Algoritmi e strutture dati

Algoritmi

Un **algoritmo** è una sequenza di istruzioni per risolvere un problema computazionale. Le istruzioni che vengono fornite sono precise, agiscono su **input** e producono un **output**, e sono **elementari**; questo vuol dire che sono singolarmente interpretabili ed eseguibili.

Problema computazionale

Un problema computazionale presenta in sé un **input** e un **output**. L'input è la collezione di dati che viene fornita all'algoritmo. Da questo, l'algoritmo esegue una serie di operazioni, per poi fornire un **output**.

Es. Ordinamento di numeri interi

Input: array o vettori di numeri interi (può essere anche vuoto)
Output: vettore di n elementi

Come scegliere un algoritmo?

Un algoritmo deve essere scelto in base a quattro criteri:

1. **Efficienza**: l'algoritmo deve essere efficiente in termini di tempo e spazio. Deve essere in grado di elaborare grandi quantità di dati, entro un tempo ragionevole, ed usare una *quantità ragionevole di memoria*
2. **Correttezza**: l'algoritmo deve produrre un output corretto per tutti gli input possibili. Deve funzionare in modo coerente e affidabile
3. **Facilità d'implementazione**: dovrebbe essere possibile codificare l'algoritmo in modo **semplice e diretto** in un linguaggio di programmazione, senza dover scrivere troppo codice o fare uso di funzionalità tecniche particolari
4. **Adattabilità**: l'algoritmo deve essere in grado di adattarsi alle diverse esigenze e situazioni, come gestire input diversi, e funzionare in varie circostanze senza essere rigido o limitato.

Il "tempo" di esecuzione

Il tempo di esecuzione di un algoritmo non è misurato in secondi, come si potrebbe pensare. Invece, varia a seconda della quantità di dati in input, e il numero di espressioni eseguite.
Possiamo quindi considerare, dato una *lunghezza n fissata*, un tempo **peggiore**, un tempo **migliore** e un tempo **medio**.

- Peggiore: il caso in cui l'algoritmo ci mette più tempo possibile ad eseguire le espressioni
- Migliore: il caso in cui l'algoritmo ci mette meno tempo possibile ad eseguire le espressioni (N.B. : no vettori vuoti!)
- Medio: tempo atteso di esecuzione di un algoritmo, che **non equivale alla media tra i due tempi**, ma alla divisione tra la probabilità delle singole espressioni, e il tempo che ognuna di esse ci impiega.

Pseudocodice

In modo da facilitare la scrittura degli algoritmi, usiamo il **pseudocodice**, un "linguaggio" a metà tra il linguaggio naturale, e un linguaggio di programmazione. In questo linguaggio, non è opportuno preoccuparsi delle regole di sintassi dei vari linguaggi di programmazione, ma è più importante la **logica**.
Le parole chiave che vengono usate sono le stesse rispetto a Java e a Visual Basic, con una differenza sull'intestazione.

Pseudocodice
```
Int RS(int v[], int k)
Pos = 1
While(v[pos] ≠ k) And (pos ≤ length(v))
   Pos++
Wend
If (pos > length(v)) Then
   Return (-1)
Else Return (Pos)
End If
```

Java
```
Public static int method{
Int v[] new int [];
Int k;
Int pos = 1;
While(v[pos] != k && pos <= v.length() ){
Pos++
}
If (pos > v.length()){
Return -1;
Else
Return pos;
}
}
```

Visual Basic
```
Public Sub RS
Dim v(1 to n) as Integer, k as
Integer, Pos as Integer
Pos = 1
While v(pos) =! K And pos < Ubound(v)
Pos = Pos + 1
Wend
If pos > Ubound(v) Then
Return -1
Else
Return pos
End If
End Sub
```

È sempre importante immettere una **condizione di uscita** in un ciclo del pseudocodice, in modo da non cadere in un ciclo infinito.

Cicli

I cicli, come nella programmazione normale, vengono usati anche nel pseudocodice, con una differenza nell'uso:

- **For**: solitamente usato per i numeri
- **While**: non usato per i numeri, usato se **le istruzioni all'interno del ciclo possono non essere eseguite**.
- **Do-while**: non usato per i numeri, usato se l'istruzione presentata deve comunque, in ogni caso, essere eseguita almeno una volta all'inizio.

Tempo di esecuzione

Il tempo di esecuzione varia in base ai dati in input forniti, e al numero di istruzioni presenti. Per questo, quando si misura il tempo di esecuzione di un algoritmo, è importante fare l'uso della **logica**.
Non è importante vedere quali siano precisamente le istruzioni presenti all'interno dell'algoritmo, ma semplicemente se ci sono presenti dei cicli che potrebbero allungare il tempo di esecuzione, e il tempo di esecuzione delle singole operazioni (*es. Un'operazione di assegnamento di una variabile ci impiegherà tempo 1*).

Es.

Pseudocodice	**Tempo di esecuzione**
`Int RS(int v[], int k)`	*Intestazione*
`Pos = 1`	C1 1
`While(v[pos] ≠ k) And (pos ≤ length(v))`	C2 T(w) + 1
`   Pos++`	C3 T(w)
`Wend`	-
`If (pos > length(v)) Then`	C4 1
`   Return (-1)`	C5 T(if)
`Else Return (Pos)`	C6 F(if)
`End If`	

Nel caso ci siano dei cicli, per ora introduciamo delle variabili. In questo caso, per esempio, le istruzioni all'interno del while vengono eseguite soltanto quando l'espressione del while è **vera**; per questo, segniamo **T(w).** Nel caso che un'espressione venga eseguita quando invece il risultato deve essere false, allora segneremo *F(w).* Più generalmente:

$$T(variabile) \text{ per il vero, } F(variabile) \text{ per il falso}$$

Per contare il totale tempo per eseguire le istruzioni, semplicemente sommiamo insieme tutte le espressioni che sono all'interno del nostro tempo di esecuzione. Nel caso dell'esempio:
$T(n) = C1 + C2 *(T(w) + 1) + C3 * (T(w))+ C4 + C5 * (T(if)) + C6 * (F(if))$

Si sommano quindi, senza moltiplicare a niente, **tutte le istruzioni che vengono eseguite una sola volta**. Per quelle che devono essere eseguite più volte, invece, bisogna moltiplicarlo **al valore della variabile scelto**.

Per una corretta analisi, dobbiamo verificare:

- **Tempo migliore**: il tempo miniere di esecuzione, che non vuol dire quando l'input è vuoto. Semplicemente, significa impostare i valori delle variabili in vero e/o falso, in modo che esca un caso in cui il tempo di esecuzione è il minore possibile
- **Tempo peggiore**: anche se sembra inutile, è opportuno verificare anche il caso in cui il tempo di esecuzione sia il più lungo possibile, quindi impostare le variabili in modo che vengano eseguite il più possibile
- **Tempo medio**: non vuol dire fare la media tra i due tempi, ma *probabilità dei casi/tempo totale*.

