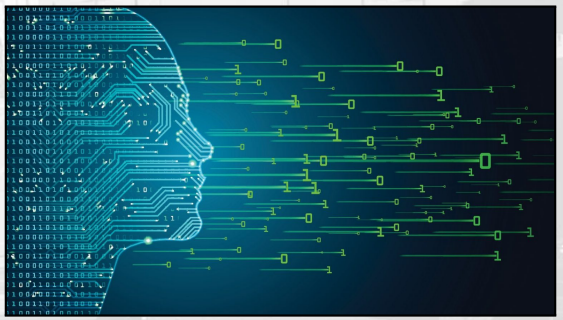
Programmazione Informatica
5. Algoritmi e strutture dati

Riccardo Zamolo
rzamolo@units.it

Università degli Studi Trieste
Ingegneria Civile e Ambientale



A.A. 2020-21

- Abbiamo visto che nella rappresentazione posizionale dei numeri avevamo ottenuto, in maniera molto naturale, due "schemi iterativi" per ottenere le cifre di un dato numero in una determinata base (il primo "schema" era relativo alle cifre della parte intera, il secondo "schema" era relativo alla parte non intera).
- Questi "schemi" sono definibili in maniera più corretta come dei *procedimenti di calcolo*.
- *Algoritmo* è un sinonimo di procedimento di calcolo, ma è meglio definito come un insieme di *istruzioni* che definiscono una sequenza di *operazioni* mediante le quali si risolvono tutti i problemi di una *classe*.
- Algoritmo per determinare le cifre di $n \in \mathbb{N}$ in base $b > 1$:

```
I1. Eseguire la divisione intera di n per b (n=q*b+r) memorizzando q ed r;
I2. Tornare ad I1 usando q al posto di n, finche' q=0.
```

  - è caratterizzato da 2 istruzioni `I1` e `I2`;
  - `I1` e `I2` definiscono una sequenza di operazioni, cioè la ripetizione della divisione intera effettuata un numero di volte dipendente da n e da b;
  - si è assunto che, indipendentemente da n e da b, il procedimento porta prima o poi a $q = 0$, condizione di interruzione del procedimento, per il quale il problema è risolto;
  - è applicabile indipendentemente da n e da b: la classe di problemi risolvibili è identificata da tutte le coppie di interi positivi n e $b > 1$.

Un algoritmo dev'essere:

- *Finito:* il numero di istruzioni che definiscono l'algoritmo deve essere finito, come il numero di operazioni da esse definite. L'algoritmo deve fornire dei risultati in tempo finito.
- *Deterministico:* le operazioni definite dalle istruzioni devono essere definite in maniera non ambigua. Esecuzioni diverse dell'algoritmo sullo stesso problema devono fornire sempre lo stesso risultato.
- *Realizzabile praticamente:* le operazioni definite dalle istruzioni devono essere eseguibili, cioè conosciute dalla macchina (o dalla persona) che eseguirà l'algoritmo.

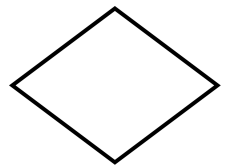
Vediamo queste proprietà nel caso dell'algoritmo precedente:

- È finito? Basta provare che la condizione di arresto $q = 0$ viene sempre raggiunta: ciò è vero in quanto ad ogni divisione intera si ottiene un quoziente q strettamente minore del dividendo n, poichè il divisore è $b > 1$, quindi prima o poi si otterrà $q = 0$.
- È deterministico? Si, poichè la divisione intera è definita in maniera univoca ed è un'operazione deterministica.
- È realizzabile praticamente? Si, se l'esecutore sa come fare una divisione intera.

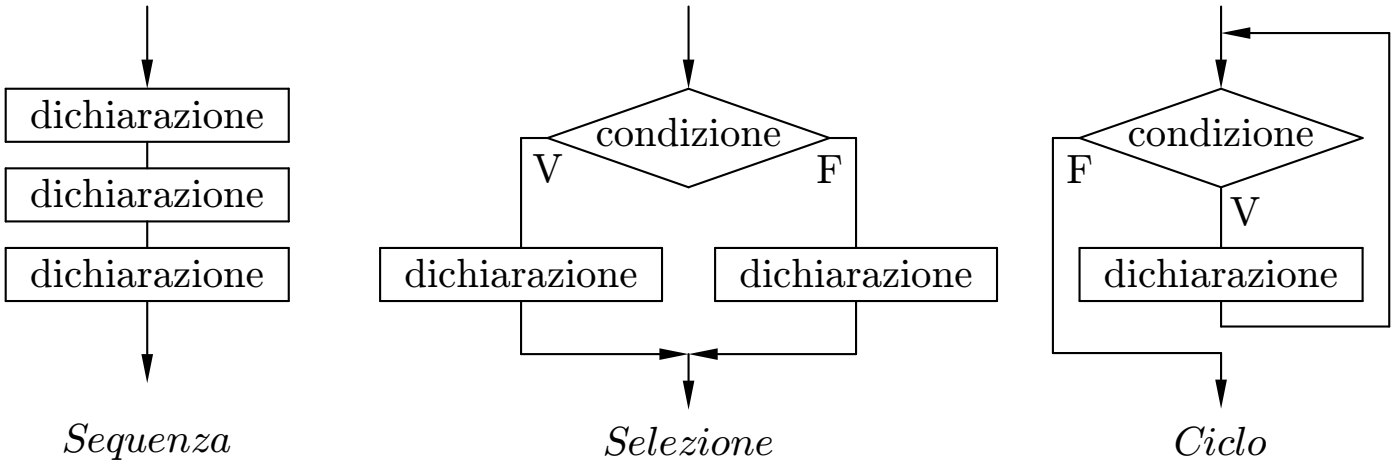
- Finora abbiamo espresso formalmente un algoritmo mediante una sequenza di istruzioni, interpretabili dall'uomo, ma senza alcun formalismo nè metodo.
- Un algoritmo può essere espresso graficamente mediante un *diagramma di flusso*, ossia una struttura costituita da blocchi connessi da frecce. Il flusso di lavoro è costituito dalle azioni contenute nei blocchi che vengono eseguite secondo l'ordine definito dalle frecce.
- Vi sono tre tipologie di blocchi:
  - *blocco terminale:* definisce l'inizio o la fine dell'algoritmo. Nel caso di blocco iniziale, esso è unico e possiede una sola freccia in uscita. I blocchi finali possono invece essere multipli e ad ognuno di esso possono arrivare più frecce;

  - *blocco funzionale:* contiene un'istruzione elementare di tipo aritmetico, di assegnazione, di lettura o di scrittura. Sono possibili più frecce entranti ma una sola freccia uscente;

  - *blocco decisionale:* contiene una condizione logica in base alla quale si decide quale delle due frecce uscenti seguire per la prosecuzione dell'esecuzione. Anche in questo caso sono possibili più frecce entranti.



- *Teorema di Jacopini-Böhm (1966):* qualunque algoritmo può essere scritto combinando tre sole *strutture di controllo*:
  - *sequenza:* lista di istruzioni/sottoprogrammi;
  - *selezione (o condizionale):* scelta tra due sottoprogrammi in base al valore di un'espressione booleana;
  - *ciclo (o iterazione):* ripetizione di un sottoprogramma finchè un'espressione booleana si mantiene vera.



