Contents

Programmazione ed Algoritmi

Realizzato da: Giuntoni Matteo

A.A 2021-2022

---

1 Introduzione

1.1 Puntatori

Gli indirizzi di memoria delle variabili sono interi (rappresentati in esadecimale) che contano i byte a partire dalla posizione 0x0000000. Gli indirizzi di memoria possono essere memorizzati in variabili come ogni altro intero.

Definizione 1.1 (Puntatori)**.** *Definiamo le variabili che memorizzano indirizzi di memoria come* ***puntatori****.*

1.1.1 Operatori sui puntatori

Sono due i principali operatori che possono essere usai con i puntatori.

- **(&) - Operatore indirizzo**. L'operatore di indirizzo è unario[1] e restituisce l'indirizzo di memoria dell'operando (può essere anche un altro puntatore, in questo caso restituisce l'indirizzo in cui è memorizzato il puntatore, cioè l'indirizzo di memoria di una variabile).
- **(*) - Operatore di indirezione o dereferenziazione**. L'operatore di indirezione è unario e restituisce il valore dell'oggetto a cui punta l'operanda.

Note 1.1.1. *Nota che & e * sono uno l'inverso dell'altro, quindi: &*aPtr == *&aPtr.*

Esempio 1.1. Di seguito un esempio di utilizzo di puntatori con anche i vari operatori.

```
1  var a:Character = 'z', b:Character = 'h';
2  ref aPtr:Character = nil                          //0x0
3  aPtr = 0;                                         //0x0
4  aPtr = &a;
5  print(&a, aPtr);                                  //0x7ffeefbff60f, 0x7ffeefbff60f
6  print(*aPtr, a);                                  //z, z
7  print(&aPtr);                                     //0x7ffeefbff60f
8  *aPtr = b;
9  print(*aPtr, a, b);                               //h, h, h
10 print(&b, &a, aPtr);                              // 3 volte 0x7ffeefbff60f
11 ref altro_aPtr:Character = &a;
12 print(altro_aPtr, *altro_aPtr);                   //0x7ffeefbff60f, h
```

Listing 1: Esempio puntatori e operatori sui puntatori

Descrizione esempio: Osservando l'esempio sopra 1.1 possiamo vedere alle righe (11) e (12) che abbiamo una dichiarazione di un nuovo puntatore che punta alla stessa cella di memoria di "aPtr", abbiamo quindi più di un puntatore che punta alla stessa variabile.

Possiamo vedere che le locazioni di memoria sono numeri interi che individuano la posizione della cella di memoria (sono numeri interi scritti in esadecimale, ma sempre numeri interi), è quindi possibili effettuare operazioni aritmetiche sui puntatori, cioè è possibile manipolare tramite operazioni aritmetiche le locazioni.

[1] Unario vuol dire che agisce su una sola variabile

Esempio 1.2. Se per esempio prendiamo un ambiente ed una memoria così composti:

$$\rho = [(aPtr, l2)] \qquad \sigma = [(l1, 13), (l2, 23)]$$

Abbiamo quini un puntatore "aPtr" che punta alla locazione l1, il quale contiene il numero 13, più una posizione l2, successiva alla l1, che contiene 23.
Se ipotizziamo che il nostro sistemi archivi le informazioni con una base di 32 bit [2] ed andiamo a sommare 32 a "aPtr" succederà che ci sposteremo di 32 posti nella memoria raggiungendo l2, quindi "aPtr" = l2.

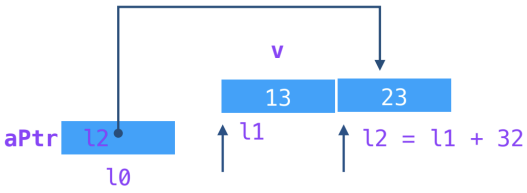


Figure 1: Operazioni algebriche su puntatori

1.2 Strutture dati

Definizione 1.2 (Struttura dati)**.** *Una* ***struttura dati*** *è un formato che serve ad organizzare e memorizzare dati in modo da renderli agevolmente disponibili agli algoritmi che li manipolano.*

Alcune caratteristiche delle strutture dati:

- Una struttura dati è detta **omogenea** se contiene dati tutti dello stesso tipo. Altrimenti è **disomogenea**.
- Una struttura dati è **statica** se la sua dimensione non varia durante l'esecuzione del programma. Altrimenti è detta **dinamica**.
- Una struttura dati è **lineare** se i dati sono organizzati come sequenze di valori. Altrimenti è detta **non lineare**.

Una struttura dati è inoltre caratterizzata dalle operazioni elementari disponibili per inserire, reperire e modificare i dati che memorizza.

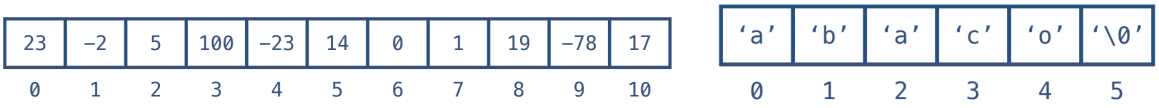
[2] Questo vuol dire che ogni vaore è salvato con 32 bit, quindi ogni 32 ci sarà un nuovo valore archiviato

2 Array

Una struttura dati molto conosciuta e chiamata array.

Definizione 2.1 (Array)**.** *Gli array sono delle strutture dati omogenea, statiche e lineari implementate mediante un gruppo di celle contigue di memoria dello stesso tipo.*

Di seguito due esempi grafici di array uno di interi ed uno di stringhe, da notare sotto la posizione degli elementi nell'array che si conta partendo dallo 0.



(a) (b)

Figure 2: In (a) un array lungo 11 di interi, in (b) un array lungo 6 di caratteri

Note 2.0.1. *Nota che nell'array di caratteri sopra nell'ultima posizione c'è sempre* \0 *(Null).*

Negli array si accede mediante l'indice della posizione nella sequenza. Si possono inoltre effetturare sugli elementi tutte le operazioni definte sul tipo corrispondete agli elementi dell'array.

