Programmazione e Algoritmica

2023/24

Mario La Gualano

Sommario

Sintassi e grammatiche

Grammatica: come comporre le frasi (mettendo insieme le parole).

Lessico: l'insieme delle parole che possono essere composte a partire da un insieme di simboli atomici detti caratteri, che rappresentano l'alfabeto e possono essere utilizzati per formare frasi.

Sintassi: Scrivere frasi corrette per un linguaggio (a prescindere dal loro significato).

Semantica: associa ad ogni elemento del dominio, uno e un solo elemento del codominio. Assegna un significato ai simboli del linguaggio e alle parole all'interno di un dominio interpretativo astratto detto **dominio semantico**.
Una definizione semantica per essere utile deve essere **composizionale**, cioè il significato di una qualunque frase viene fornito in termini del significato dei suoi componenti elementari.

Alfabeto: insieme finito e non vuoto di simboli.

Stringa: concatenazione di un insieme finito e non vuoto di simboli di un alfabeto.

Chiusura di Kleene*: contiene tutte le stringhe di qualsiasi lunghezza, che si possono formare concatenando i simboli di un determinato alfabeto.

Definizione di grammatica

Una grammatica è una quadrupla: G= <N, Σ, P, S>
N: insieme finito e non vuoto di simboli **non terminali**;
Σ: alfabeto di simboli **terminali**;
P $\subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$: insieme finito di **produzioni** (parte sinistra almeno un terminale, parte destra anche vuoto)
S $\in$ N: **simbolo distinto** (o **simbolo iniziale**);

Le grammatiche si classificano in base all'espressività del linguaggio che generano:
Gerarchia di Chomsky: classificazione basata sulla forma delle produzioni:
Macchina di Turing (**grammatica di tipo 0**)
Automi Lineari (**grammatiche dipendenti dal contesto**)
Automi a pila (**grammatiche libere da contesto**)
Automi a stati finiti (**grammatiche regolari o lineari destre**)

Definizione di linguaggio

Linguaggio: Un linguaggio L su un alfabeto A è un sottoinsieme della chiusura di Kleene di A.

Linguaggio di programmazione: l'insieme delle stringhe ammissibili (che chiamiamo programmi).

Un linguaggio può essere definito con tre metodi:
GENERATIVO: stabilisco un insieme di regole che mi permette di generarlo;
RICONOSCITIVO: stabilisco un automa per riconoscere le stringhe ammissibili;
ALGEBRICO: l'insieme delle stringhe soluzione di un sistema di equazioni algebriche;

Le fasi di un compilatore

Codice sorgente ➔ Compilatore: [**Scanner** (analizzatore lessicale) ➔ **Parser** (analizzatore sintattico) ➔ **Type Checking** (analizzatore semantico) ➔ **generatore codice oggetto** .o] ➔ Linker ➔ Eseguibile

Programma: stringa di caratteri alfanumerici ASCII.

BNF e sintassi di un linguaggio di programmazione

BNF (Backus-Naur Form):
E ::= E + E | E – E | E * E | -E | (E) | | | V
V ::== x | y | z
Simbolo iniziale: il primo simbolo della prima riga, **Non terminali**: simboli a sinistra delle produzioni, **Terminali**: insieme di tutti i simboli meno i primi di ogni riga.

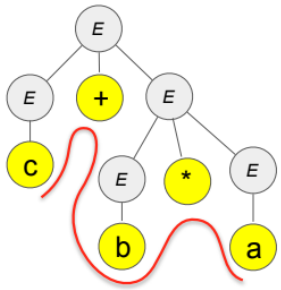
$\delta \in (N \cup \Sigma)^+$, $\gamma \in (N \cup \Sigma)^*$
δ è **un derivativo immediato** di γ ($\gamma \rightarrow \delta$) se e solo se δ si può ottenere da γ applicando *una sola* produzione della grammatica.
δ è un **derivativo** di γ ($\gamma \rightarrow^* \delta$) se δ si ottiene da γ applicando *un numero qualsiasi* di produzioni della grammatica.

Il **linguaggio generato da una grammatica** G è l'insieme delle stringhe di caratteri terminali che si possono ottenere applicando un qualsiasi numero di produzioni a partire dal simbolo iniziale:
$L(G) = \{ w \mid w \in \Sigma^* \wedge S \rightarrow^* w\}$

Due grammatiche sono **equivalenti** se producono lo stesso linguaggio.

Una derivazione è **canonica destra** (o **sinistra**) se il simbolo non terminale ad essere sostituito è sempre quello più a destra (o a sinistra).

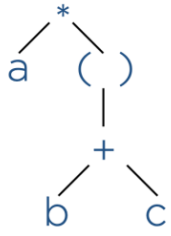
Le **derivazioni** possono essere rappresentate **tramite alberi**:



c + b * a

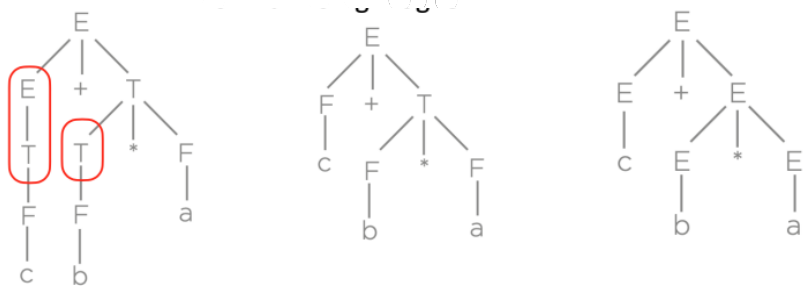
Una grammatica è **ambigua** se esiste una stringa del linguaggio generato che può essere ottenuta con due alberi di derivazione diversi. L'ambiguità deriva dal fatto che le produzioni non riescono a determinare correttamente la precedenza delle produzioni.
ES.
Per dare priorità al * rispetto al +, in una grammatica per algebra, mettiamo le produzioni per * più in baso rispetto a quelle per +.

Albero di sintassi: rappresento in modo formale le frasi del linguaggio sintatticamente corrette (è un albero di derivazione senza i non terminali):



a * (b + c)

Albero di sintassi astratta: si eliminano tutti i non terminali che non hanno figli foglia (terminali), quelli che rimangono li rinomino con il nome della radice:



c + (b * a)

Astrazione di Von Neumann

Una macchina astratta è un'astrazione di una generica architettura che definisce gli aspetti che ogni architettura ha:

- la **memoria**: insieme di celle contigue ciascuna caratterizzata da un indirizzo univoco (**locazione**) e da un contenuto (**valore**);
- il **controllo di flusso** (l'ordine di esecuzione delle istruzioni elementari)
- la **CPU**

Funzione **ambiente** ρ: Id → Loc ∪ Val, associa nomi a locazioni;
Funzione **memoria** σ: Loc → Val, associa valori alle locazioni;

Modello computazionale

Modello **RAM**

- Memoria principale infinita
  - Ogni cella contiene una quantità finita di dati
  - Impiego lo stesso tempo per accedere ad ogni cella
- Singolo processore + programma
  - In una unità di tempo svolgo lettura, esecuzione di una computazione, scrittura
  - Addizione, moltiplicazione, assegnamento, confronto, accesso a puntatore

Il modello RAM è un'astrazione dei moderni calcolatori

Modello **Macchina di Turing**

- Nastro di lunghezza infinita
  - In ogni cella può essere contenuta una quantità di informazione finita
- Una testina + un processore + programma
  - In una unità di tempo leggere o scrivere la cella di nastro corrente
  - Si muove di una cella a destra o a sinistra oppure resta ferma

Nonostante sia basilare è del tutto equivalente a un calcolatore moderno.

Linguaggio L

BNF e Sintassi

Dichiarazioni: modificano la memoria. Definiscono gli identificatori (la prima volta) associandoli ad una locazione di memoria libera. Inoltre, ci scrivono il valore iniziale. Costruiscono l'ambiente.
Una dichiarazione è ben formata se possiamo associarle un ambiente statico che contiene i legami definiti da essa.
Comandi: modificano la memoria. Descrivono il cambiamento di stato del calcolatore.
Un comando è ben formato se si può derivare a partire dall'ambiente statico.
Espressioni: non modificano né l'ambiente né la memoria. Rappresentano i valori su cui opera il programma.
Un'espressione è ben formata se a partire dall'ambiente statico le si può associare il tipo del valore che rappresenta

Ambiente statico Δ: lo usiamo per controllare i tipi e la "coerenza".
Ambiente dinamico: "simula" l'esecuzione.

Overloading degli operatori: stesso simbolo sintattico usato per indicare operazioni semanticamente diverse (moltiplicazione sia fra Int x Int che fra Double x Double).

Tipi di **errori**:

- **a tempo di compilazione** (statico) [sbagli a scrivere] **Lessicale**, **Sintattico**, **Semantico**
- **a tempo di esecuzione** (dinamico) [operazioni incalcolabili] **Logico**

Regole semantica statica

Usiamo l'ambiente statico che associa a ciascun nome un tipo, definendo anche se si tratta di una costante o di una variabile.

Scoping e blocchi (identificatori liberi e legati)

Scope: lo scope di un identificatore è la porzione di programma in cui può essere referenziato.

Block-scope: un identificatore può essere referenziato solo all'interno del blocco in cui è stato definito (nella parte di blocco seguente a dove è definita e in eventuali blocchi annidati in quello corrente)

Lo scoping può essere:

- **Statico**: lo scoping degli identificatori è stabilito a tempo di compilazione guardando gli alberi di sintassi.
- **Dinamico**: lo scoping degli identificatori è stabilito a tempo di esecuzione in base a come varia l'ambiente dinamico.

Ogni identificatore può essere **libero** (non dichiarato) o **legato** (definito e usato). Quando un identificatore viene dichiarato si dice che è in **posizione di definizione** e costituisce una **occorrenza di legame** per le altre sue occorrenze all'interno del blocco in cui è stato dichiarato (o di quelli annidati).