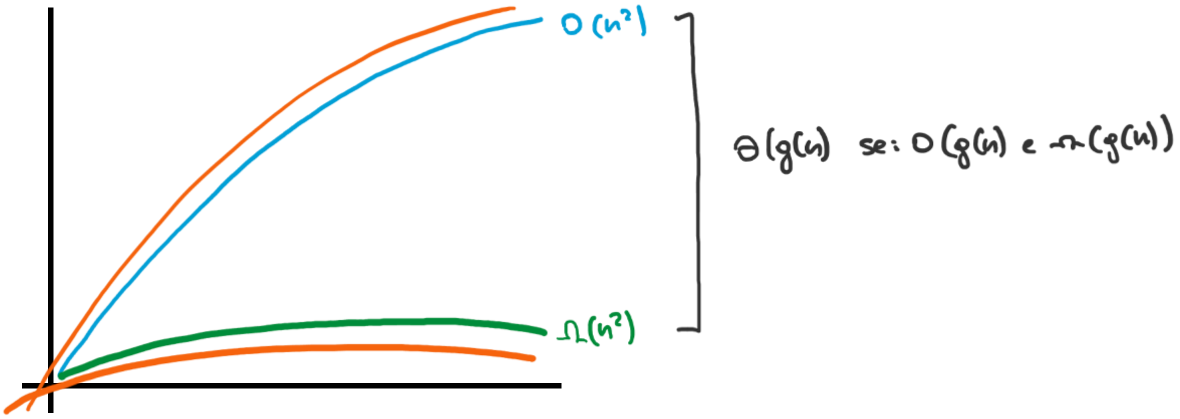
Analisi matematica

Possiamo rappresentare un tempo migliore o peggiore in base a funzioni matematiche.
Rappresntiamo il **limite grande** con omega (quindi il **tempo migliore**), mentre il **limite piccolo** con una O (**tempo peggiore**).

$O(g(n)) = \{ f(n) \mid \text{esiste } n_0 > 0, c > 0 \text{ tale che: } f(n) \leq c * g(n), \text{ per ogni } n \geq n_0 \}$: piccolo

$\Omega(g(n)) = \{ f(n) \mid \text{esiste } n_0 \geq 0, c > 0 \text{ tale che } c * g(n) \leq f(n), \text{ per ogni } n \geq n_0 \}$: grande

Questi due possono essere tracciati su un grafico come due funzioni che limitano superiormente e inferiormente la funzione dell'algoritmo.



Alcuni esempi di algoritmi

Ricerca dicotomica

La ricerca dicotomica è stata vista anche a Programmazione 1 sotto un nome diverso. Si tratta di un metodo di "divisione a metà" di un array ricorsivamente.

1. Si trova la metà dell'array presentato, e chiamiamolo **media**.
2. Si divide a metà l'array nella posizione della media.
    a. Se il numero da trovare è **minore** rispetto alla **media** (n < media), allora dobbiamo **prendere solo la metà di sinistra**.
    b. Se il numero da trovare è **maggiore** rispetto alla **media** (n > media), allora dobbiamo **prendere solo la metà di destra**.
    c. Se il numero da trovare è la **media**, allora abbiamo trovato il numero.
3. *Nel caso non soddisfiamo l'espressione 2C, allora ripetere ricorsivamente dal punto 1 fino a quando non si trova il risultato scelto.*

```
Int DIC(int v[], int k)
Sx = 1, dx = length(v)
M = (dx+sx) / 2
While (v[m] ≠ k)
      If v[m] > k Then
            Dx = m - 1
      Else
            Sx = m + 1
      End If
Wend
If sx ≤ dx
      Return (m)
Else
      Return (-1)
End If
```

C1, C2
C3
c4(tw+1)
c5(tw)
c6(tif i)
c7(fif i)
c8(tw)
c9
c10*t(if)2
c11 * f(if)2

$T_n = (C_1+C_2+C_3+C_9) + (C_4+C_3+C_8+C_5)t_w + C_6\, t_{if}\, 1 + C_7\, f_{if}\, 1 + C_{10}\, t_{if_2} + C_{11}\, f_{if}\, 1 = \Theta(n\log n)$

T_m = il numero da cercare è a metà $t_w = 0 \rightarrow v[\frac{n}{2}] = k \quad t_w = \frac{n}{2}$

$t_m(n) = (C_1+C_2+C_3+C_9)\cdot 1 + C_4 + C_{10} = 6 = \Omega(n) \rightarrow$ costante

T_p = k non esiste $k \notin v[\,]\ t_w \quad t_w = \log(\frac{n}{2})$

$t_p(n) = C_1 + C_2(n+1) + C_3\cdot n + C_4 + C_5 \approx (C_2+C_3)n + (C_1+C_2+C_4+C_5) \approx$

$\approx O(\log n)$

Cerca un carattere all'interno di un array

Possiamo cercare gli elementi di un secondo array in un primo array, e contare quante volte un elemento del secondo è presente nel primo.

1. Si imposta il contatore a zero, e si scorre l'array v2; per ogni elemento, si esegue la ricerca nello stesso indice dell'array v1
2. per cercare il numero, usa **while**: se l'elemento v2[i] è uguale a v1[j] allora ok, se no passa all'elemento successivo fino a quando non trova una corrispondenza
3. se l'indice non supera la lunghezza, significa che l'elemento di v2[i] è presente in v1[j] quindi **aumentiamo il contatore**
4. alla fine del ciclo, restituisce il valore che contiene quanti elementi di v2 sono in v1.

```
Int Cerca(v1[], v2[])
Contatore = 0
For i = 1 to length(v2)
      j = 1
      While (v2[i] ≠ v1[j] and j ≤ length(v1)
            j++
      Wend
      If i ≤ length(v1)
            contatore++
      End If
End For
Return(contatore)
```

C*1
C*n
C*n
2C*Σ(i=1, n) t(w)i
''
''
C*n
C*t(if)
''
''
C * 1

$T(n) = 2c + 3cn + C$

$T_{migl} = t_{w_i} = 0,\ \forall i = 1 \dots n$

$t_m(n) = 2c + 3cn + Cn + 0 = \Omega(n)$

T_{pegg} = nessun elemento di V_2 è in V_1 $(t_{w_i} = n)$

$t_p(n) = 2c + 3cn + C\cdot 0 + 2c\sum n =$

$= 2c + 3cn + 2cn^2 = \Theta(n^2)$

$T_{medio} = t_{w_i} = n^2$

$t_{med}(n) = \Theta(n^2)$

Somma binaria

Questo algoritmo esegue la somma binaria tra due array (formati da zeri e uno), e salva il risultato in un terzo array. L'LSB è N, mentre il MSB è 1.

```
Somma(A[n], B[n], C[n+1])
Riporto = 0
For i = n down to 1
      C[i+1]= A[i] + B[i] + riporto
      If C[i+1] ≤ 1 Then
            riporto = 0
      else
            riporto = 1
            C[i+1] = C[i+1] - 2
      End If
C[i] = riporto
End For
```

C*1
C*n
C*n
C*n
C*if
''
C*F(if)
C*F(if)
//
//
C * 1

$T(n) = 2(C\cdot 1) + 3cn + C\cdot t_{IF} + 2(f_{IF})$

$T_m(n)$ = if è sempre vero, $T_{if} = n$

$T_m(n) = 2c + 3cn + cn = 2c + 4cn = \Omega(n)$

$T_p(n)$ = if è sempre falso

$T_p(n) = 2c + 3cn + 2cn = 2c + 5cn = O(n)$

$T_{medio} = \Theta(n)$

Divisione in due per diventare pari

L'algoritmo è un'implementazione del calcolo del logaritmo in base 2 di n. Il loop "while" divide continuamente n per 2, e conteggia il numero di iterazioni necessarie per portare n a un numero pari, che è equivalente alla posizione del bit più significativo di n. Il valore finale restituito da T rappresenta la posizione di tale bit.