Sequenza *Selezione* *Ciclo*

- *Programmazione strutturata:* modalità di programmazione che prevede l'uso delle tre precedenti strutture di controllo, opportunamente combinate.
- Si evita perciò l'uso del dannoso comando `goto` (salto incondizionato ad un altro punto del programma) che, se usato non intelligentemente, rende un programma difficile da analizzare, verificare e mantenere. Esempio: nel caso dell'algoritmo introdotto ad inizio capitolo, saremmo tentati di utilizzare un `goto` per andare da `I2` ad `I1`.

- La rappresentazione degli algoritmi mediante diagrammi di flusso ha il vantaggio dell'immediatezza visiva, ma è scomodo da utilizzare in termini pratici.
- Indipendentemente dal linguaggio di programmazione che si andrà ad utilizzare, un algoritmo può equivalentemente essere espresso mediante *pseudocodice*, ossia un programma scritto utilizzando un ipotetico linguaggio ideale il più naturale possibile per l'uomo.
- L'uso di pseudocodice è finalizzato alla massima comprensione umana dell'algoritmo rappresentato e non alla sua esecuzione da parte del computer. Tutti i dettagli tecnici non strettamente necessari alla comprensione logica dell'algoritmo sono quindi omessi in favore della massima compattezza e leggibilità:

Assegnazione

```
a ← espressione
```

Selezione (if)

```
if condizione
   istruzioni
else
   istruzioni
end
```

Ciclo (while)

```
while condizione
   istruzioni
end
```

- Sintassi alternative/addizionali di frequente uso:

Ciclo (repeat)

repeat
  istruzioni
until condizione

Ciclo (for)

for $i \in I$
  istruzioni
end

Selezione (switch)

switch espressione
  case espressione1
    istruzioni1
  case espressione2
    istruzioni2
  ...
  otherwise
    istruzioni
end

- I cicli `while`, `repeat-until` e `for` sono del tutto equivalenti: ognuno si può ottenere utilizzandone un altro in maniera opportuna. È tuttavia molto comodo avere direttamente a disposizione queste tre particolari forme di ciclo, o almeno due delle tre (tipicamente `while` e `for`).
- Analogamente, la selezione `switch` si può ottenere come una sequenza di `if` e sarebbe perciò superflua, tuttavia è molto comoda nella pratica.
- Le istruzioni indicate in uno pseudocodice non sottostanno a particolari sintassi o limitazioni: si può indicare esplicitamente qualsiasi tipo di operazione a patto che sia univocamente interpretabile (es. lettura o scrittura, operazioni matematiche, ecc.).

- A questo punto siamo in grado di formalizzare mediante pseudocodice l'algoritmo per determinare le cifre di n in base b (senza usare `goto`):

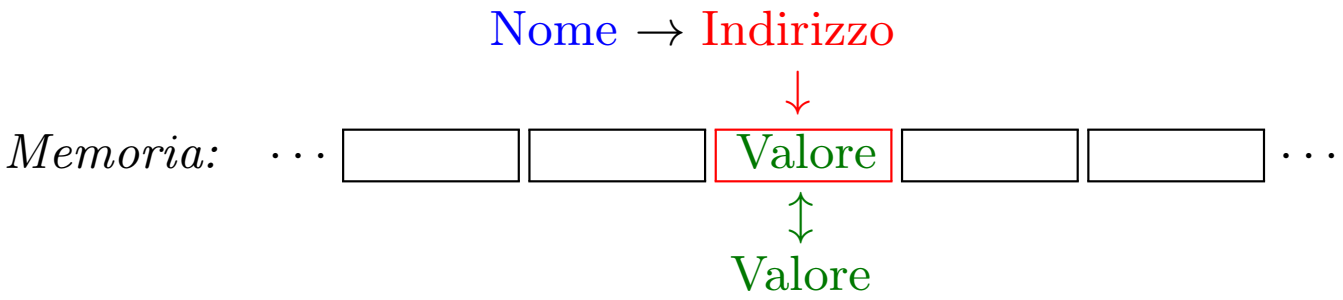
```
I1. Eseguire la divisione intera di n per b (n=q*b+r) memorizzando q ed r;
I2. Tornare ad I1 usando q al posto di n, finche' q=0.
```

- Prima di partire con la scrittura, vediamo di cosa abbiamo bisogno:
  - due variabili (intere) per gli input n e b;
  - una variabile (intera) per il quoziente q ad ogni iterazione, oltre allo spazio per memorizzare il risultato, cioè le k cifre (intere) di n in base b;
  - per evitare il `goto` utilizzeremo il ciclo `repeat-until`.

$(n, b) \leftarrow$ input
$q \leftarrow n$
$k \leftarrow 0$
repeat
  $(q, a_k) \leftarrow$ divisione intera di q per b
  $k \leftarrow k + 1$
until $q = 0$

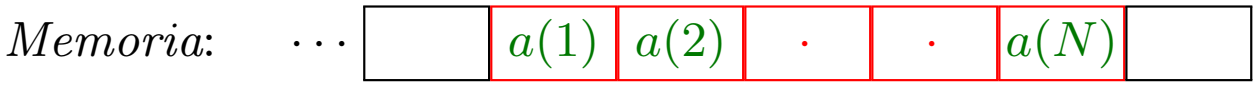
- Quando il programma si arresterà, avremo a disposizione in memoria le k cifre ricercate $a_0, a_1, \ldots, a_{k-1}$ tali che $n = (a_{k-1} \ldots a_1\ a_0)_b$.
- Non abbiamo fatto un controllo sui dati in ingresso: l'algoritmo potrebbe comportarsi in maniera non definita se i valori forniti non appartengono agli insiemi prescritti ($n, b \in \mathbb{N}$, $b > 1$).

- Nella scrittura del precedente pseudocodice, oltre alla struttura di controllo `repeat-until`, abbiamo usato "naturalmente" due elementi per memorizzare le quantità numeriche necessarie al funzionamento dell'algoritmo: delle *variabili* (n, b, q, k) e un *array* (a_k).
- Una variabile è una porzione di memoria (RAM) contenente un'informazione, identificata da un *nome*, un *indirizzo* ed un *valore*. Al nome della variabile viene associato l'indirizzo in memoria dove il valore di quella variabile viene opportunamente memorizzato:

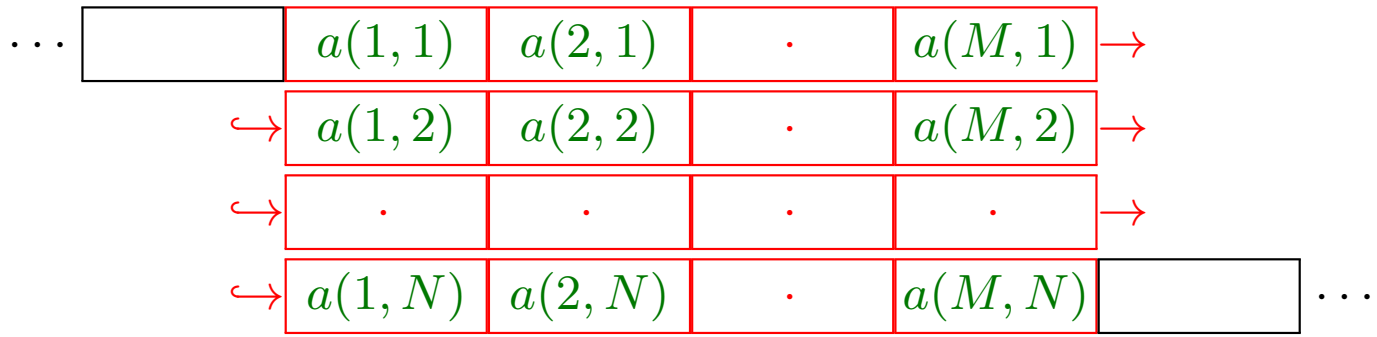


- Il *tipo* di una variabile identifica l'insieme dei possibili valori che essa può assumere: numero intero a k bit, con o senza segno, in virgola mobile in singola o doppia precisione, ecc. A seconda del linguaggio di programmazione, il tipo di una variabile deve essere dichiarato esplicitamente prima del suo utilizzo (es. C) oppure no (es. MATLAB).
- La *visibilità* di una variabile indica la possibilità di utilizzare un determinato identificatore (il nome della variabile) in un determinato punto del programma.