Record di attivazione

Catena dinamica: indirizzo del record di attivazione della funzione chiamante. Rappresenta la sequenza di chiamate per il corretto ordine di esecuzione.
Catena statica: garantisce che i nomi siano referenziati rispettando la visibilità di variabili e funzioni.
Indirizzo risultato: indirizzo nel record per memorizzare il risultato.
Indirizzo ritorno: indirizzo dell'istruzione da eseguire al termine della funzione.
Variabili locali: spazio riservato all'associazione delle variabili locali al blocco.
Parametri: spazio riservato alla associazione parametri formali – parametri attuali.
Risultato della chiamata: spazio riservato alla allocazione delle variabili temporanee generate dal compilatore.

I record di attivazione vengono salvati nello stack.

Regole semantica dinamica

Simula l'esecuzione del programma mostrando gli effetti che ogni istruzione comporta sulla macchina.

Funzioni: dichiarazione (lambda astrazione e chiusure) e chiamata

$\langle (Id, \lambda form.\{\rho'; C; return\ E\}), \sigma \rangle \qquad \begin{cases} \rho' = \rho_{FI(C)-BI(form)} \\ \rho' = nil \end{cases}$

Tipi di ricorsione

Mutua ricorsione: due o più funzioni sono definite ciascuna in termini delle altre

```
func f(){          g(){
...                ...
g()                f()
}                  }
```

Ricorsione annidata: una funzione viene passata come parametro formale nella sua stessa definizione. Molto complessa, meglio evitarla.

$$\textit{Funzione di Ackermann: } A(m,n)\begin{cases} n+1 & se\ m = 0 \\ A(m-1,1) & se\ m > 0\ \&\&\ n = 0 \\ A(m-1, A(m,n-1)) & se\ m > 0\ \&\&\ n > 0 \end{cases}$$

Ricorsione in coda: la chiamata ricorsiva è l'ultima operazione effettuata dalla funzione prima di restituire il controllo al chiamante e viene definita chiamata terminale. Quando si arriva al caso base quindi tutti i record si chiudono.
Alcune macchine approfittano della cosa modificando solo i parametri (che di fatto sono l'unico dato che cambia nel record) senza crearne uno per ogni chiamata.
È possibile trasformare la ricorsione in coda in una versione iterativa:

- Il caso base viene messo nell'else
- If diventa un while
- Il passaggio di parametri alla funzione diventa un aggiornare tali variabili
- Le istruzioni nell'else vengono eseguite fuori dal while

ES. Funzione fattoriale
Rendiamo ricorsiva in coda la funzione per calcolare il fattoriale:

```
func f(var n: Int) → Int{
if(n ≤ 1) return 1
else return n * f(n - 1)
}
```

➔

```
func fa(var n: Int, var accum: Int) → Int{
    if(n ≤ 1) return accum;
    else return fa(n – 1, n * accum
}
```

Complessità degli algoritmi

Le unità che usiamo per stimare la bontà di un programma sono: **spazio** occupato e **tempo** impiegato.

Definizione formale di O, Ω, Θ, o, ω

O(g(n)) = {f(n) : ∃ c, n_0 ≥ 0 : 0 ≤ f(n) **≤** cg(n) ∀ n ≥ n_0}
o(g(n)) = {f(n) : ∀ c > 0, ∃ n_0 > 0 : 0 ≤ f(n) **<** cg(n) ∀ n ≥ n_0}

Θ(g(n)) = {f(n) : ∃ c_1, c_2, n_0 ≥ 0 : 0 ≤ c_1g(n) ≤ f(n) ≤ c_2g(n) ∀ n ≥ n_0}
ω((g)n)) = {f(n) : ∀ c > 0, ∃ n_0 > 0 : 0 ≤ cg(n) < f(n) ∀ n ≥ n_0}

Ω(g(n)) = {f(n) : ∃ c, n_0 ≥ 0 : 0 ≤ cg(n) ≤ f(n) ∀ n ≥ n_0}

Algoritmi di ordinamento

Ordinamento *per confronti*

Insertion sort

Algoritmo efficiente per ordinare un piccolo numero di elementi.
Prendo un elemento alla volta e lo confronto con gli elementi prima, se è maggiore o uguale resta lì, se è minore lo scambio. È un algoritmo in place.
L'indice che confronto è j, perciò la sezione dell'array precedente ad esso è ordinata.

Dimostrazione di correttezza e complessità

INSERTION-SORT(A, n) COSTO: **O(n^2)**

```
1       for j = 1 to n
2               key = A[j]
3               // Inserisce A[j] nella sequenza ordinata A[0 . . j - 1].
4               i = j – 1
5               while i > 0 and A[i] > key
6                       A[i + 1] = A[i]
7                       i = i – 1
8               A[i + 1] = key
```

Invariante di ciclo: All'inizio di ogni iterazione del ciclo for, il cui indice è j, il sottoarray che è formato dagli elementi A[0 . . j – 1] costituisce la parte di array correntemente ordinato e gli elementi A[j + 1 . . n] corrispondono a quelli ancora da ordinare. Inoltre, ad ogni ciclo la parte ordinata aumenta di uno.

1) Inizializzazione (caso base): e vera prima della prima iterazione del ciclo.
2) Conservazione (passo induttivo): se è vera prima di un'iterazione del ciclo, rimane vera prima della successiva iterazione.
3) Conclusione: quando il ciclo termina, l'invariante fornisce un'utile proprietà che ci aiuta a dimostrare che l'algoritmo è corretto.

1) Iniziamo dimostrando che l'invariante di ciclo è vera prima della prima iterazione del ciclo, quando j = 1. Il sottoarray A[1 . . j - 1], quindi, è formato dal solo elemento A[0], che infatti è l'elemento originale in A[0]. Inoltre, questo sottoarray è ordinato (banale, ovviamente) e ciò dimostra che l'invariante di ciclo è vera prima della prima iterazione del ciclo.
2) Passiamo alla seconda proprietà: dimostrare che ogni iterazione conserva l'invariante di ciclo. Informalmente, il corpo del ciclo for esterno opera spostando A[j - 1], A[j - 2], A[j - 3] e così via di una posizione verso destra, finché non troverà la posizione appropriata per A[j] (righe 4–7), dove inserirà il valore di A[j] (riga 8). Il sottoarray A[1 . . j] quindi è ordinato ed è formato dagli stessi elementi che originariamente erano in A[1 . . j]. Dunque, l'incremento di j per la successiva iterazione del ciclo for preserva l'invariante di ciclo.
3) Infine, esaminiamo che cosa accade quando il ciclo termina. La condizione che determina la conclusione del ciclo for è: j > n. Poiché ogni iterazione del ciclo aumenta j di 1, alla fine del ciclo si avrà j = n + 1. Sostituendo j con n + 1 nella formulazione dell'invariante di ciclo, otteniamo che il sottoarray A[1 . . n] è formato dagli elementi ordinati che si trovavano originariamente in A[1 . . n]. Ma il sottoarray A[1 . . n] è l'intero array e dunque tutto l'array è ordinato.
Pertanto, l'algoritmo è corretto.

Selection sort

Troviamo l'elemento minore in un array e lo mettiamo all'inizio, ripetiamo la procedura per tutti gli elementi. COSTO: **O(n^2)**

Mergesort

Divide: divide la sequenza degli n elementi da ordinare in due sottosequenze di n=2 elementi ciascuna.
Impera: ordina le due sottosequenze in modo ricorsivo utilizzando l'algoritmo merge sort.
Combina: fonde le due sottosequenze ordinate per generare la sequenza ordinata.

Per effettuare la fusione, utilizziamo una procedura ausiliaria **MERGE(A, p, q, r)**, dove A è un array e p, q e r sono indici dell'array tali che $p \leq q < r$. La procedura assume che i sottoarray A[p . . q] e

A[q + 1 . . r] siano ordinati; li fonde per formare un unico sottoarray ordinato che sostituisce il sottoarray corrente A[p . . r].

Dimostrazione di correttezza e complessità

```
MERGE(A, p, q, r)
1       n1 = q - p + 1
2       n2 = r - q
3       crea due nuovi array L[1 . . n1 + 1] e R[1 . . n2 + 1]
4       for i = 1 to n1
5               L[i] = A[p + i - 1]
6       for j = 1 to n2
7               R[j] = A[q + j ]
8       L[n1 + 1] = ∞
9       R[n2 + 1] = ∞
10      i = 1
11      j = 1
12      for k = p to r
13              if L[i] ≤ R[j]
14                      A[k] = L[i]
15                      i = i + 1
16              else A[k] = R[j]
17                      j = j + 1
```

(mettiamo in fondo a ogni mazzo una ***carta sentinella***, che contiene un valore speciale ∞ che usiamo per semplificare il nostro codice)

Invariante di ciclo: All'inizio di ogni iterazione del ciclo for, righe 12–17, il sottoarray A[p . . k – 1] contiene, ordinati, i k - p elementi più piccoli di L[1 . . n_1 + 1] e R[1 . . n_2 + 1]. Inoltre, L[i] e R[j] sono i più piccoli elementi dei loro array che non sono stati copiati in A.

