Programmazione ed Algoritmi

Realizzato da: Giuntoni Matteo e Ghirardini Filippo

A.A 2022-2023

Contents

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 usati 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.

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

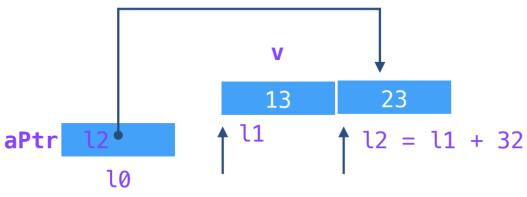


Figure 1: Operazioni algebriche su puntatori

[1] Unario vuol dire che agisce su una sola variabile
[2] Questo vuol dire che ogni vaore è salvato con 32 bit, quindi ogni 32 ci sarà un nuovo valore archiviato

a sommare 32 a "aPtr" succederà che ci sposteremo di 32 posti nella memoria raggiungendo l2, quindi "aPtr" = l2.

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**.

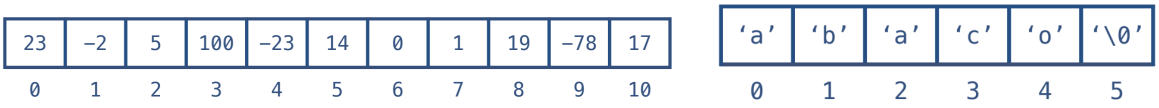
Una struttura dati è inoltre caratterizzata dalle operazioni elementari disponibili per inserire, reperire e modificare i dati che memorizza.

2 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:

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

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.

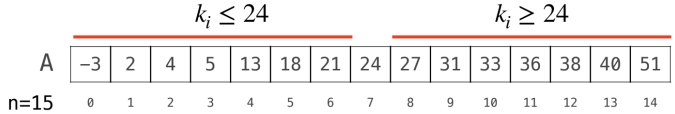


Figure 3: Array A in BinarySearch

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 Funzioni

Definizione 3.1 (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 3: 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 4: Esempio di funzione

3.1 Anatomia di una funzione

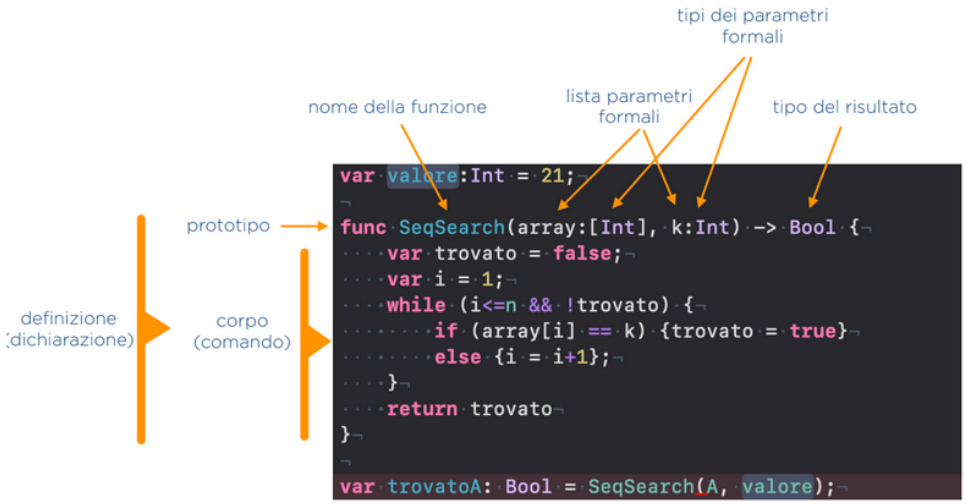


Figure 4

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

3.1.1 Ambiente statico

Definizione 3.2 (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 3.3 (Scoping statico)**.** *Le **variabili libere** nel corpo delle funzioni vengono legate a tempo di compilazione costruendo le **chiusure**.*

Definizione 3.4 (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.

$$[(nomeFunzione), \lambda(parametriformali).[(\textbf{variabili libere})], C] \tag{1}$$

```
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 5: Esempio di scoping statico

3.1.2 Ambiente 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 6: 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.

3.2 Passaggio dei parametri

3.2.1 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.

3.2.2 Per indirizzo

4 Gestione della memoria

4.1 Record di attivazione

Definizione 4.1 (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 4.2 (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 4.3 (Static chain)**.** *Implementa lo scoping statico e garantisce che i nomi siano referenziati rispettando la visibilità di variabili e funzioni.*

Definizione 4.4 (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

4.2 Divisione della memoria

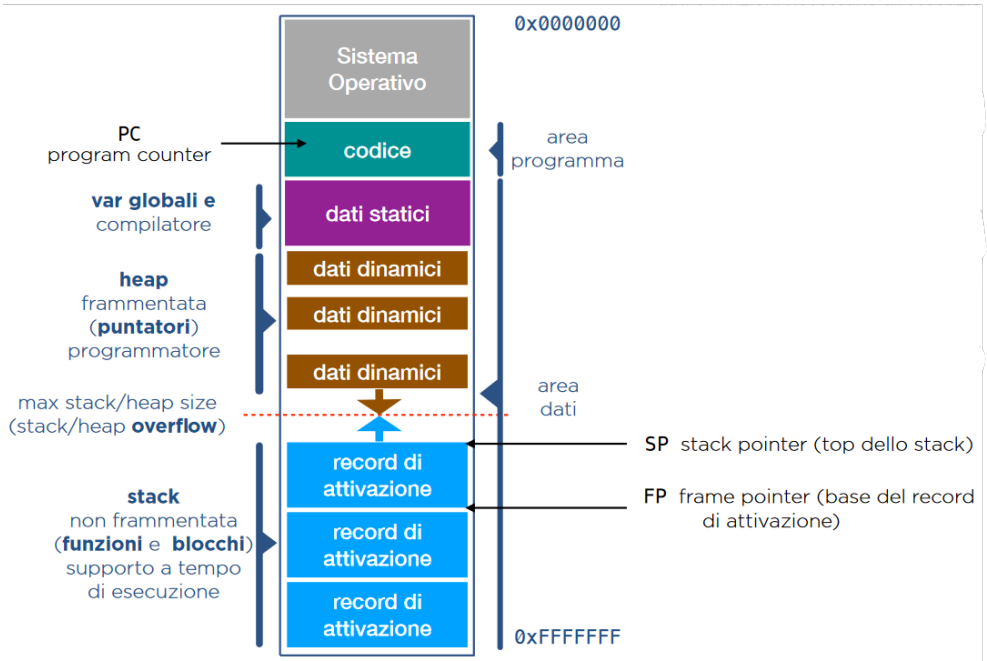


Figure 5: Gestione della memoria di un programma

Note 4.2.1. *Partiamo dal presupposto che un* ***blocco*** *sia considerato come una funzione senza parametri.*

Note 4.2.2. *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*

4.3 Tipi di ricorsione

Definizione 4.5 (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 4.6 (Mutua)**.** *Quando due o più funzioni sono definite ciascuna in termini dell'altra.*

Definizione 4.7 (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.

5 Ricorsione

Definizione 5.1 (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 5.1 (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 $0! = 1$.

$$n! = \prod_{i=1}^{n} i = n * (n-1) * \ldots * 1 \tag{2}$$

oppure definita in maniera ricorsiva:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases} \tag{3}$$

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 7: Fattoriale con ricorsione

5.1 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.

6 Condizioni

6.1 Condizioni su array

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

$$\forall i \in [0, N).\mathcal{P}(a[i]) \tag{4}$$

Esempio 6.1. Verifico che tutti gli elementi dell'array siano dispari.

$$\forall i \in [0, N).a[i] \tag{5}$$