- Un array a è una struttura dati costituita da una sequenza ordinata di variabili *omogenee*, cioè dello stesso tipo, identificate da uno o più indici e memorizzate in porzioni contigue di memoria.
- Array monodimensionale: è l'equivalente di un *vettore* di lunghezza N, perciò ogni elemento è identificato da un solo indice i. Le notazioni impiegate sono a_i (notazione vettoriale), $a(i)$ (MATLAB), $a[i]$ (C). L'indice del primo elemento può essere 0 (C) oppure 1 (MATLAB):



- Array bidimensionale: è l'equivalente di una *matrice* di dimensione M righe $\times$ N colonne. Sono necessari due indici (i, j) e la notazione è quindi $a_{i,j}$, $a(i, j)$ oppure $a[i, j]$. L'ordine di accesso in memoria può essere per per righe (C) o per colonne (MATLAB):



- Un array D-dimensionale può sempre essere indicizzato attraverso un singolo indice, perciò il numero di dimensioni utilizzabili è arbitrario.

- Nella scrittura di un programma/pseudocodice, capita spesso di dover eseguire la medesima sequenza di operazioni su dati diversi. Esempio: necessità di calcolare una determinata funzione matematica $f(x)$ per diversi valori di x in punti diversi del programma.
- Una *funzione* (o routine) è un raggruppamento di istruzioni che svolge una determinata operazione. In analogia con una funzione matematica, essa prende in ingresso degli input detti *argomenti* e fornisce degli output. Input ed output possono essere in numero variabile e possono anche non essere presenti.
- Una funzione è identificata dal suo nome, dalla lista di argomenti in input ed output, e dal suo corpo. A seconda del linguaggio di programmazione che si andrà ad utilizzare, il tipo degli argomenti in input e output può essere richiesto (C) oppure no (MATLAB).

Definizione:

function $(y_1, y_2, \ldots, y_N)$ = nome_funzione $(x_1, x_2, \ldots, x_M)$
  istruzioni che calcolano $y_1, \ldots, y_N$ utilizzando $x_1, \ldots, x_M$
end

Utilizzo (chiamata):

$(\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_N) \leftarrow$ nome_funzione $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M)$

- Le variabili che sono definite nella funzione, ossia gli input $x_1, \ldots, x_M$ e gli output $y_1, \ldots, y_N$ in primis, più eventuali altre variabili definite nel corpo della funzione, sono *variabili locali* cioè sono visibili solo all'interno del corpo della funzione stessa. Ciò significa che al di fuori della funzione il nome di queste variabili non è legato a nessun riferimento di memoria ed il loro utilizzo fuori da quel contesto causerebbe errori in fase di compilazione (linguaggi compilati) o di esecuzione (linguaggi interpretati).
- A volte è possibile risolvere elegantemente un problema attraverso l'uso di funzioni che chiamano se stesse: questo meccanismo è detto *ricorsione*.
- Esempio: calcolo di un polinomio $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_N x^N$. Raccogliendo ripetutamente x si ottiene:

$$p(x) = a_0 + x(a_1 + x(a_2 + x(\cdots + x(a_N))))$$

ossia lo schema ricorsivo è $a_k + x \cdot ricorsione$. In pseudocodice:

Definizione:

```
function y = polinomio_ricorsivo (a, x, k)
    y ← a_k
    if k < N
        y ← y + x · polinomio_ricorsivo (a, x, k + 1)
    end
end
```

- La chiamata che calcolerà tutto il polinomio sarà quindi:

$p \leftarrow$polinomio_ricorsivo $(a, x, 0)$

partendo perciò con $k = 0$, cioè dal primo coefficiente a_0.
- Nel caso di funzioni ricorsive, si ha che le variabili locali della funzione *chiamata oscurano* le omonime variabili locali della funzione *chiamante* rendendole non visibili.
- Funzioni e ricorsione *non* sono nuovi tipi di strutture di controllo che si aggiungono alle tre strutture di controllo fondamentali, ma sono semplicemente dei modi convenienti di combinarle: il flusso logico d'esecuzione e le modalità di programmazione ricadono sempre nella prassi della programmazione strutturata.
- Ovviamente è possibile calcolare il polinomio $p(x)$ anche senza ricorsione, utilizzando un più semplice ciclo `for`. In questo caso lo schema iterativo è $y \leftarrow a_k + x \cdot y$ e si partirà dall'ultimo coefficiente a_N per arrivare al primo, a_0:

$y \leftarrow 0$
for $k = N, N-1, \ldots, 1, 0$
  $y \leftarrow a_k + x \cdot y$
end

alla fine del quale il risultato del calcolo del polinomio $p(x)$ sarà contenuto nella variabile y.

- Scrivere uno pseudocodice di funzione che, dato un vettore a di N numeri $a_1, \ldots, a_N$, fornisca:
  - il massimo degli a_i;
  - l'indice i del valore massimo.
- Sarà sufficiente scorrere il vettore dall'inizio alla fine, tenendo traccia sia del valore massimo incontrato, memorizzato nella variabile Max, che del suo indice i_M.
- Si utilizzerà quindi un ciclo `for` per scorrere il vettore, dal momento che la sua dimensione N è un dato.

Funzione che determina il massimo di un vettore numerico ed il relativo indice.
Input: vettore a di N numeri
Output: il massimo Max ed il relativo indice i_M

---

function $(Max, i_M) =$ massimo_vettore(a)
  $Max \leftarrow a_1$
  $i_M \leftarrow 1$
  for $i = 2, \ldots, N$
    if $a_i > Max$
      $Max \leftarrow a_i$
      $i_M \leftarrow i$
    end
  end
end

- Scrivere uno pseudocodice per calcolare il fattoriale $n!$ di un intero positivo n utilizzando una funzione.

$$n! = \underbrace{n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1}_{\text{iterativo}} = \underbrace{n \cdot (n-1)!}_{\text{ricorsivo}}$$

- Le due formulazioni, equivalenti dal punto di vista matematico, suggeriscono due implementazioni diverse: un'implementazione iterativa, con un ciclo `for`, ed una ricorsiva.
- È buona norma inserire in testa allo pseudocodice quali sono gli input e quali gli output ed eventualmente un commento/titolo descrittivo dell'oggetto dello pseudocodice.

