

INTRODUZIONE

Ci occuperemo di progettare e analizzare algoritmi. Familiarizziamo con alcune definizioni:

- Un **algoritmo** è una sequenza di passi computazionali che trasforma dei dati di input, in dati di output. Un algoritmo serve a risolvere un ben definito problema computazionale.
- I valori che diamo in ingresso ad un algoritmo vengono definiti come **istanze del problema**.
- Un **problema** è la descrizione di una relazione binaria; ovvero una relazione che ad un certo valore di input restituisce un determinato output.
- Un **algoritmo è corretto** se, per ogni istanza del problema, l'algoritmo termina fornendo l'insieme. Un algoritmo incorretto potrebbe, in corrispondenza di alcune istanze del problema, non terminare o terminare con un insieme non corretto di valori in uscita.
- Una **struttura dati** è una modalità di memorizzazione e organizzazione dei dati pensata per facilitare l'accesso e la manipolazione dei dati.
- **efficienza di un algoritmo** è misurata in termini di numero di operazioni, in funzione della dimensione dei dati in ingresso, richiesti per produrre i dati in uscita. Una metrica generalmente usata è la complessità asintotica.

OSS: il modello di compilazione influisce notevolmente sull'efficienza. Sfruttare bene cache e caratteristiche del SO, rende efficiente usare algoritmi che paragonati ad altri risulterebbero meno efficienti.

Un algoritmo restituisce un risultato corretto, a differenza un'**euristica** fornisce un risultato che può essere buono ma a meno di un margine di errore (non dà garanzie). Usiamo le euristiche perché altrimenti il problema non sarebbe trattabile con un algoritmo.

CAPITOLO 1 - ANALISI DI CORRETTEZZA E DI COMPLESSITÀ DEGLI ALGORITMI

Abbiamo già definito cos'è la correttezza di un algoritmo, ora dobbiamo vedere come possiamo valutare la correttezza di un algoritmo.

In primo luogo, definiamo cos'è un **problema di ordinamento**:

- data una sequenza di n valori di ingresso (a_1, a_2, \dots, a_n), determinare una permutazione (a'_1, a'_2, \dots, a'_n) della sequenza di ingresso tale che ($a'_1 \leq a'_2 \leq \dots \leq a'_n$).

Il problema è utile per varie applicazioni quali: ricerca, ricerca di elementi unici e così via. In generale diremo che:

- un algoritmo di ricerca **ordina sul posto** se non utilizza memoria aggiuntiva
- un algoritmo di ricerca è **stabile** se due oggetti con chiavi uguali appaiono nello stesso ordine nell'output ordinato come appaiono nell'array non ordinato di input.

Per comprendere al meglio l'analisi di correttezza e di complessità tratteremo algoritmi specifici.

INSERTION SORT

Correttezza

Un possibile algoritmo è l'**Insertion Sort**: considera un valore alla volta e lo inserisce nella corretta posizione tra i valori che lo precedono. È un algoritmo che ordina i valori sul posto: i valori sono riordinati all'interno dell'array stesso, con al più un numero costante di essi memorizzati al di fuori dell'array in ogni momento e risulta essere **stabile**. Vogliamo dimostrare che l'algoritmo è corretto,

```
Insertion-Sort (A)
for j ← 2 to length[A]
    do key ← A[j]
        // Insert A[j] into A[1..j-1]
        i ← j-1
        while i>0 and A[i]>key
            do A[i+1] ← A[i]
            i ← i-1
        A[i+1] ← key
```

sfruttando l'induzione matematica: dimostriamo il caso base, e poi dimostriamo che se è vero il passo i -esimo allora è vero anche il passo $(i+1)$ -esimo. Per dimostrare la correttezza, dobbiamo definire un'**invariante**. È una cosa che ripeteremo poche volte, ma la strategia è sempre la seguente. Definito l'invariante dobbiamo dimostrare che:

Inizializzazione: l'invariante è vera prima di iniziare il ciclo.

Conservazione: se l'invariante è vera prima di una iterazione del ciclo, rimane vero prima della successiva iterazione (passo induttivo).

Conclusione: quando il ciclo termina, l'invariante prova che

l'algoritmo è corretto.

All'inizio di ciascuna iterazione del ciclo for, la sottosequenza $A[1..j-1]$ consiste degli elementi originali di $A[1..j-1]$ ma ordinati in senso crescente.

Ne deriva che:

- L'invariante è vera prima della prima iterazione. Per $j=2$, la sottosequenza $A[1..j-1]$ è costituita solo da $A[1]$, che è ovviamente una sequenza ordinata.
- Una iterazione del ciclo conserva la verità dell'invariante infatti, l'elemento $A[j]$ viene inserito in modo da mantenere la sequenza $A[1..j]$ ordinata.

L'invariante prova che l'algoritmo è corretto; infatti, il ciclo termina quando $j=n+1$, ovvero quando la sequenza $A[1..n]$ è ordinata.

OSS: generalmente gli indici degli array iniziano da 1 e non da 0. Molto spesso più che dimostrare la correttezza, risulta più opportuno e semplice dimostrare l'**incorrettezza**, ad esempio usando input per cui l'algoritmo fallisce. È lo stesso principio che si adopera col testing. L'analisi di incorrettezza va sempre adoperata congiuntamente all'analisi formale di correttezza.

Correttezza e Induzione

C'è

e

Come abbiamo accennato prima, abbiamo questo legame tra correttezza ed induzione, in particolare possiamo osservare che dimostrare la correttezza di sommatorie è una applicazione classica di induzione (torna utile nell'analisi degli algoritmi). Vediamo un esempio.

Esempio 1. Partiamo dal seguente problema: provare che con l'induzione che:

$$\sum_{i=1}^n i \cdot i! = (n + 1)! - 1.$$

Dimostriamo il caso base: ponendo $n=1$ otteniamo che $1 \cdot 1! = 2! - 1 \Rightarrow 1 = 1$ e quindi l'uguaglianza è verificata.

Ora vogliamo dimostrare che se l'ipotesi è vera per n allora è vera anche per $(n+1)$. Se il problema è vero per $(n+1)$, allora dobbiamo avere che $\sum_{i=1}^{n+1} i \cdot i! = ((n + 1) + 1)! - 1$. Vediamo i seguenti calcoli:

$$\begin{aligned} \sum_{i=1}^{n+1} i \cdot i! &= (n + 1)(n + 1)! + \sum_{i=1}^n i \cdot i! = (n + 1)(n + 1)! + (n + 1)! - 1 = \\ &= (n + 1)! \cdot (n + 1 + 1) - 1 = (n + 2)! - 1 = ((n + 1)! + 1) - 1 \end{aligned}$$

Possiamo quindi osservare che la tesi è dimostrata.

La dimostrazione induttiva può essere forte o debole.

Analisi di complessità

Analizzare un algoritmo significa stimare le risorse richieste in termini di:

- Occupazione di memoria.
- Impegno di banda.
- Tempo di calcolo.

I casi più tipici sono occupazione di memoria e tempo di calcolo. Occorre definire il modello del calcolatore usato per eseguire l'algoritmo. Adotteremo il seguente:

- Singolo processore.
- Memoria ad accesso casuale.
- Assenza di gerarchie di memoria (no caching).

Ulteriore assunzione che faremo è che le istruzioni aritmetiche, di accesso alla memoria e di controllo richiedono un tempo costante. Questa è un'assunzione forte, dato che non tutte le espressioni hanno lo stesso costo. Se utilizziamo modelli diversi, ad esempio architetture con parallelismo, dovremo fare osservazioni diverse.

Il tempo di esecuzione di un algoritmo è solitamente espresso in funzione della dimensione dei valori in ingresso:

- Per algoritmi di ordinamento: lunghezza della sequenza da ordinare.
- Per algoritmi che operano su grafi: numero di vertici e numero di archi.

Il tempo di esecuzione è dato dalla somma dei tempi di esecuzione di ciascuna linea dello pseudocodice (che assumiamo essere costanti), ciascuno moltiplicato per il numero di volte che la linea viene eseguita.

Analisi di complessità: Insertion Sort

<u>Insertion-Sort (A)</u>	<i>cost</i>	<i>times</i>
for $j \leftarrow 2$ to $\text{length}[A]$	c_1	n
do $\text{key} \leftarrow A[j]$	c_2	$n-1$
// Insert $A[j]$ into $A[1..j-1]$	c_4	$n-1$
$i \leftarrow j-1$	c_5	$\sum_{j=2}^n t_j$
while $i > 0$ and $A[i] > \text{key}$	c_6	$\sum_{j=2}^n (t_j - 1)$
do $A[i+1] \leftarrow A[i]$	c_7	$\sum_{j=2}^n (t_j - 1)$
$i \leftarrow i-1$	c_8	$n-1$
$A[i+1] \leftarrow \text{key}$		

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

Nel caso migliore, quando il vettore è ordinato allora t_j è sempre pari ad 1, dato che al confronto l'elemento $A[i] < A[i+1]$; da cui l'espressione diventa $= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$. Il tempo di esecuzione è quindi una funzione lineare.

Nel caso peggiore, quando il vettore è ordinato in senso decrescente, allora $t_j = j$, ovvero vengono fatti j confronti. Da cui $\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$ e $\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$. Da cui, l'espressione diventa del tipo $T(n) = an^2 + bn + c$ e quindi il tempo di esecuzione è una funzione quadratica.

OSS: Le costanti dipendono dal tempo di esecuzione delle istruzioni, ma nel calcolo della complessità asintotica non considero tali parametri e non tengo conto dei termini di ordine inferiore.

Tipicamente si considera il tempo di esecuzione nel caso peggiore. Questo:

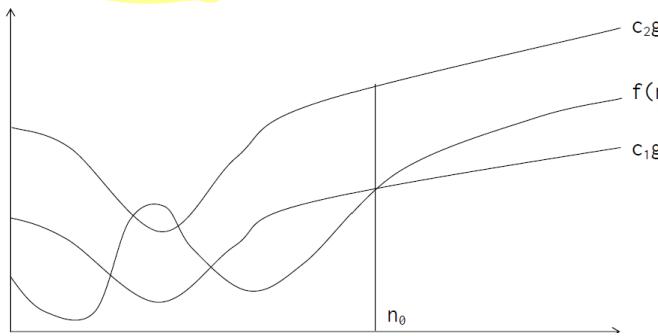
- È un limite superiore per il tempo di esecuzione di qualsiasi istanza.
- Per alcuni algoritmi, il caso peggiore si presenta piuttosto spesso.
- Il caso "medio" non è tipicamente molto migliore del caso peggiore, inoltre il caso medio è più ostico da valutare.

Per indicare sinteticamente il tempo di esecuzione di un algoritmo, si tralasciano le costanti e si indica solo il termine dominante. Ad esempio, l'Insertion Sort ha un tempo di esecuzione nel caso peggiore di $\Theta(n^2)$.

Notazione asintotica e crescita delle funzioni

Mediane tali notazioni possiamo caratterizzare l'efficienza di un algoritmo mediante il tempo di esecuzione dell'algoritmo. Quando la dimensione dei dati in ingresso è tale da rendere rilevante solo l'ordine di crescita del tempo di esecuzione, stiamo studiando l'**efficienza asintotica** di un algoritmo.

Notazione Θ → Theta



Data la funzione $g(n)$, $\Theta(g(n))$ denota l'insieme di funzioni:

$\Theta(g(n)) = \{f(n): \text{esistono costanti positive } c_1, c_2 \text{ e } n_0 \text{ tali che } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0\}$.

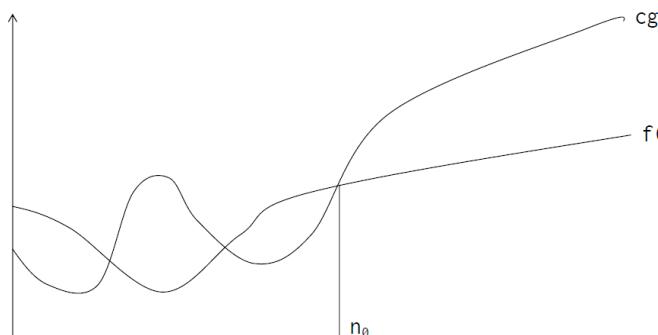
Usiamo (impropriamente) $f(n) = \Theta(g(n))$ anziché $f(n) \in \Theta(g(n))$.

$g(n)$ è un limite asintotico stretto per $f(n)$.

Nel calcolo del limite asintotico è possibile ignorare i termini della funzione di ordine inferiore. Ad esempio. $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Per verificarlo, occorre trovare c_1, c_2 e n_0 : $0 \leq c_1 \cdot n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 \cdot n^2 \forall n \geq n_0$. Dividendo per n^2 otteniamo: $c_1 \leq \frac{1}{2} - 3/n \leq c_2$ per ogni $n \geq n_0$.

La diseguaglianza a destra vale per $n \geq 1$ scegliendo $c_2 \geq \frac{1}{2}$; invece, la diseguaglianza a sinistra vale per $n \geq 7$ scegliendo $c_1 \leq \frac{1}{14}$. La definizione è verificata con $c_1 = 1/14$, $c_2 = \frac{1}{2}$ e $n_0 = 7$.

Notazione O



Data la funzione $g(n)$, $O(g(n))$ - detta anche "notazione O grande" - denota l'insieme di funzioni:

$O(g(n)) = \{f(n): \exists c > 0 \text{ e } n_0: 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}$.

$g(n)$ è un limite superiore asintotico per $f(n)$.

Notazione O e Notazione Θ

Vediamo le seguenti osservazioni:

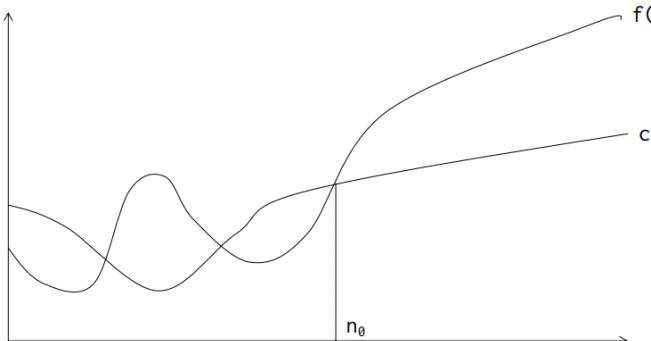
- $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$
- $\Theta(g(n)) \subseteq O(g(n))$
- Infatti: $an + b \neq \Theta(n^2)$ ma $an + b = O(n^2)$

Se il tempo di esecuzione di un algoritmo nel caso peggiore è $O(g(n))$, il tempo di esecuzione per qualunque ingresso è sempre $O(g(n))$. La notazione O fornisce un limite superiore.

Se il tempo di esecuzione di un algoritmo nel caso peggiore è $\Theta(g(n))$, non è detto che il tempo di esecuzione per qualunque ingresso sia $\Theta(g(n))$. Ad esempio, Insertion-Sort: caso peggiore $\Theta(n^2)$ ma se la sequenza è ordinata allora abbiamo una complessità $\Theta(n)$.

Notazione Ω

omaggio



Data la funzione $g(n)$, $\Omega(g(n))$ denota l'insieme di funzioni:

$$\Omega(g(n)) = \{f(n) : \exists c > 0 \text{ e } n_0 :$$

$$0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\}.$$

$g(n)$ è un limite inferiore asintotico per $f(n)$.

Notazioni a confronto

Vale la seguente relazione: $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ e } f(n) = \Omega(g(n))$.

Se il tempo di esecuzione di un algoritmo nel caso migliore è $\Omega(g(n))$, il tempo di esecuzione per qualunque ingresso è sempre $\Omega(g(n))$.

La notazione Ω fornisce un limite inferiore. Il tempo di esecuzione di Insertion Sort è $\Omega(n)$ e $O(n^2)$.

Quando una notazione asintotica appare in una equazione va interpretata come una funzione ignota che non ci interessa specificare. Ad esempio, $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ equivale a $2n^2 + 3n + 1 = 2n^2 + f(n)$, dove $f(n)$ è una qualunque funzione nell'insieme $\Theta(n)$.

Valgono inoltre le seguenti proprietà:

- **Transitiva** (valida per O , Θ , Ω):

$$f(n) = \Theta(g(n)) \text{ e } g(n) = \Theta(h(n)) \text{ allora } f(n) = \Theta(h(n))$$

- **Riflessiva** (valida per O , Θ , Ω):

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

- **Simmetrica**:

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

- **Antisimmetrica**:

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

Selection Sort

L'algoritmo seleziona di volta in volta il numero minore nella sequenza di partenza e lo sposta nella sequenza ordinata; di fatto la sequenza viene suddivisa in due parti: la sottosequenza ordinata, che occupa le prime posizioni dell'array, e la sottosequenza da ordinare, che costituisce la parte restante dell'array. Quest'algoritmo ordina sul posto ma non è stabile.

```
selection_sort(int s[], int n) {
    int i,j;          /* counters */
    int min;          /* index of minimum */
    for (i=0; i<n; i++) {
        min=i;
        for (j=i+1; j<n; j++)
            if (s[j] < s[min]) min=j;
        swap(s[i],s[min]);
    }
}
```

Di questo algoritmo ne vogliamo stimare l'**efficienza asintotica**. In primo luogo, determiniamo il costo delle varie istruzioni. Il caso peggiore è quando il vettore è ordinato in ordine decrescente e quindi si va nel ramo true dell'IF, analogamente il caso migliore è quando il vettore è già ordinato.

int i, j;	c_1	1
int min;	c_2	1
for (i=0; i<n ; i++)	c_3	$n+1$
min=i;	c_4	n
for (j=i+1; j<n ; j++)	c_5	$\sum_{i=0}^n (n - i)$
if (s[j]<s[min]) min=j;	c_6	$\sum_{i=0}^{n-1} (n - i - 1)$
swap(s[i],s[min]);	c_7	n

$$T(n) = c_1 + c_2 + c_3(n + 1) + c_4n + c_5 \left(n^2 + n - \frac{n(n + 1)}{2} \right) + c_6 \left(n^2 - \frac{n(n + 1)}{2} - n \right) + c_7 n$$

Da cui:

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

Possiamo osservare che nel caso migliore il contributo dell'IF risulta nullo ma la complessità rimane sempre $\Theta(n^2)$.

OSS: Il primo ciclo for si può fermare a $n-1$.

Per quanto riguarda l'**analisi di correttezza** possiamo osservare che:

- L'invariante è che la sottosequenza di destra, ovvero $[0, i - 1]$ è sempre ordinata in senso crescente.
- All'inizio ($i=0$) la sottosequenza di destra $[0, 0 - 1 = -1]$ è vuota e quindi per definizione è ordinata.
- Se l'invariante è vera all'istante i avrà due vettori uno non ordinato e uno ordinato (quello $[0, i - 1]$) con gli elementi più piccoli dell'array ordinati. All'istante $i+1$ andrà a cercare il minimo nella sequenza non ordinata $([i, n])$ e lo porrà in testa ed ha quindi creato un nuovo vettore ordinato con i valori più piccoli dell'array.

Progettazione Algoritmi: Divide et Impera

Per la progettazione dell'algoritmo merge sort è fondamentale introdurre il **paradigma divide et impera**. L'idea è la seguente:

- Si suddivide il problema in sottoproblemi più semplici.
- Si risolvono **ricorsivamente** i sottoproblemi.
OSS: Ci riferiamo ad una ricorsione matematica non di programmazione.
- Si combinano le soluzioni dei sottoproblemi per ottenere la soluzione del problema di partenza.

Soltanamente, il tempo di esecuzione degli algoritmi basati sull'approccio del divide et impera si determina utilizzando semplici tecniche che vedremo nel dettaglio.

Prendiamo come riferimento il gioco: Torre di Hanoi.

Merge Sort

Mentre con l'algoritmo Insertion Sort si adotta un approccio incrementale, per progettare un algoritmo di ordinamento più efficiente si può utilizzare l'approccio divide et impera.

L'approccio che si usa è il seguente:

- Si suddivide una sequenza di n elementi in due sottosequenze di $n/2$ elementi.
- Si ordinano le sottosequenze ricorsivamente.

In particolare, si divide fino ad ottenere una sottosequenza di lunghezza unitaria (ordinata per definizione) e poi si combinano ("merge") due sottosequenze ordinate per produrre l'intera sequenza ordinata. Per tale scopo usiamo due funzioni:

- **Merge:** combina due sottosequenze ordinate in una sequenza ordinata.
- **Merge-Sort:** ordina una sequenza suddividendola in due sottosequenze, ordinando ciascuna sottosequenza (mediante chiamata ricorsiva) e invocando la funzione Merge per ottenere la sequenza ordinata.

Questo tipo di algoritmo non effettua un ordinamento sul posto a differenza dell'Insertion Sort. Vediamo le funzioni nel dettaglio.

Merge-Sort

```
Merge-Sort (A,p,r)
if p < r
    q ← ⌊(p+r)/2⌋
    Merge-Sort (A,p,q)
    Merge-Sort (A,q+1,r)
    Merge (A,p,q,r)
```

Se vogliamo ordinare il vettore A con indice iniziale p e indice finale r, ciò che facciamo è in primo luogo dividere il vettore fino ad arrivare a vettori fatti da un unico elemento. Posso fare ciò mediante le due chiamate ricorsive a Merge-Sort in cui uso l'indice q che ricade nella metà del vettore.

Merge

```
Merge (A,p,q,r)
n1 ← q-p+1
n2 ← r-q
// create L[1..n1+1] and R[1..n2+1]
for i ← 1 to n1
    do L[i] ← A[p+i-1]
for j ← 1 to n2
    do R[j] ← A[q+j]
L[n1+1] ← R[n2+1] ← ∞
i ← j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
            i ← i+1
        else A[k] ← R[j]
            j ← j+1
```

Se dobbiamo fondere due sequenze $[p, q]$ e $[q + 1, r]$ creiamo due vettore L (left) e R (right) in cui andiamo ad inserire le due sottosequenze. Osserviamo che usiamo il marcatore ∞ per delimitare la fine dei vettori. Col ciclo finale ciò che facciamo è fondere i vettori.

Analisi di Correttezza: Merge

L'invariante è che all'inizio di ciascuna iterazione dell'ultimo ciclo for, la sottosequenza $A[p..k - 1]$ contiene i $k - p$ elementi più piccoli di L ed R ordinati. Inoltre, $L[i]$ e $R[j]$ sono i più piccoli elementi dei loro array a non essere stati ricoppiati in A .

L'invariante è vera prima della prima iterazione. Per $k=p$, la sottosequenza $A[p..k - 1]$ è vuota ($k - p = 0$). $L[1]$ e $R[1]$ sono i più piccoli elementi dei loro array a non essere stati ricoppiati in A .

Un'iterazione del ciclo conserva la verità dell'invariante infatti:

- $A[p..k - 1]$ contiene i $k - p$ elementi più piccoli di L ed R ordinati.
- $L[i]$ e $R[j]$ sono i più piccoli elementi dei loro array a non essere stati ricoppiati in A .
- Se $L[i] \leq R[j]$, $L[i]$ viene ricoppiato in $A[k] \Rightarrow A[p..k]$ contiene i $k - p + 1$ elementi più piccoli di L ed R ordinati. Incrementando k ed i , si ristabilisce l'invariante per la successiva iterazione.
- Analogo discorso se $R[j] < L[i]$.

L'invariante prova che l'algoritmo è corretto; infatti, il ciclo termina quando $k = r + 1$, quindi la sequenza $A[p..k - 1]$ (i.e., $A[p..r]$) contiene i $k - p$ ($= r - p + 1$) elementi più piccoli di L e R ordinati. Gli $r - p + 1$ elementi più piccoli di L e R coincidono con tutti gli elementi originariamente in $A[p..r]$. In L e R restano i due valori sentinella.

Analisi di Complessità: Merge

In questo caso l'analisi è molto semplice, basta osservare che il caso peggiore è quando ordiniamo le sequenze ottenute dopo il primo split. Abbiamo due cicli for iniziali che creano i due vettori che insieme hanno una complessità $n = n_1 + n_2$; e un ciclo for che scorre tutti i due vettori la complessità $O(n)$.

Analisi di complessità Merge-Sort

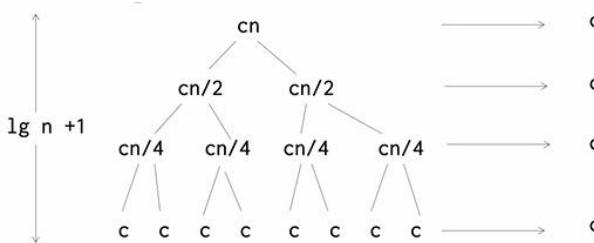
In generale sfruttiamo per tale calcolo un approccio valido generalmente per la metodologia divide et impera. Detto:

- $\Theta(1)$: il tempo per risolvere il caso "banale".
- $D(n)$: il tempo per suddividere un problema.
- $C(n)$: il tempo per ricombinare le soluzioni dei sottoproblemi.
- a : il numero di sottoproblemi in cui si divide un problema.
- $1/b$: il fattore di riduzione della dimensione di un problema.

Si ha che la complessità è esprimibile in forma di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{per il caso base } n = 1 \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{per } n > 1 \end{cases}$$

Nel Merge-Sort $D(n) = \Theta(1)$, $C(n) = \Theta(n)$ e $a = b = 2$. Possiamo infatti osservare che a partire dalla sequana base, la si divide in due sottosequenze e così via ($a = 2$) fino ad arrivare a sequenze unitarie ($b=1$).



Se c è il costo per il caso base, allora posso individuare la struttura sulla sinistra e posso osservare che il costo totale è:

$$cn [\log_2(n) + 1]$$

c è il costo per ogni passo, in realtà al passo zero sarà cn , al secondo passo $2cn$, al terzo $4cn$ e così via, ma poiché siamo interessati alla complessità consideriamo cn . Dall'espressione precedente ricaviamo che la complessità è:

$$T(n) = (n \log_2(n))$$

OSS: Per semplicità useremo la notazione $\lg n = \log_2(n)$

Problema del massimo sottoarray (Trova su internet)

Trattiamo questo problema per osservare che una soluzione bruteforce è meno efficiente di una risolta col paradigma divide et impera. Si immagina di voler comprare e vendere delle azioni e per ogni giorno abbiamo l'andamento del prezzo dell'azione. Vogliamo trovare la coppia di giorni in cui compro ad un prezzo e vendono in modo tale che val_vendita-val_acquisto sia massimo. Un primo approccio è vedere tutte le coppie ordinate che sarà dato da $\binom{n}{2}$ che cresce quarticamente. Un secondo approccio prevede l'utilizzo dell'approccio divide et impera. Questo esempio è importante per capire che non dobbiamo fermarci alla soluzione più ovvia ma dobbiamo trasformare il problema per adottare soluzioni più efficienti. (approfondisci libro)

Ricorrenze

Una ricorrenza è un'equazione o disequazione che descrive una funzione in termini del suo valore su ingressi di dimensione inferiore. Ad esempio, il Merge-Sort può essere descritto dalla ricorrenza:

$$T(n) = \begin{cases} \Theta(1) \text{ per il caso base } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) \text{ per } n > 1 \end{cases}$$

Possiamo adottare tre metodi per risolvere una ricorrenza:

- Metodo di sostituzione.
- Metodo dell'albero di ricorrenza.
- Metodo dell'esperto.

Ricorrenze: Metodo di sostituzione

Il metodo di sostituzione consiste di due passi

- Ipotizzare la forma della soluzione
- Verifica tramite principio di induzione

Può essere usato per derivare sia limiti superiori che inferiori su una ricorrenza. Ad esempio: abbiamo la ricorrenza $T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$ e utilizziamo $T(1) = 1$ ai fini della verifica.

Supponiamo che la soluzione sia $T(n) = O(n \lg n)$; occorre provare che $T(n) \leq cn \lg n$ per una data costante c e per n sufficientemente grande. A questo punto consideriamo due casi:

1. **Passo induttivo:** assumiamo il passo vero per $\left\lfloor \frac{n}{2} \right\rfloor$, dimostriamo per n .

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \quad (1)$$

Osserviamo che per ipotesi $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) = \left\lfloor \frac{n}{2} \right\rfloor \lg \left\lfloor \frac{n}{2} \right\rfloor$ è che possiamo maggiorare questa quantità con $\frac{cn}{2} \lg \frac{n}{2}$; dove c è una costante positiva. Da tale osservazione ricaviamo che:

$$T(n) \leq c n \cdot \lg\left(\frac{n}{2}\right) + n = c n \cdot \lg n - cn \cdot \lg 2 + n = cn \cdot \lg n - (c-1)n$$

$$\text{Da cui: } T(n) \leq cn \cdot \lg n \text{ se } c \geq 1 \quad (2)$$

2. **Passo Base:** Per $n = 1$: $T(1) \leq c \lg 1 = 0$, il tempo di esecuzione non può essere ≤ 0 .

Dobbiamo quindi trovare un'ipotesi per il caso base migliore per validare il passo induttivo.

Possiamo utilizzare $T(3)$ e $T(2)$ ($T(n) \leq cn \lg n$ che deve valere per $n \geq n_0$).

Occorre mostrare che $T(2) = 4 \leq c2 \lg 2$ e $T(3) = 5 \leq c3 \lg 3$; per fare ciò è sufficiente che $c \geq 2$ per verificare i casi base.

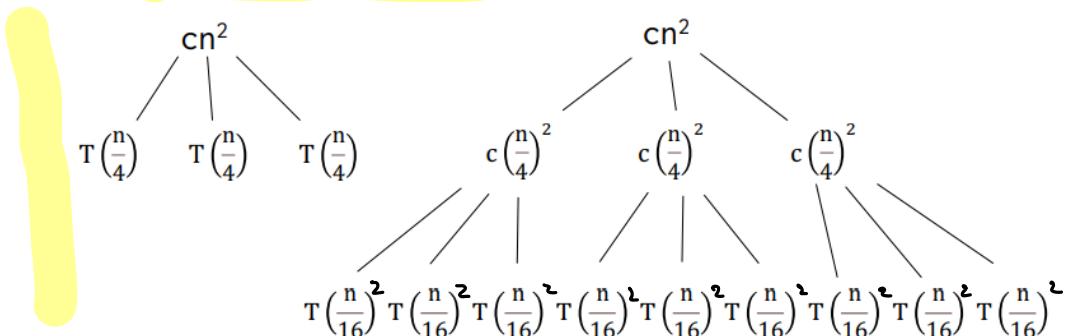
		16		
	8	17		12
4	9	18	6	13
2	10	19	3	14
	11	20	7	15
		21		
		22		
		23		

Non possiamo usare solo $n = 2$ questo perché se andiamo a sostituire nella (1) abbiamo nuovamente $T(1)$ e quindi da sola non va bene e per tale motivo uso $n=3$. Osserviamo che non posso neanche usare solo $T(3)$ perché riporterebbe nuovamente a $T(1)$. Devo quindi usarli entrambi in modo tale che per $n > 3$

arriviamo sempre ai casi base $T(2)$ o $T(3)$ che sono accettabili. Ad esempio, $T(4)$ riporta a $T(2)$ e $T(6)$ riporta a $T(3)$. Con il caso base devo coprire i casi che non copre il passo induttivo. Deduciamo che non sempre il passo base è composto da un solo passo.

Ricorrenze: Metodo dell'albero di ricorrenza

Il metodo dell'albero di ricorrenza può essere usato per generare delle buone soluzioni di tentativo eventualmente da verificare con il metodo di sostituzione se sono state fatte ipotesi semplificative per derivare le soluzioni. I nodi dell'albero di ricorrenza riportano il costo per risolvere un dato (sotto)problema. È l'approccio usato per il Merge Sort. Consideriamo $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ e costruiamo l'albero di ricorrenza per $T(n) = 3T(n/4) + cn^2$. Per fare ciò, ignoriamo l'arrotondamento per difetto (approssimazione) e supponiamo che n sia una potenza di 4 (approssimazione). Esplicitiamo inoltre il coefficiente costante.



Osserviamo che a partire dal problema iniziale, divido il problema in 3 sottoproblemi di dimensione $n/4$. Ovviamente la complessità per risolvere il sottoproblema $n/4$ è $\left(\frac{n}{4}\right)^2$. L'albero ha $\log_4(n) + 1$ livelli; infatti, in un nodo al livello i (dove con $i=0$ indichiamo livello della radice) la dimensione dei sottoproblemi è $n/4^i$ e la dimensione 1 si raggiunge per $n/4^i = 1$ ovvero $i = \log_4(n)$. Il numero di nodi al livello i è invece 3^i in cui il costo di un singolo nodo è $c(n/4^i)^2$. In generale il livello i esimo avrà complessità $3^i c(n/4^i)^2$. All'ultimo livello come detto avrà complessità $T(1)$ e quindi posso scrivere che:

$$T(n) = \sum_{i=0}^{\log_4(n)-1} 3^i c \left(\frac{n}{4^i}\right)^2 + 3^{\log_4(n)} T(1)$$

Ovvero la complessità è data dalla somma delle complessità dal livello 0 al penultimo livello a cui aggiungiamo la complessità dell'ultimo livello. Possiamo riscrivere la precedente come:

$$T(n) = \sum_{i=0}^{\log_4(n)-1} cn^2 \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3})$$

Questo perché:

$$3^{\log_4(n)} T(1) = n^{\log_4(3)\log_4(n)} = n^{\log_4(n)\log_4(3)} = \text{cambio di base} = n^{\log_4(n)\frac{\log_4(3)}{\log_4(n)}} = n^{\log_4 3}$$

Infine:

$$T(n) < \sum_{i=0}^{\infty} cn^2 \left(\frac{3}{16}\right)^i + \Theta(n^{\log_4 3}) \xrightarrow{\text{serie geometrica}} \frac{1}{1 - (\frac{3}{16})} cn^2 + \Theta(n^{\log_4 3}) \Rightarrow T(n) = O(n^2)$$

Osserviamo che non abbiamo una complessità $n \lg(n)$ come ci aspettavamo, a questo punto dimostriamo il tutto con il metodo della sostituzione. Partiamo col passo induttivo e verifichiamo che per: $T(n) = O(n^2), \exists d > 0 : T(n) \leq dn^2$.

$$\begin{aligned}
 T(n) &= 3T(\lfloor n/4 \rfloor) + \Theta(n^2) \leq 3T(\lfloor n/4 \rfloor) + cn^2 \leq 3d \left\lfloor \frac{n}{4} \right\rfloor^2 + cn^2 \leq 3d \left(\frac{n}{4} \right)^2 + cn^2 \\
 &= \left(\frac{3}{16} \right) dn^2 + cn^2 \leq dn^2 \text{ se } d \geq (16/13)c.
 \end{aligned}$$

A questo punto passiamo al caso base: per $n = 1$ $T(1) \leq d$ che è verificato per $d \geq T(1)$. In realtà si può dimostrare che $T(n) = \Omega(n^2)$ e quindi $T(n) = \Theta(n^2)$.

Ricorrenze: Metodo dell'Esperto

Un terzo modo di dimostrare le ricorrenze si basa sul teorema dell'esperto. Questo non si può applicare sempre ma devono valere delle ipotesi che vediamo nel seguente teorema.

Teorema dell'Esperto

Siano $a \geq 1$ e $b > 1$ costanti, $f(n)$ una funzione e $T(n) = aT(n/b) + f(n)$ dove n/b può essere interpretato sia come $\lfloor n/b \rfloor$ che $\lceil n/b \rceil$. Ricordiamo che $f(n)$ è la funzione della complessità del sottoproblema.

Allora:

1. Se $f(n) = O(n^{\log_b(a)-\varepsilon})$ per un dato $\varepsilon > 0$, allora $T(n) = \Theta(n^{\log_b(a)})$.
2. Se $f(n) = \Theta(n^{\log_b(a)})$, allora $T(n) = \Theta(n^{\log_b(a)} \lg(n))$.
3. Se $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ per un dato $\varepsilon > 0$ e $af(n/b) \leq cf(n)$ per una data $c < 1$ ed n sufficientemente grande, allora $T(n) = \Theta(f(n))$.

OSS:

- il termine $n^{\log_b(a)}$ ci dice che la complessità del sottoproblema dipende dal numero di sottoproblemi e dalla dimensione del sottoproblema.
- ε ci dice che le funzioni $f(n)$ differiscono per un fattore polinomiale n^ε . Diremo che $f(n)$ deve essere polinomialmente più grande o più piccola di $n^{\log_b(a)}$ (In base al fatto che ci sia \pm).
- Ci sono dei gap tra il caso 1) e 2) e tra 2) e 3). Infatti, tra 1) e 2) ci sono funzioni più piccole di $n^{\log_b(a)}$ ma non polinomialmente, analogamente tra caso 2) e 3).

Vediamo degli esempi pratici:

$$\text{ES_1} \quad T(n) = 9T(n/3) + n$$

- $a = 9, b = 3, f(n) = n$ e $n^{\log_b(a)} = n^2$
- Usando la 1 otteniamo che $n = f(n) = O(n^{\log_3(9)-\varepsilon})$ per $\varepsilon = 1$
- Quindi $T(n) = \Theta(n^2)$

$$\text{ES_2} \quad T(n) = T(2n/3) + 1$$

- $a = 1, b = 3/2, f(n) = 1$ e $n^{\log_b(a)} = n^0 = 1$
- Usando la 2 otteniamo che $1 = f(n) = \Theta(n^{\log_b(a)}) = \Theta(1)$
- Quindi $T(n) = \Theta(\lg n)$

$$\text{ES_3} \quad T(n) = 3T(n/4) + n \lg n$$

- $a = 3, b = 1/4, f(n) = n \lg n$ e $n^{\log_b(a)} = n^{\log_4(3)}$
- Usando la 3 otteniamo che $n \lg n = f(n) = \Omega(n^{\log_4(3)+\varepsilon})$ con $\varepsilon \approx 0,2$ infatti basta che $n^{\log_4(3)+\varepsilon} \leq n < n \lg n$ e poiché $\log_4(3) \approx 0,793$ la scelta di ε valida la disequazione. Dobbiamo verificare inoltre che $af(n/b) \leq cf(n)$ per una data $c < 1$ ed n sufficientemente grande. Quindi $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ con $c = 3/4$.
- Quindi $T(n) = \Theta(n \lg n)$

$$ES_4 \quad T(n) = 2T(n/2) + n \lg n$$

- $a = 2, b = 2, f(n) = n \lg n$ e $n^{\log_b(a)} = n$
- $f(n) = n \lg n$ non è polinomialmente più grande di n , infatti $\frac{n \lg n}{n} = \lg n$ che è asintoticamente più piccolo di n^ε .
- Questa ricorrenza ricade nel gap tra i casi 2) e 3)

OSS: $a = 1$ si ha ad esempio nella ricerca binaria, dato che nello split si considera solo un lato a cui poi si applica lo split. Esistono delle generalizzazioni per essere applicate a casi meno vincolati (ma che non vedremo).

\

HEAPSORT

Finora abbiamo trattato:

- Insertion Sort: che ordina sul posto ed ha complessità $\Theta(n^2)$ nel caso peggiore, $\Theta(n)$ nel caso migliore.
- Merge Sort: che non ordina sul posto ed ha complessità $\Theta(n \lg n)$.

Trattiamo ora l'Heap Sort che ordina sul posto ed ha complessità $\Theta(n \lg n)$. Questo algoritmo si basa sulla struttura dati dell'Heap.

Code a priorità

Introduciamo in primo luogo il concetto di code a priorità. Vengono usate nello scheduling e caratterizzata da due puntatori, una sulla testa e l'altro sulla coda. In testa abbiamo sempre l'elemento a maggior priorità (in base alla metrica usata) e questo rende molto efficiente l'estrazione dell'elemento che si desidera (quello a maggior priorità).

Vediamo le seguenti operazioni:

- *Insert(S, x)*: Insert x in S (seguendo la metrica).
- *Max(S)*: return max.
- *Extract_max(S)*: return max & remove max.
- *Increase_key(S, x, k)*: va a modificare il valore di x con il valore di k.

Una coda può essere implementata utilizzando la struttura dati Heap.

Heap

Un heap è una struttura dati, rappresentato da un array che può essere visualizzato come un albero che gode della proprietà **max heap property**.

OSS: è importante osservare che non è un albero, ma un ~~array~~ visualizzato come albero.

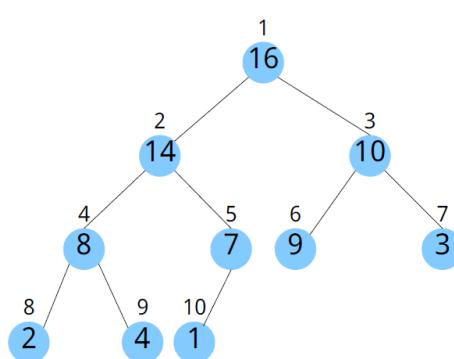
Max heap property: $key[x] \geq key[\text{children}(x)]$. In modo del tutto duale possiamo definire la proprietà **min heap** per cui $key[x] \leq key[\text{children}(i)]$

Nell'array possiamo distinguere:

- Root: primo elemento dell'array i=0.
- Parent(i): $i/2$
- Left(i) = $2i$
- Right(i) = $2i + 1$

Ad esempio:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

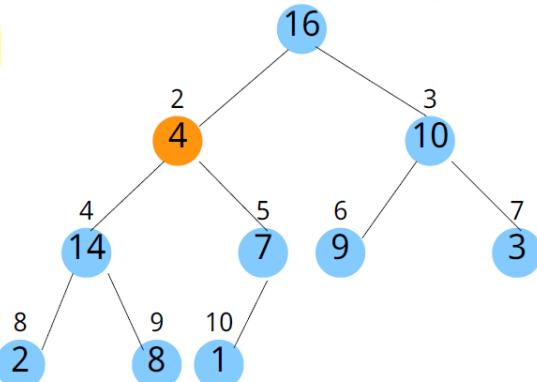


Possiamo verificare che la struttura data rispetta la priorità del max heap. Questo tipo di struttura permette di ottimizzare l'estrazione del massimo; infatti, l'estrazione del massimo ha un tempo costante. Vediamo le operazioni realizzabili:

- 1) Voglio produrre un max heap a partire da un array non ordinato: *Build_max_heap(A)*
- 2) Correggere una singola violazione: *Max_heapify(A, i)*

Max_heapify

In questo caso, dobbiamo fare l'assunzione che *Left(i)* e *Right(i)* sono max heap.