Esempio 2.1. Alcuni esempi di accesso ed operazioni su gli arrey sopra:

- a[6] == 0 a[3] == 100 b[2] == 'a'
- a[4] = a[5] + a[7] (a[5] == 14, a[7] == 1, quindi il risultato sarà 14 + 1 = 15)

Inoltre possiamo dire che gli array sono allocati in memoria quando il controllo del flusso a tempo di esecuzione entra nel blocco in cui sono definiti e sono distrutti quando il controllo esce dal blocco.

Il nome dell'array è una variabile che contiene la locazione di memoria in cui è memorizzata la prima cella. Essendo che le celle sono contigue e hanno tutte lo stesso tipo basta infatti conoscere la posizione della prima cella per poi, tramite una semplice operazione algebrica di somma, accedere a quelle successive. In generale possiamo scrivere che:

$$a[i] \quad = \quad \sigma(\rho(a) + size(type(a)) \times i)$$

Esempio 2.2. Se abbiamo un array di lunghezza 11, ed chiamiamo la prima locazione (quella dove è contenuto il primo elemento dell'array) loc1, per raggiungere la posizione numero 10 basterà eseguire l'operazione loc1 + 32*10.

Questo consente l'accesso diretto agli elementi degli array con una sola operazione indipendentemente dalla lunghezza dell'array (consto di accesso costante).

2.1 BinarySearch

Problema: Dato un elemento (o chiave) k, determinare se esiste all'interno di un array ordinato A di n elementi. Se l'elemento esiste, si restituisce la sua posizione, altrimenti -1. Soluzione con ricerca binaria.
Proprietà: $\forall i \in [0..n-1] \,.\, A[i] \leq A[i+1]$
Questa proprietà dice che l'array A deve essere obbligatoriamente ordinato, sennò la ricerca binaria non potrà esser fatta.

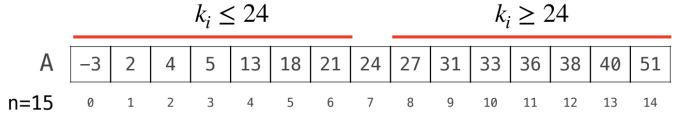


Figure 3: Array A in BinarySearch

2.1.1 Codice dell'algoritmo

```
1  function binSearch(k,A) {
2      var pos:Int = -1;
3      var sin:int = 0;
4      var dx:Int = n - 1;
5      while(sin <= dx && pos == -1){
6          const cen:int = (sin + dx)/2;
7          if (A[cen] == k) {pos = cen}
8          else if (k < A[cen]) {dx = cen - 1}
9          else {sin = cen + 1}
10     }
11     return pos;
12 }
```

Listing 2: Codice BinarySearch

Di seguito una spiegazione del funzionamento dell'algoritmo:

- **Righe 2-4:** Andiamo ad inizializzare 3 variabili: "pos" che indicherà la posizione dell'elemento da cercare, viene inizializzata a -1 perché nel caso non si trovasse ritorna così -1.
  "Sin" che indica il capo sinistro della posizione che stiamo analizzando, e "dx" che indica il capo destro, sono entrambi inizialmente inizializzati come gli estremi dell'array.
- **Riga 5:** La condizione del while dice in sintesi che finché non abbiamo trovato il valore (pos == -1) e finchè "sin" e "dx" non si scambiano (che vorrebbe dire che abbiamo finito le iterazioni possibili), continuare a ciclare.
- **Righe 6-9:** All'interno del while quello che andiamo a fare e prendere il centro della porzione dell'array che stiamo considerando (inizialmente il centro dell'interno array) e vedere se il valore che dobbiamo cercare si trova in quella posizione, e in tal caso finiamo, è minore, e quindi si troverà alla sinistra del centro, o maggiore, in tal caso si troverà alla destra; nel caso non si sia trovato ci spostiamo ad analizzare la parte destra o sinistra asseconda del risultato. Eseguiamo questa operazioni finché è consentito dal ciclo.

Note 2.1.1. *Nota che a noi non ci importa se la porzione è pari o dispari, quello che ci ritornerà esclude il resto.*

Esempio 2.3. Esempio con l'array in figura 3 cercando il valore 18.

pos	sin	dx	cen	A[cen]
-1	0	14	7	24
-1	0	6	3	5
-1	4	6	5	**18**

Iterazioni	Dimensione A
1	$n = n/2^0$
2	$n/2 = n/2^1$
3	$n/4 = n/2^2$
...	...

Table 1: Esempio di funzionamento dell'algoritmo a sinistra e numero iterazione a destra

2.1.2 Calcolo caso pessimo e migliore

Per calcolare il caso pessimo partiamo guardando la tabella sopra, notiamo che in questo algoritmo verranno eseguite $n/2^i$ operazioni, quindi il massimo possibile dipende da quanto è grande i. Per andare a trovare i basta:

$$n/2^i = 1 \qquad n = 2^i \qquad \log_2 n = \log_2 2^i \qquad i = \log_2 n \in O(\log_n)$$

Questo caso è o quando k si trova agli estremi o quando k non c'è nell'array, e quindi ritorna -1.

3 Complessità

3.1 Notazione asintotica

Quando scriviamo un algoritmo, per calcolarne il costo, bisogna fare una serie di assunzioni sulla macchina astratta su cui lavoriamo:

- L'accesso alle celle di memoria avviene in tempo costante.
- Le operazioni elementari avvengono in tempo costante:
  - Operazioni aritmetiche e logiche della ALU
  - Gli assegnamenti
  - I controlli del flusso (salti, assegnamento al registro PC)

Per calcolare il costo degli algoritmi si possono utilizzare due modelli:

1. **Word model**: tutti i dati occupano solo una cella di memoria.
2. **Bit model**: unità elementare di memoria *bit*, si usa quando le grandezze sono troppo grandi.