Funzione per il calcolo del fattoriale, versione iterativa.
Input: numero naturale n
Output: il fattoriale $m = n!$

```
function m = fattoriale_it(n)
  m ← 1
  for i = 1, ..., n
    m ← m · i
  end
end
```

Funzione per il calcolo del fattoriale, versione ricorsiva.
Input: numero naturale n
Output: il fattoriale $m = n!$

```
function m = fattoriale_ric(n)
  if n > 1
    m ← n · fattoriale_ric(n − 1)
  else
    m ← 1
  end
end
```

- Un algoritmo dev'essere finito: il numero di operazioni richieste deve essere finito.
- Dal punto di vista pratico il numero di operazioni deve essere anche ragionevolmente limitato. Come si può misurare questa proprietà?
- Abbiamo visto che ad un'istruzione può corrispondere una o più operazioni. Nel caso dell'algoritmo per la determinazione delle cifre di n in base b vi era un ciclo `repeat-until` nel quale l'operazione di divisione intera veniva ripetuto un numero finito k di volte, con k dipendente da n e da b. Come facciamo a determinare il numero k di iterazioni?
  - ad ogni iterazione viene ripetuta la divisione intera del numero in ingresso per la base;
  - otteniamo quindi, approssimativamente, la seguente sequenza di quozienti q, ognuno corrispondente ad un'iterazione:

    $$\frac{n}{b} \quad \frac{n}{b^2} \quad \frac{n}{b^3} \quad \cdots \quad \frac{n}{b^{k-1}} \qquad \frac{n}{b^k} \approx 1$$

    in quanto all'ultima iterazione si ha $q = 0$, cioè alla penultima iterazione $q < b \Rightarrow q/b < 1$ cioè 1 è un'approssimazione dell'ordine di grandezza del quoziente all'ultima iterazione;
  - risolvendo rispetto a k l'ultimo quoziente otteniamo $k \approx \log_b n$;
  - alternativamente, k è il numero di cifre di n in base b ($a_{k-1} > 0$):

    $$n = (a_{k-1} \dots a_0)_b \quad \Rightarrow \quad b^{k-1} \leq n < b^k$$

    da cui otteniamo esattamente $k = \lceil \log_b n \rceil$.

- Il numero totale p di operazioni algebriche da effettuare sarà quindi:

$$p(n) = c \cdot k = c \cdot \lceil \log_b n \rceil$$

  dove c è il numero di operazioni algebriche richieste da una divisione intera.
- Si definirà *complessità computazionale* di un algoritmo la funzione $f(n)$ che esprime il numero totale di operazioni logico-aritmetiche richieste dall'algoritmo per risolvere un problema dato un input di dimensione n.
- Le *notazioni asintotiche* si impiegano solitamente per esprimere sinteticamente le complessità degli algoritmi e forniscono delle limitazioni asintotiche alla crescita di una funzione.
  - Notazione Θ:

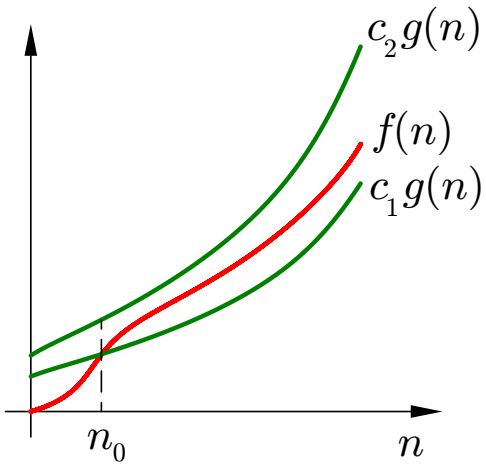
$$\Theta\big(g(n)\big) = \big\{f(n) : \exists c_1, c_2, n_0 > 0 \text{ tali per cui } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \;\; \forall n > n_0\big\}$$

  - Notazione Ω (limitazione inferiore):

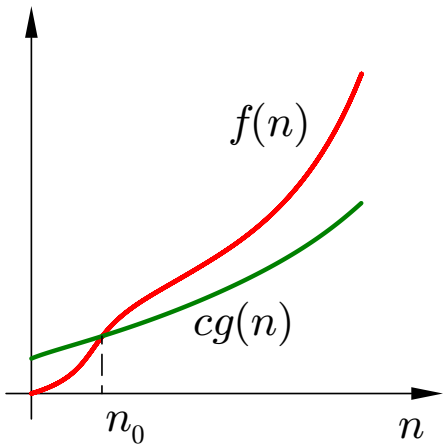
$$\Omega\big(g(n)\big) = \big\{f(n) : \exists c, n_0 > 0 \text{ tali per cui } 0 \leq c \cdot g(n) \leq f(n) \;\; \forall n > n_0\big\}$$

  - Notazione $\mathcal{O}$ (limitazione superiore):

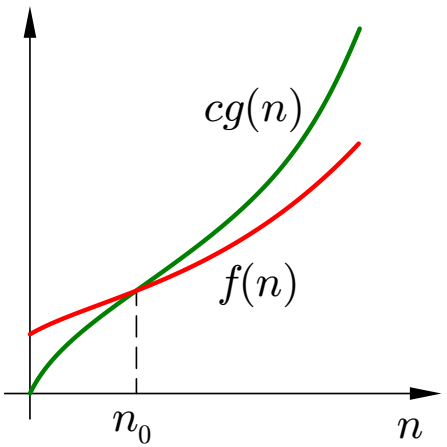
$$\mathcal{O}\big(g(n)\big) = \big\{f(n) : \exists c, n_0 > 0 \text{ tali per cui } 0 \leq f(n) \leq c \cdot g(n) \;\; \forall n > n_0\big\}$$



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



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



$f(n) = \mathcal{O}\big(g(n)\big)$

- Le precedenti notazioni asintotiche prescindono dalle costanti moltiplicative e/o additive:

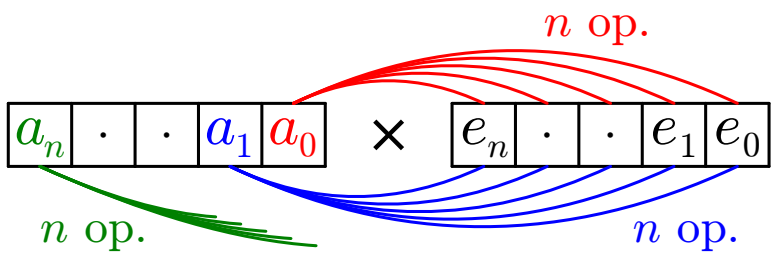
$$\Theta\big(a + c \cdot f(n)\big) = \Theta\big(f(n)\big)$$

- Potremo quindi scrivere che la complessità computazionale dell'algoritmo che stiamo analizzando è:

$$p(n) = c \cdot \lceil \log_b n \rceil = \Theta(\log n)$$

- Anche lo spazio di memoria richiesto da un algoritmo si esprime solitamente con la notazione asintotica. Nel caso del nostro algoritmo dovremo memorizzare k cifre, quindi lo spazio di memoria richiesto è anch'esso $\Theta(\log n)$.