Al nodo $i=2$, la proprietà del max heap è violata e Max_heapify deve risolvere questa violazione. Per risolvere tale violazione, dobbiamo scambiare 4 con 14. Questo non è sufficiente, infatti, potrebbe essere violata la proprietà sul figlio del nodo scambiato e quindi dobbiamo riapplicare Max_heapify sul nodo scambiato, nel nostro caso il figlio destro e quindi $2i$. Ripete ↗

Max-Heapify (A, i)

```

l ← Left(i)
r ← Right(i)
largest ← i
if l≤heap-size[A] and A[l]>A[largest]
    largest ← l
if r≤heap-size[A] and A[r]>A[largest]
    largest ← r
if largest ≠ i
    exchange A[i]↔A[largest]
    Max-Heapify (A, largest)
  
```

Possiamo vedere l'implementazione nell'immagine successiva. Per quanto riguarda la complessità osserviamo che:

- 1) Il tempo per individuare il massimo fra tre ed effettuare lo scambio è costante.
- 2) Al precedente dobbiamo aggiungere il tempo per eseguire Max-Heapify su problema di dimensione minore.

In un certo senso l'algoritmo lavora meglio quando l'albero è pieno.

Valutiamo ora la complessità del sottoproblema. Detta:

- n : dimensione del problema originario.
- m : dimensione del sottoalbero con più elementi.
- h : altezza del nodo oggetto del problema originario.

Consideriamo due casi:

Caso 1: l'albero è pieno

$$n = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

$$m = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

$$\frac{m}{n} = \frac{2^h - 1}{2^{h+1} - 1} = \frac{2^h - 1}{2 \cdot 2^h - 1} \xrightarrow[h \rightarrow \infty]{\quad} \frac{1}{2}$$

La dimensione del problema originario n è la somma di tutti i nodi dell'albero che possiamo ricavare dalla prima serie (serie geometrica). La dimensione del sottoproblema è data dai sottoalberi del nodo radice che è $\sum_{i=1}^h 2^i =$ (ponendo $k = i - 1$) $= \sum_{k=0}^h 2^i$. Valutando m/n a limite, siamo in grado di individuare la complessità m , che è lineare.

Caso 2: l'ultimo livello è pieno a metà

$$n = (2^h - 1) + (2^{h-1} - 1) + 1 = 3 \cdot 2^{h-1} - 1$$

$$m = 2^h - 1$$

$$\frac{m}{n} = \frac{2^h - 1}{3 \cdot 2^{h-1} - 1} = \frac{2 \cdot 2^{h-1} - 1}{3 \cdot 2^{h-1} - 1} \xrightarrow{h \rightarrow \infty} \frac{2}{3}$$

Analogamente a prima valutiamo la complessità del sottoproblema. Possiamo osservare che nel valutare n , il primo addendo ci restituisce il numero di nodi dal nodo radice al penultimo livello e il secondo addendo il numero di nodi all'ultimo livello. Osserviamo che la complessità, se pur lineare, rappresenta il caso peggiore. Avere un albero non completo e quindi la cui dimensione non è una potenza di due, fa aumentare la complessità.

Conclusione: Il tempo di esecuzione è $T(n) \leq T(2n/3) + \Theta(1)$, il che lo fa rientrare nel caso 2

$$\text{del teorema dell'esperto. Infatti: } n^{\frac{\log_3 1}{2}} = n^0 = 1 \in \Theta(1)$$

Ne deduciamo che la complessità di Max-Heapify è $O(\lg n)$.

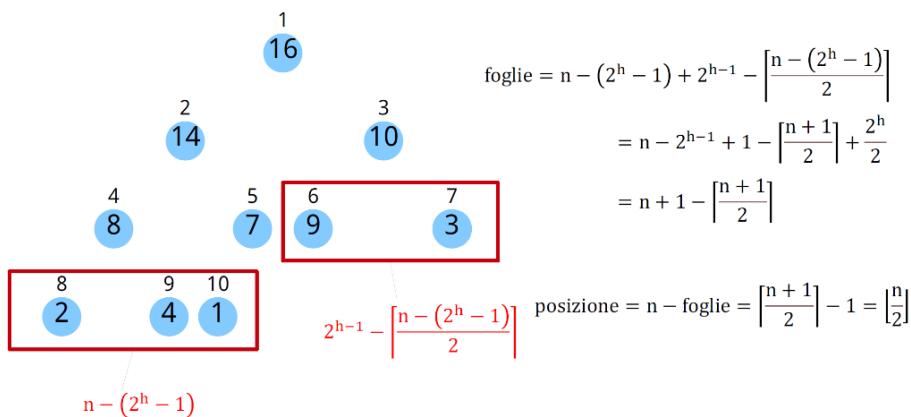
Build_max_heap

Come già detto vogliamo convertire un array in un max heap. L'algoritmo è il seguente:

Build-Max-Heap (A)

```
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ down to 1
    do Max-Heapify (A, i)
```

OSS: Partiamo da $n/2$ perché i nodi foglia sono max heap per definizione e $n/2$ è il primo nodo che ha almeno un figlio. Potremmo pensare che la complessità sia $O(n \log n)$, ma in realtà possiamo trovare un limite più stringente. Dobbiamo infatti osservare che Max_heapify lavora su alberi più piccoli all'inizio. Per dimostrare che il primo nodo che ha un figlio ha indice $n/2$, facciamo il seguente calcolo:



Possiamo osservare che il numero di foglie è dato dal numero di nodi all'ultimo livello a cui aggiungiamo il numero di nodi del livello precedente che non hanno figli. Possiamo osservare che il primo contributo è il numero di nodi totali a cui leviamo quelli presenti dal nodo radice (livello 0) al livello $h-1$. Il secondo contributo è dato dal numero di nodi al livello $h-1$ a cui leviamo i nodi che hanno dei figli al nodo successivo. Tali nodi saranno pari al primo contributo della somma (ovvero il numero di nodi nell'ultimo livello) diviso due (dato che ogni nodo padre può avere due figli); il tutto approssimato per eccesso. Infatti, nell'esempio abbiamo tre nodi all'ultimo livello e quindi questo vuol dire che avremo due nodi padre, uno con due figli e uno con un solo figlio.

Analisi di Complessità

Possiamo dimostrare semplicemente che la **complessità è $O(n)$** .

In primo luogo, osserviamo che:

- Un heap di n elementi ha altezza $h = \lfloor \lg n \rfloor$
- Un heap di n elementi ha al più $\lceil n/2^{h+1} \rceil$ elementi di altezza h .

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

Heapsort

L'idea è la seguente:

Si rende l'array un max-heap (con Build-Max-Heap) e ricordiamo che il primo elemento è l'elemento massimo. A questo punto scambiamo il primo elemento con l'ultimo: in tal modo abbiamo il massimo come ultimo elemento. Fatto ciò, si decrementa la dimensione dell'heap e si invoca Max-Heapify per mantenere la proprietà del max-heap.

```
Heap-Sort (A)
Build-Max-Heap(A)
for i ← length[A] down to 2
    exchange A[1]↔A[i]
    heap-size[A] ← heap-size[A]-1
    Max-Heapify (A,1)
```

Il tempo di esecuzione è $O(n \lg n)$ infatti:

- Build-Max-Heap impiega $O(n)$.
- $n-1$ invocazioni di Max-Heapify, in cui ciascuna impiega $O(n \lg n)$.

Operazioni Code a Priorità

Heap-Maximum

```
Heap-Maximum (A) Il Tempo di esecuzione è:  $\Theta(1)$ 
return A[1]
```

Heap-Extract-Max

```
Heap-Extract-Max (A)
if heap-size[A]<1
    error "heap empty"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
Max-Heapify (A,1)
return max
```

L'ultimo elemento viene messo al posto della radice e viene invocata Max-Heapify per ristabilire la proprietà del maxheap.

Il tempo di esecuzione è $O(\lg n)$.

Osserviamo infatti che abbiamo una sola invocazione di Max-Heapify.

Heap-Increase-Key

```
Heap-Increase-Key (A,i,key)
if key<A[i]
    error "new key is smaller"
A[i] ← key
while i>1 and A[Parent(i)]<A[i]
    exchange A[i]↔A[Parent(i)]
    i ← Parent(i)
```

Si fa risalire l'elemento di cui si cambia la chiave in direzione della radice fino a trovare la giusta collocazione. IL tempo di esecuzione è $O(\lg n)$, infatti al più abbiamo $O(\lg n)$ scambi.

Max-Heap-Insert

```
Max-Heap-Insert (A,key)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← -∞
Heap-Increase-Key (A,heap-size[A],key)
```

Il nuovo elemento viene messo dopo l'ultimo elemento con un valore $-\infty$ e viene invocata Heap-Increase-Key per impostare la chiave. Il Tempo esecuzione $O(\lg n)$; infatti invochiamo un sola volta Heap-Increase-Key.

OSS: Questo tipo di struttura ottimizza le operazioni, infatti abbiamo al più complessità $O(n \lg n)$.

QUICKSORT

Il Quick Sort è un algoritmo di ordinamento che ha un tempo di esecuzione:

- $\Theta(n^2)$ nel caso peggiore.
- $\Theta(n \lg n)$ nel caso "medio".

L'algoritmo ordina sul posto e nella pratica, risulta essere tra i più efficienti, grazie al fatto che i fattori "nascosti" in $\Theta(n \lg n)$ sono piuttosto piccoli.

Anche questo algoritmo utilizza un approccio divide et impera, come il Mergesort, ma pone maggior enfasi sul partizionamento. Vediamo l'approccio:

- **Divide:** scelto un elemento x , detto pivot, partiziona l'array $A[p..r]$ in due sottoarray tali che $A[p..q-1]$ contiene gli elementi $\leq x$ e $A[q+1..r]$ contiene gli elementi $> x$; il pivot va in $A[q]$.
- **Conquer:** ordina i due sottoarray mediante chiamate ricorsive.
- **Combine:** i sottoarray sono ordinati sul posto, quindi non occorre fare nulla.

Abbiamo una procedura **partition** che divide il vettore in due parti tali che il primo vettore contiene elementi più piccoli di un elemento detto **pivot**. Il secondo vettore invece conterrà gli elementi più grandi del pivot.

Quicksort

```
Quicksort (A,p,r)
if p < r
    q ← Partition (A,p,r)
    Quicksort (A,p,q-1)
    Quicksort (A,q+1,r)
```

q mi dirà l'indice in cui è posto il pivot. Sfrutterò questo indice per andare ad ordinare i sottovettori individuati mediante la procedura partition mediante una chiamata ricorsiva a Quicksort. Dobbiamo vedere quindi come andare a realizzare la procedura **partition**.

Partition

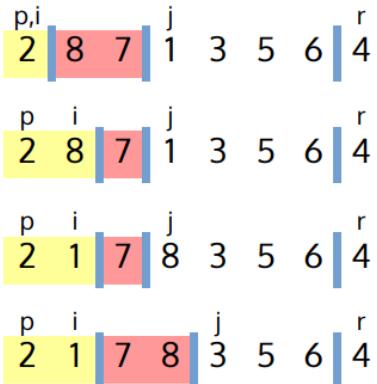
La funzione Partition utilizza come pivot l'ultimo elemento del sottoarray ($x = A[r]$). La procedura analizza, uno ad uno e a partire dal primo, tutti gli elementi compresi tra p e $r-1$.

Tale ordinamento avviene suddividendo il vettore in quattro aree:

- Gli elementi \leq del pivot nelle posizioni $[p \dots i]$.
- Gli elementi $>$ del pivot nelle posizioni $[i+1 \dots r-1]$.
- Gli elementi non ancora analizzati $[j \dots r-1]$.
- Il pivot $[r]$.

```
Partition (A,p,r)
x ← A[r]
i ← p-1
for j ← p to r-1
    if A[j] ≤ x
        i ← i+1
        exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

pongo $x = A[r]$ e quindi mi salvo il valore del pivot e pongo $i = p-1$. A questo punto j parte da p e arriva fino a $r-1$, dato che in corrispondenza di r ho il pivot. Se il valore di $A[j]$ è minore del pivot, allora incrementiamo i (se i punta all'ultimo valore dell'array sinistro, ovvero quello in cui $A[i] \leq x$; allora $i+1$ punterà al primo elemento dell'array destro) e swappiamo $A[i]$ con $A[j]$. In tal modo abbiamo esteso il vettore di sinistra.



Vediamo il perché col successivo esempio:
 $A[j] = 1$ deve essere spostato nell'array sinistro, allora incrementiamo i e swappiamo i due valori, in questo modo abbiamo ricreato le due partizioni. Alla fine, avremo tre vettori:

1. $[p, i]$
2. $[i + 2, r = j]$
3. $[i + 1]$: questo è l'elemento pivot.

Possiamo osservare che all'interno del ciclo abbiamo operazioni molto semplici. Questo caratterizza la semplicità rispetto Mergesort. Possiamo osservare che la complessità è $\Theta(n)$, con $n = r - p + 1$.

Analisi caso peggiore

Possiamo osservare che se il pivot è il max o il min del vettore, ciò che accade è che avremo sempre due partizioni, di cui una è vuota. Riduciamo il problema escludendo solo il pivot. La ricorrenza nel caso peggiore sarà:

$$T(n) = T(n - 1) + T(0) + \Theta(n)$$

Ad ogni passaggio riduciamo il problema ad un problema di dimensione $n - 1$, dato che escludiamo solo il pivot. Ciò che facciamo è partire da un problema $c \cdot n$ per arrivare ad un problema $c \cdot 2$, infatti quando il problema è composto da un unico elemento non devo ordinare dato che la condizione è $p < r$. Abbiamo quindi:

$$c[(n - 1) + (n - 2) + \dots + 2] = c \left[\frac{n(n + 1)}{2} - 1 \right]$$

Da cui nel caso peggiore la complessità sarà: $T(n) = O(n^2)$.

OSS:

- Il tempo di esecuzione di Quicksort nel caso peggiore non è migliore di quello di Insertion Sort. Il caso peggiore di Quicksort si ha quando il vettore è ordinato, caso in cui Insertion Sort richiede un tempo $O(n)$.
- Se abbiamo un solo split favorevole in cui il partizionamento genera due partizioni di dimensione pari alla metà della dimensione del problema da cui si partiva, la complessità diventa $O(n \lg n)$ come nel merge sort. Questa ipotesi identifica il caso medio, ma vediamo le seguenti analisi.

Analisi caso migliore

Il caso migliore è quando i due sottoproblemi sono perfettamente bilanciati ad ogni iterazione; ovvero le dimensioni dei sottoproblemi sono $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil - 1$. Il tempo di esecuzione sarà:

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

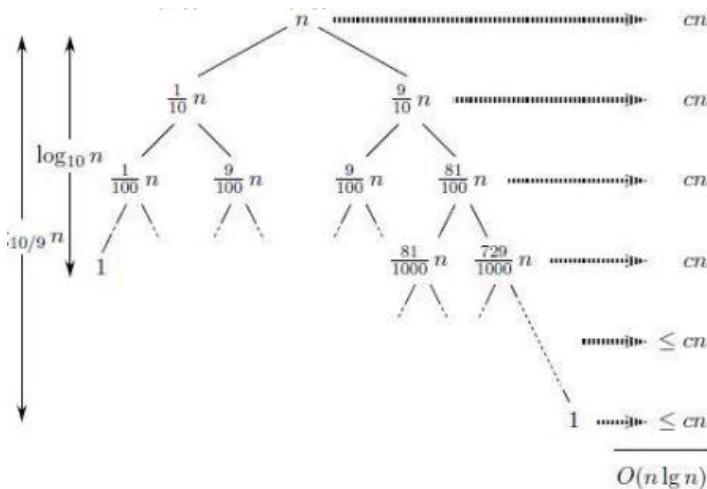
Dal caso 2 del teorema dell'esperto si ricava che $T(n) = O(n \lg n)$.

Analisi balanced partitioning

Possiamo dimostrare che il tempo medio di esecuzione di Quicksort è più vicino al caso migliore che non al caso peggiore. Supponiamo che il partizionamento produce due sottoproblemi la cui dimensione è sempre nel rapporto 9:1.

Abbiamo che: $T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + c \cdot n$

Sfruttiamo il metodo dell'albero di ricorrenza:

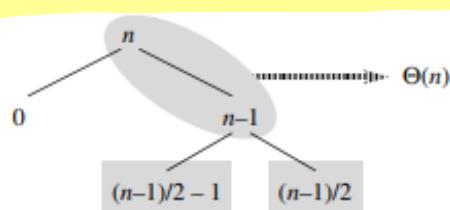


Il costo è $O(n \lg n)$, abbiamo infatti $\log_{10} n$ livelli, ciascuno di costo al più $c \cdot n$.

Qualunque partizionamento con rapporto costante (anche 99:1) produce un costo $O(n \lg n)$ perché la profondità è sempre $\Theta(\lg n)$.

Analisi caso medio

Nel caso medio, tipicamente partizioni "buone" e "cattive" si alternano. Intuitivamente, se si alternano best case e worst case, il tempo di esecuzione complessivo è quello del best case.



Il costo $\Theta(n - 1)$ della partizione cattiva viene assorbito nel costo

$\Theta(n)$ della partizione buona, e la partizione risultante è buona.

Quicksort Randomizzato

Per evitare il caso maggiore possiamo scegliere come pivot non l'ultimo elemento ma un elemento a caso. Una soluzione ancora migliore consiste nello scegliere il mediano di tre elementi presi a caso. Introduciamo quindi la funzione **Randomized-Partition** che sceglie il pivot casualmente nel sottovettore su cui lavora. La versione randomizzata di Quicksort è ritenuta la scelta migliore per ordinare array di grosse dimensioni.

```
Randomized-Partition (A, p, r)
i ← Random (p, r)
exchange A[r] ↔ A[i]
return Partition (A, p, r)
```

```
Randomized-Quicksort (A, p, r)
if p < r
    q ← Randomized-Partition (A, p, r)
    Randomized-Quicksort (A, p, q-1)
    Randomized-Quicksort (A, q+1, r)
```

Analisi caso peggiore

La complessità nel caso peggiore è $\Theta(n^2)$. Anche in questo caso il peggiore si ha quando il partizionamento crea una partizione vuota. Lo facciamo col metodo di sostituzione:

Poniamo: $T(n) \leq \max_{0 \leq q \leq n-1} [T(q) + T(n - q - 1)] + \Theta(n)$

dove il parametro q varia da 0 a $n - 1$, in quanto la procedura partition genera due sottoproblemi con dimensione totale $n - 1$. Supponiamo che la soluzione sia $T(n) \leq cn^2$ per qualche costante c . Sostituendo questa soluzione nella ricorrenza otteniamo:

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} [c \cdot q^2 + c(n - q - 1)^2] + \Theta(n) = \\ &= c \cdot \max_{0 \leq q \leq n-1} [q^2 + (n - q - 1)^2] + \Theta(n) \end{aligned}$$

L'espressione $q^2 + (n - q - 1)^2$ raggiunge il massimo nei due estremi dell'intervallo $0 \leq q \leq n - 1$ del parametro q ; infatti, la funzione ha la concavità verso l'alto (la derivata seconda rispetto a q è positiva), quindi il massimo è assunto agli estremi. Questa osservazione ci fornisce il limite:

$$\max_{0 \leq q \leq n-1} [q^2 + (n - q - 1)^2] \leq (n - 1)^2 = n^2 - 2n + 1$$

Da cui $T(n) \leq c \cdot n^2 - c(2n - 1) + \Theta(n) \leq cn^2$

Perché possiamo assegnare alla costante c un valore sufficientemente grande affinché il termine $c(2n - 1)$ prevalga sul termine $\Theta(n)$. Possiamo concludere che: $T(n) = O(n^2)$. Possiamo dimostrare analogamente che nel caso peggiore $T(n) = \Omega(n^2)$ e quindi si ha che il tempo di esecuzione nel caso peggiore è $\Theta(n^2)$.

Tempo di esecuzione atteso

Nella versione randomizzata, è di interesse il tempo di esecuzione atteso che si dimostra essere: $O(n \lg n)$. La scelta di randomizzare caratterizza l'introduzione di una variabile aleatoria che caratterizza un comportamento atteso in termini probabilistici. La complessità dell'algoritmo dipende dal numero di confronti che faccio. Definiamo:

- $x_{ij} = \begin{cases} 1 & \text{se confrontiamo } A[i] \text{ e } A[j] \\ 0 & \text{altrimenti} \end{cases}$
- i : è l' i -esimo elemento più piccolo.
- j : è il j -esimo elemento più piccolo.
- X : è il numero di confronti.

Allora se X è il numero di confronti svolti nella riga 4 di partition nell'intera esecuzione di Quicksort su un array di n elementi, allora il tempo di esecuzione di Quicksort è $O(n + X)$.

La complessità sarà quindi data dal limite globale del numero totale di confronti. Per determinare tale limite dobbiamo capire quando l'algoritmo confronta due elementi dell'array e quando non lo fa. Osserviamo in primo luogo che gli elementi sono confrontati soltanto con l'elemento pivot e, dopo che una particolare chiamata di partition finisce, il pivot utilizzato in questa chiamata non viene più confrontato con nessun altro elemento. Dobbiamo quindi valutare il numero di confronti nell'esecuzione dell'intero algoritmo. Poiché ogni coppia viene confrontata al massimo una volta, possiamo facilmente rappresentare il numero totale di confronti svolti dall'algoritmo in questo modo:

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

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

$$\begin{aligned} E[X] &= E \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n x_{ij} \right] = \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[x_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} = \text{ponendo } (k = j-i) = \\ &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{2}{k} = \sum_{i=1}^{n-1} 2 \ln n = O(n \cdot \ln n) \end{aligned}$$

dove $E[x_{ij}]$ è la probabilità che $A[i]$ o $A[j]$ siano confrontati. Alla seconda riga per passare al secondo membro dell'uguaglianza dobbiamo riflettere su quando due elementi non sono confrontati. Scelto il pivot e formate le due partizioni, saremo sicuri che nessun elemento della prima partizione sarà confrontato con un elemento della seconda partizione. Se quindi $A[i] < x < A[j]$, allora siamo sicuri che $A[i]$ e $A[j]$ non verranno confrontati in un istante successivo. Ci sarà invece il confronto se $A[i]$ o $A[j]$ è preso come pivot. Dobbiamo quindi valutare la probabilità che nell'insieme $\{A[i], \dots, A[j]\}$, venga scelto come pivot uno dei due estremi.

$\Pr\{A[i] \text{ è il pivot scelto nell'insieme}\} = \Pr\{A[j] \text{ è il pivot scelto nell'insieme}\} = \frac{1}{j-i-1}$ dove $j-i-1$ è la dimensione dell'insieme. Moltiplicando per due otterremo la probabilità desiderata. Nell'ultima riga effettuiamo la sostituzione specifica e otteniamo un serie armonica.

OSS: Possiamo dimostrare che vale l'uguaglianza $2n \cdot \ln n = 1.39n \cdot \lg n$. La situazione attesa è più vicina al tempo migliore, infatti, discosta del 40%.

Conclusione

Vantaggi:

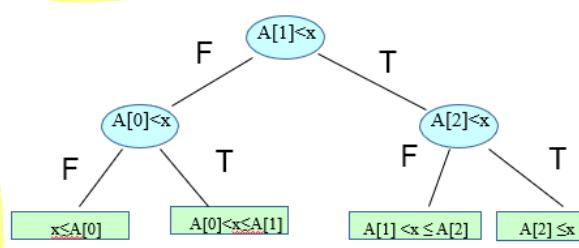
1. Quicksort è utilizzato per i piccoli fattori costanti.
2. Ordina sul posto a differenza di merge-sort (non usa spazio aggiuntivo il che caratterizza una complessità spaziale minore).
3. Sfrutta l'accesso diretto dei vettori. Generalmente gli algoritmi Mergesort e Quicksort vengono utilizzati in diversi contesti:
 - Merge-sort: per l'ordinamento su disco (linked list).
 - Quicksort: per l'ordinamento su array (sfrutta la località e sfrutta meglio la cache).
4. Uno svantaggio quicksort è che l'ordinamento non è stabile (ci sono delle varianti che risolvono questo problema).

LIMITI INFERIORI DEGLI ALGORITMI

Problemi di ricerca

Negli algoritmi di ricerca il limite inferiore è $\lg n$. Questi elementi devono essere però pre-processati (ovvero predisposti secondo una specifica metrica). Ad esempio, nel caso della ricerca binaria, gli elementi devono essere già ordinati. Vogliamo dimostrare che per questa classe di algoritmi la complessità sarà data dal limite inferiore $\Omega(\lg n)$.

Ciò che possiamo fare è costruire un **decision tree** ipotizzando la ricerca dell'elemento x in un array di tre elementi pre-processati:



In questo caso sfruttiamo il minore e il minore o uguale. Il ragionamento è il successivo: Prendiamo l'elemento centrale e vediamo se è minore dell'elemento cercato, se non è minore, allora cerchiamo a sinistra e quindi in $A[0]$. Se $A[0] < x$ allora o è $A[1]$ o è compreso tra $A[0]$ e $A[1]$.

In tale albero:

- i nodi foglia sono gli output.
- Il percorso dalla radice al nodo foglia corrisponde all'esecuzione dell'algoritmo.
- L'altezza dell'albero rappresenta il percorso più lungo e quindi è la complessità computazionale nel caso peggiore.

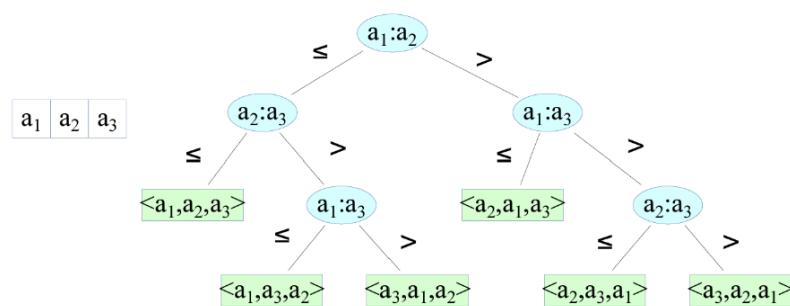
OSS: il numero di foglie può essere maggiore o uguale del numero delle possibili risposte che è maggiore o uguale di n (ovvero il numero di elementi).

L'altezza è maggiore o uguale di $\lg(n + 1) - 1$. infatti, il numero di nodi è minore o uguale di $2^{h+1} - 1$. Concludiamo che la complessità è $\lg n$.

Problemi di ordinamento

Gli algoritmi che abbiamo trattato finora si basano su confronti, parliamo anche di algoritmi **comparison model**. Il tempo computazione per questi algoritmi non può essere inferiore a $n \lg n$ (limite inferiore).

Consideriamo per semplicità un vettore di 3 elementi (a_1, a_2, a_3) e l'algoritmo Insertion sort. A partire da questo array possiamo costruire il seguente decision tree:



Consideriamo ad esempio il vettore [6,8,5].

- Confrontiamo 6 e 8 e andiamo nel ramo di sinistra.
- Confrontiamo 8 e 5 e andiamo nel ramo di destra.
- Confrontiamo 6 e 5 e quindi ci troviamo nel ramo di sinistra.

il vettore ordinato è: $(a_3=5, a_1=6, a_2=8)$

In questo caso $l = \#\text{foglie} \geq n!$, ovvero il numero di permutazioni. Maggiore o uguale perché potremmo avere più nodi foglia che rappresentano la stessa soluzione. Poiché h ci dà la complessità del problema proviamo a determinare h . In un albero di altezza h posso avere al più 2^h foglie, quindi possiamo quindi scrivere:

$$h \geq \lg n! = \lg[n \cdot (n-1) \cdot \dots \cdot 1] = \lg n + \lg(n-1) + \dots + \lg 1 = \sum_{i=1}^n \lg i$$

Possiamo osservare che:

$$\sum_{i=1}^n \lg i \geq \sum_{i=\frac{n}{2}}^n \lg i \geq \sum_{i=n/2}^n \lg\left(\frac{n}{2}\right) = \sum_{i=n/2}^n (\lg n - \lg 2) = \frac{n}{2} \lg n - n/2$$

E quindi la complessità del limite inferiore sarà proprio $\Omega(n \cdot \lg n)$.

Possiamo ricavare la complessità sfruttando l'**approssimazione di Sterling**; ovvero:

$$n! = \sqrt{2\pi \cdot n} \left(\frac{n}{e}\right)^n$$

Nel modello a confronti nessun algoritmo può fare meglio di $\Omega(n \cdot \lg n)$.

Per oltrepassare tale limite possiamo sfruttare algoritmi non a confronti che ordinano con complessità lineare.

Counting Sort

In primo luogo, dobbiamo assumere che: elementi da ordinare siano interi compresi tra 0 e k.
L'approccio è il seguente:

- Per ciascun intero i compreso tra 0 e k , si contano quanti elementi pari ad i ci sono nel vettore da ordinare.
- Per ciascun intero i compreso tra 0 e k , si determinano quanti elementi minori o uguali ad i ci sono nel vettore da ordinare. Ciò ci indica in che posizione deve stare ciascun elemento.

Counting sort utilizza due vettori di appoggio:

- B , di lunghezza n , che mantiene i valori ordinati.
- C , di lunghezza $k+1$, che indica le occorrenze di ciascun valore compreso tra 0 e k .

```
Counting-Sort (A,B,k)
for i ← 0 to k
do C[i] ← 0
for j ← 1 to length[A]
do C[A[j]] ← C[A[j]]+1
// C[i] è il numero di occorrenze di i
for i ← 1 to k
do C[i] ← C[i]+C[i-1]
// C[i] è il numero di elementi ≤ i
for j ← length[A] downto 1
do B[C[A[j]]] ← A[j]
C[A[j]] ← C[A[j]]-1
```

Il tempo di esecuzione è $\Theta(n + k)$. Nella pratica, counting sort si usa quando $k = O(n)$. In tal caso, il tempo di esecuzione è $\Theta(n)$.

Non è lineare se $k \gg n$.

Osserviamo che non ordina sul posto.

Inoltre, l'algoritmo è **stabile**: gli elementi di pari valore si presentano nel vettore risultato nello stesso ordine in cui si trovano nel vettore di partenza.

OSS: Un ulteriore problema di questo algoritmo sta nelle somme. Infatti, se i numeri eccedono nella rappresentazione dei registri del calcolatore allora i tempi delle istruzioni di somma non sono più polinomiali ma pseudopolinomiali (quindi il tempo di calcolo non è costante).

Radix Sort

Assume che gli elementi da ordinare siano rappresentati su d cifre. Poi ordino i valori a partire dalla cifra meno significativa. È indispensabile utilizzare per tale ordinamento un algoritmo stabile.

329	720	720	329	Posso utilizzare il counting sort per ordinare le singole cifre. Infatti, siamo sicuri che la singola cifra è compresa tra 0 e 9 e quindi k verosimilmente sarà $O(n)$.
457	355	329	355	
657	436	436	436	
839	457	839	457	
436	657	355	657	La correttezza si può provare per induzione.
720	329	457	720	
355	839	657	839	

```
Radix-Sort (A, d)
for i ← 1 to d
do use a stable sort to sort array A on digit i
```

La cifra di posto 1 è quella meno significativa

Il tempo di esecuzione è $\theta(d(n + k))$. Il tempo lineare se d è costante e $k = O(n)$.

OSS: Radix sort è spesso usato per ordinare informazioni aventi molteplici campi. Ad esempio, per ordinare in base alla data, si possono effettuare tre ordinamenti stabili: in base al giorno, al mese, all'anno.

In generale, si ha la flessibilità di scegliere di ordinare in base a gruppi di cifre. Se usiamo una notazione diversa, ad esempio immaginiamo di utilizzare la notazione binaria, possiamo effettuare il confronto ogni 4 bit. Quindi a partire dal valore rappresentato (supponiamo essere di b cifre), posso immaginare di raggruppare le cifre in gruppi da r cifre. Questo significa che ora avremo: $d = \lceil b/r \rceil$ e k , ovvero i valori che può assumere il gruppo di r cifre sarà: $k = 2^r$. Quindi avremo che la complessità sarà: $O\left(\frac{b}{r} \cdot (n + 2^r)\right)$. Possiamo quindi domandarci dati b e n, quale valore di r minimizza il tempo di esecuzione? Dobbiamo distinguere due casi:

- Se $b < \lfloor \lg n \rfloor$ (ovvero se i valori rappresentabili sono minori dei valori del vettore) allora:
 - $r \leq b \Rightarrow 2^r < n \Rightarrow n + 2^r = \Theta(n)$.
 - Stiamo dicendo che: se la dimensione del vettore k è minore di n allora ci conviene applicare direttamente l'algoritmo del counting sort ($r=b$) e siamo sicuri di avere una complessità lineare.
- Se $b \geq \lfloor \lg n \rfloor$, la scelta migliore è porre $r = \lg n$; infatti:
 - Per $r > \lfloor \lg n \rfloor$, 2^r cresce più rapidamente di n e quindi la complessità non è più $O(n)$.
 - Per $r < \lfloor \lg n \rfloor$, b/r cresce e $n + 2^r$ rimane $\Theta(n)$. Questa scelta ci porterebbe ad avere: $\Theta\left(\frac{b \cdot n}{\lg n}\right)$.

La complessità sarà $\Theta(n)$ se b e $\lg n$ sono paragonabili e se $b/(\lg n) < n$.

In conclusione: $r = \min\{b, \lg n\}$.

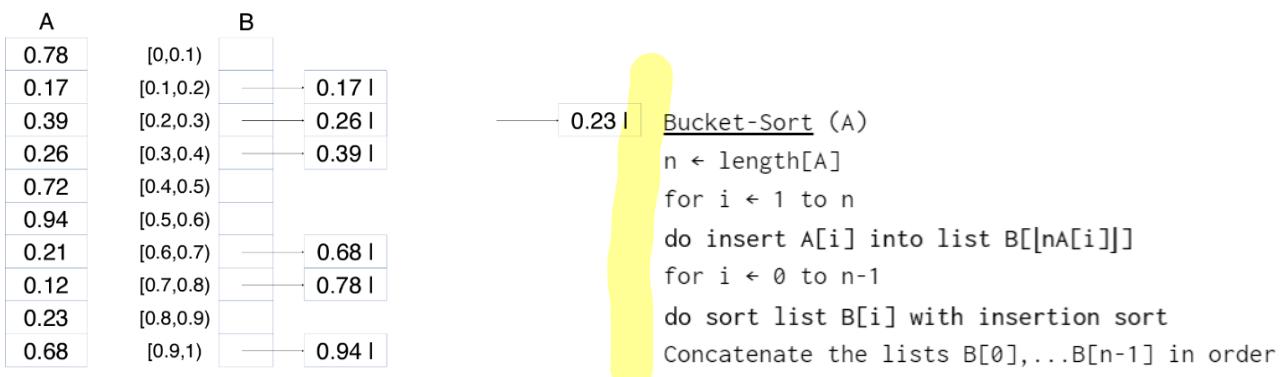
OSS: Se, come accade spesso, $b = O(\lg n)$ e $r \approx \lg n$, il tempo di esecuzione di radix sort è $\Theta(n)$, che appare migliore del $\Theta(n \lg n)$ del quick sort. D'altra parte, ogni passo di radix sort può impiegare più tempo di quick sort (può sfruttare meglio la cache). Inoltre, Radix sort, quando usa counting sort, non ordina sul posto e quindi richiede memoria aggiuntiva.

Bucket Sort

Assume che gli elementi da ordinare siano distribuiti uniformemente sull'intervallo $[0,1)$. Ovviamente è applicabile a qualsiasi problema riconducibile all'ipotesi fatta.

L'approccio consiste nel:

- Dividere l'intervallo $[0,1)$ in n sottointervalli uguali e distribuire gli n elementi in tali sottointervalli.
- Poi si devono ordinare gli elementi inseriti nello stesso sottointervallo (ad esempio con l'Insertion Sort).
- Bucket sort richiede di gestire un array ausiliario $B[0 \dots n - 1]$ di liste collegate.



Definiamo il tempo di esecuzione

Possiamo osservare che:

- L'inserimento degli elementi nelle liste (primo ciclo for) richiede $\Theta(n)$.
- La concatenazione delle liste ordinate richiede $\Theta(n)$.

Il problema è dato dall'ordinamento delle liste, infatti se usiamo l'Insertion sort la complessità di questo è n^2 . Indicando con n_i la variabile aleatoria che denota il numero di elementi nella lista $B[i]$, otteniamo che:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Poiché n_i è una variabile aleatoria, dobbiamo trattare $T(n)$ come tempo computazionale atteso, per tale motivo valutiamo $E[T(n)]$.

$$\begin{aligned} E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] = E[\Theta(n)] + \sum_{i=0}^{n-1} E[O(n_i^2)] = \\ &= \Theta(n) + O\left(\sum_{i=0}^{n-1} E[n_i^2]\right) \end{aligned}$$

A questo punto possiamo porre:

$$n_i = \sum_{j=0}^n X_{ij} \text{ dove } X_{ij} = \begin{cases} 1 & \text{se } A[j] \text{ finisce nella lista } i \\ 0 & \text{altrimenti} \end{cases}$$

A questo punto valutiamo $E[O(n_i^2)]$:

$$\begin{aligned} E[n_i^2] &= E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] = E\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij} \cdot X_{ik}\right] = E\left[\sum_{j=1}^n X_{ij}^2 + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n X_{ij} \cdot X_{ik}\right] = \\ &= \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n E[X_{ij} \cdot X_{ik}] \end{aligned}$$

Osserviamo che se ipotizziamo una distribuzione uniforme:

$$\begin{aligned} E[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n} \\ E[X_{ij} \cdot X_{ik}] &= E[X_{ij}] \cdot E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2} \end{aligned}$$

Possiamo quindi scrivere:

$$E[n_i^2] = \sum_{j=1}^n \frac{1}{n} + \sum_{j=1}^n \sum_{\substack{k=1 \\ k \neq j}}^n \frac{1}{n^2} = n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n}$$

Da cui:

$$E[T(n)] = \theta(n) + O\left(\sum_{i=0}^{n-1} E[n_i^2]\right) = \theta(n) + n \cdot O\left(2 - \frac{1}{n}\right) = \theta(n)$$

OSS: Bucket sort può ordinare in tempo lineare anche se i valori non hanno distribuzione uniforme; è sufficiente che:

$$\sum_{i=0}^{n-1} E[O(n_i^2)] \text{ cresca linearmente con } n.$$

CAPITOLO 2- STRUTTURE DATI

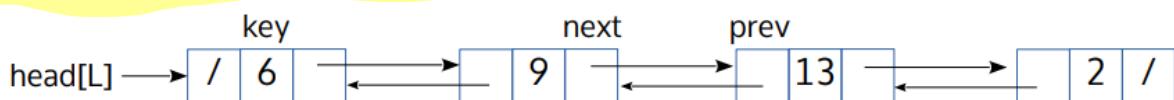
Introduzione

Partiamo da alcune definizioni:

- 1.1 **Tipo di dato astratto - ADT**: insieme di dati e operazioni possibili su di esso. Una possibile implementazione è data dalle classi della OOP. Un'alternativa sono le struct in C o con un typedef (l'utilizzo dei metodi sono sotto la responsabilità del programmatore).
- 1.2 **Strutture dati**: è un meccanismo per organizzare dati che supporta operazioni che implementano un'interfaccia (ADT). L'interfaccia è una specifica che mi dice cosa la struttura deve fare ed è diversa dall'implementazione.

Una prima classificazione può essere fatta considerando l'appartenenza di una struttura dati alla classe delle sequenze (**sequence**) o alla classe dei **set**. Una seconda classificazione può essere in base all'allocazione in memoria della struttura dati. L'allocazione può essere:

1. Contigue: (array).
2. Concatenata: uso i puntatori (lista concatenata).



3. Indicizzata: sfrutta gli indici.

OSS: Gli array sono strutture dati che possono essere visti come caso particolare di strutture più complesse (le matrici). Una possibile declinazione degli array può essere data dagli array associativi, array in cui gli elementi sono coppie (chiave, valore).

Sequence

Una sequenza è una collezione di elementi il cui ordinamento è dato dalla posizione (ordine di inserimento) e la chiave degli oggetti non è univoca. Diciamo che la sequenza presenta un **ordinamento estrinseco**. Su una sequence possiamo definire le seguenti operazioni:

- *build*
- *length*

Operazioni statiche:

- *iter_seq*: restituisce gli elementi nell'ordine dato dalla sequenza.
- *get(i)*
- *set(i)*

Operazioni dinamiche:

- *insert_at(i)*
- *deleter_at(i)*
- *insert_first/insert_last e delete_first/delete_last*

Per quanto riguarda l'implementazione:

	<i>build</i>	<i>length</i>	<i>get set</i>	<i>insert_fisrt delete_first</i>	<i>insert_last delete_last</i>	<i>insert_at(i) delete_last(i)</i>
Array	n	1	1	n	n	n
Linked List	n	1	n	1	$n \text{ o } 1$	n

OSS: gli array ottimizzano le operazioni di *get e set*, grazie all'accesso diretto; mentre le liste ottimizzano le operazioni di inserimento ed estrazione. In particolare, per l'inserimento in coda, l'operazione è ottimizzata se abbiamo un riferimento alla coda stessa.

Set

Un set è un insieme ordinato (per chiave) in cui non abbiamo duplicati (ho una chiave univoca). Parliamo di **ordinamento intrinseco**. In queste strutture gli elementi sono oggetti che possono essere caratterizzati da **chiave (key – è obbligatoria)** e **dati satellite**. In questo caso le operazioni definite sono:

- *build*
- *length*

Operazioni statiche:

- *find(key)*

Operazioni dinamiche:

- *insert(key)*
- *deleter_at(key)*

Operazioni di ordinamento:

- *iter_order*: ritorna gli elementi in ordine di chiave.
- *find_min/find_max*
- *find_prev(key)/find_next(key)*

Strutture dinamiche

Partiamo da alcune definizioni:

1.1 **Insieme dinamico**: è un insieme di elementi che "cambia" nel corso del tempo.

1.2 **Dizionario**: è un insieme dinamico che supporta le operazioni di:

- Inserimento di un elemento.
- Eliminazione di un elemento.
- Verifica dell'appartenenza di un elemento all'insieme.

Come già detto, gli elementi di tali strutture possono essere oggetti caratterizzati da:

- Una **chiave – key**, su cui può essere definita una relazione d'ordine.
- **Dati satellite** (attributi opzionali).

Vediamo nel dettaglio le principali strutture dinamiche.

Stack e Code

Stack e code sono strutture dati in cui l'operazione **DELETE** rimuove un elemento in maniera predefinita:

- In uno **stack**, l'elemento rimosso è quello più recentemente inserito - strategia **LIFO** (Last In First Out).
- In una **coda**, l'elemento rimosso è quello inserito da più tempo - strategia **FIFO** (First In First Out).

Possiamo implementare stack e code sfruttando array o liste.

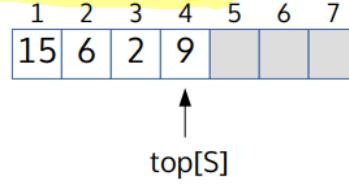
Stack: Implementazione con array

Possiamo implementare uno stack di al più n elementi mediante un array:

- Di lunghezza n.
- Con un attributo **top** che indica la posizione dell'elemento inserito più di recente (0 se vuoto).

Stack-Empty (S)

```
if top[S]=0  
    then return TRUE  
else return FALSE
```



Queste operazioni hanno complessità $O(1)$.

Nell'operazione di push trascuriamo la verifica dell'overflow.

Push (S, x)

```
top[S] ← top[S] + 1  
S[top[S]] ← x
```

Pop (S)

```
if Stack-Empty(S)  
    then error "underflow"  
else top[S] ← top[S]-1  
    return S[top[S]+1]
```

Sono definite operazioni di:

- **push**: per l'inserimento in coda.
- **pop**: per l'estrazione in coda.

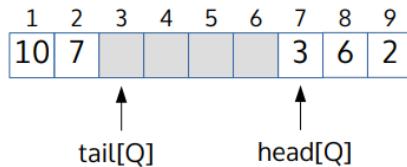
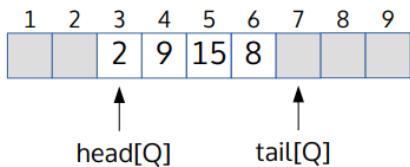
OSS: La struttura può essere realizzata anche in modo da estrarre e inserire in testa (sempre FIFO).

Code: Implementazione con array

Possiamo implementare una coda di $n - 1$ elementi mediante un array di lunghezza n con:

- un attributo $head$ che indica la posizione della testa.
- un attributo $tail$ che indica la posizione in cui inserire il prossimo elemento accodato.

OSS: La coda è vuota se $head = tail$, piena se $head = tail + 1$



Queste operazioni hanno complessità $O(1)$. In queste operazioni si trascura la verifica dell'overflow e dell'underflow.

