Contents

Appunti di Algoritmi e Strutrure dati

Marco Zanchin

February 2023

1 Introduction

1.1 Nozione di algoritmo

Informalmente, un algoritmo è un procedimento formato da una sequenza finita di istruzioni che trasforma uno o più valori di ingresso in uno o più valori di uscita. Un algoritmo definisce quindi implicitamente una funzione dall'insieme degli input a quello degli output e nel contempo descrive un procedimento effettivo che permette di determinare per ogni possibile ingresso i corrispondenti valori di uscita.

$$f_\pi : D_\pi \Rightarrow S_\pi$$

Esempio

$$f_{primalita} : \mathbf{N} \Rightarrow \{0, 1\}$$

- $f_{primalita}(10) = 0$
- $f_{primalita}(7) = 1$

1.2 Efficienza algoritmica

L'efficienza di un algoritmo riguarda la **quantità di risorse** che vengono impiegate per la soluzione di un problema.

- Efficienza in termini di tempo: il numero complessivo di istruzioni eseguite
- Efficienza in termini di spazio: quanta memoria viene occupata in seguito all'esecuzione di un algoritmo.

Il concetto di efficienza viene formalizzato tramite la nozione di **Complessità**.

Esprimiamo la complessità di un algoritmo ALG attraverso una funzione:

$$T_{ALG} : \mathbf{N} \Rightarrow \mathbf{N}$$

che indica per ogni valore intero n la quantità di tempo/spazio impiegata dall'algoritmo per elaborare dati di dimensione n

1.2.1 Casi di complessità

- La complessità **caso peggiore** $T^w_{ALG}(n)$ è una funzione che fornisce il comportamento dell'algoritmo considerando l'istanza più **sfavorevole**, il **worst case**.
- La complessità **caso migliore** $T^b_{ALG}(n)$ è una funzione che fornisce il comportamento dell'algoritmo considerando l'istanza più **favorevole**, il **best case**.
- La complessità **media** $T^a_{ALG}(n)$ è una funzione che fornisce il comportamento medio dell'algoritmo, ossia l' **average case**, calcolato eseguendo la media dei comportamenti sulle istanze aventi dimensione uguale.

  $$T^a_{ALG}(n) = \sum_{i=n} p_i c_i$$

  dove p_i è la **probabilità** dell'istanza i e c_i è la quantità di **risorse impiegate** per elaborare l'istanza i.

Indicando con In_n l'insieme dei dati in ingresso di dimensione n si ha:

$$T^a_{ALG}(n) = \sum_{i \in ln_n} p_i c_i = \sum_{k \geq 0} P_k \cdot k$$

Passiamo dunque da una somma sulle istanze a una somma di costi raccogliendo a fattor comune il costo e sommando la probabilità di tutte le istanze che hanno quel costo.
Dove

$$P_k = \sum_{i \in ln_n \ C_i = k} P_i$$

è la **somma della probabilità di tutte le istanze che hanno costo k**.
Ponendo

$$\sigma_{n,k} = \#\{i \in In_n \mid C_i = k\}$$

(Ossia il numero di istanze che hanno costo k e dimensione n)
E ipotizzando che tutte le istanze dentro l'insieme In_n siano equiprobabili otteniamo:

$$T^a_{ALG}(n) = \sum_k k \cdot \frac{\sigma_{n,k}}{\#In_n}$$

1.2.2 Complessità asintotica

La **complessità asintotica** di un algoritmo alg è il comportamento della funzione $T_{alg}(n)$ per valori di limite o particolari.

Osservazione 1.1. Non è detto che un algoritmo A1 con complessità asintotica minore di quella di un altro algoritmo A2 (per lo stesso problema) si comporti meglio per ogni dimensione.

2 Nozioni preliminari

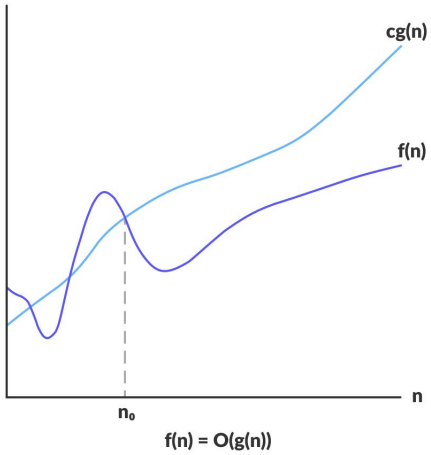
2.1 Notazioni asintotiche

2.1.1 Big O notation

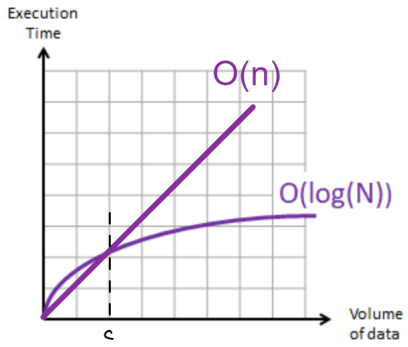
Definition 2.1 (Definizione formale di O grande)**.** Una funzione $f : \mathbf{N} \Rightarrow \mathbf{N}$ è detta **o-grande** di una funzione $g : \mathbf{N} \Rightarrow \mathbf{N}$, $f(n) = O(g(n))$, se esistono un intero n_0 e una costante $c > 0$ per cui

$$\forall n > n_0 \;\; f(n) \leq cg(n)$$

La notazione O grande rappresenta il limite superiore del tempo di esecuzione di un algoritmo. Quindi produce il **caso peggiore** di complessità.



Osservazione 2.1. Non possiamo dominare una funzione lineare $O(n)$ con una del tipo $O(log(n))$ perchè ad un certo punto c la funzione lineare sarà più lenta della logaritmica.



2.1.2 Notazione Ω

Definition 2.2 (Notazione Ω)**.** Una funzione $f : \mathbf{N} \Rightarrow \mathbf{N}$ è detta **Ω-grande** di una funzione $g : \mathbf{N} \Rightarrow \mathbf{N}$, $f(n) = \Omega(g(n))$, se esistono un intero n_0 e una costante $c > 0$ per cui

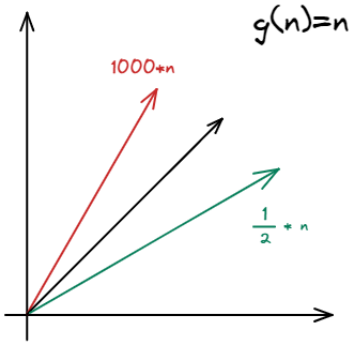
$$\forall n > n_0 \;\; f(n) \geq cg(n)$$

Questo tipo di notazione rappresenta il **best case**.

2.1.3 Notazione θ

Definition 2.3 (Notazione θ)**.** Una funzione $f : \mathbf{N} \Rightarrow \mathbf{N}$ è detta **θ-grande** di una funzione $g : \mathbf{N} \Rightarrow \mathbf{N}$, $f(n) = \theta(g(n))$, se esistono un intero n_0 e due costanti $c, d > 0$ per cui

$$\forall n > n_0 \;\; cg(n) \leq f(n) \leq dg(n)$$



Definition 2.4 ($\sim$ e o piccolo)**.** $f(n)$ è asintotica a $g(n)$, $f(n) \sim g(n)$, se:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 1$$

$f(n)$ è o piccolo di $g(n)$, $f(n) = o(g(n))$, se:

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

2.2 Proprietà delle notazioni asintotiche

- Se f $(n) = O(g(n))$ allora $g(n)$ è un limite asintotico superiore per $f(n)$
  $f(n) = O(g(n))$ se e solo se $g(n) = \Omega(f(n))$
- $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$
- $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$
  allora $f_1(n) + f_2(n) = O(g_1(n) + g_2(n)$

- $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$
  allora $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n)$
- $f_1(n) - f_2(n) \neq O(g_1(n) - g_2(n))$

Osservazione

$$O(n^3 + n^2) = O(n^3)$$

In notazione O grande, ci interessa il termine dominante della funzione all'avvicinarsi di n all'infinito.
Nell'espressione $O(n^3 + n^2)$, il termine n^3 è il termine dominante perché cresce molto più velocemente di n^2 all'avvicinarsi di n all'infinito. Quindi, possiamo dire che $O(n^3 + n^2)$ è lo stesso di $O(n^3)$.
Questo è perché al limite all'avvicinarsi di n all'infinito, qualsiasi polinomio con grado inferiore diventa trascurabile rispetto al polinomio con il grado più grande.

