if T[h(k)] = 1 then
3.     return true
4.   return false
```

Tuttavia, quando utilizziamo una funzione di hashing, può capitare che due chiavi k_1 e k_2 generino un indice identico, $h(k_1) = h(k_2)$. Tale fenomeno prende il nome di **collisione** tra due chiavi ed è inevitabile, in quanto il dominio di partenza è più grande del codominio della funzione. Esistono comunque due modi di risolvere una collisione:

- Risoluzione per **concatenazione**;
- Risoluzione per **indirizzamento aperto**.

Risoluzione per concatenazione

Se k_1 e k_2 hanno la stessa immagine per h, facciamo in modo che ogni posizione della tabella non memorizzi solamente un elemento ma una lista di elementi:

```
1. insert(T, k)
2.   list-insert(T[h(k)], k)
```

```
1. delete(t, k)
2.   list-delete(T[h(k)], k)
```

```
1. search(T, k)
2.   return list-search(T[h(k)], k)
```

Come sappiamo, il problema della ricerca all'interno di una lista è proporzionale alla lunghezza della lista stessa. Il caso peggiore, è che tutti gli $|S|$ elementi siano indirizzati nella stessa posizione della tabella, in questo caso la complessità sarà di $O(n)$. Tuttavia, una **tabella hash** offre complessità costante nel **caso medio**, ovvero nel caso in cui la funzione riesca a distribuire gli elementi in maniera uniforme sulla tabella.

Hashing uniforme semplice

Per ottenere il caso medio $O(1)$, è necessario che la probabilità che una chiave sia indirizzata a una qualsiasi cella i sia la stessa per tutte le celle della tabella. Si parla in questo caso di **hashing uniforme semplice**.

Un hashing si dice uniforme semplice se soddisfa la seguente proprietà:

$$P_i\{h(k) = i\} = \frac{1}{m}$$

Dove m è il numero totale di celle della tabella.

Introduciamo adesso un fattore α noto come **fattore di carico**:

$$\alpha = \frac{n}{m}$$

Dove m, come abbiamo detto, è il numero di celle, mentre n è il numero di elementi. α esprime dunque in proporzione **quanto è carica** la tabella. Idealmente, vogliamo che α sia circa uguale a 1, ovvero che il numero di elementi sia comparabile al numero di celle, in modo da non avere celle "sprecate" o collisioni. Nel caso medio ogni cella conterrà α elementi, e la complessità media della ricerca sarà dunque $O(\alpha)$.

Implementazione delle funzioni di hashing

Esistono due principali metodi di hashing:

- metodo della **divisione**;
- metodo della **moltiplicazione**.

Metodo della divisione

Nel metodo della divisione, si definisce la funzione di hashing come:

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

Dove k è la chiave da inserire ed m è il numero di celle della tabella. Per evitare collisioni, è preferibile scegliere un valore m che sia un numero primo.

Immaginiamo infatti di scegliere $m = 10$. In questo caso, valori che presentano la stessa cifra delle unità, finirebbero per collidere nella stessa posizione. Se scegliessimo invece una m potenza di 2, ad esempio $m = 2^p$, l'operazione di modulo darebbe come risultato una posizione basata solo sugli ultimi p bit dei valori.

Metodo della moltiplicazione (da completare)

Il **metodo della moltiplicazione** cerca di ottenere una distribuzione uniforme delle chiavi utilizzando una costante A (con $0 < A < 1$) e calcolando:

$$h(k) = \lfloor m \cdot (kA \mod 1) \rfloor$$

dove $kA \mod 1$ rappresenta la parte decimale di kA, un valore compreso tra 0 e 1.

...

Funzioni uniformi e indipendenti

Una **funzione hash uniforme e indipendente** è tale che $\forall k \in U, h(k)$ è scelto in maniera uniforme e indipendente da $\{0, 1, \dots, m-1\}$.

- **Uniforme**:

$$\forall k \in U, \forall j \in \{0, 1, ..., m-1\}, Pr[h(k) = j] = \frac{1}{m}$$

- **Indipendente**:

$$\forall k_1, k_2 \in U, k_1 \neq k_2, \forall j_1, j_2 \in \{0, 1, ..., m-1\}$$

$$Pr[h(k_1) = j_1 | h(k_2) = j_2] = Pr[h(k_1) = j_1]$$

In altre parole, una funzione di hashing si dice **uniforme** se ogni posizione associata ad ogni chiave k ha la stessa probabilità $\frac{1}{m}$ di essere associata ad una delle m posizioni, si dice **indipendente** se la probabilità di associare una posizione a una chiave k_1 è indipendente dalla probabilità di associare un'altra posizione ad una chiave k_2.

Le funzioni di hashing uniformi e indipendenti sono tuttavia un'astrazione teorica, sono infatti difficili da implementare nella pratica. In ogni caso, è possibile avere funzioni di hashing che sono approssimativamente uniformi e indipendenti.

Tabella a indirizzamento aperto

In una tabella hash a indirizzamento aperto, ogni elemento della tabella contiene o un elemento dell'insieme dinamico o `NULL`.

Le tabelle a indirizzamento aperto possono essere forzate ad avere un fattore di carico α **non superiore** a 1.

Quando un elemento deve essere inserito nella tabella, viene allocato nella posizione "prima scelta". Se la posizione in questione è gia occupata, il nuovo elemento è inserito nella posizione di "seconda scelta", e così via.

Per cercare un elemento, dunque, è necessario **esaminare sistematicamente** gli slot nell'ordine "preferito" relativo all'elemento stesso finché non si trova l'elemento cercato o una **cella vuota**. Nel caso di cella vuota, infatti, saremo certi che l'elemento non sarà neanche in alcuna delle celle successive. Il vantaggio dell'indirizzamento aperto risiede nel poter evitare del tutto i puntatori.

Nell'analisi degli algoritmi di ricerca su questo tipo di struttura, assumeremo che la funzione di hashing sia un **hashing di permutazione indipendente e uniforme**.

Hashing di permutazione

Una **funzione di hashing di permutazione** su una chiave restituisce una serie di posizioni, ovvero una permutazione delle m celle della tabella hash:

$$h(k) = < h(k, 0), h(k, 1), ..., h(k, m-1) > = \sigma \in S_m$$

Con S_m insieme delle permutazioni di m.

Esamineremo dunque gli slot dal primo all'ultimo, inserendo la chiave nel primo slot (cella) disponibile.

Perché un hashing di permutazione sia **uniforme e indipendente**, è necessario che $\forall k \in U, h(k)$ sia scelto in maniera uniforme e indipendente da $\{\sigma_1, \dots, \sigma_{m!}\}$. Nello specifico:

- **Uniforme**:

$$\forall k \in U, \forall \sigma \in \{\sigma_1, ..., \sigma_{m!}\}, Pr[h(k) = \sigma] = \frac{1}{m!}$$

- **Indipendente**:

$$\forall k_1, k_2 \in U, k_1 \neq k_2, \forall \sigma_1, \sigma_2 \in \{\sigma_1, ..., \sigma_{m!}\}$$

$$Pr[h(k_1) = \sigma_1 | h(k_2) = \sigma_2] = Pr[h(k_1) = \sigma_1]$$

Assumiamo adesso la funzione hash

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

$$k \to h(k) \to T[h(k)]$$

Sappiamo che il fattore di carico di una tabella hash è $\alpha = \frac{n}{m}$. Poniamo $n_j = T[j]$, con $j \in \{0, \dots, m-1\}$, avremo che $n = n_0 + n_1 + \dots + n_{m-1}$. Se l'hashing è uniforme e indipendente:

$$E[n_j] = \alpha$$

Ovvero, il valore atteso del numero di elementi in una cella di una tabella hash con concatenazione corrisponde al fattore di di carico α.

Assumiamo anche che calcolare $h(k)$ e accedere a $T[h(k)]$ necessiti tempo costante $\Theta(1)$.

Ricerca senza successo

Il valore atteso del tempo computazionale di una ricerca senza successo su una tabella hash con concatenazione è $\Theta(1+\alpha)$.

Dimostriamolo:

Il tempo totale per la computazione sarà ottenuto dal tempo necessario per calcolare $h(k)$ e accedere a $T[h(k)]$ più il tempo per scorrere tutta la lista $T[h(k)]$, dunque:

$$E[Tempo] = \Theta(1) + E[n_{h(k)}] = \Theta(1) + \sum_{j=0}^{m-1} \frac{1}{m} n_j = \Theta(1) + \frac{1}{m} \sum_{j=0}^{m-1} n_j =$$

$$= \Theta(1) + \frac{1}{m} n = \Theta(1) + \alpha = \Theta(1+\alpha)$$

Ricerca con successo

Una ricerca con successo su una tabella hash su cui le collisioni sono risolte per concatenazione è $\Theta(1+\alpha)$.

Dimostriamolo:

L'elemento x che stiamo cercando è memorizzato nella tabella.

Supponiamo di memorizzare i nuovi elementi in testa alla lista collegata che gli corrisponde: se x si trova in $T[h(k)]$, dobbiamo scorrere tutti gli elementi che sono stati memorizzati dopo x.

Sia x_i l'i-esimo elemento ad essere stato inserito nella tabella, $k_i = x_i.key$ ovvero la chiave dell'elemento i-esimo.

Definiamo $\forall q \in \{0, \dots, m-1\}$, $\forall k_i = k_j$ con $k_i \neq k_j$.

Sia

$$X_{ijq} = \begin{cases} 1 \; se \; stiamo \; cercando \; x_i, h(k_i) = q, h(k_j) = q \\ 0 \; altrimenti \end{cases}$$

In altre parole, definiamo X_{ijq} in modo che, per ogni coppia di chiavi diverse k_i, k_j, serva a rappresentare se c'è una collisione tra le chiavi ($h(k_i) = q, h(k_j) = q$) nella cella q. Dunque X_{ijq} assumerà valore 1 se avviene una collisione tra due le chiavi diverse, 0 altrimenti.

$$E[X_{ijq}] = 0Pr[X_{ijq} = 0] + 1Pr[X_{ijq} = 1] = Pr[X_{ijq} = 1] =$$

$$= Pr[stiamo \; cercando \; x_i] Pr[h(k_i) = q] Pr[h(k_j) = q] = \frac{1}{n} \cdot \frac{1}{m} \cdot \frac{1}{m} = \frac{1}{n \cdot m^2}$$

Sia

$$Y_j = \begin{cases} 1 \; se \; x_j \; appare \; prima \; dell'elemento \; cercato \\ 0 \; altrimenti \end{cases}$$

$$Y_j = \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}$$

In altre parole, Y_j rappresenta quante chiavi sono già state inserite nella stessa cella prima dell'elemento che stiamo cercando.

Infine, sia

$$Z = \sum_{j=1}^{n} Y_j$$

Quindi, Z rappresenta il numero totale di collisioni in tutte le celle della tabella.

Dunque:

$$E[Tempo] = \Theta(1) + E[Z+1] = \Theta(1) + E[Z] =$$

$$= \Theta(1) + E[\sum_{j=1}^{n} Y_j] = \Theta(1) + \sum_{j=1}^{n} E[Y_j] =$$

$$= \Theta(1) + \sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} E[X_{ijq}] = \Theta(1) + \sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} \frac{1}{n \cdot m^2} =$$

$$= \Theta(1) + \frac{1}{n \cdot m^2} \sum_{j=1}^{n} \sum_{q=0}^{m-1} (\sum_{i=1}^{j-1} 1) = \Theta(1) + \frac{1}{n \cdot m^2} \sum_{q=0}^{m-1} \sum_{j=1}^{n} (j-1) =$$

$$= \Theta(1) + \frac{1}{n \cdot m^2} \sum_{q=0}^{m-1} \sum_{j=1}^{n-1} j = \Theta(1) + \frac{1}{n \cdot m^2} \sum_{q=0}^{m-1} \frac{(n-1)n}{2} =$$

$$= \Theta(1) + \frac{1}{n \cdot m^2} \frac{n(n-1)}{2} m = \Theta(1) + \frac{n}{2m} - \frac{1}{2m} =$$

$$= \Theta(1) + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1 + \alpha)$$

Implementazioni per tabelle a indirizzamento aperto