```
Enqueue (Q, x)
Q[tail[Q]] ← x
if tail[Q] = length[Q]
    then tail[Q] ← 1
    else tail[Q] ← tail[Q]+1
```

```
Dequeue (Q)
x ← Q[head[Q]]
if head[Q] = length[Q]
    then head[Q] ← 1
    else head[Q] ← head[Q]+1
return x
```

Sono definite operazioni di:

- **Enqueue**: per l'inserimento in coda.
- **Dequeue**: per l'estrazione in testa.

Stack e Code: Implementazione con lista concatenata

List-Search (L, k)

```
x ← head[L]
while x ≠ NIL and key[x] ≠ k
    do x ← next[x]
return x
```

Prima di passare all'implementazione dello stack, osserviamo che in un lista concatenata non può essere effettuare una ricerca binaria, ma dobbiamo usare una ricerca lineare.

List-Insert (L, x)

```
next[x] ← head[L]
if head[L] ≠ NIL
    then prev[head[L]] ← x
head[L] ← x
prev[x] ← NIL
```

List-Delete (L, x)

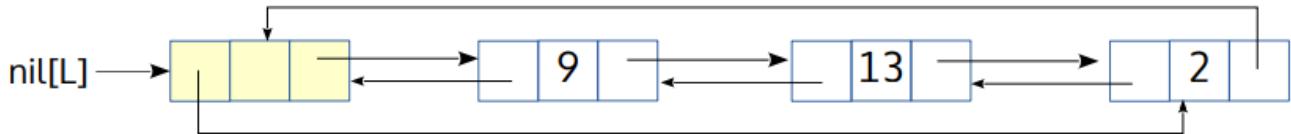
```
if prev[x] ≠ NIL
    then next[prev[x]] ← next[x]
else head[L] ← next[x]
if next[x] ≠ NIL
    then prev[next[x]] ← prev[x]
```

In questo caso l'implementazione è realitiva ad una lista generica. Stack e queue non sono altro che declinazioni di questa lista; infatti:

- Push e Enqueue si ottengono banalmente mediante list-insert.
- Pop: si ottiene mantenendo un puntatore alla coda e passando come x , tale puntatore.
- Dequeue: si ottiene passando come x , il puntatore alla testa.

OSS: Facciamo riferimento ad una lista doppiamente linkata e le operazioni di insert e delete hanno complessità O(1).

Un'alternativa alle liste concatenate sono le **liste concatenate con sentinella**. Mentre nelle liste linkate $prev[head[L]]$ e $next[queue[L]]$ hanno NIL come valore, nelle liste con sentinella $prev[head[L]]$ punta alla coda, $queue[L]$ e $next[queue[L]]$ punta alla testa, $head[L]$. In tal modo possiamo semplificare le operazioni precedenti.



Per fare ciò, NIL può essere sostituito sa $nil[L]$ che punta alla testa $head[L]$. Quando la lista è vuota sia next che prev di $nil[L]$ puntano a $nil[L]$ stesso.

List-Delete' (L, x)

```
next[prev[x]] ← next[x]
prev[next[x]] ← prev[x]
```

List-Insert' (L, x)

```
next[x] ← next[nil[L]]
prev[next[nil[[L]]]] ← x
next[nil[[L]]] ← x
prev[x] ← nil[L]
```

List-Search' (L, k)

```
x ← next[nil[L]]
while x ≠ nil[L] and key[x] ≠ k
    do x ← next[x]
return x
```

EX:

Problema: voglio realizzare un sistema di prenotazione delle pista (unica) in un aeroporto.

Proprietà del problema:

- Singola pista.
- Prenotazioni per futuri atterraggi.
- All'atterraggio si rimuove la prenotazione.
- Richiesta: è data da t ovvero il tempo entro il quale dovrà atterrare.
- Aggiungi t all'insieme se non ci sono altri atterraggi schedulati entro k minuti.

Esempio: R=(37,41,46,49,56) e k =3min

- Request(44) -> errore
- Request(53) -> si
- Request(20) -> tempo già passato, errore.

Vediamo le possibili soluzioni:

1. Sorted list (1 per l'inserimento ma n per la ricerca)
2. Sorted array ($\lg n$ per la ricerca ma n per l'inserimento)
3. Min_heap (inserimento $\lg n$ ma per il vincolo diventa n)

Per fare di meglio di $O(n)$, dobbiamo introdurre gli alberi di ricerca.

ALBERI DI RICERCA BINARI

Gli alberi di ricerca sono strutture dati che supportano molte delle operazioni definite su insiemi dinamici, come: Search, Minimum, Maximum, Predecessor, Successor, Insert, Delete. Inoltre, possono essere usati sia come dizionario che coda a priorità.

Le operazioni base richiedono un tempo proporzionale all'altezza che è:

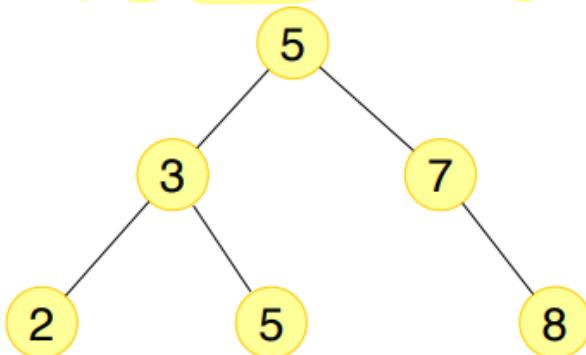
- $\lg n$ se l'albero è completo, ovvero tutti i nodi hanno due figli e le foglie sono tutte alla stessa altezza.
- n nel caso degenere di lista concatenata: è un caso particolare dato quando l'albero viene costruito a partire da una sequenza di valori ordinati secondo la priorità con cui l'albero viene costruito.

OSS: Un modo per non avere complessità lineare è costruire l'albero in maniera aleatoria.

In generale, un albero è un albero binario di ricerca se gode della seguente proprietà.

Proprietà - Sia x un nodo di tale albero e y un figlio di x :

- Se y è figlio destro allora $key[y] \geq key[x]$.
- Se y è figlio sinistro allora $key[y] \leq key[x]$.



A differenza dell'heap, questo tipo di alberi, sono alberi veri e proprio e non vettori visualizzati come alberi. I nodi di questo albero sono caratterizzati da:

1. Puntatore al padre
Nel caso del nodo radice (di cui dobbiamo tenere traccia), il padre di tale nodo sarà NIL.
2. Puntatore a figlio destro (NIL nel caso di nodo foglia).
3. Puntatore a figlio sinistro (NIL nel caso di nodo foglia).
4. (Key, Dati satellite).

Vediamo le varie operazioni nell'albero.

Ordinamento

Inorder-Tree-Walk (x)

```
if x≠NIL  
    then Inorder-Tree-Walk (left[x])  
    print key[x]  
    Inorder-Tree-Walk (right[x])
```

La proprietà di un albero di ricerca consente di stampare tutte le chiavi in ordine crescente mediante una visita in ordine dell'albero. In questo caso non ha senso ordinare l'albero.

Tale algoritmo impiega un tempo $\Theta(n)$.

Ricerca

Tree-Search (x,k)

```
if x=NIL or key[x]=k  
    then return x  
if k < key[x]  
    then return Tree-Search (left[x],k)  
else return Tree-Search (right[x],k)
```

Possiamo fare una sorta di ricerca binaria sfruttando la proprietà con cui costruiamo l'albero.

Possiamo effettuare sia una procedura ricorsiva (come in figura) che iterativa. Osserviamo che la procedura termina quando $x = NIL$ ovvero l'albero è terminato o quando troviamo l'elemento. La complessità è $O(h)$, dove h è l'altezza dell'albero.

Max e Min

Tree-Minimum (x)

```
while left[x]≠NIL  
    do x ← left[x]  
return x
```

Tree-Maximum (x)

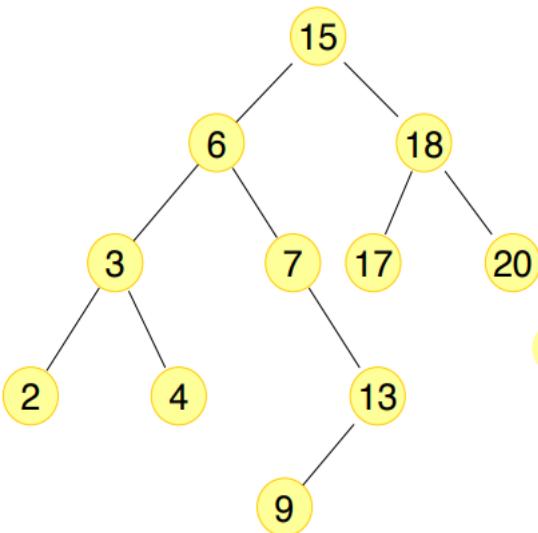
```
while right[x]≠NIL  
    do x ← right[x]  
return x
```

Infatti:

- L'elemento minimo può essere trovato percorrendo, a partire dalla radice, sempre il sottoalbero di sinistra.
- L'elemento massimo può essere trovato percorrendo, a partire dalla radice, sempre il sottoalbero di destra.

Anche in questo caso il tempo di esecuzione è $O(h)$.

Successore e Predecessore di un nodo



Per comprendere al meglio come individuare il successore, vediamo il seguente esempio:

Supponiamo di voler individuare il successore di 18 o 7, in questo caso il successore è il minimo del sottoalbero di destra. Analogamente il predecessore sarà il massimo del sottoalbero di sinistra. Questo è vero se il nodo in questione ha un figlio destro (o sinistro nel caso di predecessore). Prendiamo ora ad esempio 13, questo non ha figlio destro e il suo successore è 15, ovvero il primo antenato il cui figlio di sinistra è antenato del nodo in questione (15). Analogamente il predecessore di un nodo che non ha figlio sinistro è il primo antenato il cui figlio di destra è antenato del nodo in questione. Ad esempio, il nodo 7 ha come predecessore 6. Per semplicità riportiamo solo l'algoritmo per il calcolo del successore, per il predecessore abbiamo un algoritmo speculare. Il tempo di questi due algoritmi è $O(h)$, infatti nel caso peggiore vogliamo il successore/predecessore di un nodo foglia (il più profondo) e dobbiamo risalire fino al nodo radice.

Tree-Successor (x)

```
if right[x]≠NIL  
    then return Tree-Minimum(right[x])  
y ← p[x]  
while y≠NIL and x=right[y]  
    do x ← y  
    y ← p[y]  
return y
```

Inserimento di un elemento

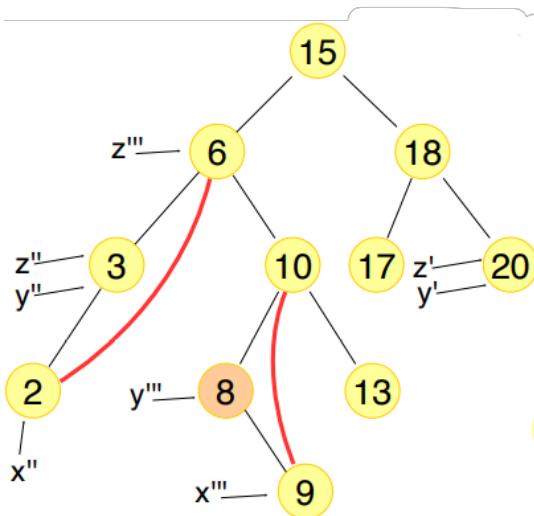
Tree-Insert (T,z)

```
y ← NIL  
x ← root[T]  
while x≠NIL  
    do y ← x  
    if key[z] < key[x]  
        then x ← left[x]  
        else x ← right[x]  
p[z] ← y  
if y = NIL  
    then root[T] ← z // albero vuoto  
else if key[z] < key[y]  
    then left[y] ← z  
    else right[y] ← z
```

L'inserimento è un'operazione semplice, dobbiamo solo a partire dal nodo radice scegliere la posizione in cui andrà inserito il nodo. Confronteremo (a partire dalla radice), la chiave del nodo da inserire con la chiave del nodo dell'albero e se è maggiore scenderemo nel sottoalbero di destra altrimenti nel sottoalbero di sinistra. Ci fermeremo quando troveremo un nodo che non ha figlio (dx o sx) in cui possiamo inserire il nodo target.

Il tempo di esecuzione è $O(h)$.

Eliminazione di un elemento



Questa è l'operazione più complessa e dobbiamo distinguere tre casi:

1. Il nodo non ha figli.
2. Il nodo ha un solo figlio.
3. Il nodo ha due figli.

Nel primo caso è sufficiente eliminare il nodo (come nel caso di un nodo foglia). Nel secondo caso possiamo eliminare il nodo, ma dobbiamo rimpiazzarlo col figlio (come nel caso del nodo 3); ovvero dobbiamo fare in modo che il padre del nodo eliminato, diventi il padre del figlio del nodo eliminato. Il terzo caso è quello più complesso, infatti dobbiamo individuare la strategia più veloce per rimpiazzare il nodo eliminato.

Supponiamo di voler eliminare il nodo 6, potremmo pensare di mettere il figlio sinistro o destro al suo posto. Il problema è che questo potrebbe avere a sua volta figlio destro e sinistro, e ciò comporterebbe un grosso problema; infatti, dovremmo riordinare l'albero a partire da quel punto. Possiamo allora pensare di rimpiazzarlo col suo successore. Infatti, per ipotesi il nodo che stiamo eliminando ha figlio destro e quindi il successore sarà il minimo del sottoalbero di destra che per definizione non avrà figlio sinistro. A questo punto possiamo rimpiazzare il nodo da eliminare col successore e l'eventuale figlio destro del nodo che stiamo spostando verrà trattato come nel caso 2.

Tree-Delete (T,z)

```

if left[z]=NIL or right[z]=NIL
    then y ← z
    else y ← Tree-Successor (z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL // y non e' foglia
    then p[x] ← p[y]
if p[y] = NIL // y è radice
    then root[T] ← x
    else if y = left[p[y]]
        then left[p[y]] ← x
        else right[p[y]] ← x
if y ≠ z // terzo caso
    then key[z] ← key[y]
    Copy y's satellite data into z
return y
  
```

Nell'algoritmo con y indichiamo il nodo che rimuoveremo.

Possiamo osservare che la complessità è $O(h)$, infatti è la complessità necessaria per individuare il successore (caso peggiore).

Nel 3° caso es. eliminiamo 6 prendiamo il successore del figlio dx di 6 che è 8 (f. dx di 6 è 10) poi sostituiamo 6 con 8, il figlio dx di 8 (9) andrà trovato con caso 2 quindi 10 diventa padre di 9.

EX: Prenotazione pista aerei

Tornando all'esercizio precedente, possiamo quindi osservare che possiamo utilizzare questa struttura, sfruttando la proprietà di inserimento. Una volta inserito l'elemento possiamo individuare predecessore e successore e valutare il vincolo sul tempo. Se il vincolo fallisce, possiamo pensare di eliminare tale elemento. La complessità sarà $O(h)$ e quindi in caso di albero completo $O(\lg n)$. Possiamo anche osservare le seguenti cose:

- Il nodo sarà inserito come nodo foglia e quindi l'eliminazione entrerà nel caso banale, potremmo quindi anche evitare di utilizzare il metodo per l'eliminazione.
- Il problema di questo algoritmo è che le varie richieste potrebbero presentarsi come sequenze ordinate. In questo caso ci troviamo nel caso in cui l'albero è visto come una lista e quindi $h = n$. Questo comporta una complessità lineare come nelle altre soluzioni valutate.

Per non incorrere nel caso precedente possiamo pensare di ordinare le richieste al tempo in t , in ordine casuale e poi inserirle nell'albero.

Aggiungiamo ora una specifica al problema: vogliamo creare un metodo che ci dica quanti aerei sono stati schedulati prima di t . Per fare ciò, posso ragionare in due modi:

1. Uso la struttura dell'albero di ricerca e la navigo contando tutti gli aerei con tempo di atterraggio $\leq t$. Questo comporterebbe una complessità lineare $O(n)$
2. Posso modificare la struttura dell'albero (diciamo che **aumentiamo la struttura dati**), inserendo un'informazione aggiuntiva ai nodi. Posso ad esempio aggiungere una variabile ai nodi che mi indica la dimensione del sottoalbero (numero di nodi) a partire da quel nodo. Ovviamente questa modifica comporterà la modifica delle operazioni di inserimento ed eliminazione.

La strategia di aumentare la struttura dati e quindi aggiungere informazioni ai nodi può essere utilizzata per realizzare particolari alberi detti **self-balancing BST (binary search tree)**. Ad esempio, la soluzione **AVL Tree**, prevede l'aggiunta di un fattore di bilancio (**balance factor -BF**) che consente di distribuire i nodi in modo da bilanciare il carico. In particolare, si richiede che:

$$BF = h(T_r[x]) - h((T_l[x])) : BF = [-1,0,1]$$

Ovvero che il valore assoluto della differenza dell'altezza del sottoalbero di destra e dell'altezza del sottoalbero di sinistra sia minore o uguale di 1. E quindi:

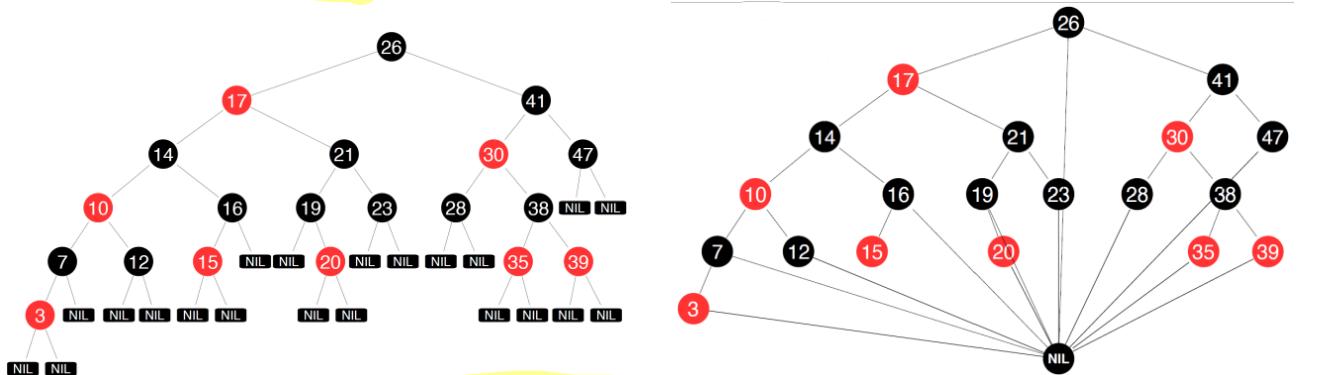
$$|h(T_r[x]) - h((T_l[x]))| \leq 1$$

RED-BLACK TREE

È un esempio di struttura dati aumentata che consente all'albero di auto bilanciarsi. Gli alberi rosso-neri sono alberi di ricerca in cui ogni nodo ha un attributo addizionale color, che può valere rosso o nero. I puntatori a NIL sono sostituiti da nodi foglia esterni (serve per definire delle proprietà e preservarle). In un albero rosso-nero, devono essere soddisfatte le seguenti proprietà:

1. Ogni nodo o è rosso o è nero.
2. La radice è nera.
3. Ogni nodo foglia esterno (NIL) è nero.
4. Se un nodo è rosso, entrambi i figli sono neri.
5. Per ogni nodo, tutti i percorsi dal nodo alle foglie contengono lo stesso numero di nodi neri (è la proprietà che garantisce il bilanciamento).

Gli alberi rosso-neri garantiscono che nessun percorso dalla radice ad una foglia è lungo più del doppio di un altro percorso.



Come rappresentazione, posso usare un'unica sentinella (nera) a cui si collegano tutti i nodi foglia, come nella rappresentazione sottostante.

Un'alternativa ai fini della visualizzazione dell'albero è escludere da questa il nodo NIL. Dobbiamo ricordare che proprio come nelle linked list, NIL è un vero e proprio nodo (nodo ausiliario), nero per definizione.

Red-Black Tree: Altezza

Definiamo **black-height ($bh(x)$)** di un nodo x il numero di nodi neri lungo un qualunque percorso da x ad una foglia (dal conteggio è escluso x ed è incluso il nodo foglia esterno, ovvero il nodo ausiliario). In particolare, la black-height di un albero rosso-nero è la black-height della radice.

TH: Un albero rosso-nero con n **nodi interni** (ovvero escludiamo i nodi NIL) ha **altezza** pari al più a $2 \cdot \lg(n + 1)$.

Dimostrazione

Dimostriamo prima che un sottoalbero con radice in un nodo x contiene almeno $2^{bh(x)} - 1$ nodi interni.

Ragioniamo per induzione sull'altezza di x :

1. Nel caso base, altezza = 0. Ovvero stiamo valutando l'altezza del sottoalbero che ha come radice un nodo foglia esterno (il nodo sentinella) e la tesi è vera perché $2^{bh(x)} - 1 = 2^0 - 1 = 0$.

2. Passo induttivo:

- Consideriamo un nodo x con $\text{altezza} > 0$ che è un nodo interno con due figli.
- Ciascun figlio ha una black-height:
 - $bh(x)$ se è rosso.
 - $bh(x) - 1$ se è nero.

Infatti, se il nodo è nero, nel calcolo del bh deve essere escluso (da cui il -1).

- Dato che l'altezza dei figli è minore di quella di x , si può applicare l'ipotesi induttiva, ovvero ogni figlio ha almeno $2^{bh(x)-1} - 1$ nodi interni.

Allora x avrà almeno $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = (2^{bh(x)} - 1) + 1$ nodi interni. Facciamo quindi l'ipotesi di caso peggiore (il nodo x ha entrambi figli neri). Inoltre, osserviamo che poiché avrà due figli per ipotesi allora dobbiamo considerare due volte il contributo $2^{bh(x)-1} - 1$ e dobbiamo aggiungere un nodo (ovvero il nodo x stesso).

Ora passiamo al dimostrare che un albero rosso-nero con n nodi interni ha altezza pari al più a $2 \cdot \lg(n+1)$.

Per la proprietà 4, almeno la metà dei nodi su ogni percorso dalla radice ad una foglia, escludendo la radice, devono essere neri. Quindi la black-height della radice deve essere almeno $h/2$, dove h è l'altezza dell'albero da cui: $h \leq 2 \cdot bh$.

Il numero di nodi interni n è almeno $2^{h/2} - 1$ e quindi:

$$n \geq 2^{bh} - 1 \geq 2^{\frac{h}{2}} - 1 \Leftrightarrow \lg(n+1) \geq \frac{h}{2} \Leftrightarrow h \leq 2 \cdot \lg(n+1)$$

Operazioni

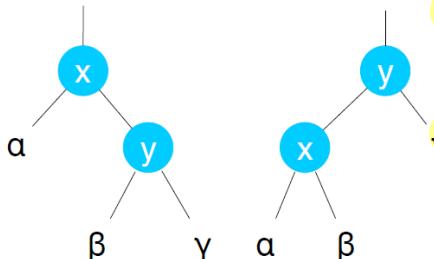
Le query Minimum, Maximum, Successor e Predecessor richiedono un tempo $O(\lg n)$ su un albero rosso-nero (proprio come in un albero di ricerca binario). Le operazioni di Insert e Delete vanno però riviste per preservare le proprietà dell'albero.

Rotazioni

Per preservare le proprietà di un albero rosso-nero, vengono effettuate delle operazioni di rotazione. Queste operazioni in generale preservano le proprietà di un albero di ricerca.

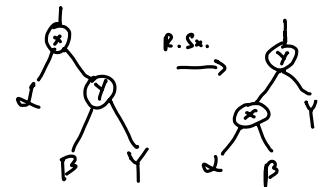
Una rotazione può essere a sinistra o a destra:

Rotazione a sinistra (sul nodo x)



- β diventa figlio destro di x .
- Il padre di x diventa il padre di y .
- x diventa il figlio di sinistra di y .

OSS: presuppone che il figlio di destra di x non sia NIL.



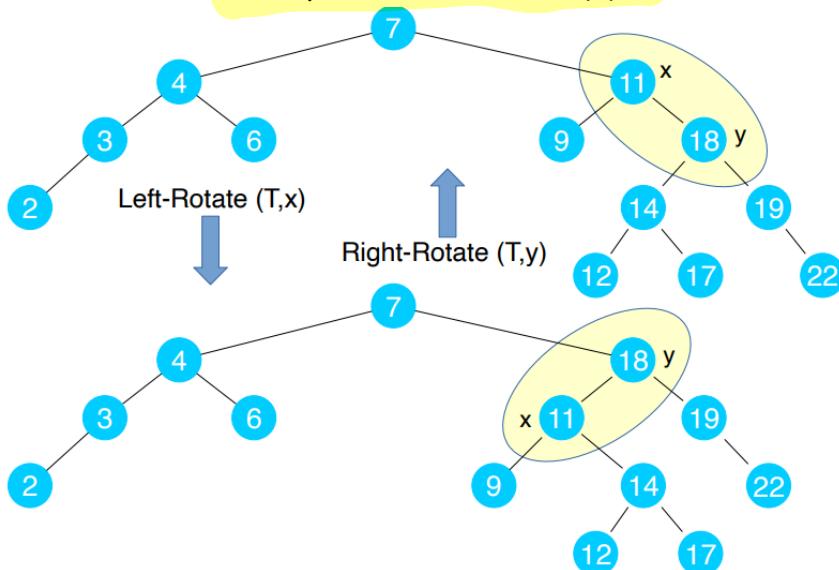
Left-Rotate (T,x)

```

y ← right[x]
// operazione 1.
right[x] ← left[y]
if left[y] ≠ nil[T]
    then p[left[y]] ← x
// operazione 2.
p[y] ← p[x]
if p[x] = nil[T]
    then root[T] ← y
    else if x = left[p[x]]
        then left[p[x]] ← y
        else right[p[x]] ← y
// operazione 3.
left[y] ← x
p[x] ← y

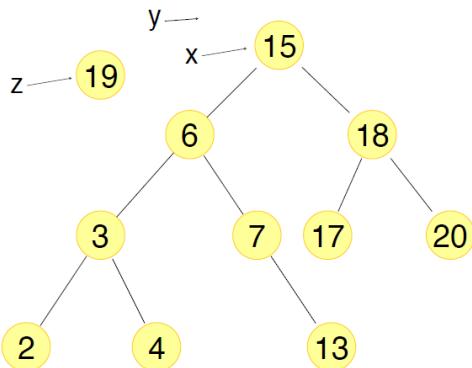
```

L'algoritmo prevede che il figlio di destra di x diventa il figlio di sinistra di y (infatti siamo sicuri che $key[x] \leq key[y]$ dato che per ipotesi $key[x] \leq key[y]$). Questa operazione va fatta a prescindere indipendentemente dal fatto che il figlio sinistro di y possa essere $nil[T]$. Se però il figlio sinistro non è $nil[T]$ va impostato il padre di questo a x (è ciò che facciamo nell'IF seguente). Impostato correttamente il nodo x , ora dobbiamo swappare x e y . Per fare ciò il padre di x deve diventare il padre di y e nel caso in cui x era il nodo radice (ovvero se $p[x]=nil[T]$) allora y diventa la nuova radice. Altrimenti x avrà un nodo padre e quindi dobbiamo distinguere due casi. Se x è il figlio sinistro allora y diventa il nuovo figlio sinistro; altrimenti (x è il figlio destro) y diventa il figlio destro. Alla fine, x diventa il figlio sinistro di y e di conseguenza il padre di x deve essere posto ad y .
Il tempo di esecuzione è $O(1)$.



OSS: Possiamo osservare che la rotazione di destra è duale. Infatti, in questo caso sarà il figlio sinistro a salire (per questo diciamo che la rotazione avviene a destra).

Inserimento di un nodo



Il tempo di esecuzione è $O(h)$

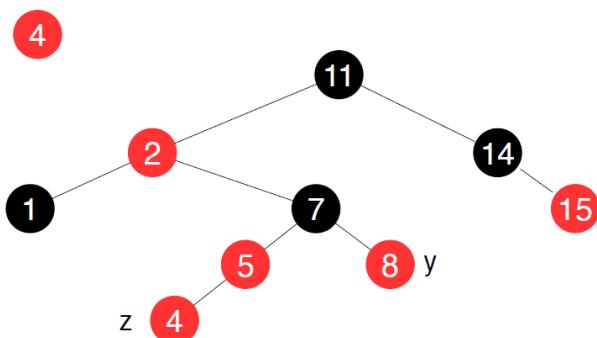
```

RB-Insert (T,z)
y ← nil[T]
x ← root[T]
while x≠nil[T]
    do y ← x
    if key[z] < key[x]
        then x ← left[x]
        else x ← right[x]
    p[z] ← y
    if y = nil[T]
        then root[T] ← z // albero vuoto
        else if key[z] < key[y]
            then left[y] ← z
            else right[y] ← z
    left[z] ← nil[T]
    right[z] ← nil[T]
    color[z] ← RED
RB-Insert-Fixup (T,z)

```

L'inserimento di un nodo avviene come in un normale albero di ricerca, utilizzando una versione (leggermente) modificata di **Tree-Insert** che chiameremo **RB-Insert**. Possiamo quindi vedere che si fa la classica ricerca binaria per trovare il posto giusto in cui inserire il nodo (sarà a prescindere un nodo foglia). Una prima differenza è che y non è inizializzato a *NIL* ma è assegnato a $nil[T]$ (che ricordiamo essere un nodo con chiave vuota). Inserito il nodo, poniamo figlio destro e sinistro pari a $nil[T]$ e poniamo il colore del nodo a rosso (nodo red). Fatto questo si invoca una funzione ausiliaria (**RB-Insert-Fixup**) per ripristinare le proprietà degli alberi rosso-neri. Il tempo di esecuzione vedremo che è $O(\lg n)$.

Se inseriamo il nodo rosso possiamo violare la proprietà due e quattro.



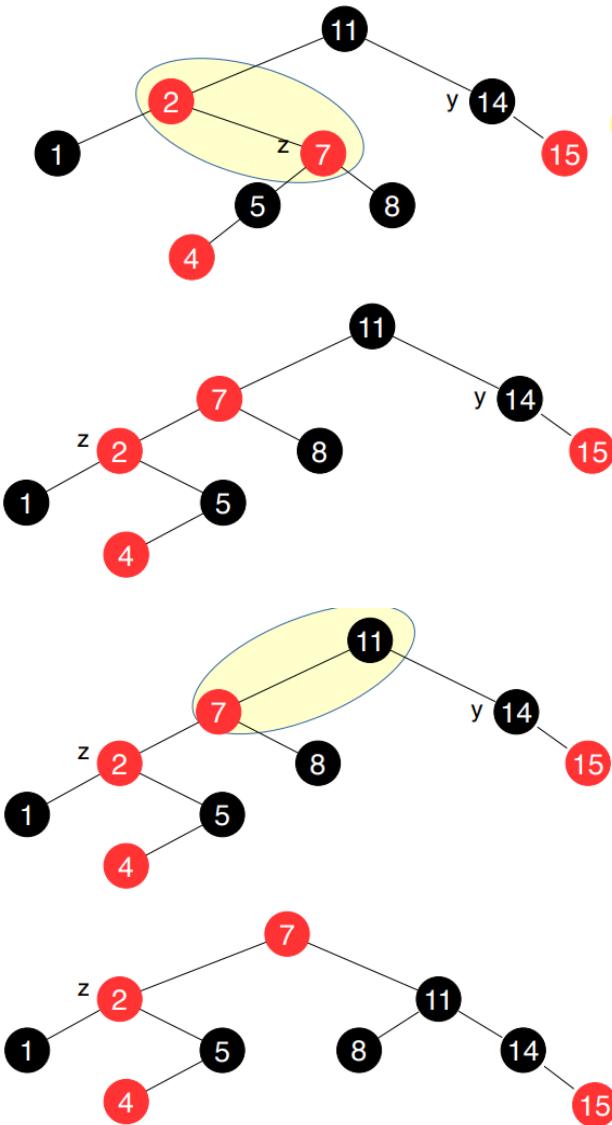
Caso base. Se l'albero è vuoto, il nodo inserito sarà rosso ma la radice per ipotesi è nera. In questo caso basta modificare il colore del nodo da rosso a nero. Vediamo ora quando è violata la proprietà quattro con un esempio. Se volessimo inserire il nodo con key 4, questo dovrebbe essere inserito come figlio sinistro del nodo con chiave 5. In questo caso è violata la proprietà quattro, ovvero i figli di cinque devono essere neri. Ovvero dobbiamo controllare

che il colore del nodo padre non deve essere rosso. Se è nero, basta inserire il nodo banalmente se invece è rosso dobbiamo prendere alcuni accorgimenti. Possiamo in primo luogo paragonare il colore del nodo inserito, col colore dello **zio** (ovvero l'altro figlio del padre di mio padre e quindi l'altro figlio di mio nonno). Da questo confronto possono nascere due casi:

Caso 1. Se il colore dello zio è rosso allora dovrò cambiare i colori di tutti i nodi considerati (nonno, padre, zio), escluso il nodo stesso (z). Questo lo possiamo fare infatti se y è rosso allora siamo sicuri che eventuali figli sono neri e di conseguenza se y diventa nero, il sottoalbero di y risulta comunque un RB-Tree. Cambiando i colori, quindi, sono sicuro che il sottoalbero che ha come radice il nonno del nodo inserito è un RB-Tree.

OSS: Se si verifica il primo caso allora siamo sicuri che il nonno del nodo inserito è un nodo nero; infatti, se fosse rosso i figli (e quindi il padre del nodo inserito) sarebbero necessariamente neri.

Dall'osservazione precedente si evince che il cambio di colori, porterebbe il nonno da nero a rosso. A questo punto potrebbe essere ri-violata la proprietà quattro. Ovvero potrebbe accadere che il padre del nonno del nodo inserito, potrebbe essere rosso. Da questa osservazione comprendiamo che il problema è ricorsivo, ovvero bisogna controllare caso 1 e caso 2, risalendo nella gerarchia, finché il padre del nodo trattato non sia rosso come il figlio. Rifacendoci all'esempio possiamo ora trattare il caso 2.



Caso 2. Se il colore dello zio è nero, la situazione è più complessa; infatti, non possiamo banalmente cambiare colore ai nodi, dato che lo zio diventerebbe rosso e il suo sottoalbero potrebbe non essere un RB-Tree, ovvero lo zio potrebbe avere un figlio rosso (violata proprietà quattro). Se si verifica tale condizione devo considerare altri due casi:

- 2.1 se sono il figlio destro di mio padre allora faccio una rotazione a sinistra.
- 2.2 se sono il figlio sinistro di mio padre allora faccio una rotazione a destra.

Nell'esempio trattato farò una rotazione a sinistra dato che z è il figlio destro. Tale rotazione porterà comunque ad una violazione, infatti la rotazione a sinistra fa sì che il padre del nodo trattato diventi il figlio sinistro del nodo trattato e quindi comunque avrà padre e figlio di colore rosso (però il nodo z, ovvero il nodo che confrontiamo con lo zio sarà il padre del nodo z precedente). A questo punto se abbiamo fatto una rotazione a sinistra ci ritroveremo sicuramente nel Caso 2 ma ora effettuiamo una rotazione a destra. È quindi sufficiente ricolorare il padre e il fratello d_r. Infatti, ritornando alla situazione prima della rotazione a sinistra avevamo che z era il nodo 7 e il padre (nodo con key=2) era rosso; quindi, per l'osservazione fatta ad inizio pagina, il nonno di 7 è nero e anche il figlio destro nel nonno di z è nero per ipotesi (è lo zio di z e ci troviamo nel Caso 2). Successivamente la rotazioni a sinistra z diventa il nodo con key=2 e il nodo con key=7 diventa il padre di tale nodo. Questi sono entrambi rossi ma effettuando una rotazione a destra questa ci porterà in una situazione in cui il figlio nel nodo che è salito nella gerarchia è sicuramente nero e i figli del nodo rotato a destra sono sicuramente neri (dato che un figlio era lo zio di z e l'altro era il figlio destro del nodo risalito che era nero, in quanto il nodo risalito era la radice di un RB-Tree). Il cambio di colore, quindi, farà sì che sia il sottoalbero di destra che di sinistra del nodo risalito siano RB-Tree.

sinistra avevamo che z era il nodo 7 e il padre (nodo con key=2) era rosso; quindi, per l'osservazione fatta ad inizio pagina, il nonno di 7 è nero e anche il figlio destro nel nonno di z è nero per ipotesi (è lo zio di z e ci troviamo nel Caso 2). Successivamente la rotazioni a sinistra z diventa il nodo con key=2 e il nodo con key=7 diventa il padre di tale nodo. Questi sono entrambi rossi ma effettuando una rotazione a destra questa ci porterà in una situazione in cui il figlio nel nodo che è salito nella gerarchia è sicuramente nero e i figli del nodo rotato a destra sono sicuramente neri (dato che un figlio era lo zio di z e l'altro era il figlio destro del nodo risalito che era nero, in quanto il nodo risalito era la radice di un RB-Tree). Il cambio di colore, quindi, farà sì che sia il sottoalbero di destra che di sinistra del nodo risalito siano RB-Tree.

RB-Insert-Fixup

Ricapitolando RB-Insert-Fixup è costituito da un ciclo while, che viene eseguito fintantoché il padre di z (inizialmente il nodo inserito) resta rosso.

All'inizio di ogni iterazione del ciclo:

- Il nodo z è rosso.
- Se $p[z]$ è la radice, allora $p[z]$ è nero.
- In tutto l'albero c'è al massimo una violazione delle proprietà degli alberi rosso-neri, ed è o di tipo 2 o di tipo 4 (sono mutuamente esclusive):
 - Se è di tipo 2, si presenta perché z è la radice ed è rosso.
 - Se è di tipo 4, si presenta perché sia z che $p[z]$ sono rossi.

Caso 1:
ZIO ROSSO
~~Comincio colore dei nodi (nodo padre, ecco) e chiama lo modo inserito~~
Andiamo a vedere nel dettaglio l'algoritmo:

RB-Insert-Fixup (T,z)