2.3 La relazione d'equivalenza Θ

- **Transitività**

  Se $f(n)$ è $\Theta(g(n))$ e $g(n)$ è $\Theta(h(n))$ allora $f(n)$ è $\Theta(h(n))$

- **Riflessività**

  $f(n)$ è $\Theta(f(n))$

- **Simmetria**

  Se $f(n)$ è $\Theta(g(n))$ allora $g(n)$ è $\Theta(f(n))$

La relazione di equivalenza rappresenta tutte le funzioni di crescita.
O e Ω godono solo della proprietà transitiva e riflessiva.

2.3.1 Proprieta di $\sim$ e o piccolo

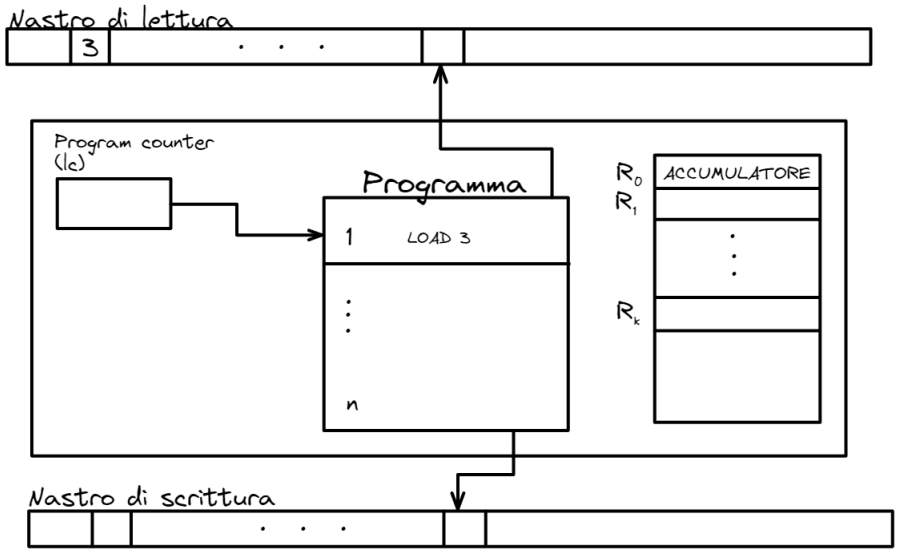
3 Modelli di calcolo

3.1 Introduzione

In un algoritmo si vogliono studiare la *semantica operazionale* e la *complessità* di esso. Entrambi dipendono da una descrizione formale del modello su cui l'algoritmo viene eseguito. Il modello di riferimento 'e quello della **macchina RAM**.

3.2 La macchina RAM

Definition 3.1 (La macchina RAM)**.** Una macchina RAM (Random Access Machine) è un **modello teorico** di un computer che serve come rappresentazione astratta di come funzionano algoritmi e calcoli. Consiste in un'ampia matrice indicizzata di posizioni di memoria a cui è possibile accedere in lettura e scrittura in tempo costante, consentendo l'accesso casuale a qualsiasi posizione di memoria. Oltre alla matrice di memoria, una macchina RAM dispone di un insieme di registri che conservano valori temporanei e di un insieme di istruzioni che le permettono di manipolare questi valori e di eseguire varie operazioni computazionali. Le macchine RAM sono comunemente utilizzate nell' informatica teorica per studiare la complessità computazionale degli algoritmi e per analizzare le prestazioni dei programmi informatici.



Osservazione

I registri R_n contenuti nella macchina RAM sono infiniti e possono contenere interi di qualsiasi dimensione. Ognuno è identificato da un indirizzo.
Il registro R_0 è l'unico sul quale si possono svolgere operazioni aritmetiche ed è chiamato **accumulatore**.

3.3 Il programma

- Il programma è una sequenza *finita* di istruzioni
- Ogni istruzione ha un'etichetta, l'indirizzo memorizzato dal program counter
- Ogni istruzione è una coppia opcode-indirizzo: L'opcode specifica l'operazione da eseguire, come ad esempio un'operazione aritmetica o una lettura/scrittura di una posizione di memoria. L'indirizzo specifica l'area di memoria su cui deve essere eseguita l'operazione.

  Ad esempio, supponiamo che un'istruzione nella macchina RAM abbia l'opcode 1 e l'indirizzo 100. Ciò significa che l'istruzione richiede di eseguire un'operazione specifica (definita dal codice operativo 1) sulla posizione di memoria 100.
- Ogni indirizzo può essere un operando o un'etichetta: Quando un indirizzo viene utilizzato come operando, viene interpretato come il **valore** memorizzato nella posizione di memoria corrispondente. Ad esempio, se l'istruzione richiede il valore memorizzato all'indirizzo 100, la macchina RAM accede alla posizione di memoria 100 e utilizza il valore memorizzato in quella posizione come operando.

  D'altra parte, quando un indirizzo viene utilizzato come etichetta, viene interpretato come un **riferimento** ad un'istruzione. Ad esempio, se l'istruzione richiede di eseguire un salto all'indirizzo 100, la macchina RAM interpreta l'indirizzo 100 come l'etichetta dell'istruzione a cui saltare.

  In pratica, questo significa che ogni posizione di memoria può essere utilizzata in modo flessibile sia come valore che come riferimento ad un'istruzione, consentendo alla macchina RAM di eseguire una vasta gamma di operazioni complesse.

3.4 Operandi e etichette

Le **etichette** sono associate solo ai comandi di salto, indicano a quale istruzione viene passato il controllo.
Un operando può assumere tre forme diverse:

1. $= i$ indica l'intero $i \in \mathbb{Z}$
2. i indica il contenuto di R_i, $i \in \mathbb{Z}$

3. $*i$ indica il contenuto di R_j dove j è il contenuto di R_i, $i, j \in \mathbb{Z}$

Lo **stato della macchina** è una funzione che associa ad ogni registro il suo contenuto e alle testine la loro posizione sul nastro.

- $S(r)$ Posizione della testina sul nastro di lettura
- $S(w)$ Posizione della testina sul nastro di scrittura
- $S(lc)$ Contenuto del program counter
- $S(k)$ Contenuto del registro R_k

Lo stato iniziale è definito nel modo seguente:

$$S_0(r) = S_0(w) = S_0(lc) = 1$$

$$S_0(k) = 0 \text{ per tutti i k} \in \text{N}$$

Un operando op allo stato S assume un valore che dipende dalla sua forma. Indichiamo tale valore con $V_s(op)$:

$$V_s(op) = \begin{cases} i & \text{se op è =1} \\ S(i) & \text{op è i} \\ S(S(i)) & \text{op è *i} \\ \perp & \text{otherwise} \end{cases} \tag{1}$$

Per tutte le istruzioni tranne che per i salti, si assume implicitamente che $S(lc) := S(lc) + 1$.