Questo può essere utile per determinare quante volte un valore deve essere diviso per 2 per diventare pari, o per determinare la dimensione di un array necessaria per contenere un certo numero di elementi.

```
Int f_y(int n)
Z = n
T = 0
While z > 0
      x = z MOD 2
      z = z div 2
      If x == 0
            For i = 1 to n
                  t++
            End For
      End If
Wend
Return(t)
```

C*1
C*1
C*T(w)
C*T(w)
C*t(w)
C*t(w)
C*n*t(if)
C*n*t(if)
//
//
C * 1

$t(n) = 3c + 4ct_w + 2cnt_{if}$

$t_w = \log n$

$t_{if} = \log n$

t_m = if mai vero, n = sequenza di bit a 1

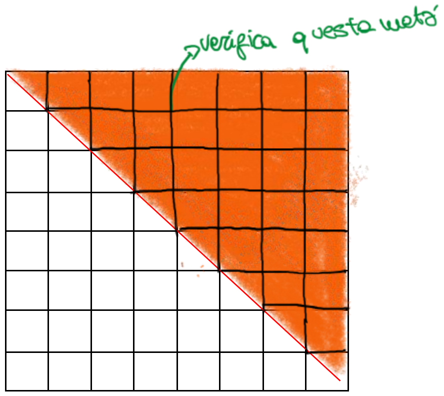
$3c + 4c(\log n) = \Omega(\log n)$

t_p = 1 seguito da zeri

$3c + 4c\log n + 2c(n\log n) = O(n\log n)$

$t_{medio}: O(n \log n)$

Matrice simmetrica



Questo algoritmo determina se una matrice M[n,n] è simmetrica o no; per farlo, non analizza l'intera matrice, ma soltanto una parte della matrice, divisa dalla "bisettrice".

In particolare, verifica se **M[a,b] == M[b,a]**; se falso, passa alla prossima casella, fino a quando non finisce la matrice.

```
Boolean simmetrica(int M[n,n])
B = 1, simm = true
While (simm == true) and (b < n)
      a = b+1
      While (simm == true) and (M[a,b] == M[b,a])
            a++
      Wend
      If a > n
            b++
      else
            Simm = false
      end if
Wend
Return(Simm)
```

2c
C*t(w1)
C*T(w1)
C*T(w2)
C*t(w2)
//
C*t(w1)
C*t(if)
//
c*f(if)
//
//
C

$T(n,n) = 3c + 3ct_{w1} + 2ct_{w2} + C \cdot t_{if} + C \cdot f_{if}$

$T_m = t_{w_1} = 1,\ t_{w_2} = 0,\ t_{if} = 0,\ f_{if} = 1$

$3c + 3c + 2\cancel{c}\,0 + \cancel{c}\,0 + c \cdot 1 = \Omega(1)$

$T_p(n)$ = M simmetrica, $t_{w_1} = n-1$, $t_{w_2} = \sum_{i=1}^{n-1} i$, $t_{if} = n-1$, $f_{if} = 0$

$4c + 3c(n-1) + 2c\frac{(n-1)(n)}{2} = O(n^2)$

$T_{medio} = O(n^2)$

Algoritmi di ordinamento dei numeri interi

Per i numeri interi, possiamo usare vari metodi. Uno di questi è il **bubble sort** (se B > A allora va bene, allora A > B bisogna scambiarlo) - questo metodo continua imperterrito.

Selection sort

Il selection sort ci permette di ordinare un array di numeri interi. L'algoritmo confronta ogni elemento dell'array con tutti gli altri elementi per trovare l'elemento più piccolo, e lo scambia con l'elemento che dovrebbe essere nella posizione in cui si trova. Questo ciclo viene ripetuto fino a quando non è stato ordinato. In pratica:

1. Si considera l'elemento più piccolo dell'array, e **si colloca nella prima posizione**
2. Si cerca l'elemento più piccolo rimanente nell'array e lo si colloca nella seconda posizione
3. Si continua a cercare l'elemento più piccolo e a collocarlo in ordine successivo, fino a quando l'intero array non è ordinato.

C(n-1)
C(n-1)
(n*(n-1))/2
(n*(n-1))/2
c*if

Questo metodo usa una **variabile di appoggio** in modo da salvare una variabile dell'array.

Insertion sort

Questo algoritmo ordina un array di elementi attraverso l'inserimento di ciascun elemento al posto giusto all'interno di una porzione già ordinata della lista.

1. Si parte dall'elemento di indice 1 dell'array, che si considera **già ordinato**
2. si prende l'elemento successivo dell'array, e si confronta con quelli precedenti finché non si trova il posto giusto in cui inserirlo.
   - Se il numero è maggiore del precedente, allora non scambia
   - se il numero è minore del precedente, allora scambia
3. Si continua fino ad arrivare all'ultimo elemento
4. L'array sarà ordinato.

Questo array porta ad avere sempre la parte sinistra dell'array ordinato, un numero k che è da controllare, e la parte destra dell'array da ordinare.

```
void insertionSort(v[])
For i = 2 to length(v)
      k = v[i]
      j = i-1
      While (K < V[j] and j > 0)
            V[j+1] = V[j]
            j--
      Wend
      V[j+1]=k
End For
```

C(n-1)
C(n-1)
C(n-1)
C* E(2)(n) tw(i)
"
"
C(n-1)

Ordinato	K	Disordinato

$t_{INS}(n) = 4c(n-1) + 3\sum_{i=2}^{n} t_{wI} \simeq O(n^2)$

$t_m(n) = t_{w_i} = 0 \quad v_i = 2..n \quad t_m(n) = 4c(n-1) \approx \Theta(n)$

v è ordinato al contrario

$t_p(n) = 4c(n-1) + 3\sum_{i=2}^{n} t_{w_i} = 4c(n-1) + 3c\left(\sum_{i=1}^{n-1}\right)\cdot 1 = 4c(n-1) + \frac{3c(n-1)n}{2} \approx \Theta(n^2)$

$t_{medio} = \frac{1}{2} t_p$

Algoritmi ricorsivi

Per algoritmi ricorsivi si intende tutta quella branca di algoritmi che implementano in se stessi una parte detta **caso base**, che fa interrompere l'esecuzione dell'algoritmo, e un caso **ricorsivo**, ovvero dove il metodo richiama se stesso più volte. Tutti gli algoritmi per ora sono stati implementati in maniera **iterativa**, ovvero senza un richiamo del metodo stesso all'interno del metodo.

Potenza di un numero

Questo algoritmo permette di calcolare la potenza di un intero elevato alla n-esima potenza, usando la tecnica della ricorsione.

La funzione riceve **due input**: se n è uguale a zero, la funzione restituisce 1; altrimenti, calcola il valore di a elevato alla potenza di n-1, usando la stessa funzione in modo ricorsivo. Il risultato di questa operazione viene quindi moltiplicato per a per ottenere la potenza di n.

```
Int potenza(int a, int n) //a > 0, n ≥ 0
If n == 0
      return(1)
Else
      r1 = a*potenza(a, n-1)
      r2 = a * r1
      return(risultato)
End if
```

C1 * t(if)
C1 * t(if)
//
C1*f(if)
C1*f(if)
C1*f(if)

$$\begin{cases} T(n) = 2ct \cdot T(n-1) & n>0 \\ 2c & n=0 \end{cases}$$

$T(n) = 2ct(if) + 3cf(if) = \Theta(1)$

$T_m(n) = n=0$

$2ct(if) = \Omega(1)$

$T_p(n) =$ tutte le esecuzioni $= \Theta(1)$

Sequenza di Fibonacci

La serie di Fibonacci è una sequenza matematica dove ogni numero è la somma dei due numeri precedenti della sequenza, quindi il primo e il secondo numero sono **1**, e ogni numero successivo è la **somma dei due precedenti** (es. *1, 1, 2, 3, 5, 8, 13, 21, 34, 55*).