```
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
        then y ← right[p[p[z]]] // zio
        if color[y] = RED
            then color[p[z]] ← BLACK
            color[y] ← BLACK
            color[p[p[z]]] ← RED
            z ← p[p[z]]
        else if z = right[p[z]]
            then z ← p[z]
            Left-Rotate(T,z)
            color[p[z]] ← BLACK
            color[p[p[z]]] ← RED
            Right-Rotate(T,p[p[z]])
        else (stesso codice di then
              con right e left scambiati)
            color[root[T]] ← BLACK
```

Caso base:

Caso base. Se l'albero è vuoto, il nodo inserito sarà rosso ma la radice per ipotesi è nera. In questo caso basta modificare il colore del nodo da rosso a nero.

(caso 2: ZIO NERO)

sono quindi se z è figlio destro nero, se farò rotazione a sinistro, se sono figlio destro nero. E se qualcosa ricorre i portano alla violazione dello RB-tree e quindi cambierò colore del padre e quindi così se necessario da di nuovo del RB-tree

Se siamo entrati nel ciclo allora il padre è rosso. Se il nodo inserito è il primo, allora non entreremo nel ciclo, dato che il padre del nodo inserito è $nil[T]$ che è nero per definizione. Analogamente se il nodo inserito è il secondo nodo, allora il padre sarà il nodo radice (e quindi è sicuramente nero). Usciti dal ciclo la radice viene posta a prescindere come colore nero (per ovviare la violazione di tipo 2). Se entriamo nel ciclo allora abbiamo una violazione di tipo 4. A questo punto dobbiamo distinguere due casi: se il padre di z è un figlio destro o sinistro. La distinzione dei due casi serve banalmente a determinare l'ordine delle rotazioni. A questo punto avremo due comportamenti simmetrici, per questo trattiamo solo un caso. Se il padre è figlio sinistro, allora lo zio e il figlio destro. A questo punto rivediamo i due casi (Caso 1 e Caso 2).

Caso 1.

Se lo zio è rosso allora si cambiano tutti i colori tranne quello del nodo z. Tale cambiamento conserva l'invariante, infatti, all'iterazione successiva $z' = p[p[z]]$ sarà rosso. Se z' è il nodo radice allora usciremo dal while ($p[z']$ è il nodo sentinella) e z' diventerà nero. I figli di destra e sinistra di z' saranno le radici di sottoalberi RB-Tree, questo perché il figlio destro di z' era lo zio di z ed inizialmente rosso (poi nero: quindi non da problemi al suo sottoalbero) mentre l'inversione di colore dell'albero di sinistra era necessario per risolvere la violazione di tipo 4 tra z e $p[z]$. se $p[z']$ è rosso, avremo una nuova violazione di tipo 4 che sarà risolta alla successiva iterazione.

Caso 2

Se lo zio è nero (vedi bene), allora dobbiamo distinguere due ulteriori casi:

- a. Il nodo z è il figlio destro. In questo caso z' diventa il padre di z e viene effettuata una rotazione a sinistra. Tale rotazione non farà altro che invertire z (il nodo all'inizio dell'iterazione) e z' (z dopo l'assegnazione $z = p[z]$). Il nodo zeta all'inizio dell'iterazione era il nodo radice di un RB-Tree e questo ci garantisce dopo la rotazione a sinistra che quando salirà nella gerarchia, il suo nodo destro sarà nero.

Questa situazione è importante per il passo successivo; infatti, ciò che faremo è cambiare i colori di $z = p[z']$ e $p[p[z']]$ che quindi diventeranno rispettivamente nero e rosso. Successivamente operiamo una rotazione a destra. Tale rotazione fa sì che $p[p[z']]$ vada a finire nel suo sottoalbero di destra e avrà come figli, il figlio destro di z e lo zio di z che abbiamo visto essere neri. Concludiamo che all'iterazione successiva il ciclo terminerà infatti il nodo z (nero) diventerà il padre di z' e quindi la condizione del while sarà falsa. Possiamo osservare che il ciclo termina indipendentemente dal fatto che $p[p[z]]$ sia o meno la radice del RB-Tree

- Il nodo z è figlio sinistro. In questo caso avremo un comportamento speculare, solo che avremo due rotazioni a destra, nello stesso ordine e nello stesso modo del caso a. La spiegazione e l'analisi di correttezza è analoga.

Come già detto nel caso in cui $p[z] = \text{right}[p[p[z]]]$ allora avremo un comportamento duale, in cui le rotazioni a destra diventano rotazioni a sinistra e viceversa.

Conclusione: L'algoritmo impiega al più $\lg n$, infatti questo rappresenta il caso peggiore in cui lo zio di z è rosso e quindi abbiamo il cambio di colore (che porterà ad una nuova violazione di tipo 4, fino ad arrivare alla radice). Nel best-case abbiamo solo le due rotazioni e quindi è costante.

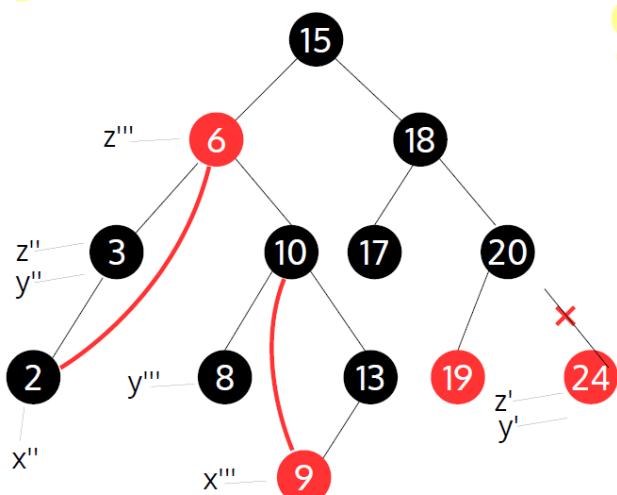
Eliminazione di un nodo

Anche l'eliminazione di un elemento richiede $O(\lg n)$.

Si invoca la Tree-Delete modificata:

- I riferimenti a NIL si sostituiscono con $\text{nil}[T]$
- Nella Tree-Delete, l'assegnazione al padre di x del padre di y (fatta per tagliare y), si fa solo se x non è nullo (y è una foglia). Nella RBDelete, questa assegnazione si fa sempre. Infatti, se x è la sentinella $\text{nil}[T]$, il padre della sentinella diventa il padre del nodo tagliato fuori.
- Se il nodo tagliato fuori è nero, si chiama RB-Delete-Fixup

Questa operazione esegue rotazioni e cambiamenti di colore per ripristinare le proprietà di un albero rosso-nero.



Se il nodo è un nodo foglia, allora è sufficiente eliminarlo.

OSS: un nodo rosso o è una foglia o ha due figli (neri), altrimenti è violata la proprietà 5.

Se quindi stiamo nel caso due dei BST allora il nodo che stiamo cancellando è necessariamente nero (e questo rappresenta il problema principale). Lo stesso problema si presenta se il nodo che vogliamo eliminare ha due figli ed è nero, se è rosso è come nel Tree-Delete.

Dizionari

Un **Dizionario** è un dynamic set che supporta operazioni di:

- Insert.
- Delete.
- Search.

Possiamo vedere un dizionario come un **set di coppie (chiave, valore)**. Vediamo alcuni applicazioni in cui questa struttura dati può essere utilizzata:

1. Word Processor: ovvero come strumento ricerca di una parola (conta anche il numero di parole). O come Doc Distance (un indice di "somiglianza" di due documenti).
2. Spelling Correction (vedi).
3. Tabelle di accesso ad un database (ad esempio quando abbiamo associato ad un username un certo oggetto).
4. Compilatori ed Interpreti. L'identificatore delle variabili viene ricercata nella **tabella dei simboli** (per la compilazione a due passi).
5. Nelle reti nei router abbiamo coppie (IP, link).
6. Nei SO nella memoria virtuale in cui dobbiamo associare un indirizzo virtuale ad uno fisico.

Supponiamo di dover allocare una struttura per l'associazione **USERNAME → OBJECT**.

Una prima soluzione è usare una **tabella ad acceso diretto**. Per l'implementazione si usa un array con m posizioni:

- nella posizione k si memorizza l'elemento la cui chiave è k .
- gli elementi devono avere chiavi distinte.
- se l'elemento con chiave k non è presente nell'insieme, nella posizione k si memorizza il puntatore NIL.

Questa soluzione prevede due problemi:

- lo spazio allocabile è finito - l'universo delle chiavi potrebbe essere troppo grande.
- La chiave deve corrispondere all'indice nella tabella - quindi la chiave è necessariamente un intero.

Pre-hashing

Il pre-hashing è una soluzione al secondo problema, in cui mappiamo i valori assunti dalla chiave con numeri interi. Questa soluzione è valida se l'universo delle chiavi è finito (generalmente è vera come ipotesi). Questa operazione può essere fatta in diversi modi che però garantiscono le seguenti proprietà:

1. Se $x = y$ allora $h(x) = h(y)$, ovvero applicando la funzione di hash sui due valori otteniamo lo stesso hash.
2. Non vale l'implicazione opposta; ovvero se $h(x) = h(y)$ allora non è necessariamente vero che $x = y$. Diremo infatti che se abbiamo $h(x) = h(y)$ per $x \neq y$ allora abbiamo una **collisione**.
Ad esempio se $x = '\text{0B}'$ e $y = '\text{0C}'$ allora $h(x) = h(y) = 64$.
3. Invariante nella stessa esecuzione: ovvero se usiamo $h(x)$ sullo stesso valore in due istanti diversi allora otteniamo lo stesso hash.

OSS: Questa ha una forte applicazione della crittografia. Viene usato per verificare l'integrità del contenuto di un pacchetto, ad esempio, nelle reti di calcolatori. Ma non solo, ad esempio se effettuiamo il download di un file possiamo calcolarne l'hash e paragonarlo con quello fornito dalla sorgente de file. Se vogliamo usarle nelle reti dobbiamo avere proprietà più stringenti:

1. Sue chiavi molto vicine portano ah hash molto lontani.
2. La funzione $h(x)$ non deve essere invertibile.

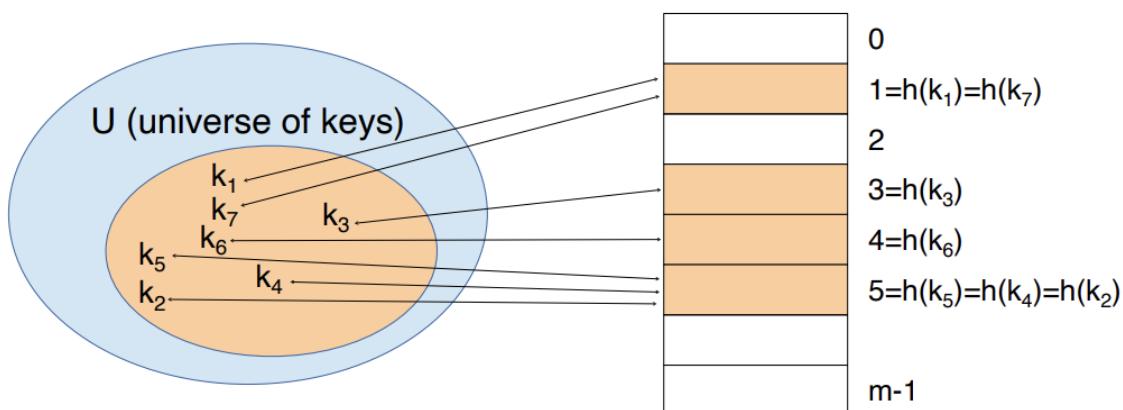
In questo caso viene utilizzato ad esempio per firme digitali.

Tabelle Hash

Per

Dobbiamo comunque risolvere il problema dello spazio allocabile. Per tale problema possiamo ridurre l'universo delle chiavi U ad una dimensione (gestibile) m , in modo tale che: $|U| \gg m$. Uso quindi una **funzione di hash** – $h: U \rightarrow \{0, 1, \dots, m - 1\}$

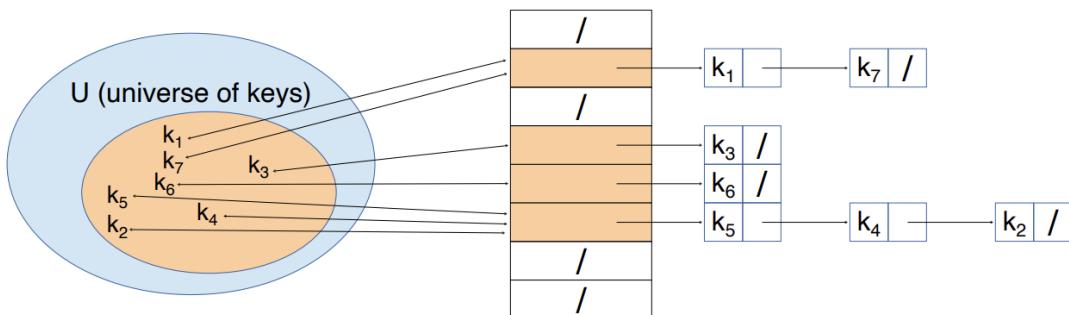
Devo quindi mappare gli elementi dell'universo nell'insieme finito m . Questo porta ovviamente a delle collisioni. Vogliamo minimizzare il numero di collisioni e ciò lo facciamo scegliendo un'opportuna **funzione hash**.



N.B: Le tabelle hash non sono una soluzione sempre valida; infatti, non supportano molte operazioni supportate da altre strutture come gli alberi (successore e predecessore, ecc.).

Come possiamo osservare dall'immagine due chiavi possono portare allo stesso hash generando una collisione. Vediamo alcune possibili soluzioni *soluz*

Chaining



Una prima soluzione è concatenare le chiavi che generano collisioni sfruttando un linked list. Questa soluzione prevede nel caso peggiore una complessità lineare; ovvero tutti i valori vengono mappati con la stessa chiave.

Chained-Hash-Insert (T,x)

insert x at the head of list T[h(key[x])]

Chained-Hash-Search (T,k)

search for an element with key k in list T[h(k)]

Chained-Hash-Delete (T,x)

delete x from the list T[h(key[x])]

Questa soluzione caratterizza un inserimento costante ma operazioni di ricerca e cancellazione lineari (questo perché operiamo su una lista dove tale operazioni sono lineari nel caso peggiore).

Per risolvere il problema della ricerca lineare dobbiamo minimizzare il numero di collisioni.

Simple Uniform Hashing

Le prestazioni di una tabella hash dipendono dalla capacità della funzione hash di distribuire gli elementi tra le liste concatenate. Dobbiamo fare un'ipotesi molto forte, ovvero che ogni chiave abbia la stessa probabilità di finire in ciascuna delle m liste concatenate, indipendentemente da dove sono finite le altre chiavi. La probabilità sarà sempre $1/n$ (dove n è il numero di chiavi nella tabella) di finire in una data posizione.

- Detta n_j la lunghezza j-esima lista allora $n = \sum_{j=0}^{m-1} n_j$.
- Definiamo inoltre il **load factor** - α : $\alpha = E[n_j] = n/m$ che quindi è proprio la lunghezza media di ogni lista. Se ho m liste e posso supporre che la distribuzione sia uniforme e quindi in ogni lista avrò mediamente n/m chiavi.

Sotto queste assunzioni; il tempo atteso per cercare un elemento sarà: $O(1 + \alpha)$, dove il +1 è dato dal calcolo dell'hash.

Nel caso base, ovvero se la **ricerca non ha successo** (non c'è la chiave nella tabella), allora:

$$E[n_j] = \frac{n}{m} = \alpha$$

Infatti, una chiave non presente nella tabella ha una uguale probabilità di essere associata ad una delle m liste. Quindi calcolato l'hash dovrà cercare proprio nella j-esima, dove avrò mediamente n/m chiavi che dovrò scorrere tutte. Ho quindi un tempo proporzionale ad α e quindi in tal caso abbiamo una complessità $\Theta(1 + \alpha)$.

Se invece la **ricerca** di un elemento x con chiave k **ha successo** allora il numero di elementi esaminati durante la ricerca è 1 più il numero di elementi che precedono x nella sua lista.

Indichiamo con: $X_{ij} = 1$ se $h(K_i) = h(K_j)$; 0 altrimenti.

La probabilità che due chiavi vanno a finire nella stessa lista ciò accada è $1/m$ e quindi:

$$E[X_{ij}] = \Pr\{h(K_i) = h(K_j)\} = \frac{1}{m}$$

Il numero di elementi esaminati durante la ricerca è 1 più il numero di elementi che precedono x nella sua lista.

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

La sommatoria più interna alla prima riga ci fornirà il numero di elementi considerati nella ricerca.

Se infatti immagino n come l'insieme di elementi che si ottiene concatenando le m liste, allora quando inserirò l' i -esimo elemento nella sua lista, non ha senso considerare gli elementi prima di i , dato che saranno inseriti nelle liste precedenti e posso considerare quindi tutti gli elementi successivi ad i , infatti se non saranno inseriti nella stessa lista di i allora $X_{ij} = 0$. Il $+1$ è invece legato al tempo di calcolo della funzione di hash. Poiché vogliamo ottenere un tempo medio di ricerca, ciò che possiamo fare è mediare su tutti gli n elementi inseriti nella tabella di hash. Quindi consideriamo la somma dei tempo di ricerca di tutti gli n elementi e poi dividiamo per n (ne facciamo quindi una media). I restanti sono semplici calcoli algebrici che ci porteranno alla conclusione che il tempo di ricerca è $\Theta(1 + \alpha)$. Inizialmente abbiamo parlato di tempo costante. Questa osservazione è vera solo se $n = O(m)$, o anche $m = \Omega(n)$. Infatti solo in questo caso α è costante e quindi $\alpha = O(1)$.

Il problema è che quest'ipotesi non vale, specialmente se non è nota la distribuzione di probabilità delle chiavi. Un modo per rendere vera l'ipotesi è approssimare il problema ad un hashing uniforme.

Lo posso fare in due modi:

1. Uso un'opportuna funziona di hash che abbia buone prestazioni quando la distribuzione delle chiavi hash non è nota.
2. Posso fare ipotesi sulla distribuzione delle chiavi, ad esempio posso immaginare di avere stringhe molti simili tra di loro (immaginiamo il caso in cui vogliamo dichiarare delle variabili).
3. Uso la randomizzazione.

Funzioni Hash

Una prima assunzione che faremo è che le chiavi appartengano all'insieme dei numeri naturali. Infatti, è facile convertire chiavi di altro tipo in numeri naturali. Ad esempio, se vogliamo convertire la stringa "pt", possiamo usare il codice ASCII su 7 bit dove 'p' = 112 e 't' = 116. A questo punto "pt" = 112 * 128 + 116 = 14452 dove 128 = 2⁷. Su questa base un primo modo per costruire la funzione è sfruttare una qualsiasi regolarità presentata dalle chiavi (ad esempio si può provare che le chiavi molto spesso sono multipli interi di 2 o di 10). Su questa ipotesi possiamo costruire il vari metodi.

Metodo della divisione

Poniamo la funzione di hash:

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

Quindi la funzione di hash restituisce interi tra [0, ..., m - 1]. Se sceglio m come potenza di 2 ($m = 2^q$) allora $h(k)$ è dato dagli ultimi q bit della rappresentazione di k. Ad esempio, se prendo in ingresso multipli di 10 (10 = 1010, 20 = 10100, 30 = 11110, 40, ...) e pongo $m = 2^2 = 4$ allora avrò hash (10=2, 00=0, 10=2, 0, ...). Questa soluzione non è ottimale, infatti, se si presentano multipli di 10 o di 2 in ingresso ho chiavi di hash che si alternano. Se immaginassimo un compilatore vorremmo che due variabili con nomi simili abbiano hash diversi. Questa situazione si presenta anche se scelgo m come multiplo di 10. In genere, una buona scelta è un numero primo non troppo vicino ad una potenza di 2. Se abbiamo una stima di n e tolleriamo liste di tre elementi, ad esempio, possiamo scegliere m come numero primo intorno a $n/3$.

Tornando all'esempio della stringa "pt" se pongo $m = 127$ e $h(k) = k \bmod m$ avremo che:

- $h("pt") = [112(127 + 1) + 116] \% 127 = 112 + 116$.
- $h("tp") = [116(127 + 1) + 112] \% 127 = 116 + 112$.

OSS: Questa scelta è invariante rispetto alle permutazioni, infatti, "pt" e "tp" portano alla stessa hash.

Metodo della moltiplicazione

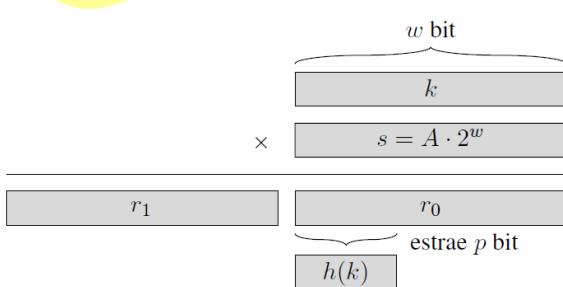
Poniamo la funzione di hash:

$$h(k) = [m(kA - \lfloor KA \rfloor)]$$

Questo metodo moltiplica k per una costante A: $0 < A < 1$ e prende la parte decimale (sottraiamo per l'approssimazione per difetto ad intero del prodotto). E infine moltiplichiamo per m e approssimiamo il tutto per difetto.

L'implementazione risulta essere efficiente se poniamo:

- $A = s/2^w$ con $s < 2^w$ e s Intero. Dove w è il numero di bit per rappresentare la chiave.
- $m = 2^p$.



Ad esempio:

Faccio in modo che la chiave dipendi dai bit centrali della rappresentazione. Se pongo $w = 4$ (la word), $k = 6$, $p = 3$, $s = 9$ otterrò: $A = 9/6 = 0,5625$ e k in binario sarà 0110 che moltiplico per A

$$k \cdot s = 110110$$

$$k \cdot A = (k \cdot s)/2^w = 11,0110$$

Di cui la parte decimale è 0,0110 che moltiplicata per $m = 11,0$ ci dà la parte intera 11 (ovvero 3).

Hashing universale

Se la funzione di hash è nota, si può determinare un insieme di chiavi che vengono associate tutte alla stessa lista (può essere sfruttato per effettuare un attacco). L'unica contromisura è rendere la funzione hash aleatoria. L'idea dell'hashing universale è quella di scegliere la funzione di hash (all'inizio dell'esecuzione) in maniera casuale da una classe di funzioni appositamente progettata. La funzione di hash non viene cambiata ad ogni utilizzo della funzione di hash ma ad ogni esecuzione del programma. In tal modo nessuna sequenza di chiavi può ricadere sempre nel caso peggiore (evitiamo la presenza di un caso peggiore sistematico).

Def: Sia H una classe finita di funzioni hash che mappano un dato universo di chiavi U sull'insieme $\{0, 1, \dots, m - 1\}$. Tale classe è detta universale se per ogni coppia di chiavi distinte $k, l \in U$, il numero di funzioni hash $h \in H$ per cui $h(k) = h(l)$ è al più $|H|/m$. In altre parole, con una funzione hash scelta a caso da H , la probabilità di una collisione fra due chiavi distinte k e l non è maggiore della probabilità $1/m$ di una collisione nel caso in cui $h(k)$ e $h(l)$ fossero scelte in modo casuale e indipendente dall'insieme $\{0, 1, \dots, m - 1\}$. Da cui La possibilità di collisione è al più $1/m$.

Realizzazione

Sia p un numero primo sufficientemente grande da assicurare che ogni chiave k è nell'intervallo $[0, \dots, p - 1]$. Consideriamo $Z_p = \{0, 1, \dots, p - 1\}$ e $Z_p^* = \{1, 2, \dots, p - 1\}$.

$$h_{a,b}(k) = ((ak + b)\%p)\%m \text{ con } a \in Z_p^* \text{ e } b \in Z_p$$

Ricordiamo che m è il numero di righe della tabella hash. In questa classe possiamo avere $p(p - 1)$ funzioni di hash. Per la costruzione di questa classe m non deve necessariamente essere un numero primo. Si può dimostrare che in questa classe vale la proprietà dell'hashing universale.

Dimostrazione

Supponiamo di scegliere due chiavi $k \neq l$ da Z_p e consideriamo le funzioni di hash:

- $r = (ak + b)\%p$
- $s = (al + b)\%p$

Parte 1: Dobbiamo verificare che $r \neq s$ dato che $r - s \neq 0$. Valutiamo quindi che:

$$r - s = a(k - l)\%p$$

Sappiamo che $a \neq 0$ e $k - l \neq 0$ per ipotesi. Questo però non è sufficiente a dire che $r - s \neq 0$. Potrebbe ad esempio accadere che $a(k - l)$ sia multiplo intero di p . Questa affermazione decade se a e $(k - l)$ sono relativamente primi con p , ovvero se non hanno divisori in comune. Questo è vero dato che a è certamente minore di p , quindi non possiamo avere un multiplo di un numero primo, per valori minori del numero primo stesso. Lo stesso vale per la differenza di due valori minori del numero primo. A questo punto abbiamo dimostrato che per $k \neq l$ allora $r \neq s$ e quindi non abbiamo collisioni.

Parte 2: In generale fissate r e s , possiamo calcolare i valori di a e b . Ancora più in generale se abbiamo $p(p - 1)$ possibili scelte della coppia (a, b) abbiamo di conseguenza $p(p - 1)$ possibili funzioni r e s . Quindi, per qualsiasi coppia di input k e l , se sceglioamo (a, b) uniformemente a caso da $Z_p^* \times Z_p$, la coppia risultante (r, s) ha la stessa probabilità di essere qualsiasi coppia di valori distinti modulo p . Ne consegue che la probabilità che le due chiavi distinte k e l collidano è uguale alla probabilità che $r = s \% m$ quando r e s sono scelte a caso come valori distinti modulo p .

Fissato r , tale probabilità

$$\Pr\{k \text{ e } l \text{ collidano con } r \text{ fissato}\} = \Pr\{r = s \% m\} = \left\lceil \frac{p}{m} \right\rceil - 1 \leq \frac{p+m-1}{m} - 1 = \frac{p-1}{m}$$

Vediamo come si arriva al terzo membro dell'equazione empiricamente. Fissiamo $p=17$, $m=4$ e $r=2$.

Fissato r vediamo per quali valori di $s < p$ si verifica $r = s \% m$.

Se $r = 2$ avverrà per 6, 10, 14 (va incluso r stesso e quindi 2). Se ora cambiamo r , ad esempio $r = 0$ avremo 4, 8, 12, 16 (non includiamo 0 dato che $a \in \mathbb{Z}_p^*$). Arriviamo al quarto membro (dopo \leq) applicando la proprietà $\left\lceil \frac{a}{b} \right\rceil \leq \frac{a+b-1}{b}$.

$$\text{In conclusione, } \Pr\{h_{a,b}(k) = h_{a,b}(l)\} = \frac{1}{(p-1)} \cdot \Pr\{k \text{ e } l \text{ collidano con } r \text{ fissato}\} \leq \frac{1}{m}$$

Dobbiamo moltiplicare per $(p-1)$, infatti poiché fissiamo r , allora abbiamo $(p-1)$ possibili valori di s .

OSS: se cambiamo funzione di hash, e abbiamo un tabella di hash già popolata, dobbiamo effettuare il **re-hashing** (ovvero ricostruire la tabella in base alla nuova funzione utilizzata).

Ridimensionamento

Con la strategia precedente questo modo abbiamo risolto il problema di avere un hashing universale e quindi intrinsecamente più sicuro. Ora dobbiamo gestire il problema del ridimensionamento. Il metodo delle liste concatenate funziona solo se $m = \Omega(n)$, se questo vincolo non è rispettato, devo ridimensionare (devo allargare la tabella). Quindi se voglio passare da m a $m' > m$, perché $n > m$, devo allocare uno spazio per la nuova tabella di dimensione m' e per ogni elemento della vecchia tabella devo effettuare il **re-hashing**. Avrò quindi un costo lineare (il re-hashing è costante) e quindi $\theta(m+n) = \theta(n)$.

Posso procedere con due approcci possibili per il ridimensionamento di m :

1. Posso ad esempio devo scegliere $m+1$, o in generale $m+k$. Quindi ogni volta $n > m$, $m' = m+k$, con k fissato. Ipotizziamo $k=1$; ogni volta n aumenta e quindi ad ogni inserimento dovrò aumentare la dimensione della tabella di hash. Questo significa che se partiamo da n elementi e tabella di dimensione m , quando arriveremo a n' elementi $> n$ (per $n' - n$ inserimenti consecutivi), allora avremo $1 + 2 + 3 + \dots + n' = \sum_{i=0}^{n'} i = O(n^2)$.

N.B:

- per semplicità $n' = n$.
- incrementare la tabella ogni volta di un fattore costante, non è una buona scelta.

2. Una seconda tecnica prevede un fattore moltiplicativo della tabella k , ogni qualvolta n diventa k volte m e quindi se $n > k \cdot m$ allora $m' = km$. Ad esempio, possiamo raddoppiare la tabella ($k=2$) e in questo modo avrò una complessità del tipo $1 + 2 + 4 + \dots + n' = \sum_{i=0}^{\lg n} 2^i = \frac{1-2^{\lg n}}{1-2} = O(n)$. Quindi in questo caso abbiamo una complessità lineare e quindi guadagno nel fatto che eseguo $\lg n$ operazioni di re-hashing.

Questa è però un analisi troppo conservativa. Ciò che si può fare è introdurre il concetto di **analisi dei costi ammortizzato**: di una sequenza di operazioni ricavo il costo medio per operazione. Il **costo ammortizzato** in questo caso è costante e quindi $\theta(1)$.

OSS: i due approcci cambiano in termini di complessità quando $n > m$. Ovvero mentre col primo approccio farò ad ogni inserimento che verifica $n > m$, un incremento di m e quindi un re-hashing sugli n elementi nella tabella. Quindi dopo che avrò fatto un certo numero di inserimenti consecutivi, avrò una complessità n^2 . Nel secondo approccio cambia il numero di operazioni di re-hashing, dato che raddoppiando la dimensione di m , dovrò effettuare un nuovo ridimensionamento dopo più inserimenti. Questo secondo approccio potrebbe però portare ad un incremento eccessivo della bella di hashing, quindi dobbiamo prevedere un'operazione di **cancellazione**. Se m diventa troppo più grande di n , dobbiamo ridurre la cancellazione e quindi dobbiamo passare da m a $m' < m$. Un primo approccio, sarebbe il duale al precedente, se prima ho raddoppiato ora dimezzo $m' = m/2$ quando $n < m/2$. Lo svantaggio è che si oscillerebbe tra raddoppiare e dimezzare m (il costo ammortizzato diventa lineare). Un altro approccio è ridurre quando $n \leq m/k$ con $k > 2$.

Indirizzamento aperto

Questo metodo prevede di non utilizzare le liste ma di implementare la tabella hash usando dei vettori di dimensione m . In questo caso la gestione delle collisioni non utilizza una lista, ma si inserisce l'elemento in una determinata posizione dell'array. Le collisioni si risolvono quindi consentendo ad un elemento con una data chiave di trovarsi in un insieme di possibili posizioni. In fase di inserimento e ricerca, si valutano solo tali posizioni.

OSS: Guadagnammo maggiore memoria per non avere memorizzato i puntatori. Il che offre alla tabella hash un gran numero di slot, a parità di memoria occupata, consentendo potenzialmente di ridurre il numero di collisioni e di accelerare le operazioni di ricerca.

Formalmente, estendiamo la funzione hash in modo da essere funzione sia della chiave che del numero di "tentativi":

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

È richiesto che la **sequenza di probing** $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$ sia una permutazione di $\langle 0, 1, \dots, m - 1 \rangle$ per ogni chiave k . Infatti, solo sotto questa condizione siamo in grado di riempire completamente l'array.

Linear probing

Data la funzione hash ausiliaria $h' : U \rightarrow \{0, 1, \dots, m - 1\}$ considera:

$$h(k, i) = (h'(k) + i) \% m : i = 0, 1, \dots, m - 1$$

Data la chiave k , il primo slot esaminato è $T[h'(k)]$, che è lo slot dato dalla funzione hash ausiliaria; il secondo slot esaminato è $T[h'(k) + 1]$ e, così via, fino allo slot $T[m - 1]$. Poi, la scansione riprende dagli slot $T[0], T[1], \dots, T[h'(k) - 1]$. Ad esempio:

0		$h'(k) = k \bmod 13$
1	79	
2		
3		
4	69	$k=43 \Rightarrow h'(43) = 43 \bmod 13 = 4$
5	98	$(4+0) \bmod 13 = 4$ occupato
6		$(4+1) \bmod 13 = 5$ occupato
7	72	$(4+2) \bmod 13 = 6$ libero
8		
9		
10		
11	50	
12		

Se non ho una permutazione non è detto che copro tutte le celle. Apriamo una breve parentesi sulle operazioni:

N.B: se devo cercare una certa chiave, ne effettuo l'hash, e cerco a partire da questo elemento con una ricerca lineare il nostro elemento, finché non trovo una cella vuota. Questa cosa è vera se non effettuo cancellazioni. Se effettuo una cancellazione, infatti, non posso semplicemente svuotare la cella, ma devo inserire un flag di valore non valido. In questo caso l'inserimento rimane lo stesso, ma scrive non alla prima cella libera ma anche in una cella non valida.

Il problema di questa soluzione è che abbiamo un effetto convoglio, ovvero vado a creare un **cluster primario** e quindi si possono formare lunghe file di slot occupati, che aumentano il tempo medio di ricerca.

Quadratic probing

Analogamente a prima ma sfruttiamo una legge quadratica. Data la funzione hash ausiliaria $h': U \rightarrow \{0, 1, \dots, m - 1\}$ considera: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \% m: i = 0, 1, \dots, m - 1$.

La prima posizione provata è $h'(k)$, le successive sono separate di un offset che cresce con legge quadratica rispetto ad i . Questa tecnica funziona molto meglio della scansione lineare, ma per fare pieno uso della tabella hash, i valori di c_1, c_2 e m sono vincolati (vanno quindi dimensionati). Se due chiavi portano alla stessa posizione iniziale di scansione (ovvero se $l \neq k$ e $h'(l) = h'(k)$) allora avrò anche la stessa sequenza di sanzione e quindi ho comunque un effetto convoglio (anche se più lieve). Parliamo in questo caso di **cluster secondario**.

Double hashing

Date le funzioni hash ausiliarie $h_1, h_2: U \rightarrow \{0, 1, \dots, m - 1\}$ usa:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \% m: i = 0, 1, \dots, m - 1$$

La prima posizione provata è $h_1(k)$, le successive sono separate di un offset che dipende anch'esso da k (tramite $h_2(k)$). Ad esempio:

0		
1	79	$h_1(k) = k \bmod 13$
2		$h_2(k) = 1 + (k \bmod 11)$
3		
4	69	
5	98	$k=14 \Rightarrow h_1(14) = 14 \bmod 13 = 1$
6		$h_2(14) = 1 + (14 \bmod 11) = 4$
7	72	
8		$(1+0*4) \bmod 13 = 1$ occupato
9		$(1+1*4) \bmod 13 = 5$ occupato
10		$(1+2*4) \bmod 13 = 9$ libero
11	50	
12		

Osserviamo che h_2 e m devono essere relativamente primi, altrimenti non riusciamo a coprire tutte le posizioni. Se ad esempio $h_1=7$, $h_2=4$ e $m=10$ allora copriremo solo le posizioni: 7, 1, 5, 9, 3, 7, 1, 5, 9, 3, 7, ...

Possibili modi per garantire questa condizione sono:

- Se m è una potenza di due e h_2 deve restituire sempre un numero dispari.
- Se m è un numero primo e h_2 restituisce sempre un intero minore di m :
$$h_1(k) = k \% m, h_2(k) = 1 + (k \% m')$$
 con $m' < m$ (es. $m' = m - 1$)

Operazioni

Inserimento

Hash-Insert (T,k)

```
i ← 0
repeat
j ← h(k,i)
if T[j] = NIL
    then T[j] ← k
        return j
    else i ← i+1
until i = m
error "hash table overflow"
```

Per inserire un elemento si provano tutte le posizioni determinate dalla sequenza dei tentativi fino a che non si trova una libera.

Ricerca

Hash-Search (T,k)

```
i ← 0
repeat
j ← h(k,i)
if T[j] = k
    then return j
i ← i+1
until T[j] = NIL or i = m
return NIL
```

Per cercare un elemento si provano tutte le posizioni determinate dalla sequenza dei tentativi fino a che non si trova una libera. In tal caso, l'elemento non è presente in tabella.

Eliminazione

Se eliminiamo una chiave, non possiamo semplicemente sostituirla con il valore NIL. Se facessimo ciò non funzionerebbe l'algoritmo di ricerca per gli elementi inseriti successivamente in altre posizioni perché la posizione dell'elemento eliminato era occupata. Possiamo utilizzare un altro valore speciale **DELETED**. Quindi l'eliminazione è banalmente una ricerca (se usiamo la chiave dell'elemento che vogliamo eliminare) che poi porrà l'elemento cercato a **DELETED**. Dovremo poi modificare l'operazione di inserimento che deve inserire se $T[j]$ è **NIL** o se è **DELETED**.

Quando utilizziamo il valore **DELETED**, però, il tempo di ricerca non dipende più dal carico $\alpha=n/m$. Infatti, andremo a considerare un n più grande di quello reale. Se le eliminazioni sono eccessive, la ricerca considererà anche le celle **DELETED**, che se sono molte aumenta la complessità della ricerca.

Quando è necessario consentire l'eliminazione delle chiavi, le tabelle hash con liste concatenate sono da preferire.

Hashing Uniforme

Nelle soluzioni analizzate si prevede che n sia minore o al più uguale di m . Se utilizziamo l'indirizzamento aperto quest'ipotesi è realizzabile proprio perché risparmiamo lo spazio usato dai puntatori in memoria. Inoltre, usando un array siamo in grado di ottimizzare le operazioni sfruttando la cache. Sostanzialmente un array sfrutta il caching e maggior numero di locazioni a parità di memoria (rispetto alla soluzione con le linked list).

Un'ulteriore ipotesi che facciamo è quella dell'**hashing uniforme**, ovvero che ogni chiave abbia la stessa probabilità di avere come sequenza di scansione una delle $m!$ permutazioni di $(0, 1, \dots, m - 1)$. L'hashing uniforme estende il concetto di hashing uniforme semplice definito precedentemente al caso in cui la funzione hash produce, non un singolo numero, ma un'intera sequenza di scansione. Poiché è difficile implementare il vero hashing uniforme, nella pratica si usano delle approssimazioni accettabili (come il doppio hashing). Con le soluzioni viste infatti riusciamo al più a generare m^2 sequenze di scansione diverse (e non $m!$).

Prestazioni

In generale le prestazioni delle operazioni di inserimento e ricerca dipendono da α . La probabilità di successo dell'inserimento è:

- Al primo tentativo $\frac{m-n}{m} = p$. Ovvero il numero di celle vuote diviso il numero di celle disponibili.
- Al secondo $\frac{m-n}{m-1} > p$, dato che avrò occupato una cella prima. Il numero di tentativi che devo fare è l'inverso di p . #tentativi $\geq \frac{1}{p} = \frac{1}{1-\alpha}$. Ricordiamo che $\alpha = m/n$.

Un ragionamento analogo viene fatto per la ricerca.

Inoltre, $\alpha \leq 1$ per quanto detto all'inizio di questo paragrafo. Quanto più α tende a 1 quanto il numero di tentativi aumenta. Ad esempio se $\alpha = 0.99$ allora devo fare cento tentativi. Quindi è vicino ad una complessità lineare. Se $\alpha > 0.5$ ho dei problemi. Si preferisce sprecare spazio per abbassare α - usando un array più grande (m maggiore).

Applicazione: string matching

Lo scopo di tale algoritmo è cercare in una stringa una determinata sottostringa.

Problema: ho una stringa T e una sottostringa S: I(T)>>I(S) e devo trovare le occorrenze di S in T.

Possiamo usare un approccio a forza bruta in cui si fa un doppio ciclo. In questo modo la complessità è $s(t-s)=st-s^2=O(st)$. Infatti, ciò che facciamo a partire dal primo carattere della stringa T e lo confrontiamo con il primo carattere della stringa S. Se il confronto è vero si continuano a confrontare gli altri caratteri. Se è falso, si passa al secondo carattere della stringa T e lo si confronta nuovamente con il primo carattere della stringa S. Quindi nel caso peggiore, scorriamo la stringa S, $(t-s)$ volte. Per fare di meglio, possiamo usare l'algoritmo: **Karp-Rabin**.

Karp-Rabin(S[1,..,n],P[1,..,m])