- Nell'analisi dell'algoritmo precedente si è considerato n, il numero del quale determinare le cifre, come dimensione dell'input.
- In questo caso è più corretto utilizzare il numero delle cifre di n, indipendentemente dalla base impiegata, come dimensione dell'input. È infatti il numero k delle cifre di n che caratterizza correttamente la quantità di informazioni in ingresso e non la grandezza del numero stesso.
- Con la precedente definizione si ottiene ovviamente che la complessità computazionale dell'algoritmo considerato è $\Theta(n)$, si dice cioè che possiede complessità *lineare*: il raddoppio della dimensione dell'input comporta un raddoppio delle operazioni logico-aritmetiche richieste.
- Nessun algoritmo può avere una complessità meno che lineare poichè il semplice compito di leggere le n informazioni in ingresso ha un costo lineare, $\Theta(n)$.
- Per esempio, la somma di due numeri interi a n cifre ha ovviamente complessità lineare, $\Theta(n)$. Il prodotto di due numeri interi a n cifre ha invece complessità quadratica, $\Theta(n^2)$: se si raddoppia n, il numero di operazioni logico-aritmetiche richieste quadruplica:

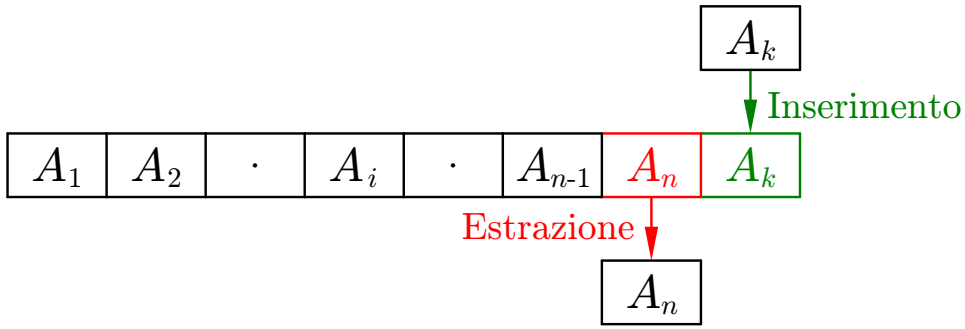


- Gli algoritmi visti finora operano su dati organizzati in maniera "semplice": variabili e array.
- Nell'affrontare problemi pratici ci si trova spesso ad operare su dati organizzati secondo strutture logiche più complesse, denominate genericamente *strutture dati*, che tuttavia si possono sempre implementare utilizzando opportunamente variabili ed array.
- *Lista lineare:* insieme ordinato di n dati *omogenei*. Ogni elemento A_i della lista lineare può essere identificato univocamente dal suo indice i:

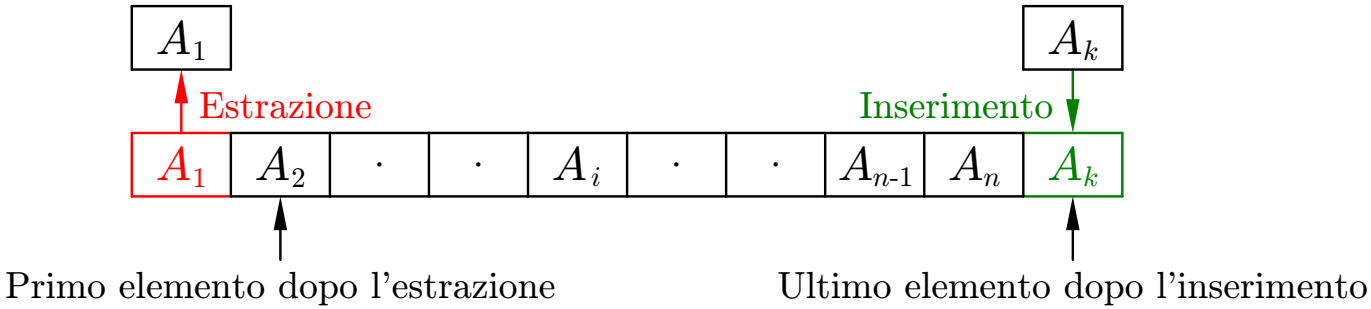
A_1	A_2	·	·	A_i	·	·	$A_{n\text{-}1}$	A_n

  ma a differenza degli array, l'accesso ad un generico elemento della lista *non* può avvenire direttamente attraverso l'indice i ma avviene in maniera sequenziale a partire dal primo elemento della lista. Un'altra differenza rispetto agli array è che la lunghezza n può essere variabile.
- Tipiche operazioni sulle liste lineari:
  - accedere ad un generico elemento i per estrarne il valore o modificarlo;
  - inserire o estrarre (ed eliminare) un elemento prima o dopo un generico elemento A_i, in particolare il primo, A_1, o l'ultimo, A_n.

- Se le operazioni di inserimento od estrazione di un elemento avvengono solo sul primo o sull'ultimo elemento, le liste prendono nomi particolari.
- *Pila:* lista lineare nella quale sia le estrazioni che gli inserimenti avvengono solo sull'ultimo elemento. È chiamata anche lista LIFO (Last In, First Out):

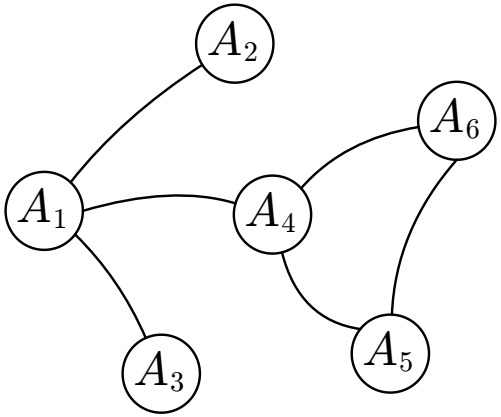


- *Coda:* lista lineare nella quale le estrazioni avvengono solo sul primo elemento (testa) e gli inserimenti avvengono solo dopo l'ultimo elemento (fondo). È chiamata anche lista FIFO (First In, First Out):

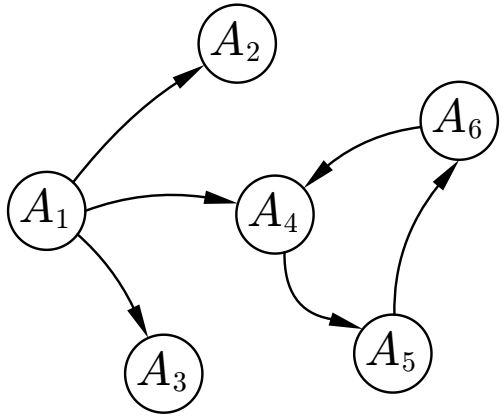


- *Doppia coda:* estrazioni ed inserimenti possono avvenire sia in testa che al fondo della lista.

- *Grafo:* struttura costituita da un insieme V di *vertici* e da un insieme E di *archi* che collegano coppie di vertici: $G = (V, E)$. Ad ogni nodo è possibile associare un dato o valore A_i.

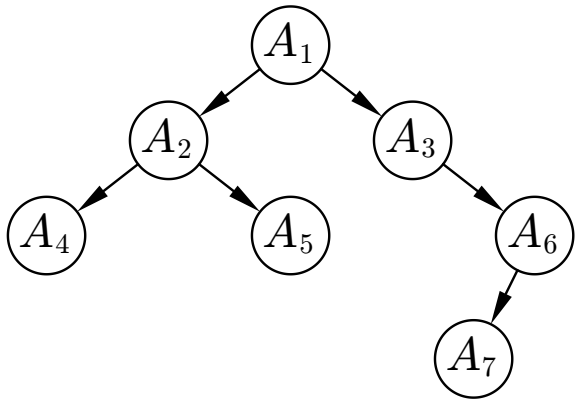


Grafo

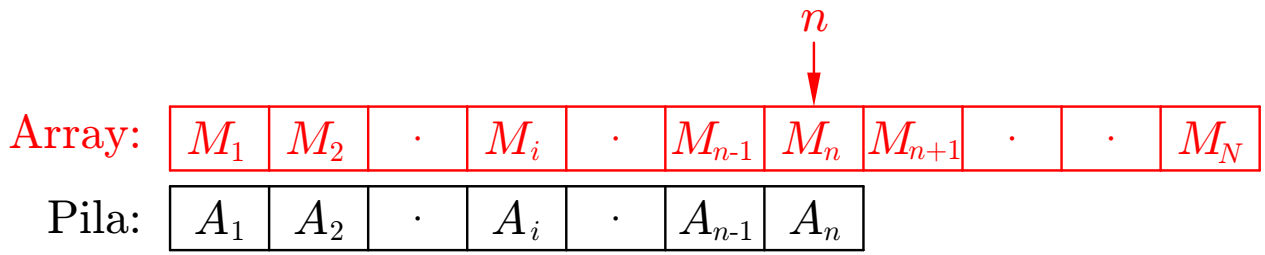


Grafo orientato

- Se gli archi sono orientati, il grafo si dice *orientato*.
- *Albero:* grafo orientato nel quale vi è un solo nodo senza archi entranti (*radice*), e nodi senza archi uscenti (*foglie*). Partendo dalla radice ed escludendo le foglie, ogni nodo (*genitore*) ha un certo numero di *figli*, ognuno dei quali non può mai avere genitori diversi. In un albero *binario* ogni nodo può avere al massimo 2 figli.

