Modello RAM

RAM = Random Access Machine, macchina ad accesso diretto. Questo modello è una schematizzazione ad altissimo livello del calcolatore ed è composto da:

- **Processore**, con
  - Unità Aritmetico Logica (**ALU**)
  - Due **registri**:
    - **Accumulatore**, per eseguire le operazioni
    - **Contatore di programma**, program counter, contenente l'indirizzo della prossima istruzione
- **Memoria** illimitata: tutti i dati si possono **memorizzare nella memoria centrale** senza appoggiarsi a dischi esterni.

Istruzioni elementari:

- Operazioni **aritmetiche** (+, -, *, /)
- Operazioni **logiche** (&&, ||, !)
- Operazioni di **confronto** (<, >, =)
- Operazioni di **trasferimento** (**dalla memoria centrale all'accumulatore** e viceversa)
- Operazioni di **controllo** (chiamate di funzioni, return, ecc...)

Si assume che **tutte le istruzioni elementari richiedano un tempo di esecuzione costante**.

Costo computazionale degli algoritmi

- Costo in **tempo**: proporzionale al **numero di istruzioni elementari**
- Costo in **spazio**: **numero di celle di memoria usate durante l'esecuzione** (senza contare le celle dei dati in input)

Si punta prima ad ottimizzare il tempo, dopodiché allo spazio: questo perché il tempo non si recupera mentre lo spazio si.

I costi si esprimono come **funzioni matematiche nella dimensione dei dati di input** (detta **istanza di input**), cioè nel **numero di bit necessari a memorizzare l'input** (o una misura equivalente).

Interessante la valutazione del tasso di crescita, cioè **come cambiano i costi al cambiare della dimensione dell'input** (**analisi asintotica**).

Vengono analizzati i casi **pessimo**, **medio** e **ottimo**. Si considerano le istanze di input I con |I| = dimensione dell'input e A algoritmo in analisi.

Pessimo = $\max_{I,\ |I| = n} \{T_A(n)\}$, cioè il massimo tempo impiegato su dimensione n, quindi **limite superiore**.

Medio = costo medio su istanze di dimensione n.

Ottimo = $\min_{I,\ |I| = n} \{T_A(n)\}$

Limiti inferiori di complessità

P problema, $L_P(n)$ è limite inferiore per P se ogni algoritmo A che risolve P è tale che $T_A(n) = \Omega(L_P(n))$, quindi sono le operazioni minime necessarie per risolvere P nel caso peggiore.

$T_A(n)$ qualsiasi, se A risolve P, è un limite superiore alla complessità. Qualunque strategia di soluzione ha un numero di soluzioni **sufficienti** a risolvere P, che possono essere **più di quante siano necessarie**.

$T_A(n) \geq L(n) \Leftrightarrow T_A(n) = \Omega(L(n))$

Se $T_A(n) = \theta(L(n))$ allora **A è ottimo**

Tecniche per trovare i limiti inferiori

- **Tecnica della dimensione dell'input**
  Se il problema P richiede l'esame di tutti i dati dell'input
  $\Rightarrow L_P(n) = \Omega(n)$
- **Tecnica dell'albero di decisione**
  Problemi risolvibili tramite sequenze di decisioni o confronti
  AdD: struttura matematica usata per **rappresentare algoritmi che risolvono tramite confronti** senza assunzioni sui dati.
  Il **costo in tempo** e il **numero di contronti** sono dello **stesso ordine di grandezza**
  Proprietà:
  - Nodi interni = **confronti**
  - Foglie = **soluzioni**, quindi $\forall$ AdD su problema P $\Rightarrow$ #foglie $\geq$ #soluzioni
  - Cammino radice-foglia = **esecuzione** dell'algoritmo su una particolare istanza
  - Lunghezza cammino = **numero di confronti** di una particolare esecuzione $\Rightarrow$ costo in tempo
  - Altezza = **confronti al caso pessimo**, $h(AdD_A) \Leftrightarrow$ costo in tempo al caso pessimo di A

  Un limite inferiore per rl'altezza di un generico AdD relativo a P equivale ad un **limite inferiore per il numero di confronti necessari a risolvere P**.
  Nel caso del problema dell'ordinamento si hanno n! possibili soluzioni $\Leftrightarrow$ n! foglie minimo
  con i confronti un generico algoritmo discrimina 2^i casi. Per essere corretto i deve essere tale che $2^i \geq n! \Leftrightarrow i \geq \log_2(n!)$
  $\Rightarrow$ **$L(n) = \Omega(\log(s(n)))$**

***Teorema*: qualsiasi algoritmo di ordinamento per confronti, senza informazioni sui dati da ordinare, richiede $\Omega(n \log n)$ confronti al caso pessimo**

Dim: dimostro **cercando un limite inferiore per l'altezza di un AdD** dove ogni permutazione compare in una foglia.

T = generico AdD con F foglie, quindi F >= n! e l'AdD è un **albero binario completo** ABC $\Leftarrow$DEF$\Rightarrow$ ogni nodo interno ha esattamente due figli.