Per valori n maggiori di 2, l'algoritmo usa la ricorsione per calcolare il valore; calcola il valore di Fibonacci(n-1) e Fibonacci(n-2), per poi sommarli per ottenere il valore finale Fibonacci(n).

```
Int Fibonacci(int n)
If n ≤ 2
      Return(1)
Else
      ris = fib(n-1) + fib(n-2)
      return(ris)
End If
```

C
C1 * t(if)
//
C1*f(if)
C1*f(if)
//

$T(n) = c + C \cdot t_{if} + 2c \cdot f_{if} = \Theta(2^n)$

$T_m(n) = n \leq 2, \; t_{if} = 1$

$T(n) = c + C \cdot t_{if} = \Omega(2^n)$

$T_p(n) = n > 2, \; f_{if} = 1, \; t_{if} = 0$

$T(n) = c + 2c \cdot f_{if} = O(2^n)$

Conta lettera

Questo algoritmo permette di calcolare quante volte una certa lettera viene ripetuta in un array. In particolare, questo è specializzato su **Z**, e implementa la ricorsione.

```
Int ContaZ(v[], int F)
If F ≤ 0 Then
      Return(0)
Else
      If V[F] == 'Z' Then
            Ris1 = 1
      else
            ris1 = 0
      End If
      ris2 = ContaZ(v[], F-1)
      Tot = Ris1 + Ris2
      Return(tot)
```

C
C1 * t(if)
//
C1*f(if)
C1*f(if)*t(if2)
//
C*f(if)*f(if2)
//
C
C
C

$T(n) = k \cdot 5c + T(n-k) = n \cdot 5c + T(n-n) =$

$= 5c + 2c \approx \Theta(n)$

$T_m(n) = F \leq 0$

$T_m(n) = c \cdot c \cdot t_{if} \approx \Omega(n)$

$T_p(n) = F > 0$

$T_p(n) \approx O(n)$

```
End If
```

Conta il successore

L'algoritmo conta il numero di sequenze di elementi consecutivi in un array, in cui ogni elemento è il successore dell'elemento precedente.

```
Int contaSucc(v[], int f)
If f < 2 then
      Return(0)
Else
      If v[f] == v[f-1]+1 then
            Ris1 = 1
      Else
            Ris1 = 0
      End if
      Ris2 = contaSucc(v[], f-1)
      Tot = Ris1 + Ris2
      Return(Tot)
End if
```

C
C1 * t(if)
//
C1*f(if)
C1*f(if)*t(if2)
//
C*f(if)*f(if2)
//
C
C
C

$T(n) = k \cdot 5c + T(n-k) = n \cdot 5c + T(n-n) =$
$= n \cdot 5c + T(0) = n \cdot 5c + 2c \approx \Theta(n)$
↳ se già ordinato
se no $O(n^2)$

$T_p(n) \approx O(n)$ $T_m(n) \approx \Omega(n)$

Massimo comune divisore

L'algoritmo MCD calcola il massimo comune divisore di un dato numero usando **l'algoritmo di Euclide**: dati due numeri naturali a e b, con a > b: se b è un divisore esatto di a, b è il massimo comun divisore; altrimenti, detto r il resto della divisione tra a e b, il **MCD** tra a e b è uguale al **MCD tra b e r**.

```
Mcd(0,n) = n
Mcd(m,n) = Mcd(m,n-m)
Not Mcd(m,n) = Mcd(n,m)

Int Mcd(int n, int m)
If m > n Then
      App = m        *
      M = n
      N = app
End If
If m == 0 Then
      Return(n)
Else
      Ris = Mcd(m, n-m)
      Return(Ris)
End If
```

*
C
C1 * t(if)
C1 * t(if)
C1*t(if)
//
C
C*t(if)
//
C*f(if)
C*f(if)
//

$T(n) = 2kc + 4kc(t_{if}) + 2kc(f_{if}) \approx \Theta(n)$
$T_p(n) = O(n)$ $T_m(n) = \Omega(n)$

Stampa ricorsiva

L'algoritmo stampa ricorsivamente ogni singola lettera presente in un array. Ricorsivamente, questo sarebbe stato implementato con un ciclo for.

```
Void f_y(A[], int i)
If i ≤ length(A)
      Print(A[i])
      F_y(A, i+1)
End If
```

C
C*t(if)
C * t(if)
//

$T(n) = kc + 2kc(t_{if}) \approx \Theta(n)$
T_p = $i = length(A)$
$T_p(n) = kc + 2kc(t_{if}) = O(n)$
T_m : $i = 1$

$T_m(n) = c + 2c(t_{if}) = \Omega(n)$

Somma

Questo algoritmo calcola la somma degli elementi nell'array V dal primo indice I al valore dell'ultimo indice F.
Viene usata la tecnica ricorsiva per **suddividere la somma totale in somme più piccole** tra gli elementi dell'array; gli elementi prima e dopo l'indice attuale vengono aggiunti alla somma attuale per produrre la somma parziale.
Questo procedimento viene ripetuto ricorsivamente fino a quando non viene raggiunto il caso base dell'indice finale uguale all'indice iniziale.
Se l'indice iniziale supera l'indice finale, allora l'algoritmo restituisce 0 perché non ci sono elementi.

```
Int somma(v[], int I, int F)
If I == F
        Return(V[I])
End If
If I > F
        Return(0)
Else
        R1 = V[I] + V[F]
        R2 = somma(V[], I+1, F-1)
        Return(R1+R2)
End If
```

C
C*t(if)
//
C
C*t(if2)
//
C*f(if2)
C*f(if2)
C*f(if2)
//

$$T(n) = \begin{cases} \Theta(1) & n=1, n=0 \\ 4c + T(n-2) \end{cases}$$

$$4c + [4c + T(n-4)] = 2 \cdot 4c + T(n-4) = 2 \cdot 4c + [4c + T(n-4)] = k \cdot 4c + T(n-2k) = k \cdot 4c + T(2n-2k) = \frac{n}{2} 4c + T(0) = 2nc + 3c \simeq \Theta(n)$$

$T_p(n) = O(n)$ $T_m(n) = I = F = \Omega(1)$

Parola palindroma

Questo algoritmo verifica se una parola è palindroma o meno, usando due indici inizio e fine, e richiamandosi ricorsivamente ogni volta, facendo finire la verifica in mezzo.

```
Boolean Pal(A[], int I, int F)
If I ≥ F Then
        Return(True)
Else
        If A[I] ≠ A[F] Then
                Return(False)
        Else
                Ris = Pal(A[], I+1, F-1)
                Return(Ris)
        End If
End If
```

C
C*t(if)
//
C*F(if)
C*F(if)*T(if2)
//
C*F(if)*F(if2)
C*F(if)*F(if2)
//
//

$$T(N) = \begin{cases} 3c & n \le 1 \\ 2c + c \cdot T(if) + c \cdot F(if) + F(if) \cdot T(n-2) \end{cases}$$

$T_m(n) = A[1] \neq A[n]$ $n = length(A)$

$2c + c + 0 + 0 = 3c = \Omega(1)$

$T_p(n) = 2c + 0 + c + T(n-2) = 3c + T(n-2) = 3c + T(n-2) = 3c + [3c + T(n-4)] = 2 \cdot 3c + T(n-4) = 2 \cdot 3c + [3c + T(n-6)] = 3 \cdot 3c + T(n-6)$

$\vdots$

$k \cdot 3c + T(n-2k)$

$$T(*) = \frac{n}{2} \cdot 3c + T\left(n - 2\frac{n}{2}\right) - \frac{n}{2} \cdot 3c + T(0) = \frac{3}{2} nc + 2c = O(n)$$

Massimo in algoritmo

Questo algoritmo permette di calcolare il massimo valore presente in un array, usando un metodo ricorsivo. Usando un valore di contatore per l'inizio, verifica per ogni singolo elemento dell'array quale sia il massimo.

```
Int Max(A[], int I)
If I == length(A) Then
        Return(I)
Else
        Max_R = Max(A[], I+1)
        If A[Max_R] > A[I] Then
                Return (Max_R)
        Else
                Return(I)
        End If
End If
```

Divide et impera

La logica "divide et impera" consiste nel risolvere un problema più grande suddividendolo in sotto-problemi più piccoli, risolvere questi sotto-problemi e quindi **combinare** i risultati di questi problemi, in modo da ottenere la soluzione al principale. Ha tre fasi:

1. **Divide**: divide il problema principale in sotto-problemi
2. **Impera**: risolve ciascuno di questi sotto-problemi
3. **Combina**: combina i risultati di tutti questi sotto-problemi per risolvere il problema più grande.