3.5 Semantica dei comandi

3.5.1 Istruzioni di spostamento tra registri

Istr.	Semantica
LOAD a	$S(0) := V_S(a)$
STORE i	$S(i) := S(0)$
STORE $*i$	$S(S(i)) := S(0)$

"LOAD a" carica in $R0$ il valore $VS(a)$. (a potrà essere i, =i, *i)
"STORE i" carica nel registro i il valore dell'accumulatore

3.5.2 Istruzioni aritmetiche

Istr.		Semantica
ADD	a	$S(0) := S(0) + V_S(a)$
SUB	a	$S(0) := S(0) - V_S(a)$
MULT	a	$S(0) := S(0) \times V_S(a)$
DIV	a	$S(0) := S(0) \div V_S(a)$

3.5.3 Istruzioni di lettura e scrittura sui nastri

Istr.		Semantica
READ	i	$S(i) := x_{S(r)}$ e $S(r) := S(r)+1$
READ	$*i$	$S(S(i)) := x_{S(r)}$ e $S(r) := S(r) + 1$
WRITE	a	Stampa $V_S(a)$ nella cella $S(w)$ del nastro di scrittura e $S(w) := S(w) + 1$

3.5.4 Istruzioni di salto

Istr.		Semantica	
JUMP	b	$S(lc) := b$	
JGTZ	b	Se $S(0) > 0$	allora $S(lc) := b$ altrimenti $S(lc) := S(lc) + 1$
JZERO	b	Se $S(0) = 0$	allora $S(lc) := b$ altrimenti $S(lc) := S(lc) + 1$
JBLANK	b	Se $x_{S(r)} = \flat$	allora $S(lc) := b$ altrimenti $S(lc) := S(lc) + 1$

3.6 La semantica del linguaggio

La semantica del linguaggio RAM si ottiene associando ad ogni programma P la funzione parziale

$$F_p : \bigcup_{n=0}^{\infty} Z^n \rightarrow \bigcup_{n=0}^{\infty} Z^n \cup \perp$$

se la computazione si arresta, FP(x) è il vettore di interi risultante sul nastro di uscita, quindi le n-uple z^n.
Altrimenti $F_p(x) = \perp$

3.7 La complessità computazionale

Definition 3.2 (Complessità computazionale)**.** La complessità computazionale di un algoritmo e la quantità di risorse da esso utilizzata durante la computazione.
Tali risorse vengono definite come costi in termini di *tempo* e *spazio*.

3.7.1 Criterio di costo uniforme

Il criterio di costo uniforme assume che ogni operazione abbia costo unitario in tempo e spazio. Dunque che ogni operando abbia la stessa dimensione e che lo spazio utilizzato da ogni registro sia unitario.

```
     READ    1
 2   JBLANK  10
     LOAD    1
     READ    2
     SUB     2
     JGTZ    2
     LOAD    2
     STORE   1
     JUMP    2
10   WRITE   1
     HALT
```

In questo programma d'esempio (ricerca massimo tra n numeri) notiamo che $S_p(x) = 3$ (numero di registri utilizzati). Mentre per la complessità temporale abbiamo:

$$T_p(x) = 5 \cdot (n-1) + 4 \text{ se x e decrescente (caso migliore)}$$

La prima parte $5(n-1)$ si riferisce al loop interno, quindi il caricamento del numero successivo che viene fermato alla **quinta** istruzione se la sottrazione è uguale a zero. mentre il $+4$ si riferisce alle istruzioni che vengono eseguite 1 volta (fuori dal loop)

$$T_p(x) = 8 \cdot (n-1) + 4 \text{ se x e crescente (caso peggiore)}$$

Stessa situazione di prima, ma il loop arriva fino alla fine.

3.7.2 Criterio di costo logaritmico

Un criterio più **preciso** e realistico è il Criterio di Costo Logaritmico (CCL). Dove il costo dell'istruzione dipende dalla dimensione dell'operando e la dimensione di un intero è il numero di bit necessari alla sua memorizzazione.

$$l(k) = \log_2(\mid k \mid) + 1$$

Il costo degli operandi in CCL è il seguente:

Operando a	Costo $t_S(a)$
=i	$l(i)$
i	$l(i)+l(S(i))$
*i	$l(i) + l(S(i))+l(S(S(i)))$

3.8 Il tempo di calcolo logaritmico

Definition 3.3 (Il tempo di calcolo logaritmico)**.** Il tempo di calcolo logaritmico è la somma dei costi logaritmici delle istruzioni eseguite nella computazione di P su input X.

- $T_p(x) \leq T_p^l(x)$

3.9 Confronto tra CCU e CCL

Consideriamo il seguente programma, esso svolge il calcolo di $z = 3^{2^n}$ su input $n \in N$

```
            READ          1        read n
            LOAD         =3
            STORE         2        y ⟵ 3
            LOAD          1        x ⟵ n
while       JZERO    endwhile      while x ≠ 0 do
            LOAD          2
            MULT          2
            STORE         2          y ⟵ y × y
            LOAD          1
            SUB          =1
            STORE         1          x ⟵ x − 1
            JUMP      while         end while
endwhile    WRITE         2        write y
            HALT
```

- **CCU**
  7 istruzioni sempre eseguite, 8 in loop. Dunque

  $$T(p) = 8n + 7 = \theta(n)$$

  Ossia ha una **crescita lineare**

- **CCL** Dopo k iterazioni R_2 contiene 3^{2^k}

  - LOAD 2
  - MULT 2
  - STORE 2

  Hanno costo logaritmico $l(3^{2^k}) = 2^k \cdot \log_2 3$
  Dunque il tempo logaritmico complessivo sarà:

  $$T_p^l(n) = \theta(\sum_{k=0}^{n} 2^k) = \theta(2^{n+1} - 1) = \theta(2^n)$$

  Lo spazio logaritmico complessivo sarà:

  $$S_p^l(x) = \max_{i \geq 0}\{\sum_{j \geq 0} l(s_i(j))\}$$

  Con:

  - $S_i(j)$ il contenuto del registro j nello stato i.

– $l(S_i(j))$ numero di bit (lunghezza) del contenuto del registro

– $\sum_{j\geq 0} l(s_i(j))$ La sommatoria della grandezza di ogni registro. Scegliamo il massimo di questa serie di valori.

Confrontiamo il seguente programma che permette di trovare il numero maggiore tra n inseriti nel nastro di lettura.

```
    READ    1     M ⟵ read(x1), i ⟵ 2
2   JBLANK  10    while xi ≠ ♭ do
    LOAD    1
    READ    2       x ⟵ read(xi), i ⟵ i + 1
    SUB     2
    JGTZ    2       if x > M then
    LOAD    2
    STORE   1         M ⟵ x
    JUMP    2     end while
10  WRITE   1     write(M)
    HALT
```

- **CCU**
  $S_p(x) = O(1)$
- **CCL**
  Supponiamo che gli interi in input siano tutti tra 1 e k. $T_p^l(x) = O(n \cdot \log k)$ $S_p^l(x) = O(\log k)$ (logaritmo dell'intero più grande che possiamo trovare sul nastro.) Utilizziamo O grande e non θ perchè il valore è **massimo** k. (caso peggiore)

3.10 La macchina RASP

Il modello RASP (*Random Access Stored Program*) è più vicino alle funzionalità di un processore rispetto al modello RAM.
Il Modello RASP mantiene il programma in memoria e permette di **modificare le istruzioni durante l'esecuzione**. Ha un set di istruzioni identico, con la differenza che manca l'indirizzamento diretto (*i).
Ad ogni istruzione sono associati due registri:

- Il primo codifica l'istruzione e il tipo di operando
- Il secondo contiene l'indirizzo

Istr.	Cod.	Istr.	Cod.	Istr.	Cod.
LOAD ()	1	SUB =()	7	WRITE ()	13
LOAD =()	2	MULT ()	8	WRITE =()	14
STORE ()	3	MULT =()	9	JUMP ()	15
ADD ()	4	DIV ()	10	JGTZ ()	16
ADD =()	5	DIV =()	11	JZERO ()	17
SUB ()	6	READ ()	12	JBLANK ()	18
				HALT	19

3	17
4	8
5	8
17	10
19	0

```
STORE  17
ADD     8
ADD    =8
JZERO  10
HALT
```

Esistono alcune variazioni rispetto al modello RAM:

- il registro lc viene automaticamente *incrementato di 2*;
- la memoria contiene inizialmente il programma in una sequenza prefissata di registri;

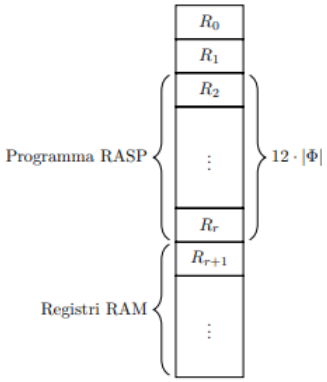
3.11 Equivalenza tra RAM e RASP

Teorema: $\forall \Phi$ programma RAM, $\exists \Psi$ programma RASP tale che $\forall n \in \mathbb{N}$, $\forall x \in Z^n$

- $F_\Phi(x) = F_\psi(x)$
- $T_\psi(x) \leq 6 \cdot T_\Phi(x)$
- $T^l_\psi(x) \leq C \cdot T^l_\Phi(x)$

Dimostriamo quindi che ogni istruzione RAM (code,op) può essere tradotta in una sequenza finita di al più 6 istruzioni RASP.