Esistono una serie di parametri da **analizzare** quando scriviamo una algoritmo. Essi permettono di garantire il suo corretto funzionamento e la sua ottimizzazione. Sono i seguenti:

- **Complessità**: ovvero l'analisi dell'utilizzo delle risorse:
  - tempo di esecuzione
  - spazio di memoria per i dati in ingresso e in uscita. Viene rappresentato astrattamente dal numero di celle di memoria (word model)
  - banda di comunicazione (per esempio nel caso il calcolo sia distribuito)

  Non sarà quasi mai possibile avere un programma che è sia efficiente in termini di tempo che in di spazio (*coperta corta*).
- **Correttezza**: Indica se l'algoritmo fa quello per cui è stato progettato. Si esegue in due modi:
  - dimostrazione formale la quale permette di dimostrare la correttezza risolvendo tutte le istanze del problema
  - ispezione formale nella quale si usano metodi come il **testing** o il **profiling**. Il primo prevede di provare il programma nelle situazioni critiche, il secondo analizza il tempo che la CPU impiega per elaborare una determinata parte del programma.
- **Semplicità**: Indica se l'algoritmo è facile da capire e manutenere. Un algoritmo è semplice quando usa identificatori significativi, quando è ben commentato, se usa strutture dati adeguate e se rispetta gli standard.

Definizione 3.1 (Complessità di un problema)**.** *La complessità di un problema P è la complessità del miglior algoritmo A che lo risolve.*

Per trovare la complessità del problema partiamo dal fatto che, dato un algoritmo A, la complessità di A determina un limite superiore alla complessità di P (cioè quando si verifica il caso peggiore uso A per risolvere P).
Se riusciamo a determinare un limite inferiore $g(n)$ per P, per ogni algoritmo A che risolve P ho che $A \in \Omega(g(n))$, dove $g(n)$ è il minimo numero di operazione che posso impiegare per risolvere P. Quindi possiamo dire che:

$$A \in \Theta(g(n)) \implies A \text{ ottimo}^3 \tag{1}$$

Per fare ciò bisogna anche andare a calcolare il limite inferiore del caso pessimo, e ciò è possibile tramite 3 metodi: la **dimensione dei dati**, gli **eventi contabili** e gli **alberi decisionali**.

[3] Ricorda che dire che $A \in \Theta(g(n))$ vuol dire che $A \in O(g(n))$ e $A \in \Omega(g(n))$

- **Dimensione dei dati**: Se la soluzione di un problema richiede l'esame di tutti i dati in input, allora $\Omega(n)$ è un limite inferiore. *E.g. sommare tutti gli elementi di un array.*
- **Eventi contabili**: se la soluzione di un problema richiede la ripetizione di un certo evento, allora il numero di volte che l'evento si ripete (moltiplicato per il suo costo) è un limite inferiore.
- **Alberi di decisione**: sono alberi in cui
  - ogni nodo non foglia effettua un test su un attributo
  - ogni arco uscente da un nodo è un possibile valore dell'attributo
  - ogni nodo foglia assegna una classificazione

  Si applica a problemi risolubili attraverso sequenze di decisioni che via via riducono lo spazio delle soluzioni.

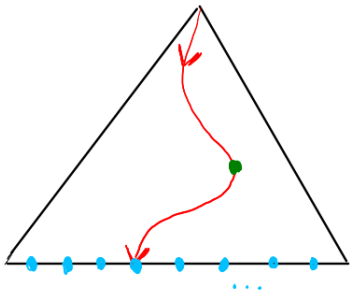


Figure 4: Albero decisionale

In figura vediamo che dalla situazione iniziale, tramite un percorso radice-foglia (ovvero un'esecuzione dell'algoritmo), otteniamo una tra le possibili soluzioni (foglie) passando per diverse decisioni (nodi interni).

Note 3.1.1. *Alcune formule importanti per gli alberi:*

- ***Foglie**: n^d*
- ***Profondità** $d \leq \log_n$ foglie (è esattamente uguale solo se l'albero è completo)*
- ***Nodi**: $n^{d+1} - 1$*

Esempio 3.1. Ricerca binaria di un elemento k in un array A di n elementi. Ogni confronto tra k e $A[cen]$ può generare 3 possibili risposte:

- $k < A[cen]$ ramo sinistro
- $k == A[cen]$ ramo centrale
- $k > A[cen]$ ramo destro

Abbiamo quindi che ogni confronto apre 3 possibili vie e dopo i confronti avremo 3^i vie. Le possibili soluzioni sono $n + 1$ (k può essere in ognuna delle n posizioni o non esserci). Avremo quindi:

$$3^i \geq n + 1 > n \implies \text{binSearch} \in \Omega(\log_n) \tag{2}$$

3.2 Big-O notation

La notazione Big-O ha molteplici scopi nella scrittura di un algoritmo.

- Serve a rappresentare la complessità relativa di un algoritmo.
- Descrive le prestazioni di un algoritmo e come queste scalano al cresce dei dati in input.
- Descrive un limite superiore al tasso di crescita di una funzione ed è il caso peggiore.

3.2.1 Limite superiore asintotico

Definizione 3.2 (Limite superiore asintotico)**.** *Il limite superiore asintotico* [4] *si definisce come:*

$$O(g(n)) = \{f(n) \mid \exists c, n_0 > 0. \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n)\} \tag{3}$$

Si scrive come $f(n) \in O(g(n))$ oppure $f(n) = O(g(n))$ e si legge $f(n)$ è nell'ordine O grande di $g(n)$.

Esempio 3.2. Esempio di calcolo del limite superiore asintotico

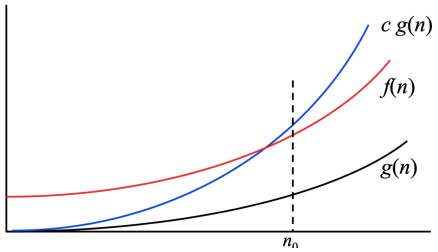


Figure 5: Limite superiore asintotico

Prendiamo due funzioni e determiniamo i punti n_0 e c per cui è soddisfatta la definizione.

$$f(n) = 3n^2 + 5 \qquad g(x) = n^2$$

Stabiliamo un $c = 4$ e $n_0 = 3$.

1. $4 \cdot g(n) = 4n^2 = 3n^2 + n^2$
2. $3n^2 + n^2 \geq 3n^2 + 9$ (per ogni $n \geq 3$)
3. $3n^2 + 9 > 3n^2 + 5 \implies 4 \cdot g(n) > f(n)$

Note 3.2.1. *Abbiamo disegnato solo il primo quadrante perché sia i dati in input che le operazioni da eseguire saranno sempre in numero positivo.*

3.2.2 Limite inferiore asintotico

Definizione 3.3 (Limite inferiore asintotico)**.** *Il limite inferiore asintotico si definisce come:*

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0. \forall n > n_0, 0 \leq c \cdot g(n) \leq f(n)\} \tag{4}$$

Si scrive come $f(n) \in \Omega(g(n))$ oppure $f(n) = \Omega(g(n))$ e si legge $f(n)$ è nell'ordine Ω grande di $g(n)$. Indica che quell'algoritmo non potrà mai fare di meglio.