Gli algoritmi con logica **divide et impera** possono essere risolti usando tre metodi di risoluzione:

1. **Metodo di suddivisione**: prevede la suddivisione del problema in sottoproblemi più piccoli, che vengono risolti in modo ricorsivo e poi combinati per ottenere la soluzione finale.
    a. *Suddivisione*
    b. *Risoluzione in modo ricorsivo*
    c. *Combinazione*

2. **Metodo esperto**: prevede di utilizzare un esperto, o una funzione di valutazione, per valutare la soluzione di un sottoproblema e quindi utilizzare questa informazione per prendere decisioni sul modo migliore per risolvere il problema completo.
    a. *Identificare il problema*
    b. *Identificare l'esperto (può essere algoritmo o funzione di valutazione)*
    c. *Generare una soluzione iniziale*
    d. *Valutazione della soluzione iniziale*
    e. *Applicazione delle modifiche e valutazione della soluzione migliorata*
    f. *Ripetere fino alla soluzione finale*
3. **Metodo di conquista**: prevede l'identificazione di una **soluzione "base"** per il problema completo, e la sua espansione per risolvere i sottoproblemi. In questo modo, l'algoritmo lavora **iterativamente** suddividendo e conquistando, cercando di raggiungere la soluzione finale. Può sembrare simile al metodo di conquista, ma la differenza è che per **ogni sottoproblema si cerca di raggiungere la soluzione di base.**
    a. *Identificare il problema completo*
    b. *Identificare una soluzione base*
    c. *Suddividere il problema in sottoproblemi più piccoli e gestibili*
    d. *Risolvere i sottoproblemi, usando di nuovo la stessa tecnica*
    e. *Combinare le soluzioni dei sottoproblemi*
    f. *Verificare la soluzione finale*
    g. *Ottimizzare la soluzione*

Merge Sort

Questo algoritmo suddivide la lista di elementi da ordinare in due parti uguali, ordina separatamente i due sottovettori generati, e poi li unisce per formare una **lista ordinata finale**. Il processo di divisione, ordinamento e unione continua fino a quando la lista non può più essere divisa, ovvero quando c'è un solo elemento. (O(n log n))

```
Void mergeSort(A[], int I, int F)
If I < F Then
      n = (I+F) div 2
      mergeSort(A[], I, n) //sx
      mergeSort(A[], n+1, F) //dx
      merge(A[], I, n, F)
End If

Void merge(A[], int psx, int pm, int pdx)
I1 = psx, I2 = pm+1, IB = psx
While I1 ≤ pm and I2 ≤ pdx
      If A[I1] ≤ A[I2] Then
            B[IB] = A[I1], IB++, I1++
      Else
            B[IB] = A[I2], IB++, I2++
      End If
Wend
While I1 ≤ pm
      B[IB] = A[I1]
      IB++, I1++
Wend
While I2 ≤ pdx
      B[IB] = A[I2]
      IB++, I2++
Wend
For IB = psx to pdx
      A[IB] = B[IB]
End For
```

4C

3c

3c

Tre sezioni:
- DIVIDE
- IMPERA
- COMBINA

$c = \Theta(1)$

$D(n) + I(n) + C(n)$ $\quad n=1$

$c + 2t_{n_s}(n) + \Theta(n)$

come un ciclo for?

$tw_1 + tw_2 + tw_3 = n$

$4Ctw_1 + 3Ctw_2 + 3Ctw_2 \approx \Theta(n)$

Un algoritmo divide et impera può essere usato con tre metodi:

① metodo di suddivisione

$T_{ms} = \begin{cases} 2 & n=1 \\ 2t_{ms}\left(\frac{n}{2}\right)+n & n\log n \end{cases}$ $\qquad k = n = 2^k$ $\quad k=1 \to 2^k = 2$

supp k $\quad T(n=2^k) = 2^k \log 2^k$

dim $k+1$ $\quad T(2^{k+1}) = 2^{k+1} \log 2^{k+1}$

$T(2^{k+1}) = 2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1} = 2T(2^k) + 2^{k+1} = 2(2^k \log 2^k) + 2^{k+1} = 2^{k+1}\log 2^k + 2^{k+1} = 2^{k+1}((\log 2^k)+1) =$

$= 2^{k+1}(\log 2^k + \log_2 2) = 2^{k+1}(\log 2^k \cdot 2) = 2^{k+1}\log 2^{k+1}$

$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left[2t\left(\frac{n}{2}\right) + \frac{n}{2}\right] + n = 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n = 2^2\left[2t\left(\frac{n}{2}\right) + \left(\frac{n}{2^2}\right)\right] + 2n = 2^3T\left(\frac{n}{2^3}\right) + n + 2n = 2^kT\left(\frac{n}{2^n} + kn\right) =$

2 Metodo dell'esperto

$$\begin{cases} T(n) = aT\left(\frac{n}{b}\right) + f(n) & a \geq 1 \\ T(r) = \Theta(1) & B \geq 1 \end{cases}$$

O(1) = caso base
A = n. sottoproblemi
n/b = dimensione sottoproblemi

(a = 2, b = 2)

T(0) = Θ(1)
T(n) = 2T(n/2) + n
$n^{logb(a)}$

CONFRONTARE LA PARTE RICORSIVA E ITERATIVA

1. Ric > It
   $f(n) = O(n^{\log_b a - \varepsilon})$, $\varepsilon > 0$
2. $F(n) = \Theta(n^{\log_b a}) \rightarrow T(n) = \Theta(\log n \cdot n^{\log_b a})$
3. $F(n) = \Omega(n^{\log_b a + \varepsilon}) \rightarrow T(n) = \Theta(f(n) \cdot \log n)$
   $\hookrightarrow a \cdot f(\frac{n}{b}) \leq k f(n)$
   $T(n) = \Theta(f(n))$

Tempo divisione: O(1)
Combina: O(n)

Conta coppie

Questo algoritmo conta il numero di coppie di elementi consecutivi all'interno di un array S che sono uguali tra loro. Lo fa **dividendo l'array in due parti**, e richiamando se stesso ricorsivamente sulla parte sinistra e destra dell'array diviso, finchè **l'intervallo di ricerca è ridotto a un singolo elemento**.

Nella fase di conquista successivamente, verifica se ci sono coppie uguali tra l'ultimo elemento della parte sinistra, e il primo dell'ultima parte dell'array (in posizione m), o tra gli elementi in posizione precedente e successiva ad esso.

```
Int contaCoppia(S[], int I, int F)
If I ≥ F then
      Return(0)
Else
      Misp(I+F) div 2
End If
Sx = contaCoppia(S[], I, m)
Dx = contaCoppia(S[], n+1, F)
If S[m] == S[m+1] and S[m+1] == S[m+2] then
      Tot++
End If
If S[m-1] == S[m] and S[m] == S[m+1] then
      Tot++
End If
Return(tot)
```

$$T(n) = \begin{cases} 2c & n \leq 1 \\ c + 2T\left(\frac{n}{2}\right) + 5c & n > 1 \end{cases}$$

metodo dell'esperto

$f(n) \leftrightarrow n^{\log_b a} \Rightarrow 6c \leftrightarrow n^{\log_2 2} = n^1 = n$

$\hookrightarrow$ caso 1: $f(n) = O(n^{\log_b a - \varepsilon}) = O(n^{1-\varepsilon})$, $\varepsilon > 0$

$\varepsilon = 1$, $\varepsilon = \frac{1}{2}$, $6c = O(n^{1-\frac{1}{2}})$ $T(n) = \Theta(n)$