Un ABC di altezza h **ha al più 2^h foglie**

Dim: per induzione su h

Base: h = 0, #foglie = $2^0 = 1 = 2^h$ OK

Ipotesi induttiva: $\forall$ ABC di altezza $k \leq h-1 \Rightarrow$ #foglie $\leq 2^k$

Passo: T è ABC con altezza h, f = #foglie di T, f_{sx} = #foglie di Tsx e f_{dx} = #foglie di Tdx

Quindi $f = f_{sx} + f_{dx}$, e l'altezza massima di Tsx e Tdx è h - 1

$\Rightarrow$ per ip. ind. $f_{sx} + f_{dx} \leq 2^{h-1} + 2^{h-1} = 2^h$ **CVD**

$\Rightarrow f \leq 2^h$, quindi h = #confronti al caso pessimo

$n! \leq f \leq 2^h \Leftrightarrow n! \leq 2^h \Leftrightarrow h \geq \log n!$ **limite inferiore**

$n! = n(n-1)(n-2)...n/2\ (n/2 - 1)\ ...\ 1 > (n/2)^{n/2}(1)^{n/2} = (n/2)^{n/2}$

$L(n) = h \geq \log n! > \log(n/2)^{n/2} = n/2 \log(n/2) = \Omega(n \log n)$ **CVD**

Teoria della calcolabilità
Questione fondamentale circa la **potenza** e le **limitazioni** dei sistemi di calcolo. Esplora concetti di:

- computazione
- algoritmo
- problema risolvibile per via algoritmica

Dimostra che esistono problemi che non ammettono un algoritmo di soluzione: **problemi non decidibili**.
Problemi computazionali: formulati matematicamente e di cui cerchiamo una soluzione algoritmica.
Possono essere:

- non decidibili
- decidibili
    - trattabili (costo polinomiale)
    - intrattabili (costo esponenziale)

Calcolabilità ⇒ algoritmo e problema non decidibile (risolvibili e non)
Complessità ⇒ algoritmo efficiente e problema intrattabile (facili e difficili)
Qualsiasi modello si scelga (astratto, RAM, PC, …), gli algoritmi devono essere **descritti da sequenze finite di caratteri di un alfabeto finito**, quindi sono infiniti ma numerabili.
I problemi computazionali (funzioni matematiche che associano ai dati il rispettivo risultato) **non sono numerabili**.
|{Algoritmi}| << |{Problemi}| ⇒ ∃ problemi privi di un algoritmo risolvente
Esistono quindi problemi non calcolabili, quelli che si presentano spontaneamente sono tutti calcolabili.

Problema dell'arresto, Turing 1930
Presi ad arbitrio un algoritmo A ed i suoi dati in input D, decidere **in tempo finito** se la computazione di A su D temina o no (va in loop).

- Algoritmo che indaga sulle proprietà di un altro algoritmo (trattato come input)
- Legittimo: stesso alfabeto per codificare algoritmi e i loro dati di ingresso
- Una stessa sequenza di simboli può quindi essere interpretata sia come un programma che come un input per un altro programma

Un algoritmo A può operare sulla rappresentazione di un altro algoritmo B
⇒ possiamo calcolare A(B), può avere senso calcolare A(A)

***Teorema*: il problema dell'arresto è indecidibile**
Dim: se fosse decidibile allora esisterebbe un algoritmo arresto che presi A e D in input determina in tempo finito:

- arresto(A, D) = 1 se A(D) termina
- arresto(A, D) = 0 se A(D) non termina

Oss: arresto non può consistere in un algoritmo che simuli A(D) perché se A(D) non termina allora arresto non finirebbe in tempo finito.
Scegliamo D = A, arresto(A, A) = 1 quindi A(A) non termina
Quindi se esistesse arresto esisterebbe anche

```
paradosso(A) {
        while(arresto(A, A)) {
        }
}
```

L'ispezione di paradosso mostra che paradosso(A) termina se e solo se arresto(A, A) = 0 cioè A(A) non termina.
Calcolando paradosso(paradosso) esso termina se e solo se arresto(paradosso, paradosso) = 0 quindi se paradosso(paradosso) non termina.
Si ha una **contraddizione**, quindi paradosso non esiste, quindi arresto non esiste quindi è indecidibile.
L'algoritmo arresto costituirebbe uno strumento molto potente per dismostrare congettura ancora aperte.

Tesi di Church-Turing: la decidibilità è una proprietà del problema. Incrementi qualitativi a macchine o linguaggi servono solo a renderle più efficienti o facili da usare.
Non è dimostrabile.

***Teorema*: LCS ha una sottostruttura ottima**

LCS stringa costruibile a partire dalle LCS delle sottostringhe

Siano $X = x_1...x_m$, $Y = y_1...y_n$ due stringhe

Sia $Z = z_1...z_k$ una LCS di X e Y, cioè Z = LCS(X, Y)

- $x_m = y_n$ allora $z_k = x_m$ e $Z_{k.1} = LCS(X_{m-1}, Y_{n-1})$
  Cioè se due stringhe terminano con lo stesso carattere allora anche la LCS terminerà con quel carattere.
- x_m != y_n allora (z_k != x_m allora $Z_{k-1} = LCS(X_{m-1}, Y)$)
  z_k potrebbe essere y_n
- x_m != y_n allora (z_k != y_n allora $Z_{k-1} = LCS(X, Y_{n-1})$)
  z_k potrebbe essere x_m

Dim: 1 per assurdo z_k != x_m allora $W = Zx_m$ quindi W = CS(X, Y) ma |W| = k + 1 però Z = LCS(X, Y) con |Z| = k per assurdo.

Dimostro che $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$. Certamente $Z = CS(X_{m-1}, Y_{n-1})$ ma, per assurdo, $Z != LCS(X_{m-1}, Y_{n-1})$

$|Z_{k-1}| = k - 1$ quindi esiste W con $|W| > k - 1$ | $W = LCS(X_{m-1}, Y_{n-1})$

Quindi Wx_m, $|Wx_m| = |W| + 1 > (k - 1) + 1 = k$ e W = CS(X, Y)

Contraddizione con Z = LCS(X, Y) e |Z| = k

Quindi se due stringhe terminano con lo stesso carattere allora ogni LCS finisce con quel carattere.

2 $Z = z_1...z_k = CS(X_{m-1}, Y)$

Dimostro che Z è anche $LCS(X_{m-1}, Y)$. Per assurdo esiste W | $W = CS(X_{m-1}, Y)$ e $|W| > |Z|$.

Ma allora $W = CS(X_{m-1}, Y) = CS(X, Y)$, contraddizione perché $|W| > |Z|$ e Z = LCS(X, Y)

3 analogo al 2

Concludiamo che una LCS(X, Y) contiene una LCS dei loro prefissi, cioè sottostruttura ottima.