```
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 8: 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:

$$\exists i \in [0, N).\mathcal{P}(a[i]) \tag{6}$$

Esempio 6.2. Verifico che almeno un elemento dell'array è uguale a 26.

$$\exists i \in [0, N).a[i] == 26 \tag{7}$$

```
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 9: 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*.

6.2 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.

$$\forall i \in [0, N), \forall j \in [0, M).\mathcal{P}(a[i, j]) \tag{8}$$

$$\exists i \in [0, N), \exists j \in [0, M).\mathcal{P}(a[i, j]) \tag{9}$$

Definizione 6.1 (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:*

$$\exists i, j \in [0, N).\mathcal{P}(a[i, j]) \tag{10}$$

Esempio 6.3. Verifico se tutti gli elementi della matrice sono positivi.

$$\forall i \in [0, N), \forall j \in [0, M).a[i, j] > 0 \tag{11}$$

```
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 10: Verifica di proprietà su tutti gli elementi della matrice

Definizione 6.2 (Matrice simmetrica)**.** *Una matrice è* ***simmetrica*** *se è quadrata e se le posizioni simmetriche rispetto alla diagonale principale contengono gli stessi elementi.*

Definizione 6.3 (Matrice triangolare)**.** *Una matrice è* ***triangolare*** *superiore o inferiore se le posizioni rispettivamente sopra o sotto la diagonale contengono tutti 0.*

Definizione 6.4 (Matrice tridiagonale)**.** *Una matrice* ***tridiagonale*** *può avere elementi non nulli solo sulla diagonale principale e la sua diagonale superiore ed inferiore.*

6.3 Contare elementi che verificano una proprietà

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

$$\#\{i | i \in [0, N-1] \land \mathcal{P}(a[i])\} \tag{12}$$

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

$$\#\{(i, j) | i \in [0, N-1] \land j \in [0, M-1]\} \tag{13}$$

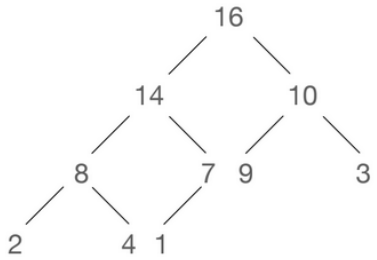
7 Heap

Definizione 7.1 (Heap binario)**.** *Un* ***albero*** *quasi completo.*

Definizione 7.2 (Albero binario completo)**.** *Un albero dove ogni nodo è foglia oppure ha due figli.*

Definizione 7.3 (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

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

(b) Heap

Di seguito alcune formule utili:

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

7.1 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.

$$\forall i \neq 1, A[parent(i)] \geq A[i] \tag{14}$$

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

$$\forall i \neq 1, A[parent(i)] \leq A[i] \tag{15}$$

7.2 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*

8 Algoritmi

8.1 Divide et impera

È una tecnica di risoluzione di problemi che consiste in tre passi:

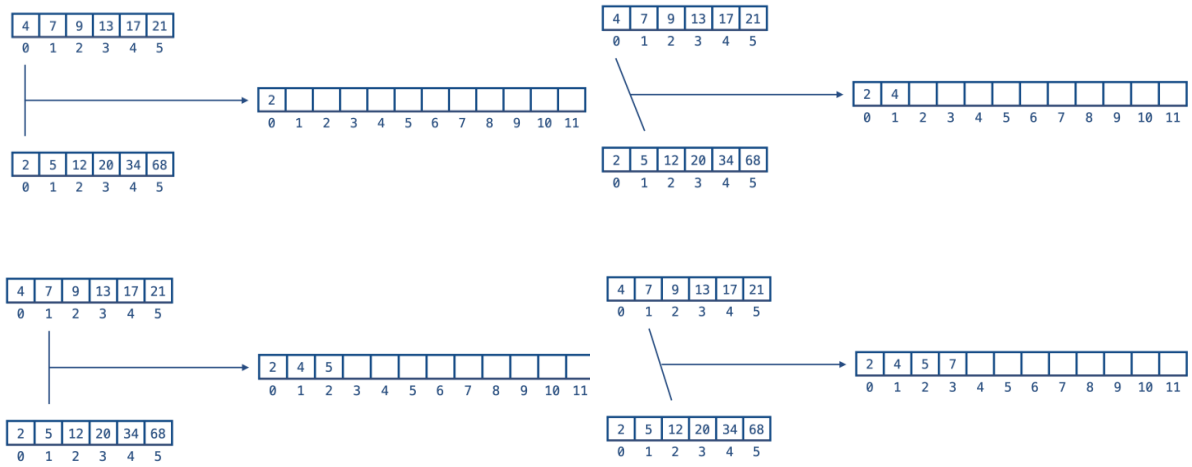
- **Dividere** il problema in 2 o più sotto problemi identici ma di dimensione ridotta rispetto a quello originale
- **Risolvere** i sotto problemi *ricorsivamente*
- **Combinare** le soluzioni dei sotto problemi per ottenere la soluzione del problema iniziale

8.2 Ordinamento

Definizione 8.1 (Algoritmo stabile)**.** *Un algoritmo di ordinamento si dice stabile quando preserva l'ordine iniziale tra due elementi con la stessa chiave.*

8.2.1 Merge sort

L'idea è di usare la tecnica precedentemente descritta del **Divide et Impera** e di spezzare l'array in due sotto-array di uguale dimensione, ordinarli e poi fonderli in uno unico.
La fusione verrà fatta confrontando i primi due elementi di ogni sotto-array, copiando il più piccolo nell'array finale, e proseguendo con il confronto del più grande con il successivo.



Esempio di implementazione:

```
1   void merge_sort(int a[], size_t dim, char order) {
2     sort(a, 0, dim-1, order);
3   }
4
5   void sort(int a[], size_t inizio, size_t fine, char order) {
6     if ((fine - inizio) >= 1) {
7       // Passo ricorsivo
8       size_t centro1 = (inizio + fine)/2;
9       zie_t centro2 = centro1 + 1;
10
11      sort(a, inizio, centro1, order);
12      sort(a, centro2, fine, order);
13
14      merge(a, inizio, centro1, centro2, fine, order);
15    }
16    // Il caso base non serve, un array di un elemento e' ordinato
17  }
18
19  void merge(int a[], size_t sin, size_t centro1, size_t centro2, size_t dx, char
        order) {
20    size_t sin i = sin;
```

```
21      size_t dx_i = centro2;
22      size_t fondi i = 0;
23      int temp_a[dx - sin + 1];
24
25      while (sin_i <= centro1 && dx_i <= dx) {
26        switch (order) {
27          case 'I':
28            if (a[sin_i] <= a[dx_i]) {
29              temp_a[fondi_i++] = a[sin_i++];
30            } else {
31              temp_a[fondi_i++] = a[dx_i++];
32            }
33            break;
34          default:
35            if (a[sin_i] <= a[dx_i]) {
36              temp_a[fondi_i++] = a[dx_i++];
37            } else {
38              temp_a[fondi_i++] = a[sin_i++];
39            }
40            break;
41        }
42      }
43
44      // Se esaurisco il sotto-array sinistro
45      if (sin_i == centro2) {
46        while (dx_i <= dx) {
47          temp_a[fondi_i++] = a[dx_i++];
48        }
49      } else {
50        // Se esaurisco quello destro
51        while (sin_i <= centro1) {
52          temp_a[fondi_i++] = a[sin_i++];
53        }
54      }
55
56      // Copio l'array temporaneo in quello originale
57      for (size_t i = sin; i <= dx; i++) {
58        a[i] = temp_a[i-sin];
59      }
60    }
```

Listing 11: Algoritmo merge sort

8.2.2 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 12: Algoritmo insertion sort

Complessità:

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

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

- 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.