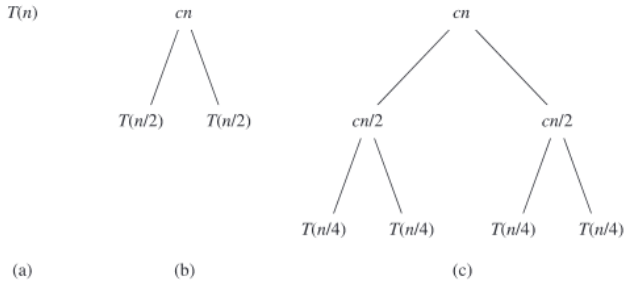
1) Inizializzazione: prima della prima iterazione del ciclo, abbiamo k = p, quindi il sottoarray A[p . . k – 1] è vuoto. Questo sottoarray vuoto contiene i k - p = 0 elementi più piccoli di L e R; poiché i = j = 1, L[i] e R[j] sono i più piccoli elementi, nei rispettivi array, tra quelli che non sono ancora stati copiati in A.
2) Conservazione: per verificare che ogni iterazione conserva l'invariante di ciclo, supponiamo dapprima che L[i] ≤ R[j]; quindi L[i] è l'elemento più piccolo che non è stato ancora copiato in A. Poiché A[p . . k – 1] contiene i k - p elementi più piccoli, dopo che la riga 14 ha copiato L[i] in A[k], il sottoarray A[p . . k] conterrà i k - p + 1 elementi più piccoli. Incrementando k (aggiornamento del ciclo for) e i (riga 15), si ristabilisce l'invariante di ciclo per la successiva iterazione. Se, invece, L[i] > R[j], allora le righe 16–17 svolgono l'azione appropriata per conservare l'invariante di ciclo.
3) Conclusione: alla fine del ciclo, k = r + 1. Per l'invariante di ciclo, il sottoarray A[p . . k – 1], che è A[p . . r], contiene k - p = r - p + 1 elementi ordinati che sono i più piccoli di L[1 . . n_1 + 1] e R[1 . . n_2 + 1]. Assieme, gli array L e R contengono $n_1 + n_2 + 2 = r - p + 3$ elementi. Tutti gli elementi, tranne i due più grandi, sono stati copiati in A; questi due elementi sono le sentinelle.

```
MERGE-SORT(A, p, r)
1       if p < r
2               q = ⌊(p + r) / 2⌋
3               MERGE-SORT(A, p, q)
4               MERGE-SORT(A, q + 1, r)
5               MERGE(A, p, q, r)
```

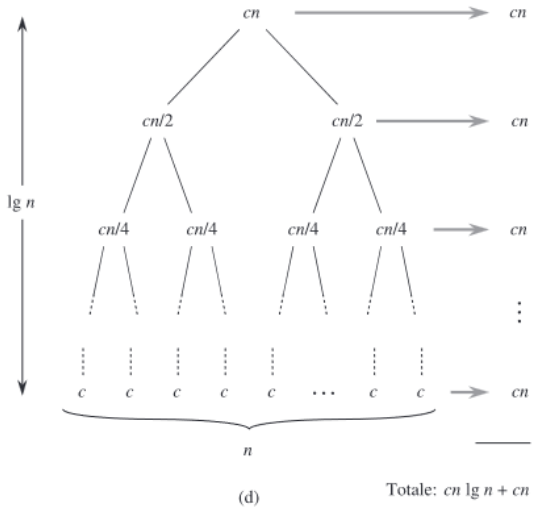
Costo:

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(\mathrm{n}) & se\ n > 1 \end{cases}$$

Per risolverla la rappresentiamo tramite un albero:



Ogni livello costa n e l'altezza dell'albero è $log\ n\ +\ 1$, quindi la somma di tutti i costi è $nlogn\ +\ n$.



Quicksort

Divide: partizionare l'array A[p . . r] in due sottoarray A[p . . q – 1] e A[q + 1 . . r] (eventualmente vuoti) tali che ogni elemento di A[p . . q – 1] sia minore o uguale ad A[q] che, a sua volta, è minore o uguale a ogni elemento di A[q + 1 . . r]. Calcolare l'indice q come parte di questa procedura di partizionamento.
Impera: ordinare i due sottoarray A[p . . q – 1] e A[q + 1 . . r] chiamando ricorsivamente quicksort.
Combina: poiché i sottoarray sono già ordinati, non occorre alcun lavoro per combinarli: l'intero array A[p . . r] è ordinato

Dimostrazione di correttezza

```
QUICKSORT(A, p, r)
1       if p < r
2               q = PARTITION(A, p, r)
3               QUICKSORT(A, p, q - 1)
4               QUICKSORT(A, q + 1, r)
```

L'elemento chiave dell'algoritmo è la procedura **PARTITION**, che riarrangia il sottoarray A[p . . r] sul posto.

```
PARTITION(A, p, r)
1       x = A[r]
2       i = p - 1
3       for j = p to r - 1
4               if A[j] ≤ x
5                       i = i + 1
6                       scambia A[i] con A[j]
7       scambia A[i + 1] con A[r]
8       return i + 1
```

Invariante di ciclo: All'inizio di ogni iterazione del ciclo, righe 3–6, per qualsiasi indice k dell'array,
1. Se p ≤ k ≤ i, allora A[k] ≤ x.
2. Se i + 1 ≤ k ≤ j - 1, allora A[k] > x.
3. Se k = r, allora A[k] = x.

Inizializzazione: prima della prima iterazione del ciclo, i = p - 1 e j = p. Non ci sono valori fra p e i né fra i + 1 e j - 1, quindi le prime due condizioni dell'invariante di ciclo sono soddisfatte. L'assegnazione nella riga 1 soddisfa la terza condizione.
Conservazione: ci sono due casi da considerare, a seconda del risultato del test nella riga 4.

- Quando A[j] > x; l'unica azione nel ciclo è incrementare j. Dopo l'incremento di j , la condizione 2 è soddisfatta per A[j -1] e tutte le altre posizioni non cambiano.
- Quando A[j] ≤ x; viene incrementato l'indice i, vengono scambiati A[i] e A[j] e, poi, viene incrementato l'indice j.

In seguito allo scambio, adesso abbiamo A[i] ≤ x e la condizione 1 è soddisfatta. Analogamente, abbiamo anche A[j – 1] > x, in quanto l'elemento che è stato spostato in A[j – 1] è, per l'invariante di ciclo, più grande di x.
Conclusione: alla fine del ciclo, j = r. Pertanto, ogni posizione dell'array si trova in uno dei tre insiemi descritti dall'invariante e noi abbiamo ripartito i valori dell'array in tre insiemi: quelli minori o uguali a x, quelli maggiori di x e un insieme di un solo elemento che contiene x.

Costo:

- Caso migliore: $\Theta(nlogn)$
- Caso peggiore: $\Theta(n^2)$

Dimostrazione complessità al caso medio

Scelgo un pivot casuale e lo scambio con l'ultimo elemento (per non modificare il codice).

A	B	A

A) caso peggiore all'estremo esterno: $T(n) = T(n-1) + O(n)$

B) caso peggiore agli estremi: $T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)$

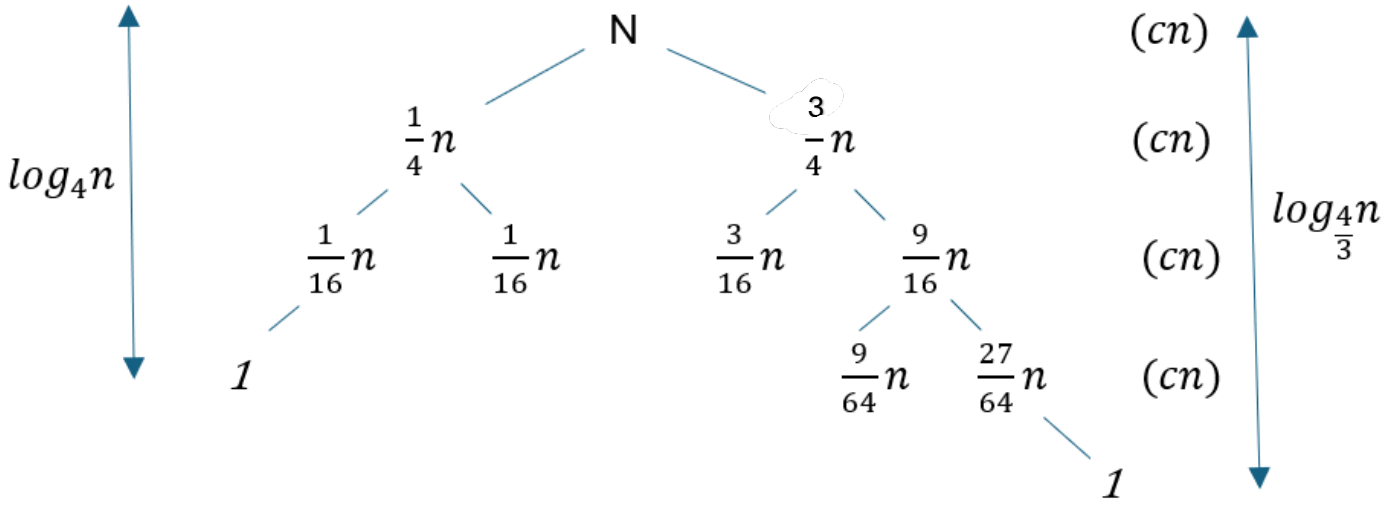
A e B sono equiprobabili, quindi $T(n) \leq \frac{1}{2}A + \frac{1}{2}B = \frac{1}{2}[A + B]$

(abbiamo maggiorato)

$$T(n) = \frac{1}{2}\left[T(n-1) + T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)\right] \leq \frac{1}{2}\left[T(n) + T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)\right]$$

$$T(n) \leq \frac{1}{2}\left[T(n) + T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)\right] \rightarrow 2T \leq T(N) + T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)$$

Se risolviamo tramite albero:



Moltiplico $log_{\frac{4}{3}}n$ per il costo di ogni livello (n): $T(n)\ nlogn$

Dimostrazione lower bound Ω(n log n) problema di ordinamento per confronti

Tutti gli algoritmi di ordinamento tramite confronti devono produrre tutte le possibili permutazioni dei dati da ordinare, altrimenti non è in grado di esprimere un certo ordine. Esprimendo tramite un albero in cui i nodi sono i confronti avremo n! foglie.
Un albero binario di altezza h ha 2^h foglie massimo.
$n! \leq foglie \leq 2^h$.
Per calcolare h: $2h \geq n!$ → applico log: $h \geq log(n!)$ → per la formula di Stirling: $h = \Omega(nlogn)$

Ordinamenti *senza confronti*

Counting sort

```
COUNTING-SORT(A, B, k, n)
1       sia C[0 . . k] un nuovo array
2       for i = 0 to k
3               C[i] = 0
4       for j = 1 to n
5               C[A[j]] = C[A[j]] + 1
6       // C[i] adesso contiene il numero di elementi uguali a i.
7       for i = 1 to k
8               C[i] = C[i] + C[i - 1]
9       // C[i] adesso contiene il numero di elementi minori o uguali a i.
10      for j = n downto 1
11              B[C[A[j]]] = A[j]
12              C[A[j]] = C[A[j]] – 1
```

1) Uso un array di supporto per segnarmi il numero di ricorrenze di un certo numero i inserendolo in posizione i e incrementandolo ogni volta che ricompare nell'array da ordinare.
2) Per ogni elemento i nell'array di supporto sommo il valore della posizione i – 1, in questo modo so quanti valori più piccoli o uguali sono presenti.
3) A questo punto scorrendo l'array di supporto da destra a sinistra scrivo i valori che leggo nell'indice indicato in essi (- 1 poiché l'array parte da 0) e lo decremento di uno.