```
1 def RabinKarp(s, p):
2     n, m = len(s), len(p)
3     hp = hash(p)
4     for i in range(0, n-m):
5         hs = hash(s[i:i+m])
6         if hs == hp:
7             if s[i:i+m] == p[0:m]:
8                 return i
9     return -1 # not found
```

Ciò che facciamo è calcolare l'hash delle sottostringa in questione P e poi $n-m$ volte andiamo a calcolare l'hash di S, ovviamente considerando solo m caratteri. Ad ogni iterazione andiamo a confrontare gli hash, se effettivamente sono uguali allora abbiamo una buona probabilità di aver individuato la sottostringa nella stringa S. Infatti se la condizione è verificata allora si confrontano le stringhe. Osserviamo che questo algoritmo avrebbe

complessità (idealmente) solo nel caso peggiore pari a $O(m(n - m)) = O(nm)$, ovvero solo quando dobbiamo confrontare le due sottostringhe. In realtà questo non è vero dato che il calcolo dell'hash della sottostringa di lunghezza ha proprio complessità m , dato che l'hash va calcolato per ogni carattere. Ad esempio, se stiamo utilizzando una rappresentazione a 256 (8 bit), abbiamo che la stringa "abr" viene codificata come: $abr = a \cdot 256^2 + b \cdot 256 + r = 97 \cdot 256^2 + 98 \cdot 256 + 114$. Se ora applichiamo la funzione hash, otteniamo che l'hash può essere ricavato come la somma dell'hash dei singoli addendi. Possiamo ora però sfruttare due caratteristiche:

1. Ipotizziamo che la funzione hash della sottostringa non da collisioni. Ad esempio, possiamo porre la funzione $hash = key \% max_rappresentazione$. Con 3 caratteri avremo $256 \cdot 256^2 + 256 \cdot 256 + 256$
2. Ad ogni iterazione la sottostringa che consideriamo differisce al più per un carattere; infatti, la nuova sottostringa è data dalla precedente a cui si rimuove il primo carattere (skip) e se aggiunge in coda un altro carattere (append).

Sulla base della seconda caratteristica possiamo realizzare il **rolling hash**, rendendo costante il calcolo dell'hash. A partire dalla seconda iterazione, si sfrutta la chiave hash ottenuta in precedenza e:

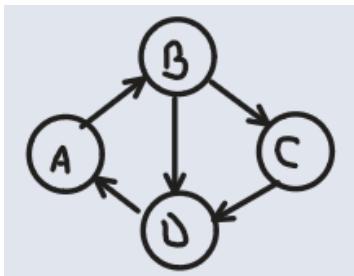
1. Si effettua uno **skip** e quindi rimuoviamo il primo carattere della vecchia sottostringa (carattere $i-1$). Per fare ciò è sufficiente sottrarre al vecchio hash $S[i - 1] \cdot 256^2$.
2. Successivamente effettuiamo un **append**. Per fare ciò moltiplichiamo quello che abbiamo ottenuto precedentemente per 256 e poi ci aggiungiamo il nuovo termine $S[i + m]$ e ne calcoliamo l'hash.

CAPITOLO 3 - BACKTRACKING

Problema: Sia $G = (V, E)$ un grafo diretto e pesato tale che tutti i pesi siano positivi. Siano v e w due vertici in G , e $k \leq |V|$ (numero intero). Progettare un algoritmo per trovare il percorso più breve da v a w che contiene esattamente k archi.

OSS: Il problema di dover trovare il minimo percorso lo si ritrova soprattutto nelle reti di calcolatori (Dijkstra).

Percorso minimo tra due nodi



Voglio trovare il percorso minimi da un nodo x ad uno y , con k - numero di archi percorsi fissato. Denotiamo con $d(x, y, k)$ il costo per andare da x a y con k archi. Ad esempio:

- $d(A, D, 3) = \{A, B, C, D\} = 11$
- $d(A, D, 4) = \text{non esiste} = \infty$
- $d(A, D, 5) = \{A, B, D, A, B, D\} = 26$

Per ottenere tale risultato possiamo usare vari approcci.

Soluzione 1 - Brute Force

Possiamo usare una **ricerca esaustiva**, in cui proviamo tutti i percorsi possibili con k archi. Si parte dal nodo iniziale e si prova ad andare ad un secondo nodo (provando tutti i nodi). Dobbiamo costruire tutti i possibili percorsi e si sceglie il minimo. Questo tipo di approccio ha una complessità $O(n^k)$ per un grafo con n nodi.

Soluzione 2 – Programmazione Dinamica

Possiamo pensare di decomporre il nostro problema (ci rifacciamo ad un ragionamento di tipo induttivo). Possiamo porre:

$\delta(x, y, k) = \min_z \{\delta(x, z, k - 1) + w(z, y)\}$ dove $w(z, y) = \delta(z, y, 1)$ ovvero è il costo per andare da x a y . Abbiamo però bisogno di un caso base. Il caso base dipende da come poniamo il problema:

$$1. \text{ Posso porre } k = 0 \text{ da cui } \delta(x, y, 0) = \begin{cases} 0 & \text{se } x = y \\ \infty & \text{se } x \neq y \end{cases}$$

o in alternativa

$$2. \text{ Posso porre } k = 1 \text{ da cui } \delta(x, y, 1) = \begin{cases} \infty, & \text{se non esiste l'arco} \\ w(x, y), & \text{altrimenti} \end{cases}$$

Dobbiamo ora valutare complessità di questa soluzione. Alla prima iterazione ciò che farò è fissare $w(z, y)$ e poi far variare z in $\delta(c, z, k - 1)$. Poiché z viene scelto tra $n - 1$ nodi, avrò $n \cdot n$, dato che fisserò $w(z, y)$ n volte. Ovviamente non termina qui, infatti questo verrà ripetuto k volte. Non è ancora sufficiente per la complessità, infatti ogni volta viene fatta una ricerca del minimo che comporta una complessità aggiuntiva $O(n)$. Ne concludiamo che la complessità è $O(kn^2n) = O(kn^3)$.

Possiamo rappresentare questa soluzione come una matrice. Devo creare k matrici che memorizzano il minimo percorso da x a y per qualsiasi nodo $x \neq y$, impiegando k archi.

La complessità per costruire è riempire tali matrici sarà k (ovvero il numero di matrici che dovrà costruire) $\times n$ (numero di nodi e quindi dimensione della matrice). Le $k - 1$ matrici avranno come costo il calcolo del minimo e quindi la complessità è kn^2n dove n è il costo per il calcolo del minimo. Per semplicità di notazione poniamo $\delta(x, y, k) = \delta_k(x, y)$.

Se provassimo ad esempio a costruire le matrici, avremmo:

Per $k = 0$ abbiamo una matrice con tutti 0 banalmente.

Per $k = 1$

	A	B	C	D
A	∞	1	∞	∞
B	∞	∞	2	10
C	∞	∞	∞	3
D	4	∞	∞	∞

Per costruire questa tabella è sufficiente guardare il grafo e quindi gli archi da un nodo x ad un nodo y .

Per $k = 2$

	A	B	C	D
A	∞	∞	3	11
B		∞		
C			∞	
D				∞

Se voglio andare da A a D con $k = 2$, devo vedere i nodi da cui partono archi entranti in D e quindi C e B (devo vedere quali nodi della tabella per $k = 1$ hanno un valore che non sia infinito nella colonna D). In tal modo valuto $w(B, D)$ e $w(C, D)$ guardando la tabella per $k = 1$. A questo punto devo valutare $\delta_1(A, B)$ e $\delta_1(A, C)$. Per valutare questi valori devo guardare la tabella precedente.

Per $k=3$

	A	B	C	D
A	∞			6
B		∞		
C			∞	
D				∞

Per andare da A a D con 3 archi devo valutare $w(B, D)$ e $w(C, D)$ e valutare $\delta_2(A, B)$ e $\delta_2(A, C)$. Il primo termine ci restituirà infinito, il secondo vedendo della tabella ottenuta per $k = 2$ ci da 3. Quindi dobbiamo calcolare il minimo tra $\{10 + \infty, 3 + 3\} = 6$

Questa è una soluzione di programmazione dinamica (approccio Bottom-Up). Cerco di memorizzare soluzioni a sottoproblemi (soluzioni parziali) che posso riutilizzare. La struttura ottima deve contenere sottostrutture ottime, infatti solo in tal modo posso arrivare ad un ottimo globale per incrementi successivi. Queste tecniche permettono di arrivare ad un ottimo globale a scapito della complessità spaziale.

Soluzione 3 – Riduzioni

Ipotizziamo l'esempio precedente con $k = 3$. Posso srotolare il grafo e creare altri tre grafi più semplici da trattare (non ho archi che vanno indietro – sono senza cicli). Queste trasformazioni si chiamano anche riduzioni (**reduction**) e permettono di trasformare un grafo diretto ciclico in un DAG (grafo diretto aciclico). Posso poi applicare su questi grafi trasformati più semplici Dijkstra. Dijkstra ha complessità quadratica (in realtà possiamo avere varie complessità in base alle ipotesi di base). Dobbiamo applicare questa trasformazione perché Dijkstra non lavora con un fissato numero di archi, e in questo modo il nuovo grafo avrà certamente k archi fissati.

La complessità sarà $O(k^2n^2)$.

OSS: Ci domandiamo quando utilizziamo la soluzione 2 e quando la 3. Se $k = \theta(n^2)$ allora soluzione uno porta ad una complessità: $k = \theta(n^5)$ e soluzione due $k = \theta(n^6)$. È difficile che k sia così. Vanno quindi fatte delle osservazioni su k per valutare quale soluzione adoperare.

Backtracking

Il backtracking è un approccio sistematico per individuare tutte le possibili soluzioni presenti in uno spazio di ricerca. In generale tale spazio potrebbe essere dato da tutte le possibili permutazioni o da un sottoinsieme ottenuto aggiungendo dei vincoli. In generale tutti questi problemi hanno in comune il fatto che devono essere generate ogni possibile combinazione senza ripetizioni.

Ad esempio, se indago una soluzione e mi rendo conto che questo percorso per qualche motivo non può essere minimo o viola dei vincoli, allora taglio questo percorso e vado avanti (riduco lo spazio di ricerca). Quindi con questi approcci cerco di non esplorare soluzioni che non mi porteranno a niente. Questi approcci si usano per problemi combinatori in cui abbiamo dei vincoli che possiamo sfruttare.

Per comprenderlo al meglio vediamo degli esempi pratici. Inizialmente vedremo il backtracking per risolvere il gioco del Sudoku (che presenta vincoli), poi lo vedremo applicato a problemi più classici.

Un primo problema che dobbiamo porci è come modellare il problema. Supponiamo di modellare la soluzione al problema come un vettore $a = (a_1, a_2, \dots, a_n)$, dove ogni elemento a_i è selezionato da un set ordinato finito S_i . Ad esempio, tale vettore potrebbe rappresentare una disposizione dove a_i contiene l'i-esimo elemento di una delle possibili permutazioni; dove la permutazione è proprio S_i .

Dato un problema in cui dobbiamo trovare la soluzione ottima, potremmo pensare di effettuare un'operazione di **pruning**, ovvero una potatura dello spazio di ricerca e quindi ridurre lo spazio di ricerca. L'idea è quella di partire da una soluzione parziale e di estenderla incrementalmente. Partiamo da una soluzione parziale $a = (a_1, a_2, \dots, a_k)$ e la estendiamo aggiungendo un altro elemento da S . Se estendendo arriviamo ad una soluzione parziale lontana da quella completa allora effettuiamo un undo.

Il Backtracking costruisce l'albero delle soluzioni parziali, dove ogni nodo rappresenta una soluzione parziale.

Recursive Backtracking

N.B: il problema di questi algoritmi è la ricerca di un ottimo globale e non locale. In questo gioco posso ottenere la soluzione ottima.

```
Backtrack(a, k)
if a is a solution, print(a)
else {
    k = k + 1
    compute Sk
    while Sk ≠ ∅ do
        ak = an element in Sk
        Sk = Sk - ak
        Backtrack(a, k)
}
```

Se a è una soluzione allora processiamo a (in questo caso la stampiamo banalmente), altrimenti dobbiamo effettuare una ricorsione. Ciò che facciamo è incrementare k e scegliamo i possibili candidati (costruiamo S_k). Ad esempio, nel Sudoku, aggiungiamo in S_k un valore solo se questo non è già presente nel quadrato, nella riga o nella colonna corrispondente. Poi finché l'insieme non è vuoto, si preleva un valore da tale insieme (rimuovendolo) e si richiama ricorsivamente Backtrack. Questa è una ricerca in profondità nell'albero delle soluzioni.

OSS: La chiamata ricorsiva è detta anche **chiamata in coda** e questo è un vantaggio per gli algoritmi ricorsivi perché non devo salvare nulla sullo stack.

In conclusione, facciamo una ricerca esaustiva in cui non si visita un possibile stato (possibile soluzione) più di una volta. Diremo che il backtracking enumera tutte le possibili soluzioni evitando conflitti (soluzione che si ripete due volte).

Backtrack - Implementazione

```
void backtrack(int a[], int k, data input) {
    int c[MAXCANDIDATES];           /* candidates for next position */
    int nc;                         /* next position candidate count */
    int i;                           /* counter */

    if (is_a_solution(a, k, input)) {
        process_solution(a, k, input);
    } else {
        k = k + 1;
        construct_candidates(a, k, input, c, &nc);
        for (i = 0; i < nc; i++) {
            a[k] = c[i];
            make_move(a, k, input);
            backtrack(a, k, input);
            unmake_move(a, k, input);

            if (finished) {
                return;                  /* terminate early */
            }
        }
    }
}
```

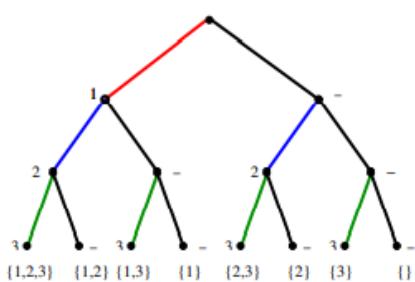
Utilizziamo le seguenti funzioni:

- **Is_a_solution(*a, k, input*):** è una funzione booleana verifica se i primi *k* elementi del vettore *A*, è una soluzione completa per il problema dato. L'ultimo argomento, *input*, ci permette di passare informazioni generali nella routine per valutare se *A* è una soluzione.
- **Construct_candidates(*a, k, input, c, nc*):** questa routine riempie l'array *c* con l'insieme completo di possibili candidati che il vettore *a* può assumere alla *k*-esima posizione, dato il contenuto prima di *k*. In *nc* è inserito il numero di candidati dell'array *c*.
- **Process_solution(*a, k*):** questa routine stampa, conta o in qualche modo elabora una soluzione completa.
- **Make_move(*a, k*) e Unmake_move(*a, k*):** queste routine possono modificare la soluzione *s* di conseguenza all'ultima mossa effettuata.

Costruzione di tutti i Sub-set

Dati n valori, vogliamo generare tutti i possibili sottoinsiemi. Ad esempio, dati i valori $\{1,2,3\}$ vogliamo generare l'insieme vuoto, l'insieme costituito solo da 1, solo da 2, solo da 3, da 1 e 2, da 1 e 3 e così via. In generale dati n elementi, avremo 2^n possibili sottoinsiemi; infatti, l'elemento nell'insieme può esserci o non esserci e quindi a due possibili stati. Per generare tali sottoinsieme dobbiamo in primo luogo rappresentare la soluzione. Possiamo impostare un array/vettore di n celle, dove il valore a_i è o vero o falso. Tale valore indica se l' i -esimo elemento è o non è nel sottoinsieme. Quindi implicitamente, la prima cella sarà il numero 1, la seconda il numero due e la terza il numero tre.

Questa cosa è ancora più evidente nel metodo `process_solution` dove si stampa proprio la posizione i . Per usare la notazione dell'algoritmo generale di backtrack, dobbiamo indicare con l'insieme dei possibili valori che come ben comprendiamo è: $S_k = \{\text{vero}, \text{falso}\}$, e v è una soluzione $\forall k \geq n$.



Se ad esempio vogliamo generare tutti sottoinsiemi di $\{1,2,3\}$, otteniamo l'albero a sinistra.

Vediamo come strutturare le funzioni del problema generale del backtracking.

Is_a_solution e Construct_candidate

```
int is_a_solution(int a[], int k, int n) {
    return (k == n);
}

void construct_candidates(int a[], int k, int n, int c[], int *nc) {
    c[0] = true;
    c[1] = false;
    *nc = 2;
}
```

Il vettore a è una soluzione del problema se la sua lunghezza k è proprio pari a n , ovvero tutti i valori dell'insieme di partenza. Quando invece dobbiamo costruire i candidati (ovvero l'insieme S_k) sarà sufficiente avere due valori vero o falso e quindi la dimensione di tale insieme (nc) è proprio 2.

Process_solution

```
void process_solution(int a[], int k, int input) {
    int i; /* counter */

    printf("{");
    for (i = 1; i <= k; i++) {
        if (a[i] == true) {
            printf(" %d", i);
        }
    }
    printf(" }\n");
}
```

Non fa altro che stampare l'elemento a_i se questo è vero, ovvero è presente nell'array della soluzione.

Generate_subsets

```
void generate_subsets(int n) {
    int a[NMAX];                                /* solution vector */

    backtrack(a, 0, n);
}
```

Questo non è altro che il metodo driver.

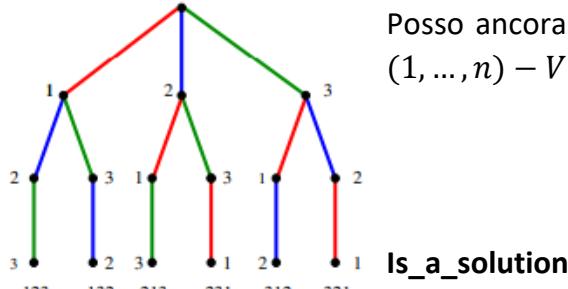
OSS: In questo caso non abbiamo bisogno di make e unmake move.

Costruzione delle Permutazioni

Vogliamo calcolare le possibili permutazioni di $\{1, 2, \dots, n\}$. In generale sappiamo che dato un insieme di n valori allora il numero di possibili permutazioni sarà $n!$.

Per rappresentare una soluzione possiamo usare un vettore di n celle, dove l' i -esimo elemento è uno dei possibili valori non presenti nelle celle precedenti del vettore.

Posso ancora una volta adottare un approccio incrementale. $S_k = (1, \dots, n) - V$ dove V è l'insieme dei valori già usati.



```
int is_a_solution(int a[], int k, int n) {
    return (k == n);
}
```

Ancora una volta il vettore a , di k valori è una soluzione se ha considerato tutti gli n valori e quindi $k == n$.

Construct_candidates

```
void construct_candidates(int a[], int k, int n, int c[], int *nc) {
    int i; /* counter */
    bool in_perm[NMAX]; /* what is now in the permutation? */

    for (i = 1; i < NMAX; i++) {
        in_perm[i] = false;
    }

    for (i = 1; i < k; i++) {
        in_perm[a[i]] = true;
    }

    *nc = 0;
    for (i = 1; i <= n; i++) {
        if (!in_perm[i]) {
            c[*nc] = i;
            *nc = *nc + 1;
        }
    }
}
```

Per costruire i candidati, in primo luogo creiamo la maschera `in_perm`, ovvero un vettore booleano dove il j -esimo elemento è `true` se $j = a[i]$. Praticamente se la soluzione $a = [1, 3]$ e quindi $k = 2$, allora vuol dire che `in_perm = [true, false, true]` infatti:

- Per $i = 1, a[1] = 1$ e quindi $c[1] = true$.
- Per $i = 2, a[2] = 3$ e quindi $c[3] = true$.
- I restanti saranno false, dato che la maschera ha tutti valori false all'inizio.

A questo punto possiamo costruire il vettore, inserendo i valori per cui $c[i]$ è `false`, ovvero per gli elementi ancora non presenti in a .

Process_solution

```

void process_solution(int a[], int k, int input) {
    int i; /* counter */

    for (i = 1; i <= k; i++) {
        printf(" %d", a[i]);
    }
    printf("\n");
}

```

Process solution stampa banalmente il vettore a .

Generate_permutations

```

void generate_permutations(int n) {
    int a[NMAX]; /* solution vector */

    backtrack(a, 0, n);
}

```

È il metodo driver che avvia la il processo di stampa delle permutazioni di un insieme di n elementi: $\{1, 2, \dots, n\}$.

Sudoku

OSS: Aggiungendo dei vincoli nella costruzione di S_k , siamo in grado di generare meno soluzioni e rendere più efficiente un algoritmo combinatorio. Nel Sudoku possiamo scegliere arbitrariamente il quadrato su cui lavorare (ad esempio possiamo selezionare il quadrato con più numeri - parliamo anche di scelta Most-Constrained e quindi più vincolante).

Dobbiamo ora costruire l'insieme dei possibili valori. Possiamo fare una ricerca locale (sfruttando i vincoli su riga, colonna, quadrato) o possiamo cercare un ottimo globale (guardiamo avanti: guardando gli altri quadrati). Possiamo costruire un albero per analizzare i valori dei vari quadrati (parliamo di **look ahead**). Nel grafico in basso possiamo vedere in base al tipo di selezione fatta, in base ai candidati considerati quante sono il numero di soluzioni individuate in base alla difficoltà del sudoku (vincoli iniziale del gioco).

Selezione	Candidati	Difficoltà		
		Easy	Medium	Hard
Arbitrario	local	1904832	865k	No Solution
Arbitrario	Look ahead	127	142	12M
Most Constrained	Local	48	84	1M
Most Constrained	Look ahead	48	65	10K

Backtracking

Certamente! Il backtracking è una tecnica di risoluzione dei problemi che si basa su una ricerca esaustiva all'interno di uno spazio di soluzioni possibili. Si utilizza spesso per risolvere problemi di combinazioni, permutazioni, puzzle, problemi di ottimizzazione e altro ancora.

In sostanza, il backtracking funziona esplorando tutte le possibili soluzioni per un problema in modo ricorsivo, ma interrompendo e tornando indietro ("backtracking") quando si verifica una condizione che rende impossibile trovare ulteriori soluzioni lungo quel percorso specifico. In questo modo, il backtracking risparmia tempo escludendo i rami dell'albero delle decisioni che non possono portare a una soluzione valida.

Ecco i passaggi fondamentali del backtracking:

Scelta di una soluzione parziale: Si inizia con una soluzione parziale e si tenta di estenderla fino a ottenere una soluzione completa.

Generazione delle possibili scelte: Si determinano tutte le possibili scelte valide per estendere la soluzione parziale.

Validazione della scelta: Si verifica se la scelta corrente è valida.

Applicazione della scelta: Se la scelta è valida, viene applicata alla soluzione parziale.

Ricorsione: Si prosegue in modo ricorsivo per la soluzione parziale estesa, tornando indietro in caso di fallimento.

Ritorno indietro (backtrack): Se nessuna delle scelte possibili porta a una soluzione valida, si ritorna al punto precedente e si esplorano altre opzioni.

Un esempio comune di problema risolto con il backtracking è il problema delle otto regine, in cui si deve posizionare un certo numero di regine su una scacchiera senza che si minaccino a vicenda.

È importante notare che l'efficienza del backtracking dipende dalla riduzione del numero di scelte e dalla prontezza nel riconoscere quando una soluzione parziale non può essere estesa in una soluzione completa. Ottimizzare l'algoritmo di backtracking può coinvolgere strategie come l'ordinamento delle scelte, l'uso di euristiche per la selezione delle opzioni e la gestione efficiente della memoria. La complessità del backtracking può variare ampiamente a seconda del problema specifico che viene risolto e della struttura dei dati coinvolta. La complessità dipende principalmente da due fattori:

Dimensione del problema: La dimensione del problema si riferisce al numero di decisioni da prendere o al numero di passi necessari per trovare la soluzione. Ad esempio, nel problema delle otto regine, la dimensione del problema è legata alla dimensione della scacchiera (8x8) e al numero di regine da posizionare.

Struttura dello spazio delle soluzioni: Questo è il numero di scelte possibili per ogni decisione e come queste scelte sono connesse tra loro. Se il numero di scelte possibili è elevato o se la struttura delle scelte è complessa, l'algoritmo di backtracking potrebbe richiedere più tempo per esplorare tutte le possibilità. La complessità del backtracking può essere valutata utilizzando la notazione O(), che indica il tempo (o lo spazio) necessario in funzione della dimensione del problema.

In generale, la complessità del backtracking è espressa come una funzione esponenziale di solito nella forma di $O(b^d)$, dove:

b rappresenta il fattore di ramificazione, ovvero il numero medio di scelte possibili per ogni decisione.
d è la profondità dell'albero delle decisioni, che rappresenta il numero di decisioni che devono essere prese per raggiungere una soluzione.

Quindi, se il problema ha una struttura che richiede l'esplorazione di tutte le possibili combinazioni senza alcuna riduzione delle scelte (ad esempio, nel caso peggiore), la complessità del backtracking può essere elevata.

Tuttavia, è importante notare che spesso ci sono modi per migliorare l'efficienza del backtracking attraverso tecniche come la potatura dell'albero delle decisioni (pruning), l'uso di euristiche per guidare la ricerca verso le soluzioni più promettenti e la riduzione delle scelte non promettenti. Queste ottimizzazioni possono ridurre significativamente la complessità del backtracking in certi casi.

CAPITOLO 4 - PROGRAMMAZIONE DINAMICA - DP

È una tecnica molto potente. Vediamo alcune possibili definizioni:

- Può essere definita come una ricerca brute-force intelligente (un po' come il backtracking) detta anche **brute-force controllato**. Possiamo pensare la soluzione del problema come la risoluzione di sotto-problemi più semplici che ci permettono di arrivare ad un ottimo globale. Diciamo che il problema presenta una **sotto-struttura ottima**.
- Può essere vista come una ricorsione a cui si aggiunge il riutilizzo di sotto-problemi. Non è sempre possibile usare questo approccio, ad esempio nel quick sort non si presenta mai lo stesso sotto-problema.
- Ricorsione + memoization (approccio Top-Down), e quindi di memorizzano i sottoproblemi.

Nonostante il nome programmazione dinamica, questo è un approccio di ricerca matematica.

La vedremo applicata a problemi veri.

Problema 1: Sequenza di Fibonacci L'EIGO |

La sequenza di Fibonacci si presenta come una sequenza del tipo 1,1,2,3,5,8,13, dove l'n-esimo elemento è calcolato come somma dei due elementi precedenti. Possiamo scrivere il problema come: $F_1 = F_2 = 1$ o $F_n = F_{n-1} + F_{n-2}$.

In generale vale la seguente espressione per n che tende ad infinito: $\frac{F_n}{F_{n-1}} = \varphi = \frac{1+\sqrt{5}}{2} = e$

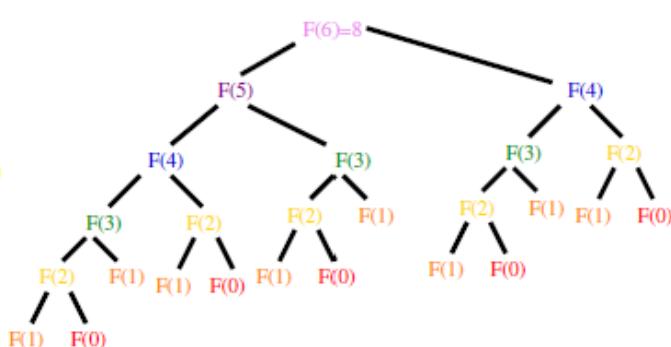
Algoritmo base

```
long fib_r(int n) {
    if (n == 0) {
        return(0);
    }
    if (n == 1) {
        return(1);
    }
    return(fib_r(n-1) + fib_r(n-2));
}
```

La complessità computazionale sarà:

$$T(n) = T(n-1) + T(n+1) + \theta(1)$$
$$\geq 2T(n-2) + \theta(1) = 2^{\frac{n}{2}}$$

Infatti, ogni nodo avrà due figli e avendo l'albero profondità $n/2$ la complessità sarà proprio esponenziale. Provando a disegnare l'albero per $n = 6$ otteniamo l'albero a sinistra. Possiamo osservare che tale approccio porta a calcolare ogni volta parte del



sotto problema. Ovvero il sotto-problema $F(4)$ verrà calcolato sia nel sottoalbero destro di $F(6)$ che nel sottoalbero sinistro di $F(5)$.

```

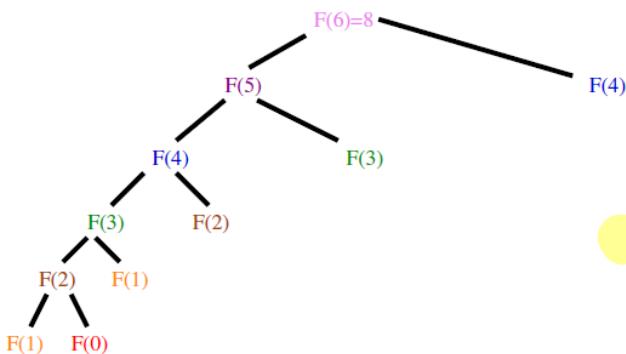
long memo[];
long fib(int n){
    long f;
    if(memo[n]) return memo[n];
    else if(n<=2) f=1;
    else f=fib(n-1)+fib(n-2);
    memo[n]=f;
    return f;
}

```

memoized) ricorsive.

N.B: non memorized ma proprio memoized.

Analisi di complessità:



Algoritmo DP – Top Down

Posso quindi inizializzare una struttura *memo* = {} in cui mi salvo i risultati parziali. Con tale struttura posso scrivere: ciò che faccio all'inizio di ogni iterazione è valutare se la *fib(n)* già è stato calcolato. Se non è stato calcolato, lo calcolo e lo inserisco in *memo[n]*. In questo modo il tempo calcolo diventa lineare dato che avrò solo *n* chiamate (che definiremo

Al netto della ricorsione ogni chiamata è costante, infatti, al più accederò ad un elemento del vettore. Ed è semplice osservare che è solo l'albero più a sinistra a scendere dato che quando andrò a generare *fib(n - 1)* andrò a calcolare inevitabilmente anche *fib(n - 2)* e quindi l'albero a sinistra. Da cui abbiamo una complessità lineare.

OSS: Per l'analisi di complessità posso andare a vedere come il tempo di:

1. Quanti sotto-problemi andrò a risolvere.
2. Quando costa ogni sotto-problema.

$$\text{tempo} = \# \text{sottoproblemi} * (\text{tempo}/\text{sottoproblema})$$

Algoritmo DP – Bottom Up

```

long fib(int n){
    long f[n];
    long temp;
    for(int k=0 ; k<n; k++){
        if(n<=2) temp=1;
        else temp=f[k-1]+f[k-2];
        f[k]=temp;
    }
    return f[n-1]
}

```

Questo è un primo approccio alla programmazione dinamica. Un altro possibile approccio è quello Bottom-Up.

Abbiamo anche in questo caso una complessità lineare. Il vantaggio è che è più chiara la scrittura (più intuitiva) dato che usa i cicli e non la ricorsione. La ricorsione potrebbe essere più pesante data le chiamate allo stack. In questo tipo di approccio non c'è però la chiara distinzione tra problema e sotto-problema.

In realtà un approccio più efficiente usa un array di dimensione due e una variabile di appoggio *f*. Questo perché è necessario salvare di volta in volta solo due elementi per calcolare il successivo.

OSS: Con questo approccio parto dal caso base per poi risalire fino alla soluzione.

Problema 2: Shortest Path

Lo scopo di tale problema è individuare il minimo percorso tra una sorgente s e un nodo v , per ogni nodo v diverso da s .

Algoritmo per DAG

Se indichiamo con V l'insieme dei nodi abbiamo che

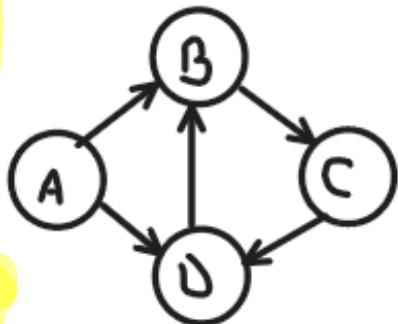
$$\delta(s, v) = \min\{\delta(s, u) + w(u, v) : w(u, v) \in E\}$$

dove $w(s, u)$ è il peso tra s e u ed E è l'insieme di archi (edge). Il caso base è quando $\delta(s, s) = 0$.

OSS: non considero $w(s, u) + \delta(u, v)$ altrimenti cambierei la sorgente ad ogni ricorsione.

Questo tipo di algoritmo funziona bene quando non ho loop e i costi degli archi sono tutti positivi.

Se ad esempio considero il grafo:



Nel grafo considerato ho però un ciclo $\{B, C, D\}$ che mi porta ad un loop infinito. Infatti, ponendo come nodo sorgente A e come nodo destinazione C , avrò le seguenti chiamate:

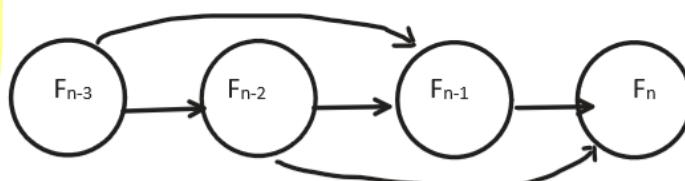
$$\{\delta(A, C), \delta(A, B), \delta(A, A), \delta(A, D), \delta(A, A), \delta(A, C)\}.$$

Il che mi comporta un ciclo.

Questo approccio funziona bene in un DAG in cui ho complessità data dalla somma degli $\text{indegree}(v)$ al variare di v a cui aggiungiamo un più uno (scelta conservativa) nel caso in cui il nodo v non ha indegree (infatti viene comunque fatto un controllo). Osserviamo che la somma di tutti gli indegree per tutti i nodi ci restituisce al più il numero di archi presenti nel grafo e quindi $|E|$. In conclusione, la complessità è: $\sum_{v \in V} [\text{indegree}(v) + 1] = |E| + |V|$.

E quindi abbiamo una complessità $\theta(|V| + |E|)$ dove $|E|$ è il numero di archi.

OSS: Il grafo dei sotto problemi deve essere aciclico. Non dobbiamo avere dipendenze mutue tra i vari sotto problemi.

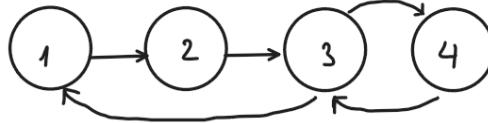


problema di ricerca di minimo percorso in un DAG dei sotto-problemi.

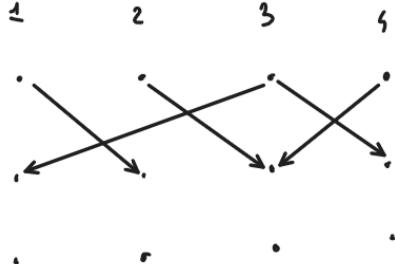
OSS: In questo caso il riuso sta nel fatto che posso riusare i minimi percorsi tra s e u , in tal modo posso rendere polinomiale e non esponenziale la ricerca.

OSS: Se ho cicli posso pensare di effettuare delle trasformazioni per rendere il grafo aliciclico. In questo modo crescono però i sotto problemi (dato che devo far variare k). Quindi posso usare l'algoritmo precedente aumentando però la complessità. Partendo da queste ipotesi posso effettuare le riduzioni e ottenere a partire da questo grafo:

Ad esempio, nel caso di Fibonacci questo avrebbe funzionato dato che non abbiamo cicli. Infatti, il grafo dei sotto-problemi è questo rappresentato a sinistra. In generale posso vedere un problema di DP come un problema di ricerca di minimo percorso in un DAG dei sotto-problemi.



Il seguente grafo:



Effettuando una la **reduction** ottengo un nuovo grafo senza cicli. Lo costruisco con k iterazioni e considererò $k \cdot |V|$ nodi. Come possiamo vedere alla prima iterazione, ovvero per $k = 1$, partendo da un nodo sorgente (del primo livello) dovrà arrivare al secondo livello con un solo arco (dato che $k = 1$). Quindi ad esempio con un solo arco il primo nodo può arrivare solo al secondo nodo. Invece il nodo 3, con un solo arco può andare o al nodo 4 o al nodo 1.

Bellman-Ford

Possiamo a questo punto declinare il problema nel seguente modo: denotiamo con $\delta_{\leq k}(s, v)$ il minimo percorso tra s e v con al più k edges, possiamo scrivere la ricorrenza come:

$$\delta_{\leq k}(s, v) = \min\{\delta_{\leq k-1}(s, u) + w(s, u) : (u, v) \in E\}$$

Il caso base sarà: $\delta_0(s, v) = \infty$ per $s \neq v$ e $\delta_k(s, s) = 0 \forall k$

Questo funziona in assenza di cicli negativi, ovvero cicli in cui ci sono archi con peso negativo. Tali archi infatti potrebbero portare a situazioni in cui i cicli caratterizzano un percorso che sarebbe più breve rispetto ad uno senza cicli (sono condizioni molto particolari).

Quindi con tale algoritmo stiamo ammettendo anche la presenza di archi a peso negativo. Per evitare la condizione precedente possiamo imporre $\delta(s, v) = \delta_{\leq |V|-1}(s, v)$. Ovvero il se impiegasse più della distanza con $|V| - 1$ archi vuol dire ho almeno un ciclo negativo. Con $\delta_{\leq |V|-1}(s, v)$, considero i percorsi semplici (simple path) – se non ho cicli negativi non entro in cicli.

Analisi di Complessità

Vediamo come impatta sulla complessità. Per calcolare il tempo uso la formula precedente quindi dobbiamo valutare il numero di sotto-problemi e la complessità per trattarli al netto della ricorsione.

Facendo un ipotesi di caso peggiore ottengo $O(|V|^3)$.

Devo considerare infatti, che per ogni vertice di destinazione (in totale $|V| - 1$ vertici) avremo $|V|$ possibili percorsi (ovvero per k che va da 0 a $|V| - 1$). Inoltre, ogni sotto-problema verrà trattato generalmente con complessità costante a cui si aggiunge però il costo per la ricerca del minimo. Dato che avremo $|V|$ percorsi la ricerca del minimo avrà proprio complessità $|V|$.

Questa è però un ipotesi molto irrealistica, infatti devo considerare in realtà solo gli $\text{indegree}(v)$ ovvero gli archi entranti (corrispondono ai nodi che hanno un arco verso il nodo v). Quindi avrò:

$$T(n) = O(|V|) \cdot \sum_{v \in V} \text{indegree}(v) = |V| \cdot |E|$$

Dove $|E|$ è il numero il numero di archi del grafo. Tale valore nel caso peggiore è pari a $|V|^2$ e quindi ci riportiamo al caso peggiore. Questo è conosciuto anche come **Algoritmo di Bellman-Ford**.

OSS: Per essere ancora più precisi quando ho 0 archi entranti, comunque devo fare un controllo quindi in realtà dovrò considerare $\text{indegree}(v) + 1$ da cui la complessità è: $\theta(|V||E| + |V|^2)$. Questa è però una scelta molto conservativa dato che il $+1$ lo considero anche quando ci sono i v e quindi i controlli che vengono fatti vengono considerati dal termine E .

Osserviamo inoltre che se fisso sia il nodo sorgente, che il nodo destinazione non avrà il fattore moltiplicativo $|V|$.

OSS: Questi tipologie di problemi sono esempi di DP dato che applico una forza bruta dato che provo tutti i percorsi possibili. È però controllato dato che non esploro tutto lo spazio di stato che altrimenti sarebbe esponenziale. Quando sto risolvendo un sotto-problema devo esplorare tutte le possibili scelte (forza bruta), ad esempio gli archi entranti nel nodo v (Guessing). Ricordiamo che nel calcolo del tempo valuto Tempo/Sottoproblema che è calcolato al netto della ricorsione, dato che poi memorizzo tali valori e li riutilizzerò quando si ripresenteranno. Quindi la prima volta devo valutare l'effettivo costo ma poi lo poiché dovrò riutilizzare i restanti sottoproblemi che si presenteranno avranno complessità costante.

Possiamo definire la programmazione dinamica come **Ricorsione + Memoization + Guessing** (individuare una serie di possibili scelte da esplorare).

Schema Programmazione Dinamica

Possiamo adottare uno schema di principio per lo sviluppo di una soluzione con programmazione dinamica (DP). Possiamo risolvere un problema di programmazione dinamica seguendo cinque step fondamentali:

1. **Definizione dei sotto-problemi:** Quando risolvo l' i -esimo sotto-problema, posso sfruttare questa per trovare l'ottimo globale. Vado a conservare una soluzione ottima parziale.
1.1 Definiti i sotto-problemi poi dovrò contarli. Quindi valutare: #sottoproblemi
2. **Guessing:** in Fibonacci non abbiamo scelte, però nello shortest-path devo individuare il minimo.
2.1 Devo contare le possibili scelte. Si devono valutare: #scelte
3. **Collegare i sotto-problemi:** ovvero devo comprendere il legame tra i sotto-problemi e quindi scrivere la ricorrenza.
3.1 In tale passaggio posso calcolare il tempo per sotto-problema (tempo/sottoproblema).
4. **Ricorsione + Memoization:** devo quindi scrivere l'algoritmo. Questo è l'approccio top-down. Possiamo adottare anche un approccio **bottom-up** e quindi devo andare a costruire la tabella. In questo passaggio devo quindi controllare che il grafo dei sotto-problemi sia aciclico (check del DAG).
4.1 Devo calcolare il tempo di esecuzione dell'algoritmo come:

$$T(n) = \text{sottoproblemi} \cdot \left(\frac{\text{tempo}}{\text{sottoproblema}} \right)$$

5. **Risolvere problema originario:** devo verificare che effettivamente sia risolto il problema principale. Potrei dover aggiungere qualcosa per far in modo che i sotto-problemi legati tra

di loro mi permettono di risolvere il problema originario. Non sempre possiamo applicare una semplice ricorsione su un sotto-problema per arrivare al problema originale.

5.1 Dobbiamo considerare in tal caso un extra-time.

Rivedendo tale approccio per i problemi già trattati abbiamo:

	Fibonacci	Shortest Paths
Sotto-problemi 1. #sotto-problemi	F_k con $k = 1, \dots, n$ 1. n	$\delta_k(s, v), \forall v \in V, \forall k \in \{0, \dots, V - 1\}$ 1. V^2
Guessing 1. #Possibili scelte	No 1. Costante $O(1)$	#archi_entranti_in_v 1. $\text{indegree}(v) + 1$
Ricorrenza 1. Tempo/sottoprob	$F_k = F_{k-1} + F_{k-2}$ 1. $\theta(1)$	$\delta_k(s, v) = \min\{\delta_{k-1}(s, u) + w(s, u) : (u, v) \in E\}$ 1. $\theta(\text{indegree}(v) + 1)$
Check DAG 1. Time	OK 1. $\theta(n)$	OK 1. $\theta(V E + V ^2)$
Problema Originario 1. Extra Time	F_n 1. $\theta(1)$	Abbiamo risolto $\delta_{ V -1}(s, v)$ 1. $\theta(V)$

OSS: Nello shortest path, costruita la ricorrenza, non conosciamo effettivamente il minimo percorso, ma il peso del minimo percorso. Per fare ciò dobbiamo ricordare ad ogni passaggio le scelte fatte e quindi gli archi considerati. Vedremo come soluzione i parent pointer.

Problema 3: Text Justification

Proprio come si fa in word, potremmo voler giustificare il testo e quindi fare in modo che una certa stringa ricopri l'intera linea.

Vogliamo passare ad esempio da:

Testo non giustificato	Testo giustificato
Aaaaaaaaaaaaaajjjjjjsssssssssfjnvj vkfs c dv vm c svckd f k fs cdvkjc f ckdj vdjdncnsddmc	Aaaaaaaaaaaaaajjjjjjsssssssssfjnvj vkfs c dv vm c svckd f k fs cdvkjc f ckdj vdjdncnsddmc

In generale abbiamo un problema di giustificazione quando abbiamo un numero di spazi maggiore del numero di parole in una linea. Le versioni precedenti di word, si usava un algoritmo Greedy che inseriva in ogni riga il maggior numero di parole possibili. In tal modo, si ha un ottimo locale per ogni riga e questo porterebbe alle ultime righe ad avere a disposizione un numero di parole minori per il quale risulta eccessivo il numero di spazi rispetto alle parole. Latex utilizza invece un algoritmo di DP. Data una lista di n parole ovvero $w[a:n]$, posso individuare una sotto-lista $w[i:j]$, ovvero una lista che va dalla stringa i fino alla stringa j . Su $w[i:j]$ posso definire lo **score di Badness** come:

$$Badness(i,j) = \begin{cases} \infty, & \text{se } Length > PageWidth \\ PageWidth - Length, & \text{altrimenti} \end{cases}$$

OSS: Length non è il numero di stringhe ma il numero totale di caratteri delle stringhe nella lista. Latex addirittura considera $(PageWidth - Length)^3$; ovviamente è una scelta empirica utilizzata per penalizzare le linee in cui ci sono troppi spazi. Quanto è più grande tale valore quanti più spazi avrò nella linea.

Obiettivo: suddividere le parole in linee in modo tale da minimizzare la somma di tutte le Badness (ovvero minimizzo il numero di spazi in ogni riga).

N.B: poter scrivere il problema come una sequenza è un ottimo punto di partenza, infatti posso vedere un sotto-problema banalmente come una sotto-sequenza della sequenza di partenza.

Questo è un problema di partizionamento, ovvero dato un insieme di n elementi, vogliamo partizionare gli n elementi in sottoinsiemi. In maniera esaustiva potrei partizionare le parole in modo tale se la parola è una parola di inizio linea. Le possibili configurazioni saranno 2^n , infatti sto dando la possibilità ad ogni parola di essere la prima parola di inizio linea.

N.B: Posso usare un vettore booleano di n bit, dove i-esimo bit è *true* se la i-esima parola è una parola di inizio linea, *false* altrimenti.

Se vogliamo usare un approccio di DP, dobbiamo individuare dei sotto-problemi che potrei utilizzare per comporre la soluzione ottima. Il mio algoritmo deve minimizzare la Badness e quindi è un problema di ricerca di minimo. Il caso base è quando nell'insieme delle parole che posso utilizzare ho una sola parola e quindi termina la ricorrenza (ovvero l'ultima parola).

Devo definire un sotto-problema che sia più piccolo del problema originale. Sfruttando un approccio top-down considero come sotto-problema il calcolo della Badness per $w[i:] = w[i: n]$ ovvero della Badness da i in poi; a cui devo sommare la Badness dei precedenti. Per la trattazione dell'algoritmo distinguiamo il **prefisso** ovvero l'insieme delle parole inserite e il **suffisso** ovvero le parole restanti.

Vediamo nel dettaglio come utilizzare l'approccio DP in tale problema:

1. Definizione del sotto-problema:

Denotiamo con $DP(i)$ la Badness per il suffisso $w[i:]$. Il caso base è quando sto considerando $DP(n) = 0$.

1.1 In tal modo avrò un numero di sotto-problemi pari al numeri di parole:
 $\#sottoproblemi = O(n)$.

2. Guessing:

Corrisponde a dove iniziare la prossima linea. Quindi dato il sotto-problema $DP(i)$, da che parola in poi $j > i$ dovrò spezzare la linea e andare a capo. Devo individuare il numero di split possibili:

2.1 $\#split = n - i = O(n)$

3. Ricorrenza:

$$DP(i) = \min_{j \in \{i, \dots, n\}} \{Badness(i, j) + DP(j + 1)\}$$

Se sto alla quarta parola e considero come split la quinta parola, metto una parola in questa linea vado a capo e calcolo il sotto-problema $DP(j)$. Poi torno indietro e calcolo la Badness su due parole nella linea, vado a capo e calcolo il sotto-problema da $(i + 2)$ e così via.

Quindi quando calcolo $Badness(i, j)$ ipotizzo di inserire le parole da i a j , ne calcolo la Badness e calcolo la linea successiva. Il caso base lo ho quando valuto $DP(n + 1) = 0$. Il tempo per sotto-problema sarà proprio il tempo per calcolare il minimo tra le j possibilità.

3.1 $TempoPerSottoproblema = \theta(n)$.

4. Check DAG

Il grafo non presenta cicli dato che consideriamo sempre una lista di stringhe che esclude la prima stringa nella lista.

4.1 Tempo Totale = $\theta(n^2)$.

5. Soluzione Finale

Non abbiamo un extra-time la ricorrenza ci dà proprio il testo giustificato.

Cerchiamo di comprendere dove risiede la Memoization. Supponiamo di avere cinque stringhe:

Word Index	Word	Number of char
0	Tushas	6
1	Ray	3
2	like	5
3	To	2
4	Code	4

	0	1	2	3	4
0	9	0	INF	INF	INF
1		343	1	INF	INF
2			125	8	INF
3				512	27
4					216

Ipotizziamo di avere $width = 10$.

Partendo dalla prima riga il ragionamento è il seguente: parto dalla parola zero e ne calcolo la Badness aggiungendo di volta in volta una nuova parola. Quindi la prima volta ipotizzo di inserire una sola parola nella riga ottenendo $Badness(0,0) = (10 - 6)^3$, poi passo alla seconda riga dove avrò $Badness(0,1) = (10 - 6 - 3 - 1)^3 = 0$, i restanti termini saranno *INF* ovvero infinito. Passiamo ora alla seconda riga, ovvero ipotizziamo di far partire una nuova riga a partire dalla seconda parola e otteniamo: $Badness(1,1) = (10 - 3)^3$ e così via. Iteriamo questo approccio e costruiamo la matrice a sinistra. A partire da questa matrice possiamo trovare l'ottimo globale.

Problema 4: Black Jack

Analizziamo alcune informazioni sul gioco:

- Abbiamo un singolo generatore
- Ho un mazzo di n carte da $\{c_0, c_1, \dots, c_{n-1}\}$
- Ogni partita costa \$1.
- Il mazziere sta bene con un valore maggiore o uguale di 17 (stand-on 17).
- Perfect Information Blackjack: abbiamo una conoscenza perfetta delle carte che escono nel mazzo.
- L'obiettivo è massimizzare il guadagno fino al termine della partita. Il guadagno parte da zero, si somma 1 se si vince, -1 se si perde.

Step DP:

1. Estratta una carta massimizziamo il guadagno con le restanti $\{c_i, c_{i+1}, \dots, c_{n-1}\}$. Tale valore è denotato con $BJ[i]$.
Il numero di sottoproblemi = n .
2. Il numero di scelte può consistere nel mantenere le proprie carte o pescare una nuova carta.
In generale il numero di scelte è pari al numero di carte pescate $\#hits$. In generale tale valore è minore di n (non capiterà mai il caso $\#hits = n$).
3. Devo ora scrivere la ricorrenza

$$BJ(i) = \max\{outcome + BJ(i + \#carte_usate)\}$$

Outcome è l'esito e può essere $\{-1, 0 - 1\}$. Inoltre, $BJ(i + \#carte_usate)$ va fatta per ogni hits e assumendo che la partita sia ancora valida. Tale condizione è data da $\sum carte_usate < 21$.

4. In questo caso il tempo è dato da:

$$T(n) = \sum_{i=0}^{n-1} \sum_{\#hits=0}^{n-i-O(1)} \theta(n - i - \#hits) = O(n^3)$$

Il passo base è $BJ(0)$

5. Non abbiamo un exceeding time.