8.2.3 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 13: Algoritmo selection sort

Complessità

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

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

- 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.

8.2.4 Bubble sort

Questo algoritmo scorre l'array e, a coppie, ordina gli elementi facendo più passate. Il nome *bubble* deriva dal fatto che ad ogni passata i numeri più grandi (o piccoli) si spostano verso la fine dell'array come le bolle d'aria salgono a galla.

```
1   void bubble_sort (int a[], size_t dim, char order) {
2     int temp;
3     for (unsigned int passate = 0; passate < dim; passate++) {
4       for (size_t i=0; i < (dim - 1); i++) {
5         switch (order) {
6           case 'I':
7             if (a[i] > a[i+1]) {
8               temp = a[i];
9               a[i] = a[i+1];
10              a[i+1] = temp;
11            }
12            break;
13          default:
14            if (a[i] < a[i+1]) {
15              temp = a[i];
16              a[i] = a[i+1];
17              a[i+1] = temp;
18               break;
19            }
20        }
21      }
22    }
23  }
```

Listing 14: Algoritmo bubble sort

Complessità
Il primo *for* esegue n cicli e quello interno ne esegue $n-1$. Di conseguenza la complessità è $n \cdot (n-1)$, ovvero $\mathbf{n^2}$.

8.3 Linear sort

Gli algoritmi di ordinamento di questo tipo sfruttano il fatto che l'array da ordinare abbia determinate proprietà.

Esempio 8.1. Dato un array A di n interi compresi tra 1 e k:

$$\forall 0 < j \leq n . A[j] \in [1, \ldots, k]$$

```
1   linearSort(A:[Int], B:[Int], k:Int) -> Void {
2     // Inizializzo un array che tiene conto dei numeri da 1 a k
3     for (var i:Int = 1; i<=k; i++) C[i] = 0;  Θ(k)
4
5     var j:Int = 1;
6     // Conto quante volte compare ogni numero nell'array originale
7     for (j=1; j<=n; j++) C[A[j]] += 1;  Θ(n)
8
9     j=1;
10    var z:Int = 1;
11    // Dispongo ogni numero nell'array finale in ordine sapendo quante volte compare
12    for (z=1; z <= k; z++) {  Θ(k)
13      for (var v:Int = 0; v < C[z]; v++) {  Θ(n)
14        B[j] = z;
15        j++;
16      }
17    }
18  }
```

Listing 15: Algoritmo linear sort

Complessità
In questo caso la complessità è $\Theta(n+k)$ e si usa quando $k \in O(n)$.

8.3.1 Radix sort

Questo algoritmo funziona in maniera simile a come il cervello umano ordina gruppi di numeri: si ordinano (tramite un algoritmo di ordinamento **stabile** prima le cifre delle migliaia, poi quelle delle centinaia, quelle delle decine ed infine le unità. Notiamo però che il risultato NON è corretto.

1094	**9**86	10**9**4	12**5**	1120
986	**2**34	1**2**5	112**0**	234
234	**1**25	11**2**0	23**4**	1094
125	**1**094	2**3**4	98**6**	125
1120	**1**120	9**8**6	109**4**	986

Per farlo funzionare dobbiamo ordinare le cifre partendo da quelle meno significative, quindi dalle unità.

109**4**	11**2**0	**1**120	**1**094	125
98**6**	10**9**4	**1**25	**1**120	234
23**4**	2**3**4	**2**34	125	986
12**5**	1**2**5	**9**86	234	1094
112**0**	9**8**6	**1**094	986	1120

9 Complessità

9.1 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 9.1 (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:

$$A \in \Theta(g(n)) \implies A \text{ ottimo}^3 \tag{16}$$

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**.

- **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

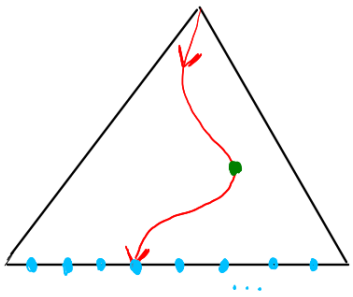


Figure 9: Albero decisionale

Si applica a problemi risolubili attraverso sequenze di decisioni che via via riducono lo spazio delle soluzioni.
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 9.1.1. *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 9.1. 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:

$$3^i \geq n + 1 > n \implies \text{binSearch} \in \Omega(\log_n) \tag{17}$$

9.2 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.

9.2.1 Limite superiore asintotico

Definizione 9.2 (Limite superiore asintotico)**.** *Il limite superiore asintotico* [4] *si definisce come:*

$$O(g(n)) = \{f(n) \mid \exists c, n_0 > 0. \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n)\} \tag{18}$$

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 9.2. 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 $A \in \Omega(g(n))$
[4] *Asintotico* indica che la definizione deve essere valida solo da un certo punto in poi scelto arbitrariamente.

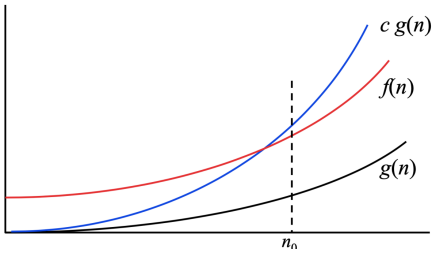


Figure 10: Limite superiore asintotico

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

9.2.2 Limite inferiore asintotico

Definizione 9.3 (Limite inferiore asintotico)**.** *Il limite inferiore asintotico si definisce come:*

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0. \forall n > n_0, 0 \leq c \cdot g(n) \leq f(n)\} \tag{19}$$

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

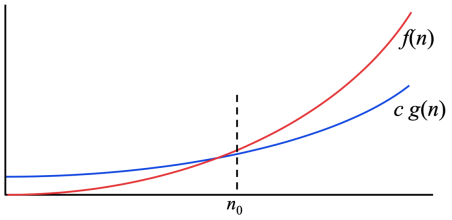


Figure 11: Limite superiore asintotico

9.2.3 Limite asintotico stretto

Definizione 9.4 (Limite asintotico stretto)**.** *Il limite asintotico stretto si definisce come:*

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0. \forall n > n_0, 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\} \tag{20}$$

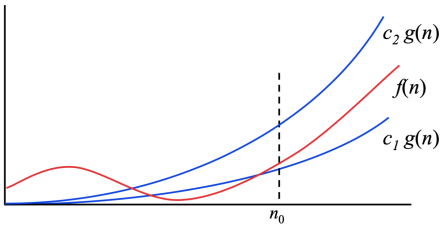


Figure 12: Limite asintotico stretto

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:

$$f(n) \in \Theta(g(n)) \iff f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n)) \tag{21}$$

9.2.4 Teoremi sulla notazione asintotica

Teorema 9.1. *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 $d \implies f(n) = \Theta(n^d)$*

9.2.5 Limite superiore asintotico non stretto

Definizione 9.5 (Limite superiore asintotico non stretto)**.** *Il limite superiore asintotico non stretto si definisce come:*

$$o(g(n)) = \{f(n) \mid \forall c \exists n_0 > 0. \forall n > n_0, 0 \leq f(n) \leq c \cdot g(n)\} \tag{22}$$

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.

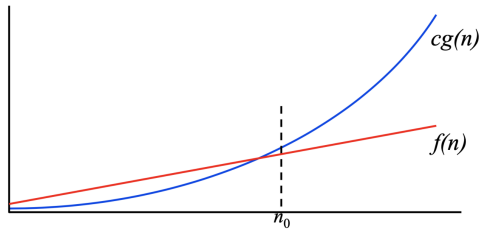


Figure 13: Limite superiore non stretto

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:

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

9.2.6 Limite inferiore asintotico non stretto

Definizione 9.6 (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.