Esempio 3.3. Esempio di calcolo del limite inferiore asintotico. Prendiamo due funzioni e determiniamo i punti n_0 e c per cui è soddisfatta la definizione.

$$f(n) = \frac{n^2}{2} - 7 \qquad g(x) = n^2$$

Stabiliamo un $c = \frac{1}{4}$ e $n_0 = 6$.

1. $\frac{1}{4} \cdot g(n) = \frac{n^2}{4} = \frac{n^2}{2} - \frac{n^2}{4}$
2. $\frac{n^2}{2} - \frac{n^2}{4} \leq \frac{n^2}{2} - 9$ (per ogni $n \geq 6$)
3. $\frac{n^2}{2} - 9 > \frac{n^2}{2} - 7 \implies \frac{1}{4} \cdot g(n) < f(n)$

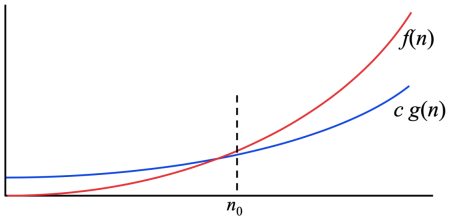


Figure 6: Limite superiore asintotico

3.2.3 Limite asintotico stretto

Definizione 3.4 (Limite asintotico stretto)**.** *Il limite asintotico stretto si definisce come:*

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0. \forall n > n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\} \tag{5}$$

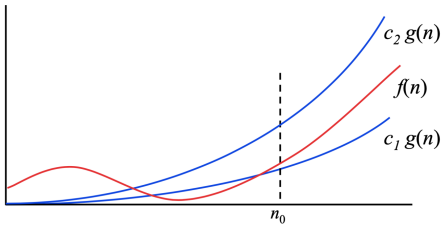


Figure 7: Limite asintotico stretto

Si scrive come $f(n) \in \Theta(g(n))$ oppure $f(n) = \Theta(g(n))$ e si legge $f(n)$ è nell'ordine Θ grande di $g(n)$.

Dalla definizione deriva che:

$$f(n) \in \Theta(g(n)) \iff f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n)) \tag{6}$$

[4] *Asintotico* indica che la definizione deve essere valida solo da un certo punto in poi scelto arbitrariamente.

3.2.4 Teoremi sulla notazione asintotica

Teorema 3.1. *Per ogni $f(n)$ e $g(n)$ vale che:*

1. $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$
2. *Se* $f_1(n) = O(f_2(n)) \wedge f_2(n) = O(f_3(n)) \implies f_1(n) = O(f_3(n))$
3. *Se* $f_1(n) = \Omega(f_2(n)) \wedge f_2(n) = \Omega(f_3(n)) \implies f_1(n) = \Omega(f_3(n))$
4. *Se* $f_1(n) = \Theta(f_2(n)) \wedge f_2(n) = \Theta(f_3(n)) \implies f_1(n) = \Theta(f_3(n))$
5. *Se* $f_1(n) = O(g_1(n)) \wedge f_2(n) = O(g_2(n)) \implies O(f_1(n) + f_2(n)) = O(max\{g_1(n), g_2(n)\})$
6. *Se $f(n)$ è un polinomio di grado $d \implies f(n) = \Theta(n^d)$*

3.2.5 Limite superiore asintotico non stretto

Definizione 3.5 (Limite superiore asintotico non stretto)**.** *Il limite superiore asintotico non stretto si definisce come:*

$$o(g(n)) = \{f(n) \mid \forall c \exists n_0 > 0. \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n)\} \tag{7}$$

Si scrive come $f(n) \in o(g(n))$ oppure $f(n) = o(g(n))$ e si legge $f(n)$ è nell'ordine o piccolo di $g(n)$. $f(n)$ è limitata superiormente da $g(n)$, ma non la raggiunge mai.

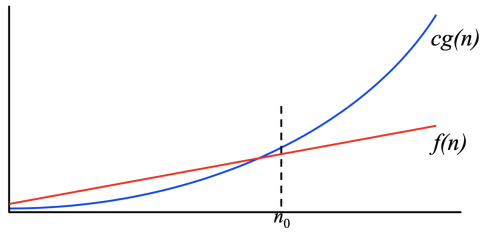


Figure 8: Limite superiore non stretto

E immediato dalla definizione che:

$$o(g(n)) \implies O(g(n))$$

Non vale il contrario:

$$2n^2 \in O(n^2) \wedge 2n^2 \notin o(n^2)$$

Definizione alternativa:

$$f(n) \in o(g(n)) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty$$

3.2.6 Limite inferiore asintotico non stretto

Definizione 3.6 (Limite inferiore asintotico non stretto)**.** *Il limite inferiore asintotico non stretto si definisce come:*

$$\omega(g(n)) = \{f(n) \mid \forall c \, \exists n_0 > 0. \forall n > n_0, 0 \leq c \cdot g(n) \leq f(n)\}$$

Si scrive come $f(n) \in \omega(g(n))$ oppure $f(n) = \omega(g(n))$ e si legge $f(n)$ è nell'ordine ω piccolo di $g(n)$. $f(n)$ è limitata inferiormente da $g(n)$, ma non la raggiunge mai.