CVD

Teorema dell'esperto

***Teorema*: date le costanti $a \geq 1$ e $b > 1$ e la funzione f(n) (non negativa perché esprime un costo), sia T(n) una funzione definita sugli interi non negativi dalla ricorrenza**

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

dove n/b rappresenta la parte intera superiore o inferiore a seconda dei casi.

Allora T(n) può essere asintoticamente limitata nei seguenti modi

- $f(n) = O(n^{\log_b(a) - \varepsilon})$ per qualche costante $\varepsilon > 0 \Rightarrow T(n) = \theta(n^{\log_b(a)})$
- $f(n) = \theta(n^{\log_b(a)}) \Rightarrow T(n) = \theta(n^{\log_b(a)} \cdot \log n)$
- $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$ per qualche costante $\varepsilon > 0$ e $af(n/b) \leq cf(n/b)$ per qualche costante $c < 1$ (**condizione di regolarità**) e n sufficientemente grande $\Rightarrow T(n) = \theta(f(n))$

In pratica **confrontiamo la funzione con $n^{\log_b(a)}$ e la soluzione è la più grande rispetto ad un fattore $n^{\pm\varepsilon}$.**

Dim: per potenze esatte di b, $n = b^j$ con $j \in \mathbb{N}$

T(n) = aT(n/b) + f(n) applichiamo la definizione ricorsiva

$T(n) = a[aT(1/b \cdot n/b) + f(n/b)]+f(n) = a^2T(n/b^2) + af(n/b) + f(n) = \ldots = a^iT(n/b^i) + \sum_{j = n...i-1}(a^jf(n/b^j))$

Poniamo $i = \log_b n$

$T(n) = a^{\log_b(n)}T(n/b^{\log_b(n)}) + \sum_{j = n...\log_b(n)-1}(a^jf(n/b^j)) = n^{\log_b(a)}T(1) + \sum_{j = n...\log_b(n)-1}(a^jf(n/b^j))$ [= g(n)]

Poiché $T(1) = \theta(1)$ otteniamo:

$T(n) = \theta(n^{\log_b(a)}) + g(n)$ dove

- $\theta(n^{\log_b(a)})$ è il contributo per la soluzione diretta dei sottoproblemi di dimensione 1
- g(n) è il costo globale della divisione e ricombinazione (parte ricorsiva)

Ora vediamo i casi del teorema:

1. $f(n) = O(n^{\log_b(a) - \varepsilon})$ con $\varepsilon > 0 \Rightarrow g(n) = O(n^{\log_b(a)})$ e si ha
   $T(n) = \theta(n^{\log_b(a)}) + O(n^{\log_b(a)}) \Rightarrow T(n) = \theta(n^{\log_b(a)})$
2. $f(n) = \theta(n^{\log_b(a)}) \Rightarrow g(n) = \theta(n^{\log_b(a)}\log(n))$ e si ha
   $T(n) = \theta(n^{\log_b(a)}) + \theta(n^{\log_b(a)}\log(n)) \Rightarrow T(n) = \theta(n^{\log_b(a)}\log(n))$
3. $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$ con $\varepsilon > 0$ e $af(n/b) \leq cf(n)$ (per $c < 1$) $\Rightarrow g(n) = \theta(f(n))$ allora
   $T(n) = \theta(n^{\log_b(a)}) + \theta(f(n))$ e, poiché $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$, allora $T(n) = \theta(f(n))$

CVD

Paradigma Divide et Impera

- **Dividere** il problema da risolvere in 2 o più sottoproblemi analoghi che operano su input di dimensioni inferiori
- **Ricorsione**: risolvere i sottoproblemi ricorsivamente o direttamente se gli input sono sufficientemente piccoli
- **Combinare** le soluzioni ai sottoproblemi per ottenere la soluzione del problema originale

$T(n) = \theta(1)$ se $n \leq n_0$, $D(n) + T(n_1) + \ldots + T(n_a) + C(n)$ se $n > n_0$
Con D(n) costo della **divisione**, **fuori dalla chiamata ricorsiva**, $T(n_i)$ costo della **ricorsione** e C(n) costo della **combinazione**, anche questa **fuori dalla ricorsione**.
La divisione è spesso eseguita su divisioni costanti n/b, quindi i sottoproblemi sono sempre su istante di dimensione n/b.
Da questo:
$T(n) = \theta(1)$ se $n \leq n_0$, $D(n) + aT(n/b) + C(n)$ se $n > n_0$

Usato in contesti in cui il problema è ricorsivo per natura e **i sottoproblemi sono tutti indipendenti** tra loro, operano su **insiemi disgiunti di dati** e ognuno è **incontrato una sola volta**.
Su questi algoritmi D&I la correttezza è dimostrata tramite l'**induzione**:
Caso base: **direttamente**, dimostrando per ispezione del codice che su istanze sufficientemente piccola la ricorsione termina. In pratica si **controllano i casi base della ricorsione**.
Ipotesi induttiva: l'algoritmo **risolve correttamente i sottoproblemi**.
Passo: si verifica la **correttezza della fase di ricombinazione**. Si dimostra la correttezza dell'algoritmo su problemi generali esaminando la combinazione dei risultati parziali.

Programmazione Dinamica

- **Definizione dei sottoproblemi** e dimensionamento della tabella
- **Soluzione diretta dei sottoproblemi elementari** e scrittura nella tabella
- **Definizione della regola ricorsiva** per ottenere la soluzione di un problema a partire dalle soluzioni dei sottoproblemi già risolti (**regola di riempimento della tabella**)
- **Restituzione del risultato**

Il paradigma di Programmazione Dinamica, alternativo al Divide et Impera, viene usato in contesi in cui il problema è **per definizione ricorsivo** ma il D&I **risulta inefficiente** perché **i sottoproblemi non sono indipendenti** tra loro, e ci sarebbero chiamate ripetute sugli stessi sottoproblemi.
L'idea principale è di **risolvere un sottoproblema e scrivere la soluzione in una tabella** che verrà consultata quando il sottoproblema verrà reincontrato.
Un problema per cui è ideale usare la programmazione dinamica deve godere di una **sottostruttura ottima**, cioè **la soluzione ottima del problema deve derivare dalla soluzione ottima dei suoi sottoproblemi**.