- operazione $\neq *i$ traduzione diretta
- Operazione $= i$, (Il modello RASP non possiede l'indirizzamento diretto, quindi bisogna simularlo nel seguente modo:
  - il programma RASP viene memorizzato nei registri da R_2 a R_r, con $r = 12 \cdot \mid \Phi \mid +1$. utilizziamo 12 perchè abbiamo un programma RAM da 6 istruzioni e in RASP ogni istruzione è divisa in indirizzo e istruzione.
  - supponiamo di simulare MULT *k nei registri da R_M a $R_M + 11$.



Ind.	Conten.	Signific.		Commento
M M+1	3 1	STORE	1	$S(1) := S(0)$
M+2 M+3	1 r+k	LOAD	r+k	$S(0) := S(r+k)$
M+4 M+5	5 r	ADD	=r	$S(0) := S(0) + r$
M+6 M+7	3 M+11	STORE	M+11	$S(M+11) := S(0)$
M+8 M+9	1 1	LOAD	1	$S(0) := S(1)$
M+10 M+11	8 -	MULT	$S(r+k)+r$	$S(0) :=$ $S(0) \times S(S(r+k)+r)$

3.12 La calcolabilità

3.12.1 Funzioni calcolabili e Tesi di Church-Turing

Dai teoremi precedenti abbiamo che:

- $\mathcal{F}_{RAM} = \{F \mid$ P programma RAM $\}$
- $\mathcal{F}_{RASP} = \{F \mid$ P programma RASP $\}$

$$\mathcal{F}_{RAM} \equiv \mathcal{F}_{RASP}$$

Definition 3.4. $\mathcal{F}$ è la classe delle **funzioni ricorsive parziali** ossia tutte quelle funzioni $f : \mathbb{N} \to \mathbb{N}$ che sono *calcolabili* in senso intuitivo e formale.

Definition 3.5 (Tesi di Church Turing)**.** La **tesi di Church Turing** é una tesi sulla natura delle funzioni calcolabili. Essa stabilisce che una funziona sui numeri naturali può essere calcolata da un metodo effettivo se e solo se é calcolabile da una *macchina di turing*.

3.13 Calcolabilità effettiva

Possiamo affermare che:

- $\mathcal{F}_{RAM}(f) = \{F \in \mathcal{F}_{RAM} \mid T_p^l(n) = O(f(n))\}$
- $\mathcal{F}_{RASP}(f) = \{F \in \mathcal{F}_{RASP} \mid T_p^l(n) = O(f(n))\}\}$

Dove $\mathcal{F}_{RAM}(f)$ e $\mathcal{F}_{RASP}(f)$ sono gli insiemi di funzioni calcolabili dei programmi RAM e RASP con complessità $O(f(n))$ nel caso peggiore.
Si ha che

$$\mathcal{F}_{RAM} \equiv \mathcal{F}_{RASP}$$

ma, se si sceglie una funzione f precisa, questa proprietà non vale in generale con altri linguaggi (perché il costo della simulazione non è sempre un fattore costante.

3.13.1 Problemi polinomiali

Consideriamo la classe:

$$\mathcal{P}_{RAM} = \{F_P \in \mathcal{F}_{RAM} \mid \exists k \text{ tale che } T_p^l(n) = O(n^k)\}$$

Definition 3.6. $\mathcal{P}$ è la classe dei **problemi risolubili in tempo polinomiale**.

3.14 Un linguaggio ad alto livello: AG

Il linguaggio AG, rispetto al linguaggio RAM e RASP consente una comprensione migliore delle istruzione e una maggiore intuitività, permettendo di analizzare la complessità senza esplicita traduzione in linguaggio RAM.

3.14.1 Variabili in linguaggio AG

Definition 3.7 (Variabile)**.** Una variabile è un identificatore X di una particolare zona di memoria.

Ad ogni variabile è associato un insieme $\mathcal{U}$ di possibili valori (il tipo della variabile).

3.14.2 Espressioni in linguaggio AG

Definition 3.8 (Espressione)**.** Un'espressione è un termine che denota l'applicazione di simboli di operazioni a variabili o a valori costanti.

Come le variabili, anche le espressioni hanno un valore corrente durante l'esecuzione, risultato dell'applicazione delle operazioni sui valori correnti delle variabili.

3.15 Condizioni in linguaggio AG

Definition 3.9 (Condizione)**.** Una condizione è un simbolo di predicato applicato a una o più espressioni. Il valore di essa può essere *vero* o *falso*.

3.16 Sintassi del linguaggio AG

Comando	Sintassi
Assegnamento	$V := E$
Selezione	`if` P `then` C_1 `else` C_2
Iterazione	
Ciclo `for`	`for` $k = 1$ `to` n `do` C
Ciclo `while`	`while` P `do` C
Sequenza	`begin` $C_1; C_2; \ldots C_n;$ `end`
Comando etichettato	$e : C$
Salto	`goto` e

Dove V è una variabile, E è un'espressione, P è una condizione, e è un'etichetta e $C, C_1, \ldots, C_n$ sono comandi.

3.16.1 Assegnamento

$$V := E$$

Assegna alla variabile V il valore corrente dell'espressione E.

Complessità: somma di

- Costo della valutazione di E
- Costo di modifica dello stato di V (dipende dal numero di bit necessari a rappresentare i valori del tipo di V).

3.16.2 Selezione

$$\text{if } P \text{ then } C_1 \text{ else } C_2$$

Esegue C_1 se la condizione P è vera, altrimenti esegue C_2.

Complessità: somma di

- Costo della valutazione di P
- Costo di C_1 se P è vera, altrimenti costo di C_2.

3.16.3 Iterazione

Ciclo **for**:

$$\text{for } k = 1 \text{ to } n \text{ do } C$$

Esegue in successione C, in cui la variabile k assume valore $1, 2, \ldots, n$

Complessità:

$$\sum_{k=1}^{n} \text{Costo di C}$$

Ciclo **while**:

$$\text{while } P \text{ do } C$$

Se P è vera, esegue C, e ripete l'operazione finchè P diventa falsa.

Complessità:

$$\sum_{k=1}^{n} \text{Costo di C + costo valutazione P}$$

3.17 Sequenza

$$\text{begin } C_1; C_2; \ldots C_n \text{ end}$$

Esegue in successione $C_1, C_2, \ldots, C_n$

Complessità:

$$\sum_{k=1}^{n} \text{Costo di } C_i$$

3.18 Comando etichettato e salto

Comando etichettato

$$e : C$$

Esegue C. Può essere la destinazione di un comando di salto.

Complessità: Costo di C **Comando di salto**

goto e

Rimanda il flusso di esecuzione al comando con etichetta *ss*.