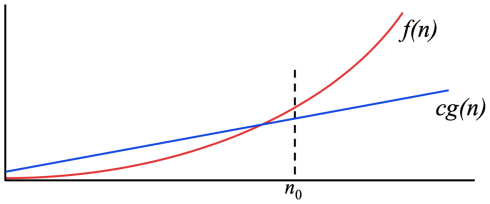


Figure 9: Limite asintotico stretto

E immediato dalla definizione che:

$$\omega(g(n)) \implies \Omega(g(n))$$

Non vale il viceversa:

$$\tfrac{1}{5}n^2 \in \Omega(n^2) \wedge \tfrac{1}{2}n^2 \notin \omega(n^2)$$

Definizione alternativa:

$$f(n) \in \omega(g(n)) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

3.3 Ordinamento

3.3.1 Insertion sort

Proprietà: al termine del passo j-esimo dell'algoritmo l'elemento j-esimo viene in inserito al posto giusto e i primi $j+1$ elementi sono ordinati.

```
1   insertionSort(A) =
2   var j:Int = 0;
3   var i:Int = 0;      Θ(1)
4   var k:int = 0;
5   for (j=1; j<n; j++) {    n − 1 volte
6     k = A[j];
7     i = j-1;      Θ(1) n − 1 volte
8     while(i >= 0 && A[i]>k) {
9       A[i+1] = A[i];        Θ(1) Σ_{j=1}^{n−1} (t_j − 1) volte
10      i=i-1;
11    }
12    A[i+1] = k;    Θ(1) n − 1 volte
13  }
```

Listing 3: Algoritmo insertion sort

0	1	2	3	4	5	j	i	k	while
5	2	4	6	1	3	0	0	0	no
5	2	4	6	1	3	1	0	2	si
5	5	4	6	1	3	1	-1	2	no
2	5	4	6	1	3	1	-1	2	no
2	5	4	6	1	3	2	1	4	si
2	5	5	6	1	3	2	0	4	no
2	4	5	6	1	3	2	0	4	no
2	4	5	6	1	3	3	2	6	no
2	4	5	6	1	3	3	2	6	no

Table 2: Esempio di esecuzione

Complessità:

$$\sum_{j=1}^{n-1} t_j \tag{8}$$

- Caso pessimo: l'array è ordinato decrescente e quindi ogni volta devo scalare l'elemento fino alla prima posizione. Abbiamo che $t_j = j$ e $\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$, quindi $O(n^2)$
- Caso migliore: l'array è ordinato crescente e quindi per ogni iterazione non entro nel while perché la condizione è falsa. Abbiamo $t_j = 1$ e $\sum_{j=1}^{n-1} j = n-1$, quindi $O(n)$
- Caso medio: come il caso pessimo $O(n^2)$

Correttezza:

- dimostro l'**invariante di ciclo** per assicurarmi che la mia proprietà venga mantenuta durante tutta l'esecuzione. Lo faccio tramite *induzione*:
  - Caso base: per $j = 1$
  - Hp induttiva: per $j = n'$
  - Passo induttivo: dimostro che vale anche per $j = n' + 1$
- verifico la **terminazione**: il *for* è eseguito esattamente $n-1$ volte e il *while* al più $j-1$ volte, quindi tutte le iterazioni sono finite e l'algoritmo termina.

Memoria impiegata: ordina in loco quindi non usa memoria aggiuntiva.

3.3.2 Selection sort

Proprietà: al termine del passo j-esimo dell'algoritmo i primi $j+1$ elementi di A sono ordinati e contengono i $j+1$ elementi più piccoli di A.

```
1   insertionSort(A) =
2   var j:Int = 0;
3   var i:Int = 0;      Θ(1)
4   var min:int = 0;
5   for (i=0; i<n-1; i++) {    n − 1 volte
6     min = i;      Θ(1) n − 1 volte
7     for(j=i+1; j<n; j++) {
8       if A[j] < A[min] {min = j};      Θ(1) ∑_{j=1}^{n−1} (t_j − 1) volte
9     }
10    swap(A[i],A[min]);     Θ(1) n − 1 volte
11  }
```

Listing 4: Algoritmo selection sort

0	1	2	3	4	5	j	i	min
5	2	4	6	1	3	0	0	0
1	2	4	6	5	3	1	0	4
1	2	4	6	5	3	2	1	1
1	2	3	6	5	4	3	2	5
1	2	3	4	5	6	4	3	3
1	2	3	4	5	6	5	4	4

Table 3: Esempio di esecuzione

Complessità

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} \in O(n^2) \tag{9}$$

- Caso pessimo: $O(n^2)$
- Caso migliore: $O(n^2)$
- Caso medio: $O(n^2)$

Correttezza:

- dimostro l'**invariante di ciclo** per assicurarmi che la mia proprietà venga mantenuta durante tutta l'esecuzione. Sempre tramite induzione.
- verifico la **terminazione** in maniera analoga all'insertion sort.

Memoria impiegata: ordina in loco quindi non usa memoria aggiuntiva.

4 Funzioni

Definizione 4.1 (Principio di astrazione)**.** *Ridurre la duplicazione di informazione nei programmi utilizzando* ***funzioni*** *definite dal programmatore e quelle disponibili nelle librerie standard. Facilita la* manutenzione *e* comprensione *del codice.*

```
1   var trovatoA : Bool = false;
2   var trovatoB : Bool = false;
3   i = 1;
4   while (i<=n && !trovatoA) {
5     if (A[i] == k) trovatoA = true;
6     else i = i + 1;
7   }
```

```
8   while (i<=n && !trovatoB) {        // Uguale al ciclo precedente se non per l'array
9     if (B[i] == k) trovatoB = true;
10    else i = i + 1;
11  }
12  if (trovatoA && trovatoB) {
13    C
14  }
```

Listing 5: Esempio di codice astraibile

Possiamo semplificarlo tramite la creazione della seguente funzione:

```
1   var trovatoA : Bool = false;
2   var trovatoB : Bool = false;
3   func seqSearch(array:[Int], k:Int) -> Bool {
4     var trovato = false;
5     var i = 1;
6     while (i<=n && !trovato) {
7       if (array[i] == k) {trovato = true}
8       else {i = i+1};
9     }
10    return trovato;
11  }
12  if (trovatoA && trovatoB) {
13    C
14  }
```