Radix sort

Ordino prima le unità, poi le decine, poi le centinaia e così via verso sinistra.

Dimostrazione induttiva sul numero di passi:
Assumiamo che le cifre meno significative siano state ordinate. Dimostriamo che ordinare sulla successiva cifra i lascia l'array correttamente ordinato (fino a i).
Se due cifre i sono differenti ordinandole l'induzione è dimostrata.
Se le cifre sono uguali, utilizzando un ordinamento stabile, sono già ordinati.

Divide-et-impera: definizione e problemi

Un **algoritmo** si dice **ricorsivo** se al suo interno sono presenti chiamate a sé stesso per gestire sottoproblemi analoghi a quello dato.

Divide-et-impera: **dividere** un problema in sottoproblemi che, a loro volta vengono suddivisi in sottoproblemi, risolverli e **ricombinare** i risultati.

Trovare valore **massimo** (o minimo) **in un array**: richiamo la funzione su metà array fino a raggiungere un punto in cui controllo elemento per elemento (caso base) a quel punto risalgo combinando i risultati.

$$\mathrm{T(n)} = \begin{cases} c & se\ n \leq k \\ D(n) + \sum_{i=1}^{h} T(n_i) + C(n) & se\ n > k \end{cases}$$

Divide + Impera + Combina

Equazioni di ricorrenza e loro risoluzione

Quando un algoritmo contiene una chiamata ricorsiva a sé stesso, il suo tempo di esecuzione spesso può essere descritto con una **equazione di ricorrenza** o ricorrenza, che esprime il tempo di esecuzione totale di un problema di dimensione n in funzione del tempo di esecuzione per input più piccoli.

Metodo iterativo

Espandi un'equazione di ricorrenza finché non vedi un pattern, sostituisci le costanti con k, mi metto nell'ipotesi del caso base, risostuisco a k n, fine.

Risoluzione mediante albero

Disegno livello per livello come varia l'input dato alla funzione ricorsiva, capisco quanto ancora va avanti e quale pattern fino ad arrivare al caso base, moltiplico il costo di ogni livello per l'altezza.

Master Theorem - Enunciato

Applicabile solo a ricorrenze della forma $\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + f(n)},\ dove\ f(n) = D(n) + C(n),$
$a \geq 1, b > 1\ e\ f$ è asintoticamente positiva.

Si confrontano f(n) e $n^{log_b a}$:
CASO I:

- f(n) cresce polinomialmente **più lentamente** di $n^{log_b a}$ (di un fattore n^{ε})

$f(n) = O(n^{log_b a - \varepsilon})$ per qualche costante $\varepsilon > 0$.
Soluzione: $T(n) = \Theta(n^{log_b a})$

CASO II:

- f(n) cresce **allo stesso modo** di $n^{log_b a}$

$f(n) = \Theta(n^{log_b a}\ lg^k n)$ per qualche costante $k \geq 0$.
Soluzione: $T(n) = \Theta(n^{log_b a}\ lg^{k+1} n)$

CASO III:

- f(n) cresce polinomialmente **più velocemente** di $n^{log_b a}$ (di un fattore n^{ε})
- f(n) **soddisfa la condizione** che:
  - $af(n/b) \leq cf(n)$ per qualche costante $c < 1\ \ e\ \ n > n_0$

$f(n) = \Omega(n^{log_b a + \varepsilon})$ per qualche costante $\varepsilon > 0$.
Soluzione: $T(n) = \Theta(f(n))$

Ricerca di un elemento in una collezione

Ricerca lineare (progettazione algoritmo, correttezza, limite inferiore al problema e complessità della soluzione progettata)

```
RICERCA-LINEARE(A, x, n)
1       for i = 0 to n                          O(n)
2               if x == A[i] then return A[i]   O(1)      Costo: O(n)
3       return "non esiste"                     O(1)
```

Ricerca binaria mediante divide-et-impera (progettazione algoritmo, correttezza e complessità)

```
RICERCA-BINARIA(A, x, p, q)
1       if p == q return "non esiste"                   O(1)
2       var i:Int = n / 2                               O(1)
3       if A[i] == x then return A[i]                   O(1)      Costo: O(logn)
4       if x > A[i] then RICERCA-BINARIA(A, x, p + i, q)  O(logn)
5       else RICERCA-BINARIA(A, x, p, q – i)            O(logn)
```

Heap

Proprietà strutturale e sulle informazioni (di massimo e di minimo)

Un heap è un albero binario quasi completo da sinistra a destra. Esistono due tipi di heap, max-heap e min-heap. La proprietà del max-heap è che la chiave di ogni nodo è maggiore o uguale della chiave dei figli (inverso per min-heap).
La **ricerca di un nodo** in un heap, non potendo escludere nessun sottoalbero costa O(n).

Costruire uno heap (correttezza e complessità)

Funzione **max-heapify**: supponiamo che il nodo su cui l'abbiamo chiamata abbia i sottoalberi heap. Eseguiamo heapification-downward fino a che il padre non diventa maggiore. Costo: O(logn).

Build-max-heap(A, n: numero di nodi): Escludendo le foglie (che sono già un heap) chiamo max-heapify su tutti i nodi interni a partire dal nodo più destra dell'ultimo livello e percorrendoli riga per riga da destra a sinistra.

```
BUILD-MAX-HEAP(A, n)
1       A.heap-size = n
2       for(i = ⌊n / 2⌋, i ≥ 1, i --)
3               MAX-HEAPIFY(A, i)
```

Dimostrazione:
Invariante: All'inizio di ogni iterazione del for, alle linee 2-3, ogniuno dei nodi i + 1, i + 2, … , n è la radice di un max-heap.
1) Inizializzazione: Invariante vera all'inizio del primo ciclo. Ogni nodo i + 1, i + 2, … , n è una foglia, quindi la proprietà è vera.
2) Mantenimento: Se vera prima di un ciclo, allora vera prima del prossimo. I figli del nodo i sono maggiori di i, quindi radici di un max-heap, perciò chiamare max-heapify è legale. Al prossimo giro decrementando di 1 i renderò l'affermazione vera dato che ho appena reso max-heap il nodo i.
3) Terminazione: Vera alla fine del ciclo. Alla fine, i = 0, perciò tutti gli altri nodi sono max-heap.

Costo:
Eseguiamo una chiamata ricorsiva dal costo O(logn) n volte, ma il costo dell'algoritmo non è davvero O(nlogn) poiché la maggior parte dei nodi (ovvero quelli vicini alla base dell'heap) hanno un'altezza molto minore di logn. Ogni chiamata a max-heapify costa O(h), perciò per calcolare il costo dividiamo il calcolo per livelli e per ogni livello moltiplichiamo il costo a quel livello per il numero di nodi:

$$numero\ di\ nodi\ al\ livello\ i\ =\ n_i\ =\ 2^i$$

$$T(n) = \sum_{i=0}^{h} n_i h_i \rightarrow\ Sostituiscoi\ valori\ di\ n_i\ e\ h_i \rightarrow \sum_{i=0}^{h} 2^i(h-i)$$

$$\sum_{i=0}^{h} 2^i(h-i)\ \rightarrow Moltiplico\ per\ \frac{2^h}{2^h}\ e\ scrivo\ 2^i\ come\ \frac{1}{2^{-i}} \rightarrow \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$

$$\sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h \rightarrow Sostituisco\ k = h + i \rightarrow\ 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$

$$2^h \sum_{k=0}^{h} \frac{k}{2^k}\ \rightarrow Maggioro\ 2^h\ sostituendo\ h\ con\ logn\ e\ lo\ tendo\ a\ \infty \rightarrow\ n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

$$Il\ risultato\ \text{è}\ 2, quindi\ O(2n)\ \simeq\ O(n)$$

Inserimento di un nodo e estrazione della radice

Funzione per inserire un nodo: Scendo tra i nodi tramite confronti fino a raggiungere la posizione giusta. Una volta lì faccio SWAP con i padri ricorsivamente fino a raggiungere un padre il cui valore non sia maggiore di quello del nodo da aggiungere. Questa procedura è detta **reheapification-up**.

Funzione per rimuovere un nodo: Scambio la radice con l'ultima foglia e la elimino, faccio swap verso il basso dalla radice fino a quando non mi ritrovo ad essere minore o uguale al padre e maggiore o uguale al figlio.

Heapsort (correttezza e complessità)

A: array da ordinare, eseguo build-max-heap(A) → prendo il primo elemento (il maggiore), lo metto alla fine → rieseguo build-max-heap sul resto dell'array → continuo per tutti gli elementi.

```
HEAPSORT(A, n)
1       BUILD-MAX-HEAP(A, n)
1       for(i = n; i ≥ 2; i --)
2               Scambia A[1] e A[i]
3               A.heap-size = A.heap-size --
4               MAX-HEAPIFY(A, 1)
```

Costo: build-max-heap impiega $O(n)$ e ciascuna delle n – 1 chiamate di max-heapify impiega $O(logn)$, per un totale di $O(nlogn)$.

Code di priorità

Ogni elemento è associato ad un valore (priorità). La chiave con la più bassa (o alta) priorità è estratta prima.

