# Complessità dei programmi ricorsivi

# Programmi ricorsivi : definizioni iterative e induttive

Fattoriale di un numero naturale : n!

0!=1

$\underbrace{n! = 1 \times 2 \times \ldots\ n}$ se n>0 definizione iterativa

0!=1 caso base

$\underbrace{n!=n*(n-1)!}$ se n>0 definizione induttiva (o ricorsiva)

# fattoriale: algoritmo iterativo

0!=1

n! = 1 x 2 x … n

```
int fact(int n) {
   if (n == 0)
      return 1;
   int a=1;
   for (int i=1; i<=n; i++)
      a=a*i;
   return a;
}
```

1 x 2 x 3 x 4

# fattoriale: algoritmo ricorsivo

0!=1

n!=n*(n-1)!  se n>0

```
int fact(int x) {
  if (x == 0)
      return 1;
  else
      return x*fact(x-1);
}
```

# Programmi ricorsivi: moltiplicazione

mult (0, y) = 0

mult (n,y)= y + mult (n-1,y)   se n>0

```
int mult(int x, int y) {
    if (x == 0)
      return 0;
    return y + mult(x-1,y);
}
```

# Programmi ricorsivi : pari e massimo comun divisore

```
int pari(int x) {
    if (x == 0) return 1;
    if (x == 1) return 0;
    return pari(x-2);
 }
```

## Algoritmo di Euclide

```
int MCD (int x, int y) {
    if (x == y)  return x;
    if (x < y) return MCD (x, y-x);
    return MCD (x-y, y);
 }
```

# Regole da rispettare

Regola 1

individuare i casi base in cui la funzione è definita

immeditamente

Regola 2

effettuare le chiamate ricorsive su un insieme più “piccolo” di dati

Regola 3

fare in modo che alla fine di ogni sequenza di chiamate ricorsive, si ricada in uno

dei casi base

# Regole da rispettare (2)

```
int pari_errata(int x) {
    if (x == 0) return 1;
    return pari_errata(x-2);
}
```

```
int MCD _errata(int x, int y) {
    if (x == y)  return x;
    if (x < y) return MCD _errata(x, y-x);
    return MCD _errata(x, y);
}
```

# programmi ricorsivi su liste

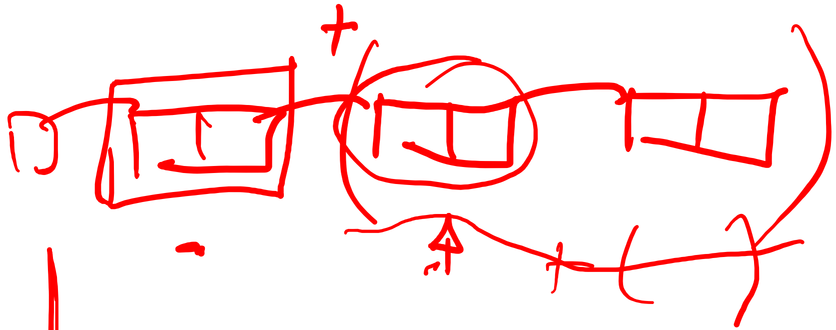
definizione di LISTA

- NULL (sequenza vuota) è una LISTA
- un elemento seguito da una

  **LISTA è una LISTA**

```
struct Elem {
        InfoType inf;
        Elem*    next;
};
```

# programmi ricorsivi su liste



```
int length(Elem* p) {
    if (p == NULL) return 0;        // (! p)
    return 1+length(p->next);
}


int howMany(Elem* p, int x) {
    if (p == NULL) return 0;
    return (p->inf == x)+howMany(p->next, x);
}
```

## programmi ricorsivi su liste

```
int belongs(Elem *l, int x) {
     if (l == NULL) return 0;
     if (l->inf == x) return 1;
     return belongs(l->next, x);
 }

void tailDelete(Elem * & l) {
     if (l == NULL) return;
     if (l->next == NULL) {
       delete l;
       l=NULL;
     }
     else tailDelete(l->next);
 }
```

CASI BASE

# programmi ricorsivi su liste

```
void tailInsert(Elem* & l, int x) {
    if (l == NULL) {
      l=new Elem;
      l->inf=x;
      l->next=NULL;
    }
    else tailInsert(l->next,x);
 }
```

# programmi ricorsivi su liste

```
void append(Elem* & l1, Elem* l2) {
    if (l1 == NULL) l1=l2;
    else append(l1->next, l2);
}
```

```
Elem* append(Elem* l1, Elem* l2) {
    if (l1 == NULL)
        return l2;
    l1->next=append( l1->next, l2 );
    return l1;
}
```

## Induzione naturale

Sia P una proprietà sui naturali.