Listing 6: Esempio di funzione

4.0.1 Anatomia di una funzione

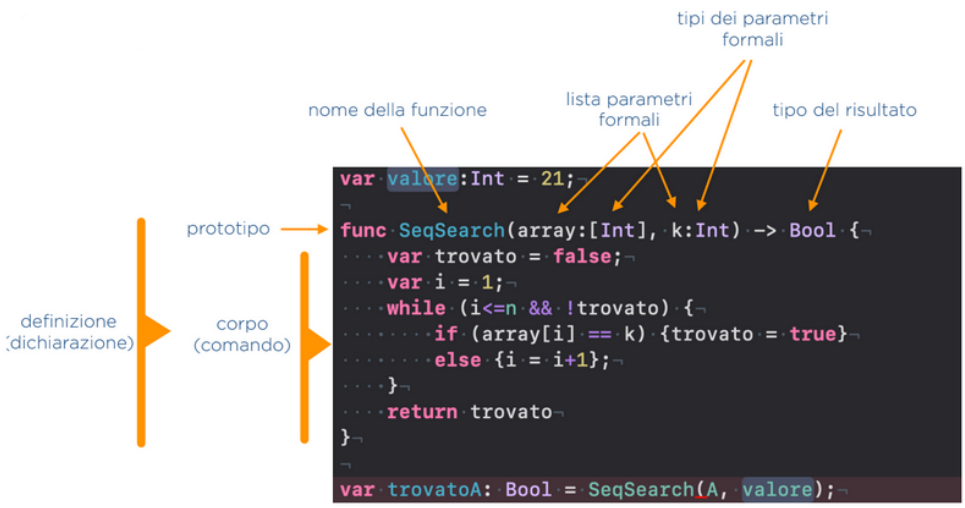


Figure 10

Il **compilatore** dovrà poi eseguire le seguenti verifiche:

- il **numero** di parametri *attuali* deve coincidere con quello dei parametri *formali*
- i **nomi** dei parametri *formali* devono essere tutti distinti
- i **tipi** degli *attuali* e dei *formali* nella stessa posizione devono essere uguali
- non ci devono essere **variabili libere** nel corpo della funzione che non possono essere legate
- deve esserci un **return statement** nel corpo della funzione
- il **tipo** dell'espressione nel *return statement* deve coincidere con quello della dichiarazione

Definizione 4.2 (Principio di corrispondenza)**.** *Parti di programma che hanno effetti simili devono avere una sintassi simile. Facilità di apprendimento del linguaggio e di interpretazione dei programmi.*

Definizione 4.3 (Scoping statico)**.** *Le **variabili libere** nel corpo delle funzioni vengono legate a tempo di compilazione costruendo le **chiusure**.*

Definizione 4.4 (Chiusura)**.** *Ciò che viene registrato nell'**ambiente dinamico** al momento dell'elaborazione della **dichiarazione** di funzione, associando al nome della funzione tutto ciò che sta alla sinistra.*

La dichiarazione di funzione genera nell'**ambiente dinamico** un legame tra il **nome della funzione** e una'**astrazione** che contiene tutte le informazioni necessarie ad eseguire la chiamata della funzione.

$$[(nomeFunzione), \lambda(parametriformali).[(\textbf{variabili libere})], C] \tag{10}$$

```
1   var COVID19:Bool = true;
2
3   var mioCosto:Double = 100;
4   var aliquota:Double = 21;
5
6   // Chiusura
7   // [(calcolaIVA, λ(let costo) . {[(aliquota, 12)]; return costo*aliquota/100})]
8
9   func calcolaIVA(let costo:Double) -> Double {
10    return costo*aliquota/100;
11  }
12
13  if (COVID19) {
14    var aliquota:Double = 23;
15    print(calcolaIVA(mioCosto));
16  } else {
17    print(calcolaIVA(mioCosto));
18  }
```

Listing 7: Esempio di scoping statico

4.0.2 Dinamico

Le **variabili libere** vengono legate a tempo di **esecuzione** quando vengono utilizzate. Nella chiusura della funzione registro solo il suo corpo al momento della dichiarazione.

```
1   var COVID19:Bool = true;
2
3   var mioCosto:Double = 100;
4   var aliquota:Double = 21;
5
6   // Chiusura
7   // [(calcolaIVA, λ(let costo) . {return costo*aliquota/100})]
8
9   func calcolaIVA(let costo:Double) -> Double {
10    return costo*aliquota/100;
11  }
12
13  if (COVID19) {
14    var aliquota:Double = 23;
15    print(calcolaIVA(mioCosto));
16  } else {
17    print(calcolaIVA(mioCosto));
18  }
```

Listing 8: Esempio di scoping dinamico

IMPORTANTE: la semantica statica ha senso solamente in presenza di uno **scoping statico**, il quale a differenza di quello dinamico ci garantisce la mancanza di errori. Di conseguenza in presenza dello **scoping dinamico** MAIverificare la correttezza sintattica.

4.1 Passaggio dei parametri

4.1.1 Per valore

Viene fatta una copia degli identificatori passati come parametri attuali tra le variabili locali del corpo della funzione. Non modifico il valore esterno di questi identificatori.

4.1.2 Per indirizzo

5 Gestione della memoria

5.1 Record di attivazione

Definizione 5.1 (Supporto a tempo di esecuzione)**.** *È l'insieme di strutture dati e funzioni necessarie all'esecuzione dei programmi e viene aggiunto al codice eseguibile dal compilatore.*

Definizione 5.2 (Dynamic chain o call chain)**.** *Rappresenta la sequenza di chiamate e serve a garantire il corretto ordine di esecuzione. Implementa il passaggio del controllo in caso di chiamate annidate e tiene traccia dell'ordine.*

Definizione 5.3 (Static chain)**.** *Implementa lo scoping statico e garantisce che i nomi siano referenziati rispettando la visibilità di variabili e funzioni.*

Definizione 5.4 (Activation record o stack frame)**.** *Contiene tutte le informazioni necessarie all'esecuzione del blocco o della funzione.*

Puntatore catena dinamica	Indirizzo del record di attivazione della funzione chiamante
Puntatore catena statica	Indirizzo del prossimo record di attivazione dove risolvere i nomi non presenti nel blocco corrente (implementazione dello scoping statico)
Indirizzo di ritorno	Indirizzo dell'istruzione da eseguire al termine della funzione/blocco corrente
Indirizzo risultato	Indirizzo nel record di attivazione del chiamante per memorizzare il risultato
Parametri	Spazio riservato alla associazione parametri formali - parametri attuali
Variabili locali	Spazio riservato alla allocazione delle variabili locali al blocco
Risultati temporanei	Spazio riservato alla allocazione delle variabili temporanee generate dal compilatore