Implementazione

Tramite:	**Liste collegate**:	**Heap**:
- INSERT(S, x, k)	O(1)	O(logn)
- EXTRACT-MAX(S)	O(n)	O(logn)
- MAXIMUM(S)	O(n)	O(1)
- INCREASE-KEY(S, x, k)	O(1)	O(logn)

```
HEAP-MAXIMUM(A)
1   if A.heap-size == 0
2       error “underflow”
3   return A[1]
```

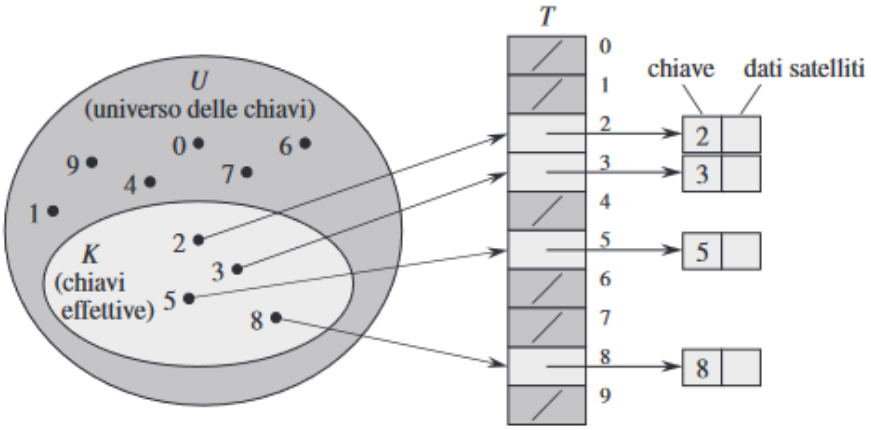
```
HEAP-INCREASE-KEY(A, i, key)
1   if key < A[i]
2       error “key minore di chiave corrente”
3   A[i] ← key
4   while i > 1 && A[PARENT(i)] < A[i]
5       do SCAMBIA A[i] e A[PARENT(i)]
6           i ← PARENT(i)
```

```
EXTRACT-MAX-HEAP(A)
1   max ← HEAP-MAXIMUM(A)
2   A[1] ← A[A.heap-size]
3   A.heap-size --
4   MAX-HEAPIFY(A, 1)
5   return
```

```
MAX-HEAP-INSERT(A, i, key)
1   if A.heap-size == n
2       error “overflow”
3   A.heap-size ++
4   A[A.heap-size] ← - ∞
5   HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Tabelle e funzioni hash

Dizionario: consiste di coppie di chiave-elemento. Può essere ordinato in base alle chiavi.



T: tavola a indirizzamento diretto che indicheremo con T[0 … m – 1].

Se una cella è vuota viene indicato.

In alcuni casi non è necessario memorizzare le chiavi in una tabella che indicizza ad elementi con tali chiavi, ma è possibile salvare i dati nella tabella stessa.

Invece di utilizzare indirizzamento diretto (elemento i in indice i) possiamo applicare una funzione sulla chiave per calcolare l’indice nel quale salvare questo dato.

Gestione collisioni

C’è un problema: due chiavi possono essere mappate nella stessa cella. Questo evento si chiama **collisione**.

Chaining (liste di trabocco)

Più elementi nello stesso indice concatenate tramite liste collegate. **Inserimento** e **cancellazione** costano O(1) se le liste sono doppiamente collegate, la **ricerca** è proporzionale alla lunghezza della lista.

Load factor λ: numero di elementi memorizzati nella tabella rispetto a quelli disponibili $\frac{n}{m}$. Il load factor affinché inserimento, ricerca e cancellazione costino O(1) deve essere n = k * m con k costante.

Cosa rende buona una funzione hash? È veloce, distribuisce i dati uniformemente, utilizza l'intera tabella.

Utilizzando il **modulo** come funzione hash dobbiamo evitare m potenza di due, altrimenti se m = 2^p il risultato di h(k) = p bit meno significativi di k. In genere scegliamo m primo non troppo vicino ad una potenza di due.

Un'altra funzione per evitare il problema delle potenze di due è il **multiplication method**:
1) moltiplichiamo k per una costante A, con 0 < A < 1, ed estraiamo la parte frazionaria.

$$kA - \lfloor kA \rfloor$$

2) moltiplichiamo questo valore per m e prendiamo l'approssimazione inferiore del risultato

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

Probing-Open hash

In ogni cella della tabella viene salvato direttamente l'elemento mappato lì (o nil) e se tentando di inserire un elemento si incappa in una cella già occupata verrà effettuato **probing** fino a trovare una cella vuota (a meno che la tabella non sia piena).
Con l'indirizzamento aperto si richiede che, per ogni chiave k, la sequenza di ispezione sia una **permutazione** di $\langle 0, 1, \dots, m-1 \rangle$, in modo che ogni posizione della tavola hash possa essere considerata.

Lineare

H(k, i) = (h'(k) + i) mod m

Al primo tentativo si accede all'indirizzo generato da h', se tale indirizzo è occupato verrà controllato h' + 1, e così via fino a controllarli tutti.
Il probing lineare presenta il problema dell'**addensamento primario**, ovvero la creazione di lunghe file di celle occupate che aumentano il tempo medio di ricerca.

Costo ricerca:

- **con successo**: $\frac{1}{\lambda} log \frac{1}{1-\lambda}$
- **senza successo**: al primo tentativo la possibilità di trovare l'elemento giusto è n/m, al secondo (escludendo l'elemento appena cercato) n – 1/ m – 1 e così via, sono tutti maggiorabili con n/m = λ^{i-1} risultante in $\frac{1}{1-\lambda}$.

Quadratico

H(k, i) = (h'(k) + c_1i + $c_2$$i^2$) mod m

h' è una funzione hash ausiliaria, c_1 e c_2 ≠ 0 sono costanti ausiliarie e i = 0, 1, … , m - 1. Le posizioni esaminate ai vari passi i dipendono in modo quadratico da i.

Doppio hash

$H(k, i) = (h_1(k) + ih_2(k)) \bmod m$

h_1 e h_2 sono funzioni hash ausiliarie. Il valore **$h_2(k)$** deve essere relativamente **primo** con la dimensione **m** della tavola hash perché venga ispezionata l'intera tavola.
Un modo pratico per garantire questa condizione è scegliere m potenza di 2 e definire h_2 in modo che produca sempre un numero dispari. Un altro modo è scegliere m primo e definire h_2 in modo che generi sempre un numero intero positivo minore di m.
Per esempio, potremmo scegliere m primo e porre

- $h_1(k) = k \bmod m$
- $h_2(k) = 1 + (k \bmod m')$

dove m' deve essere scelto un po' più piccolo di m (come m - 1).

Costi e complessità (dimostrazioni al caso medio, almeno l'idea intuitiva)

Analisi caso medio: Assumiamo un caso di **hashing uniforme semplice**, ovvero che la probabilità che un elemento sia mappato in una certa cella è uguale per ogni cella (indicando con n_j la lunghezza della lista T[j], $E[n_j] = \lambda = \frac{n}{m}$).

- **Ricerca senza successo**: con hashing uniforme semplice il tempo atteso per la ricerca è pari al tempo per il calcolo della funzione hash (che supponiamo essere O(1)) + lo scorrere l'intero array ($O(\frac{n}{m})$) per un totale di $\Theta(1 + \lambda)$.
- **Ricerca con successo**: a differenza di quella senza successo qui dobbiamo fermarci ad un certo punto; quindi, il costo è pari al numero di elementi prima di x più 1 (gli elementi prima di x sono tutti gli elementi inseriti dopo x). Indichiamo con x_i l'i-esimo elemento inserito nella tavola, definiamo la variabile casuale indicatrice $X_{ij} = I\{h(k_i) = h(k_j)\}$ (vale 1 se accade, 0 altrimenti). La probabilità che X_{ij} sia 1 è $\frac{1}{m}$.
  Dunque, il numero atteso di elementi esaminati è:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=j+1}^{n}X_{ij}\right)\right] \rightarrow \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=j+1}^{n}[X_{ij}]\right) \rightarrow \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=j+1}^{n}\frac{1}{m}\right) \rightarrow$$

$$1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \rightarrow 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \rightarrow 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \rightarrow$$

$$1+\frac{n-1}{2m} \rightarrow 1+\frac{\lambda}{2}-\frac{\lambda}{2n}$$

  In conclusione, il tempo totale (meno il tempo per calcolare la funzione hash) è

$$\Theta(2 + \lambda / 2 - \lambda / 2n) = \mathbf{\Theta (1 + \lambda)}$$

  Qual è e il significato di questa analisi? Se il numero di celle della tavola hash è almeno proporzionale al numero di elementi della tavola, abbiamo n = O(m) e, di conseguenza,

$$\lambda = n / m = O(m) / m = O(1)$$

Alberi binari

Rappresentazione alberi

Tramite **array**:

- ***Vantaggi***: accesso diretto ai nodi;
- ***Svantaggi***: l'altezza dell'albero deve essere nota (per poter allocare l'array), spreco di memoria (spazio allocato anche per i nodi vuoti), inserire e cancellare è complicato;

Tramite **liste collegate**:
Ogni nodo ha un collegamento per il figlio DX, uno per il figlio SX (opzionale: un collegamento al padre).

- ***Vantaggi***: l'altezza non deve essere nota, nessuno spreco di memoria, inserimento e cancellazione "ok";
- ***Svantaggi***: non abbiamo accesso diretto ai nodi, abbiamo bisogno di memoria aggiuntiva per i riferimenti DX e SX;

Definizione, e altezza nel caso peggiore e ottimo

Albero binario completo: tutti i nodi hanno grado due e le foglie sono sullo stesso livello.
Nodi: $2^{h+1} - 1$ **Foglie**: 2^h

Albero binario quasi completo: albero binario completo tranne che per almeno l'ultima foglia dell'ultimo livello.

Nel **caso peggiore** l'altezza di un albero binario è pari al numero di nodi (completamente sbilanciato a destra o a sinistra), nel **caso ottimo** (albero binario completo) è logn;

Visite: simmetrica, anticipata e posticipata

Anticipata: Radice, SX, DX [puntino a sinistra dei nodi]
Posticipata: SX, DX, Radice [puntino a destra dei nodi]
Simmetrica: SX, Radice, DX [puntino sotto ai nodi]

Alberi Binari di Ricerca

Definizione, e altezza nel caso peggiore e ottimo

Alberi binari che soddisfano la seguente proprietà:

- Chiavi ordinate;
- Per ogni nodo vale:
  - y.key ≤ x.key, ∀ y ∈ sottoalbero sx di x;
  - y.key ≥ x.key, ∀ y ∈ sottoalbero dx di x;

La ***ricerca simmetrica*** effettuata su questi alberi restituisce i ***valori ordinati***.

Interrogazioni (ricerca, Min, Max, Successore, Predecessore) e operazioni di modifica (inserimento, cancellazione) e loro costi

La **ricerca** in un ABR ricorsiva costa O(h) (h: altezza dell'albero).

La ricerca del **minimo** valore è implementata cercando la foglia più a sinistra (o il nodo che ha solo figli destri). La ricerca del **massimo** è uguale ma a destra. Costo O(h).

Successore: il successore di un nodo x è il nodo col valore maggiore a x più piccolo. Per trovarlo cerchiamo il minimo del sottoalbero DX (se ha almeno un figlio DX), altrimenti risaliamo la lista di antenati (padri) fino a raggiungere il primo nodo per il quale x sia figlio SX.
Per il predecessore il procedimento è analogo sostituendo a SX, DX e min con max. Costo: O(h)

Per **inserire un nodo** faccio confronti con i nodi interni e lo metto come foglia. Costo: O(h)

Per **cancellare un nodo**:

- se non ha figli: lo cancello;
- se ha un solo figlio: li scambio e cancello il figlio;
- se ha due figli: sostituisco con il predecessore o il successore;

Il predecessore non avrà figli DX, se avesse un figlio SX sostituirei il suo valore con quello del nodo da cancellare. Costo: O(h)

2-3 Alberi

Definizione, e altezza nel caso peggiore e ottimo

Un albero 2-4 è un albero che in cui ogni nodo interno ha 2 o 3 figli e tutti i cammini radice-foglia hanno la stessa lunghezza.

I dati sono memorizzati nelle foglie in ordine crescente da sinistra a destra.
I nodi interni v mantengono (al più) due info supplementari:

- **S[v]**: massima chiave nel sottoalbero del primo figlio di v;
- **M[v]**: (se ha tre figli) massima chiave nel sottoalbero del secondo figlio di v;

Lemma:
1) $2^{h+1} - 1 \leq n \leq \frac{3^{h+1}-1}{2}$
2) $2^h \leq f \leq 3^h$
Dimostrazione induttiva su h:
Base: se h = 0, l'albero ha un solo nodo.
Passo induttivo: diciamo che 1) e 2) sono vere fino ad una certa altezza h, dobbiamo dimostrare che siano vere per l'albero T di altezza h + 1, creiamo l'albero T', ovvero rimuoviamo l'ultimo livello a T (n' e f' sono i nodi e le foglie di T').
Sappiamo che:

- $2^{h+1} - 1 \leq n' \leq \frac{3^{h+1}-1}{2}$
- $2^h \leq f' \leq 3^h$

Ogni foglia in T' quanti figli ha? Tra 2 e 3, quindi $n = n' + f \geq 2^{h+1} - 1 + f$ sostituendo a f → 2^h (ovvero il secondo triangolo qui sopra) => $n \geq 2^{h+1} - 1 + 2^h$ e quindi $n \geq 2^{h+2} - 1$ (stessa cosa per la parte destra della disequazione).
L'altezza sarà quindi dell'ordine di $\Theta(logn)$.

Operazioni di ricerca, inserimento e cancellazione e loro costi

Ricerca di un elemento:

- Nodo con **tre figli**:
  - Confrontiamo x con S e M:
    - ≤ S → cerca in figlio 1;
    - ≤ M → cerca in figlio 2;
    - > M → cerca in figlio 3;
- Nodo con **due figli**:
  - Confrontiamo x con S:
    - ≤ S → cerca in figlio 1;
    - > S → cerca in figlio 2;

Costo: $\Theta(logn)$.

Inserimento di un elemento:
Scendiamo tra i nodi come abbiamo fatto prima fino a raggiungere le foglie, troviamo il posto giusto per il nodo.

- Se il padre del posto trovato ha **due figli**: aggiungo il nodo e aggiorno S e M del nodo e di tutti i suoi antenati;
- Se il padre del posto trovato ha **tre figli**: aggiungiamo il nodo creandone così uno da 4 figli ed eseguo SPLIT su di esso dividendolo in due nodi ogniuno con due figli. Se splittando il nodo finisco con aver creato un nodo con quattro figli continuo ad eseguire split in maniera ricorsiva verso l'alto. Se arrivo fino alla radice la splitto e creo un nuovo nodo radice.

Costo: $\Theta(logn)$.

Cancellare un elemento:
Scendiamo ancora come fatto in precedenza e arriviamo alla foglia da eliminare:

- Se il padre della foglia ha **tre figli**: elimino la foglia;
- Se il padre della foglia ha **due figli**: unisco il padre della foglia con suo fratello
  - Se il fratello ha **tre figli**: rubo un figlio e ho due nodi da due figli ciascuno;
  - Se il fratello ha **due figli**: li unisco creando un nodo con tre figli. Se unendo i due fratelli il padre finisse per avere un solo figlio continuerei in maniera ricorsiva verso l'alto ad unire fratelli, se la procedura raggiunge la radice lasciandola con un solo figlio la eliminerei facendo diventare tale figlio la nuova radice;

Costo: $\Theta(logn)$.

Programmazione dinamica

Calcolo efficiente di una funzione ricorsiva mediante memorizzazione dei suoi valori intermedi in una tabella detta **tabella di programmazione dinamica**. I casi in cui è utile utilizzare questo approccio sono i problemi ricorsivi che lavorano sulle stesse parti di input

Longest Common Subsequence

Una sequenza S di lunghezza k è sottosequenza di una sequenza A di lunghezza n se e solo se può essere ottenuta da A cancellando alcuni elementi. Una sottosequenza è comune quando è presente in due sequenze distinte.

Partiamo immaginando di sapere che la LCS vale k in un prefisso di A e di B
A questo punto il passo successivo è controllare la prossima lettera:

- Se sono uguali la LCS varrà k + 1
- Se diversi resta k

Procedimento:

- prima riga e prima colonna inizializzati a 0;
- se $x_i == y_j$ sommo 1 al valore in alto a SX;
- se $x_i \neq y_j$ max{SX, alto};

Tempo: O(mn), **Spazio:** O(mn)

Edit Distance

Vogliamo trovare l'allineamento ottimo fra le due sequenze, cioè quello che minimizza la distanza fra di esse. Possiamo inserire spazi nelle sequenze, la distanza è la somma delle distanze fra le coppie di caratteri allineati (uguali vale 0, diversi vale 1).

1) $x_1, x_2, \dots, x_i$ VS $y_1, y_2, \dots, y_j$ → $M[i-1, j-1] + p(i, j)$ $\begin{cases} 0 & se\ uguali \\ 1 & se\ diversi \end{cases}$

2) $x_1, x_2, \dots, -$ VS $y_1, y_2, \dots, y_j$ → $M[i-1, j] + 1$

3) $x_1, x_2, \dots, x_i$ VS $y_1, y_2, \dots, -$ → $M[i, j-1] + 1$

Procedimento:

- inizializzo prima riga e prima colonna [0, 1, … , i] e [0, 1, … , j];
- MIN{(1), (2), (3)};

Zaino

Abbiamo uno zaino e vogliamo riempirlo con un po' di oggetti. Ogni oggetto ha un peso e un valore, possiamo avere al massimo un certo peso e vogliamo massimizzare il guadagno.

Analizziamo tutte le opportunità di peso trasportabile (se potessimo portare 1, se potessimo portare 2 e così via) e per ogniuna di esse analizziamo la migliore combinazione analizzando prima solo il primo elemento, poi i primi due, poi i primi tre, decidendo, con l'aumentare della disponibilità di peso, se prendere un oggetto in più o cambiare l'oggetto appena preso.

Greedy (zaino frazionario)

Prendiamo la maggiore quantità possibile di oggetto dal valore più alto, se lo prendo tutto passo al secondo e così via.

Grafi

BFS (Breadth First Search)