```
BJ(i):
    if(n-i)<4: return 0; //non ho più carte
    for(p in range (2,n-i-1)):
        player = sum(Ci,Ci+2, {Ci+4:Ci+p+2}); //do prima una carta a me poi se la prende il mazziere
        if(player>21) sbalallo;
            option.append(-1+BJ(i+p+2));
            break;
        for(d in range (2,n-i-p)):
            dealer = sum(Ci+1,Ci+3, {Ci+p+2:Ci+p+d});
            if(dealer >= 17 ) break;
            if(dealer>21) dealer=0;
            option.append(comp(player,dealer)+BJ(i+p+d));

    return max(option);
```

Sopra è riportato lo pseudo-codice per il calcolo di $BJ(i)$.

Parent Pointers

Quando ho trovato il minino o il massimo nel sottoproblema i -esimo, dovrò salvarmi tale valore per poi costruire la soluzione ottima. Per fare ciò uso una struttura detta **parent pointer**.

Facciamo riferimento al problema della text justification, ogni qual volta trovo il minimo calcolando $DP(i)$, devo capire qual è la stringa che effettivamente mi porta a minimizzare la Badness al passo i -esimo. Posso quindi fare in modo che quando calcolo il minimo, allora mi salvo l'indice $j + 1$ che mi porta a minimizzare il calcolo. E questo lo farà ad ogni qual volta sto risolvendo il problema i -esimo. Dall'esempio trattato la struttura di parent pointer sarà un vettore di n elementi (numero di stringhe) di questo tipo: $\{1,3,3,5,5\}$. Questa soluzione ci dice che alla prima riga avrò le stringhe $w[0:0]$ successivamente dovrò andare a capo e troverò le parole della seconda riga andando proprio alla posizione 1 dell'array (quindi la seconda cella). Quindi alla seconda riga stamperò le stringhe $w[1:2]$. Alla terza riga stamperò le stringhe $w[3:4]$.

Problema 5: Parentesizzazione

Data una catena di n matrici $\{A_1, \dots, A_n\}$ vogliamo calcolare il prodotto: $A_1 \cdot A_2 \cdots A_n$. Potremmo pensare di utilizzare un algoritmo classico per effettuare il prodotto tra matrici partendo dalle matrici più a sinistra. In realtà se consideriamo il prodotto $A_1 \cdot A_2 \cdot A_3 \cdot A_4$, possiamo strutturare tale prodotto in cinque approcci distinti:

- $(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4)))$
- $(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4))$
- $((A_1 \cdot A_2) \cdot (A_3 \cdot A_4))$
- $((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4)$
- $((((A_1 \cdot A_2) \cdot A_3) \cdot A_4)$

Il modo in cui parentesizziamo una sequenza di matrici può avere un impatto notevole sul costo totale per calcolare il prodotto delle matrici. L'idea è minimizzare il numero di operazioni scalari.

Prodotto di matrici

```
MATRIX-MULTIPLY( $A, B$ )
1 if  $\text{columns}[A] \neq \text{rows}[B]$ 
2   then error "dimensioni non compatibili"
3   else for  $i \leftarrow 1$  to  $\text{rows}[A]$ 
4     do for  $j \leftarrow 1$  to  $\text{columns}[B]$ 
5       do  $C[i, j] \leftarrow 0$ 
6       for  $k \leftarrow 1$  to  $\text{columns}[A]$ 
7         do  $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$ 
8   return  $C$ 
```

i -esima sarà ottenuta moltiplicando la riga i -esima della prima matrice di volta in volta per ogni colonna della seconda matrice. In generale possiamo osservare che la complessità è cubica ovvero avremo una complessità $O(n \cdot q \cdot m) = O(n^3)$. Cerchiamo ora di comprendere come la parentesizzazione può influenzare la complessità. Supponiamo di avere tre matrici: A di dimensione 100×1 , B di dimensione 1×100 e C di dimensione 100×1 . Consideriamo la complessità di:

1. $((A \times B) \times C)$: $A \times B$ porta ad una complessità $100 \times 100 \times 1 = 10.000$; moltiplicando poi per C dobbiamo sommare ancora la stessa quantità e quindi avremo una complessità temporale 20.000. In realtà questo approccio influenza notevolmente anche la complessità spaziale; infatti, dalla primo prodotto abbiamo una matrice di dimensione 100×100 (occupiamo 10.000 unità di memoria) e successivamente col prodotto $(A \times B) \times C$ otteniamo una matrice di dimensione 100×1 (occupiamo altre 100 unità di memoria). In conclusione, abbiamo una complessità temporale di 20.000 e una spaziale di 10.100
2. $(A \times (B \times C))$: in questo caso avremo $1 \times 1 \times 100 + 100 \times 1 \times 1 = 200$. Ragionando in maniera analoga avremo una complessità temporale di 200 e una spaziale di 101.

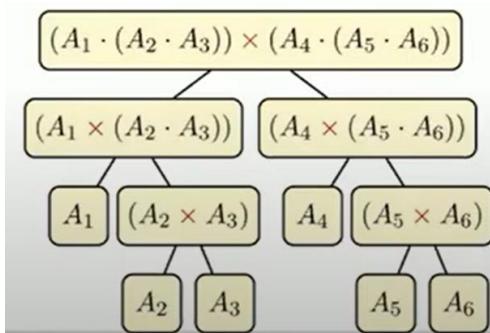
In primo luogo, vediamo se le due matrici sono compatibili, ovvero date due matrici $n \times m$ e $p \times q$ dobbiamo controllare se $m = p$ e la matrice risultante sarà una matrice $n \times q$. Quindi possiamo pensare di riempire la matrice per righe: in particolare la riga

Possiamo quindi comprendere che l'ordine con cui si effettuano i prodotti influenza notevolmente la complessità temporale e spaziale.

Possiamo formulare il problema nella seguente maniera: data una sequenza di n matrici $\{A_1, A_2, \dots, A_n\}$ dove A_i ha per dimensione $p_{i-1} \times p_i$ per $i = 1, 2, \dots, n$, vogliamo determinare lo schema di parentesizzazione che minimizza il numero di prodotti scalari (**parentesizzazione ottima**). Osserviamo che lo scopo è trovare lo schema di parentesizzazione ottimo e non il prodotto; quindi, è una fase di preprocessing dei dati che ha senso fare in quanto consentirà successivamente di ottimizzare il tempo di calcolo dei prodotti.

Se indichiamo con $P(n)$ il numero di parentesizzazioni alternative di una sequenza di n matrici, possiamo osservare che:

- Quando $n = 1$, c'è una sola matrice e quindi una solo schema.
- Quando $n \geq 2$, un prodotto di matrici completamente parentesizzato è il prodotto di due sottoprodotti di matrici completamente parentesizzati e la suddivisione fra i due sottoprodotti può avvenire fra la k -esima matrice e la $(k + 1)$ -esima matrice per qualsiasi $k = 1, 2, \dots, n - 1$. Come possiamo notare dall'esempio possiamo considerare il numero di parentesizzazioni alternative, fissando un k e suddividendo il problema in due sottoproblemi che trattano due gruppi di matrici da 1 a k e da $k + 1$ a n .



Se quindi abbiamo $n = 6$ matrici e fissiamo $k = 3$ allora avremo due gruppi da tre matrici come in figura.

Quindi, otteniamo la ricorrenza:

$$P(n) = \begin{cases} 1, & \text{se } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k), & \text{se } n \geq 2 \end{cases}$$

Si può dimostrare che la complessità è pari al **numero di catalan**; ovvero $P(n) = C(n) = \frac{1}{n+1} \binom{2n}{n} = \theta\left(\frac{4^n}{n^2}\right)$ e sfruttando la **sequenza dei numeri catalani** otteniamo che il numero di possibili parentesizzazioni cresce in modo esponenziale; infatti, è molto semplice dimostrare che $C(n) = \Omega(2^n)$. Comprendiamo che esplorare tutte le parentesizzazioni non è un approccio possibile in quanto computazionalmente intrattabile.

Problema: vogliamo ottimizzare il prodotto $A_1 \cdot A_2 \cdots A_n$ in particolare denotiamo con c_{i-1} il numero di righe della matrice A_{i-1} e con c_i il numero di colonne della matrice A_i . Con $A[i \dots j]$ denotiamo il sottoprodotto della matrici da i a j e con $P[i \dots j]$ una parentesizzazione per $A[i \dots j]$ (non necessariamente ottima).

1. **Sottoproblema:** consideriamo il sottoproblema data dal sottoprodotto $A[i \dots j]$. Poiché inizialmente tale insieme è dato dalle n matrici. Su tale insieme vado a calcolare $DP(i, j)$ che è il minimo numero di moltiplicazioni scalari per calcolare il prodotto sull'insieme. Partendo dall'i-esimo elemento, confronterò questo con i restanti $n - 1$ elementi.

$$\#\text{numero_sottoproblemi} = n^2$$

2. Ad ogni iterazione $DP(i, j)$ è dato dal minimo di tutti gli elementi sulla riga e quindi il minimo tra n elementi. E quindi $\#\text{scelte} = O(n)$.

3. **Ricorrenza:**

$$DP(i, j) = \begin{cases} 0, & \text{se } i = j \\ \min_{i \leq k < j} \{DP(i, k) + DP(k + 1, j) + c_{i-1} \cdot c_k \cdot c_j\}, & x \geq 0 \end{cases}$$

Infatti, se $i = j$ stiamo considerando la parentesizzazione fatta da una sola matrice (non ci sono prodotti); altrimenti consideriamo le parentesizzazioni ottime che si ottengono separando l'insieme (fissando k). I due nuovi insiemi restituiranno due matrici ottenute moltiplicando $A[i \dots k]$ e $A[k + 1 \dots j]$.

Il risultato del primo prodotto sarà una matrice di $c_{i-1} \times c_k$ ovvero una matrice che il numero di righe della matrice A_i e il numero di colonne della matrice A_k . Analogamente il prodotto del secondo insieme restituirà una matrice di dimensione $c_k \times c_j$. Per ottenere quindi il numero di prodotti scalari per ottenere $DP(i, j)$, non è sufficiente sommare il numero di operazioni per ottenere $DP(i, k)$ e $DP(k + 1, j)$ ma dobbiamo considerare anche il numero di operazioni per calcolare il prodotto delle matrici risultanti dai due sottoinsiemi. Il costo per trattare il sottoproblema al netto della ricorsione è quindi **costoSottoproblema** = $O(j - i) = O(n)$, ovvero la complessità per calcolare il minimo numero di operazioni al variare di k .

4. **Ricorsione + Memoization:** non abbiamo cicli e quindi:

$$T(n) = O(n^2) \cdot O(n) = O(n^3).$$

5. Non abbiamo bisogno di un extra-time per calcolare la soluzione finale.

Problema 6: Edit Distance

Determinare il costo minimo per trasformare una stringa in un'altra. Il modello di costo dipende dalle trasformazioni permesse (le trasformazioni possono avere costi diversi), ad esempio: inserimento nuovo carattere, cancellazione di un carattere ecc...

Possiamo avere diverse applicazioni quali:

- Confronto di due stringhe (stimare quanto sono diverse tra di loro).
- Confronto di sequenze di DNA in cui si confrontano le mutazioni delle basi (la più importante).
- Correzione automatica del testo – controllo ortografico.
- Antiplagio – rilevamento del plagio.

Il calcolo della similarità tra due elementi è paragonabile ad un calcolo di distanza con opportuna metrica.

Obiettivo: date due stringhe x e y , determinare la sequenza di operazioni di **edit** per trasformare x in y .

Modello di costo: dobbiamo ora introdurre un modello di costo per le operazioni di edit; ad esempio, nel confronto di DNA, un trasformazione $C \rightarrow G$ è molto frequente e quindi associamo un basso costo. Il costo totale sarà dato dalla somma dei singoli costi legati alla sequenza di operazioni di edit.

Date le stringhe (non necessariamente di uguale lunghezza):

- $X[1, \dots, m]$
- $Y[1, \dots, n]$
- Z – array per salvare i risultati intermedi. La dimensione deve essere sufficientemente grande per contenere tutte le operazioni possibili di trasformazione. L'array è inizialmente vuoto e al termine dovrà essere uguale a Y ; ovvero: $Z[j] = Y[j]: \forall j = 1, \dots, n$. Abbiamo inoltre un indice i che parte da 1 e alla fine deve essere uguale a $m + 1$ (serve a scorrere X).

Supponiamo di avere a disposizione sei operazioni:

1. **COPIA (C1):** copia un carattere c da X a $Y \Rightarrow Z[j] = X[i]; i + 1; j + 1;$
2. **SOSTITUZIONE (C2):** sostituisce un carattere c che appartiene a X con c'
 $\Rightarrow Z[j] = c'; i + 1; j + 1;$
3. **CANCELLAZIONE (C3):** cancella un carattere di c di X ($c \in X \Rightarrow i + 1$; (non viene modificato Z con quest'operazione))
4. **INSERIMENTO (C4):** inserisce un carattere c in $Z \Rightarrow Z[j] = c, i + 1;$ (*non scorriamo X*).
5. **SCAMBIO (C5):** scambia $Z[j] = X[i + 1], Z[j + 1] = X[i] \Rightarrow i += 2; e j += 2;$

quest'operazione differisce dalla sostituzione perché caratterizza da non costo diverso.

DISTRUGG (C6): distrugge la parte restante di $x \Rightarrow i = m + 1$.

Non consociamo il costo, supponiamo però che sia una costante nota. COPIA e SOSTITUZIONE potrebbero essere realizzati mediante operazioni successive di cancellazione e inserimento; quindi ipotizziamo che: $C1 \text{ e } C2 < C3 + C4$.

Vediamo un esempio:

- X="algorithm"
- Y="altruistic"

Costruiamo Z

OP	X	Z	Y
	algorithm		Altruistic
COPIA	algorithm	a	Altruistic
COPIA	Algorithm	al	Altruistic
SOSTITUZIONE	algorithm	Alt	Altruistic
CANCELLA	algorithm	Alt	Altruistic
COPIA	algorithm	Altr	Altruistic
INSERIMENTO 'u'	algorithm	Altru	Altruistic
INSERIMENTO 'i'	Algorithm	altrui	
INSERIMENTO 's'	algorithm	Altruis	
SCAMBIO	algorithm	Altruisti	
INSERISCI 'c'	algorithm	Altruistic	
DISTRUGGI	Algorithm_	Altruistic	

Questo è un esempio di funzionamento, che non necessariamente porta ad una soluzione ottima.

$$Costo = 3 \cdot C1 + C2 + C3 + 4 \cdot C4 + C5 + C6$$

Facciamo un algoritmo di programmazione dinamica che effettua il numero di operazioni ottimale.

1. Sottoproblema: combinazioni di suffissi/prefissi/sottostringhe

$$DP(i, j) \xrightarrow{eq} editDistance(X[i:], Y[j:]) \text{ dove: } 0 < i < |X| = m \text{ e } 0 < j < |Y| = n \\ \#sottoproblemi = O(|X||Y|).$$

2. Scelte: sono le operazioni di editing che posso fare.

3. Ricorrenza

$DP(i, j) = \min_k \{C_k + DP|k\}$ dove $DP|k$, vuol dire DP condizionato da k e quindi dal sottoproblema che si genera in base all'operazione scelta. Ad esempio se scelgo l'operazione di COPIA allora avrò: $DP(i, j) = \min_k \{C1 + DP(i+1, j+1)\}$ e quindi più in generale abbiamo:

$$DP(i, j) = \min_k \{C_k + DP|k\} = \min \{C1 + DP(i+1, j+1); C2 + DP(i+1, j+1); \\ C3 + DP(i+1, j); C4 + DP(i, j+1); C5 + DP(i+2, j+2), C6 + DP(m+1, j)\}$$

Un vincolo potrebbe essere che quando abbiamo $m+1$, allora torniamo al caso base; infatti se svolgo quest'operazione ho finito dato che $Z = Y$.

$$\frac{\text{Tempo}}{\text{sottoproblema}} = \theta(1)$$

- Infatti, è costante ed è pari al numeri di operazioni di editing (6).
4. Data un posizione (i, j) che immaginiamo come elementi di una matrice, possiamo al più scegliere l'elemento a destro $(i, j + 1)$, l'elemento al di sotto $(i + 1, j)$ o l'elemento successivo sulla diagonale $(i + 1, j + 1)$. Deduciamo che non abbiamo cicli.
- $$TempoTotale = \theta(|X||Y|)$$
5. Per sapere la sequenza migliore devo raccogliere l'argomento che mi restituisce il minimo e vanno quindi usati i Parent Pointer.

Problema 7: Problema dello Zaino (Knapsack Problem)

Abbiamo in tale problema un knapsack di dimensione fissata S e un insieme di oggetti di dimensione s_i (interi) e valore v_i (reali). L'obbiettivo è scegliere un sottoinsieme di oggetti tali da massimizzare la somma dei valori degli oggetti inseriti nello zaino rispettando il vincolo che la somma delle dimensioni di tali oggetti sia minore o uguale della capienza S . Quindi:

- Massimizziamo $\sum v_i$.
- $\sum s_i \leq S$.
- $V[:]$ – vettore degli oggetti disponibili.

Immaginiamo di voler utilizzare l'approccio dove si valuta la densità $d_i = v_i/s_i$ e si schedula prima quelli a densità maggiore. Possono ad esempio considerare tre oggetti A, B, C:

	s_i	v_i
A	10	20
B	10	20
C	50	50

E ipotizziamo $S = 50$. Con tali dati saremmo indotti a inserire nello zaino A e B.
Ma la scelta ottima sarebbe data dall'inserimento di C.

Per effettuare la scelta ottima possiamo usare un approccio di programmazione dinamica. Vediamo quindi i vari step:

1. Definizione dei sottoproblemi:

Possiamo considerare un sotto-vettore che rappresenta la scelta di prendere o meno un elemento in $V[:]$. In tal caso sotto-problema posso definire come un suffisso $V[i:]$ a cui devo aggiungere la nuova capacità dello zaino (che denoto con X). In tali condizioni il caso peggiore prevede: #sottoproblemi = $n \cdot S$.

2. Possibili scelte:

La scelta consiste nel prendere o meno l'elemento i -esimo. #scelte = 2

3. Ricorrenza:

$$DP(i, X) = \max \{v_i + DP(i + 1, X - s_i) \text{ se } s_i < X, DP(i + 1, X)\}$$

Quindi la ricorrenza è strutturata in due parti: il caso in cui inseriamo l'elemento e quindi sommiamo al valore dell'oggetto inserito il sotto-problema sul suffisso; altrimenti ho solo il sotto-problema dato dal suffisso (ovvero non inseriamo l'elemento). In tal modo non consideriamo il vincolo sullo stato dello zaino e quindi non riesco a capire se con l'elemento inserito sfido la capienza dello zaino. Per aggiungere l'informazione sulla capienza dello zaino (e quindi sullo stato) possiamo aggiungere una variabile X che rappresenta la situazione attuale. Il caso base è quando $DP(i, X) = 0$ per $i = n$. Il tempo per sottoproblema è quindi $O(1)$.

4. Ricorsione+memoization

Possiamo considerare il $TempoTotale = O(n \cdot S)$

5. Problema Originario:

Abbiamo bisogno del Parent Pointer per recuperare la soluzione.

Il tempo di esecuzione è quindi $O(n \cdot S)$. Possiamo quindi dire che la complessità dipende dalla grandezza effettiva di S , rispetto a n . Questa complessità può essere considerata polinomiale nella dimensione dell'input ovvero $\theta(n)$ se il valore di S entra in una word (è sufficientemente piccolo). Per esprimere S , mi servono $\lg S$ bit. La dimensione in bit influenza la complessità delle operazioni elementari che si vanno a realizzare su S .

Il problema è che la complessità per trattare S è esponenziale (dipende dalla sua rappresentazione) e quindi la complessità è esponenziale in $\lg S$. In realtà non è propriamente esponenziale dato che dipende dalla rappresentazione di S . Parliamo quindi di **complessità pseudo-polinomiale** ovvero polinomiale della dimensione dell'input del problema ma non nel valore (ovvero S). In generale la classificazione può essere:

$$\text{Polinomiale} < \text{Pseudo - Polinomiale} < \text{Esponenziale}$$

Se S è molto grande il numero di operazioni elementari su S , cresce in modo esponenziale.

Problema 8: Longest Common Subsequence

Date le due stringhe "AXYT" e "AYZX" la sottosequenza comune di dimensione maggiore può essere "AX" e "AY" (devono essere trovate secondo un certo ordine). Supponiamo di avere due stringhe di dimensione n e m , se adottiamo un approccio bruteforce le possibili combinazioni della prima stringa sono 2^n . Dobbiamo ora confrontare queste combinazioni con la seconda stringa e quindi abbiamo una complessità $O(m \cdot 2^n)$.

1. **Sottoproblema:** Consideriamo sottostringhe di lunghezza minore e quindi consideriamo il suffisso delle stringhe. Possiamo quindi identificare il sottoproblema $DP(i, j)$ che valuta il numero di caratteri in comune tra $str1[i:]$ e $str2[j:]$. Il numero di sottoproblemi sarà $O(nm)$.
2. Il numero di scelte che possiamo fare è costante.
3. La ricorrenza sarà:

$$DP(i, j) = \begin{cases} 1 + DP(i + 1, j + 1), & \text{se } str1[i] = str2[j] \\ \max\{DP(i + 1, j), DP(i, j + 1)\}, & \text{altimenti} \end{cases}$$

Il caso base lo si ha quando si sfiora una delle due stringhe è terminate e quindi $DP(i, j) = 0$ se $i = n$ o $j = n$, ovviamente in questo caso si chiamerà $DP(0,0)$. Possiamo quindi osservare che l'idea alla base è abbastanza semplice partendo dai primi due caratteri, confrontiamo questi, se sono uguali allora è l'inizio di una sottosequenza che sicuro avrà almeno lunghezza 1, e quindi possiamo cercare il resto della sequenza comune chiamando ricorsivamente il metodo sui suffissi $str1(i + 1:)$ e $str2(j + 1:)$. Altrimenti cerchiamo la sottosequenza comune considerando $str1(i:)$ e $str2(j + 1:)$ o $str1(i + 1:)$ e $str2(j:)$. Valuteremo il massimo tra le due sottosequenze. Il tempo per sottoproblema è costante dato che al più valutiamo il massimo tra due valori.

4. **Ricorsione+memoization**
Possiamo considerare il $TempoTotale = O(n \cdot m)$.
5. Non abbiamo extra time.

Problema 9: Cutting a Rod

Data una panca di n unità, e dato il vettore v che rappresenta effettuando un taglio di grandezza i quanto possiamo ricavare dalla pezzo di panca ottenuto ($v[i]$). Vogliamo trovare il modo migliore per tagliare una panca e quindi in modo da massimizzare il guadagno. L'idea è abbastanza semplice suppongo di effettuare un taglio di un'unità, sulla panca restante proverò ad effettuare altri tagli cercando di massimizzare il numero di tagli sulla panca rimanente. Il problema è spiegato nel file.cpp in AlgoritmiDP.

Problema 10: Piano

Il guessing finora lo abbiamo visto come un caso standard in cui dobbiamo scegliere tra i sottoproblemi definiti quali mi servono per risolvere il sottoproblema trattato. Una situazione più complessa nel problema dello zaino, dove abbiamo aumentato la complessità del problema per aggiungere un'informazione sullo stato dello zaino (spazio rimanente nello zaino). Vediamo altri problemi dove dobbiamo aumentare il numero di sottoproblemi per tenere traccia delle informazioni necessarie a caratterizzare lo stato del problema.

Il problema è il seguente:

- Abbiamo un brano musicale, visto come sequenza di note “singole”.
- Immaginiamo di usare una sola mano (la destra) e quindi possiamo usare da $f = 1$ a $f = 5$ dita. Osserviamo che $f = 1$ è il pollice. Possiamo indicare con F l'insieme delle dita e quindi la cardinalità di F con una mano sarà: $|F| = 5$.
- Introduciamo una metrica di difficoltà $d(p, f, q, g)$: $f, g \in F, p, q \in \text{note}$. Abbiamo un valore altro di difficoltà quando:
 1. $1 < f < g$ e $p > q$
 2. $p \ll q$
 3. *Legato (non si staccano le mani) porta ad una difficoltà ∞ se $f = g$*
 4. Evitare $g \in \{4,5\}$
 5. Evitare le seguenti transizioni: $3 \rightarrow 4$ e $4 \rightarrow 3$

Non sempre usando dei sottoproblemi arriviamo ad una soluzione ottima. In questi casi poiché non abbiamo il riuso non ha senso usare un approccio di programmazione dinamica. Possiamo osservare che il sottoproblema che possiamo individuare e suonare al meglio una sottosequenza di note. Identificato il sottoproblema dobbiamo ora vedere che scelte fare. In questo caso abbiamo un concetto di stato, infatti dobbiamo memorizzare le dita usate per poi valutare le transizioni. Possiamo ad esempio pensare:

1. **Sottoproblema:** minimizzare la difficoltà d per il suffisso (note rimanenti) $[i:]$. Questo va però rivisto introducendo il concetto di stato dato che ho trovato il dito f da usare sulla nota $[i]$. Da cui: #sottoProblemi = $n \cdot |F|$ dove n è il numero di note della sequenza.
2. **Guessing:** dobbiamo scegliere quale dito dobbiamo usare per la nota $[i+1]$.
3. **Ricorrenza:**

$$DP[i] = \min_{g \in F} \{DP[i + 1] + d(\text{nota}[i], f, \text{nota}[i + 1], g)\}$$

Il problema in questa ricorrenza è che non abbiamo l'informazione sullo stato; f infatti non è una scelta, ma ci viene dato dal problema che stiamo risolvendo. Dobbiamo quindi scrivere:

$$DP[i, f] = \min_{g \in F} \{DP[i + 1, g] + d(nota[i], f, nota[i + 1], g)\}$$

Il caso base sarà $DP[n, f] = 0$. Da cui: $\text{TempoPerSottoproblema} = \theta(|F|)$ ovvero il minimo sulle possibili dita che possiamo fare. È quindi costante tale valore dato che abbiamo fissato che possiamo usare cinque dita.

4. Non abbiamo cicli quindi la complessità sarà: $\text{TempoTotale} = \theta(n \cdot |F|^2)$.
5. Il problema originario lo risolviamo cercando il $\min_{f=1 \dots F} \{DP[0, f]\}$ ovvero dobbiamo scegliere da quale dito partire.

Problema 11: Tetris

Ne trattiamo una versione semplificata; analizziamone il problema:

- Ho una sequenza di n pezzi.
- Una board inizialmente vuota di larghezza w e altezza h .
- Dobbiamo scegliere la coordinata x del pezzo (dove lasciare il pezzo) e l'orientamento (il pezzo può girare di 90°).
- Lasciato cadere il pezzo finché non ne urta un'altro.
- Le righe non si cancellano.
- L'obiettivo è sopravvivere e quindi consumare tutti i pezzi senza eccedere nell'altezza h .

Vediamo i seguenti passi:

1. **Sottoproblema:** il sottoproblema è sopravvivere con una sottosequenza (il suffisso $[i:]$ che denota i pezzi disponibili). Anche questa volta dobbiamo introdurre il concetto di stato che è legato ai numeri di pezzi disponibili, e alla capacità residua della board (e quindi all'altezza e la base residua). Quindi diventa sopravvivere a $[i:]$ dato lo stato della board. Possiamo vedere lo stato della board come un vettore $h = \{h_0, h_1, \dots, h_w\}$, ovvero un vettore che tiene traccia dell'altezza per ogni unità della base. Tale vettore viene definito anche **skyline**. Il numero di sottoproblemi è dato da: $\#\text{sottoproblemi} = n \cdot h^w$; infatti per ogni colonna abbiamo h possibilità e questo va fatto per gli n pezzi.
2. **Guessing:** Dato un pezzo, dobbiamo scegliere tra i 4 possibili orientamenti. Ma non solo, considerando il caso peggiore (il pezzo è messo in modo che occupa la massima altezza) dobbiamo considerare il caso in cui lo posizioniamo nelle w posizioni della base.
3. **Ricorsione:**

$$DP[i, h] = \max_{\text{possibili scelte}} \{DP[i + 1, h'] | \text{scelta}\}$$

Vogliamo massimizzare l'esito che sarà 0 o 1, ovvero vinto o perso. A partire dalla disposizione di i , dobbiamo decidere dove porre l'elemento $i + 1$, consideriamo il nuovo stato h' condizionato dalla scelta fatta nel posizionamento di i .

Il tempo per sottoproblema è $\theta(w)$

4. Non abbiamo cicli quindi: $\text{TempoTotale} = n \cdot w \cdot h^w$.
5. La soluzione originale la ricaviamo con $DP[0, 0]$. Dove il secondo h , mi indica il vettore skyline inizializzato a 0 (infatti inizialmente non abbiamo pezzi).

Problema 12: Super Mario Bros

OSS: nei giochi abbiamo un concetto di stato molto forte, infatti partendo da uno stato, si può effettuare un'azione e passare ad uno stato successivo.

Il problema è:

- Consideriamo il livello 1.
- Abbiamo una bassa risoluzione $w \cdot h$.
- In questo caso il concetto di stato è dato da una possibile configurazione, caratterizzata da una serie di variabili. Sicuro tra le possibili variabili, dobbiamo avere la posizione nello schermo di super Mario e gli ostacoli presenti nel frame. Vediamoli nel dettaglio:
 1. Posizione del frame (quanto è avanzato rispetto all'inizio). N
 2. Stato di: oggetti, nemici, monete, ostacoli. Per semplicità diciamo che per ogni punto dello schermo abbiamo più C elementi. Quindi tale stato avrà dimensione: $C^{w \cdot h}$
 3. Posizione di Super Mario: $\theta(w)$
 4. Score S
 5. Tempo T
- A partire da un certo stato, avrò una funzione di transizione che mi dirà in che stato arrivo h' , effettuando una certa azione nello stato h .

$$\delta(\text{configurazione}, \text{azione}) = \text{configurazione}'$$

Dove $\text{azione} = \{\text{sopra, sotto, destra, sinistra, press/release}\}$

Quindi abbiamo:

1. **Sottoproblema:** best score o tempo a partire dalla configurazione cnf . $O(n \cdot w \cdot C^{w \cdot h} \cdot S \cdot T)$.
2. **Guessing:** azioni $O(1)$ ovvero le 5 azioni che posso fare.
3. **Ricorsione:**

$$DP(\text{cnf}) = f(x) = \begin{cases} \infty, & \text{se sono morto o è scaduto il tempo} \\ \text{cnf.score}, & \text{se sono arrivato alla fine} \\ \max_{\forall A \in \text{Azioni}} \{DP[\delta(\text{cnf}, A)]\} & \end{cases}$$

Quindi: $\text{TempoPerSottoproblema} = O(1)$ ovvero il tempo per sottoproblema

4. $\text{TempoTotale} = \#\text{Sottoproblemi}$
5. $DP(\text{cnf}_0)$

OSS: ulteriore problema è che alcune variabili che caratterizzano lo stato portano ad una complessità pseudo-polynomiale.

Algoritmi Greedy

Per alcuni problemi di ottimizzazione, un approccio greedy risulta più efficiente della programmazione dinamica. Questa, infatti, è una variante della programmazione dinamica che però non porta necessariamente ad una soluzione ottima; infatti, non esploriamo tutto lo spazio di stato ma ci limitiamo a considerare gli ottimi locali per costruire la soluzione globale. Questo approccio genera quindi delle euristiche, che possono approssimare l'ottimo globale risparmiando notevolmente in complessità. Questi algoritmi vengono usati per risolvere problemi *NP – Hard*, ovvero algoritmi che sarebbero intrattabili con approcci di Dynamic Programming, con complessità ragionevoli. Ad esempio, l'algoritmo di Dijkstra per il calcolo del percorso minimo utilizza un approccio greedy.

Problema 13: Problema dello zaino frazionato

Nel problema dello zaino abbiamo visto che un'oggetto lo possiamo prendere per intero o no. Una variante prevede di prendere solo parte di questo oggetto. In questo modo possiamo trovare una soluzione più efficiente rispetto a quella considerata. Abbiamo anche visto che potevamo considerare una funzione che valutava la densità degli oggetti (rapporto valore/dimensione) e ordinare il vettore in base a tale densità. Questo è una soluzione ma è un approccio Greedy che non mi porta necessariamente ad una soluzione ottima. Possiamo riciclare questa idea e utilizzare questo approccio per risolvere questa variante arrivando però ad un ottimo globale.

Facciamo riferimento ad un problema leggermente diverso:

- Consideriamo un insieme S di n attività che richiedono l'uso esclusivo di una risorsa comune. Ogni attività i ha un tempo di inizio s_i e un tempo di fine f_i . Ogni attività impiega la risorsa comune nell'intervallo $[s_i, f_i]$. L'obiettivo è selezionare un insieme di attività mutuamente compatibili di cardinalità massima.
- Dati due elementi a_i e a_j sono **compatibili** se $s_i \geq f_j$ o $s_j \geq f_i$; ovvero uno dei due task inizia dopo che l'altro termina. Quindi dati i seguenti elementi:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Sono mutuamente esclusivi gli elementi: $\{a_3, a_9, a_{11}\}, \{a_1, a_4, a_8, a_{11}\}, \{a_2, a_4, a_9, a_{11}\}$.

Dato questo problema immaginiamo di applicare una soluzione di programmazione dinamica. Definiamo in primo luogo il **sottoproblema** da trattare: se identifichiamo con S_{ij} l'insieme delle attività compatibili con a_i e a_j allora $S_{ij} = \{a_k \in S : f_i \leq s_k \leq f_k \leq s_j\}$. Quindi ogni elemento interno all'insieme deve iniziare dopo a_i e finire prima di a_j . Per comodità possiamo aggiungere due attività fintizie a_0 e a_{n+1} : $f_0 = 0$ e $s_{n+1} = \infty$.

Ora ipotizziamo che le attività sono ordinate per tempo di fine crescente: $f_0 \leq f_1 \leq f_2 \leq \dots \leq f_n < f_{n+1}$ e proviamo che $S_{ij} = \emptyset$ se $i \geq j$. Questa dimostrazione è abbastanza semplice, infatti, se per assurdo esiste $a_k \in S_{ij}$ avremmo $f_i \leq s_k \leq f_j$ e quindi $f_i < f_j$ che contraddice l'ipotesi dato che l'elemento i -esimo dovrebbe precedere l'elemento j -esimo.

Quindi i sottoproblemi tra cui scegliere sono del tipo: determinare l'insieme massimo di attività mutuamente compatibili tra quelle appartenenti a S_{ij} , con $0 \leq i < j \leq n + 1$ (gli altri sono vuoti).

Quindi sia a_k un'attività inclusa in S_{ij} e quindi ($f_i \leq s_k < f_k \leq s_j$) allora si generano due sottoproblemi:

- S_{ik} (attività che iniziano dopo che a_i finisce e finiscono prima che a_k inizi).
- S_{kj} (attività che iniziano dopo che a_k finisce e finiscono prima che a_j inizi).

Una soluzione ad S_{ij} è data dall'unione delle soluzioni a S_{ik} e S_{kj} più a_k . A questo punto detta A_{ij} una struttura ottima di S_{ij} che include a_k , le sottosoluzioni A_{ik} di S_{ik} e A_{kj} di S_{kj} devono essere ottime. Possiamo dimostrare questo per assurdo; infatti, se esistesse una soluzione A'_{ik} di S_{ik} con più attività di A_{ik} potremmo sostituire A'_{ik} a A_{ik} in A_{ij} , ottenendo un insieme di cardinalità maggiore rispetto alla soluzione ottima (analogo per A_{kj}). Il passo successivo è definire il valore di una soluzione ottima in maniera ricorsiva. Se a_k appartiene alla soluzione ottima di S_{ij} , il valore della soluzione ottima di S_{ij} sarà: $c[i,j] = c[i,k] + c[k,j] + 1$. Tuttavia, k non è noto, può essere una delle attività comprese tra $a_i + 1$ e $a_j - 1$. Possiamo quindi fare $j - i - 1$ scelte. Quindi:

$$c[i,j] = \begin{cases} 0, & \text{se } S_{ij} \in \emptyset \\ \max_{i < k < j} \{c[i,k] + c[k,j] + 1\}, & \text{altrimenti} \end{cases}$$

A questo punto, si potrebbe utilizzare l'approccio bottom-up della programmazione dinamica. Vediamo ora come possiamo usare un approccio greedy. Partiamo dal seguente teorema:

Teorema 1: Sia S_{ij} un sottoproblema non vuoto e a_m l'attività in S_{ij} con il tempo di fine più piccolo $f_m = \min\{f_k : a_k \in S_{ij}\}$. Allora:

- L'attività a_m è usata in qualche insieme di dimensione massima di attività mutualmente compatibili in S_{ij} .
- Il sottoproblema S_{im} è vuoto; quindi, scegliendo a_m si ha un solo sottoproblema non vuoto S_{mj} .

Questo risultato ci consente di:

- Avere una sola scelta quando si risolve un sottoproblema (l'attività che finisce prima), facile da calcolare.
- Dover risolvere un solo sottoproblema (l'altro è vuoto).

Quindi piuttosto che scegliere il massimo su k , possiamo pensare di prendere ad ogni ricorsione l'elemento con tempo di fine minimo che rispetta la compatibilità. Un possibile algoritmo è il successivo:

```
Recursive-Activity-Selector (s, f, i, j)
m ← i+1
while m < j and s_m < f_i // find the first activity in S_ij
    do m ← m + 1
if m < j
    then return {a_m} ∪ Recursive-Activity-Selector (s, f, m, j)
    else return ∅
```

Quindi ad ogni iterazione cerchiamo l'elemento a_m con tempo di fine minimo che inizia dopo l'elemento a_i . La complessità è $\Theta(n)$, dato che ogni attività è esaminata una sola volta nel ciclo while.

OSS: procedura vista ha una ricorsione “quasi” in coda e le procedure ricorsive in coda sono facilmente convertibili in procedure iterative.

Greedy-Activity-Selector (s, f) Sulla sinistra possiamo osservare una procedura iterativa (bottom-up) e anche questa presenta una complessità lineare $\theta(n)$.

```
n ← length[s]
A ← { $a_1$ }
i ← 1
for m ← 2 to n
    do if  $s_m \geq f_i$ 
        then A ← A ∪ { $a_m$ }
        i ← m
return A
```

OSS:

1. Un primo segno dell'applicabilità di un approccio greedy è la possibilità di effettuare la scelta “greedy”. La soluzione globalmente ottima può essere raggiunta mediante una sequenza di scelte greedy, ovvero scelte che tendono a selezionare un ottimo locale. Nella programmazione dinamica, la scelta tra sottoproblemi può essere effettuata dopo aver risolto i sottoproblemi (approccio bottom-up). Gli algoritmi greedy prima effettuano la scelta e poi risolvono il sottoproblema scelto. Occorre però provare che la scelta greedy conduce all'ottimo globale.
2. Un altro segno dell'applicabilità di un approccio greedy (e della programmazione dinamica) è la presenza della sottostruttura ottima. L'approccio va scelto in base alla possibilità di dimostrare che una scelta greedy conduce all'ottimo globale

Attenzione a non:

- Utilizzare la programmazione dinamica quando l'approccio greedy è applicabile.
- Utilizzare un approccio greedy quando è richiesta la programmazione dinamica.

Sono praticamente tecniche mutamente esclusive.

CAPITOLO 8 - ALGORITMO PARALLELI

Gli algoritmi visti fin ora sono seriali, ovvero vengono eseguiti su macchine con un solo processore, che esegue una istruzione alla volta. Quando parliamo di algoritmi paralleli parliamo di algoritmi eseguiti su multiprocessore che può eseguire più istruzioni contemporaneamente. Possiamo quindi considerare architetture multi-core, multi-processore, cluster di macchine, supercomputer.

Dobbiamo distinguere:

- **parallelismo:** processi che lavorano su processori fisici diversi
- **concorrenza:** processi che si sospendono in attesa che una risorsa si liberi e danno spazio ad un altro processo di eseguire. Due processi sono concorrenti quando uno inizia prima che l'altro termini.

Possiamo riscrivere gli algoritmi visti, in modo da sfruttare parallelismo e concorrenza col fine di ottimizzare l'algoritmo stesso. Possiamo scrivere un algoritmo su più thread che dialogano tra di loro (si scambiano messaggi o usano memorie condivise). Quando abbiamo architetture più complesse e quindi con più processori diventa fondamentale gestire problematiche che non si hanno con sistemi mono-processore e quindi problematiche quali lo scheduling dei task, gestione della cache, gestione della memoria condivisa ecc..

OSS: se abbiamo un ciclo, possiamo pensare di parallelizzare questo. Il problema è che non è detto che il codice lo consenta; ad esempio, se voglio ottenere $b[i] = a[i] + a[i - 1]$ non è un problema parallelizzare. Altre situazioni in cui le operazioni devono essere svolte con un certo ordine temporale non posso parallelizzare; posso però pensare di riscrivere il codice per renderlo parallelizzabile. Problema base è che non tutto il codice è parallelizzabile.

Negli esempi che vedremo assumeremo il modello del multithreading dinamico. Questa è una semplice estensione del modello di programmazione seriale.

OSS: Dai programmi che vedremo eliminando le parole chiave della concorrenza (parallel, spawn e sync) si ottiene un programma seriale funzionante.

Fibonacci

```

FIB (n)
if n <= 1
    return n
else x = FIB (n-1)
    y = FIB (n-2)
    return x+y

```

Prendiamo come esempio il calcolo del numero di Fibonacci.
Rifacciamoci all'implementazione ricorsiva.

Per parallelizzare posso usare parole chiave:

- **spawn**: generare una procedura (biforcazione come la fork).
- **sync**: per attendere e quindi definire un punto di sincronizzazione.
- **parallel**: è usato nei for per parallelizzarne le operazioni.

```

P-FIB (n)
1 if n <= 1
2     return n
3 else x = spawn P-FIB (n-1)
4     y = P-FIB (n-2)
5     sync
6     return x+y

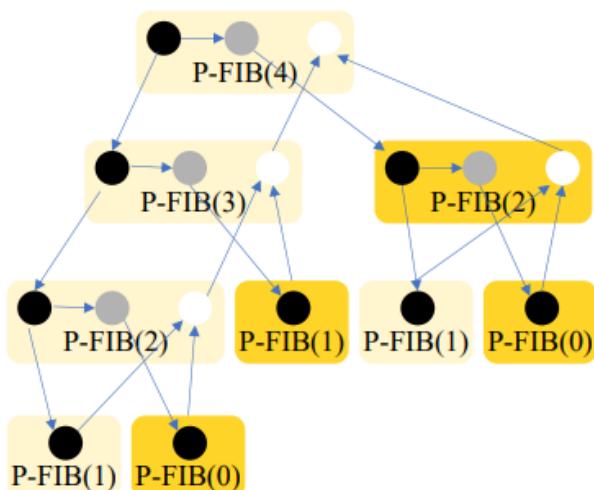
```

Con **spawn**, la procedura madre può generare una procedura figlia e continuare la propria esecuzione. È compito dello scheduler determinare quali procedure eseguire contemporaneamente, assegnandole ai processori disponibili. **sync** indica che la procedura madre deve aspettare che siano terminate tutte le procedure figlie prima di procedere oltre

È utile rappresentare l'insieme delle istruzioni di un programma multi-thread come un grafo orientato aciclico; dove:

- Ogni vertice è un'istruzione (o sequenza di istruzioni che non contiene nessun controllo parallelo – parliamo di **strand**).
- Gli archi rappresentano le dipendenze tra istruzioni: un arco (u, v) indica che u deve essere eseguita prima di v .

Se un vertice ha due successori, uno di essi deve essere stato generato da un'istruzione **spawn**. Un vertice con più predecessori indica che i predecessori si sono ricongiunti a causa di un'istruzione **sync**. Se c'è un cammino da u a v , essi sono logicamente in serie, altrimenti sono logicamente in parallelo. Nel grafo:



• Ogni cerchio è uno strand.

• Cerchio nero → righe 1-3.

• Cerchio grigio → righe 4-5.

• Cerchio bianco → riga 6.

Nell'immagine abbiamo nodi in giallo chiaro, che indicano le procedure generate (con **spawn**). E nodi in giallo scuro che indicano le procedure chiamate.

Possiamo distinguere vari archi nel grafo:

- **Archi di generazione** e **archi di chiamata** (chiamata ricorsiva) verso il basso.
- **Archi di continuazione** sono orizzontali e sono relativi ad uno stesso nodo.
- **Archi di ritorno** verso l'alto, che permettono la sincronizzazione.

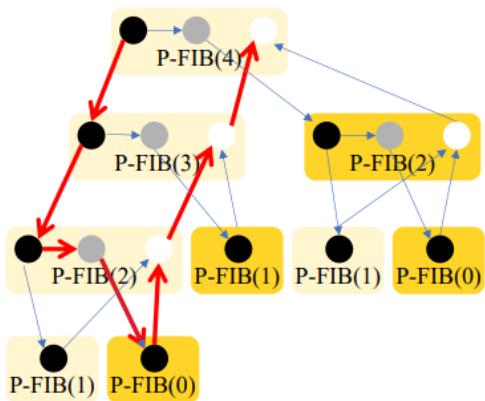
Metriche

Posso individuare due possibili metriche per misurare l'efficienza di un algoritmo multithread:

- **Lavoro**: il tempo totale per eseguire un intero calcolo in un processore, ovvero la somma dei tempi impiegati da ciascuno degli strand che formano un DAG di calcolo.
- **Durata**: il tempo più lungo per eseguire gli strand lungo un qualsiasi cammino nel DAG.

Se ogni strand nel DAG richiede un'unità di tempo:

- Il **lavoro** è il numero di vertici nel DAG.
- La **durata** è pari al numero di vertici sul cammino critico nel DAG.



Trovare un cammino critico in un DAG richiede $\Theta(|V| + |E|)$; approfondiremo meglio questo aspetto con la teoria dei grafi. Osserviamo che se ogni strand richiede un'unità di tempo, allora nell'esempio di Fibonacci abbiamo:

- Lavoro (numero di vertici) = 17.
- Durata (lunghezza cammino critico) = 8.

Il tempo di esecuzione effettivo di un calcolo multithread dipende non solo da lavoro e durata, ma anche da:

- Numero di processori disponibili.
- Modo in cui lo scheduler alloca gli strand sui processori.

Se indichiamo con T_P il tempo di esecuzione di un calcolo su P processori; allora T_1 è il lavoro (tempo con un solo processore), T_∞ è la durata con un numero infinito di processori.

Osserviamo che un computer parallelo con P processori non può essere più veloce di un computer con un numero illimitato di processori da cui la **legge della durata**: $T_P \geq T_\infty$.

Per generalizzare il discorso definiamo **un'unità di lavoro** come la quantità di lavoro che un processore può eseguire nell'unità di tempo.

Un computer parallelo con P processori può eseguire P unità di lavoro nell'unità di tempo da cui:

- può eseguire una quantità di lavoro pari a PT_P in un tempo T_P .
- $PT_P \geq T_1$ (il lavoro totale da svolgere è T_1).
- Dalla precedente ricaviamo la **legge del lavoro**: $T_P \geq T_1/P$.

Definiamo **speedup** di un calcolo su P processori il rapporto T_1/T_P . Dalla legge del lavoro, lo speedup può essere al più pari a P . Il fattore di riduzione del tempo di esecuzione è al più pari a P e quindi:

- Se lo speedup è $\Theta(P)$, il calcolo presenta uno **speedup lineare**.
- Se lo speedup è P , si ha uno **speedup lineare perfetto**.

Il rapporto tra lavoro e durata (T_1/T_∞) ci restituisce il **parallelismo del calcolo multi-thread**:

- Può fungere da limite superiore: indica lo speedup massimo che può essere ottenuto con un numero qualsiasi di processori.
- Rappresenta un limite alla possibilità di ottenere uno speedup lineare perfetto.

Quando il numero di processori supera il parallelismo, il calcolo non può ottenere uno speedup lineare perfetto. Quanti più processori vengono utilizzati oltre il parallelismo, tanto meno perfetto sarà lo speedup. Come rapporto, indica la quantità media di lavoro che può essere eseguito in parallelo in ogni passo lungo il cammino critico.

Ritornando all'esempio di Fibonacci, quando abbiamo $n=4$ abbiamo $T_1/T_\omega = 17/8 = 2.125$. Non è possibile raggiungere uno speedup molto più che doppio, indipendentemente dal numero di processori usati. Per valori più grandi, il parallelismo risulta essere più elevato.

Possiamo indicare con il lavoro $T_1(n)$ il tempo di esecuzione dell'algoritmo seriale:

$$T_1(n) = \Theta(\phi^n) \text{ dove } \phi = (1 + \sqrt{5})/2 \text{ è il rapporto aureo.}$$

Se vogliamo determinare la durata della composizione di due strand; questa è la somma delle durate se gli strand sono in serie e il massimo delle durate se sono in parallelo. Da cui

$$T_\omega(n) = \max(T_\omega(n-1), T_\omega(n-2)) + \Theta(1) = T_\omega(n-1) + \Theta(1), \text{ da cui } T_\omega(n) = \Theta(n)$$

Possiamo calcolare il parallelismo di P-FIB come $\frac{T_1}{T_\omega} = \frac{\Theta(\phi^n)}{\Theta(n)} = \Theta\left(\frac{\phi^n}{n}\right)$ che quindi cresce molto velocemente con n . Da cui un valore piccolo di n è sufficiente per ottenere uno speedup lineare quasi perfetto perfino nei computer più grandi.

Merge Sort Parallello

Partiamo da un'osservazione: il divide et impera ben si presta alla parallelizzazione. Essendo basato sul paradigma divide et impera, Merge Sort ben si presta ad una implementazione multi-thread

<u>Merge-Sort'</u> (A, p, r)	Il lavoro $MS'_1(n) = 2 MS'_1(n/2) + \Theta(n) = \Theta(nlgn)$.
if $p < r$	La durata $MS'_\infty(n) = MS'_\infty(n/2) + \Theta(n) = \Theta(n)$
then $q \leftarrow \lfloor(p+r)/2\rfloor$	Il parallelismo non è elevato infatti:
spawn Merge-Sort (A, p, q)	$\Theta(nlgn)/\Theta(n) = \Theta(lgn)$
Merge-Sort (A, q+1, r)	
sync	
Merge (A, p, q, r)	

OSS: Per incrementare il parallelismo è possibile rendere anche la funzione Merge multithread e per fare ciò la costruiamo con un approccio divide et impera. Partendo dal principio di funzionamento del merge, vogliamo fondere due sottoarray:

- $T[p_1 \dots r_1]$ di lunghezza $n_1 = r_1 - p_1 + 1$.
- $T[p_2 \dots r_2]$ di lunghezza $n_2 = r_2 - p_2 + 1$.

in un sottoarray $A[p_3 \dots r_3]$ di lunghezza $r_3 - p_3 + 1 = n_1 + n_2$. A questo punto se ipotizziamo $n_1 \geq n_2$, possiamo individuare la mediana di $T[p_1 \dots r_1]$ (ovvero l'elemento di mezzo) che indichiamo con q_1 e utilizziamo la ricerca binaria per trovare l'indice q_2 che denota il primo elemento di $T[p_2 \dots r_2]$ maggiore o uguale a $x = T[q_1]$. A questo punto possiamo calcolare $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$ e poniamo $A[q_3] = x$. Successivamente possiamo fondere ricorsivamente i due sottoarray $T[q_1 + 1 \dots r_1]$ e $T[q_2 \dots r_2]$ e inserire il risultato nel sottoarray $A[q_3 + 1 \dots r_3]$.

Il **caso base** lo si ha quando i sottoarray sono vuoti e quindi non c'è nulla da fare. Con tale approccio ad ogni passo la dimensione complessiva dei sottoproblemi diminuisce di un unità (ovvero la mediana selezionata dal primo sottoarray).

Binary Search

```
Binary-Search (x,T,p,r)
  low ← p
  high ← max (p, r+1)
  while low < high
    mid ← ⌊(low + high)/2⌋
    if x ≤ T[mid]
      high ← mid
    else low ← mid + 1
  return high
```

In primo luogo, per utilizzare questo approccio dobbiamo essere in grado di individuare q_2 nel secondo sottoarray e per fare ciò possiamo usare una ricerca binaria modificata. Dato quindi un sottoarray $T[p \dots r]$ ordinato e una chiave x , la procedura deve restituire:

- p , se $T[p \dots r]$ è vuoto ($r < p$) oppure $x \leq T[p]$.
- Il massimo indice q ($p < q \leq r + 1$) tale che $T[q - 1] < x$, altrimenti.

La complessità di questo algoritmo è:

$$\Theta(\lg n), \text{ con } n = r - p + 1$$

P-Merge

```
P-Merge (T,p1,r1,p2,r2,A,p3)
  n1 ← r1-p1+1
  n2 ← r2-p2+1
  if (n1 < n2) // assicura che n1 ≥ n2
    scambia p1 con p2
    scambia r1 con r2
    scambia n1 con n2
  if n1 = 0 // entrambi vuoti?
    return
  else q1 ← ⌊(p1+r1)/2⌋
    q2 ← Binary-Search (T[q1],T,p2,r2)
    q3 ← p3+(q1-p1)+(q2-p2)
    A[q3] ← T[q1]
    spawn P-Merge (T,p1,q1-1,p2,q2-1,A,p3)
    P-Merge (T,q1+1,r1,q2,r2,A,q3+1)
    sync
```

$$n_2 = (n_2 + n_2)/2 \leq (n_1 + n_2)/2 = n/2$$

$$\text{da cui: } \lfloor n_1/2 \rfloor + n_2 \leq n_1/2 + n_2/2 + n_2/2 \leq (n_1 + n_2)/2 + n/4 = 3n/4$$

$$\text{Possiamo scrivere: } PM_\omega(n) = PM_\omega(3n/4) + \Theta(\lg n)$$

In questo caso non si può applicare il teorema dell'esperto, ma si può dimostrare che la soluzione è $PM_\omega(n) = \Theta(\lg^2 n)$.

Passiamo ora a stimare il lavoro:

Il lavoro $PM_1(n) = \Omega(n)$ perché bisogna copiare n elementi da T ad A . Nel caso peggiore, considerato che le due invocazioni ricorsive fondono al più n elementi ($n-1$, per la precisione), allora abbiamo: $PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + O(\lg n)$; con $\frac{1}{4} \leq \alpha \leq \frac{3}{4}$ (per quanto dimostrato prima) che può variare in ogni livello di ricorsione. Con il metodo di sostituzione si può dimostrare che $PM_1(n) = O(n)$; di conseguenza, $PM_1(n) = \Theta(n)$. Il parallelismo di P-Merge è $PM_1(n)/PM_\omega(n) = \Theta(n/\lg^2 n)$.

A differenza di Merge, P-Merge:

- Non richiede che i due sottoarray da fondere siano adiacenti.
- Gli elementi “fusi” vengono inseriti in un vettore passato come parametro.

Stimiamo in primo luogo la **durata**:

Indichiamo con $M_\omega(n)$ la durata corrispondente ad una invocazione di P-Merge per fondere $n = n_1 + n_2$ elementi.

La generazione e la chiamata di P-Merge operano in parallelo, quindi occorre considerare quella più costosa. Nel caso peggiore, una chiamata fonde $\lfloor n_1/2 \rfloor + n_2$ elementi. Quindi ipotizziamo che la mediana del primo vettore sia più grande di ogni valore del secondo sottoarray. A questo punto possiamo scrivere:

P-Merge-Sort

```
P-Merge-Sort (A,p,r,B,s)
n ← r-p+1
if n = 1
    B[s] ← A[p]
else sia T[1..n] un nuovo array
    q ← ⌊(p+r)/2⌋
    q' ← q-p+1
    spawn P-Merge-Sort (A,p,q,T,1)
    P-Merge-Sort (A,q+1,r,T,q'+1)
    sync
    P-Merge (T,1,q',q'+1,n,B,s)
```

Anche in questo caso non si può applicare il teorema dell'esperto, ma si può dimostrare che la soluzione è $PMS_\omega(n) = \Theta(\lg^3 n)$ da cui il parallelismo di P-Merge-Sort è $\Theta(n \lg n)/\Theta(\lg^3 n) = \Theta(n/\lg^2 n)$ che è molto migliore di quello di Merge-Sort' ($\Theta(\lg n)$)

Sfruttando tale metodo possiamo riscrivere il P-Merge-Sort come in figura. Osserviamo che il metodo riceve come argomento un sottoarray B che conterrà l'array ordinato. T è un array temporaneo usato per memorizzare i due sottoarray ordinati che poi vengono fusi. A questo punto il lavoro sarà:

$$\begin{aligned} PMS_1(n) &= 2PMS_1(n/2) + PM_1(n) \\ &= 2PMS_1(n/2) + \Theta(n) \end{aligned}$$

E quindi: $PMS_1(n) = \Theta(n \lg n)$.

Per quanto riguarda la durata, le due chiamate ricorsive operano in parallelo e hanno costo uguale:

$$\begin{aligned} PMS_\omega(n) &= PMS_\omega(n/2) + PM_\omega(n) \\ &= PMS_\omega(n/2) + \Theta(\lg^2 n) \end{aligned}$$

CAPITOLO 6 - TEORIA DEI GRAFI

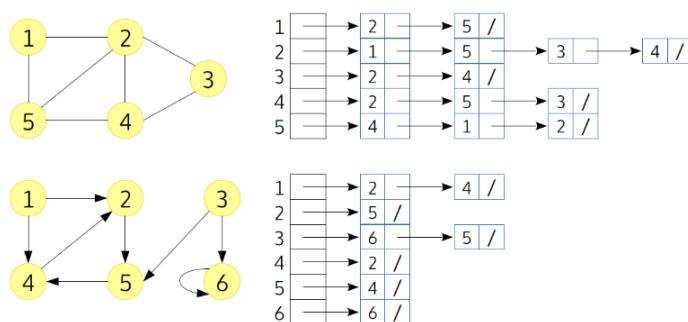
I grafi vengono usati in molteplici applicazioni; alcune possibili applicazioni sono:

1. Web crawling: usato dai motori di ricerca per capire la raggiungibilità delle pagine mediante grafi.
2. Social Networks.
3. Routing: problemi di istradamento.
4. Garbage Collection: Java ha un processo che si occupa di eliminare oggetti non utilizzati; quindi, bisogna determinare su un certo oggetto e raggiungibile o meno per decidere se eliminarlo o meno)
5. Model Checking: tecnica di verifica del software, dove i nodi rappresentano gli stati e gli archi il flusso di esecuzione degli stati. Nella verifica vogliamo dimostrare che il software abbia certe proprietà (non è testing). Ad esempio, dimostrare che il software non arriverà mai in un certo stato critico o che a partire da un certo stato arrivi esattamente in uno stato desiderato.
6. Puzzles/Games.

Nomenclatura di base

Un grafo consiste di un insieme di nodi V connessi mediante un insieme di archi E (questi possono essere direzionali e non direzionali). Possiamo rappresentare i grafi in diversi modi:

Lista di adiacenza

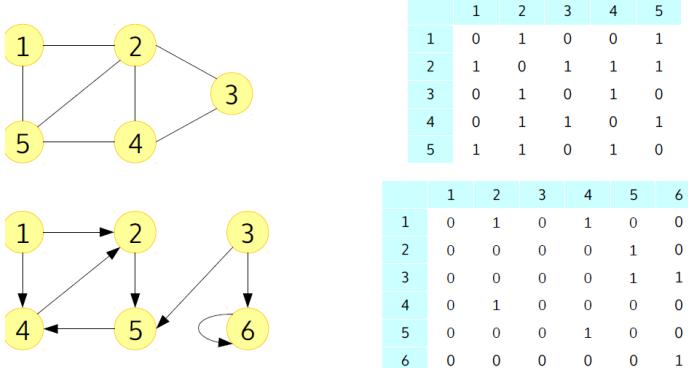


Per ogni elemento abbiamo una lista che ci indica a partire dal nodo considerato in quali altri nodi posso arrivare.

In questo caso abbiamo come limite inferiore $\theta(|V| + |E|)$, infatti dobbiamo sommare per ogni nodo il numero di archi che questo possiede e quindi sarebbe la somma degli indegree. In generale se abbiamo archi

orientati allora la complessità spaziale è $|E|$ altrimenti $2|E|$. In generale ad ogni elemento della lista è associato l'id del nodo collegato e il peso dell'arco.

Matrice di adiacenza



In questo caso la cella (i, j) vale 0 se i nodi non sono collegati o il peso dell'arco se sono collegati. La complessità spaziale in questo caso è $O(|V|^2)$.

OSS: La scelta tra le due soluzioni è come al solito un tradeoff che dipende dall'applicazione. Se il numero di archi è molto più grande dei nodi non conviene la matrice, d'altra parte la matrice consente di

effettuare accessi diretti a differenza dalla lista che prevede ricerche lineari. Altro aspetto da considerare è il caso in cui gli archi hanno o meno un peso. La matrice di adiacenza si preferisce in genere per grafi di piccole dimensioni e quando i grafi non sono pesati (0 o 1 consuma 1 bit).

Grafi – Algoritmi di ricerca

Sono algoritmi di ricerca in termini di esplorazione dei grafi. Possiamo pensare di esplorare tutti i nodi (o quelli che rispettano una proprietà) o tutti gli archi.

In generale le due strategie più adottate sono:

- **Esplorazione in ampiezza.**
- **Esplorazione in profondità.**

Visita in Aampiezza - BFS

Consideriamo il seguente problema: vogliamo risolvere il cubo di Rubik 2x2x2 partendo da una configurazione qualsiasi. Possiamo modellare questo problema come un grafo dove ogni nodo rappresenta una possibile configurazione e gli archi hanno costo unitario. Due nodi sono collegati da un arco se per passare da una configurazione all'altra bisogna fare una mossa.

Se consideriamo tutte le possibili configurazioni abbiamo $\#nodi = 8! \cdot 3^8 = 264\text{ milioni}$. Potremmo ridurre lo spazio degli stati considerando le configurazioni simmetriche (ovvero quelle che si ottengono rotando il cubo).

Un possibile approccio è utilizzare una visita in ampiezza partendo dalla configurazione desiderata e si può dimostrare che abbiamo undici livelli in tale grafo.

La visita in ampiezza (*BFS* – Breadth First Search) è un semplice algoritmo per “visitare” i nodi di un grafo. Alcuni importanti algoritmi si basano su questo approccio sono:

- Algoritmo di Dijkstra per lo shortest path.
- Algoritmo di Prim per il minimum-spanning-tree.

BFS: A partire da un nodo sorgente s , si visitano tutti i nodi adiacenti di un nodo e poi procede a visitare allo stesso modo i nodi adiacenti. Quindi tutti i nodi che distano k dalla sorgente vengono visitati prima del primo nodo che dista $k + 1$.

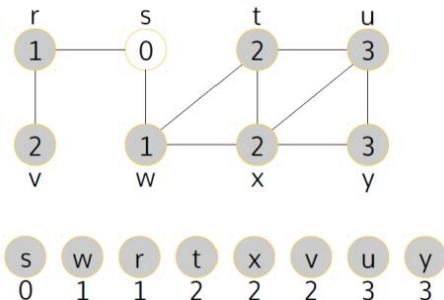
BFS produce un albero con radice in s che contiene tutti i nodi raggiungibili da s . Il percorso da s a v nell'albero è il più breve da s a v nel grafo. Possiamo pensare di utilizzare questo algoritmo per una ricerca di minimo percorso in un grafo in cui gli archi hanno peso unitario.

BFS può operare sia su grafi direzionali che non direzionali. Questo algoritmo nella navigazione dei nodi, in ogni istante può colorare i nodi con vari colori:

- **Bianco**, se non è ancora stato visitato.
- **Grigio**, se è stato visitato ma non tutti i suoi vicini sono stati visitati.
- **Nero**, se è stato visitato insieme a tutti i suoi vicini.

OSS:

- Se il nodo v viene visitato in quanto vicino di u , allora u è il predecessore di v . I predecessori vengono salvati per costruire l'albero.
- Ogni nodo viene visitato al più una volta e quindi ha al più un predecessore.



Inizialmente tutti i nodi tranne quello sorgente vengono marcati un valore infinito e il colore è posto a bianco. A partire da s , navigo prima w e poi r (i nodi adiacenti). A partire da w , visito tutti i nodi adiacenti (x, t) e poi quelli adiacenti a r (v). Quando visiterò i nodi di adiacenti di x , non visiterò nuovamente t dato che è già stato navigato da q (che viene colorato di grigio).

Algoritmo

```

BFS ( $G, s$ )
for each vertex  $u \in V[G] - \{s\}$ 
    do  $\text{color}[u] \leftarrow \text{WHITE}$ 
         $d[u] \leftarrow \infty$ 
         $\pi[u] \leftarrow \text{NIL}$ 
     $\text{color}[s] \leftarrow \text{GRAY}$ 
     $d[s] \leftarrow 0$ 
     $\pi[s] \leftarrow \text{NIL}$ 
     $Q \leftarrow \emptyset$ 
    Enqueue ( $Q, s$ )
    while  $Q \neq \emptyset$ 
        do  $u \leftarrow \text{Dequeue} (Q)$ 
            for each  $v \in \text{Adj}[u]$ 
                do if  $\text{color}[v] = \text{WHITE}$ 
                    then  $\text{color}[v] \leftarrow \text{GRAY}$ 
                         $d[v] \leftarrow d[u] + 1$ 
                         $\pi[v] \leftarrow u$ 
                    Enqueue ( $Q, v$ )
             $\text{color}[u] \leftarrow \text{BLACK}$ 

```

navigati e quindi si colora il nodo di nero.

OSS: l'algoritmo assume una rappresentazione mediante liste di adiacenza.

Si inizia con un ciclo di inizializzazione che colora tutti i nodi di bianco e pone $d(u) = \infty$, ovvero la distanza dal nodo s al nodo u . Inoltre, per ogni nodo viene salvato il predecessore $\pi(u) = \text{NIL}$ (incluso il nodo sorgente). Successivamente, partendo dal nodo sorgente che viene marcato di grigio, si definisce una coda Q che contiene tutti i nodi grigi, in cui viene inserito il nodo sorgente stesso. La seconda parte prevede un ciclo che viene ripetuto fintantoché nella coda non abbiamo nodi grigi. Ad ogni iterazione, si preleva il primo nodo dalla coda (dato che diventerà nero), e per ogni nodo collegato a u di colore bianco, (ovvero che non è stato navigato): si colora questo di grigio, si incrementa la distanza, si aggiunge il predecessore e si inserisce il nodo grigio nella coda. Alla fine del ciclo, tutti i nodi adiacenti ad u sono stati

Con questa strategia quando si esplora un nodo bianco, questo viene ricolorato di grigio e quindi non verrà più esplorato. Per quanto riguarda la complessità abbiamo che:

- L'overhead legato all'inizializzazione è $O(|V|)$.
- Il tempo legato alle operazioni di accodamento/deaccodamento è $O(|V|)$; in particolare le operazioni di enqueue e dequeue richiedono $O(1)$. Infatti, un nodo viene inserito nella coda al più una volta.
- Ad ogni esplorazione di un nodo si esplorano al più i nodi collegati a questo (in realtà solo quelli bianchi) e quindi abbiamo una complessità $O(|E|)$, ovvero la somma delle lunghezze delle liste di adiacenza).

In conclusione, la **complessità** è $O(|V| + |E|)$.

Ricerca di minimo percorso

Definiamo la distanza minima $\delta(s, v)$ tra s e v come il numero minimo di archi di qualunque percorso tra s e v .

TH. *BFS* scopre tutti i nodi raggiungibili dalla sorgente s e al termine si ha che $d[v] = \delta(s, v)$ per tutti i vertici v . Per ogni vertice v raggiungibile da s , uno shortest path da s a v è uno shortest path da s a $\pi[v]$ seguito dall'arco $(\pi[v], v)$.

Abbiamo visto che BFS costruisce un albero detto anche **breadth-first tree**.

Definiamo ora il **predecessor subgraph** di G , come:

- $G_\pi = (V_\pi, E_\pi)$ dove: $V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$ e $E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$

Quindi quest'albero è composto dai un insieme di nodi dato dai nodi che possiedono un predecessore a cui viene aggiunto il nodo sorgente e un insieme di archi dato proprio dagli archi tra i nodi $v \in V_\pi$ e i predecessori di v . Possiamo osservare che questo è effettivamente un albero dato che consideriamo proprio $|E| = |V_\pi| - 1$ archi. Osserviamo inoltre che in V_π abbiamo tutti i nodi raggiungibili dalla sorgente s e per ogni $v \in V_\pi$, esiste un unico percorso semplice da s a v in G_π che è anche lo shortest path da s a v in G .

Lemma: Il predecessor subgraph restituito da *BFS* è un breadth-first tree.