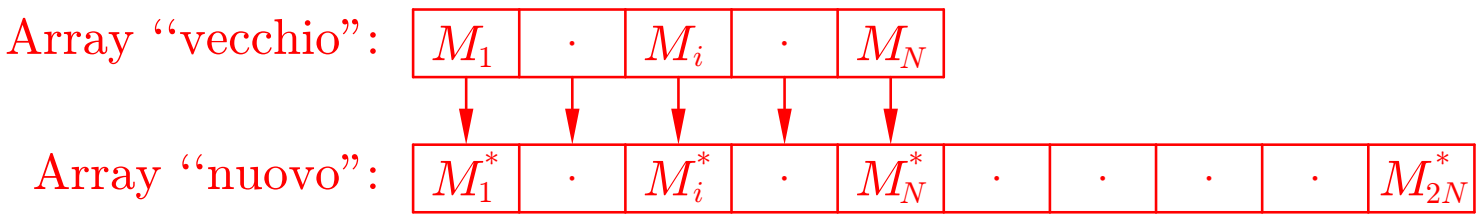


- Le strutture dati appena viste (liste e grafi) sono strutture logiche che per essere utilizzate in un computer necessitano di un'implementazione concreta.
- L'implementazione concreta avviene utilizzando gli strumenti di cui disponiamo: variabili ed array, opportunamente combinati.
- Abbiamo visto che un array, da solo, non può mai essere utilizzato per implementare direttamente una lista in quanto le operazioni di inserimento ed estrazione ne modificano continuamente la lunghezza.
- È necessario "arricchire" un semplice array M per poterlo utilizzare come lista, usando altri elementi.
- Nel caso della pila, dove le estrazioni e gli inserimenti possono avvenire solo sull'ultimo elemento, è sufficiente aggiungere una variabile intera n che rappresenta l'indice dell'ultimo elemento della pila nell'array M:

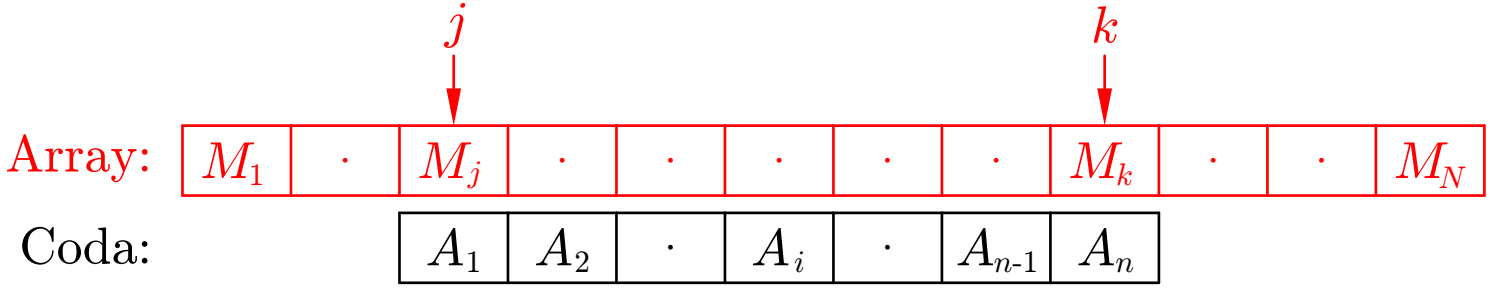


- Ad ogni estrazione si provvederà a leggere l'array M in posizione n, cioè $M(n)$ ed ottenendo A_n, e si decrementerà n di 1.
- Ad ogni inserimento si provvederà ad incrementare n di 1 e si scriverà nell'array M sempre in posizione n, cioè $M(n) \leftarrow A_{new}$.

- Nel particolare caso della pila implementata mediante un array, è possibile accedere direttamente ad ogni elemento della pila utilizzando l'indice i di lista poichè non ci sono inserimenti o estrazioni in posizioni intermedie della pila, ma solo in coda, quindi ogni elemento della pila A_i coinciderà sempre il corrispondente elemento dell'array $M(i)$, $1 \leq i \leq n$.
- Ovviamente bisogna prendere in considerazione i casi estremi di estrazione quando la pila è vuota ($n = 0$), ed inserimento quando l'array è completamente occupato dalla pila ($n = N$, dove N è la dimensione dell'array M).
- Nel primo caso (pila vuota) l'estrazione dovrà essere proibita per non accedere a celle di memoria non indicizzate ($n < 1$), mentre nel secondo caso (array pieno) bisognerà eventualmente predisporre un aumento della dimensione N dell'array di supporto.
- Siccome l'allocazione di memoria aggiuntiva per un array non può avvenire generalmente sulle celle di memoria contigue a quelle dell'array di partenza, l'aumento della dimensione N dell'array può avvenire solo allocando un altro array di dimensione maggiore di N e copiando i primi N valori:

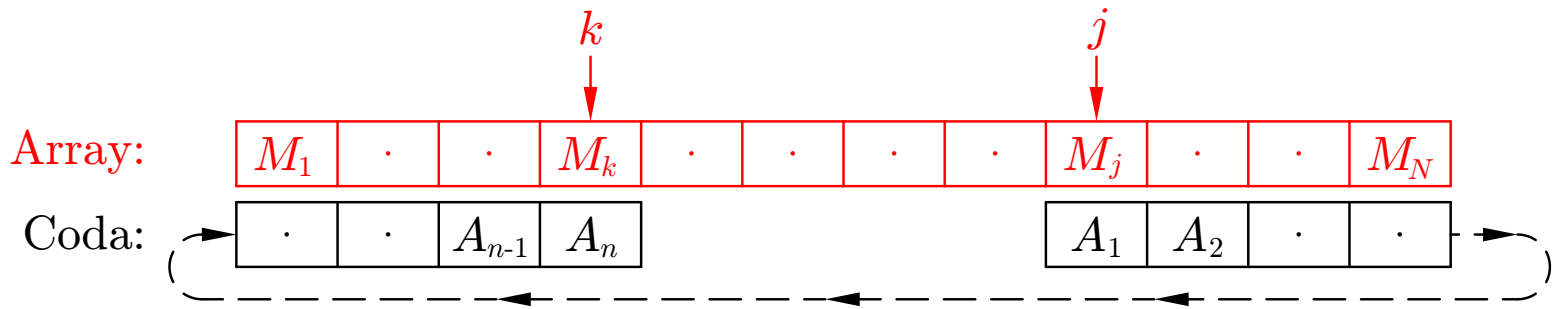


- Nel caso della coda, dove le estrazioni avvengono in testa e gli inserimenti avvengono al fondo, è ancora possibile utilizzare un array come struttura di supporto per l'implementazione.
- In analogia con la pila, è necessario utilizzare due variabili intere j ed k per rappresentare, rispettivamente, l'indice del primo e dell'ultimo elemento della coda nell'array M:

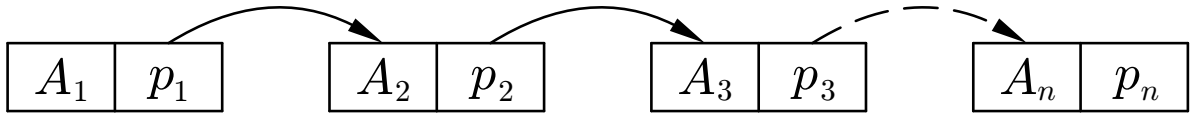


- Ad ogni estrazione in testa si provvederà a leggere l'array M in posizione j, cioè $M(j)$ ed ottenendo A_1, e si incrementerà j di 1.
- Ad ogni inserimento si provvederà ad incrementare k di 1 e si scriverà nell'array M sempre in posizione k, cioè $M(k) \leftarrow A_{new}$.
- In questo particolare caso di coda implementata mediante un array, è ancora possibile accedere direttamente ad ogni elemento della coda utilizzando l'indice i di lista poichè gli elementi sono comunque contigui in memoria: sono semplicemente spostati a destra di $(j-1)$ posizioni rispetto alla prima cella $M(1)$. L'accesso avverrà quindi come $M(i+j-1)$.

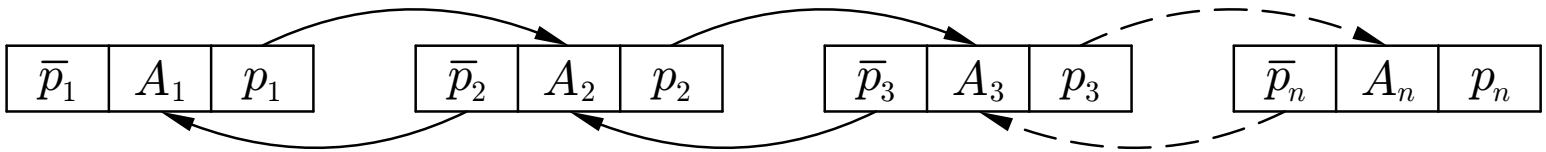
- Anche in questo caso bisognerà prendere in considerazione i casi estremi di estrazione quando la coda è vuota, ed inserimento quando l'array è completamente occupato dalla coda.
- Nel primo caso (coda vuota) l'estrazione dovrà essere proibita, anche se essa non comporta l'accesso a celle di memoria non indicizzate, mentre nel secondo caso (array pieno) bisognerà comunque predisporre un aumento della dimensione N dell'array di supporto, in maniera analoga a quanto fatto nel caso della pila.
- A differenza della pila, l'accesso a celle di memoria che eccedono la dimensione N dell'array di supporto può verificarsi anche se la lunghezza effettiva della coda n è minore dello spazio a disposizione N: ad ogni coppia di estrazioni/inserimenti successivi la coda si "sposta a destra" di una cella in memoria, mentre la lunghezza n resta invariata, arrivando prima o poi alla fine dell'array ($k > N$).
- Per far fronte a questa possibilità senza dover aumentare la dimensione dell'array, è possibile sfruttare l'array in maniera *circolare*:



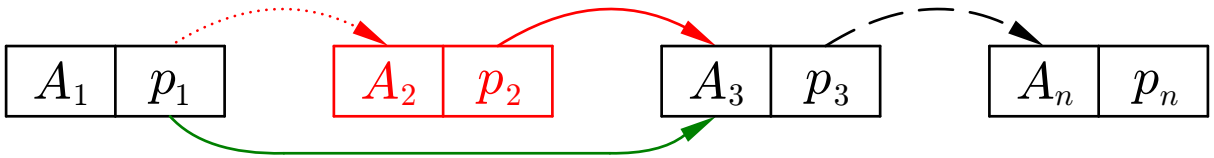
- Nel caso di liste generiche con inserimenti ed estrazioni in posizioni arbitrarie della lista, l'uso dell'array con uno/due indici di testa/fondo non è ovviamente più possibile perchè la contiguità dei dati non è più rispettata.
- In questo caso si utilizzano strutture *concatenate*: per ogni elemento della lista viene memorizzato non solo il valore A_i di quell'elemento ma anche un *puntatore* p_i che rappresenta l'indirizzo dell'elemento successivo A_{i+1}. Si dice che p_i *punta ad* A_{i+1}:



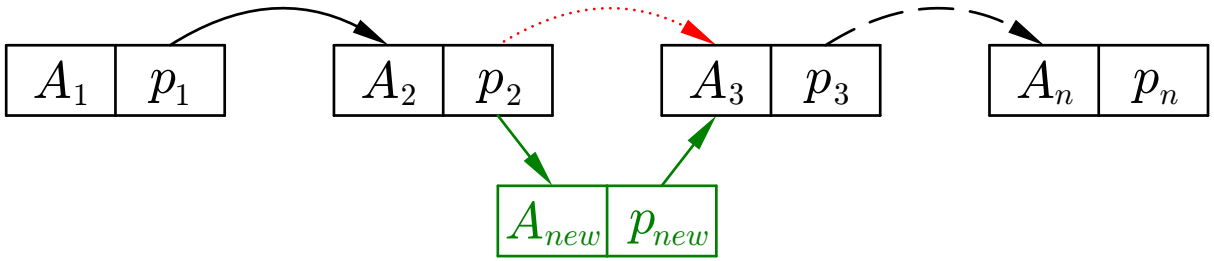
- La precedente struttura concatenata che impiega un solo puntatore all'elemento successivo (*puntatore avanti*) è chiamata *catena semplice* o *unidirezionale*.
- Oltre al puntatore avanti, per ogni elemento si può memorizzare anche il *puntatore indietro* $\bar{p}_i$ che punta all'elemento precedente A_{i-1}, dando origine alla *catena doppia* o *bidirezionale*:



- Nelle strutture concatenate è sempre necessario impiegare almeno una variabile aggiuntiva che identifica il primo elemento della lista, per esempio un puntatore p_0 che punta ad A_1. Il puntatore avanti p_n dell'ultimo elemento non punta a nulla, così come il puntatore indietro $\bar{p}_1$ del primo elemento.
- Con la particolare implementazione concatenata, le operazioni di estrazione ed inserimento diventano molto semplici.
- Consideriamo per semplicità l'estrazione (eliminazione) di un elemento i da una catena semplice: sarà sufficiente reindirizzare il puntatore p_{i-1} dell'elemento precedente all'elemento successivo $i+1$:



- Nel caso invece di inserimento di un nuovo elemento tra le poisizione i ed $i+1$, sarà sufficiente reindirizzare il puntatore p_i al nuovo elemento, il cui puntatore indirizzerà all'elemento $i+1$:



- Scrivere uno pseudocodice di funzione che, data una pila implementata mediante un array A di N elementi e l'indice n dell'ultimo elemento, acceda (legga) all'elemento i-esimo.
- Abbiamo visto che l'accesso alle pile implementate con array può avvenire direttamente tramite l'indice i. Bisognerà comunque precludere la possibilità di accesso al di fuori dell'array. In tal caso si fornirà un messaggio di *warning*.