```
BFS(G, s)
1       for ogni vertice u ∈ G.V – {s}
2               u.color = WHITE
3               u.d = 1
4               u.π = NIL
5       s.color = GRAY
6       s.d = 0
7       s.π = NIL
8       Q = ∅
9       ENQUEUE(Q, s)
10      while Q ≠ ∅
11              u = DEQUEUE(Q)
12              for ogni v ∈ G.Adj[u]
13                      if v.color == WHITE
14                              v.color = GRAY
15                              v.d = u.d + 1
16                              v.π = u
17                              ENQUEUE(Q, v)
18              u.color = BLACK
```

Esplora un grafo trasformandolo in un albero (di copertura). Espande la frontiera in ampiezza.

Sceglie un vertice sorgente come radice e esplora prima i nodi a distanza uno, poi a distanza due...
Ogni nodo ha tre possibili stati:

- **Bianco**: non visitato;
- **Grigio**: messo in coda (visitato);
- **Nero**: rimosso dalla coda (visitati tutti i figli);

Ogni nodo ha due valori, **d** (distanza dalla radice) e **p** (nodo genitore).
Costo: O(n + m)

Lemma 22.1
Se G = (V, E) è un grafo orientato o non orientato e s ∈ V è un vertice arbitrario, allora per qualsiasi arco (u, v) ∈ E si ha **δ(s, v) ≤ δ(s, u) + 1**

Dimostrazione: Se u è un vertice raggiungibile da s, allora lo è anche v. In questo caso, il cammino minimo da s a v non può essere più lungo del cammino minimo da s a u seguito dall'arco (u, v), quindi la disuguaglianza è valida. Se u non è raggiungibile da s, allora δ(s, u) = ∞ e la disuguaglianza è valida.

Lemma 22.2
Sia G = (V, E) un grafo orientato o non orientato e supponiamo che BFS venga eseguita sul grafo G da un dato vertice sorgente s ∈ V . Allora, al termine della procedura, per ogni vertice v ∈ V , il valore v.d calcolato da BFS soddisfa la relazione v.d ≥ δ(s, v).

Dimostrazione: Applichiamo l'induzione sul numero di operazioni ENQUEUE. La nostra ipotesi induttiva è che v.d ≥ δ(s, v) per ogni v ∈ V. Il caso base dell'induzione è la situazione che si ha subito dopo che il vertice s è stato inserito nella coda (riga 9 di BFS). L'ipotesi induttiva è vera qui, perché s.d = 0 = δ(s, s) e v.d = ∞ ≥ δ(s, v) per ogni v ∈ V – {s}.
Per il passo induttivo, consideriamo un vertice bianco v che viene scoperto durante la visita a partire da un vertice u. L'ipotesi induttiva implica che u.d ≥ δ(s, u). Per l'assegnazione eseguita nella riga 15 e per il Lemma 22.1, si ha

v.d = u.d + 1
≥ δ(s, u) + 1
≥ δ(s, v)

Il vertice v viene inserito nella coda e non sarà mai più reinserito perché viene anche colorato di grigio e la clausola then (righe 14–17) viene eseguita soltanto per i vertici bianchi. Quindi, il valore di v.d non cambierà più e l'ipotesi induttiva resta valida.

Corollario 22.4
Supponiamo che i vertici v_i e v_j siano inseriti nella coda durante l'esecuzione di BFS e che v_i sia inserito nella coda prima di v_j. Allora $v_i.d \leq v_j.d$ nell'istante in cui v_j viene inserito nella coda.

Teorema 22.5 (Correttezza della visita in ampiezza)
Sia G = (V, E) un grafo orientato o non orientato e supponiamo che la procedura BFS venga eseguita sul grafo G da un dato vertice sorgente s ∈ V. Allora, durante la sua esecuzione, BFS scopre tutti i vertici v ∈ V che sono raggiungibili dalla sorgente s e, alla fine dell'esecuzione, v.d = δ(s, v) per ogni v ∈ V. Inoltre, per qualsiasi vertice v ≠ s che è raggiungibile da s, uno dei cammini minimi da s a v è un cammino minimo da s a v.π seguito dall'arco (v.π, v).

Dimostrazione: Supponiamo, per assurdo, che qualche vertice riceva un valore d che non è uguale alla distanza del suo cammino minimo. Sia v il vertice con il minimo δ(s, v) che riceve questo valore errato d; chiaramente v ≠ s. Per il Lemma 22.2, v.d ≥ δ(s, v), e quindi v.d > δ(s, v). Il vertice v deve essere raggiungibile da s perché, se non lo fosse, allora δ(s, v) = ∞ ≥ v.d. Sia u il vertice che precede immediatamente v in un cammino minimo da s a v, cosicché δ(s, v) = δ(s, u) + 1. Poiché δ(s, u) < δ(s, v) e per come abbiamo scelto v, deve essere u.d = δ(s, u). Ponendo insieme queste proprietà, si ha v.d > δ(s, v) = δ(s, u) + 1 = u.d + 1 (22.1).

Adesso consideriamo il momento in cui BFS sceglie di eliminare il vertice u dalla coda Q (riga 11). In quel momento, il vertice v può essere bianco, grigio o nero. Dimostreremo che in ciascuno di questi casi, si arriva a una contraddizione della disuguaglianza (22.1).

- Se il vertice v è **bianco**, allora la riga 15 imposta v.d = u.d + 1, che contraddice la disuguaglianza (22.1).
- Se il vertice v è **nero**, allora era stato già rimosso dalla coda e, per il Corollario 22.4, si ha v.d ≤ u.d, che contraddice ancora la disuguaglianza (22.1).
- Se il vertice v è **grigio**, allora era stato colorato di grigio dopo la cancellazione dalla coda di qualche vertice w, che era stato cancellato da Q prima di u e per il quale v.d = w.d + 1. Per il Corollario 22.4, però, w.d ≤ u.d, e quindi si ha v.d = w.d + 1 ≤ u.d + 1, che contraddice ancora la disuguaglianza (22.1).

Dunque, possiamo concludere che v.d = δ(s, v) per ogni v ∈ V. Tutti i vertici raggiungibili da s devono essere scoperti, perché se non lo fossero, avrebbero ∞ = v.d > δ(s, v). Per concludere la dimostrazione del teorema, osserviamo che, se v.π = u, allora v.d = u.d + 1. Quindi, possiamo ottenere un cammino minimo da s a v prendendo un cammino minimo da s a v.π e poi attraversando l'arco (v.π, v).

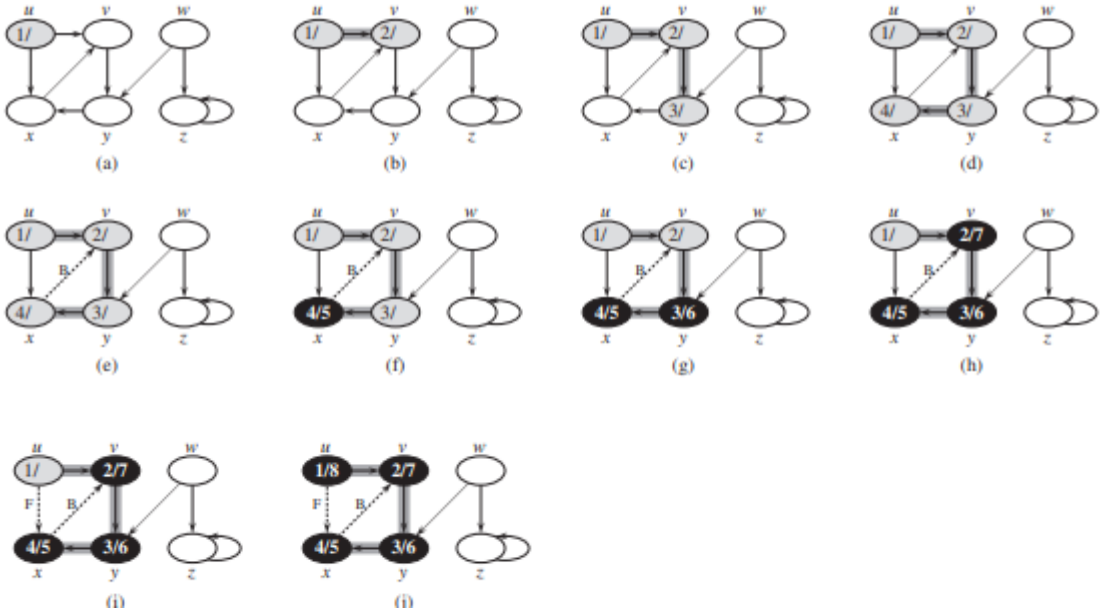
DFS (Depth First Search)