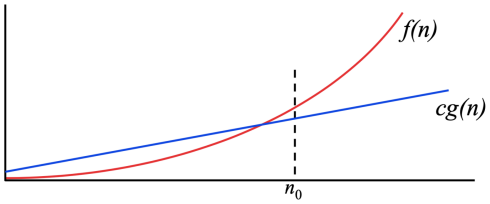


Figure 14: Limite asintotico stretto

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:

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

9.3 Equazioni di ricorrenza

Quando un algoritmo contiene una chiamata *ricorsiva* a se stesso, il suo tempo di esecuzione può essere descritto da una **equazione di ricorrenza**.

Definizione 9.7 (Ricorrenza)**.** *Una ricorrenza è un'equazione o una disequazione che descrive una funzione in termini del suo valore su input sempre più piccoli.*
*Un'**equazione ricorsiva** esprime il valore di $T(n)$ come combinazione di $T(n_1), \ldots, T(n_h)$ dove $n_i < n, i = 1, \ldots, h$:*

$$T(n) = \begin{cases} c & n \leq k \\ D(n) + \sum_{i=1}^{h} T(n_i) + C(n) & n > k \end{cases} \tag{23}$$

In un'equazione di ricorrenza:

- $\mathbf{T(n)}$ è il *tempo di esecuzione* di un problema di dimensione n
- Suddividiamo un problema in a **sotto problemi**, ciascuno di dimensione n/b:
  - $\mathbf{D(n)}$ è il tempo impiegato per *suddividere*
  - $\mathbf{C(n)}$ per *combinare* le soluzioni

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ a \cdot T(\frac{n}{b}) + D(n) + C(n)) & n > k \end{cases} \tag{24}$$

Alcuni casi particolari di equazioni ricorrenza sono quelle di **ordine k**:

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ \alpha_1 \cdot T(n-1) + \ldots + \alpha_k \cdot T(n-k) + f(n) & n > k \end{cases} \tag{25}$$

e quelle **bilanciate**:

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ a \cdot T(\frac{n}{b}) + f(n) & n > k \end{cases} \tag{26}$$

con $a \geq 1$, $b > 1$, f asintoticamente positiva.

9.3.1 Metodo iterativo

Prendiamo come esempio l'algoritmo di ordinamento *merge sort* e analizziamone la complessità:

```
1   void sort(int a[], size_t inizio, size_t fine, char order) {
2     if ((fine - inizio) >= 1) {
3       size_t centro1 = (inizio + fine)/2; Θ(1)
4       zie_t centro2 = centro1 + 1; Θ(1)
5
6       sort(a, inizio, centro1, order); T()
7       sort(a, centro2, fine, order);
8
9       merge(a, inizio, centro1, centro2, fine, order);
10    }
11  }
```

Listing 16: Algoritmo merge sort

Ora scriviamo l'equazione di ricorrenza come segue:

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n)) & n \geq 2 \end{cases} \tag{27}$$

Possiamo risolverla utilizzando il metodo iterativo, sapendo che dovremo fare $i = \log_2 n$ iterazioni:

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n}{2}) + c \cdot n = 2 \cdot (2 \cdot T(\frac{n}{4}) + c \cdot \frac{n}{2}) + c \cdot n = \\ &= 4 \cdot T(\frac{n}{4}) + c \cdot n + c \cdot n = 4 \cdot (2 \cdot T(\frac{n}{8}) + c \cdot \frac{n}{4}) + 2 \cdot c \cdot n = \\ &= 8 \cdot T(\frac{n}{8}) + c \cdot n + 2 \cdot c \cdot n = 8 \cdot T(\frac{n}{8}) + 3 \cdot c \cdot n = \\ &= \ldots = 2^i \cdot T(\frac{n}{2^i}) + i \cdot c \cdot n = 2^{\log_2 n} \cdot T(1) + \log_2 n \cdot c \cdot n = \\ &= n \cdot \Theta(1) + c \cdot n \cdot \log n = \Theta(n \cdot \log n) \end{aligned} \tag{28}$$

9.3.2 Albero di ricorsione

9.3.3 Master's Theorem

Quando si tratta di risolvere equazioni di ricorrenza **bilanciate**, è possibile utilizzare il Master's Theorem.

$$T(n) = \begin{cases} \Theta(1) & n \leq k \\ a \cdot T(\frac{n}{b}) + f(n) & n > k \end{cases} \tag{29}$$

L'intuizione consiste nel fare un confronto tra $f(n)$ e $n^{\log_b a}$, ovvero quante volte viene eseguito il passo ricorsivo.
Ci sono tre casi possibili:

- **Minore**: $f(n) = O(n^{\log_b a - \epsilon})$ per qualche costante $\epsilon > 0$.
  $f(n)$ cresce **polinomialmente** più lentamente di $n^{\log_b a}$ di un fattore n^ϵ.
  Soluzione: $T(n) = \Theta(n^{\log_b a})$

  Esempio 9.4. Data la seguente equazione di ricorrenza:

  $$T(n) = 9 \cdot T(\frac{n}{3}) + n \tag{30}$$

  Abbiamo che $a = 9$, $b = 3$, $f(n) = n$, $n^{\log_3 9} = n^2$. Possiamo dedurre quindi che, per un $\epsilon = 1$:

  $$f(n) = n = O(n^{\log_3 9 - \epsilon}) = O(n) \tag{31}$$

- **Uguale**: $f(n) = \Theta(n^{\log_b a} \cdot \ln^k n)$ per qualche costante $k \geq 0$.
  $f(n)$ e $n^{\log_b a}$ crescono allo stesso modo.
  Soluzione: $T(n) = \Theta(n^{\log_b a} \cdot \ln^{k+1} n)$

- **Maggiore**: $f(n) = \Omega(n^{\log_b a + \epsilon})$ per qualche costante $\epsilon > 0$.
  $f(n)$ cresce **polinomialmente** più in fretta di $n^{\log_b a}$ di un fattore N^ϵ e rispetta la **condizione di regolarità**: $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ per qualche costante $c < 1$ e $n > n_0$.
  Soluzione: $T(n) = \Theta(f(n))$

Osservazione 9.3.1. Il Master's Theorem si può usare solamente quando $f(x)$ cresce **polinomialmente** più in fretta o lentamente di $n^{\log_b a}$. Ad esempio se avessimo $f(x) = \log n$ non potremmo utilizzarlo.

10 Liste

Una **struct** (struttura) **autoreferenziale** ha un membro puntatore che punta a una struttura dello stesso tipo, chiamato **link**, e che serve a creare una *catena* (lista) di nodi collegati tra loro.

```
1  struct nodo {
2    int dato;
3    struct nodo *nextPtr;
4  } n1, n2, n3;
```

Listing 17: Esempio di una struct

Definizione 10.1 (Struttura dinamica)**.** *È una struttura dati che può **variare** la sua dimensione a tempo di esecuzione, aumentando e diminuendo. Alcuni esempi sono proprio le **liste**, oltre che le pile, le code e gli alberi binari.*

Note 10.0.1. *L'ultimo nodo di una lista conterrà nel link **null**.*

10.1 Confronto tra liste e array

Liste	**Array**
Contiene sequenze di dati	Contiene sequenze di dati
Create dinamicamente a tempo di esecuzione e la dimensione non può essere prevista a tempo di compilazione	Creati staticamente a tempo di esecuzione e la dimensione deve essere calcolabile a tempo di compilazione
La dimensione è variabile	La dimensione è costante e tutti gli elementi sono allocati a tempo di definizione
Diventa piena solo quando termina la memoria disponibile nell'heap	Diventa pieno quando sono pieni tutti gli elementi
Non sono memorizzate in celle contigue	Sono in celle contigue
Possono essere manipolate senza spostare elementi	Per manipolarli bisogna spostare gli elementi