Base. P vale per 0

Passo induttivo. per ogni naturale n è vero che:

Se P vale per n allora P vale per (n+1)

**P vale per tutti i naturali**

## Somma dei primi n numeri naturali

**Dimostrare con il principio di induzione naturale che la somma dei primi n numeri è n(n+1)/2**

$$\Sigma\_0..n = n(n+1)/2$$

**Base**: Σ_0..0 = (0*1)/2= 0

**Passo induttivo**:

**Ip**: Σ_0..n= n(n+1)/2

**Tesi**: Σ _0..n+1= [(n+1)(n+2)]/2

**Dim**:

Σ _0..n+1 = Σ _0..n +(n+1) def

= n(n+1)/2 + (n+1) ip

= [ n(n+1)+2(n+1) ]/2

= [(n+1)(n+2) ]/2

## Induzione completa

Sia P una proprietà sui naturali.

Base. P vale per 0

Passo induttivo. per ogni naturale n è vero che:

Se P vale per ogni m <= n allora P vale per (n+1)

**P vale per tutti i naturali**

## Induzione ben fondata

**Insieme ordinato S**

Base. P vale per i minimali di S

Passo induttivo. per ogni elemento E di S è vero che:

Se P vale per ogni elemento minore di E allora P vale per E

**P vale per S**

# Complessità dei programmi ricorsivi

$T(n)$ ?

$O(1)$ $\quad$ $O(1)$

```
int fact(int x) {
   if (x == 0) return 1;
   else return x*fact(x-1);
}
```

$O(1) + O(1) = O(1)$

$T(0) = a$

$T(n) = b + T(n-1)$

T ( 0 ) = a
T ( n ) = b + T(n-1)

Relazione di ricorrenza

# soluzione

$$T(0) = a$$
$$T(n) = b + T(n-1)$$

$T(0) = a$

$T(1) = b + a$

$T(2) = b + b + a = 2b + a$

$T(3) = b + 2b + a = 3b + a$

.

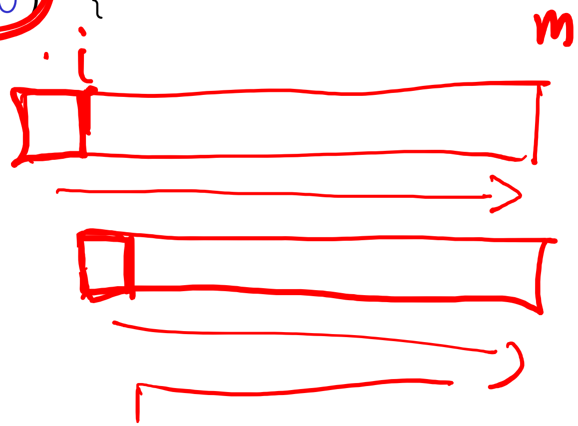
.

$T(n) = nb + a$

**T(n) è O(n)**

## selection sort ricorsiva

```
void r_selectionSort (int* A, int m, int i=0) {
      if (i == m-1)  return;
      int min = i;
      for (int j=i+1; j <m; j++)
            if (A[j] < A[min]) min=j;
      exchange(A[i],A[min]);
      r_selectionSort (A, m, i+1)
}
```




$O(n)$

$+ T(n-1)$

$T(1) = a$

$T(n) = bn + T(n-1)$

## soluzione

$$T(1) = a$$
$$T(n) = bn + T(n-1)$$

$T(1) = a$

$T(2) = 2b + a$

$T(3) = 3b + 2b + a$

.

.

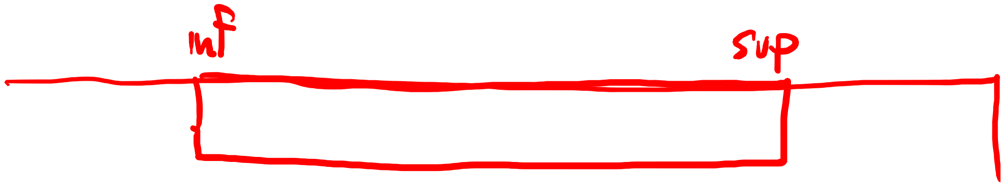
$T(n) = (n + n\text{-}1 + n\text{-}2 + \ldots 2)\, b + a$

$= (n(n+1)/2 - 1)b + a$

$O(n^2)$

**T(n) è $O(n^2)$**

# quicksort




**quicksort(array A ,inf, sup)**

**Lavora sulla porzione di array compresa fra inf e sup:**

1. **Scegli un perno**
2. **Dividi l'array in due parti: nella prima metti gli elementi <= perno e nella seconda quelli >= perno**
3. **chiama quicksort sulla prima parte (se contiene almeno 2 elementi)**
4. **chiama quicksort sulla seconda parte (se contiene almeno 2 elementi)**

# quicksort

**la divisione in due parti dell'array avviene mediante scambi fra elementi utilizzando due cursori s e d, inizialmente posti uno su inf e l'altro su sup:**

**1. Fino a quando s<d:**

**2. {Porta avanti s fino a che A[s] < perno;**
**Porta indietro d fino a che A[d] > perno;**
**Scambia A[s] con A[d];**
**S++;**
**D--;**
**}**

**3. Se inf<d chiama quicksort sulla prima parte (da inf a d);**
**4. Se s<sup chiama quicksort sulla seconda parte (da s a sup);**