```
DFS(G)
1       for ogni vertice u ∈ G.V
2               u.color = WHITE
3               u.π = NIL
4       time = 0
5       for ogni vertice u ∈ G.V
6               if u.color == WHITE
7                       DFS-VISIT(G, u)

DFS-VISIT(G, u)
1       time = time + 1                 // Il vertice bianco u e stato appena scoperto. `
2       u.d = time
3       u.color = GRAY
4       for ogni v ∈ G.Adj[u]           // Ispeziona l'arco (u, v)
5               if v.color == WHITE
6                       v.π = u
7                       DFS-VISIT(G, v)
8       u.color = BLACK                 // Colora di nero u, visita completata.
9       time = time + 1
10      u.f = time
```

Partendo dalla radice mette in coda un suo vicino, poi lo esplora mettendo in coda un suo vicino, e così fino a raggiungere un nodo senza vicini (o con soli vicini già esplorati), a questo punto torna indietro e rimuove dalla coda i nodi nell'ordine inverso in cui sono stati scoperti.

u.d = tempo in cui il nodo u è stato scoperto (è diventato grigio);
u.f = tempo in cui il nodo u è stato completamente esplorato (è diventato nero);



Teorema 22.7 (Teorema delle parentesi)
In qualsiasi visita in profondità di un grafo G = (V, E) (orientato o non orientato), per ogni coppia di vertici u e v, è soddisfatta una sola delle seguenti tre condizioni:

- Gli intervalli [u.d, u.f] e [v.d, v.f] sono completamente disgiunti; inoltre, u e v non sono discendenti l'uno dell'altro nella foresta DF.
- L'intervallo [u.d, u.f] è interamente contenuto nell'intervallo [v.d, v.f]; inoltre u è un discendente di v in un albero DF.
- L'intervallo [v.d, v.f] è interamente contenuto nell'intervallo [u.d, u.f]; inoltre v è un discendente di u in un albero DF.

Dimostrazione: Iniziamo con il caso in cui u.d < v.d. Ci sono due sotto casi da considerare, a seconda che v.d < u.f oppure no. Il primo sotto caso si verifica quando v.d < u.f; quindi, v è stato scoperto mentre u era ancora grigio. Questo implica che v è un discendente di u. Inoltre, poiché v è stato scoperto più recentemente di u, vengono ispezionati tutti i suoi archi uscenti e l'ispezione di v viene completata prima che la visita riprenda da u e venga completata l'ispezione di u. In questo caso, quindi, l'intervallo [v.d, v.f] è interamente contenuto nell'intervallo [u.d, u.f]. Nel secondo sotto caso, u.f < v.d e, per la disuguaglianza (22.2), u.d < u.f < v.d < v.f ; quindi, gli intervalli [u.d, u.f] e [v.d, v.f] sono disgiunti. Poiché gli intervalli sono disgiunti, nessuno dei due vertici è stato scoperto mentre l'altro era grigio, quindi nessuno dei due vertici è un discendente dell'altro. Il caso in cui v.d < u.d è simile, perché basta invertire i ruoli di u e v nella precedente discussione.

Teorema 22.8 (Annidamento degli intervalli dei discendenti)
Il vertice v è un discendente proprio del vertice u nella foresta DF per un grafo G (orientato o non orientato) se e soltanto se u.d < v.d < v.f < u.f.

Dimostrazione: E' una conseguenza immediata del Teorema 22.7.

Teorema 22.10
In una visita in profondità di un grafo non orientato G, gli archi di G possono essere archi d'albero o archi all'indietro.

Dimostrazione: Sia (u, v) un arco arbitrario di G e, senza perdere in generalità, supponiamo che u.d < v.d. Allora, il vertice v deve essere scoperto e completato prima che sia completato u (mentre u è grigio), perché v si trova nella lista di adiacenza di u. Se l'arco (u, v) viene ispezionato prima nella direzione da u a v, allora v è un vertice non scoperto (bianco) fino a quel momento, perché altrimenti avremmo già ispezionato questo arco nella direzione da v a u. Quindi, (u, v) diventa un arco d'albero.

Se (u, v) viene ispezionato prima nella direzione da v a u, allora (u, v) è un arco all'indietro, perché u è ancora grigio quando l'arco viene esplorato per la prima volta.

Topological sort (vedi *Cormen,* con dimostrazione del Lemma 22.11 e del Teorema 22.12)

Per poter eseguire questo sorting il grafo deve essere **aciclico**.
Eseguo DFS per calcolare tutti i tempi dei nodi e non appena un nodo termina lo metto in testa alla lista. Il risultato sarà la lista in ordine decrescente dei tempi f.

Lemma 22.11
Un grafo orientato G è aciclico se e soltanto se una visita in profondità di G non genera archi all'indietro.

Dimostrazione: => supponiamo che ci sia un arco all'indietro (u, v). Allora, il vertice v è un antenato del vertice u nella foresta DF. Quindi, esiste un cammino che va da v a u nel grafo G e l'arco all'indietro (u, v) completa un ciclo.
<= supponiamo che il grafo G contenga un ciclo c. Dimostriamo che una visita in profondità di G genera un arco all'indietro. Sia v il primo vertice che viene scoperto in c e sia (u, v) l'arco precedente in c. Al tempo v.d, i vertici di c formano un cammino di vertici bianchi da v a u. Per il teorema del cammino bianco, il vertice u diventa un discendente di v nella foresta DF. Dunque, (u, v) è un arco all'indietro.

Teorema 22.12
TOPOLOGICAL-SORT(G) produce un ordinamento topologico di un grafo orientato aciclico G.

Dimostrazione: Supponiamo che la procedura DFS venga eseguita su un dato dag G = (V, E) per determinare i tempi di completamento dei suoi vertici. È sufficiente dimostrare che per una coppia qualsiasi di vertici distinti u, v ∈ V, se esiste un arco in G che va da u a v, allora v.f < u.f. Consideriamo un arco qualsiasi (u, v) ispezionato da DFS(G). Quando questo arco viene ispezionato, il vertice v non può essere grigio, perché altrimenti v sarebbe un antenato di u e (u, v) sarebbe un arco all'indietro, contraddicendo il Lemma 22.11. Quindi, il vertice v deve essere bianco o nero. Se v è bianco, diventa un discendente di u e quindi v.f < u.f . Se v è nero, la sua ispezione è stata già completata, quindi il valore di v.f è già stato impostato. Poiché stiamo ancora ispezionando dal vertice u, dobbiamo ancora assegnare un'informazione temporale a u.f e, quando lo faremo, avremo ancora v.f < u.f . Quindi, per qualsiasi arco (u, v) nel dag, si ha v.f < u.f , e questo dimostra il teorema.

Cammini minimi

Dijkstra

Eseguiamo BFS ma nel momento in cui dobbiamo decidere l'arco da prendere lo facciamo in base al costo di quelli disponibili.
1) tutti i nodi partono con un valore distanza pari a infinito;
2) partendo dalla radice metto tutti i nodi vicini in lista e la ordino;
3) prendo il primo in lista e ripeto il passo 2) con l'accortezza di sommare il costo dell'arco "radice-nodo" al costo dell'arco "nodo-nodo da controllare";
4) continuo fin quando la lista non è vuota;

Costo: aggiungere un nodo alla lista O(log|V|), estrarli O(log|V|), modificarne le proprietà O(log|V|), per un totale di $O((|V| + |E|) \log |V|)$

Bellman-Ford

Applica il rilassamento su ogni arco **|V| - 1 volte** (il numero totale di archi) se alla volta successiva trova ancora un possibile miglioramento vuol dire che c'è un arco di costo complessivo negativo e restituisce un valore booleano che lo indica.
1) tutti i nodi partono con un valore distanza pari a infinito;
2) controllo tutti gli archi per i nodi adiacenti alla radice e metto tali nodi nell'insieme dei nodi da prendere;
3) controllo tutti i nodi adiacenti a quelli appena presi, se il costo per arrivare a tali nodi (sommato al costo ai primi dalla radice) è minore del loro costo (inizialmente infinito) prenderò quell'arco;
4) continuo |V| - 1 volte;

Costo: O(VE)

P – NP

Problema della fermata: Dato un generico algoritmo (o programma) in input, esso termina o va in loop?
Supponiamo esista un algoritmo TERMINA(A, D) che, in tempo finito, restituisca:

- Va in loop
- Non va in loop

OSS.
1) Una sequenza di simboli può essere interpretata come dato o programma;
2) Un programma può essere dato in input a un altro programma.
Queste due osservazioni implicano che sia legale invocare TERMINA(A, D).

Assumendo che tutto ciò sia vero, immaginiamo questo algoritmo:

```
PARADOSSO(A){
    If(TERMINA(A, A)) LOOP()
    else false
}
```

A questo punto cosa succederebbe se eseguissi (PARADOSSO(PARADOSSO))? ***Contraddizione***
Non si può creare un programma che mi dice se un qualsiasi programma termini o meno, quindi il problema si dice **intrattabile**.

Problemi trattabili: problemi per i quali si può provare che la soluzione esista ed è di costo polinomiale. Tutti gli altri sono intrattabili.

Un **problema** è una relazione I (Insieme delle istanze in ingresso) x S (Insieme delle soluzioni). Possiamo inoltre immaginare un predicato che presa un'istanza $x \in I$ ed una soluzione $s \in S$, restituisce 1 se $(x, s) \in P$.

Problemi di decisione:

- Richiedono una risposta binaria;
- Istanze positive ($(x, 1) \in P$) p negative ($(x, 0) \in P$);

Problemi di ricerca:

- Richiedono di trovare una soluzione tale che $(x, s) \in P$;

Problemi di ottimizzazione:

- Data un'istanza x si vuole trovare la migliore soluzione s tra tutte le possibili s per cui $(x, s) \in P$

La teoria della **complessità computazionale** è definita principalmente in termini di problemi di decisione (essendo la soluzione binaria, non ci preoccupiamo del tempo richiesto per restituirla, tutto il tempo è speso per il calcolo).
Ogni problema di ottimizzazione si può trasformare in un problema di decisione poiché, in possesso di una soluzione, chiediamo solo di dire se va bene o meno. I problemi decisionali sono quindi un limite inferiore per i rispettivi problemi di ottimizzazione (per quanto riguarda la complessità).
Data una qualunque funzione f(n) chiamiamo **TIME**(f(n)) e **SPACE**(f(n)) gli insiemi decisionali che possono essere risolti rispettivamente in tempo e spazio O(f(n)).

Problema di soddisfacibilità di formule booleane: Verificare se esiste un'assegnazione di valori di verità alle variabili che rende l'espressione vera. Se la formula dice ∃ basta trovare una combinazione, se invece dice ∀ dobbiamo trovare tutte le possibili combinazioni, e neanche una macchina di Turing che prevede il futuro può certificare una soluzione in tempi polinomiali.

Definizioni di P, NP, NP-arduo, NP-completo

Classe P: Tutti gli algoritmi che hanno un costo in tempo e spazio polinomiale [n^c ($c, n_0 > 0$)]

Algoritmi non deterministici: oltre alle normali istruzioni, può eseguire istruzioni tipo INDOVINA z ∈ {0, 1} oppure RANDOM(0, 5). Il valore di z influenza la prosecuzione della computazione.

Classe NP: è la classe di problemi decisionali risolvibili in tempo polinomiale da una macchina di Turing non deterministica nella dimensione n dell'istanza di ingresso.

Riducibilità polinomiale: Dati due problemi decisionali P1⊆ I1 x{0, 1} e P2 ⊆ I2 x{0, 1} diremo che P1 è riducibile polinomialmente a P2 (P1 < P2) se esiste una funzione f: I1 → I2 tale che f sia calcolabile in tempo polinomiale e che per ogni istanza x di P1 ed ogni soluzione s ∈ {0, 1}
⇒ (x, s) ∈ P1 ⇔ $(f(x)) \in P2$. Ovvero traduco gli input del primo problema in modo che sia comprensibile per un altro problema. I risultati con input originale per P1 saranno uguali ai risultati con gli input "tradotti" per P2.
OSS. Anche se la funzione di traduzione devo eseguirla un numero polinomiale di volte la definizione di riducibilità resta valida.

Classe NP-arduo: un problema decisionale P1 si dice NP-arduo se ogni problema Q ∈ NP è riducibile polinomialmente a P1.

Classe NP-completo: un problema decisionale P1 si dice NP-completo se:

- Appartiene a NP (certificato in tempo polinomiale);
- È arduo;

Esempio di riduzione 3SAT

Problema della clique: Una clique è un grafo con tutti i nodi connessi tra loro.
Controlla se in un grafo c'è una clique.
È NP?
Diamo un certificato (un insieme di vertici che forma una clique).
È NP-arduo?
Mostriamo che 3-SAT (versione di SAT dove ogni clausola è formata da 3 letterali, anche 3-SAT è NP-completo) si riduce ad esso:
3-SAT < CLIQUE
Data un'espressione booleana FNC (Formula Normale Congiunta "senza ∀ e ∃") con k clausole, costruire in tempo polinomiale un grafo G che contiene una clique di dimensione k se e solo se F è soddisfacibile.

Costruiamo un grafo utilizzando formule booleane:

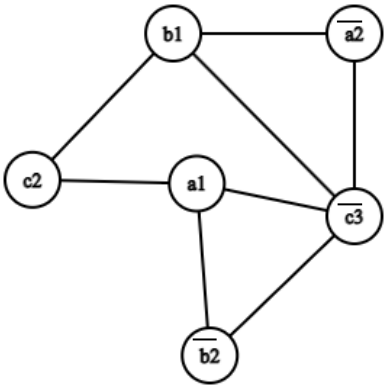
- Ad ogni letterale in ciascuna clausola corrisponde un vertice in G;
- Due letterali adiacenti in G (due nodi collegati):
  - Appartengono a clausole diverse;
  - Possono essere veri contemporaneamente;

$F = (a \vee b) \wedge (\overline{a} \vee \overline{b} \vee c) \wedge \overline{c}$ $G = (V, E)$ $(x_i, y_i) \in E \leftrightarrow i \neq j \wedge x \neq \overline{y}$

V = {a1, b1, a2, b2, c2, c3}

Mostriamo che F è soddisfacibile se e solo se G contiene una clique di k vertici:

Abbiamo (in questo caso) k = 3, abbiamo tre clausole; quindi, nel momento in cui le tre clausole valgono 1 vuol dire che tre nodi adiacenti sono connessi e formano quindi una clique.



Σ: Sigma, **δ**: Delta, **γ**: Gamma, **ρ**: Rho**, σ**: Sigma, **Ω**: Omega, **Θ**: Theta, **λ**: Lambda