10.2 Operazioni sulle liste

10.2.1 Insert

Viene codificata come segue:

```
1  void insert(NodoPtr *lPtr, int val) {
2    // Alloco nuovo nodo
3    NodoPtr nuovoPtr = malloc(sizeof(Nodo));
4    if (nuovoPtr != NULL) {
5      // Inizializzo nodo
6      nuovoPtr->dato = val;
7      nuovoPtr->prossimoPtr = NULL;
8      NodoPtr precedentePtr = NULL;
9      NodoPtr correntePtr = *lPtr;
10     while (correntePtr != NULL && val > correntePtr->dato) {
11       precedentePtr = correntePtr;
12       correntePtr = correntePtr->prossimoPtr;
13     }
14     if (precedentePtr == NULL) {
15       // Inserimento all'inizio della lista
16       nuovoPtr->prossimoPtr = *lPtr;
17       *lPtr = nuovoPtr;
18     }
19     else {
20       // Inserimento tra due nodi
```

```
21          precedentePtr->prossimoPtr = nuovoPtr;
22          nuovoPtr->prossimoPtr = correntePtr;
23        }
24      }
25      else {
26        puts("Memoria esaurita");
27      }
28    }
```

Listing 18: Inserimento in una lista

10.2.2 Delete

Viene codificata come segue:

```
1  void delete(NodoPtr *lPtr, int val) {
2    if (*lPtr != NULL) {
3      if (val == (*lPtr)->dato) {
4        NodoPtr tempPtr = *lPtr;
5        *lPtr = (*lPtr)->prossimoPtr;
6        free(tempPtr);
7      }
8      else {
9        NodoPtr precedentePtr = *lPtr;
10       NodoPtr correntePtr = (*lPtr)->prossimoPtr;
11       while (correntePtr != NULL && correntePtr->dato != val) {
12         precedentePtr = correntePtr;
13         correntePtr = correntePtr->prossimoPtr;
14       }
15       if (correntePtr != NULL) {
16         NodoPtr tempPtr = correntePtr;
17         precedentePtr->prossimoPtr = correntePtr->prossimoPtr;
18         free(tempPtr);
19       }
20     }
21   }
22 }
```

Listing 19: Cancellazione in una lista

10.2.3 Verifica se è vuota

```
1  int is_empty(NodoPtr lPtr) {
2    return lPtr == NULL;
3  }
```

Listing 20: Verificare se la lista è vuota

10.3 Liste particolari

10.3.1 Pile

Definizione 10.2. *Una **pila** è una lista in cui inserimenti e cancellazioni possono essere fatte solo sulla testa della lista (politica **LIFO**).*

Le operazioni che possono essere eseguite sulle pile sono:

- **Push**: inserimento di un nuovo nodo in testa

```
1      void push(NodoPtr *topPtr, int val) {
2        // alloco nuovo nodo
3        NodoPtr nuovoPtr = malloc(sizeof(Nodo));
4        if (nuovoPtr != NULL) { // Spazio disponibile
5          // inizializzo nodo
6          nuovoPtr->dato = val;
7          nuovoPtr->prossimoPtr = *topPtr;
8          *topPtr = nuovoPtr;
9        }
```

```
10        else {
11          puts("Memoria esaurita");
12        }
13      }
```

- **Pop**: cancellazione di un elemento in testa

```
1       int pop(NodoPtr *topPtr) {
2         int val = (*topPtr)->dato;
3         NodoPtr tempPtr = *topPtr;
4         *topPtr = (*topPtr)->prossimoPtr;
5         free(tempPtr);
6         return val;
7       }
```

- **is_empty**, **stampa_pila**

10.3.2 Code

Definizione 10.3. *Una **pila** è una lista in cui inserimenti e cancellazioni possono essere fatte solo alla fine della lista (politica **FIFO**).*

Le operazioni che possono essere eseguite sulle pile sono:

- **Enqueue**: inserimento di un nuovo nodo alla fine della coda

```
1       void enqueue(NodoPtr *testaPtr, NodoPtr * codaPtr, int val) {
2         // alloco nuovo nodo
3         NodoPtr nuovoPtr = malloc(sizeof(Nodo));
4         if (nuovoPtr != NULL) { // Spazio disponibile
5           // inizializzo nodo
6           nuovoPtr->dato = val;
7           nuovoPtr->prossimoPtr = NULL;
8           if (is_empty(*testaPtr)) {
9             *testaPtr = nuovoPtr;
10          }
11          else {
12            (*codaPtr)->prossimoPtr = nuovoPtr;
13            *codaPtr = nuovoPtr;
14          }
15        }
16        else {
17          puts("Memoria esaurita");
18        }
19      }
```

- **Dequeue**: cancellazione di un elemento in testa

```
1       int dequeue(NodoPtr *testaPtr, NodoPtr * codaPtr) {
2         int val = (*testaPtr)->dato;
3         NodoPtr tempPtr = *testaPtr;
4         *testaPtr = (*testaPtr)->prossimoPtr;
5         if (*testaPtr == NULL) {
6           *codaPtr = NULL;
7         }
8         free(tempPtr);
9         return val;
10      }
```

- **is_empty**, **stampa_coda**

11 Dizionari

Definizione 11.1. *Un dizionario è una* ***struttura dati astratta*** *e consiste in una collezione di* ***coppie*** *della forma:*

- ***Chiave****: è il meccanismo di accesso all'elemento ed è* ***univoca*** *nella collazione*
- ***Elemento****: è un qualsiasi tipo*

Le operazioni che possono essere eseguite sui dizionari sono *inserimento*, *ricerca* e *cancellazione*.

11.1 Indirizzamento diretto

Ogni elemento del dizionario viene mappato con una chiave estratta da un determinato universo U e che corrisponde alla posizione nell'array dove viene inserito.
La complessità di questa tecnica è $O(1)$ per tutte e tre le operazioni possibili sui dizionari. Il problema si presenta quando le chiavi usate N sono molto minori dell'universo iniziale U e abbiamo quindi che $\frac{N}{U} < 1$ con un conseguente **spreco di spazio di memoria** in quanto lo spazio nell'array dovrà comunque essere allocato.

11.2 Chaining

Un modo per evitare i problemi dell'*indirizzamento diretto* è utilizzando un array più piccolo dell'universo possibile di chiavi. Questo ovviamente causa situazioni in cui c'è un **conflitto** perché due o più elementi voglio utilizzare la stessa chiave. Viene quindi creata una **lista** di elementi associata alla posizione in cui si è creato il conflitto e ogni nuovo elemento che vuole inserirsi lì verrà messo sulla testa della lista.
Per capire ogni elemento a quale chiave deve essere associato si utilizza una **funzione di hash** che deve essere definita in modo da ridurre il numero di collisioni e da generare sempre lo stesso indice per la stessa chiave.

11.3 Open addressing

I conflitti di indirizzo vengono risolti cercando la prima posizione libera. Questo processo di ricerca viene chiamato **probing**. Questa operazione può avvenire in diversi modi:

- *Lineare*: vado alla posizione successiva
- *Quadratico*: uso una funzione quadratica
- *Doppio hash*: utilizzo la funzione hash per cercare la posizione successiva

Esempio 11.1. Esempio di esecuzione di *probing*:

$$\begin{gathered} probe(0) \longrightarrow h(i) \\ probe(1) \longrightarrow (h(i) + f(1)) \\ \vdots \\ probe(j) \longrightarrow (h(i) + f(j)) \\ \vdots \\ probe(m-1) \longrightarrow Fail \end{gathered}$$

Esempio 11.2. Esempio di *probing lineare*.