5.2 Divisione della memoria

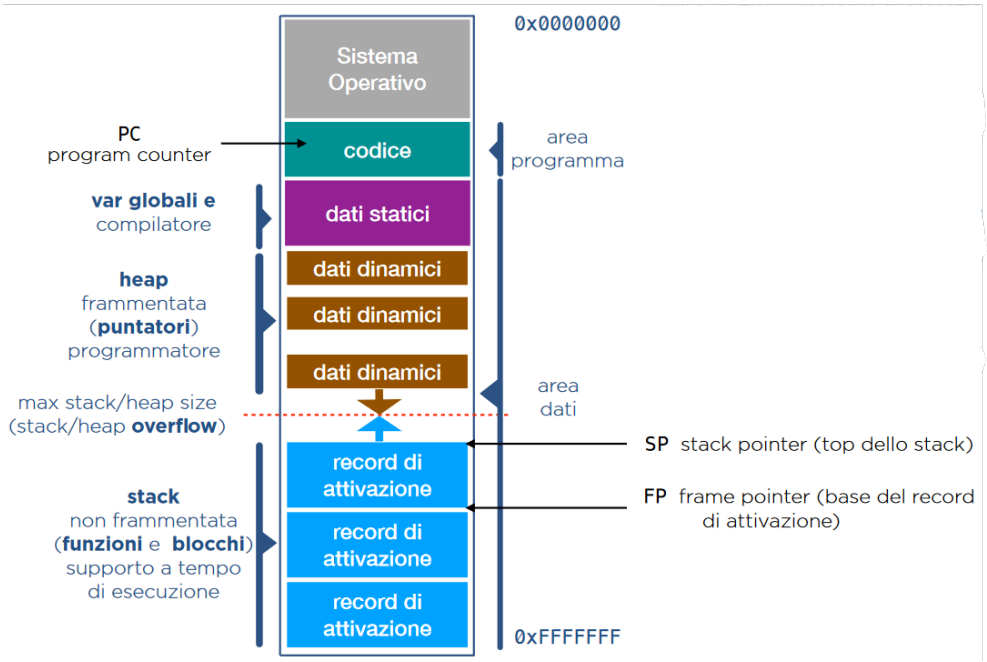


Figure 11: Gestione della memoria di un programma

Note 5.2.1. *Partiamo dal presupposto che un* ***blocco*** *sia considerato come una funzione senza parametri.*

Note 5.2.2. *Lo* ***stack*** *funziona tramite operazioni di* ***push*** *(inserimento di un elemento in cima)e* ***pop*** *(rimozione dell'elemento in cima). Può lavorare in due modi:*

- ***LIFO*** *(*Last In First Out*): l'ultimo elemento inserito è il primo ad essere rimosso*
- ***FIFO*** *(*First In First Out*): il primo elemento inserito è il primo ad essere rimosso*

5.3 Tipi di ricorsione

Definizione 5.5 (Non lineare)**.** *Viene eseguita più di una* ***chiamata ricorsiva*** *nel blocco.*
Un caso particolare è quando la funzione viene passata come parametro formale della sua stessa definizione e si dice ***annidata****.*

Definizione 5.6 (Mutua)**.** *Quando due o più funzioni sono definite ciascuna in termini dell'altra.*

Definizione 5.7 (In coda)**.** *La chiamata ricorsiva è l'unica operazione effettuata dalla funzione prima di restituire il controllo alla chiamata. La chiamata in questo caso si definisce* ***terminale****.*
Questa modalità consente di risparmiare spazio di memoria per la gestione dello stack di esecuzione in quando viene gestito come se fosse iterativo e viene creato un solo record di attivazione in più per la gestione dell'indirizzo di ritorno.

6 Ricorsione

Definizione 6.1 (Ricorsione)**.** *A tempo di* ***compilazione****: una funzione usa il suo nome (chiama se stessa) nel suo corpo. A tempo di* ***esecuzione****: chiamate annidate della* ***stessa*** *funzione*

Una funzione ricorsiva è chiamata per risolvere un problema scomposto in:

- **Caso base**: la funzione restituisce un valore
- **Passo ricorsivo**: la funzione viene chiamata su un problema analogo a quello iniziale ma di dimensioni minori, avvicinandosi al *caso base*

Quando si arriva al caso base viene effettuata una sequenza inversa di return statement, combinando i risultati parziali in quello finale.

Esempio 6.1 (Fattoriale)**.** Il fattoriale di un intero non negativo n è il prodotto degli interi positivi $<= n$ escluso lo 0. Si indica con $n!$ e si impone per definizione $0! = 1$.

$$n! = \prod_{i=1}^{n} i = n * (n-1) * \ldots * 1 \tag{11}$$

oppure definita in maniera ricorsiva:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases} \tag{12}$$

In maniera programmatica possiamo scriverlo come:

```
1     func F(var n: Int) -> Int {
2       if (n-1) {
3         return 1
4       } else {
5         return n * F(n-1)
6       }
7     }
```

Listing 9: Fattoriale con ricorsione

6.1 Ricorsione e iterazione

	Ricorsione	**Iterazione**
Controllo di terminazione	Condizione di ricorsione	Condizione di controllo nel loop
Ripetizioni	Chiamate ricorsive della funzione	Esecuzione ripetuta del corpo dell'iterazione
Convergenza alla terminazione	I passi ricorsivi riducono il problema al caso base	Il contatore si avvicina al valore di termine
Ripetizione infinita	Il passo ricorsivo non riduce il problema e non si avvicina al caso base	La condizione di controllo non è mai falsa

Nella *ricorsione*, al contrario dell'*iterazione*, ogni chiamata alla funzione genera un nuovo record di attivazione contenente una nuova copia delle variabili e consumando lo stack di esecuzione. Questo può generare **overhead**.
In generale ogni problema *ricorsivo* può essere anche scritto *iterativamente*. È consigliato scriverlo ricorsivamente quando ciò facilita la lettura del problema stesso.