Complessità: costante (non dipende da nessun tipo di imput es: O(1))

3.18.1 Sottoprogrammi

I sottoprogrammi in linguaggio AG sono di due forme:

- Funzione: Calcola una **funzione** esplicitamente utilizzata dal programma principale;
- Procedura: **Modifica lo stato** del programma principale;

La differenza sostanziale tra le due è che una procedura non "ritorna" nulla.
Funzioni

Procedura Nome(λ) C;

- Nome: identificatore del **sottoprogramma**
- λ: lista dei **parametri formali**
- C: Comando contenente **istruzioni di ritorno** ("return E")

Invocazione:

$$A := \text{ Nome } (B)$$

- B: Lista dei **parametri attuali**. In λ vengono copiati i valori o gli indirizzi (passaggio per valore o riferimento) di B.

- Il flusso di controllo viene passato al sottoprogramma (Nome)
- Quando si esegue un comando “return E” il valore di E viene copiato in A

Procedure

Procedura Nome (λ) C;

- Nome e λ: stesso significato delle funzioni.
- C: Comando qualsiasi (può non contenere istruzioni di ritorno).

Invocazione:

Nome (B)

- Il passaggio di parametri e del controllo avviene nello stesso modo delle funzioni.
- I La procedura effettua delle modifiche dello stato delle variabili del programma, ma **non restituisce nessun valore**.

3.19 Passaggio di parametri e costi nei sottoprogrammi

Passaggio per valore: l’eventuale modifica del parametro formale non ha effetto sul parametro attuale (viene utilizzata una sua ”copia” per i calcoli).

Complessità: costo del sottoprogramma + lunghezza dei parametri passati per valore.

Passaggio per riferimento: Ogni modifica del parametro formale si riflette sul parametro attuale. *Complessità*: costo del sottoprogramma.

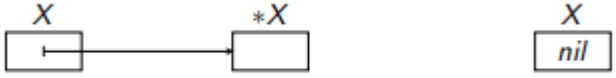
Inoltre, se si tratta di una funzione, bisogna sommare anche la lunghezza dell’espressione restituita

3.20 Puntatori

Un puntatore è una variabile che assume come valore corrente l’**indirizzo** di un’altra variabile.

- La variabile puntata da X viene denotata da *X.

è una visione più ad alto livello dell’indirizzamento indiretto.



Comando	*Significato*
$*X := *Y$	var. puntata da $X \longleftarrow$ var. puntata da Y
$*X := Z$	var. puntata da $X \longleftarrow Z$
$Z := *X$	$Z \longleftarrow$ var. puntata da X
$X := Y$	$X \longleftarrow Y$

3.21 Esempio di complessità

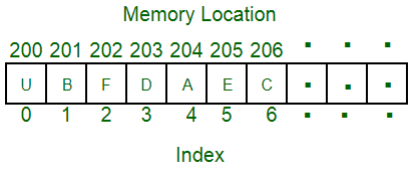
Se *X e *Y contengono matrici $n \times n$ allora:

- *X := *Y ha costo $\theta(n^2)$ perchè assegna a *X una copia della matrice
- X := Y ha costo $\theta(1)$ perché copia in X l'indirizzo a cui punta Y, e gli indirizzi hanno dimensione costante, indipendente dall'oggetto a cui puntano.

4 Strutture dati elementari

4.1 Array

Fissato un tipo base $\mathcal{U}$ e $n \in \mathbb{N}$, un vettore di lunghezza n è un elemento di $\mathcal{U}^n = \mathcal{U} \times \mathcal{U} \times \cdots \times \mathcal{U}$ (n volte)



Possiamo effettuare due operazioni:

- Proiezione (π): *Lettura di una posizione in un array.*

$$\mathcal{U}^n \times \mathbb{N} \Rightarrow \mathcal{U} \cup \{\perp\}$$

$$\pi(\text{arr}, i) = \begin{cases} arr[i] & A = (a_i, \ldots, a_n), 1 \leq i \leq n \\ \perp & \text{altrimenti} \end{cases}$$

- Sostituzione (σ): *Scrittura di un dato in un array.*

$$\mathcal{U}^n \times \mathbb{N} \times \mathcal{U} \Rightarrow \mathcal{U}^n \cup \{\perp\}$$

$$\sigma(\text{arr}, i, a) = \begin{cases} (a_i, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_n) & A = (a_i, \ldots, a_n), 1 \leq i \leq n \\ \perp & \text{altrimenti} \end{cases}$$

Se un oggetto di tipo $\mathcal{U}$ richiede k registri allora l'array viene memorizzato su $k \cdot n$ registri consecutivi, dove n è la grandezza dell'array. Ricapitolando, abbiamo:

- $\mathcal{U}$: Tipo base del vettore
- k: Numero di bit richiesti per memorizzare un dato di tipo $\mathcal{U}$
- n: Dimensione del vettore
- α: Indirizzo base del vettore. *Solitamente* $\alpha = 0$

Supponiamo che il primo elemento di V sia $V[0]$. L'indirizzo in memoria $V[i]$ è:

$$\alpha + k \cdot i$$

L'indirizzo viene calcolato in tempo costatnte $O(1)$.

4.2 Matrici

Fissato un tipo base $\mathcal{U}$ e $m, n \in \mathbb{N}$ una matrice di ordine $m \times n$ è un elemento di $\mathcal{U}^{|m \times n|}$.

	Col1	Col2	Col3	Col4	....
Row1	Arr[0][0]	Arr[0][1]	Arr[0][2]	Arr[0][3]	
Row2	Arr[1][0]	Arr[1][1]	Arr[1][2]	Arr[1][3]	
Row3	Arr[2][0]	Arr[2][1]	Arr[2][2]	Arr[2][3]	
Row4	Arr[3][0]	Arr[3][1]	Arr[3][2]	Arr[3][3]	
⋮					

Possiamo effettuare due operazioni:

- Proiezione (π): *Lettura di una cella della matrice.*

$$\mathcal{U}^{|m \times n|} \times \mathbb{N} \times \mathbb{N} \Rightarrow \mathcal{U} \cup \{\bot\}$$

$$\pi(\text{mat}, i, j) = \begin{cases} mat_{ij} & 1 \leq i \leq m, 1 \leq j \leq n \\ \bot & \text{altrimenti} \end{cases}$$

- Sostituzione (σ): *Scrittura di un dato in una cella della matrice.*

$$\mathcal{U}^n \times \mathbb{N} \times \mathcal{U} \Rightarrow \mathcal{U}^n \cup \{\bot\}$$

$$\sigma(\text{mat}, i, j, a) = \begin{cases} B & 1 \leq i \leq m, 1 \leq j \leq n \text{ dove } b_{pq} = a_{pq},\ p \neq i \vee q \neq j, b_{ij} = a \\ \bot & \text{altrimenti} \end{cases}$$

Se un oggetto di tipo $\mathcal{U}$ richiede k registri allora la matrice viene memorizzata su $k \cdot n \cdot m$ registri consecutivi, dove n e m è la grandezza delle righe e delle colonne.
Ricapitolando, abbiamo:

- $\mathcal{U}$: Tipo base della matrice
- k: Numero di bit occupati da un elemento di tipo $\mathcal{U}$
- $n \times m$: Ordine della matrice M ($M \in \mathcal{U}^{m \times n}$ m righe e n colonne).
- α: Indirizzo base della matrice.

Esempio: l'indirizzo in memoria $m[i][j]$ partendo da $m[0][0]$ sarà:

$$\alpha + i \cdot n \cdot k + j \cdot k \text{ (memorizzazione per righe)}$$

$$\alpha + i \cdot m \cdot k + j \cdot k \text{ (memorizzazione per colonne)}$$

Calcolato in tempo costante $O(1)$

4.3 Record

Un record è un contenitore di informazioni **non uniforme**.