***Teorema*: ogni ABC ha esattamente n-1 nodi interni**
Dim: per induzione su n
Caso base: n = 1
La radice è anche foglia, non ci sono nodi interni ⇒ #nodiInterni = 0 = 1 - 1
Ipotesi induttiva: ogni ABC di k nodi ha k - 1 nodi interni (con k < n)
Passo: n = #nodi interni di T, $n_{sx/dx}$ = #nodi interni di T_{sx}/T_{dx}
Poiché la radice è nodo interno si ottiene
$n = n_{sx} + n_{dx} + 1 = nodi_{sx} - 1 + nodi_{dx} - 1 + 1 = nodi_{sx} + nodi_{dx} - 1 = nodi - 1$ **CVD**

Tabelle Hash

Indirizzamento aperto: T[h(x.key)]

$x \in S$, con $|S| = n \leq m$ = **dimensione della tabella hash**

$\alpha = n/m \leq 1$, **fattore di carico**

Funzione hash: **da una chiave e un indice di ispezione** (numero di tentativi, al più tanti quante sono le celle occupate trovate) **restituisce una posizione** della tabella.

$h : U \times [0, m - 1] \rightarrow [0, m - 1]$

U **insieme delle chiavi** x **indice di ispezione** → **posizione libera della tabella**

Probing: ispezione/scansione, ricerca di una cella libera

$\forall\ k \in U \Rightarrow$ **sequenza di scansione** = $\langle h(k, 0), h(k, 1), \ldots, h(k, m - 1)\rangle$, cioè una **permutazione delle m posizioni della tabella**.

Operazioni di dizionario

Ipotesi di **hashing uniforme**: ogni chiave ha la **stessa possibilità** di avere, come sequenza di scansione, una qualunque delle n! permutazioni di {0, ..., m - 1}

***Teorema*: se vale l'hashing uniforme, data una tabella hash con un fattore di carico** $\alpha = n/m < 1$ (cioè con **almeno una cella libera**), **il numero atteso di accessi nelle operazioni di dizionario è al massimo** $1/(1 - \alpha) > 1$.

Dim: #accessi per inserimento al caso medio

Ipotesi:

- analisi in funzione di α
- hashing uniforme
- tabella mai piena $0 \leq n < m - 1 \Rightarrow \alpha < 1$
- nessuna cancellazione

x = #accessi

$E[x] = \sum_{i=1}^{inf} (i \cdot Pr[x = i]) = \sum_{i=1}^{inf} (Pr[x \geq i])$

$i = 1 \Rightarrow Pr[x \geq 1] = 1$

$i = 2 \Rightarrow Pr[x \geq 2]$ = probabilità di trovare la prima cella occupata = α

$i = 3 \Rightarrow Pr[x \geq 3]$ = probabilità di trovare le prime due celle occupate = $n/m \cdot (n - 1)/(m - 1) = \alpha \cdot \alpha$

Sempre dipendente dal fattore di carico e mai dalle dimensioni assolute.

$Pr[x \geq i]$ = probabilità di trovare le prime i celle occupate =

$= n/m \cdot (n - 1)/(m - 1) \cdot \ldots \cdot (n - i+2)/(m - i+2) \leq (n/m)^{i-1} = \alpha^{i-1}$

$E[x] = \sum_{i=1}^{inf} (Pr[x \geq i]) \leq \sum_{i=1}^{inf} \alpha^{i-1} = \sum_{i=0}^{inf} \alpha^{i} = 1/(1 - \alpha)$ **CVD**

Calcolo delle sequenze di scansione

Scansione lineare

m = numero primo

$h(k, i) = (h'(k) + i) \bmod m$, con h' funzione di hash ausiliare $h' : U \rightarrow [0, m - 1]$

Primo accesso casuale (da h'(k), dipendente da k) ma passo costante (da + i, indipendente da k)

m sequenze diverse, m << m!, quindi problema: formazione di **agglomerati** (**cluster**) **primari**, cioè lunghi tratti di celle occupate che crescono in lunghezza, causato da passi lunghi 1.

$\langle h'(k) \bmod m, h'(k) + 1 \bmod m, \ldots, m - 1, 0, 1, \ldots\rangle$

Scansione quadratica
m = numero primo
c_1, $c_2 \neq 0$ costanti
h' funzione di hash ausiliare h' : U $\rightarrow$ [0, m - 1]
$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
Primo accesso casuale come prima ma passo non costante, dipendente in modo quadratico da i. c_1 e c_2 scelte in modo da garantire la generazione di una sequenza di scansione che sia una permutazione di tutte le celle della tabella.
Cluster secondari: chiavi in **collisione** (cioè con **stesso valore hash** h') hanno comunque la stessa sequenza di scansione (poiché hanno lo stesso punto di partenza).
m sequenze diverse, m << m!.

Doppio hashing
Si avvicina all'hashing uniforme
Il punto di partenza e il passo dipendono entrambi dalla chiave.
h_1 e h_2 funzioni di hash ausiliarie
$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$
Primo accesso casuale da $h_1(k)$ e passo dipendente in modo pseudocasuale da k con $h_2(k)$.
$<h_1(k) \bmod m, (h_1(k) + h_2(k)) \bmod m, ..., m - 1, ...>$
Per avere una permutazione di tutte le posizioni $MCD(h_2(k), m) = 1$, ad esempio:

- m = potenza di 2, h_2 = sempre valore dispari
- m = numero primo, $h_1(k) = k \bmod m$, $h_2(k) = 1 + k \bmod (m - 1)$
  $\forall$ k $1 \leq h_2(k) \leq m - 1$
  $h(k, i) = (k \bmod m + i(1 + k \bmod (m - 1)) \bmod m$

Risolve il problema dei cluster, **primari** e **secondari**.
sequenze diverse = $\theta(m^2) < m!$
$h_1(k)$ e $h_2(k)$ determinano la sequenza.