```
1. hash-insert(T, k)
2.   if(n = m) return
3.   i = 0
4.   while(T[h(k, i)] != null) // AND T[h(k, i)] != D
5.     i++
6.   T[h(k, i)] = k
7.   n++
```

La complessità della procedura è $O(n)$.

```
1. hash-search(T, k)
2.   i = 0
3.   while(i < m AND T[h(k, i)] != null)
4.     if(T[h(k, i)] = k
5.       return true
6.     i++
7.   return false
```

A differenza della ricerca su un array, la ricerca su una tabella hash mi permette di verificare la presenza di un elemento in minor tempo, poiché **la procedura si interrompe alla prima occorrenza di una cella vuota**, con la certezza che l'elemento cercato non sarà neanche nelle celle successive. Dunque, questo tipo di struttura permette di avere un caso medio migliore, pur mantenendo complessità $O(n)$ nel caso pessimo.

Chiaramente, questa procedura porterebbe a delle ambiguità nel caso in cui, cancellando un elemento dalla tabella, la cella venisse posta uguale a `NULL`.

La soluzione a questo problema è quella di indicare gli elementi rimossi come **deleted** evitando di assegnare la cella a `NULL`.

Il numero di ispezioni medio prima di trovare un elemento sarà di $\frac{1}{1-\alpha}$. Se la tabella fosse ad esempio piena a metà, ovvero con $\alpha = 0,5$, impiegheremmo in media 2 ispezioni per trovare l'elemento cercato. Se α fosse $0,9$, ovvero se la tabella fosse quasi del tutto piena, impiegheremmo circa 10 ispezioni per trovare l'elemento. Anche in una ricerca senza successo, come abbiamo visto, il numero di ispezioni risulterebbe comunque basso.

Probabilità in una tabella a indirizzamento aperto

$$Pr\{h(k) = i\} = \frac{1}{m}$$

Ovvero, avremo una possibilità su m che la funzione di hashing assegni alla chiave k una posizione i.

Indichiamo con P_m l'insieme di tutte le permutazioni sull'insieme $\{0, \dots, m-1\}$, allora sapremo che $|P_m| = m!$.

Dunque:

$$Pr\{h(k) = 1\} = \frac{1}{m!}$$

Scansione lineare

Supponiamo di avere una tabella con 10 celle, dunque $m = 10$, e definiamo la nostra funzione ausiliaria:

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

Immaginiamo di dover calcolare la funzione per $k = 13$:

$$h'(13) = 13 \mod 10 = 3$$

Calcoliamola adesso per $k = 63$:

$$h'(63) = 63 \mod 10 = 3$$

Ma la posizione 3 è già occupata dalla chiave 13, dunque:

$$(h'(63) + i) \mod 10 = (h'(63) + 1) \mod 10 = 4$$

La posizione 4 è libera, ma se fosse occupata anche questa:

$$(h'(g3) + 2) \mod 10 = 5$$

Se tutte le posizioni successive fossero comunque occupate, dopo l'ottavo tentativo tornerei alla posizione 0, infatti:

$$(h'(63) + 7) \mod 10 = 0$$

Al numero 72 ad esempio, sarà associata la sequenza $< 2, 3, 4, 5, 6, 7, 8, 9, 0, 1 >$, al numero 7 la sequenza $< 7, 8, 9, 0, 1, 2, 3, 4, 5, 6 >$. Per posizioni iniziali diverse, dunque, otterremo scansioni diverse. Avrò quindi m, ma non $m!$, diverse sequenze, poiché non tutte le permutazioni sono equiprobabili.

Questo tipo di scansione soffre inoltre del **fenomeno dell'agglomerazione primaria**, ovvero i blocchi più grandi di elementi adiacenti tenderanno a diventare sempre più grandi, a discapito degli elementi più isolati.

Questo fenomeno si verifica perché la funzione della scansione lineare avrà più probabilità di raggiungere una cella vuota dopo aver calcolato una cella occupata, dunque se disporremo di più celle occupate adiacenti, la probabilità di trovare la cella libera subito al di sotto del blocco non sarà più solamente $\frac{1}{m}$. Nel caso della nostra tabella con 10 celle, una cella libera al di sotto di un blocco di 4 celle occupate avrà $\frac{5}{10}$ probabilità di essere raggiunta.

Scansione quadratica

Per ovviare a questo problema, oltre alla scansione lineare esiste anche la **scansione quadratica**:

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

In questa definizione, è importante individuare i valori c_1 e c_2 che permettono di scansionare la tabella per intero. Tuttavia, questo approccio **non risolve il problema delle permutazioni**.

Hashing doppio

Se vogliamo provare a risolvere il problema delle permutazioni, dovremo ricorrere all'approccio dell'hashing doppio, che utilizza due funzioni ausiliarie:

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

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

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

Questo approccio riuscirà a generare m^2 permutazioni, che rimane comunque un numero molto più piccolo di $m!$.

Analisi dell'efficienza degli algoritmi di ricerca nelle tabelle a indirizzamento aperto

In una tabella hash a indirizzamento aperto, ogni cella della tabella contiene un elemento del nostro insieme universo (o insieme dinamico) U delle chiavi, oppure è vuota.

Il fattore di carico non può quindi essere maggiore di 1.

Sappiamo che la funzione di hashing di permutazione su una chiave fornisce una serie di posizioni, ovvero una permutazione σ delle m celle di una tabella:

$$h(k) = < h(k,0), h(k,1), ..., h(k, m-1) > = \sigma \in S_m$$

Rispettivamente posizione di prima scelta, di seconda scelta, fino alla posizione di scelta $m-1$.

Assumiamo che l'hashing di permutazione sia:

- **Uniforme**: $\forall k, \forall \sigma \in S_m,\ Pr[h(k) = \sigma] = \frac{1}{m!}$
- **Indipendente**: $Pr[h(k_1) = \sigma_1 | h(k-2) = \sigma_2] = Pr[h(k_1)) = \sigma_1]$
  $Pr[h(k_1) = \sigma_1 \wedge h(k_2) = \sigma_2] = Pr[h(k_1) = \sigma_1] Pr[h(k_2) = \sigma_2]$

Assumiamo anche che $\alpha \leq 1$, ovvero che ogni cella sia occupata al più da una chiave.

Analisi della ricerca senza successo (da completare)

Sia data una tabella hash in cui le collisioni sono risolte mediante indirizzamento aperto, con fattore di carico $\alpha < 1$. Il numero di prove delle celle è al più $\frac{1}{1-\alpha}$, quindi il tempo computazionale atteso è $O(\frac{1}{1-\alpha})$.

Dimostriamolo:

Ogni prova incontra una cella occupata, eccetto l'ultima.

$$T(n) = \Theta(1) + E[X] = E[X]$$

Con X: numero di prove in una ricerca senza successo.

...

Red-Black Trees

I **Red-Black Trees** sono alberi binari che devono rispettare particolari proprietà:

- Ogni nodo può essere o rosso o nero;
- La radice è dell'albero è sempre nera;
- Un nodo rosso deve può avere solo figli neri;
- Un cammino da un qualsiasi nodo a una sua foglia ha sempre lo stesso numero di nodi neri;
- Le foglie sono sempre nere;

Se le regole sono rispettate, allora il RB-Tree è bilanciato, ovvero la sua altezza è $h = \Theta(\log n)$, questo assicura, come negli alberi binari di ricerca, che per la ricerca impiegheremo al massimo $O(\log n)$.

Sappiamo inoltre che l'altezza dell'albero h è sempre compresa tra bh e $2bh$.

Le funzioni definite sui RB-Trees sono:

- Right-Rotate e Left-Rotate;
- Inserimento;
- Cancellazione;

Rotazioni

Le rotazioni sono un tipo di operazioni definite sugli RB-Trees:

- Rotazione sinistra: avviene quando un nodo x prende il suo figlio destro y come nuovo genitore. Viene utilizzata quando l'albero è sbilanciato a destra;
- Rotazione destra: è l'operazione speculare alla rotazione sinistra: un nodo y prende il suo figlio sinistro x come nuovo genitore.

Ricolorazione

La ricolorazione di un nodo, banalmente, avviene quando un nodo nero diventa rosso e viceversa.

Inserimento

Durante l'inserimento, il nuovo elemento inserito è rosso. Potrebbe succedere che le proprietà non vengano rispettate, in questo caso ricorreremo alla funzione insert-fixup per correggere e ribilanciare l'albero.

Possono presentarsi 4 casi:

- il nodo inserito, z, è la radice, in questo caso ricoloro semplicemente il nodo;
- z è esterno e ha zio nero, in questo caso si esegue una rotazione del grandparent di z nella direzione opposta a z e si ricolorano i vecchi parent e grandparent;
- z è interno e ha zio nero , in questo caso si effettua una rotazione del parent di z nella direzione opposta a z e, se necessario, si applicano gli altri casi.
- z ha zio rosso, in questo caso si ricolorano parent, grand parent e zio e si applicano gli altri casi dove necessario.

Vediamo adesso la procedura di cancellazione di un nodo dall'albero.

Prima di passare alla rimozione negli RB-Trees, rivediamo brevemente come funziona la rimozione negli alberi binari di ricerca:

- Il nodo cancellato era una foglia: in questo caso banalmente cancello il nodo;
- Il nodo cancellato ha un solo figlio: in questo caso l'unico figlio prende il posto del padre;
- Il nodo cancellato ha due figli: sostituisco il nodo eliminato con la chiave più a sinistra del suo sottoalbero destro.

Adesso possiamo passare alla rimozione su RB-Trees, indicheremo il nodo da rimuovere con z.

- z è rosso con padre nero: questo è un caso banale, è infatti sufficiente rimuovere z.
- z è rosso con un solo figlio e padre nero: in questo caso rimuovo semplicemente il nodo e l'unico figlio diventa figlio del parent di z.
- z è nero con figlio rosso: il figlio prende il posto di z e diventa nero.
- z è una foglia nera, denoto il NIL come doppio-nero e cancello il nodo;
- il nodo da cancellare è nero e ha un figlio nero: in questo caso, come nel precedente, rimuovere semplicemente il nodo e sostituirlo con il figlio sbilancerebbe l'albero perché mancherebbe un nodo nero nel cammino dalla radice alla foglia. Dunque, denoto il figlio del nodo rimosso come doppio-nero, che indica che il cammino dalla radice fino a quel nodo ha un nodo nero in meno.

Vediamo adesso come sistemare gli alberi nel caso in cui siano presenti dei nodi doppio-nero. Possono presentarsi 6 casi, ognuno dei quali ha anche un caso simmetricamente equivalente, per comodità chiameremo w il fratello del nodo doppio-nero:

- w è nero con figli neri: in questo caso ricoloro w e il parent:
  - Se il padre era già nero, diventerà doppio-nero. In questo caso, chiamo la funzione sul nuovo nodo doppio-nero.
  - Se il padre era rosso la procedura è finita.
- w è nero e uno dei suoi due figli è rosso:
  - Se il figlio esterno è rosso, ruoto il parent di w facendo diventare w radice del sottoalbero, il doppio-nero diventa nero e il figlio esterno (precedentemente rosso) diventa nero.
  - Se il figlio interno è rosso, lo ruoto verso l'esterno facendolo diventare parent di w nero, che diventerà figlio esterno rosso, ci troveremo nella situazione precedente.
- w è rosso, ruoto w facendolo salire e richiamo la funzione sul doppio-nero.
- Il nodo doppio-nero è la radice dell'albero: in questo caso diventa semplicemente nera.

Altezza degli RB-Trees

Un albero rosso-nero con n nodi interni ha altezza al più

$$h \leq 2\log_2(n+1) = O(\log n)$$

Dimostriamolo:

Claim: Il sottoalbero radicato a un nodo x contiene almeno $2^{bh(x)} - 1$ nodi interni.

Dimostriamolo per induzione:

- **Base dell'induzione**: $h(x) = 0 \implies x$ è una foglia $\implies$ il sottoalbero radicato a x ha 0 nodi interni. $0 \geq 2^0 - 1 = 1 - 1$.
- **Ipotesi induttiva**: assumiamo che il claim sia vero per qualunque nodo di altezza $h < h(x)$ per $h(x) > 0$.
- **Passo induttivo**: Siccome $h(x) > 0 \implies x$ ha due figli. Se $x.child$ è nero $\implies bh(x) = bh(x.child) + 1$. Se $x.child$ è rosso $\implies bh(x) = bh(x.child)$. Il tutto implica che $bh(x.child) \geq bh(x) - 1$. Inoltre, $h(x.child) < h(x)$, quindi possiamo applicare l'ipotesi induttiva a $x.child$. Il sottoalbero radicato a x contiene i sottoalberi radicati ai suoi figli e quindi il numero di nodi interni è $\#$ nodi interni all'albero radicato a $x.left + \#$ nodi interni all'albero radicato a $x.right + 1$. Dunque:

$$(2^{bh(x.left)} - 1) + (2^{bh(x.right)} - 1) + 1 \geq 2(2^{bh(x.child)} - 1) + 1 \geq$$

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

Adesso dimostriamo il lemma usando il claim.

Sia h^* l'altezza dell'intero albero, dobbiamo dimostrare che $h^* \leq 2\log(n+1)$ con $n = \#$ nodi interni.

$h^* = h(root)$ quindi applichiamo il claim con $x = root$:

1. $n \geq 2^{bh(root)} - 1$
2. $bh(root) \geq \frac{h^*}{2}$ (almeno metà dei nodi lungo un cammino root-leaf sono neri)

E dunque:

$$n \geq 2^{\frac{h^*}{2}} - 1 \implies 2^{\frac{h^*}{2}} \leq n+1 \implies \frac{h^*}{2} \leq \log(n+1) \implies h^* \leq 2\log(n+1)$$

Proprietà di sottostruttura ottima di Matrix Chain Multiplication

Input: una sequenza $< A_1, ..., A_n >$ di n matrici di dimensioni tali che:

$$\#col(A_i) = \#row(A_i + 1), \forall i = 1, \ldots, n-1$$

Goal: calcolare l'ordine di moltiplicazione di $A_1 \cdot \ldots \cdot A_n$ che minimizza il numero di moltiplicazioni scalari.

Il problema è un problema di ottimizzazione in quanto presenta diverse soluzioni basate sulla metrica della funzione obiettivo ed è quindi possibile trovare una soluzione ottima che sia migliore di tutte le altre.

Una soluzione è una parentesizzazione:

$$costo(P) = \#moltiplicazioni\ scalari\ della\ parentesizzazione$$

Prop: Il problema **Matrix Chain Multiplication** ha la proprietà di sottostruttura ottima.

Proof: Sia P_{opt} una parentesizzazione ottima, ovvero tale che:

$$costo(P_{opt}) \leq costo(P), \forall\ parentesizzazione\ P\ di\ A_1, \dots, A_n$$

Esiste un punto di divisione che divide la parentesizzazione in due sottosequenze:

$$P_{opt}(A_1 \cdot ... \cdot A_n) = P_1(A_1 \cdot ... \cdot A_{i-1})P_2(A_i \cdot ... \cdot A_n)$$

$$\exists\ P_1, P_2; \exists\ i \in \{2, ..., n\}$$

$$costo(P_{opt}) = costo(P_1) + costo(P_2) + h$$

h : # di moltiplicazioni scalari per moltiplicare le due matrici risultanti (la matrice risultante da $A_1 \cdot \ldots \cdot A_{i-1}$ per quella risultante da $A_i \cdot \ldots \cdot A_n$). Questo valore non dipende da P_1 o da P_2 ma solo da i (che è fissato).

Restringiamo P_{opt} ai sottoproblemi:

$$< A_1, ..., A_{i-1} > \rightarrow P_1, < A_i, ..., A_n > \rightarrow P_2$$

Per dimostrare che Matrix Chain Multiplication ha la proprietà di sottostruttura ottima dobbiamo dimostrare che:

- P_1 è una parentesizzazione ottima di $A_1 \cdot \ldots \cdot A_{i-1}$;
- P_2 è una parentesizzazione ottima di $A_i \cdot \ldots \cdot A_n$.

Supponiamo per assurdo che P_1 non sia una parentesizzazione ottima di $A_1 \cdot \ldots \cdot A_{i-1}$, questo vuol dire che $\exists$ una parentesizzazione P_1' di $A_1 \cdot \ldots \cdot A_{i-1}$ che ha costo inferiore a quello di P_1:

$$costo(P_1') < costo(P_1)$$

Definisco allora la parentesizzazione P^*:

$$P^*(A_1 \cdot ... \cdot A_n) = P_1'(A_1 \cdot ... \cdot A_{i-1})P_2(A_i \cdot ... \cdot A_n)$$

$$costo(P^*) = costo(P_1') + costo(P_2) + h < costo(P_1) + costo(P_2) + h \implies$$

$$costo(P^*) = costo(P_1') + costo(P_2) + h < costo(P_{opt})$$

Il che è assurdo, poiché nessuna parentesizzazione di $A_1 \cdot \ldots \cdot A_n$ può essere inferiore di P_{opt}, poiché è, per definizione, ottima.

Dunque, P_1 è ottima per $A_1 \cdot \ldots \cdot A_{i-1}$, e analogamente si dimostra che P_2 è ottima.

Programmazione dinamica

Perché un problema possa essere risolto attraverso la programmazione dinamica deve presentare alcune caratteristiche:

- **Overlapping Subproblems**: deve avere un numero polinomiale di sottoproblemi che si ripetono;
- **Proprietà di sottostruttura ottima**: Data una soluzione ottima, le sue restrizioni ai sottoproblemi saranno ottime per i sottoproblemi.

Quando è possibile applicare la programmazione dinamica

Consideriamo i seguenti due problemi aventi lo stesso input.

Input: Un grafo $G = (V, E)$ e due vertici $u, v \in V$.

Unweighted Shortest Path: trovare un cammino (in un grafo non pesato) da u a v che abbia lunghezza minima.

Longest Simple Path: trovare un cammino semplice da u a v che abbia lunghezza massima.

La lunghezza di un cammino $p = < a_0, a_1, \dots, a_n >$ è $l(p) = n$, ovvero il numero di archi che contiene.

L'**Unweighted Shortest Path** gode di proprietà di sottostruttura ottima, dunque possiamo risolvere questo problema con la programmazione dinamica.

Il **Longest Simple Path** invece non gode di questa proprietà.

Longest Common Subsequence (LCS)

Input: due sequenze $X = < x_1, \dots, x_m >$ e $Y = < y_1, \dots, y_n >$.

Goal: trovare una sottosequenza Z comune a X e Y che abbia lunghezza massima.

Esempio:

$$X = < A, B, C, B, D, A, B > m = 7$$

$$Y = < B, D, C, A, B, A > n = 6$$

$$Z = LCS(X, Y) = < B, C, B, A >, lenght(Z) = 4$$

E non esistono sequenze comuni di lunghezza 5.

Data una sequenza $X = < x_1, x_2, \dots, x_m >$ e una sequenza $Z = < z_1, \dots, z_k >$, diciamo che Z è una **sottosequenza** di X se esiste una sequenza crescente di indici $\{i_1, \dots, i_k\}$ di X, $x_{ij} = z_j$.

Nell'esempio precedente $Z = < B, C, B, A >$ la sequenza di indici di X è $< 2, 3, 4, 6 >$, infatti:

$$X_2 = B = Z_1 \rightarrow i_1 = 2$$

$$X_3 = C = Z_2 \rightarrow i_2 = 3$$

$$X_4 = B = Z_3 \rightarrow i_3 = 4$$

$$X_6 = A = Z_3 \rightarrow i_4 = 6$$

$$X_{i1} X_{i2} X_{i3} X_{i4} = Z_1 Z_2 Z_3 Z_4$$

Approccio brute-force

Enumerare tutte le sottosequenze di X e controllare per ognuna di esse se è anche una sottosequenza di Y, tenendo traccia della sottosequenza comune più lunga. Ma questa procedura richiede tempo esponenziale.

Proprietà di sottostruttura ottima di LCS

L'i-esimo prefisso di $X = < x_1, x_2, \dots, x_m >$ è $X_i = < x_1, \dots, x_i >$, per $i = 0, 1, \dots, m$.

Teorema (Sottostruttura ottima di LCS)

Siano $X = < x_1, x_2, \dots, x_m >$ e $Y = < y_1, y_2, \dots, y_n >$ sequenze, e sia $Z = < z_1, z_2, \dots, z_k >$ una LCS di X e Y.

1. Se $x_m = y_n \implies z_k = x_m = y_n$ e Z_{k-1} è LCS di X_{m-1} e y_{n-1}. In altre parole, se l'ultimo elemento di X e l'ultimo elemento di Y coincidono, allora questo elemento fa parte della LCS Z, e il resto della LCS è la LCS delle sottosequenze X_{m-1} e Y_{m-1}.
2. Se $x_m \neq y_n \implies z_k \neq x_m$ e Z è LCS di X_{m-1} e Y. In questo caso, se l'ultimo elemento di X e l'ultimo elemento di Y non coincidono, allora x_m non fa parte della LCS Z e dunque la LCS è la LCS tra X_{m-1} e Y.
3. Se $x_m \neq y_n \implies z_k \neq y_n$ e Z è LCS di X e Y_{n-1}. In questo ultimo caso, analogamente, se l'ultimo elemento di X e l'ultimo elemento di Y non coincidono, allora y_n non fa parte della LCS Z e dunque la LCS è la LCS tra X e Y_{n-1}.

Dimostriamolo:

1. Se $z_k \neq x_m$, allora potremmo aggiungere $x_m = y_n$ a Z ottenendo una sequenza comune di lunghezza $k + 1 *$. Z è la sequenza più lunga quindi $z_k = x_m = y_n$.

   Z_{k-1} è comune a $Xm - 1$ e Y_{n-1} e ha lunghezza $k - 1$. Se esistesse una sequenza W comune a x_{m-1} e Y_{n-1} più lunga di Z_{k-1}, allora $W \cup \{x_m\}$ avrebbe lunghezza maggiore di k e sarebbe più lunga di $Z *$.

2. Se $z_k \neq x_m$ allora Z è comune a X_{m-1} e Y. Se esistesse W comune a X_{m-1} e Y di lunghezza maggiore a Z, allora W sarebbe comune a X e Y e dunque maggiore di Z, contraddicendo l'ipotesi $*$.
3. Il punto 3 si dimostra in maniera simmetricamente analoga al punto 2.

Il teorema dunque suggerisce un approccio ricorsivo:

Sia $c[i,j]$ la lunghezza della LCS tra i primi i elementi di X e i primi j elementi di Y, allora:

$$c[i,j] = \begin{cases} 0 \; se \; i = 0, j = 0 \\ c[i-1, j-1] + 1 \; se \; i,j > 0 \; e \; x_i = y_j \\ max(c[i, j-1], c[i-1, j]) \; se \; i,j > 0 \; e \; x_i \neq y_j \end{cases}$$

Basandosi su questa definizione possiamo scrivere un algoritmo ricorsivo. Inoltre, lo spazio di questi sottoproblemi ha dimensione polinomiale, infatti il numero di sottoproblemi distinti è uguale al numero di $c[i,j]$ distinti, $0 \leq i \leq m$ e $0 \leq j \leq n \implies \Theta(mn)$.

Utilizziamo un approccio basato sulla programmazione dinamica:

La procedura **LCS Lenght** prende in input due sequenze $X = < x_1, x_2, \dots, x_m >$ e $Y = < y_1, y_2, \dots, y_n >$ assieme alle loro lunghezze, memorizza i valori di $c[i,j]$ in una tabella $c[0:m, 0:n]$ e ne calcola gli elementi in ordine di riga crescente da sx a dx. Mantiene pure la tabella $b[1:m, 1:n]$ di "frecce".

Il tempo di questa procedura è $\Theta(mn)$, dato che ogni elemento è calcolato in $\Theta(1)$.