Massimo

Questo algoritmo divide ricorsivamente l'array in due metà, e confronta i valori delle due metà utilizzando la funzione stessa fino a quando non rimane un solo elemento. Poi, individua l'indice del valore massimo tra i due elementi rimanenti e lo restituisce come output dell'intera funzione. La complessità dell'algoritmo è di tipo O(log n).

```
Int max(v[], int I, int F)
If I == F
      Return(I)
Else
      M = (I+F) DIV 2
      MS = Max(v[], I, M)
      MD = Max(v[], M+1, F)
      If v[MS] ≥ v[MD] then
```

```
            Return(MS)
      Else
            Return(MD)
      End If
End If
```

Ricerca dicotomica

La ricerca dicotomica può essere implementata non solo in maniera ricorsiva, ma anche usando il metodo divide et impera.

```
Int RicDic(v[], int I, int F, int K)
If I == F then
      If V[I] == K then
            Return(I)
      Else
            Return(-1)
      End If
Else
      M=RicDic(i++) div 2
End If
If V[m] == k then
      Return(m)
Else If V[m] > k then
      Ris = RicDic(V[], I, m-1, k)
Else
      Ris = RicDic(v[], m+1, F, K)
End If
Return(Ris)
```

Calcola

Questo algoritmo calcola il prodotto di una funzione ricorsiva chiamata su due sottosequenze dell'array, divise a metà. La funzione ricorsiva stessa **calcola la somma degli elementi dell'arrabbiata a partire dalla posizione I fino a F**, **aggiunge 2** e restituisce il risultato. Il tempo di esecuzione è **=(n logn)**.

```
Int calc(A[], int I, int F)
If I == F then
      Return(A[i] + 2)
Else
      M = (I+F) DIV 2
      R1 = calc(A[], I, m)
      R2 = calc(A[], m1, F)
      Tot = R1 * R2
End If
Return(Tot)
```

Valuta

Questo algoritmo cerca se esiste un elemento nell'array A che abbia lo stesso valore dell'indice corrispondente. Usa una tecnica di ricerca binaria, che suddivise l'array in metà ripetutamente, finchè non trova o esclude l'elemento desiderato.
Il tempo di esecuzione dipende dalla dimensione dell'array; l'algoritmo richiede un tempo O(log(n)).

```
Boolean val(A[], I, F)
If I == F Then
      If A[i] == I then
            Return(true)
      Else
            Return(false)
      End If
Else
      M = (I + F) DIV 2
End If
If A[m] == m then
      Return(true)
Else if A[m] < m then
      R = Val(A[], m+1, F)
Else
      R = Val(A[], I, m-1)
End If
Return(R)
```

Quicksort

Un algoritmo quicksort è un algoritmo di tipo divide et impera, che lavora segnando un **pivot**, e ordina ricorsivamente la prima e seconda parte.

Possiamo dividere l'algoritmo in tre parti:

1. **Divide**: si sceglie un elemento dell'array chiamato **pivot**, e si suddivide l'array in due sotto-array: uno contiene *tutti gli elementi minori del pivot*, e l'altro invece contiene *tutti gli elementi maggiori del pivot*
2. **Impera**: si ordina ricorsivamente i due sotto-array
3. **Combina**: niente

Esistono due versioni: la versione di Hoare è instabile, mentre l'algoritmo di Lomuto preserva l'ordine relativo degli elementi che non soddisfano il predicato, mentre l'altro non lo fa.

- **Hoare:** sceglie due indici che attraversano il vettore da entrambe le estremità e si incontrano al centro.

Es.

Si possono prendere ogni elemento come pivot, meno l'ultimo.

1	19	9	5	12	2	7	4	11	2	6	21

pivot → 1 0

pivot >> incremento fino a quando trovo uno più grande TROVATO!

1 F

da qua parto a trovare al uno più piccolo TROVATO

6 < 19, > 19

19 scambio con 6

- **Lomuto**: sceglie l'ultimo elemento come pivot, e lo sposta alla sua posizione corretta nel vettore.

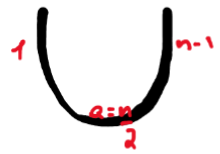
```
Void QS(A[], int I, int F)
If I < F Then
      Partition(A[], I, F)
      QS(A[], I, Q)
      QS(A[], Q+1, F)
End If
```

→ pivot

$\Theta(n)$

```
Int Partition(A[], int In, int Fi)
Pivot = A[In], sx = In-1, dx = Fi+1
While sx < dx
      Do
            Sx++
      while A[sx] < pivot
      End Do
      do
            dx-
      while A[dx] > pivot
      End Do
      If sx < dx then
            App = A[sx]
            A[sx] = A[dx]
            A[dx] = App
Wend
Return(dx)
```

$$\begin{cases} T(n) = \Theta(n) \\ T(1) = \Theta(1) \\ C + T_{QS}(n-1) + \Theta(n) \end{cases}$$



$T_{QS}(n) = 2T_Q\left(\frac{n}{2}\right) + \Theta(n)$

$= T(1) + T(n-1) + \Theta(n)$

$T_{QS}(n) = T_{QS}(q) + T(n-q) + \Theta(n) = 2+3+4+5+\dots+n+n = \sum_{i=1}^{n} i + (n-1) = \frac{n(n+1)}{2} + (n-1) = \Theta(n^2)$

HP. $T_{QS}(n) = O(n^2) \leq cn^2$

$T_{QS}(n) \leq cq^2 + c(n-q)^2 + \Theta(n)$

MIGLIORE:

$\max T_{QS}(n) \leq cq^2 + c(n^2 + q^2 - 2nq) + n$

$1 \leq q \leq n-1$

$\frac{d}{dq}$: $= 2cq^2 + cn^2 + \dots - 2cnq + n$ → $4cq - 2cn$

$T_{QS} = O(n^2)$, $\Omega(n \log n)$

Quick Sort randomizzato

L'algoritmo è una variante del Quick Sort, che usa una particolare tecnica di scelta dei pivot casualizzati, al fine di migliorare le prestazioni della funzione.
Mentre nella variante classica si sceglie un determinato elemento della lista, nella variante randomizzata **viene scelto casualmente dalla lista**; questo può aiutare ad evitare i casi peggiori, che possono verificarsi quando gli elementi sono già ordinati o in ordine diverso.

```
Void QSRand(A[], int I, int F)
If I < F Then
      Q = RandomizedPartition(A[], I, F)
      QSRand(A[], I, Q)
      QSRand(A[], Q+1, F)
End If
```

$T(n) = O(\log n)$

Peggiore: pivot è sempre elemento più piccolo o grande dell'array $= O(n^2)$

Migliore: pivot diviso in due parti uguali $= O(n \log n)$

```
Int RandomizedPartition(A[], int In, int Fi)
RandIndex = Random(In, Fi)
Swap(A[In], A[RandIndex])
Return Partition(A[], In, Fi)
```

1. Prendiamo un numero random tra In e Fi
2. Scambiamo l'elemento al primo indice con quello all'indice random (per non farlo interferire nella partizione)
3. Esegui la partizione

```
Int Random(int min, int max)
Return rand() % (max-min+1) + min
```

rand(): numero casuale tra 0 e RAND_MAX
(Max-min+1) + min: genera all'interno di intervallo

```
Void Swap(int a, int b)
Temp = a
A = b
B = temp
```

```
Int Partition(A[], int In, int Fi)
Pivot = A[In]
Sx = In-1
Dx = Fi + 1
      End Do
      If sx < dx then
            Swap(A[sx], A[dx])
      End If
Wend
Return(dx)
```

Partiziona array in base al pivot scelto.