Numero medio di passi in una ricerca con successo

α	10%	50%	75%	90%
Lineare	1.06	1.50	2.50	5.50
Quadratica	1.05	1.44	1.99	2.79
Doppio Hash	1.05	1.38	1.83	2.55

Alberi Binari per implementare dizionari ordinati
Sono **strutture concatenate** in cui ogni nodo x **rappresenta un elemento del dizionario**:

- x. key, chiave
- x.dati, dati satellite
- x.p, padre
- x.left, figlio sx
- x.right, figlio dx

Struttura preferita: **ABR**, Albero Binario di Ricerca.
Proprietà degli ABR: dato un nodo k, **tutti i figli del sottoalbero sx hanno chiave < k e tutti i figli del sottoalbero dx hanno chiave > k**
$\forall$ x $\in$ ABR $\Rightarrow$ ($\forall$ nodo y $\in$ sottoalbero sx $\Rightarrow$ y.key < x.key) $\wedge$ ($\forall$ nodo z $\in$ sottoalbero dx $\Rightarrow$ z.key > x.key)
La visita simmetrica di un ABR fornisce la sequenza ordinata delle chiavi (elementi del dizionario): si visita ricorsivamente prima il sottoalbero sx, poi la radice e poi il sottoalbero dx.

***Teorema*: sia H uno heap. Se i è indice di un nodo in H, allora:**

- parent(i) = parteinterainferiore(i/2), non definito per la radice
- left(i) = 2i
- right(i) = 2i + 1

Dim: per induzione su i
Caso base: i = 1
left(1) = 2, rifght(1) = 3
i = 2 ⇒ parent(2) = 1
i = 3 ⇒ parent(3) = 1
Ipotesi induttiva: P(i) vero ⇒ P(i + 1) vero
Passo: per left e right le regole valgono, per parent si distinguono due casi:

- i pari ⇒ i + 1 dispari
  In questo caso parent(i) = parent(i + 1) perché i + 1 è dispari
  Parent(i + 1) = parteinterainferiore(i/2) = i/2 = parteinterainferiore((i + 1)/2)
- i dispari ⇒ i + 1 pari
  In questo caso parent(i) ≠ parent(i + 1), ma parent(i + 1) = parent(i) + 1
  parent(i) + 1 = parentinterainferiore(i/2) + 1 = (i - 1)/2 + 1 $^{\text{perché dispari}}$ = (i + 1)/2 = parteinterainferiore((i + 1)/2) = parent(i + 1) **CVD**

***Teorema*: uno heap di n elementi ha altezza O(log n) e, più precisamente, ha altezza parteinterainferiore(log n).**
Dim: sia h = altezza dello heap. Abbiamo che (I) $2^h \leq$ (II) $2^{h+1} - 1 < 2^{h+1}$
(I) $1 + \sum_{k=0}^{h-1} 2^k = 1 + 2^h - 1 = 2^h$
(II) $\sum_{k=0}^{h-1} 2^k = 2^{h+1} - 1$
(I) ∧ (II) ⇒ $2^h \leq n \leq 2^{h+1} - 1$ ⇒ $h \leq \log_2 n \leq h + 1$ ⇒ $\log_2 n - 1 \leq h \leq \log_2 n$
Quindi h = parteinterainferiore(log n) **CVD**

***Teorema*: uno heap di n nodi contiene parteinterasuperiore(n/2) foglie.**
Dim: chiamiamo x il numero di foglie presenti sull'ultimo livello. Allora $n = 2^h - 1 + x$ perché $(2^h - 1)$ è il numero di nodi per heap di altezza h - 1.
Caso 1: x è pari
Abbiamo 2^{k-1} nodi (nodi sul penultimo livello) di cui x/2 sono padri delle foglie. Allora
foglie = $2^{h-1} - x/2 + x = 2^{h-1} + x/2 = (2^h + x)/2$ = parteinterasuperiore($(2^h + x)/2$) = parteinterasuperiore($(2^h + x - 1)/2$) = parteinterasuperiore(n/2)
Caso 2: x è dispari
Abbiamo (x + 1)/2 padri di foglie di cui
foglie = $2^{h-1} - (x + 1)/2 + x = 2^{h-1} + (2x - x + 1)/2 = 2^{h-1} + (x - 1)/2 = (2^h + x - 1)/2$ = parteinterasuperiore(n /2) **CVD**

***Teorema*: in uno heap di n nodi ci sono al più parteinterasuperiore($n/(2^{h+1})$) nodi di altezza h (ed esattamente parteinterasuperiore($n/(2^{h+1})$) su lo heap è un ABCB).**
Dim: per induzione su h
Base: h = 0
$n / 2^{h+1} = n/2 = 1$
Ipotesi induttiva: P(k) vera per k < h ⇒ p(h) vera
Passo: sia n_h = # nodi di altezza h in un albero con n nodi. Chiamiamo T' l'albero ottenuto da T rimuovendo tutte le foglie. T' ha n' nodi, allora:
$n' = n - n_0 = n$ - parteinterasuperiore(n/2) = parteinterainferiore(n/2)
Abbiamo che n'_{h-1} = # foglie di altezza h - 1 in T'. Allora:
$n'_{h-1} = n_{h-1}$ $^{\text{per ip ind}} \leq$ parteinterasuperiore(n'/2) = parteinterasuperiore(parteinterainferiore(n/2)/2^h) ≤ parteinterasuperiore((n/2)/2^h) = parteinterasuperiore($n/2^{h+1}$) **CVD**

