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 conano i byte a partire dalla posizione 0x0000000. Gli indirizzi di memoria possono esere memorizazti in varibili come ogni altro intero.

Definizione 1.1 (Puntatori)**.** *Andiamo così a definire le variabile che memorizzano indirizzi di memoria come* ***puntatori****, per distinguerli anche dagli altri tipi di variabili.*

1.2 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 dereferenziazzione**. L'operatore di indirezione è unario e restituisce il valore dell'oggetto a cui punta l'operanda.

Note 1.2.1. *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 un una sola varibile

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.

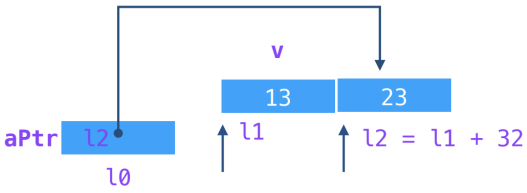


Figure 1: Operazioni algebriche su puntatori

1.3 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.

1.4 Notazione asintotica

Quando andiamo a scrivere un algoritmo, per calcolare il costo di tale algoritmo, bisogna andare a fare una serie di assunzioni che appunto facciamo basandoci sul fatto che stiamo lavorando su una macchina astratta, quindi con caratteristiche fissate.

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

Noi per andare a calcolare il costo degli algoritmi si possono utilizzare due modelli di calcolo:

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

Esistono una serie di parametri da analizzare quando scriviamo una algoritmo, essi infatti permettono di garantire il suo corretto funzionamento, e o la sua ottimizzazione, essi sono i seguenti.

- **Complessità:** Può essere sintetizzato con l'analisi dell'utilizzo delle risorse che sarebbero: tempo di esecuzione, spazio di memoria, banda di comunicazione.

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

- **Correttezza:** Indica se l'algoritmo fa quello per cui è stato progettato, e si fa tramite una dimostrazione formale il quale permette di dimostrare la correttezza andando a risolvere tutte le istanze del problema, o una ispezione formale, quest'ultima spesso ha che fare con la struttura de programma e si usano metodi come il testing o il profalig, il primo prevede di provare il programma nelle situazioni critiche, il secondo analizza il tempo che la CPU elabora una determinata parte del programma.
- **Semplicità:** Questo paramento indica se l'algoritmo è facile da capire e manutenere e si fa tramite identificatori significativi, algoritmo ben commentato, strutture dati adeguate, rispetto degli standard.

Definizione 1.3 (Complessità di un problema)**.** *La complessità di un problema P è la complessità del miglior algoritmo A che lo risolve.*

Per andare a trovare la complessità del problema partiamo dal fatto che, preso un algortimo 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 ogmo 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:

$$A \in \Theta(g(n)) \implies \text{A ottimo } [3]$$

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

- **Dimensioni dei dati:** Se la soluzione di un problema richiede l'esame di tutti i dati in input, allora $\Omega(n)$ è un limite inferiore.

Esempio 1.3. 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.

Esempio 1.4. Se voglio costruire un array i cui elementi sono la radice quadrata degli elementi di un array dato, dovrò fare un ciclo che scorre l'array dato e applica la radice quadrata. Gli eventi contabili sono quindi il numero di iterazioni.

1.5 Big-O notation

La notazione Big O ha molteplici scopi della scritture 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 limte superiore al tasso di crescita di na funzione ed il caso peggiore.

1.5.1 Limite superiore asintotico

Definizione 1.4 (Limite superiore asintotico)**.** *Il limite superiore asintotico si definisce come:*

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

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 1.5. Esempio di calcolo del limite superiore asintotico

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

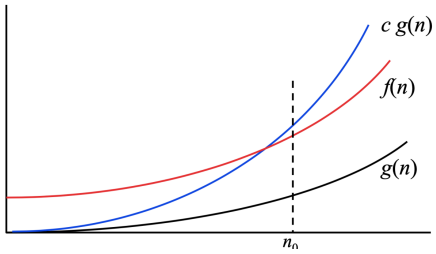


Figure 2: Limite superiore asintotico

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. $3n^2 + 9 > 3n^2 + 5 \implies 4 \cdot g(n) > f(n)$

1.5.2 Limite inferiore asintotico

Definizione 1.5 (Limite inferiore asintotico)**.** *Il limite inferiore asintotico si definisce come:*

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

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 $g(n)$.