Osservazione 11.3.1. Il problema che nasce dell'open addressing lineare è il **clustering primario**. Quando infatti si verifica una collisione, viene riempita una cella di memoria nelle vicinanze, aumentando così il rischio di collisione e aumentando il tempo necessario per la ricerca.
Questo avviene perché la *probabilità* che uno slot preceduto da slot pieni sia riempito è pari a $\frac{i+1}{m}$.

Osservazione 11.3.2. Il problema che deriva dall'utilizzo di open addressing quadratico è il **clustering secondario**. Infatti in questo caso se due chiavi hanno la stessa posizione iniziale la loro sequenza di tentativi sarà la stessa.

12 Alberi

12.1 Rappresentazione

12.1.1 Array

Per rappresentare un albero *binario* di profondità d possiamo utilizzare un **array** di dimensione $2^{d+1}-1$. Questa scelta può portare a dei vantaggi e svantaggi:

- **Vantaggi**:
  - Accesso diretto ai nodi
- **Svantaggi**:
  - L'*altezza* dell'albero deve essere nota
  - Spreco di memoria
  - *Inserimento* e *cancellazione* sono operazioni complicate

Per questi motivi gli array si usano raramente per la rappresentazione di alberi.

12.1.2 Liste

Il modo più usato per la rappresentazione di alberi è quello delle **liste**, codificandoli come segue:

```
1  struct node {
2    int data;
3    struct node *left;
4    struct node *right;
5  }
```

Listing 21: Alberi con liste

Questa scelta ci porta a vantaggi e svantaggi:

- **Vantaggi**:
  - L'*altezza* dell'albero non deve essere nota
  - Nessuno spreco di memoria
  - *Inserimento* e *cancellazione* sono operazioni facili
- **Svantaggi**:
  - Mancanza di accesso diretto ai nodi
  - Memoria aggiuntiva per memorizzare figlio destro e sinistro

12.2 Visitare

Possiamo effettuare l'operazione di **visita** su un albero binario in tre modi diversi. Tutti questi algoritmi avranno **complessità** $O(n)$.

12.2.1 Anticipata

```
1  Anticipata(x):
2    if x != NULL
3      print(x.key)
4      Anticipata(x.left)
5      Anticipata(x.right)
```

Listing 22: Alberi con liste

Esempio 12.1.

12.2.2 Posticipata

```
1  Posticipata(x):
2    if x != NULL
3      Posticipata(x.left)
4      Posticipata(x.right)
5      print(x.key)
```

Listing 23: Alberi con liste

Esempio 12.2.

12.2.3 Simmetrica

```
1  Simmetrica(x):
2    if x != NULL
3      Simmetrica(x.left)
4      print(x.key)
5      Simmetrica(x.right)
```

Listing 24: Alberi con liste

Esempio 12.3.

12.3 Albero binario di ricerca

Un caso particolare di albero binario è un albero binario di ricerca. Questo rispecchia le seguenti proprietà, dato un nodo x, applicate ricorsivamente:

- $x.left.key \leq x.key$, ovvero tutti i nodi alla sinistra sono i minori di x
- $x.right.key \geq x.key$, ovvero tutti i nodi alla destra sono i maggiori di x

Note 12.3.1. *Per effettuare la stampa **ordinata** degli elementi dobbiamo utilizzare la visita **simmetrica**.*

12.3.1 Ricerca

Algoritmo ricorsivo per la ricerca di un elemento:

```
1  RicercaABR_R(x, k)
2    if x == NULL OR k == x.key
3      return x
4    if k < x.key
5      return RicercaABR_R(x.left, k)
6    else return RicercaABR_R(x.right, k)
```

Listing 25: Ricerca ricorsiva

Algoritmi per la ricerca del valore minimo e massimo di un albero:

```
1  RicercaMIN_I(x)
2    while x.left != NULL
3      x = x.left
4    return x
```

Listing 26: Ricerca del minimo

```
1  RicercaMAX_I(x)
2    while x.right != NULL
3      x = x.right
4    return x
```

Listing 27: Ricerca del massimo

13 Grafi

Per la definizione di grafo si rimanda agli appunti di Fondamenti dell'Informatica. Ricordiamo solo le caratteristiche principali:

- **V** rappresenta l'insieme di vertici
- **E** rappresenta l'insieme di archi
- Un grafo si dice **denso** quando $|E| \approx |V|^2$
- Un grafo si dice **sparso** quando $|E| \approx |V|$
- Un grafo può essere **orientato** o **non orientato**
- Il grafo è **pesato** quando viene associato un *peso* agli archi o ai nodi

13.1 Rappresentazione

Dato un grafo con n vertici tale che $V = \{1, 2, \ldots, n\}$ abbiamo due possibili modi di rappresentarlo.

13.1.1 Matrice di adiacenza

Una **matrice di adiacenza** rappresenta il grafo come una **matrice** A di dimensione $n \times n$. In questa matrice se un arco (i, j) esiste allora l'elemento $A[i, j]$ sarà valorizzato a 1, altrimenti sarà 0. Lo *spazio occupato* da questa matrice è $|V|^2$ il che la rende efficiente solo per grafi piccoli.

Note 13.1.1. *Si noti che se il grafo è* ***non orientato*** *serve solo metà matrice, in quanto* $(i, j) = (j, i)$.

13.1.2 Lista di adiacenza

Una **lista di adiacenza** rappresenta il grafo come un array di liste la cui dimensione dipende dal numero di archi presenti. Occupa $n + m$ spazio.

13.2 Ricerca in un grafo

Per cercare un elemento in un grafo dobbiamo esplorare ogni vertice e arco a partire da una sorgente s. Come risultato otteniamo un **albero** basato sul grafo iniziale che ha la sorgente come radice (o una foresta se il grafo non è connesso).

Definizione 13.1 (Percorso minimo)**.** *Un percorso minimo in un grafo G tra due vertici s, v è un percorso da s a v che contiene il minimo numero di archi. La sua lunghezza è detta* ***distanza minima*** *e viene indicata come $\delta(s, v)$.*
Una proprietà importante dato un arco tra due nodi (u, v) e un nodo v è la seguente:

$$\delta(s, v) \leq \delta(s, u) + 1$$

13.2.1 Breadth-First Search

Questo algoritmo esplora il grafo un vertice alla volta espandendo la frontiera dei vertici esplorati in **ampiezza**. Per evitare di attraversare più volte nodi già visitati associamo alcuni "colori" ad ogni nodo:

- **Bianco**: vertici non ancora visitati
- **Grigio**: vertici visitati ma non ancora esplorati
- **Nero**: vertici completamente esplorati

Il concetto quindi è di esplorare i vertici scorrendo le *liste di adiacenza* dei vertici grigi. Ogni vertice da esplorare viene aggiunto in una **queue** (coda) e appena vengono terminati e diventano neri vengono rimossi. Di seguito una possibile implementazione dell'algoritmo:

```
1  BFS(G, s) {
2    inizializza vertici; // Vertici inizializzati ad ∞
3    Q = {s}; // Q coda inizializzata con s e 0
4    while (Q non vuota) {
5      u = RemoveTop(Q);
6      for each v ∈ u->adj {
7        if (v->d == infinito)
8        v->d = u->d + 1;
9        v->p = u;
10       Enqueue(Q, v);
11     }
12   }
13 }
```

Listing 28: Implementazione di BFS

La **complessità** in tempo di questo algoritmo è $O(V + E)$ mentre quella in spazio è $O(V)$.
L'algoritmo in questione genera un **albero breadth-first**, dove i cammini verso la radice rappresentano i cammini minimi nel grafo G.