```
LCS-Lenght(X,Y,m,n)
1. initialize c[0:m,0:m] e b[1:m,1:n]
2. for i = 0 to m
3.   c[i,0] = 0
4. for j = 0 to n
5.   c[0,j] = 0
6. for i = 1 to m
7.   for j = 1 to n
8.     if x[i] == y[j]
9.       c[i,j] = c[i-1,j-1]+1
10.       b[i,j] = "↖"
11.    else if c[i-1,j] >= c[i,j-1]
12.      c[i,j] = c[i-1,j]
13.      b[i,j] = "↑"
14.    else
15.      c[i,j] = c[i,j-1]
16.      b[i,j] = "←"
17. return c, b
```

Facciamo un esempio per chiarire l'approccio di questa procedura:

Siano $X = < A, B, C, D, A, F >$ e $Y = < A, C, B, C, F >$, costruiamo una tabella seguendo l'algoritmo appena visto:

Il primo passo è quello di inizializzare a 0 ogni elemento delle due sequenze:

		A	**B**	**C**	**D**	**A**	**F**
	0	0	0	0	0	0	0
A	0						
C	0						
B	0						
C	0						
F	0						

Da adesso, per ogni coppia di elementi delle sequenze, confrontiamo i due elementi e comportiamoci in questo modo:

- Se i due elementi sono uguali, allora riempiamo la casella corrispondente con il valore della casella diagonalmente precedente e aggiungiamo 1. Allo stesso tempo, nella tabella b delle frecce, inseriamo nella posizione corrispondente la freccia "↖".
- Altrimenti, se gli elementi sono diversi e il valore nella casella in posizione superiore alla nostra è maggiore o uguale al valore nella casella a sinistra della nostra, riempiamo la nostra casella con il valore della casella superiore e inseriamo nella tabella b la freccia "↑".
- Altrimenti, la nostra casella assumerà il valore della casella alla sua sinistra e inseriremo nella tabella b la freccia "←".

Riempiamo adesso la tabella basandoci su questa procedura:

		A	**B**	**C**	**D**	**A**	**F**
		0	0	0	0	0	0
A	0	1	1	1	1	1	1
C	0	1	1	2	2	2	2
B	0	1	2	2	2	2	2
C	0	1	2	3	3	3	3
F	0	1	2	3	3	3	4

La lunghezza della **LCS** sarà dunque 4. Contestualmente, avremo riempito anche la tabella b:

		A	**B**	**C**	**D**	**A**	**F**
A		↖	←	←	←	←	←
C		↑	↑	↖	←	←	←
B		↑	↖	↑	↑	↑	↑
C		↑	↑	↖	←	←	←
F		↑	↑	↑	↑	↑	↖

Adesso sarà sufficiente seguire le frecce dall'ultima posizione (F, F) e, ogni volta che incontriamo la freccia "↖", tenere conto dell'elemento corrispondente:

La **LCS** sarà dunque la sequenza $< A, B, C, F >$.

Vediamo adesso la procedura per il print:

```
1. if i == 0 OR j == 0
2.   return
3. if b[i,j] == "↖"
4.   PrintLCS(b, X, i-1, j-1)
5.   print "xi"
6. else if b[i,j] == "↑"
7.   PrintLCS(b, X, i-1, j)
8. else PrintLCS(b, X, i, j-1)
```

PrintLCS impiega $\Theta(m+n) - Time$ perché decrementa almeno uno tra m ed n ad ogni passo temporale.

Si può migliorare il codice di PrintLCS eliminando la tabella b. Infatti, c[i, j] dipende solamente da c[i-1, j-1], c[i, j-1] e c[i-1, j]. Dato c[i, j], si può determinare in tempo $\Theta(1)$ quale tra questi tre valori è stato utilizzato per c[i, j]. SI può così ricostruire una LCS in tempo $\Theta(m+n) - Space$, ma comunque lo spazio usato non diminuisce asintoticamente perché la tabella richiede lo stesso spazio.

Elementi della strategia greedy

Un algoritmo greedy ottiene una soluzione ottima ad un problema di ottimizzazione attraverso una serie di scelte localmente ottime. Ad ogni "bivio" l'algoritmo fa le "scelte" che sembrano migliori al momento, senza tenere conto di come questo potrebbe influenzare le scelte successive.

Tuttavia questa strategia non produce sempre un risultato efficace.

Come sviluppare un algoritmo greedy

1. Formulare il problema di ottimizzazione come uno nel quale si fa una scelta e si rimane con un unico sottoproblema.
2. Dimostrare che esiste sempre una soluzione ottima che contiene la scelta greedy così che la scelta greedy sia "sicura".
3. Dimostrare la proprietà di sottostruttura ottima.

Proprietà di scelta greedy

La proprietà di scelta greedy vale se esiste una soluzione ottima che contiene la scelta greedy ad ogni iterazione.

Per dimostrare che un problema ha la proprietà di scelta greedy si esamina una soluzione globale e poi si mostra che modificarla sostituendo la scelta greedy a qualche altra scelta non cambia il fatto che la soluzione sia ancora ottima.

Proprietà di scelta greedy per il problema activity selection

Input:

- Un insieme $S = \{a_1, ..., a_n\}$ di attività;
- Per ogni attività a_i sono specificati un tempo di inizio (starting time) s_i e un tempo di terminazione (finish time) $f_i, 0 \leq s_i < f_i < \infty$ e $a_i = [s_i, f_i)$, dunque ogni attività rappresenta un intervallo di tempo.

Goal:

- Selezionare un sottoinsieme $A \subseteq S$ di attività tali che A abbia cardinalità massima e le attività in A siano compatibili, ovvero in modo tale che non si sovrappongano: $\forall a_i, a_j \in A \; [s_i, f_i) \cap [s_j, f_j) = \emptyset$.

Possiamo ordinare le attività in ordine crescente di tempo di terminazione: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Euristica: attività che termina prima, in modo tale che ogni attività che scelgo non si sovrapponga a quella già selezionata. L'algoritmo che ottengo ha complessità $O(n \log n) - Time$.

Teorema (scelta greedy di activity selection)

Sia S_k un sottoproblema non vuoto, e sia $a_m \in S_k$ l'attività con il più piccolo tempo di terminazione f_m. Allora, a_m è inclusa in qualche sottoinsieme di dimensione massima di attività compatibili di S_k, ovvero in una soluzione ottima per S_k.

Dimostriamolo:

Supponiamo che A_k sia una soluzione ottima per S_k.

Sia a_j l'attività che finisce prima di in A_k:

- Se $a_m = a_j \implies a_m \in A_k$ (abbiamo finito, perché a_j fa già parte di A_k che è soluzione ottima);
- Se $a_m \neq a_j \implies a_m \notin A_k$

Definiamo allora una soluzione che sostituisca a_j con a_m: $A'_k := (A_k \setminus \{a_j\}) \cup \{a_m\}$:

A'_k è soluzione ottima: infatti $(A_k \setminus \{a_j\}) \cap \{a_j\} = \emptyset$ e inoltre a_m finisce prima di a_j:

$$\emptyset = (A_k \setminus \{a_j\}) \cap [s_j, f_j) \subseteq [f_j, +\infty) \implies \emptyset = (A_k \setminus \{a_j\}) \cap [s_m, f_m) \subseteq [f_j, +\infty)$$

In altre parole, poiché a_m finisce prima di a_j, possiamo aggiungerla ad A'_k senza che si sovrapponga con nessun'altra attività. Inoltre:

$$|A'_k| = (|A_k| - 1) + 1 = |A_k|$$

abbiamo sostituito un'attività che termina più tardi con una che termina prima, e dunque la soluzione è anch'essa ottima.

Proprietà di scelta greedy per l'algoritmo di Huffman

L'**algoritmo di Huffman** costruisce un **albero di codifica prefix-free** in cui le foglie corrispondono ai caratteri da codificare. Si interpreta la codifica di un carattere con il cammino semplice dalla radice al carattere:

- "0" : vai al nodo figlio sx;

- "1" : vai al nodo figlio dx.

L'albero ha $|C|$ foglie (C è l'alfabeto), una per ogni carattere e ha esattamente $|C| - 1$ nodi interni.

Notazione

Sia T un albero di codifica prefix-free:

- $c.freq$: frequenza del carattere c, $\forall c \in C$;
- $d_T(c)$: profondità di c nell'albero T, che corrisponde alla lunghezza della codifica di c (numero di bit necessari);

Il numero totale di bit per codificare un file è

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

L'idea è quella di assegnare ai caratteri con frequenza maggiore un numero minore di bit, mentre i caratteri con frequenza minore saranno più lunghi, in modo tale da minimizzare il numero di bit necessari per codificare un alfabeto.

Euristica: carattere più frequente.

Proprietà di scelta greedy per i codici prefix-free (da completare)

Sia C un alfabeto in cui ogni carattere c ha frequenza $c.freq$. Siano $x, y \in C$ i caratteri con frequenza minima. Allora, esiste una codifica prefix-free ottima in cui le codifiche di x e y hanno la stessa lunghezza massima (profondità dell'albero) e differiscono per l'ultimo bit (infatti uno sarà figlio sx mentre l'altro sarà figlio dx).

Dimostriamolo:

Sia T un albero che rappresenta una codifica prefix-free ottima. Siano a e b i due caratteri che corrispondono alle foglie di massima profondità di T.

Senza perdere generalità, possiamo assumere che $a.freq \leq b.freq$ e $x.freq \leq y.freq$. Avremo che

$$\begin{cases} x.freq \leq a.freq \\ y.freq \leq b.freq \end{cases}$$

...

Il problema dello zaino (knapsack problem)

Il problema dello zaino è un problema di ottimizzazione combinatoria.

Dato uno zaino di portata k chilogrammi e una collezione A di n oggetti.

$$A = \{a_1, a_2, ..., a_{n-1,} a_n\}$$

Indicheremo con S l'insieme degli oggetti che decideremo di inserire nello zaino, definito come:

$$S \subseteq A \sum_{i \in S} p(x) \leq k$$

Dove $p(x)$ è una funzione che ritorna il peso dell'oggetto x, e dunque è necessariamente minore della portata massima dello zaino, k.

Massimizzare il numero di oggetti

Supponiamo di voler massimizzare il numero di oggetti da inserire nello zaino, dunque cercheremo di trasportare il maggior numero di oggetti, prediligendo quelli con un peso $p(x)$ minore.

Partiamo da un array A ordinato in maniera decrescente:

$$Z(k, n) = \begin{cases} 0 \; se \; n = 0 \\ 0 \; se \; k < p(n) \\ 1 + Z(k - p(n), n - 1) \end{cases}$$

Stiamo dunque scorrendo l'array di oggetti A e richiamando ricorsivamente la funzione. I casi che interrompono la ricorsione sono la fine degli oggetti nella collezione A ($n = 0$) e il superamento della capienza massima k dello zaino ($k < p(n)$).

Per ogni elemento aggiunto allo zaino, incrementiamo il nostro risultato e decrementiamo la capienza residua e il numero di oggetti della collezione.

Massimizzare il peso degli oggetti

In questa istanza del problema invece cerchiamo di riempire lo zaino il più possibile.

$$Z(k.n) = \begin{cases} 0 \; se \; n = 0 \\ -\infty \; se \; k < 0 \\ max((p(n) + Z(k - p(n), n - 1)), Z(k, n - 1)) \end{cases}$$

In questo caso, l'algoritmo interrompe la ricorsione se finiscono gli oggetti e il peso viene posto a $-\infty$ se viene superata la portata. Altrimenti, la funzione calcola ricorsivamente le combinazioni di oggetti e sceglie la combinazione migliore (max).

Massimizzare il valore degli oggetti

Introduciamo un valore associato ad ogni oggetto, denotato con la funzione $v(i) = a$.

Stiamo cercando di trovare il valore complessivo massimo.

L'algoritmo sarà il seguente:

$$Z(k, n) = \begin{cases} 0 \; se \; n = 0 \\ 0 \; se \; k = 0 \\ -\infty \; se \; k < 0 \\ max(v(n) + Z(k - p(n), n - 1, Z(k, n - 1)) \end{cases}$$

Proprietà di sottostruttura ottima e scelta greedy nel problema dello zaino

Knapsack

Input:

- n oggetti tali che per ogni oggetto i sia specificato un peso w_i e un valore v_i;
- un limite di peso W.

Goal:

- Scegliere un sottoinsieme $J \subseteq \{1, ..., n\}$ di oggetti di valore complessivo massimo e peso complessivo limitato superiormente da W o, equivalentemente, trovare un assegnamento per le variabili $\{x_1, ..., x_n\}$ (ognuna corrisponde ad un oggetto: $x = 0$ non prendo l'oggetto, $x = 1$ prendo l'oggetto).

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i \in \{0, 1\}, \forall \; i \in \{1, ..., n\}$$

Fractional knapsack

Input:

- n oggetti tali che per ogni oggetto i sia specificato un peso w_i e un valore v_i;
- un limite di peso W.

Goal:

- Trovare un assegnamento per le variabili $\{x_1, ..., x_n\}$ tale che

$$\max \sum_{i=1}^{n} v_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$0 \leq x_i \leq 1, x_i \in Q \ \ \forall i \in \{1, ..., n\}$$

A differenza del $\{0,1\}$-knapsack, il **fractional knapsack** permette quindi di prendere anche solo parte di un oggetto e non necessariamente un oggetto intero.

Prop: $\{0,1\}$-knapsack ha la proprietà di sottostruttura ottima. Ricordiamo che dobbiamo mostrare che una soluzione ottima per un sottoinsieme di oggetti può essere estesa per ottenere una soluzione ottima per l'intero problema.

Dimostriamolo:

Sia $\{x_1^*, x_2^*, \ldots, x_n^*\}$ una soluzione ottima al problema, allora:

$$\sum_{i=1}^{n} w_i x_i^* \leq W \ e \ \sum_{i=1}^{n} v_i x_i^* \ è \ massimo$$