Funzione che accede ad un elemento di una pila.
Input: pila rappresentata dall'array A di N elementi e dall'indice di fondo n
l'indice i dell'elemento a cui accedere
Output: il valore A_{acc} dell'elemento richiesto

---

function A_{acc} = accedi_pila(A,n,i)
  if $i > 0$ & $i \leq n$
    $A_{acc} \leftarrow A(i)$
  else
    $A_{acc} \leftarrow$ *null*
    warning " Accesso invalido! "
  end
end

- & è il simbolo dell'operatore logico (Booleano) AND.

- Scrivere uno pseudocodice di funzione che, data una pila implementata mediante un array A di N elementi e l'indice n dell'ultimo elemento, inserisca al fondo della pila un nuovo elemento dato A_{new}.
- Bisognerà tenere conto dell'eventuale riempimento dell'array A, la cui dimensione andrà quindi adeguatamente aumentata come abbiamo visto.

Funzione che inserisce un elemento in una pila.
Input: pila rappresentata dall'array A di N elementi e dall'indice di fondo n
nuovo elemento A_{new}
Output: pila rappresentata nello stesso modo

---

function $(A, n) =$ inserisci_in_pila(A,n,A_{new})
  $n \leftarrow n + 1$
  if $n > N$
    $A \leftarrow [A; A]$
  end
  $A(n) \leftarrow A_{new}$
end

- L'istruzione $A \leftarrow [A; A]$ sta a significare che l'array A viene sostituito da un nuovo array ottenuto concatenando A con se stesso, quindi raddoppiandone la lunghezza e mantenendo invariati i primi N elementi.

- Scrivere uno pseudocodice di funzione che, data una pila implementata mediante un array A di N elementi e l'indice n dell'ultimo elemento, estragga l'ultimo elemento della pila.
- Bisognerà tenere conto dell'eventuale svuotamento della pila per non accedere a celle di memoria non indirizzate. In tal caso bisognerà semplicemente fornire un messaggio di *warning*.

Funzione che estrae un elemento da una pila.
Input: pila rappresentata dall'array A di N elementi e dall'indice di fondo n
Output: pila rappresentata nello stesso modo
il valore dell'elemento estratto A_{ex}

---

function (A, n, A_{ex}) = estrai_da_pila(A,n)
  if $n > 0$
    $A_{ex} \leftarrow A(n)$
    $n \leftarrow n - 1$
  else
    $A_{ex} \leftarrow$ *null*
    warning " Pila vuota! "
  end
end

- Abbiamo visto che la coda si può implementare mediante un array e due indici per memorizzare le posizioni di testa e di fondo della coda nell'array.
- Dal punto di vista della leggibilità e della compattezza di un programma, sarebbe comodo "impacchettare" i tre precedenti elementi, non omogenei, in un'unica entità, dal momento che è il loro insieme che definisce la coda.
- Questo nuovo tipo di dato è chiamato *record* ed è un insieme ordinato di elementi eterogenei detti *campi*, ognuno individuato dal proprio nome.
- Nel caso della coda, potremo definire il tipo record *coda* come la sequenza di tre elementi, nell'ordine:
  - un array di supporto, identificato dal nome A, per esempio;
  - un intero che rappresenta l'indice di testa della coda, identificato dal nome $indice\_testa$, per esempio;
  - un intero che rappresenta l'indice di fondo della coda, identificato dal nome $indice\_fondo$, per esempio.
- Dato un record r, si può accedere per leggere o modificare ognuno dei suoi campi con la notazione "punto": $r.nome\_campo$.
- Nel caso della coda, in conseguenza alla predente definizione del tipo *coda*, potremo accedere ai tre campi di un record c, di tipo *coda*, nel seguente modo:
  - $c.A$ per accedere all'array di supporto;
  - $c.indice\_testa$ per accedere all'indice di testa nell'array di supporto;
  - $c.indice\_fondo$ per accedere all'indice di fondo nell'array di supporto.

- Sia data una coda c espressa in forma di record di tipo *coda*, costituita perciò da un array di dimensione N e dai due indici di testa e di fondo. Scrivere uno pseudocodice di funzione che inserisce al fondo della coda un nuovo elemento dato A_{new}. Si assume per semplicità che la dimensione dell'array è sempre maggiore della lunghezza della coda.
- Bisognerà tenere conto dell'eventuale raggiungimento dell'ultima posizione disponibile nell'array in fase di inserimento, sfruttando la circolarità sull'array:

Funzione che inserisce un elemento in una coda.
Input: coda c di tipo *coda*
nuovo elemento A_{new}
Output: coda c rappresentata nello stesso modo

---

function c = inserisci_in_coda(c, A_{new})
  $c.indice\_fondo \leftarrow c.indice\_fondo + 1$
  if $c.indice\_fondo > N$
    $c.indice\_fondo \leftarrow c.indice\_fondo - N$
  end
  $n \leftarrow c.indice\_fondo$
  $c.A(n) \leftarrow A_{new}$
end

- Sia data una coda c espressa in forma di record di tipo *coda*, costituita perciò da un array di dimensione N e dai due indici di testa e di fondo. Scrivere uno pseudocodice di funzione che estrae un elemento dalla testa della coda.
- Bisognerà tenere conto dell'eventuale raggiungimento dell'ultima posizione disponibile nell'array in fase di estrazione, sfruttando la circolarità sull'array:

Funzione che estrae un elemento da una coda.
Input: coda c di tipo *coda*
Output: coda c rappresentata nello stesso modo
il valore dell'elemento estratto A_{ex}

---

function $(c, A_{ex}) =$ estrai_da_coda(c)
  $n \leftarrow c.indice\_testa$
  $A_{ex} \leftarrow c.A(n)$
  $c.indice\_testa \leftarrow c.indice\_testa + 1$
  if $c.indice\_testa > N$
    $c.indice\_testa \leftarrow c.indice\_testa - N$
  end
end

- Sia data una coda c espressa in forma di record di tipo *coda*, costituita perciò da un array di dimensione N e dai due indici di testa e di fondo. Scrivere uno pseudocodice di funzione che fornisce la lunghezza n della coda. Si assume per semplicità che la dimensione dell'array è sempre maggiore della lunghezza della coda ($N > n$).
- Per le definizioni degli indici di testa e di fondo, la lunghezza della coda sarà data da $n = indice\_fondo - indice\_testa + 1$. Bisognerà però tenere opportunamente conto dell'eventuale circolarità sull'array (coda a cavallo tra fine ed inizio array):
  - A1. $indice\_testa \leq indice\_fondo$ e $n < N \Rightarrow$ caso senza circolarità;
  - A2. $indice\_testa \leq indice\_fondo$ e $n = N \Rightarrow$ coda vuota con circolarità;
  - B. $indice\_testa = indice\_fondo + 1 \Rightarrow$ coda vuota senza circolarità;
  - C. $indice\_testa > indice\_fondo + 1 \Rightarrow$ caso con circolarità.

Funzione che determina la lunghezza di una coda.
Input: coda c di tipo *coda*
Output: lunghezza n dela coda

```
function n = lunghezza_coda(c)
    n ← c.indice_fondo − c.indice_testa + 1
    if n = N
        n ← 0
    end
    if n < 0
        n ← n + N
    end
end
```

- Sapendo che la lunghezza n di una coda implementata con un array di lunghezza N deve soddisfare $0 \leq n < N$, si può impiegare il seguente "trucco" per scrivere la precedente funzione in maniera molto più compatta:

Funzione che determina la lunghezza di una coda.
Input: coda c di tipo *coda*
Output: lunghezza n dela coda

---

function n = lunghezza_coda(c)
  $n \leftarrow (c.indice\_fondo - c.indice\_testa + 1 + N) \mod N$
end

- "$m \mod N$" indica il resto della divisione intera di m per N.