Definizione 13.2 (Albero breadth-first)**.** *Dato un grafo $G = \langle V, E \rangle$ con sorgente s, un albero breadth-first è un albero $G' = \langle V', E' \rangle, V' \subseteq V, E' \subseteq E$ tale che:*

- *G' è un **sottografo** di G*
- *Un vertice v appartiene all'albero se e solo se quel vertice è **raggiungibile** dalla sorgente nel grafo G*
- *Per ogni vertice dell'albero il percorso dalla sorgente è **minimo***

13.2.2 Depth-First Search

14 Programmazione dinamica

Si applica negli algoritmi ricorsivi in cui i sotto problemi ottenuti dalla tecnica *Divide et impera* si ripropongono più volte, causando uno spreco di risorse considerevole. Si dice che questi sotto problemi non sono **indipendenti**.
La tecnica consiste nel memorizzare le soluzioni in una **tabella** dei sotto problemi in modo da potervi accedere quando li si incontra di nuovo senza doverli risolvere nuovamente.

Esempio 14.1 (Fibonacci)**.** Generare la sequenza di Fibonacci, che ricordiamo essere definita come:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

```
1     Fib(n)
2       if(n==0) return 0;
3       if(n==1) return 1;
4       return Fib(n-1)+Fib(n-2);
```

Listing 29: Fibonacci

In questa soluzione di codice le somme eseguite sono in numero esponenziale:

$$T(n) = \begin{cases} 0 & n \leq 1 \\ 1 + T(n-1) + T(n-2) & n > 1 \end{cases}$$

Per calcolare la successione di Fibonacci con **memoization** di tipo **top down** usiamo il seguente algoritmo, sempre ricorsivo:

```
1     Fib(n)
2       F: new array(0:n) // Dimensione n+1
3       for i=0 to n
4         F(i) = -1
5       return FibRic(n, F)
6
7     FibRic(n, F)
8       if(n==0 || n==1) return n;
9       if(F(n) != -1) return F[n]
10      else
11        F[n] = FibRic(n-1, F) + FibRic(n-2, F);
12        return F[n];
```

Listing 30: Fibonacci dinamico top-down

Utilizzando invece il metodo **bottom-up**:

```
1     Fib(n)
2       F: new array(0:n) // Dimensione n+1
3       F[0] = 0;
4       F[1] = 1;
5       for i=2 to n
6         F(i) = F[i-1] + F[i-2]
7       return F[n]
```

Listing 31: Fibonacci dinamico bottom-up

La complessità di questi algoritmi è $\Theta(n)$ in tempo e $\Theta(n)$ in spazio, a differenza dell'algoritmo non dinamico che ha una complessità di ϕ^n in tempo e Per ottimizzare l'algoritmo in spazio possiamo fare nel modo seguente:

```
1     Fib(n)
2       if(n==0) return 0;
3       if(n==1 || n==2) return 1;
4       a = 1
5       b = 1
6       for i=3 to n
7         c = a+b
8         a = b
```

```
9           b = c
10        return b;
```

Listing 32: Fibonacci spazio costante

Osservazione 14.0.1 (Complessità in spazio)**.** Come sappiamo il numero di cifre necessarie per rappresentare un numero n è $\log_x n$ dove x è la base in cui scriviamo il numero. Di conseguenza in quest'ultimo algoritmo in realtà è **pseudo-polinomiale** in quanto passiamo da un'istanza di input logaritmica ad una complessità lineare.

14.1 Struttura di un algoritmo

La programmazione dinamica si applica a problemi di ottimizzazione con queste caratteristiche:

- **Sottostruttura ottima**: la soluzione ottima del problema si può costruire dalle soluzioni ottime dei sottoproblemi
- **Sovrapposizione dei sottoproblemi** (o ripetizione)

Gli algoritmi sono costituiti da 4 fasi:

1. Definizione dei sotto problemi e dimensionamento della tabella.
2. Soluzione diretta dei sotto problemi elementari e inserimento di questi nella tabella (approccio **bottom-up**)
3. Definizione della regola ricorsiva per ottenere la soluzione di un sotto problema a partire dalle soluzioni dei sotto problemi già risolti (già nella tabella)
4. Restituzione del risultato relativo al problema di dimensione n

Esempio 14.2 (Taglio della corda)**.** Data una corda di lunghezza n e una tabella di prezzi (un pezzo di dimensione diversa ha un valore diverso), trovare il taglio ottimale della corda per massimizzare il guadagno.
Un possibile modo di affrontare il problema è tramite **brute-force**, che comporta analizzare ogni possibile taglio della corda. Notiamo che possiamo dividere la corda in 2^{n-1} possibili modi.
Un approccio **ricorsivo** al problema è quello di tagliare la corda al punto i. In questo modo abbiamo che il ricavo massimo ottenibile sarà il prezzo del pezzo tagliato $p(i)$ e il ricavo massimo della corda restante r_{n-i}. In generale il ricavo massimo quindi sarà $r_n = max_{1 \leq i \leq n}(p(i) + r(n-i))$.

```
1     CUT_ROD(P, n)
2       if(n==0) return 0;
3       q = -∞
4       for i=1 to n
5         q = max{q, p[i] + CUT_ROD(P, n-i)}
6       return q;
```

Listing 33: Taglio della corda

Questo algoritmo, per quanto breve, è estremamente inefficiente (esponenziale) a causa della ripetizione degli stessi sotto problemi. È quindi necessaria la *programmazione dinamica*.

Esempio 14.3 (Longest common subsequence)**.**

Esempio 14.4 (Edit distance)**.** Date due parole X, Y determinarne la distanza.
La prima fase è quella dell'**allineamento**: ad ogni carattere o spazio di X corrisponde un carattere o uno spazio di Y.
Una volta eseguito l'allineamento, si calcola la distanza con queste regole:

- **MATCH** (caratteri corrispondenti uguali) → distanza + 0
- **MISMATCH** (caratteri corrispondenti diversi) → distanza + 1
- **SPACE** (carattere e spazio) → distanza + 1

Il problema della edit-distance è determinare la distanza *minima*.

14.2 Tecnica Greedy

Esempio 14.5 (Zaino frazionario)**.** Il problema è quello dello zaino visto in precedenza ma in questo caso il ladro può prendere anche frazioni di oggetti.

Correttezza: occorre dimostrare che il problema soddisfi gli elementi della tecnica greedy:

- *Sottostruttura ottima*
- *Proprietà della scelta greedy*: per assurdo, supponiamo che qualche scelta non sia greedy (localmente ottima). Anziché scegliere un oggetto m di valore specifico $\frac{v_m}{w_m}$, scegliamo p kg dell'oggetto q di valore specifico $\frac{v_q}{w_q}$.

$$\frac{v_q}{w_q} < \frac{v_m}{w_m}$$

Sia $r = min\{p, w_m\}$

Esempio 14.6 (Scheduling delle attività)**.** Ogni attività inizia e finisce in due istanti di tempo diversi. Il problema consiste nel massimizzare il numero di attività eseguibili rispettando il vincolo di *non*

i	1	2	3	4	5	6	7	8	9	10	11
S_i											
T_i											

sovrapposizione.
Utilizziamo la strategia greedy:

- Seleziona l'attività che finisce prima
- Elimina le attività che non sono compatibili, ovvero che si sovrappongono con quella selezionata
- Ripeti

15 Teoria della calcolabilità

Si occupa delle questioni fondamentali circa la **potenza** e le **limitazioni** dei sistemi di calcolo.
L'origine risale alla prima metà del ventesimo secolo, quando i logici matematici iniziarono ad esplorare i concetti di:

- Computazione
- Algoritmo
- Problema risolvibile per via algoritmica

e dimostrano l'esistenza di problemi che non ammettono un algoritmo di risoluzione.