- $\mathcal{U}_1, \ldots, \mathcal{U}_n$: **tipi** dei campi
- k_i: numero di byte occupati da un elemento $\mathcal{U}_i$
- n: numero dei campi del record R
- e_i: etichetta del campo i-esimo
- α: indirizzo base di R

L'indirizzo in memoria del campo R è:

$$\alpha + \sum_{j=0}^{i} k_j$$

che viene calcolato in fase di compilazione, quindi il tempo di accesso in esecuzione è $O(1)$. In compenso, proprio perché i calcoli sono effettuati in compilazione, non è possibile utilizzare una variabile come indice per accedere a un record.

4.4 Liste

Una lista è un vettore con dimensione variabile.// Fissato un tipo base $\mathcal{U}$, una lista di oggetti di tipo $\mathcal{U}$ è un elemento di $\mathcal{U}^* = \bigcup_{i \geq 0} \mathcal{U}^i$

- La lunghezza di $L = (a_1, a_2, \ldots, a_n)$ è $\mid L \mid = n$
- $\wedge$ indica la lista vuota

La lista è una struttura dati dinamica perché $\mid L \mid$ può variare in fase di esecuzione.

4.4.1 Operazioni basilari su liste

IsEmpty:
Controlla se la lista è vuota.
`IS-EMPTY` : $\mathcal{L} \to \{\text{Vero, Falso}\}$

$$\texttt{IS-EMPTY(L)} \ : \ = \begin{cases} Vero & L = \wedge \\ Falso & \text{altrimenti} \end{cases}$$

El: Accesso in lettura.
`El` : $\mathcal{L} \times \mathbb{N} \to \mathcal{U} \cup \{\perp\}$

$$\texttt{EL(L, k)} \ : \ = \begin{cases} a_k & L = (a_1, \ldots, a_n),\ 1 \leq k \leq n \\ \perp & \text{altrimenti} \end{cases}$$

Insert: Inserisci elemento nella lista.
`Ins` : $\mathcal{L} \times \mathbb{N} \times \mathcal{U} \to \mathcal{L} \cup \{\perp\}$

$$\texttt{Insert(L, k, u)} \ : \ = \begin{cases} (u) & L = \wedge \text{ e } k = 1 \\ (a_1, \ldots, a_{k-1}, u, a_k, \ldots, a_n) & L = (a_1, \ldots, a_n),\ 1 \leq k \leq n+1 \\ \perp & \text{altrimenti} \end{cases}$$

($n+1$ perchè possiamo aggiungere alle spalle dell'ultimo elemento)

Remove: Rimuovi un elemento dalla lista.
`Remove` : $\mathcal{L} \times \mathbb{N} \to \mathcal{L} \cup \{\perp\}$

$$\texttt{Remove(L, k)} \ : \ = \begin{cases} (a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n) & L = (a_1, \ldots, a_n),\ 1 \leq k \leq n \\ \perp & \text{altrimenti} \end{cases}$$

4.4.2 Operazioni derivate su liste

Sostituzione di un elemento

$$\text{Cambia}(L, k, u) = \text{Togli}(\text{Ins}(L, k, u), k + 1)$$

Inserisco un elemento in posizione k, e l'elemento translato (precedentemente in posizione k) verrà cancellato

Modifica e accesso in testa

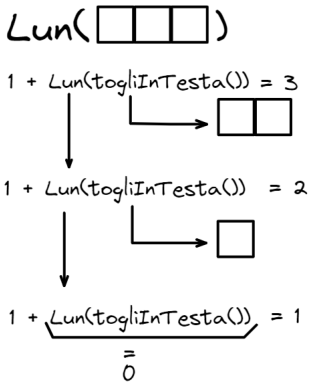
$$\text{Testa}(L) = \text{El}(L, 1)$$

$$\text{InsInTesta}(L, u) = \text{Ins}(L, 1, u)$$

$$\text{TogliInTesta}(L) = \text{Togli}(L, 1)$$

Lunghezza di una lista Utilizziamo una funzione **ricorsiva**.

$$\text{Lun(L)} = \begin{cases} 0 & \text{isEmpty(L)} \\ 1 + \text{Lun}(\text{TogliInTesta}(L)) & \text{Altrimenti} \end{cases}$$



Modifica e accesso in coda:

$$\text{Coda}(L) = \text{El}(L, \text{Lun}(L))$$

$$\text{InsInCoda}(L, u) = \text{Ins}(L, \text{Lun}(L) + 1, u)$$

$$\text{TogliInCoda}(L) = \text{Togli}(L, \text{Lun}(L))$$

4.4.3 Scorrimento di una lista

```
Procedura Occ(L,a)
begin
    n := 0;
    for b in L do
        if b = a then n := n + 1;
    return n;
end
```

$$\text{Occ(L,a)} = \begin{cases} 0 & \text{isEmpty(L)} \\ 1 + \text{Occ(TogliInTesta}(L), a) & L \neq \wedge \text{ e } a = Testa(L) \\ \text{Occ(TogliInTesta}(L), a) & L \neq \wedge \text{ e } a \neq Testa(L) \end{cases}$$

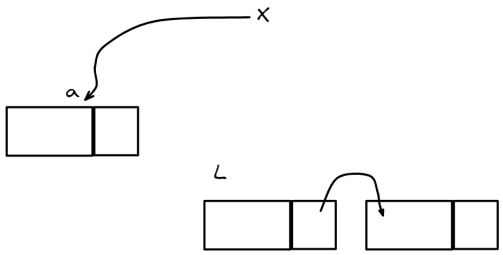
4.4.4 Implementazione di liste tramite puntatori

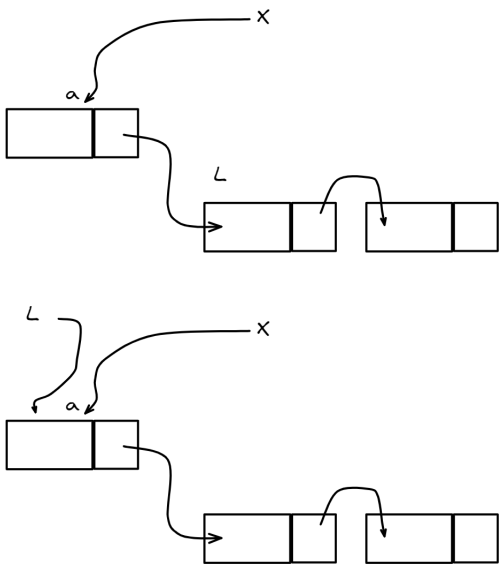
Controllo se la lista è vuota:

```
Procedura IsEmpty(L)
if L = nil then return Vero
           else return Falso;
```

Inserimento in testa:

```
Procedura Ins in Testa(L,a)
    begin
        X := Crea Nodo(a);
        (*X).succ := L;
        L := X;
    end
```





Ricerca di un elemento nella lista

```
Procedura Trova(L,a)
begin
    if L = nil then return Falso
        else
        begin
            R := *L;
            while R.elem != a && R.succ != nil do R := *(R.succ);
            if R.elem = a then return Vero
                        else return Falso;
        end
end
```

Continuo a scorrere la lista finchè l'elemento contenuto nel nodo R è uguale ad a oppure finche non arrivo all'ultimo elemento della lista.