Se gli indici non si sono incrociati scambia gli elementi alle posizioni

Incrementa sx finchè non trova un elemento ≥ pivot

Decrementa dx finchè non trova un elemento < pivot

Funzione di Ackerman

Questa funzione permette di calcolare la funzione di Ackerman, che viene definita com'è qua a seguito. È un metodo ricorsivo, e fa uso di tre if uno di seguito all'altro, per poter implementare i 3 stati in cui può essere.

$$A(m,n) = \begin{cases} N+1 & \text{se } m = 0 \\ A(m-1, 1) & \text{se } m > 0, n=0 \\ A(m-1, A(m, n-1)) & \text{se } m > 0, n > 0 \end{cases}$$

```
Public int ack(int m, int n)
If m==0 then
      Return(n+1)
End if
If m > 0 and n = 0 then
      Return(A(m-1,1))
End if
If m > 0 and n > 0 then
      Return(A(m-1, A(m,n-1)))
End If
```

C
C*t(if)
//
C
C*t(if2)
//
C
C*t(if3)

$T_n = 3C + 3CT_{if}$

Formula master

$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ a = sotto, b = dim, f(n) = fun

cui gli sottoproblemi nel livello

Conta le cifre

Questo algoritmo conta il numero di cifre in un dato numero in maniera ricorsivo. Se n è minore di 10, la funzione restituisce 1 perché tutti i numeri sotto 10 hanno una sola cifra. Altrimenti, la funzione restituisce 1 più il risultato della funzione chiamata ricorsivamente, con n diviso per 10 come argomento. La funzione continua a chiamare se stessa fino a quando n diventa minore di 10, e quindi restituisce il numero totale di cifre in n.

```
Int contaCifre(int n)
If n < 10 then
      Return 1
Else
      Return 1 + count(n/10)
```

C
C*t(if)
//
C*f(if)
//

$T(n) = C + C \cdot T_{IF} + C \cdot F_{IF} \cdot T\left(\frac{n}{2}\right) = \Theta$

```
End if
```

Funzione ricorsiva

Questo algoritmo permette di calcolare ricorsivamente una funzione f(n) = x+3, ogni volta abbassando la x, fino a quando arriva a 0, dove a quel punto la funzione non viene più calcolata.

```
Int funzione(int x)
If x == 0 then
      Return 5;
Else
      Return(funzione(x-1)+3)
End If
```

C
C*t(if)
//
C*f(if)
//

$T(n) = C + C \cdot T_{if} + C \cdot F_{if} \cdot T(n-1) = \Theta(n)$

Compare two arrays

Questo algoritmo confronta due array di interi, e successivamente riporta quanti elementi del secondo array compaiono nel primo argomento. Ovviamente, se non ci sono elementi in comune, riporta 0.

```
Int confronta(int v1[], int v2[])
Int tot = 0

For i = 0 to length(v2)
      Int cont = 0
      while != v1[cont] && cont < length(v1)
            cont++
      wend
      if cont < v1.length -1 then
            tot++
      End If
Cont = 0
End For
```

C
//
Cn
Cn
Cn*Tw(i)
C*Tw
//
Cn
C*T(if)
//
C*n
//

$T(n) = 2C + 4C_n + \sum_{i=0}^{n} t_{if} + C \cdot t_w + C \cdot n \cdot t_{wi}$

migliore: tutti gli elementi di V_2 sono uguali al primo elemento di V_1

$t_w = 0$

$t_{if} = n$

$+ C \cdot n + C_0 + C \cdot (n-1) = 2C + 4C_n$

peggiore: tutti gli (n) elementi di V_2 sono uguali all'ultimo di V_1

$t_w = n$

$2C + 4C_n + C \cdot n + C \cdot n + C \cdot n^2 = O(n^2)$

medio: $V2[J]$ viene trovato in $V1[\frac{n}{2}]$

$2C + 4C_n + C_n + 2C \sum_{i=1}^{n} \frac{n}{2} = \Theta(n^2)$

Strutture dati

Le strutture dati sono un insieme di dati aggregati; se sono dello stesso tipo, allora si parla di array. Possono essere di tipo **statico**, come gli array, o dinamici, come le liste.

Se di tipo statico, allora alla creazione viene richiesta la dimensione della memoria allocata alla creazione, e successivamente viene allocata *in modo contiguo*. I vantaggi sono un *tempo di accesso ridotto, occupa memoria inutilmente, è impossibile aggiungerci spazio*.

Liste

Le liste sono invece strutture dati di tipo **dinamico**, ovvero dove la memoria è allocata solo quando richiesto, e successivamente deallocata se non utilizzata. Le celle possono essere non contigue; l'unico svantaggio che abbiamo è che *il tempo di accesso è più lungo.*

Abbiamo vari tipi di liste:

- **Dinamica semplice**: il puntatore si chiama **head[L]**, ed è l'indirizzo del primo elemento inserito in L. Si accede al successivo con **next**, e se in next c'è un elemento null, allora la lista è terminata.
- **Dinamica doppia**: aggiunge, oltre a next, anche **prev**, che tiene traccia dell'elemento successivo
- **Dinamica circolare**: possono essere semplici o doppie. La differenza è che, all'ultimo elemento, **il next è head[L]**; questo vuol dire che, teoricamente si può sempre andare avanti senza mai trovare una fine.

Operazioni
Per ogni lista possiamo fare una serie di operazioni:

Search
Questo serve per cercare all'interno di una lista un dato elemento. Dato una lista L e un parametro X, ritorna un valore *p.att* che ritorna la posizione nella lista L dell'elemento X. Returna null se non trova alcun elemento.

```
Search(L, K)
patt = head[L]
While patt ≠ K and patt ≠ null
      patt = patt.next
Wend
Return(patt)
```

Il tempo di esecuzione **dipende dalla posizione dell'elemento**; se l'elemento K si trova all'inizio di L, allora il tempo sarà **costante** (O(1)); altrimenti, sarà **lineare** (O(n)).

Insert
Questo algoritmo serve per inserire un nuovo elemento x all'interno della lista L, all'inizio di essa.

```
Insert(L,x)
X.next = head[L]
If head[L] ≠ null then
      head[L].prev = x
End If
head[L] = x
```

Il tempo di esecuzione è sempre **costante** (O(1)), perché non dipende dalla posizione dell'elemento x nella lista L.

Delete
Questo algoritmo serve per eliminare un elemento x all'interno della lista L..

```
Delete(L,x)
If x.prev ≠ null then
      (x.prev).next = x.next
Else
      head[L] = x.next
End If
If x.next ≠ null then
      (x.next).prev = x.prev
End If
```

Il tempo di esecuzione è sempre **costante** (O(1)), perché non dipende dalla posizione dell'elemento x nella lista L.

Update
Questo modifica il valore già presente.

```
Update(x,k)
X.key = k //valore
```

Successore
Trova il più piccolo dei più grandi di k.

```
Succ(L,K)
      Patt = head[L]
      Psucc = null
      While psucc == null
            If patt.key > k
                  Psucc = patt
            End if
            Patt.next
            Patt.next
      If psucc == null return null
```

Palindromo

Verifica se la sequenza di caratteri all'interno della struttura dati è palindroma.

```
Boolean pal(L)
     P1 = head[L]
     P2 = tail[L]
     Pal = true
     While p1 ≠ p2 and p2.next ≠ p1 and pal == true
          If p1.key ≠ p2.key then
               Pal = false
          Else
               P1 = p1.next
               P2 = p2.prev
          End if
     Wend
     Return pal
```

$T(n) = 4c + 2c\, t_w + ct_{if} + cf_{if}$

Migliore: key della prima ≠ key ultima

$T_w = 0$; $t_m(n) = 4c + 2c + 0 = 7c = \Omega(1)$

Peggiore: $t_w = n/2$

$\mathbf{T_p(n)} = 4 + c_n + c_n = 4c + 2c_n = 0(n)$

Cancella multipla

```
Void canc(L,K)
     Patt = head[L]
     While patt.key == K
          Head[L] = patt.next
          Free(patt)
          Patt = nead[L]
     Wend
     Pprec = patt
     Patt = patt.next
     If patt ≠ null then
          Patt = patt.next
     End if
     While patt ≠ null
          If patt.key == k then
               Pprec.next = patt.next
               Free(patt)
               Patt = pprec.next
          Else
               Pprec = patt
               Patt = patt.next
          End if
     Wend
```