Definizione 15.1 (Problemi computazionali)**.** *Problemi formulati **matematicamente** di cui cerchiamo una soluzione algoritmica. Si classificano in:*

- *Problemi **non decidibili**, non ammettono un algoritmo di risoluzione*
- *Problemi **decidibili**, che a loro volta possono essere:*
  - ***Trattabili**, ovvero di costo polinomiale*
  - ***Non trattabili**, ovvero di costo esponenziale*

Facciamo ora la distinzione tra:

- **Calcolabilità**: sfrutta le nozioni di *algoritmo* e di *problema non decidibile*. Ha lo scopo di classificare i problemi in risolvibili e non risolvibili.
- **Complessità**: sfrutta le nozioni di *algoritmo efficiente* e di *problema non trattabile*. Ha lo scopo di classificare i problemi in "facili" e "difficili".

15.1 Problemi indecidibili

15.2 Problemi decidibili ma intrattabili

Esempio 15.1 (Torre di Hanoi)**.**

```
TorriHanoi(n, p, t, s)
    if(n==1) print p->t;
    else{
      TorriHanoi(n-1, p, s, t);
      print p->t;
      TorriHanoi(n-1, s, t, p);
    }
```

Listing 34: Torre di Hanoi

Scriviamo la relazione di ricorrenza:

$$\begin{cases} 1 & n = 1 \\ 2M(n-1)+1 & n > 1 \end{cases}$$

Non essendo risolvibile con il Master's Theorem, proviamo con il metodo di sostituzione: Dalla sostituzione sembra che $M(n) = 2^n - 1$. Dimostriamolo per induzione su n:

n	1	2	3	4	...	i

- **Caso base**: per $n = 1$, $M(1) = 1$ e $2^1 - 1 = 2 - 1 = 1$
- **Ipotesi induttiva**: $M(i) = 2^i - 1, \forall i < n$
- **Passo induttivo**: per $n > 1$, $M(n) = 2M(n-1) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 1$

Abbiamo quindi dimostrato che Ora vogliamo capire qual'è il numero di mosse **necessarie**

La conclusione è che l'algoritmo ricorsivo dimostrato in precedenza è **ottimo** in quanto $2^n - 1$ mosse sono **necessarie** e **sufficienti**.
Se i monaci spostassero 1 disco al secondo, per spostare i 64 dischi ci vorrebbero comunque $2^{64} - 1$ secondi, ovvero circa $585 \cdot 10^9$ anni.
È quindi lecito dire che un problema con soluzione esponenziale è spesso a livello pratico assimilabile ad un problema indecidibile.

Esempio 15.2 (Generazione delle sequenze binarie)**.** Dato $A = \{a_0, a_1, \ldots, a_{n-1}\}$ insieme di n oggetti. Il numero di sottoinsiemi di A è 2^n in quanto lo possiamo descivere con sequenze binarie. Ad esempio dato $A' \subseteq A$ tale che $A' = \{a_0, a_3, a_5, a_6\}$:

$$\Big\{2$$

16 Teoria della complessità

Come abbiamo visto nella definizione 15.1 dividiamo i problemi in diverse categorie.
I problemi che vedremo in questa sezione sono **presumibilmente intrattabili**, ovvero che abbiamo a disposizione solo algoritmi di costo *esponenziale* per risolverli ma che nessuno ha dimostrato effettivamente che non possano esistere algoritmi polinomiali.

Esempio 16.1 (Problema della clique)**.** Dato un grafo $F = (V, E)$ e un intero $k > 0$, stabilire se G contiene un clique (sottografo completo) di almeno k nodi.

Esempio 16.2 (Problema del cammino (o ciclo) Hamiltoniano)**.** Dato un grafo $G = (V, E)$, trovare un cammino (o ciclo) semplice che passa da tutti i vertici di G una ed una sola volta.

16.1 Velocità dei calcolatori

Studiamo ora la dimensione dei dati trattabili in funzione dell'incremento della velocità dei calcolatori per dimostrare che lo sviluppo tecnologico non riesce a bilanciare un algoritmo inefficiente.

Dati due calcolatori C_1, C_2 con C_2 k volte più veloce di C_1. Il tempo di calcolo a disposizione è t e:

- n_1 rappresenta i dati trattabili nel tempo t su C_1
- n_2 rappresenta i dati trattabili nel tempo t su C_2

Osservazione 16.1.1. Usare C_2 per un tempo t equivale a usare C_1 per un tempo $k \cdot t$.

Algoritmo polinomiale che risolve il problema in $c \cdot n^s$ secondi (c, s costanti).

- C_1: $c \cdot n_1^s = t \longrightarrow n_1 = (\frac{t}{c})^{\frac{1}{s}}$
- C_2: $c \cdot n_2^s = t \longrightarrow n_1 = (k \cdot \frac{t}{c})^{\frac{1}{s}} = k^{\frac{1}{s}} \cdot (\frac{t}{c})^{\frac{1}{s}}$

Concludiamo quindi che il miglioramento è di un fattore moltiplicativo $K^{\frac{1}{s}}$. Ad esempio per $k = 10^9$ e $s = 3$ i dati trattabili saranno moltiplicati per 10^3.

16.2 Tipi di problemi

I tipi di problemi che possiamo studiare sono i seguenti:

- **Problemi decisionali**: richiedono una risposta binaria, ad esempio determinare se un numero è primo
- **Problemi di ricerca**: data un'istanza x, richiedono di restituire una soluzione s, ad esempio trovare un cammino tra due vertici.
- **Problemi di ottimizzazione**: data un'istanza x, si vuole trovare la *migliore* soluzione s tra tutte le soluzioni possibili. Ad esempio la ricerca della clique di dimensione massima.

16.3 Problemi decisionali

Nella teoria della complessità si studiano solamente i problemi **decisionali**, questo perché:

- Essendo la risposta binaria, non ci si deve preoccupare del tempo richiesto per restituire la soluzione e quindi tutto il tempo è speso per il calcolo
- La difficoltà di un problema è già presente nella usa versione decisionale. Tutti i problemi di ottimizzazione sono esprimibili in forma decisionale, chiedendo l'esistenza di una soluzione che soddisfi una certa proprietà. Il problema di **ottimizzazione** è quindi almeno tanto difficile quanto quello decisionale e quindi mi basta caratterizzare la complessità di quest'ultimo per dare un limite inferiore alla complessità del primo.

16.4 Classi di complessità

Dato un problema decisionale Π ed un algoritmo A, diciamo che A risolve Π se, data un'istanza di input x

$$A(x) = 1 \iff \Pi(x) = 1$$

A risolve P in tempo $t(n)$ e spazio $s(n)$ se il tempo di esecuzione e l'occupazione di memoria di A sono rispettivamente $t(n)$ e $s(n)$. Data una qualunque funzione $f(n)$:

- $Time(f(n))$: insieme dei problemi decisionali che possono essere risolti in **tempo** $O(f(n))$.
- $Space(f(n))$: insieme dei problemi decisionali che possono essere risolti in spazio $O(f(n))$

16.4.1 Classe P

Definizione 16.1 (Algoritmo polinomiale in tempo)**.** *esistono due costanti c, n0 ¿ 0 t.c. il numero di passi elementari è al più nc per ogni input di dimensione n e per ogni n ¿ n0*

Definizione 16.2 (Classe P)**.** *è la classe dei problemi risolvibili in tempo polinomiale nella dimensione n dell'istanza di ingresso*