Consideriamo il sottoproblema (escludiamo un oggetto j) le cui variabili sono

$$\{x_1, ..., x_{j-1}, x_{j+1}, ..., x_n\}$$

Supponiamo per assurdo che la restrizione $\{x_1^*, ..., x_{j-1}^*, x_{j+1}^*, ..., x_n^*\}$ non sia ottima per questo sottoproblema, cioè esiste una soluzione alternativa $S' = \{x_1', ..., x_{j-1}', x_{j+1}', ..., x_n'\}$ tale che:

$$\sum_{i=1, i \neq j}^{n} v_i x_i' > \sum_{i=1, i \neq j}^{n} v_i x_i^*$$

$$\sum_{i=1, i \neq j}^{n} w_i x_i' \leq W - x_j^* w_j$$

allora S' sembrerebbe una soluzione al problema originario migliore di $\{x_1^*, x_2^*, ..., x_n^*\}$. Combiniamo questa soluzione con l'oggetto escluso:

$$\sum_{i=1, i \neq j}^{n} v_i x_i' + v_j x_J^* > \sum_{i=1, i \neq j}^{n} v_i x_i^* + v_j x_j^* = \sum_{i=1}^{n} v_i x_i *$$

Questa disuguaglianza implica che la soluzione combinata sia migliore della soluzione ottima orinale per il problema, il che è assurdo.

Poiché arriviamo a una contraddizione, possiamo concludere che la soluzione $\{x_1^*, x_2^*, ..., x_n^*\}$ è ottima per il problema completo **e** anche per i sottoproblemi. Pertanto, il problema ha la **proprietà di sottostruttura ottima.**

La stessa dimostrazione può essere utilizzata per dimostrare che anche il fractional knapsack ha proprietà di sottostruttura ottima. Questo ci fa concludere che è possibile utilizzare la programmazione dinamica per entrambi i problemi.

Proprietà di scelta greedy per il fractional knapsack

Euristica: scegliere l'oggetto con più grande valore per unità di peso.

Per ogni oggetti prendo l'oggetto con $k_i = \frac{v_i}{w_i}$ più grande.

Dimostriamo che il **fractional knapsack** ha la **proprietà di scelta greedy**:

Ordiniamo gli oggetti in **ordine decrescente** in base al rapporto k: $k_1 \geq k_2 \geq ... \geq k_n$. Dunque k_1 sarà l'oggetto con il rapporto peso/valore migliore, e k_n quello con il rapporto peggiore. Questo ordinamento garantisce che gli oggetti con **massimo valore per unità di peso** vengano sicuramente inclusi nella soluzione.

Come abbiamo detto, x_i è la **frazione di un oggetto** i inclusa nello zaino, dunque **il valore totale** degli oggetti nello zaino è dato da:

$$\sum_{i=1}^{n} x_i v_i$$

Introduciamo $q_i = x_i w_i$, inteso come il peso effettivo dell'oggetto incluso nello zaino.

Ora, il valore dell'oggetto i incluso nello zaino, $x_i v_i$, può essere scritto come:

$$x_i v_i = q_i k_i$$

Sostituendo questa relazione al valore totale degli oggetti nello zaino, otteniamo:

$$\sum_{i=1}^{n} x_i v_i = \sum_{i=1}^{n} q_i k_i$$

L'obiettivo del problema è massimizzare

$$\sum_{i=1}^{n} q_i k_i$$

rispetto al vincolo di peso massimo dello zaino

$$\sum_{i=1}^{n} q_i \leq W$$

Sia $\{q_1, ..., q_n\}$ una soluzione ottima che non contiene k_1 (la scelta greedy), e dunque $q_1 = 0$ (ovvero l'oggetto con il massimo valore per unità di peso è stato completamente ignorato).

Supponiamo allora che il secondo oggetto sia interamente o parzialmente incluso, dunque $q_2 \neq 0$.

Allora possiamo sostituire il peso dell'oggetto 2 con lo stesso peso preso dall'oggetto 1, ottenendo una nuova soluzione:

$$\{q'_1 = q_2, q'_2 = 0, q'_3 = q_3, ..., q'_n = q_n\}$$

Allora il peso:

$$\sum_{i=1}^{n} q'_i = q'_1 + q'_2 + \sum_{i=3}^{n} q_i = q_2 + 0 + \sum_{i=3}^{n} q_i$$

Dunque il peso della soluzione rimane invariato e rispetta il vincolo, inoltre:

$$\sum_{i=1}^{n} q'_i k_i = q'_1 k_1 + q'_2 k_2 + \sum_{i=3}^{n} q_i k_i = q_2 k_1 + 0 k_2 + \sum_{i=3}^{n} q_i k_i \geq$$

$$\geq q_2 k_2 + \sum_{i=3}^{n} q_i k_i$$

Poiché $k_1 \geq k_2$ in quanto sono ordinati in ordine decrescente per rapporto valore/peso.

In altre parole, la nuova soluzione (con il peso dell'oggetto 2 sostituito) è uguale o migliore della soluzione senza k_1, dunque ho trovato una soluzione ottima che contiene la scelta greedy.

Proprietà di scelta greedy per $\{0, 1\}$-knapsack

Supponiamo di avere 3 oggetti:

$$i = 1, w_1 = 10kg, v_1 = 60 \implies k_1 = 6$$

$$i = 2, w_2 = 20kg, v_2 = 100 \implies k_2 = 5$$

$$i = 3, w_3 = 30kg, v_3 = 120 \implies k_3 = 4$$

$$W = 50kg$$

Se utilizzassimo la strategia greedy per scegliere gli oggetti, ordinandoli dunque in ordine decrescente, otterremmo:

$$k_1 \geq k_2 \geq k_3$$

E potremmo selezionare solo gli oggetti 1 e 2 a causa del limite di peso. Questa soluzione non rappresenta però la soluzione ottima, che è invece quella che comprende k_2 e k_3, massimizzando il valore per il peso a disposizione.

Dunque questo problema **non** ha proprietà di scelta greedy.

Ricerca Depth First (DFS)

Sia $G = (V, E)$ un grafo (digrafo). La ricerca **DFS** esplora gli archi che dipartono dal vertice v scoperto più recentemente e che sono ancora inesplorati. Una volta che tutti gli archi uscenti da v sono stati esplorati, la ricerca fa "retromarcia" per esplorare gli archi uscenti dal vertice u da cui aveva raggiunto v.

Il processo continua finché tutti i vertici raggiungibili dal vertice iniziale (sorgente) non sono stati esplorati. Se nel grafo ci sono ancora dei vertici non esplorati la ricerca DFS seleziona un nuovo vertice sorgente e ripete la procedura.

Grafo dei predecessori

Un **grafo dei predecessori** è una rappresentazione di un grafo diretto in cui, per ogni nodo, si tiene traccia dei nodi che puntano a esso (cioè i suoi predecessori). In altre parole, è un modo per indicare quali nodi hanno archi diretti verso un certo nodo.

Topological sort

Sia $G = (V, E)$ un grafo. Un ordinamento topologico di G è un ordinamento lineare $<$ dei suoi vertici tale che

$$(u, v) \in E \implies u < v$$

L'algoritmo **topological sort** prende in input un grafo orientato aciclico (**DAG**, o "direct acyclic graph) $G = (V, E)$ e restituisce in output un ordinamento topologico di G. Questo algoritmo è definito solo su DAG.

La procedura

1. Invoca la ricerca DFS su G per calcolare il finish time $v.f, \forall v \in V$ (passo temporale a cui termina l'esplorazione di un vertice v);
2. al termine dell'esplorazione di ogni vertice, lo inserisce in una lista collegata L;
3. restituisce la lista L.

Il tempo computazionale del topological sort è $\Theta(|V| + |E|)$ dato che si visita ogni vertice una volta e si percorre ogni arco una sola volta.

Lemma 1

G è un DAG, dunque DFS(G) non genera archi all'indietro.

Dimostriamolo:

Supponiamo che una DFS su G produca un arco all'indietro (u, v). Allora v è un predecessore di u nel grafo dei predecessori. Quindi G contiene un arco diretto (v, u) e adesso (u, v) chiude un ciclo, il che è assurdo perché G è un DAG.

Supponiamo adesso che G contenga un ciclo r e dimostriamo che in tal caso una ricerca DFS genererebbe un arco all'indietro. Sia v il primo vertice in r ad essere scoperto dalla DFS. Al momento $v.d$ (scoperta di v) i vertici di r formano un cammino bianco (di vertici non ancora esplorati) da v al predecessore di v in r che chiamiamo u. Troveremo un arco (u, v) grigio, perché v è ancora in esplorazione.,

Allora, per il **unite path theorem**, u è un discendente di v nel grafo dei predecessori e (u, v) è un arco all'indietro.

Teorema (correttezza del topological sort)

Per mostrare la correttezza del **topological sort**, è sufficiente dimostrare che se $(u, v) \in E$, allora $u < v$, e dunque $v.f < u.f$.

Facciamolo osservando che quando DFS esplora u, il vertice v non può essere grigio, altrimenti v sarebbe un predecessore di u e (u, v) sarebbe un arco all'indietro (impossibile perché G è un DAG). A questo punto, v può essere di colore:

- **bianco**: v è discendente di u e quindi viene esplorato completamente prima che l'esplorazione di u sia completa, e dunque $v.f < u.f$;
- **nero**: v è già stato scoperto ed esplorato, quindi $v.f$ è già stato determinato e dunque necessariamente $v.f < u.f$.

Single-Source Shortest Path

Consideriamo il seguente problema:

Input:

- un grafo $G = (V, E)$;
- un vertice $s \in V$, chiamato sorgente;
- una funzione di peso sugli archi $w : E \to \mathbb{R}$.

Goal:

Trovare un cammino minimo da s a v, $\forall v \in V$, ovvero $p :=< n_0, n_1, ..., n_n >$ con $n_0 = s$ e $n_n = v$ tale che $(n_{i-1}, n_i) \in E$ e inoltre vogliamo che $w(p) := \sum_{i=1}^{n} w(n_{i-1}, n_i)$, ovvero la somma dei pesi degli archi del cammino sia minima.

In altre parole, il problema consiste nel trovare, in un grafo con archi pesati, il cammino minimo dal vertice sorgente ad ogni vertice del grafo, ovvero il cammino la cui somma dei pesi degli archi sia minima.

Notazione:

$$d(s, v) := \begin{cases} \min s \to^p v, w(p) = \min \sum_{i=1}^{n} w(n_{i-1}, n_i) \\ \infty \; se \; non \; esiste \; alcun \; cammino \; da \; s \; a \; v \end{cases}$$

Osservazione: Ci potrebbero essere degli archi con peso negativo.

- Se G **non** contiene cicli di peso negativo raggiungibili da s, allora $d(s, v)$ è ben definito per ogni v.
- Se G contiene un ciclo di peso negativo raggiungibile da s, allora $d(s, v)$ non è ben definito. Infatti, posso sempre trovare un cammino di peso inferiore ripetendo il ciclo. Pertanto, in tal caso definiamo $d(s, v) = -\infty$.

Algoritmo di Bellman-Ford

Risolve il **Single-Source Shortest Path** nel caso generale in cui i pesi degli archi possano essere negativi ($w : E \to \mathbb{R}$).

L'algoritmo opera in questo modo:

Dato un grafo $G = (V, E)$ orientato, $s \in V$ **vertice sorgente** e $w : E \to \mathbb{R}$ la funzione peso, l'algoritmo Bellman-Ford restituisce un **valore booleano**: `false` se esiste un ciclo di peso negativo raggiungibile dal vertice sorgente, `true` se invece $d(s, v)$ è ben definita $\forall v \in V$ e in tal caso **è possibile ricostruire un cammino minimo** da s a $v, \forall v \in V$.

Vediamo la procedura:

```
bellman-ford(G,s,w)
1. initialize-single-source(G,s)
2. for i = 1 to (|V|-1)
3.   for (u,v) in E
4.     relax(u,v,w)
5. for each (u,v) in E
6.   if v.d > u.d + w(u,v)
7.     return false
8. return true
```

```
initialize-single-source(G,s)
1. for each v in V
2.   v.d = ∞
3.   v.π = null
4. s.d = 0
```

```
relax(u,v,w)
1. if v.d > u.d + w(u,v)
2.   v.d = u.d + w(u,v)
3.   v.π = u
```

In altre parole, questo algoritmo prende in input un grafo e un vertice sorgente, inizializza inizialmente la distanza dal vertice sorgente agli altri vertici come "∞" e la distanza dal vertice sorgente a se stesso come "0". Sapendo che in un arco aciclico sono presenti al massimo $|V|-1$ archi, per $|V|-1$ volte si esaminano tutti i vertici, cercando ad ogni iterazione un cammino migliore del precedente dal vertice sorgente fino al vertice esaminato, se esiste un cammino migliore passando per un altro arco, la distanza viene aggiornata con il nuovo peso totale. Se dopo $|V|-1$ iterazioni la distanza può ancora essere migliorata, allora esiste un ciclo negativo nel grafo.

L'algoritmo di Bellman-Ford impiega $O(|V|^2+|V||E|)-Time$ quando il grafo è rappresentato dalla sua lista di adiacenza.

Proprietà dei cammini minimi e dei rilassamenti

- **Disuguaglianza triangolare dei cammini minimi**: $\forall(u,v)\in E, d(s,v)\leq d(s,u)+w(u,v)$;
- **Upper-Bound Property**: Per tutto il tempo della computazione abbiamo $v.d\geq d(s,v)\forall v\in V$ e quando $v.d=d(s,v)$ non cambia più.
- **No-Path Property**: Se non ci sono cammini da s a $v, d(s,v)=\infty$;
- **Convergence Property**: Se $s\to u\to v$ è un cammino minimo in G per qualche $u,v\in V$ e $u.d=d(s,u)$ a qualunque istante precedente $relax(u,v,w)\implies v.d=d(s,v)$ a tutti gli istanti successivi;
- **Predecessor-Subgraph Property**: una volta che $v.d=d(s,v)\forall v\in V$ il grafo dei predecessori è un albero dei cammini minimi radicato ad s;
- **Path-Relaxation Property**: se $p=<v_o,v_1,...,v_k>$ da $s=v_0$ a v_k e se gli archi di p vengono rilassati secondo l'ordine $((v_0,v_1),(v_1,v_2),...,(v_{k_1},v_k))$ allora $v_k.d=d(s,v_k)$ a prescindere dagli altri passi di rilassamento che possono occorrere, anche se inframmezzati da rilassamenti degli archi di p.

Correttezza dell'algoritmo di Bellman-Ford

Lemma: Sia $G=(V,E)$ un grafo orientato, $s\in V$ un vertice sorgente, $w:E\to\mathbb{R}$ funzione peso.

Assumiamo che G **non** contenga cicli di peso negativo raggiungibili da s. Allora, dopo $|V|-1$ iterazioni del ciclo for dell'algoritmo di Bellman-Ford, $v.d=d(s,v)$ per ogni v raggiungibile da s.

Dimostriamolo:

Si consideri v un qualunque vertice raggiungibile da s e sia $p=<n_0,n_1,...,n_k>$ un cammino con $n_0=s$ e $n_k=v$ da s a v. Poiché i cammini minimi sono semplici, p ha al più $|V|-1$ archi, pertanto $k\leq|V|-1$.

Ognuna delle $|V|-1$ iterazioni del ciclo for rilassa tutti gli archi di E. Tra gli archi rilassati alla i-esima iterazione c'è sicuramente (n_{i-1},n_i) e questo vale per ogni $i\in\{1,...,k\}$. Per la Path-Relaxation Property allora $v.d=v_k.d=d(s,v_k)=d(s,v)$.

L'algoritmo di Dijkstra

Risolve il Single-Source Shortest Path Problem nel caso in cui $w:E\to\mathbb{R}_0^+$, ovvero $w(u,v)\geq0,\forall(u,v)\in E$.

Con una buona implementazione, il tempo computazionale di questo algoritmo è inferiore a quello di Bellman-Ford.

Possiamo pensare all'algoritmo di Dijkstra come a una generalizzazione della ricerca BFS ai grafi pesati.

Esso mantiene un insieme S di vertici i cui pesi dei cammini minimi da s sono già stati determinati. L'algoritmo seleziona ripetutamente il vertice $u \in V \setminus S$ con la stima del cammino minimo più piccola, aggiunge u ad S e rilassa tutti gli archi uscenti da u.

```
dijkstra(G,s,w)
1. initialize-single-source(G,s)
2. S = {0}
3. Q = {0}
4. for each u in V
5.   insert(Q,u)
6. while Q != {0}
7.   u = extract-min(Q)
8.   S = S U {u}
9.   for each v in GAdj[u]
10.    relax(u,v,w)
11.    if the call of relax decreased v.d
12.      decrease-key(Q,v,v.d)
```

In altre parole, si inizializza il vertice sorgente e le distanze degli agli vertici dal vertice sorgente vengono poste a infinito. Come abbiamo detto, S rappresenta l'insieme dei nodi per i quali il percorso più breve è già stato trovato, inizialmente sarà dunque vuoto. Q è invece una coda con priorità che contiene tutti i nodi V. Il nodo con la distanza minima dalla sorgente s ha priorità maggiore. Successivamente, si popola la coda con tutti i nodi del grafo, per poi estrarne il minimo (ovvero il nodo con distanza minima dal vertice sorgente) finché la coda non è vuota. Si aggiunge il nodo estratto all'insieme S, indicando che il percorso più breve per u è stato trovato. Per ogni nodo adiacente v ad u, si prova a migliorare il percorso attraverso la funzione relax. Se la distanza viene effettivamente aggiornata, bisogna aggiornare la posizione di v nella coda Q diminuendo il peso assegnato.

Correttezza Dijkstra

L'algoritmo di Dijkstra eseguito su un grafo orientato $G = (V, E)$ con una funzione di peso $w : E \to R_+^0$ e vertice sorgente s, termina con $u.d = d(s, u) \forall u \in V$.

Dimostriamolo:

Dimostriamo che, all'inizio di ogni iterazione del ciclo while (righe 6-12), $v.d = d(s, v) \forall v \in S$. Questo sarà sufficiente a provare la correttezza dell'algoritmo perché quando l'algoritmo termina $S = V$ così che $v.d = d(s, v) \forall v \in V$. Questo può essere dimostrato per induzione sul numero di iterazioni del ciclo while già avvenute, tale numero è uguale a $|S|$ all'inizio di ogni iterazione.

Casi base:

- $|S| = 0 \implies S = \emptyset \implies$ il claim è banalmente vero.
- $|S| = 1 \implies S = \{s\}$ e $s.d = 0 = d(s, s)$.

Ipotesi induttiva: $v.d = d(s, v) \forall v \in S$.

Passo induttivo: viene estratto $u \in Q = V \setminus S$ e viene aggiunto ad S, dobbiamo mostrare che $u.d = d(s, u)$ a questo istante.

- Se non ci sono cammini da s a u allora per la no-path property $u.d = \infty = d(s, u)$.
- Se esiste un cammino da s a u allora sia y il primo vertice su un cammino minimo da s a u che non sia in S e sia x il predecessore. Siccome y appare non successivamente a u e i pesi sono non negativi: $d(s, y) \leq d(s, u)$. Inoltre, extract-min (riga 7) ha estratto u perché $u.d$ è minimo in Q, quindi poiché $y \notin S$, $u.d \leq y.d$ e per la upper.bound property $d(s, u) \leq u.d$. Siccome $x \in S$, per l'ipotesi induttiva $x.d = d(s, x)$. Durante l'iterazione che ha aggiunto x ad S (x,y) è stato rilassato quindi y.d = d(s,y) per la convergence property.

  Allora:

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

  E per la upper-bound property sappiamo che questo valore non cambierà più.

Analisi dell'algoritmo di Dijkstra

L'algoritmo mantiene la coda di priorità Q invocando tre operazioni: Insert, Exctract-Min e Decrease-Key. L'algoritmo invoca Insert ed Extract-Min per ogni vertice. Ogni vertice è aggiunto ad S esattamente una volta, ogni arco in Adj[u] è esaminato nel ciclo for esattamente una volta durante il corso dell'intero algoritmo. Questo ciclo viene iterato per $|E|$ volte. Il tempo computazionale dell'algoritmo dipende dalla specifica implementazione della coda di priorità Q.

- Se implemento Q come un array che enumera i vertici e memorizza $v.d$ nella v-esima posizione dell'array, allora il tempo computazionale è $O(V^2)$.
- Se implemento Q come min-heap, ottengo tempo globale $O((V+E)\log V)$.

All-Pairs-Shortes-Path

Input: $G=(V,E)$ grafo orientato, $w: E \in \mathbb{R}$ funzione di peso.

Goal: trovare per ogni $(u,v) \in V$ un cammino minimo da u a v.

L'output di questo algoritmo è tipicamente una matrice le cui righe e colonne sono indirizzate dai vertici di G e l'elemento della riga u e colonna v è il peso del cammino minimo da u a v.

Applicazione: calcolo del diametro di una rete.

Osserviamo che possiamo risolvere APSP usando uno degli algoritmi per SSSP per $|V|$ volte.

- Se gli archi hanno peso non negativo possiamo usare l'algoritmo di Dijkstra:
  - Implementando la coda di priorità con array lineare otteniamo un algoritmo con runtime $O(V^3)$, ovvero la complessità di Dijkstra ripetuta per $|V|$ volte.
  - Se implementiamo la coda di priorità con min-heap, otterremo tempo computazionale uguale a $O(V(V+E)\log V)$.
- Se il grafo contiene invece archi di peso negativo, possiamo usare Bellman-Ford su ogni vertice, ottenendo un tempo computazionale uguale a $O(V^2E)$ che diventa $O(V^4)$ su un grafo denso.

Si può risolvere APSP in maniera più efficiente?

Assumiamo che il grafo pesato sia rappresentato dalla sua matrice di adiacenza (per SSSP era stato conveniente utilizzare la lista di adiacenza).

Enumeriamo i vertici di G, $V=\{1,2,...,n\}$, $|V|=n$. Allora l'input è rappresentato dalla matrice $W=(w_{ij}), n \cdot n$ definita da:

$$w_{ij} = \begin{cases} 0 \; se \; i=j \\ w_{ij} \; se \; i \neq j \; e \; (i,j) \in E \\ \infty \; se \; i \neq j \; e \; (i,j) \notin E \end{cases}$$

In altre parole, la matrice avrà 0 in posizione i,j se $i=j$, avrà il valore associato dalla funzione peso all'arco da i a j se $i \neq j$ e c'è un arco diretto da i a j, e avrà ∞ se $i \neq j$ e non c'è un arco diretto da i a j.

Assumiamo che il grafo non contenga cicli di peso negativo.

Una soluzione completa a All-Pairs-Shortest-Path include non solo la tabella coi pesi dei cammini minimi, ma anche una matrice $\Pi=(\pi_{ij})$ dei predecessori, con $\pi_{ij}=$ NIL se $i=j$ o se non ci sono cammini da i a j ($d(i,j)=\infty$) e π_{ij} è uguale al predecessore di j in un cammino minimo da i.

La funzione Print-All-Pairs-Shortest-Path stampa un cammino minimo da i a j.