7 Condizioni

7.1 Condizioni su array

Dato un array **a** di dimensione N, voglio verificare se la proprietà P vale per tutti gli elementi dell'array.

$$\forall i \in [0, N).\mathcal{P}(a[i]) \tag{13}$$

Esempio 7.1. Verifico che tutti gli elementi dell'array siano dispari.

$$\forall i \in [0, N).a[i] \tag{14}$$

```
1   int check_array_dispari(int a[], size_t dim) {
2     int indice = 0;
3     while (indice < dim && a[indice]%2 == 1){
4       indice++;
5     }
6     if (indice == dim) {
7       return 1;
8     } else {
9       return 0;
10    }
11  }
```

Listing 10: Verifica di proprietà su tutti gli elementi mathescape

Blocco lo scorrimento dell'array quando la proprietà **NON** viene soddisfatta almeno una volta.

Se invece voglio verificare che la proprietà P valga per almeno un elemento:

$$\exists i \in [0, N).\mathcal{P}(a[i]) \tag{15}$$

Esempio 7.2. Verifico che almeno un elemento dell'array è uguale a 26.

$$\exists i \in [0, N).a[i] == 26 \tag{16}$$

```
1   int esiste_in_array(int a[], size_t dim, in n) {
2     size_t indice = 0;
3     _Bool trovato = 0;
4     while (indice < dim && !trovato){
5       if(a[indice] == n) {
6         trovato = 1;
7       }
8       indice++;
9     }
10    return trovato;
11  }
```

Listing 11: Verifica di proprietà su almeno un elemento

Blocco lo scorrimento dell'array nel momento in cui trovo un elemento che soddisfa la proprietà, utilizzando un *flag*.

7.2 Condizioni su matrici

Una **matrice** è un array di array. Può essere *multidimensionale* $N \times M$ e voglio verificare se tutti i suoi elementi oppure solo uno di essi verificano una proprietà P.

$$\forall i \in [0, N), \forall j \in [0, M).\mathcal{P}(a[i, j]) \tag{17}$$

$$\exists i \in [0, N), \exists j \in [0, M).\mathcal{P}(a[i, j]) \tag{18}$$

Definizione 7.1 (Matrice quadrata)**.** *Una matrice è* ***quadrata*** *se a lo stesso numero di righe e di colonne. In questo caso per scorrerla si può usare un solo indice:*

$$\exists i, j \in [0, N).\mathcal{P}(a[i, j]) \tag{19}$$

Esempio 7.3. Verifico se tutti gli elementi della matrice sono positivi.

$$\forall i \in [0, N), \forall j \in [0, M).a[i, j] > 0 \tag{20}$$

```
1   int check_matrice_pos(int a[][COL], size_t dim) {
2     size_t row, col;
3     row = col = 0;
4     while (row < dim && a[row][col] > 0) {
5       col = 0;
6       while (col < COL && a[row][col] > 0) {
7         col++;
8       }
9       if (col == COL) {
10        row++;
11      }
12    }
13    if (row == dim && col == COL) {
14      return 1;
15    }
16    else {
17      return 0;
18    }
19  }
```

Listing 12: Verifica di proprietà su tutti gli elementi della matrice

Definizione 7.2 (Matrice simmetrica)**.** *Una matrice è* ***simmetrica*** *se è quadrata e se le posizioni simmetriche rispetto alla diagonale principale contengono gli stessi elementi.*

Definizione 7.3 (Matrice triangolare)**.** *Una matrice è* ***triangolare*** *superiore o inferiore se le posizioni rispettivamente sopra o sotto la diagonale contengono tutti 0.*

Definizione 7.4 (Matrice tridiagonale)**.** *Una matrice* ***tridiagonale*** *può avere elementi non nulli solo sulla diagonale principale e la sua diagonale superiore ed inferiore.*

7.3 Contare elementi che verificano una proprietà

Dato un array **a** di dimensione N per contare tutti gli elementi che verificano una proprietà P:

$$\#\{i | i \in [0, N-1] \wedge \mathcal{P}(a[i])\} \tag{21}$$

Data invece una matrice **a** di dimensione $N \times M$:

$$\#\{(i, j) | i \in [0, N-1] \wedge j \in [0, M-1]\} \tag{22}$$

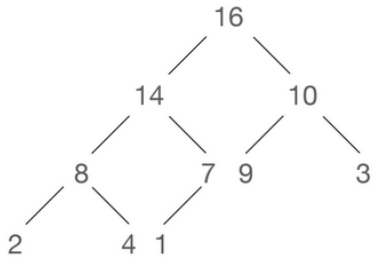
8 Heap

Definizione 8.1 (Heap binario)**.** *Un* ***albero*** *quasi completo.*

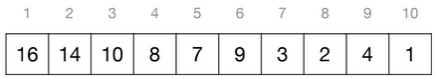
Definizione 8.2 (Albero binario completo)**.** *Un albero dove ogni nodo è foglia oppure ha due figli.*

Definizione 8.3 (Albero binario quasi completo)**.** *Se h è l'altezza dell'albero, tutte le foglie hanno profondità h oppure $h-1$. Tutti i nodi hanno 2 figli eccetto al più 1. Il nodo con un solo figlio, se esiste:*

- *ha profondità $h-1$*
- *tutti i nodi alla sua destra sono* ***foglie***
- *e il suo unico figlio è un figlio* ***sinistro***



(a) Albero binario quasi completo



(b) Heap

Di seguito alcune formule utili:

- parent(i) $= \lfloor i/2 \rfloor$
- left(i) $= 2i$
- right(i) $= 2i+1$

8.1 Max e min heap

Dato un heap, se gli elementi di ogni sotto-albero sono più piccoli della radice del sotto-albero, allora abbiamo un **max-heap** e il massimo valore sarà memorizzato sempre nella radice.

$$\forall i \neq 1, A[parent(i)] \geq A[i] \tag{23}$$

Analogamente per il **min-heap** il minimo valore sarà nella radice.

$$\forall i \neq 1, A[parent(i)] \leq A[i] \tag{24}$$

8.2 Proprietà

- **Proprietà 1**: un *heap* di n elementi ha altezza $\theta(\log n)$, precisamente $\lfloor \log n \rfloor$
- **Proprietà 2**: un *heap* di n elementi contiene $\lceil n/2 \rceil$ foglie
- **Proprietà 3**: un *heap* di n elementi ha al più $\lceil n/2^{h+1} \rceil$ nodi di altezza h, esattamente $\lceil n/2^{h+1} \rceil$ se è un albero *bilanciato completo*