Liste circolari: il problema di Giuseppe Flavio

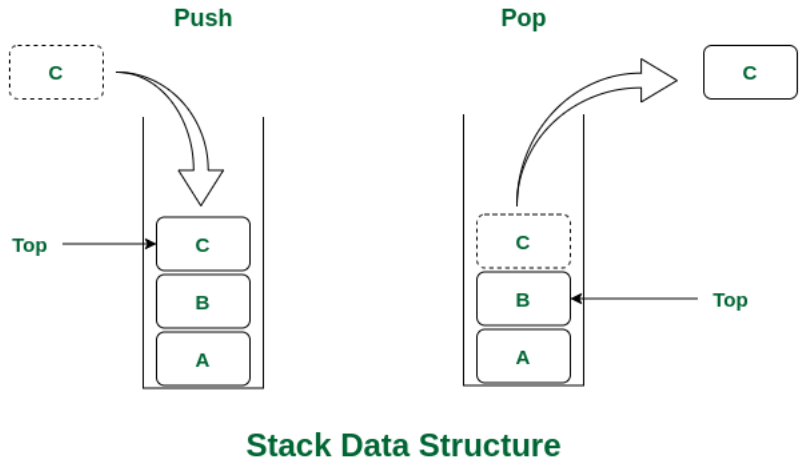
```
// Costruzione della catena circolare

    begin
    T := Crea Nodo(1);
    X := T;
    (*T).succ := T;
    for i = 2 to N do
        begin
            (*X).succ := Crea Nodo(i);
            X := (*X).succ;
```

```
            (*X).succ := T;
        end;
    end
```

```
// Eliminazione modulo M
    begin
        while X := (*X).succ do
            begin
                for i = 1 to M-1 do
                X := (*X).succ;
                (*X).succ := (*(*X).succ).succ;
            end;
        write((*X).elem);
    end
```

4.5 Pila

La pila è una struttura dati LIFO (Last In First Out)



Stack Data Structure

4.5.1 Operazioni disponibili per la struttura dati Pila

$$\texttt{IS-EMPTY(P)} = \begin{cases} Vero & P = \wedge \\ Falso & \text{altrimenti} \end{cases}$$

$$\texttt{Top(P)} = \begin{cases} a_n & S = (a_1, \ldots, a_n) \\ \perp & P = \wedge \end{cases}$$

$$\texttt{Pop(P)} = \begin{cases} (a_1, \ldots, a_{n-1}) & S = (a_1, \ldots, a_n) \\ \perp & P = \wedge \end{cases}$$

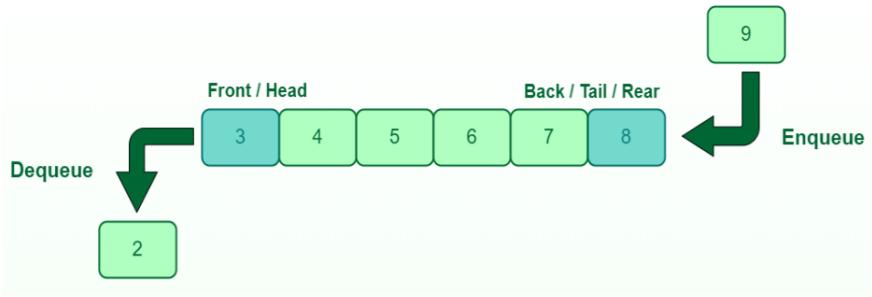
$$\texttt{Push(P)} = \begin{cases} (a_1, \ldots, a_n, a) & S = (a_1, \ldots, a_n) \\ (a) & P = \wedge \end{cases}$$

4.5.2 Implementazione della pila con liste concatenate

```
Procedura Is empty(S)
if S = nil then return True
       else return False

Procedura Top(S) return (*S).elem

Procedura Pop(S)
    if Is empty(S) then return -1
           else begin S := (*S).succ; return 0; end;

Procedura Push(S, a)
    X := Crea nodo(a);
    (*X).succ := S
    S := X;
```

4.6 Coda

La coda è una struttura dati FIFO (First In First Out)



Queue Data Structure

4.6.1 Operazioni disponibili per la struttura dati Coda

$$\texttt{IS-EMPTY(C)} = \begin{cases} Vero & C = \wedge \\ Falso & \text{altrimenti} \end{cases}$$

$$\texttt{Front(C)} = \begin{cases} a_1 & S = (a_1, \ldots, a_n) \\ \bot & P = \wedge \end{cases}$$

$$\texttt{Dequeue(C)} = \begin{cases} \wedge & C = (a_1) \\ (a_2, \ldots, a_n) & S = (a_1, \ldots, a_n) \\ \bot & P = \wedge \end{cases}$$

$$\texttt{Enqueue(C, a)} = \begin{cases} (a_1, \ldots, a_n, a) & S = (a_1, \ldots, a_n) \\ (a) & C = \wedge \end{cases}$$

4.6.2 Implementazione con vettore della coda

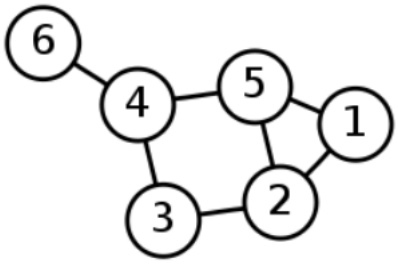
5 Grafi

Definition 5.1. Grafo non orientato Un grafo G è una coppia di insiemi V ed E

$$G = \langle V, E \rangle$$

- V inisiemi finito (Vertici o nodi)
- $E \subseteq V^{(2)}$ (lati) insieme degli insiemi di due vertici ($\{3,2\} = \{2,3\}$)

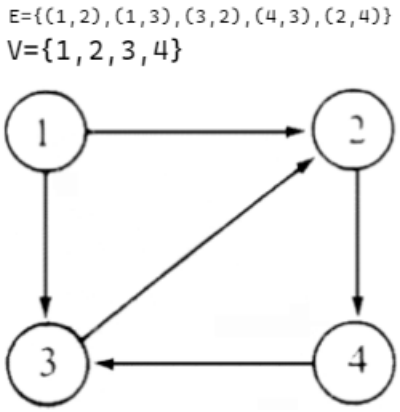
E={(6,4),(4,5),(4,3),(3,2),(5,2),(2,1),(5,1)}
V={1,2,3,4,5,6}



Definition 5.2. Grafo orientato Un grafo G è una coppia di insiemi V ed E

$$G = \langle V, E \rangle$$

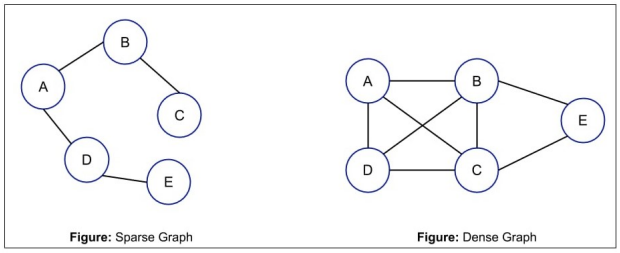
- V inisiemi finito (Vertici o nodi)
- $E \subseteq V^{(2)}$ (archi) insieme degli insiemi di due vertici ($\{3,2\} = \{2,3\}$)



Siano $G = \langle V, E \rangle, n =\mid V \mid \ m =\mid E \mid$

- $0 \leq m \leq n^2$ G orientato
- $0 \leq m \leq \frac{n(n-1)}{2}$ G non orientato
- G è **sparso** se $m = O(n)$
- G è **denso** se $m = \theta(n^2)$

Definition 5.3 (Grafi sparsi e densi)**.** Un grafo con "molti" spigoli rispetto al numero dei vertici è chiamato **denso**, mentre al contrario un grafo con "pochi" spigoli è detto **sparso**



Definition 5.4 (Sottografo)**.** $G' = \langle V', E' \rangle$ è **sottografo** di $G = \langle V, E \rangle$ se

- $V' \subseteq V$
- $E' \subseteq E \cup V' \times V'$

5.0.1 Alcune definizioni

Definition 5.5 (Cappio)**.** (x, x) è detto **cappio**

Definition 5.6 (Adiacenza)**.** w è adiacente a v se esiste un lato che da v porta a w. *formalmente:*

$$\text{se } (v, w) \in E(\{v, w\} \in E)$$

Definition 5.7 (Insieme di adiacenza)**.** L'insieme di adiacenza di un nodo v sono tutti i vertici che possiamo raggiungere uscendo da v con un lato solo.

$$Adiac(v) = \{w \mid (v, w) \in E\}$$

Definition 5.8 (Cammino)**.** Insieme di vertici collegati da lati

$$x_1, x_2, \ldots, x_k \mid (x_i, x_{i+1}) \in E, 1 \leq i < k$$

La *lunghezza* di un cammino è $k - 1$

Un cammino è detto *semplice* se non contiene *cicli*.