***Teorema*: Fib_h è albero 1-blianciato e minimale: rimuovendo un nodo si perde o l'1-bilanciamento o l'altezza h.**
Dim: per induzione su h
Base h = 0, Fib_0 è 1-bilanciato e minimale (un solo nodo)
h = 1, Fib_1 è 1-bilanciato e minimale (nodo con figlio sx o dx)
Ipotesi induttiva: Fib_k con k < h è 1-bilanciato e minimale
Passo: supponiamo che Fib_h non sia minimale. Poiché non si può rimuovere la radice dobbiamo rimuovere un nodo da Fib_{h-1} o Fib_{h-2} (minimali per ipotesi induttiva) senza perdere il bilanciamento o l'altezza.
Caso 1: rimuoviamo u da Fib_{h-1}**.** Poiché è minimale, per non perdere l'1-bilanciamento possiamo solo ridurre l'altezza a h - 2: in questo modo però si riduce l'altezza di Fib_h a h - 1, in contrasto con l'ipotesi.
Caso 2: rimuoviamo u da Fib_{h-2}. Anche qui riduciamo l'altezza di Fib_{h-2} a h - 3. Così facendo si perde l'1-bilanciamento sulla radice, di nuovo in contrasto con l'ipotesi.
Quindi Fib_h è 1-bilanciato minimale.

***Teorema*: la visita simmetrica di un ABR fornisce la sequenza ordinata degli elementi del dizionario.**
Dim: per induzione su n = # elementi
Caso base: n = 0 o n = 1
La sequenza generata (vuota o 1 elemento) è banalemente ordinata.
Ipotesi induttiva: P(k) vera per $k < n \Rightarrow$ P(n) vera
Passo: abbiamo un ABR di n elementi, con $n = n_{sx} + n_{dx} + 1$, dove $n_{sx/dx}$ = # nodi del sottoalbero sx/dx.
La visita simmetrica restituisce prima tutti gli elementi di T_{sx} (ordinati per ipotesi induttiva), poi la radice (> di ogni elemento in T_{sx} e < di ogni elemento in T_{dx}) e infine gli elementi di T_{dx}, anch'essi ordinati per ipotesi. Quindi la visita simmetrica restituisce la sequenza ordinati degli elementi dell'ABR.

***Teorema*: sia n_h il # nodi di Fib_h: $n_h = F_{h+3} - 1$**
Dim: per induzione su h
Caso base: h = 0 o h = 1
Fib_0 = unico nodo $\Rightarrow F_{h+3} - 1 = F_3 - 1 = 2 - 1 = 1$ ok
Fib_1 = due nodi, figlio sx o dx $\Rightarrow F_{h+3} - 1 = F_4 - 1 = 3 - 1 = 2$ ok
Ipotesi induttiva: l'albero Fib_k con k < h ha $n_k = F_{k+3} - 1$
Passo:
$n_h = n_{h-1} + n_{h-2} + 1$
Per ipotesi induttiva $n_h = F_{h-1+3} - 1 + F_{h-2+3} - 1 + 1 = F_{h+2} + F_{h+1} - 1 = F_{h+3} - 1$ **CVD**

***Teorema Lemma Φ:* sia n_h = # nodi di Fib_h, allora $n_h \geq \Phi^h \ \forall\ h \geq 0$ (con $\Phi = 1.618...$)**
Dim: per induzione su h
Caso base: h = 0 o h = 1
Fib_0 = unico nodo, $\Phi^0 = 1 \Rightarrow 1 \geq 1$ ok
Fib_0 = due nodi, $\Phi^1 = 1.618... \Rightarrow 2 \geq \Phi^1$ ok
Ipotesi induttiva: n_k di Fib_k con k < h è $\geq \Phi^k$
Passo: $n_h = n_{h-1} + n_{h-2}$. Per ipotesi induttiva
$n_h \geq \Phi^{h-1} + \Phi^{h-2}$, raccolgo
$\Phi^{h-1} + \Phi^{h-2} = \Phi^{h-2}(1 + \Phi)$. Poiché Φ rappresenta la sezione aurea so che $(\Phi + 1) = \Phi^2$. Sostituisco:
$\Phi^{h-2}(1 + \Phi) = \Phi^2\Phi^{h-2} = \Phi^h$
Quindi $n_h \geq \Phi^h$ **CVD**

***Teorema:* ogni albero 1-bilanciato con n nodi ha altezza h = O(log n)**
Dim: sia h = altezza dell'albero.

- $n \geq n_h$ (per minimalità di Fib_h)
- $n_h \geq c^h$ per c > 1, $c = \Phi$ (per il lemma Φ)

allora si ha $n \geq c^h \Leftrightarrow \log_c n \geq h$.
Quindi l'altezza dell'albero è logaritmica in n, cioè h = O(log n) **CVD**