Conta

Conta quanti elementi ci sono all'interno della struttura dati.

```
Conta(patt, k)
     If patt == null then
          Return 0
     Else
          If patt.key == k then
               R = 1
          Else
               R = 0
          End if
          R2 = Conta(patt.next, k )
          Return (r + r2)
```

$$T(n) = \begin{cases} 2c & (n = 0), \\ 4c+T(n-1) & (n > 0) \end{cases}$$

$T(n) = 4c + T(n-1) = 4c + [4c + T(n-2)] = 2*4c + T(n-2) = 2 * 4c + [4c + T(n-3)]$

Algoritmi liste d'appoggio

Una lista di appoggio è utilizzata per supportare operazioni su una lista principale. Si copia l'elemento nella lista di appoggio, e quindi lo si elimina dalla lista originale. Questo ha un tempo di esecuzione di **ϕ(n)**, ed è utilizzata in algoritmi come *selection sort* e *insertion sort*.

Pile

È una struttura dati che segue il principio **LIFO** (Last In, First Out). Gli elementi si aggiungono e si rimuovono solo dalla cima della pila. Finchè c'è memoria disponibile, è possibile inserire elementi.
Le **operazioni principali** sono:

- **Push(s, x)**: inserisce l'elemento x nella pila s
- **Pop(s)**: rimuove l'elemento in cima alla pila s
- **StackEmpty(s)**: verifica se la pila è vuota
- **Top(s)**: restituisce l'elemento in cima alla pila senza rimuoverlo

Come errori comuni possiamo avere un errore di **overflow**, quando si cerca di inserire un elemento in una pila piena, o di **underflow**, quando si cerca di rimuovere un elemento da una pila vuota.

Cancella

```
Cancella(p,k)
     While not (stackempty(p))
          R = pop(p)
          If r ≠ k then
               Push(p2, r)
          End if
     Wend
     While not(stackempty(p2))
          R = pop(p2)
          Push(p,r)
     Wend
```

$T(n) = 3ct_{w1} + ct_{if} + 3ct_{w2} = \phi(n)$
$T_{w1} = n$, con $0 \leq t_{if}/t_{w2} \leq n$
Migliore: $t_{if} = t_{w2} = 0$, ci solo occorrenze di k
$T(n) = 3c*n + 0 + 0 = 3cn = \Omega(n)$

Peggiore: $t_{if} = t_{w2} = 2$, $k \notin P$
$T(n) = 3cn + cn + 3cn = 7cn = O(n)$

Copia

Copia gli elementi da una coda all'altra.

```
Copia(s)
     While (not(stackempty(s))
          R = pop(s)
          Push(sapp, r)
     Wend
     While (not(stackempty(sapp))
          R = pop(sapp)
          Push(s1, r)
          Push(s2, r)
```

$T(n) = 2c + 5ct_{w1} + ct_{if1} + 5ct_{if2} + ct_{if3} + 5ct_{if4} + c$

Check

Verifica se la sequenza di parentesi date sono tutte chiuse o no.

```
Check(seq[])
     I = 1, err = false
     While i ≤ length(seq) and err == false
          If seq[i] == '(' or seq[i] == '[' then
               Push(s, seq[i])
          End if
          If seq[i] == ')' then
               If stackempty(s) then
                    Err = true, rit = -1
               End if
          Else
```

```
                R = pop(s)
                If r == ‘]’ then
                        Err = true, rit = -2
                End if
        End if
        I++
Wend
If (not(stackempty))
        Rit = -5
End if
Return rit
```

Code

È una struttura dati che segue il principio **FIFO** (First In, First Out).
Le operazioni princiapli sono:

- **Enqueue(y)**: inserisce un elemento in coda
- **Dequeue()**: rimuove un elemento dalla testa
- **QueueEmpty(y)**: controlla se la coda è vuota

Cancella

```
Cancella(q,k) //solo valori positivi, no strutture dati di appoggio
        Enqueue(q, -1) //indica la fine della coda
        R = dequeue(q)
        While r ≠ -1
                If r ≠ k then
                        Enqueue(q,r)
                End if
                R = dequeue®
        Wend
```

$T(n) = 2c + 3ct_w + ct_{if} = \phi(n)$
$T_w = n$
$0 < t_{if} < n$
Migliore: $t_{if} = 0$, sempre $r = k$
$T_m(n) = 2c + 3cn = \Omega(n)$

Peggiore: $k \neq r$
$T_p(n) = 2c + 3cn + cn = O(n)$

Alberi binari

Un albero binario è una struttura dati in cui ogni nodo ha al massimo due figli: **left** e **right**.
Se un nodo non ha figli, allora si chiama **foglia**. Se un nodo ha un genitore, allora è un **discendente** rispetto ad esso. La **profondità** è la lunghezza del cammino dalla radice a un nodo.

Un **albero binario completo** è un albero in cui tutti i livelli, tranne forse l'ultimo, sono completamente riempiti, e tutti i nodi sono il più a sinistra possibile.
Il numero di nodi in un albero è qua a destra.

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Un **albero binario di ricerca** (ABR/BST/SBT) è un albero binario che soddisfa la seguente proprietà:

- Per ogni nodo t, tutti gli elementi del sottoalbero **sinistro** hanno chiavi **minori** di *t.key*.
- Tutti gli elementi del sottoalbero **destro** hanno chiavi **maggiori** di *t.key*.

Visita

La visita in ordine segue questo schema ricorsivo: visita il sottoalbero sinistro, stampa la chiave del nodo corrente, visita il sottoalbero destro.

```
Visita(p)
        If p ≠ null
                Print(p.key)
                Visita(p.left)
                visita(p.right)
```

$T(n) = 2T(n/2) + c$
Divide et impera: $T(n) \in \phi(n)$

Versione ricorsiva possible, ma con stack

Ricerca

```
SBT_Search(P,K)
     If p == null or p.key == k then
          Return p
     Else
          If p.key < k then
               R = sbt_search(p.right, k)
          Else
               R = sbt_search(p.left,k)
          End if
     Return r
```

Se compattato: O(h) = log n
Se non compattato: n

$T(n) = 3c + T(h)$
$N^{\log 2}1 = n^a = 1$
$T(n) = \phi(n * \log n) = \phi(\log n)$

Minimo

```
SBT_min(P pointer)
     If p == null return p
     While p.left ≠ null
          P = p.left
     Wend
     Return p
```

Inserimento

```
SBT_insert(T,x)
     If (root[T] == null) then
          Root[T] = x
     Else
          P = root[T]
          p.prec = null
          while p ≠ null
               if p.key ≥ x.key then
                    p.prec = p
                    p = p.left
               else
                    p.prec = p
                    p = p.right
               end if
          wend
          if (p.prec.key ≥ x.key) then
               p.prec.left = x
          else
               p.prec.right = x
          end if
     end if
```

Cancellazione

```
SBT_delete(T,x)
     If (x.left == null and x.right == null)
          If x == parent.left then
               Parent.left = null
          Else
               Parent.right = null
     End if
     If (x.left == null xor x.right == null) then
          If (x == x.parent.left and x.left ≠ null) then
               x.parent.left = x.left
               (ripeti con left-right, right-left, right-right)
          End if
     Else
          Z = sbt_pred(t,x)
          x.key = z.key
```

```
        sbt_delete(t,z)
    end if
```

Grafi

I grafi sono una struttura dati che viene utilizzata per rappresentare relazioni tra oggetti.
La visita può essere effettuata **in ampiezza**, dove visito tutti i nodi più vicini, e poi tutti i sottografi, ripetendo sempre questa procedura.