Esempio 1.6. 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 $c = 4$ e $n_0 = 3$.

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. $\frac{n^2}{2} - 9 > \frac{n^2}{2} - 7 \implies \frac{1}{4} \cdot g(n) < f(n)$

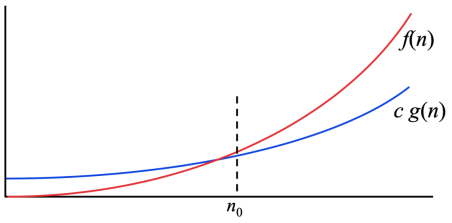


Figure 3: Limite superiore asintotico

1.5.3 Limite asintotico stretto

Definizione 1.6 (Limite inferiore asintotico)**.** *Il limite asintotico stretto si definisce come:*

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

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:
$f(n) \in \Theta(g(n)) \iff f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n))$

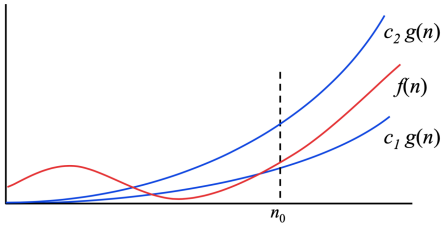


Figure 4: Limite asintotico stretto

1.5.4 Teoremi sulla notazione asintotica

Teorema 1.1. *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 grafo d* $\implies f(n) = \Theta(n^d)$

1.5.5 Limite superiore asintotico non stretto

Definizione 1.7 (Limite superiore asintotico non stretto)**.** *Il limite superiore asintotico non strettosi definisce come:*

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

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.

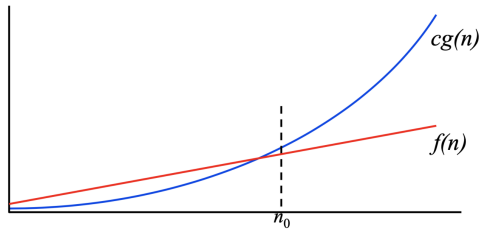


Figure 5: Limite superiore non stretto

E immediato dalla definizione che:

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

Non vale il viceversa:

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

Definizione alternativa:

$$f(n) \in o(g(n)) \Longleftrightarrow \lim_{n\to\infty} \tfrac{g(n)}{f(n)} = \infty$$

1.5.6 Limite inferiore asintotico non stretto

Definizione 1.8 (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.

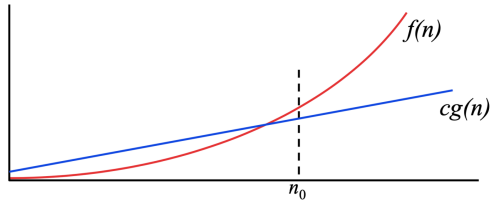


Figure 6: Limite asintotico stretto

E immediato dalla definizione che:

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

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:

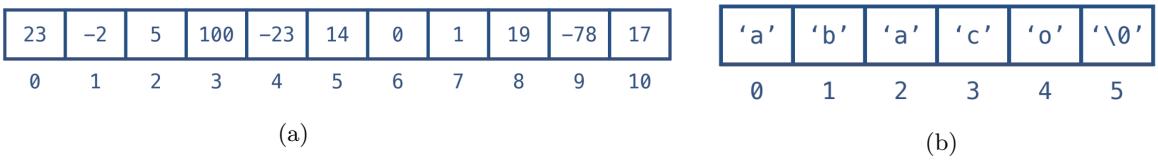
$$f(n) \in \omega(g(n)) \Longleftrightarrow \lim_{n\to\infty} \tfrac{g(n)}{f(n)} = 0$$

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 7: 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.

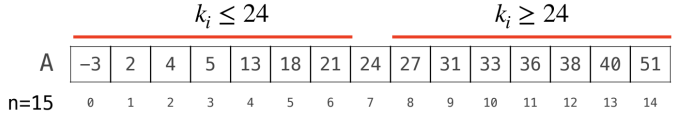


Figure 8: Array A in BinarySearch

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 8 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.