```
Print-Path (G, s, v)
if v=s
then print s
else if π[v] = NIL
    then print "no path from s to v exists"
    else Print-Path (G, s, π[v])
        print v
```

Il metodo sulla sinistra stampa i vertici lungo uno shortest path da s a v , sfruttando chiamate ricorsive al predecessore di v , fino ad arrivare al caso base in cui il predecessore è proprio il nodo sorgente. Ovviamente questo algoritmo ha senso se si assume di aver eseguito *BFS* in precedenza.

Il tempo di esecuzione è lineare nel numero di vertici che costituiscono il percorso. In realtà poiché prevede di applicare prima *BFS* allora la complessità è $O(|V| + |E|)$.

Visita in profondità – DFS

Depth-first search (*DFS*) esplora i vicini del nodo scoperto più di recente che ha ancora vicini non esplorati e quindi la visita procede per l'appunto “in profondità”.

OSS:

- Se ci sono nodi non esplorati, la ricerca riparte da uno di essi e quindi tutti i vertici sono esplorati.
- Proprio come *BFS* anche *DFS* tiene traccia dei predecessori di un nodo.
- Il **predecessor subgraph** in questo caso può essere composto da più alberi e per questo motivo, è definito in maniera leggermente diversa:
 - $G = (V, E_\pi)$
 - $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$

In questo caso quindi andiamo a considerare tutti i nodi. Il predecessor subgraph forma quindi una **depth-first forest** composta di diversi **depth-first tree**.

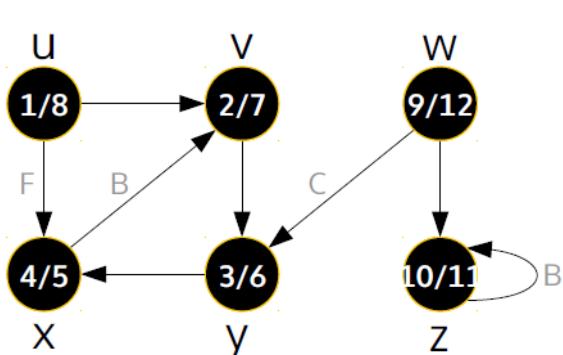
Abbiamo alcune applicazioni come la **Cycle detection**: molto utile per comprendere quale approccio usare ad esempio nella ricerca del minimo percorso.

Come in *BFS*, i **nodi** sono colorati di **bianco, grigio e nero**. Ciò assicura che ogni vertice finisce in un solo depth-first tree e quindi gli alberi della depth forest sono disgiunti.

DFS marca ciascun nodo con due “**timestamp**”:

- $d[v]$: segna l'istante di tempo in cui v è scoperto (e colorato di grigio).
- $f[v]$: segna l'istante di tempo in cui termina la ricerca dei vicini di v (che viene colorato di nero).

I **timestamp** sono interi compresi tra 1 e $2 \cdot |V|$. Per ogni nodo, c'è un istante di scoperta e uno di completamento. Ovviamente per ogni vertice u , $d[u] < f[u]$



Se consideriamo come nodo sorgente u , si visita prima il nodo v . Da v si visita y e da y si visita x . Osserviamo che il timestamp di $f[u] = 8$ dato che devo considerare il tempo per andare da u a x a cui sommo il tempo per tornare da x a u .

Algoritmo

```

DFS (G)
for each vertex  $u \in V[G]$ 
    do  $\text{color}[u] \leftarrow \text{WHITE}$ 
         $\pi[u] \leftarrow \text{NIL}$ 
    time  $\leftarrow 0$ 
for each vertex  $u \in V[G]$ 
    do if  $\text{color}[u] = \text{WHITE}$ 
        then DFS-Visit ( $u$ )
DFS-Visit ( $u$ )
color[ $u$ ]  $\leftarrow \text{GRAY}$ 
time  $\leftarrow \text{time} + 1$ 
d[ $u$ ]  $\leftarrow \text{time}$ 
for each  $v \in \text{Adj}[u]$ 
    do if  $\text{color}[v] = \text{WHITE}$ 
        then  $\pi[v] \leftarrow u$ 
            DFS-Visit ( $v$ )
color[ $u$ ]  $\leftarrow \text{BLACK}$ 
f[ $u$ ]  $\leftarrow \text{time} \leftarrow \text{time} + 1$ 

```

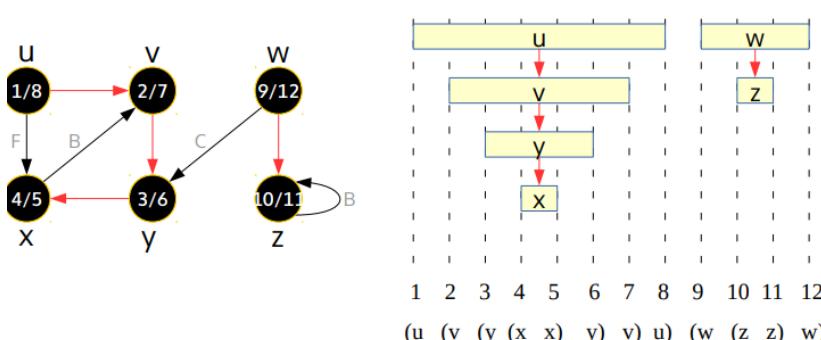
In questo caso dobbiamo distinguere due procedure: DFS() e DFS_Visit().

La prima procedura è composta da un primo ciclo di inizializzazione, analogo a BFS. Successivamente per ogni nodo u , se il nodo è bianco (non è stato già esplorato) effettua una visita dell'albero a partire proprio da u .

I due cicli hanno complessità $\theta(|V|)$ infatti si inizializzano i nodi e poi si prova a visitare tutti i nodi.

La visita si compone di una prima parte dove si setta il timestamp del nodo e poi lo si colora di grigio. A partire da questo nodo per ogni nodo bianco collegato a u con un arco, si crea il legame di predecessore e poi si chiama ricorsivamente la visita sul nodo trattato. Alla fine del ciclo sono stati esplorati tutti i nodi adiacenti ad esso e quindi si può colorare di nero il nodo. Infine, si definisce il timestamp di fine. Possiamo osservare che ancora una volta si percorre al più una volta ogni arco e quindi la complessità di DFS è $O(|V| + |E|)$.

OSS: abbiamo una notevole somiglianza col backtracking.



Una proprietà della DFS è che i tempi di scoperta e di terminazione dei nodi hanno una struttura a parentesi. Infatti, il tempo di chiusura del primo nodo è maggiore del tempo di chiusura degli altri nodi navigati.

TH: per ogni coppia di nodi u e v , si può verificare solo una delle seguenti condizioni:

1. Se gli intervalli $[d[u], f[u]]$ e $[d[v], f[v]]$ sono disgiunti allora u non è discendente di v né v è discendente di u nella depth-first forest. Stiamo quindi dicendo che i nodi appartengono a diversi depth-first tree.
2. Se l'intervallo $[d[u], f[u]]$ è contenuto interamente nell'intervallo $[d[v], f[v]]$ allora u è un discendente di v in un depth-first tree. E quindi appartengono allo stesso depth-first tree.
3. Se l'intervallo $[d[v], f[v]]$ è contenuto interamente nell'intervallo $[d[u], f[u]]$ allora v è un discendente di u in un depth-first tree. Praticamente è la vista duale prima.

Corollario: un vertice v è un discendente di u nella depth-first forest $\Leftrightarrow d[u] < d[v] < f[v] < f[u]$

Archi

La caratteristica di *DFS* è che partiziona gli archi di un grafo in 4 insiemi:

- **Tree edges:** sono gli archi in una depth-first forest. Quindi l'arco (u, v) è un tree edge se v è scoperto esplorando i vicini di u .
- **Back edges:** sono gli archi (u, v) con v antenato di u in un depth-first tree. I self-loop sono considerati back edges.
- **Forward edges:** sono gli archi (u, v) con u antenato di v in un depth-first tree.
- **Cross edges:** sono tutti gli altri archi. Abbiamo quindi archi tra nodi di diversi depth-first tree o dello stesso depth-first tree ma che non sono l'uno il discendente dell'altro.

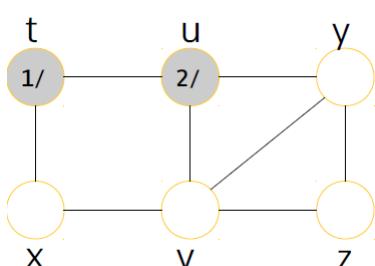
OSS: per i grafi non direzionali, ci può essere ambiguità tra (u, v) e (v, u) , che sono lo stesso arco. Un arco viene classificato sulla base della "direzione" che viene incontrata prima.

DFS può classificare gli archi man mano che li incontra. Un arco (u, v) può essere classificato sulla base del colore di v nel momento in cui l'arco è esplorato per la prima volta:

- Se v è bianco, (u, v) è un tree edge.
- Se v è grigio, (u, v) è un back edge.
- Se v è nero ($f[v] < f[u]$), (u, v) può essere un forward edge o un cross edge; in particolare:
 - Se $d[u] < d[v]$, (u, v) è un forward edge ($d[u] < d[v] < f[v] < f[u]$).
 - Se $d[u] > d[v]$, (u, v) è un cross edge.

TH: Il grafo G ha un ciclo se e solo se la ricerca in profondità trova un backedge.

OSS: una *DFS* su un grafo non direzionali, ogni arco è o un tree edge o un back edge. Questo è

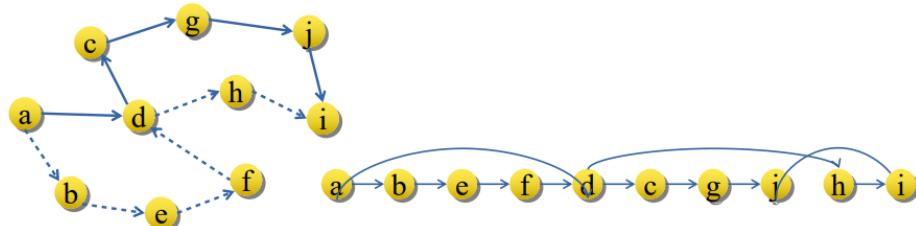


legato al fatto che tutti gli archi sono collegati tra di loro e quindi potrò arrivare a partire da un qualsiasi nodo ad un qualsiasi altro nodo. Quando percorrerò un nodo e provo a tornare su un nodo già percorso allora posso evidenziare solo un backedge.

Ordinamento topologico

DFS può essere usato per realizzare un ordinamento topologico di un **directed acyclic graph** (DAG). Un ordinamento topologico di un DAG determina un ordine dei nodi tale che se esiste un arco da u a v ($u \rightarrow v$) allora u precede v .

OSS: Se il grafo non è aciclico, non è possibile un tale ordinamento.



Tale ordinamento è utile, ad esempio, quando gli archi indicano una relazione di precedenza ad esempio nel job scheduling.

Topological-Sort (G)

Call $\text{DFS}(G)$ to compute finishing times $f[v]$ for each vertex v

As each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices

Si calcolano i tempi di fine sfruttando $\text{DFS}()$; in particolare ogni volta che un nodo termina, si inserisce il nodo nella testa di una linked list. Alla fine, avremo ordinata la lista per tempo di fine decrescente e quindi possiamo stampare banalmente tale lista. Il tempo di esecuzione è $\Theta(V + E)$; infatti $\text{DFS}()$ richiede $\Theta(V + E)$ e l'inserimento di $|V|$ vertici nella lista richiede $O(|V|)$.

Algoritmo di ricerca di percorso minimo in un grafo

In primo luogo, formalizziamo la definizione di grafo come:

- Insieme di nodi V e insieme di archi E (possono essere diretti o indiretti).
- Denotiamo con $w(u, v)$ il peso dell'arco da w a u .
- Possiamo definire un percorso come una successione di nodi: $p = \{V_0, V_1, \dots, V_{k-1}\}$.
- Il costo del percorso p è dato da $w(p) = \sum_{i=0}^{k-1} w(V_i, V_{i+1})$

Il problema di ricerca del percorso a costo minimo può essere espresso come:

$$\delta(u, v) = \begin{cases} \min\{w(p)\}, & \text{se } \exists p \\ \infty, & \text{altrimenti} \end{cases}$$

Algoritmo naive

Questo algoritmo assume assenza di cicli negativi, ovvero cicli in cui il costo totale è negativo. Possiamo quindi assumere la presenza di archi a peso negativo. Supponiamo inoltre di avere a disposizione l'operazione di Relax.

INIT:

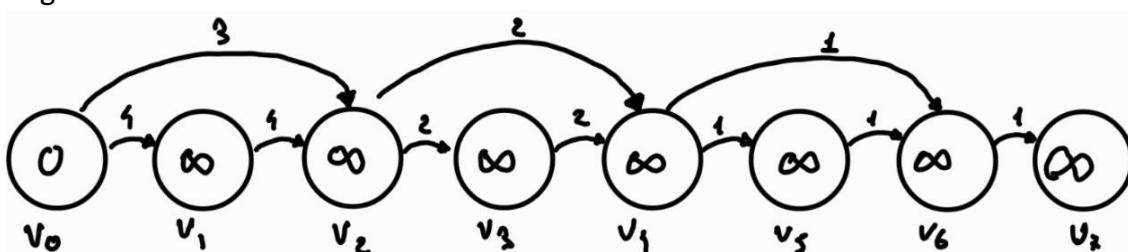
```
for(v in V):
    d[v]=infinite;
    PI[v]=NIL;
```

MAIN:

```
do:
    select edge(u,v); //arbitrariamente
    relax(u,v);
until d[v] <= d[u]+w(u,v) per ogni arco in E.
```

Un primo approccio potrebbe essere applicare l'operazione di *Relax* su tutti i nodi. Quindi l'idea è ripetere questo finché non posso più applicare l'operatore di *Relax* su nessuno degli archi dell'insieme.

Il problema di questo approccio è che la complessità dipende dall'ordine con cui vengono selezionati gli archi. Nel caso peggiore può portare ad una complessità esponenziale (intrattabile). Immaginiamo di avere il seguente grafo:



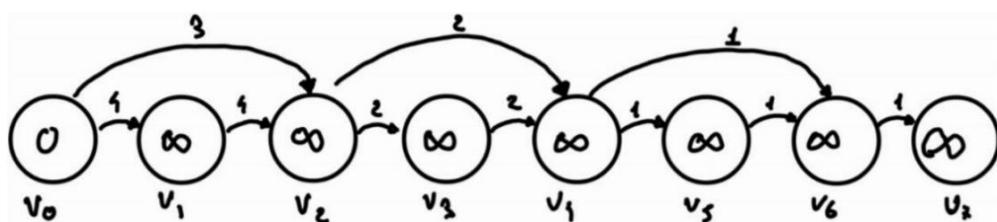
Supponiamo di scegliere il percorso lineare tra v_0 e v_7 , scegliendo come nodo iniziale v_0 (la sua distanza all'inizio è inizializzata a zero) quando calcoliamo i valori delle distanze mediante l'operatore di *Relax*, avremmo le distanze: 4, 8, 10, 12, 13, 14. Supponiamo ora di rilassare l'arco (v_6, v_4) abbiamo che $d[v_6] = 13$. Analogamente se poi rilassiamo (v_4, v_2) otteniamo $d[v_4] = 10$, che poi porterà a rilassare nuovamente l'arco (v_6, v_4) che pone $d[v_6] = 11$. E così via...

Comprendiamo che l'ordine con cui consideriamo gli archi nella selezione e quindi rispetto a quale applichiamo l'operatore di *Relax* influisce sul numero di operazioni di *Relax* e quindi sulla complessità della soluzione. In questo specifico caso, per come sono stati scelti i pesi degli archi e la loro disposizione porta ad un ordine di crescita è $O\left(2^{\frac{|V|}{2}}\right)$. Questo approccio è quindi un approccio a forza bruta che mi porta a rivisitare gli archi più volte. In generale la complessità è $O\left(2^{|V|}\right)$.

Algoritmo per DAG

Supponiamo di avere un qualsiasi DAG, vogliamo determinare il minimo percorso. Se gli archi sono unitari posso usare *BFS*, trattiamo ora un caso in cui il peso degli archi è arbitrario. Se ho un DAG, posso pensare di effettuare un ordinamento topologico e successivamente applicare il seguente ragionamento. In primo luogo, definiamo la seguente funzione di relax.

L'obiettivo è calcolare il minimo percorso da un nodo sorgente a tutti gli altri nodi. In primo luogo, osserviamo che ogni nodo è caratterizzato da un valore $d[u]$ che è proprio la distanza per andare dal nodo s al nodo u e da un valore $\pi(u) = \pi(u)$ che rappresenta il predecessore di u . Prima di effettuare la ricerca, si inizializzano i valori $d[u] = \infty$, tranne $d[s] = 0$. A questo punto per ogni nodo all'interno di V , si calcola $d[u] = \delta[s, u]$. Per ogni nodo, si valuta la funzione Relax per ogni arco entrante nel nodo e si considera il minimo tra le varie distanze.



Prendendo il grafo (ottenuto mediante TopologicalSort) posso osservare che i nodi b e s non vengono considerati. A partire da t , si applica l'operatore di relax e si ottiene che $d[t] = 2$ e il predecessore di t diventa s . A questo punto di passa al nodo x , in cui si applica l'operatore di relax per tutti i nodi entranti e quindi di ottiene che:

- $\text{Relax}(t, x) \rightarrow d[x] = d[t] + w(t, x) = 2 + 7 = 9$.
- $\text{Relax}(s, x) \rightarrow d[x] = d[s] + w(s, x) = 0 + 6 = 6$

E si sceglie il minimo tra i due e quindi $d[x] = 6$ e $\pi(x) = s$. Si passa ora al nodo y in cui abbiamo $\min\{\text{Relax}(y, x), \text{Relax}(y, t)\} = 5$. Quindi si effettua tale operazione per ogni nodo del grafo.

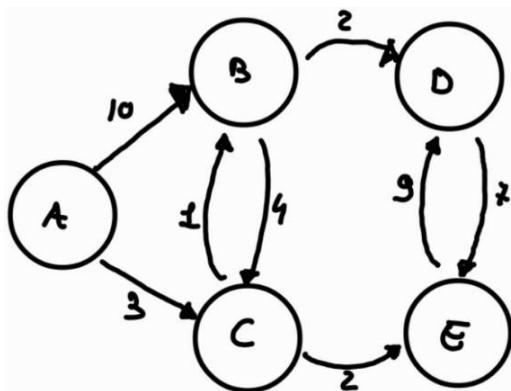
OSS:

- Funziona anche con archi a peso negativo.
- Questo è un approccio greedy.
- Devo fare prima l'ordinamento topologico per sapere a quali nodi applicare prima la relax.
- Questa soluzione è $\Theta(|V| + |E|)$: $|V|$ per l'inizializzazione e poi la somma degli indegree $|E|$.

Algoritmo di Dijkstra

```
Dijkstra(G,W,s):
    init(G,s);
    d[s]<-0;
    Q<-V[G];
    while(Q!=0):
        u<-extractMin(Q);
        S=S U {u};
        for each v in Adj[u]:
            relax(u,v,w);
```

L'inizializzazione prevede che tutti i nodi hanno distanza infinita da dal nodo s , e hanno predecessore NIL . Inserisco poi tutti i vertici del grafo nella coda. Ad ogni iterazione estraggo il minimo, e per ogni vertice adiacente al nodo estratto si fa una *Relax* (praticamente è una visita in ampiezza). L'estrazione del minimo è fatta in base al valore di distanza. Possiamo quindi osservare che ogni volta rilassa gli archi entranti; infatti, se v è adiacente ad u , allora l'arco è (u, v) e quindi va da u a v e quindi rilassiamo proprio la distanza di v .



Consideriamo l'esempio a sinistra. Inizialmente:

$$S = \{ \} \text{ e } Q = \{A, B, C, D, E\}$$

Alle successivamente iterazioni abbiamo:

- $S = \{A\}$ e $0, \infty, \infty, \infty, \infty$
- $S = \{A\}$ e $0, 10, 3, \infty, \infty$
- $S = \{A, C\}$ e $0, 7, 3, 11, 5$
- $S = \{A, C, E\}$ e $0, 7, 3, 11, 5$
- $S = \{A, C, E, B\}$ e $0, 7, 3, 9, 5$

Estratto il minimo, non lo estrarrò più. Inoltre, se ho archi negativi questo algoritmo non funziona e se ho cicli negativi allora il problema è non definito. Questa è una strategia greedy, dato che scelgo ogni volta il nodo con la distanza minima e non ci torno più. Per la complessità dobbiamo tenere in considerazione le seguenti complessità:

1. $\theta(|V|)$ per l'inizializzazione e per l'inserimento in Q .
2. $\theta(|V|)$ per l'estrazione del minimo. (potremmo pensare di usare una struttura ad Heap).
3. $\theta(|E|)$ per l'operazione di decrease_key e quindi il numero di volte che è invocata l'operazione di *Relax*.

Possiamo pensare di risparmiare sull'estrazione possiamo utilizzare varie strutture:

- Con una **coda con array** avrei che:
 - Extract-Min ha complessità $\theta(|V|)$
 - Decrease-Key $\theta(1)$.

In totale abbiamo una complessità $\theta(|V|^2 + |E|)$. Osserviamo che è $\theta(|V|^2)$ perché dobbiamo ripetere ad ogni estrazione, un operazione di Extract-Min

- Con un **heap binario** (min heap) abbiamo:
 - Extract-Min $\theta(\lg |V|)$, in realtà è costante ma poi si deve aggiustare la struttura.
 - Decrease-Key $\theta(\lg |V|)$.

In totale abbiamo una complessità $\theta(|V| \cdot \lg |V| + |E| \cdot \lg |V|)$.

- Con l'**heap di fibonacci**:
 - Extract-Min ha complessità $\theta(\lg |V|)$
 - Decrease-Key $\theta(1)$.

In totale abbiamo una complessità $\theta(|V| \lg |V| + |E|)$. Questa è la soluzione migliore che si può avere e discosta davvero di poco della ricerca con *BFS*.

Algoritmo di Bellman Ford

```
Bellman-Ford(G,W,s):  
    init(G,s);  
    for(i=1 to |V|-1):  
        for(each (u,v) in E):  
            Relax(u,v,w);  
    for each (u,v) in E:  
        if(d[v]>d[u]+w(u,v)):  
            report negative cycle;
```

In questo caso prendiamo gli archi $|V| - 1$ volte. Infatti, se il numero di archi è al più $|V| - 1$ allora vuol dire che il percorso è un **simple-path** e quindi non ha archi negativi.

Rispetto alla soluzione di DP che abbiamo analizzato, il calcolo del minimo viene fatto implicitamente sfruttando l'operatore di Relax.

In realtà anche in questo caso l'ordine in cui scegliamo gli archi influenza la complessità, ma nel caso peggiore abbiamo una complessità di $O(|V| \cdot |E|)$.

ANALISI AMMORTIZZATA DEI COSTI

Nell'analisi ammortizzata, il tempo richiesto per eseguire una sequenza di operazioni su una struttura dati è mediato su tutte le operazioni effettuate. Non è però un'analisi del caso medio di una operazione, ma un'analisi delle prestazioni medie nel caso peggiore. Vedremo tre metodi:

- **Metodo dell'aggregazione.**
- **Metodo degli accantonamenti.**
- **Metodo del potenziale.**

OSS: Eseguire l'analisi ammortizzata può fornire degli spunti per ottimizzare una struttura dati.

Metodo dell'aggregazione

Nell'analisi aggregata si mostra che, per ogni n , una sequenza di n operazioni impiega in totale, nel caso peggiore, un tempo $T(n)$; da cui il **costo ammortizzato di ciascuna operazione** è $T(n)/n$. Vediamo questo metodo con due esempi: Pila con operazione Multi-Pop e Contatore binario con operazione Increment.

Operazioni su pila (inizialmente vuota) dotata di Multipop

Multipop (S, k)

```
while not Stack-Empty(S) and k ≠ 0
    do Pop(S)
    k ← k-1
```

Questo metodo estrae (al più) k elementi dalla testa della pila; osserviamo infatti che la pila potrebbe avere meno di k elementi. Il costo di Multi-pop su uno stack di s elementi è il $\min(s, k)$, assumendo che Pop abbia un costo unitario.

Consideriamo una sequenza di n operazioni di Push, Pop, Multi-pop su una pila inizialmente vuota. Il costo di Multi-pop nel caso peggiore è $O(n)$ perché lo stack contiene al più n elementi. Il caso peggiore complessivamente si ha con n operazioni Multi-pop da cui il costo è $O(n^2)$. Il fatto è che non posso avere ad ogni chiamata Multi-pop una complessità $O(n)$ e non posso avere n chiamate di Multi-pop. Ad ogni chiamata la pila si svuota e quindi devo considerare il nuovo stato dello stack; infatti, ogni elemento può essere estratto al massimo una volta per ogni volta che viene inserito. Dato che possono essere inseriti al massimo n elementi (ognuno una sola volta), possono essere estratti al massimo n elementi. Dunque, il costo complessivo nel caso peggiore è $O(n)$ e il costo medio di un'operazione, nel caso peggiore, è $\frac{O(n)}{n} = O(1)$. Da cui il costo ammortizzato di ciascuna operazione è $O(1)$.

Operazioni Increment su contatore binario (inizialmente nullo)

Increment (A)

```
i ← 0
while i < length[A] and A[i]=1
    do A[i] ← 0
    i ← i+1
if i < length[A]
    then A[i] ← 1
```

Consideriamo un contatore binario (a crescere) su k bit: contatore modulo 2^k . Il costo di Increment è legato al numero di bit che vengono invertiti. Il costo di Increment nel caso peggiore si ha quando vengono invertiti tutti i bit è $\Theta(k)$. Quando il contatore si resetta, il costo nel caso peggiore di n operazioni di Increment (su un contatore inizialmente nullo) è di $O(nk)$. Ancora una volta possiamo ottenere un bound più stretto. Il punto è che non tutti i bit vengono sempre invertiti infatti: $A[0]$ è invertito tutte le volte, $A[1]$ è invertito la metà delle volte, $A[2]$ è invertito un quarto delle volte. Il numero di bit invertiti durante n operazioni di Increment è: $\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$. Quindi il tempo complessivo nel caso peggiore è $O(n)$ e il costo medio di ciascun incremento è $\frac{O(n)}{n} = O(1)$.

Metodo dell'accantonamento

Si assegna a ciascuna operazione un (diverso) costo ammortizzato, che può differire dal costo reale dell'operazione.

- Se il costo ammortizzato supera quello reale, la differenza viene mantenuta come credito.
- Il credito viene consumato quando il costo ammortizzato è inferiore a quello reale.
- Il vincolo da rispettare è che il costo ammortizzato totale di una qualunque sequenza di operazioni deve essere un limite superiore per il costo reale totale. Quindi il credito non deve mai essere negativo.

Operazioni su pila (inizialmente vuota) dotata di Multipop

Poniamo:

- Il costo reale: $push(1)$, $pop(1)$, $multipop(min(k, s))$
- Il costo ammortizzato sarà: $push(2)$, $pop(0)$, $multipop \in (0)$

Interpretazione come pila di piatti

- Per ogni operazione di push "riceviamo" due monete, una la usiamo per pagare il costo reale di push, l'altra la lasciamo nel piatto.
- Le monete presenti nei piatti rappresentano il credito.
- La moneta presente in ciascun piatto serve a ripagare il costo reale di pop o multipop.

Il credito non è mai negativo (per fare un pop devo prima fare un push) \Rightarrow il costo reale totale non supera mai il costo ammortizzato totale (che vale $O(n)$ per n operazioni). Il costo reale totale è quindi $O(n)$.

OSS: Vado a differenziare sulle singole operazioni.

Operazioni Increment su contatore binario (inizialmente nullo).

- Costo reale: inversione di un bit (1)
- Costo ammortizzato: bit set (2), bit reset (0)

Per ogni operazione di set "riceviamo" due monete, una la usiamo per pagare il costo reale dell'operazione, l'altra la conserviamo per pagare la successiva operazione di reset.

In ogni istante, ogni bit pari a uno ha una moneta associata.

Il credito non è mai negativo \Rightarrow il costo reale totale non supera mai il costo ammortizzato totale. Il costo ammortizzato di Increment è 2 (setta a uno al più un bit). Il costo reale totale è quindi $O(n)$.

Metodo del Potenziale

Invece di utilizzare un credito associato ai singoli elementi, utilizza il concetto di “energia potenziale” o “potenziale” associato all’intera struttura dati. Quindi date n operazioni indicate con $i = 1, 2, \dots, n$ e costo reale c_i ; allora possiamo distinguere:

- D_0 : struttura dati nello stato iniziale.
- D_i : struttura dati dopo l’i-esima operazione.
- D_n : struttura dati dopo le n operazioni.

Denotiamo ora con $\phi(D_i)$ una funzione che fa corrispondere a D_i un numero reale. Allora il costo ammortizzato della i-esima operazione (c'_i) rispetto alla funzione potenziale è dato da:

$$c'_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

Quindi il costo ammortizzato di ciascuna operazione è il costo reale più l’incremento di potenziale dovuto all’operazione che modifica la struttura dati. Possiamo ora scrivere il costo ammortizzato totale delle n operazioni come la somma dei costi ammortizzati delle singole operazioni, e quindi:

$$\sum_{i=1}^n c'_i = \sum_{i=1}^n [c_i + \phi(D_i) - \phi(D_{i-1})] = \text{serie telescopica} = \sum_{i=1}^n [c_i] + \phi(D_n) - \phi(D_0)$$

Se $\phi: \phi(D_n) \geq \phi(D_0)$ allora il costo ammortizzato totale è un limite superiore per il costo reale totale. Osserviamo che nella pratica non è sempre noto.

Ciò che possiamo fare è richiedere che $\phi(D_i) \geq \phi(D_0) \forall i$.

- In maniera analoga al metodo dell’accantonamento.
- Ad esempio possiamo porre $\phi(D_0) = 0$ e facendo in modo che $\phi(D_i) \geq 0$.

Operazioni su pila (inizialmente vuota) dotata di Multi-pop

Definiamo $\phi(D_i)$ come il numero di elementi nella pila:

- $\phi(D_0) = 0$ e $\phi(D_i) \geq 0$ (il numero di elementi non è mai negativo)

Il costo ammortizzato di push su stack di s elementi è:

$$\bullet \quad c'_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (s+1) - s = 2$$

Costo ammortizzato di $multipop(k)$ su stack di s elementi

$$\bullet \quad c'_i = c_i + \phi(D_i) - \phi(D_{i-1}) = \min(s, k) - \min(s, k) = 0$$

Costo ammortizzato di pop su stack di s elementi:

$$\bullet \quad c'_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + (-1) = 0$$

Da cui:

- Il costo ammortizzato di ogni operazioni è $O(1)$.
- Il costo ammortizzato totale di n operazioni è $O(n)$.

Operazioni Increment su contatore binario (inizialmente nullo)

Definiamo $\phi(D_i)$ pari a b_i , il numero di bit 1 nel contatore dopo l'operazione i . Quindi:

- $\phi(D_0) = 0$ e $\phi(D_i) \geq 0$ e quindi il numero di elementi non è mai negativo.

Se indichiamo con t_i il numero di bit resettati nell'operazione i allora il costo c_i è al più $t_i + 1$ (se settiamo un bit a 1).

- Se $b_i = 0$ l'operazione i ha resettato tutti i k bit $\Rightarrow b_{i-1} = k$ e $t_i = k$
- Se $b_i > 0$ allora $b_i = b_{i-1} - t_i + 1$

OSS: In entrambi i casi $b_i \leq b_{i-1} - t_i + 1$ e la differenza di potenziale è:

- $\phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$

Quindi $c_i = c_i + \phi(D_i) - \phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2 = O(1)$ e il costo ammortizzato totale di n operazioni è $O(n)$.

Consideriamo il caso in cui il contatore non parte da zero; inizialmente ci sono b_0 bit pari a 1, alla fine avremo b_n bit pari a 1. Dato che $b_0 \leq k$, se $k = O(n)$ allora il costo reale totale è $O(n)$. In altre parole, se eseguiamo $n = \Omega(k)$ operazioni di incremento, il costo reale totale è $O(n)$, indipendentemente dal valore iniziale del contatore.

ALBERI AUTO-AGGIUSTANTI

Gli alberi auto-aggiustanti (**splay trees**) sono alberi binari di ricerca che non impongono ulteriori vincoli sulla disposizione degli elementi ma che ogni volta che si accede ad un elemento dell'albero, l'albero viene riaggiustato in modo che tale elemento viene posto nella radice.

Utilizzando delle rotazioni per riaggiustare l'albero, si ottengono tempi di esecuzione ammortizzati di tipo logaritmico su una sequenza di n operazioni.

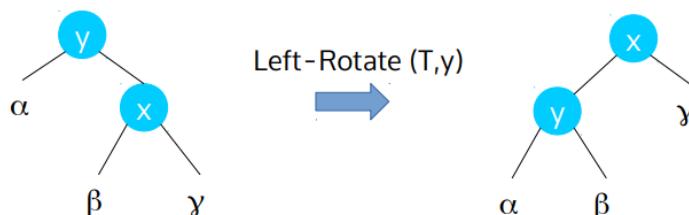
Il vantaggio pratico è che se gli elementi cui si accede più di frequente si trovano nella vicinanza della radice si accede più velocemente ad essi (stesso principio del caching).

Operazione Splay

L'operazione **splay** su un nodo x consiste nell'effettuare una serie di rotazioni che portano x nella radice dell'albero (sono le rotazioni a garantire la proprietà di BST). Trattiamo vari casi:

Caso 1:

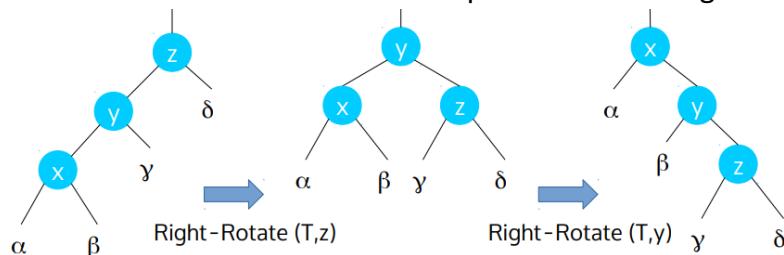
Il nodo x è figlio della radice.



In questo caso è fare una rotazione sulla radice a destra (se è figlio sinistro) o a sinistra (se l'elemento è figlio destro).

Caso 2:

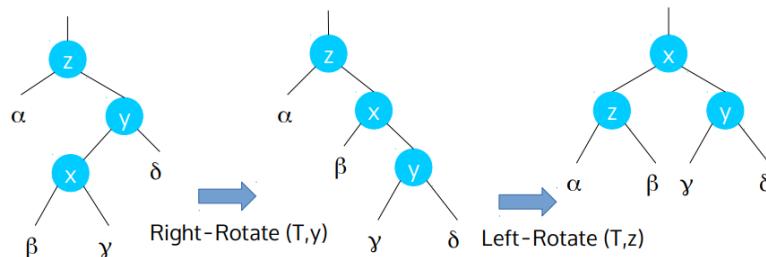
Il nodo x ha un nonno e sia x che il padre di x sono figli di sinistra (o di destra) del proprio genitore.



In questo caso possiamo fare due rotazioni a destra (se sono figli sinistri) o due rotazioni a sinistra (se entrambi x e $p[x]$ sono figli destri).

Caso 3:

Il nodo x ha un nonno e x è figlio di sinistra (o di destra) e il padre di x è figlio di destra (o sinistra).



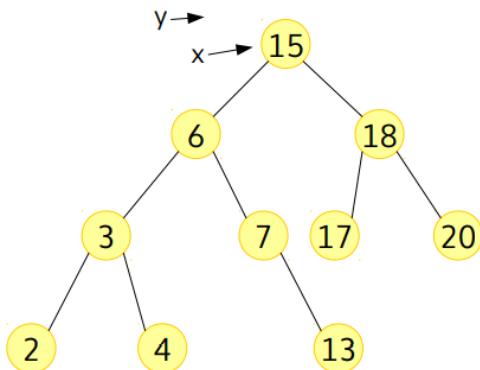
Supponiamo di trovarci nel caso in cui:

- $p[x] = r[p[p[x]]]$
- $x = [p[x]]$

In questo caso si fa prima una rotazione a destra su $p[x]$ e poi una rotazione a sinistra su x . Il caso speculare prevede operazioni duali.

Operazione di Ricerca

- Se l'albero è vuoto, non si fa nulla

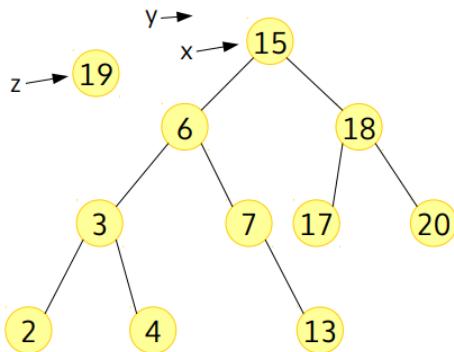


```

Splay-Tree-Search (T,k)
x ← root[T]
while x≠NIL and key[x]≠k
  do y ← x
  if k < key[x]
    then x ← left[x]
    else x ← right[x]
  if x≠NIL
    then Splay (T,x)
  else if root[T]≠NIL
    then Splay (T,y)
return x
  
```

Dopo aver cercato una chiave, si esegue un'operazione splay sul nodo che contiene la chiave o sulla foglia su cui la ricerca è terminata (sempre per validare il principio di località). Se l'albero è vuoto ovviamente non si fa nulla.

Inserimento di un Elemento

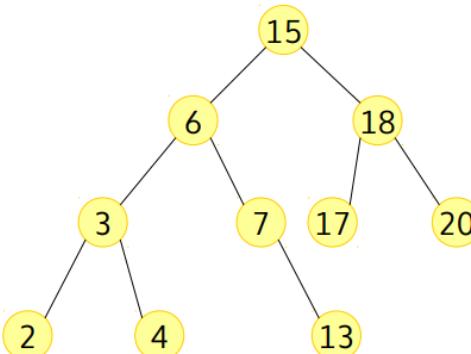


```

Splay-Tree-Insert (T,z)
Tree-Insert (z)
Splay (T,z)
  
```

Dopo aver inserito un elemento, si esegue un'operazione splay sul nodo che contiene il nuovo elemento inserito.

Eliminazione



```

Splay-Tree-Delete (T,z)
Tree-Delete (z)
if p[z]≠NIL
  then Splay (T,p[z])
  
```

Dopo aver eliminato un nodo z, si esegue un'operazione splay sul nodo che era il padre di z prima dell'eliminazione di z. Se il padre di z è NIL (z è la radice) non si fa nulla

Analisi ammortizzata

Utilizziamo il metodo del potenziale per calcolare il costo ammortizzato di una operazione di splay e di una sequenza di operazioni (search, insert, delete).

La funzione potenziale è $\phi(T) = \sum_{x \in T} r(x)$.

Dove il rango del nodo x è il logaritmo del numero di discendenti di x : $r(x) = \lg d(x)$. Più l'albero è bilanciato, più basso è il potenziale (tiene conto del bilanciamento).

Un albero con un solo nodo ha un potenziale pari a 0. Se consideriamo di iniziare una sequenza di operazioni (search, insert, delete) su un albero T_0 con un solo nodo, avremo $\phi(T_0) = 0$.

$\phi(T_i) \geq \phi(T_0)$ dato che non si esegue l'operazione splay su un albero vuoto.

OSS: Vediamo alcuni teoremi che ci aiuteranno nell'analisi.

TH1: Se x è padre di y e z , allora: $r(x) > 1 + \min\{r(y), r(z)\}$

Possiamo porre $d(x) = d(y) + d(z) + 1$, ovvero se y e z sono i figli di x , allora il numero di discendenti di x è dato dalla somma del numero di discendenti dei figli a cui aggiungiamo il nodo stesso (+1).

Osserviamo che: $d(y) + d(z) \geq 2 \cdot \min\{d(y), d(z)\} \Rightarrow d(x) \geq 2 \cdot \min\{d(y), d(z)\} + 1$ e quindi: $d(x) > 2 \cdot \min\{d(y), d(z)\}$

Valutiamo ora $r(x) = \lg d(x) > \lg(2 \cdot \min\{d(y), d(z)\})$. Questo perché il logaritmo è una funzione crescente. Da cui: $r(x) > \lg 2 + \lg(\min\{d(y), d(z)\})$. Considerando sempre il fatto che il logaritmo è una funzione crescente possiamo scrivere che: $r(x) > 1 + \min\{\lg d(y), \lg d(z)\}$
E quindi la tesi è vera.

Forniamo un limite superiore per la variazione di potenziale legata ad un passo dell'operazione splay (sale sempre verso l'alto).

TH2: Se i è un passo di un'operazione splay sul nodo x , allora:

1. $r_i(x) \geq r_{i-1}(x)$
2. Se $p[x]$ è radice, $\phi(T_i) - \phi(T_{i-1}) < r_i(x) - r_{i-1}(x)$
3. Se $p[x]$ non è radice, $\phi(T_i) - \phi(T_{i-1}) < 3(r_i(x) - r_{i-1}(x)) - 1$

In primo luogo, definiamo la variazione del potenziale come: $\Delta\phi_i = \phi_i(T_i) - \phi(T_{i-1})$.

Punto 1. La prima affermazione è banale dato che un passo di un'operazione di splay consiste in una o più rotazioni che fanno salire x verso l'alto e quindi aumenta il numero di discendenti di x .

Punto 2. Se x è la radice al passo i , vuol dire che al passo precedente era figlio (destro o sinistro) di un nodo y (che era la radice) che ora è a sua volta figlio (sinistro o destro) di x per via della rotazione. La rotazione fa in modo che solo x e y cambiano di rango e quindi possiamo valutare $\Delta\phi_i$ considerando solo i contributi di x e y . In base a quanto detto: $r_i(x) = r_{i-1}(y)$. Possiamo valutare: $\Delta\phi_i = (r_i(x) - r_{i-1}(x)) + (r_i(y) - r_{i-1}(y)) = r_i(y) - r_{i-1}(x) < r_i(x) - r_{i-1}(x)$ dato che x è padre di y .

Punto 3. Stiamo ipotizzando che x non è un nodo radice, e quindi al passo $i - 1$ aveva un padre e un nonno e ha preso il posto del nonno. Per via delle rotazioni possiamo osservare che indipendentemente dal caso di splay in cui ci troviamo, gli unici nodi che cambiano di rango sono $x, p[x] = y, p[y] = z$. Possiamo quindi valutare : $\Delta\phi_i = (r_i(x) - r_{i-1}(x)) + (r_i(y) - r_{i-1}(y)) + (r_i(z) - r_{i-1}(z))$. Possiamo osservare che al passo i , $r_i(y) < r_i(x)$ dato che x è risalito e al passo precedente $r_{i-1}(y) > r_{i-1}(x) \Rightarrow r_i(y) - r_{i-1}(y) < r_i(x) - r_{i-1}(x)$ e quindi: $\Delta\phi_i < 2(r_i(x) - r_{i-1}(x)) + (r_i(z) - r_{i-1}(z))$.

Sfruttando il TH1 possiamo osservare che passando dal passo $i - 1$ al passo i , z è sceso nella gerarchia (due volte) e x ha preso il suo posto quindi $r_i(x) = r_{i-1}(z) > 1 + \min\{r_{i-1}(x), r_i(z)\}$ quindi: $r_{i-1}(z) > 1 + r_i(z)$. In realtà in modo duale, x è salito di due posizioni e quindi $r_i(x) > 1 + r_{i-1}(x)$. Possiamo scrivere che essendo $r_i(z) < r_{i-1}(z)$; allora:

$$r_i(z) - r_{i-1}(z) < r_i(x) - r_{i-1}(x) - 1$$

$$\text{Infine: } \Delta\phi_i < 3(r_i(x) - r_{i-1}(x)) - 1$$

Analisi Ammortizzata

Ora vediamo effettivamente l'analisi ammortizzata dei costi:

Il costo ammortizzato di una operazione splay in un albero auto-aggiustante con n nodi è $O(\lg n)$, il costo reale di un passo dell'operazione splay $c_i = O(1)$ da cui il costo ammortizzato di un passo dell'operazione splay su x è: $c'_i = c_i + \Delta\phi_i$. Se x è figlio della radice allora: $c'_i < r_i(x) - r_{i-1}(x) + O(1)$ altrimenti $c'_i < O(1) + 3(r_i(x) - r_{i-1}(x)) - 1 < O(1) + 3(r_i(x) - r_{i-1}(x))$

Supponiamo l'operazione splay consista di k passi, allora:

$$\sum_{i=1}^k c'_i < O(1) + 3 \sum_{i=1}^k (r_i(x) - r_{i-1}(x)) = O(1) + 3(r_k(x) - r_0(x))$$

Il costo ammortizzato è inferiore a $O(1) + 3(r_k(x) - r_0(x))$. Alla fine dell'operazione splay, x è la radice ed ha n discendenti, quindi $r_k(x) = \lg n$ e quindi il costo ammortizzato è $O(\lg n)$. Si può provare inoltre il seguente teorema:

Il tempo ammortizzato di una sequenza di m operazioni di insert, search e delete in un albero auto-aggiustante è $O(m \cdot \lg n)$ dove n è il numero massimo di nodi dell'albero durante le operazioni. Quindi il costo ammortizzato di una singola operazione è $O(\lg n)$.