Grafi

Non orientati G = (V, E)
V = insieme dei vertici (o nodi)
$E \subseteq VxV$ = insieme degli archi, coppie **non ordinate** di nodi
$(u, v) \in E \Rightarrow (u, v) = (v, u)$ con $u, v \in V$
Se $u, v \in V$ e $(u, v) \in E$ allora u e v sono **adiacenti** (cioè **esiste un arco che li collega**) e l'arco (u, v) è **incidente sui vertici** u e v.
$|V| = n$ = **ordine del grafo**
$|E| = m \leq \binom{n}{2} = (n(n-1))/2 = \theta(n^2)$, $0 \leq |E| \leq \binom{n}{2}$
Grafo **sparso** se $|E| = O(n)$, **denso** se $|E| = \theta(n^2)$
Graficamente sono nodi collegati da archi, la disposizione è irrilevante fintanto che gli archi siano collegati correttamente.
Def: **grado del vertice** $\delta(v)$ = #archi incidenti su v. Se $\delta(v) = 0$ allora il vertice è **isolato**.
$\sum_{v \in V}(\delta(v)) = 2|E| = 2m$ perché ogni arco incide su due veritici, quindi contribuisce con un fattore 2.
Def: **sequenza di vertici** $u = x_0, x_1, \ldots, x_k = v \mid \forall i$ con $1 \leq i \leq k$ e $(x_{i-1}, x_i) \in E$
Def: **cammino semplice** se **passa da ogni vertice una sola volta**, quindi è privo di cicli e, quindi, tutti i vertici sono distinti.
Def: **lunghezza** di un cammino è il numero di archi.
Def: **distanza** tra due vertici è il **numero minimo di archi da percorrere** per andare da un vertice all'altro. Se un vertice è isolato o se non esiste un cammino $\Rightarrow d = \infty$
Def: **ciclo** è un cammino che **parte e arriva nel solito vertice**.
Def: un grafo si dice **connesso** se **esiste un cammino tra due vertici qualsiasi sia la coppia scelta**.
Quindi il grafo è connesso $\Leftrightarrow (\forall u, v \in V \Rightarrow \exists u \rightsquigarrow v)$
Def: un grafo si dice **completo** se **ha tutti i possibili archi**. Quindi se $\forall u, v \in V \Rightarrow \exists (u, v) \in E$
Questo significa che $|E| = m = \binom{n}{2} = (n(n-1))/2$. Il grafo viene definito **clique** o **cricca**.
Quindi $\forall v \in V \Rightarrow \delta(v) = |V| - 1 = n - 1$
Def: **sottografo** di G = (V, E) è G' = (V', E') con $V' \subseteq V$ e $E' \subseteq V'xV'$ e $E' \subseteq E$
Def: **componente connessa** di un grafo è un **sottografo di G connesso e massimale** (non ulteriormente estendibile perché non sarebbe più connesso).

Orientati (o **diretti**) G = (V, E) come prima ma con E = insieme di coppie **ordinate** di vertici.
Quindi $(u, v) \neq (v, u)$, poiché gli archi **hanno un verso di percorrenza**, $0 \leq |E| \leq 2\binom{n}{2} = n(n-1)$
$\delta(v)$
$\delta(v) = \delta_e(v) + \delta_u(v) \Rightarrow \sum_{u \in V}(\delta_u(v)) = |E| = m$
Def: **cammino orientato** è una sequenza di vertici adiacenti a due a due **orientati dal primo verso il secondo**. $u, v \in V$ sono **connessi** $\Leftrightarrow \exists$ cammino orientato da u verso v
Def: un grafo è **fortemente connesso** se **ogni coppia di vertici è connessa**.
$\forall u, v \in V \Rightarrow \exists u \rightsquigarrow v$ e $v \rightsquigarrow u$, cioè se u e v sono **mutualmente raggiungibili**.
Def: la **componente fortemente connessa** è il **sottografo fortemente connesso e massimale**.
Def: un grafo è **aciclico** se è **privo di cicli**.

Albero: grafo aciclico non orientato e connesso.
$|E| = |V| - 1$, perché se ho $< |V| - 1$ allora non è connesso, se ho $> |V| - 1$ allora ho un ciclo.

Foresta: grafo aciclico non orientato le cui componenti connesse sono alberi.

Rappresentazione in memoria dei grafi

- **Matrice di adiacenza**
  Ogni vertice è etichettato con un numero tra 1 e |V| = n
  A = matrice nxn | $\forall$ i, j con $1 \leq i, j \leq n \Rightarrow A[i, j] = 1$ se $(i, j) \in E$, 0 altrimenti
  Se G è non orientato allora A è simmetrica, cioè $\forall$ i, j $\Rightarrow A[i, j] = A[j, i]$
  Grado δ (i) in θ (n)
  adiacenti(i, j), aggiungiarco(i, j) e rimuoviarco(i, j) in θ (1)
  $S(n, m) = \theta(n^2)$, quindi ok per i grafi densi ma troppa memoria per quelli sparsi.
- **Liste di adiacenza**
  Lista dell'i-esimo vertica = vertici adiacenti ad i
  Adj = array di n liste doppie con Adj[i] = vertice j | $(i, j) \in E$ se $\delta(i) > 0$, $\varnothing$ se i è isolato
  $\forall$ i con $1 \leq i \leq n \Rightarrow |Adj[i]| = \delta(i)$ se non orientato, $\delta_u(i)$ se orientato
  Quindi $\sum_{i=1}^{n}(|Adj[i]|) = 2|E|$ se non orientato, $|E|$ se orientato
  Se G è non orientato ogni arco viene rappresentato 2 volte, altrimenti una sola.
  Grado δ (i) in $\theta(\delta(i))$
  adiacenti(i) in $O(\delta(i)) = O(n)$ al caso peggiore
  aggiungiarco(i, j) in θ (1) perché j in i se orientato, j in i e i in j se non orientato
  rimuoviarco(i, j) in $O(\delta(i) + \delta(j))$ se non orientato tolgo j da i e i da j, $O(\delta(i))$ se orientato tolgo j da i
  $S(n, m) = \theta(n + m)$, se sparso quindi $|E| = O(n)$ allora $S(n, m) = \theta(n)$

Problemi sui grafi

- **Matching perfetto**
  Dato grafo G = (V, E) trovare $E' \subseteq E$ | $\forall$ $v \in V$ allora v occorre in 1 e 1 solo arco in E'
  Tempo polinomiale
- **Cammino Hamiltoniano** e **Ciclo Hamiltoniano**
  Trovare un cammino semplice (o ciclo) che passa da ogni nodo una e una sola volta
  NP completo
- **Ciclo Euleriano**
  Trovare un ciclo che passa da tutti gli archi una e una sola volta. Esiste se e solo se il grafo è connesso e tutti i vertici hanno grado pari.
  Ammette un algoritmo polinomiale.