```
Print-All-Pairs-Shortest-Path(P,i,j)
1. if i == j
2.   print i
3. else if pij == NIL
4.   print "non esistono cammini da i a j"
5. else Print-All-Pairs-Shortest-Path(P,i,pij)
6.   print j
```

Approccio basato sulla programmazione dinamica

Ci serve una formulazione ricorsiva del problema.

Sia $l_{ij}^{(r)}$ il peso di un cammino minimo da i a j che contiene al più r archi:

$$l_{ij}^{(0)} = \begin{cases} 0 \; i = j \\ \infty \; i \neq j \end{cases}$$

Per $r \geq 1$, un modo per ottenere un cammino minimo da i a j con al più r archi è considerare un cammino minimo con al più $r-1$ archi, così che $l_{ij}^{(r)} = l_{ij}^{(r-1)}$ in quanto minimo, oppure, considerare un cammino minimo con al più $r-1$ archi da i a un vertice intermedio k, per qualche $i \leq k \leq n$, e poi aggiungere l'arco (k, j), così $l_{ij}^{(r)} = l_{ik}^{(r-1)} + w_{kj}$.

Allora, possiamo completare la nostra formulazione ricorsiva:

$$l_{ij}^{(0)} = \begin{cases} 0 \; i = j \\ \infty \; i \neq j \end{cases}$$

$$r \geq 1, l_{ij}^{(r)} = min\{l_{ik}^{(r-1)} + w_{ij}\}$$

Che include anche $l_{ij}^{(r-1)}$ nel caso in cui $w_{ij} = 0$.

In altre parole ad ogni iterazione si genera la matrice $l^{(r)}$ contenente i pesi associati ai cammini minimi dal vertice i al vertice j e si controlla la matrice A^{r-1} per capire se è possibile migliorare il cammino minimo passando per un vertice k intermedio. In questo senso, la funzione "sceglierà" il minimo tra il cammino minimo già trovato nella matrice $l_{ij}^{(r-1)}$ e il cammino passando per un vertice intermedio k, ovvero $l_{ik}^{(r-1)} + w_{ij}$.

Algoritmo per APSP basato sulla programmazione dinamica e il prodotto tra matrici

Presa in input la matrice $W = (w_{ij})$, calcoliamo una serie di matrici $L^{(0)}, L^{(1)}, ..., L^{(n-1)}$ dove $L^{(r)} = (l_{ij}^{(r)})$ per $r = 0, ..., n-1$.

La matrice finale $L^{(n-1)}$ conterrà i pesi dei cammini minimi

$$L^{(n-1)} = (l_{ij}^{(n-1)}) = (d(i, j))$$

e pertanto sarà una soluzione al problema.

Il cuore dell'algoritmo è la procedura Extend-Shortest-Paths che estende da un arco i cammini minimi trovati all'iterazione precedente. Calcola $L^{(r)}$ conoscendo $L^{(r-1)}, W, n$.

```
extend-shortest-path(L(r-1),W,L(r),n)
1. for i = 1 to n
2.   for j = 1 to n
3.     for k = 1 to n
4.       lij(r) = min{lik^(r-1) + wkj, lij^(r)}
```

Quindi possiamo risolvera All-Pairs-Shortest-Paths "moltiplicando" ripetutamente delle matrici. Iniziando da $L^{(0)}$ e otteniamo:

$$L^{(1)} = L^{(0)} \cdot W = W$$

$$L^{(2)} = L^{(1)} \cdot W = W^2$$

$$L^{(3)} = L^{(2)} \cdot W = W^3$$

$$...$$

$$L^{(n-1)} = L^{(n-2)} \cdot W = W^{n-1}$$

L'algoritmo Slow-Apsp calcola questa sequenza in tempo $\Theta(V^4)$.

```
Slow-Apsp(W,L(0),n)ù
1. L = (lij), M = (mij) // due nuove matrici n * n
2. L = L(0)
3.   for r = 1 to n-1
4.     M = (∞)
5.     Extend-Shortest-Paths(L,W,M,n)
6.     L = M
7. return L
```

Ci sono $n-1$ invocazioni a Extend-Shortest-Paths che impiegano $\Theta(V^3)$ ognuna, dunque il tempo globale è $\Theta(n^4)$.

Osserviamo tuttavia che a noi interessa solo $L^{(n-1)}$ e non tutte le $L^{(r)}$ per $0 \leq r \leq n-1$. Possiamo calcolare $L^{(n-1)}$ eseguendo solo $\log(n-1)$ prodotti tra matrici usando le tecnica dei quadrati ripetuti.

L'algoritmo Faster-Apsp implementa questa idea e impiega $\Theta(n^3 \log n)$.

```
Faster-Apsp(W,n)
1. L,M
2. L = W
3. r = 1
4. while r < n-1
5.   M = (∞)
6.   Extend-Shortest-Path(L,L,M,n)
7.   r = 2r
8.   L = M
9. return L
```

Floyd-Warshall

L'algoritmo di Floyd Warshall risolve APSP in tempo $\Theta(V^3)$.

Un vertice intermedio di un cammino semplice $p =< v_1, v_2, ..., v_l >$ è un qualunque vertice di p diverso da v_1 e da v_l.

Enumeriamo i vertici di $G, V = \{1, ..., n\}$ e consideriamo un sottoinsieme $\{1, 2, ..., k\}$ di vertici per qualche $1 \leq k \leq n$.

Per ogni coppia di vertici (i, j) consideriamo tutti i cammini da i a j i cui vertici intermedi appartengono a $\{1, ..., k\}$ e sia p un cammino di peso minimo tra questi.

- Se k non è un vertice intermedio di p allora tutti i vertici intermedi di p appartengono a $\{1, .., k-1\}$;
- Se k è un vertice intermedio di p allora possiamo decomporre p in i \rarr k \rarr j e k non è un vertice intermedio

Rod-Cutting (da completare)

Abbiamo una barra di lunghezza n e una tabella di prezzi $p_i, i \in \{1, ..., n\}$ dei pezzi di lunghezza i.

Dunque ad ogni pezzo della barra verrà associato un prezzo p_i in base alla lunghezza.

L'obiettivo è quello di massimizzare il guadagno delle vendite.

$$R_n = max(p_{i1}, p_{i2}, ..., p_{ik}); i_1, ..., i_k, i_1 + ... + i_k = n$$

Dopo il primo taglio otteniamo due istanze che sono indipendenti:

...

Counting Sort

A differenza di altri algoritmi di ordinamento, questo algoritmo non si basa sul confronto di valori ma, come suggerisce il nome stesso, sul conteggio.

Per applicare il counting sort, è necessario conoscere a priori il range di valori contenuti nel vettore di partenza. L'idea su cui si basa questo algoritmo è quella di contare le occorrenze di ogni elemento o valore all'interno dell'array di partenza e successivamente costruire un secondo array con numero di elementi uguale al range di valori dell'array di

partenza. In ogni posizione i del nuovo array andremo ad indicare le occorrenze dell'elemento di valore i dell'array di partenza.

Successivamente, il secondo vettore subisce un'ulteriore trasformazione: in particolare, in ogni posizione dell'array andremo a inserire la somma dei valori nella posizione corrente e nella posizione precedente. Adesso sapremo esattamente, per ogni posizione i, quanti elementi minori o uguali al valore i esistono nel nostro vettore di partenza. Sapremo dunque esattamente dove posizionare l'elemento i nel vettore finale.

Esempio

Vettore di partenza:

$$A = [7][2][2][7][7][1][4][5][3][2]$$

Quindi il range di valori del nostro array di partenza è $\{1, ..., 7\}$. Costruiamo il secondo array:

$$B = [1][3][1][1][1][0][3]$$

$$B = [1][4][5][6][7][7][10]$$

Adesso partendo dall'ultimo dell'array di partenza vediamo per ogni elemento in quale posizione inserirlo nel nuovo array:

$$R = [1][2][2][2][3][4][5][7][7][7]$$

Vediamo quindi lo pseudocodice:

```
CountingSort(A,n,k)
1. for i = 1 to k
2.   B[i] = 0
3. for i = 1 to n
4.   B[A[i]] = B[A[i]]+1
5. for i = 2 to k
6.   B[i] = B[i] + B[i-1]
7. for i = n down to 1
8.   R[B[A[i]]] = A[i]
9.   B[A[i]] = B[A[i]] - 1
```