Definition 5.9 (Ciclo)**.** Un ciclo è un cammino dove il primo e l'ultimo nodo sono uguali.

$$x_1, x_2, \ldots, x_k \mid x_1 = x_k$$

è detto *semplice* se non presenta uguaglianze tra nodi oltre a quella tra il primo e l'ultimo.

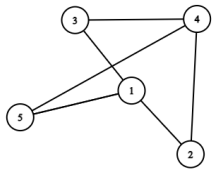
Definition 5.10 (Connessione)**.** v è connesso a w, $v \diamond w$, se esiste un cammino da v a w.

Se il grafo non è orientato $v \diamond w$ implica $w \diamond v$. (Relazione di equivalenza)

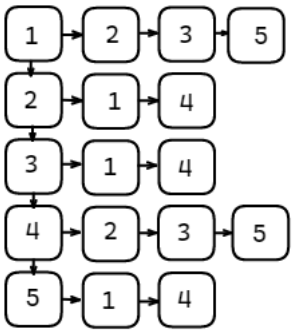
Definition 5.11 (Grafo connesso)**.** Un grafo $G = \langle V, E \rangle$ è connesso se $\forall v, w \in V$ esiste un cammino da v a w.

5.1 Rappresentazioni grafo

Un grafo può essere rappresentato tramite:



- Liste di adiacenza



  - Vantaggi: spazio occupato ($\theta(n+m)$)
  - Svantaggi: tempo per decidere se $(x,y) \in E$ ($O(n)$)

- Matrice di adiacenza

$$\begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix}$$

  - Vantaggi: tempo per decidere se $(x,y) \in E$ ($O(1)$)
  - Vantaggi: spazio occupato ($\theta(n^2)$)

6 Alberi

6.1 Introduzione e definizioni

Definition 6.1 (Albero)**.** Gli alberi sono **grafi** non orientati, connessi e aciclici che ammettono uno e un solo cammino tra una qualunque coppia di nodi.

Un albero con radice è un albero dove si evidenzia un nodo.

Definition 6.2 (Livello o profondità)**.** Il livello di un nodo è la lunghezza del cammino semplice che collega la radice al nodo.

Definition 6.3 (Analogie "familiari")**.** x è **padre** di y se x precede y nel cammino semplice che congiunge la radice a y.

y è **figlio** di x se x è padre di y

x è **fratello** di y se x e y hanno lo stesso padre

x è **antenato** di y se x compare nel cammino semplice che congiunge la radice a y

x è **discendente** di y se y è antenato di x

I nodi in un albero si dividono in due categorie:

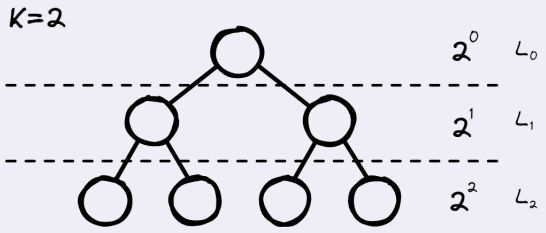
- *Nodi interni*: tutti i nodi che hanno figli
- *Foglie*: tutti i nodi che non hanno figli

Definition 6.4 (Grado di un nodo)**.** Il **grado** di un nodo è il numero di figli posseduti.

Definition 6.5 (Albero k-ario)**.** Un albero **k-ario** è un albero in cui i nodi hanno grado al più k ed esiste almeno un nodo di grado k.

Osservazione

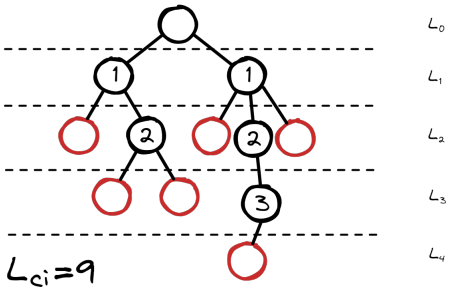
In un albero k-ario il numero di nodi il cui livello è i è minore o uguale a k^i



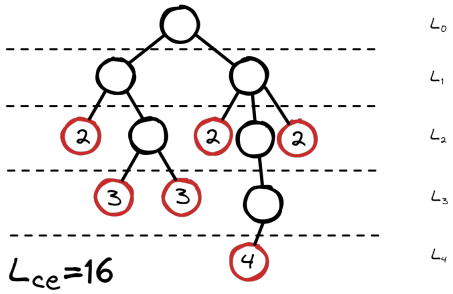
Un livello si dice **saturo** se presenta il massimo numero possibile di nodi.

Definition 6.6 (Lunghezza dei cammini)**.**

- La *lunghezza del cammino interno* di un albero è la somma dei livelli di tutti i nodi interni.



- La *lunghezza del cammino esterno* di un albero è la somma dei livelli di tutte le foglie.



  La *lunghezza del cammino di un albero* è data dalla somma del cammino interno più il cammino esterno.

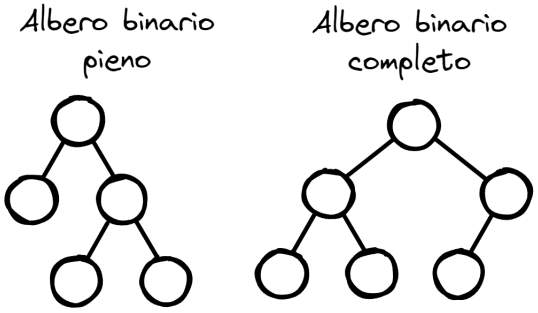
6.2 Alberi binari

Definition 6.7 (Albero binario)**.** Un **albero binario** è un albero con radice ordinato in cui

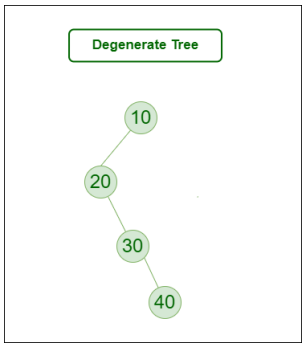
- I nodi possono avere massimo 2 figli
- I figli di un nodo sono distinti come figlio sinistro e figlio destro

Definition 6.8 (Albero binario pieno e completo)**.** Un albero binario si dice **pieno** se tutti i livelli sono saturi tranne eventualmente l'ultimo.

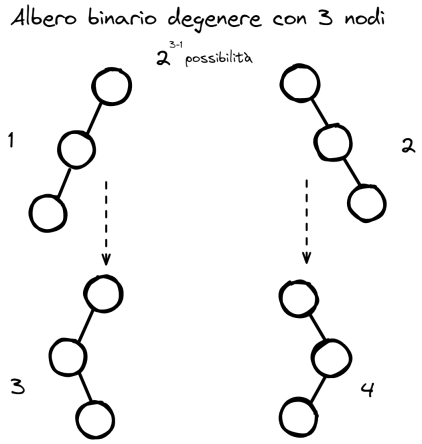
Un albero binario si dice **completo** se è pieno e i nodi sull'ultimo livello sono tutti disposti il più a sinistra possibile



Definition 6.9 (Albero binario degenere)**.** Un albero binario si dice **degenere** se ha n nodi e altezza $n-1$
Ogni nodo padre ha quindi un solo figlio associato



esistono (considerando solo la "forma") 2^{n-1} alberi binari degeneri, dato che essi hanno una corrispondenza biunivoca con le stringhe binarie di lunghezza n-1: per ogni nodo, eccetto la radice, bisogna scegliere se posizionarlo come figlio sinistro o destro.



6.3 Proprietà matematiche degli alberi

Fatto. Un albero binario completo di altezza h ha un numero di nodi n che soddisfa la relazione

$$2^h \leq n \leq 2^{h+1} - 1$$

Di conseguenza l'altezza di un albero binario con n nodi è approssimativamente

$$log_2 n$$