Visita di grafi

Visita in ampiezza BFS breath-first-search
G = (V, E), sorgente $s \in V$
Scopre in vertici in ordine di distanza crescente dalla sorgente

- Uso della coda
- Marcatura dei vertici, colorazione
  - Bianco, non ancora scoperto
  - Grigio, scoperto
  - Nero, scoperto e lista di adiacenza completamente esaminata
- Scopre tutti e soli i vertici raggiungibili da s (se è connesso, tutto G)
- Calcola le distanze da s di tutti i vertici scoperti

$\Rightarrow \forall v \in V$

- v.color
- v.d, distanza da s
- $v.\pi$, predecessore di v, vertice u dal quale v è stato scoperto tramite l'arco (u, v)

Albero BFS: è un sottografo di G costruito con la BFS
$BF = (V_\pi, E_\pi)$ con $V_\pi = (v \in V \mid v.\pi \neq NIL\} \cup \{s\}$ e $E_\pi = \{(v.\pi, v) \mid v \in V_\pi - \{s\}\}$
BF è connesso, $|E_\pi| = |V_\pi| - 1 \Rightarrow$ BF è un albero
Proprietà: $\forall v \in V_\pi$ il cammino s ⍰ v in BF è un cammino minimo in G.
Analisi:

- i vertici entrano ed escono dalla coda al più uno alla volta (quando da bianchi diventano grigi)
- inserimenti ed estrazioni costano complessivamente O(|V|)
- la lista Adj(u) del vertice u viene esaminata al più una volta (quando u viene estratto)
- il costo del while è O(|E|), ogni arco al più due visite se G non è orientato, una se è orientato

$T(|V|, |E|) = O(|V| + |E|) = \theta(|V| + |E|)$ se G è connesso, quindi lineare, quindi BFS è ottimo.

Visita in profondità DFS depth-first-search

- Procede in profondità a partire dai vertici incontrati
- Visita tutto il grafo, eventualmente selezionando più di una sorgente
- Natura ricorsiva perché la visita riparte dal vertice appena scoperto
- Costruisce una foresta DF (cioè un albero DF per ogni componente connessa)
- Non calcola le distanze

$\Rightarrow \forall v \in V$

- v.d, momento in cui v è stato scoperto e colorato di grigio
- v.f, momento in cui v diventa nero perché tutto ciò che è raggiungibile da v è stato scoperto

$1 \leq v.d < v.f \leq 2|V|$
$T(|V|, |E|) = \theta(|V| + |E|)$
$G_\pi = (V, E_\pi)$ con $E_\pi = \{(v.\pi, v) \mid v \in V \text{ e } v.\pi \neq NIL\}$ è una foresta DF
v è un discendente di u (v è sottoalbero di radice u nella foresta DF)
$\Leftrightarrow$
v è stato scoperto quando u era grigio (visista di Adj[u] ancora in corso)

Classificazione degli archi nei grafi orientati

$\forall$ (u, v) $\in$ E per G = (V, E), (u, v) può essere classificato come

- **Arco dell'albero**
  v è bianco quando (u, v) viene ispezionato
- **Arco all'indietro**
  v è grigio quando (u, v) viene ispezionato
  $\Rightarrow$ v è un antenato di u in un albero DF
  $\Rightarrow$ u è un discendente di v
- **Arco in avanti**
  v è nero e u.d < v.d (u è stato scoperto prima di v) quando (u, v) viene ispezionato
  $\Rightarrow$ v è un discendente di u in un albero DF
- **Arco trasversale**
  v è nero e u.d > v.d (u è stato scoperto dopo di v) quando (u, v) viene ispezionato
  $\Rightarrow$ v e u non sono l'uno antenato dell'altro

***Teorema* classificazione degli archi in un grafo non orientato: in una DFS di un grafo non orientato G gli archi possono essere solo archi dell'albero o archi all'indietro**.

Dim: sia (u, v) $\in$ E e u.d < v.d (con u.d/v.d = momento in cui è stato scoperto u/v). Allora u è grigio sia quando v viene scoperto che quando diventa nero. Sono possibili due casi:

- Caso 1: l'arco viene esplorato la prima volta da u verso v. Allora v è bianco $\Leftrightarrow$ l'arco (u, v) è arco dell'albero
- Caso 2: l'arco viene esplorato la prima volta da v verso u. Allora u è grigio $\Leftrightarrow$ l'arco (u, v) è arco all'indietro

CVD

Ordinamento topologico di un grafo orientato aciclico (DAG, grafo diretto aciclico)

Trovare un ordinamento di vertici | $\forall$ (u, v) $\in$ E u precede v nell'ordinamento

Si trova usando la DFS

TopologicalSort(G), con G DAG

- DFS(G) calcolando i tempi di fine visita di ogni v $\in$ V
- quando un vertice diventa nero, si inserisce in testa ad una lista concatenata
- return lista concatenata di vertici

T(|V|, |E|) = θ (|V| + |E|)

Lemma: un grafo è aciclico $\Leftrightarrow$ non ci sono archi all'indietro

***Teorema*: TopologicalSort(G) produce un ordinamento topologico di un DAG G**

Dim: Ordinamento Topologico: $\forall$ (u, v) $\in$ E $\Rightarrow$

- u deve precedere v nell'OT
- u deve essere inserita in lista dopo v (perché inserimenti in testa
- deve valere v.f < u.f (v va in lista prima di u)

Dimostro che $\forall$ (u, v) $\in$ E $\Rightarrow$ v.f < u.f

Sia $\forall$ (u, v) $\in$ E, quando si ispeziona l'arco (u, v) v non può essere grigio (perché G è aciclico, quindi no archi all'indietro)

Caso 1: v è bianco, allora u è grigio -> scopro v -> chiamo DFSVisit(G, v). v allora diventa nero prima di u, quindi v.f < u.f OK

Caso 2: v è nero, allora la lista di v è già terminata mentre quella di u è ancora in corso, quindi v.f < u.f OK **CVD**