# QuickSort

```
void quickSort(int A[], int inf=0, int sup=n-1) {
      int perno = A[(inf + sup) / 2], s = inf, d = sup;
      while (s < d) {
            while (A[s] < perno) s++;
            while ( A[d] > perno) d--;
            if (s > d) break;
            exchange(A[s], A[d]);
            s++;
            d--;
      };
      if (inf < d)
            quickSort(A, inf, d);
      if (s < sup)
            quickSort(A, s, sup);
}
```

O(n)

| | | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| **Array A** | **2** | **5** | **3** | **1** | **7** |

**`quicksort(A, 0, 4)`**

Inf=0 , Sup=4, perno=$\boxed{3}$

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| **2** | **5** | **3** | **1** | **7** |
| s | | | | d |

while

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 5 | 3 | 1 | 7 |
| | s | | d | |

Scambio, poi s++, d--

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 1 | 3 | 5 | 7 |
| | | s, d | | |

while

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **2** | **1** | **3** | **5** | **7** |
| | | s, d | | |

Scambio, poi s++, d--

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **2** | **1** | **3** | **5** | **7** |
| | d | | s | |

s>d: fine del while

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| **2** | **1** | **3** | **5** | **7** |
| | d | | s | |

**Due chiamate ricorsive:**

```
                  d
quicksort(A, 0, 1);
quicksort(A, 3, 4);
                  s
```

**`quicksort(A, 0, 1);`**

Inf=0 , Sup=1, perno=2

| 0 | 1 | 2 | 3 | 4 |
| --- | --- | --- | --- | --- |
| **2** | **1** | **3** | **5** | **7** |
| s | d | | | |

while

| 0 | 1 | 2 | 3 | 4 |
| --- | --- | --- | --- | --- |
| **2** | **1** | **3** | **5** | **7** |
| s | d | | | |

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE



UNIVERSITÀ DI PISA

Scambio, poi s++, d--

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **1** | **2** | **3** | **5** | **7** |
| d | s | | | |

s>d: fine del while,
inf=d, sup=s:
nessuna chiamata ricorsiva

`quicksort(A, 3, 4);`

Inf=3 , Sup=4, perno=5

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **1** | **2** | **3** | **5** | **7** |
| | | | s | d |

while

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **1** | **2** | **3** | **5** | **7** |
| | | | s,d | |

Scambio, poi s++, d--

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **1** | **2** | **3** | **5** | **7** |
| | | d | | s |

s>d: fine del while,
inf>d, sup=s:
nessuna chiamata ricorsiva

# QuickSort

```
quicksort( [3,5,2,1,1], 0, 4)

quicksort( [1,1,2,5,3], 0, 1)

quicksort( [1,1,2,5,3], 3, 4)
```

# Quicksort

$T(1) = a$

$T(n) = bn + T(k) + T(n\text{-}k)$

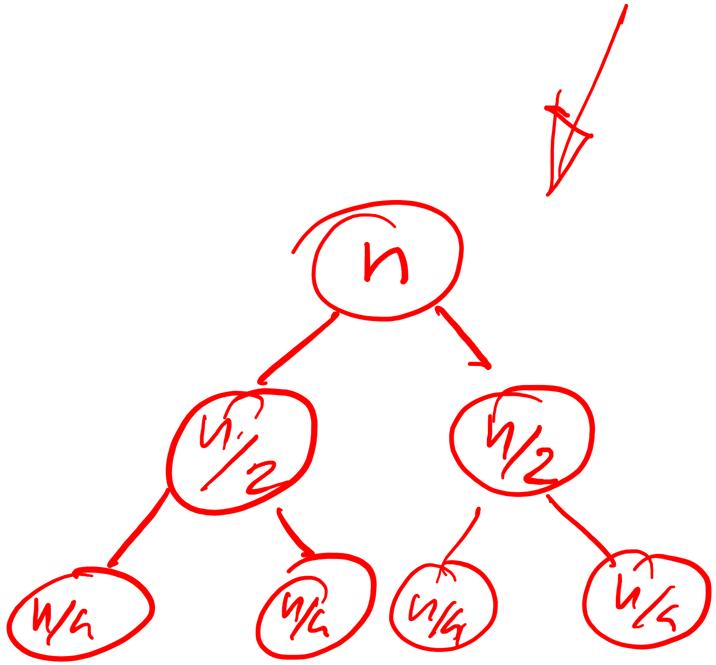
1 n-1

Se k=1:

$T(1) = a$

$O(n^2)$

$T(n) = bn + T(n\text{-}1)$




Se k=n/2:

$T(1) = a$

$T(n) = bn + 2T(n/2)$

## soluzione

$$T(1) = a$$
$$T(n) = bn + 2T(n/2)$$

$T(1) = a$

$T(2) = 2b + 2a$

$T(4) = 4b + 4b + 4a$

8

$T(8) = 8b + 8b + 8b + 8a = 3(8b) + 8a$

16

$T(16) = 16b + 16b + 16b + + 16\ b + 16a = 4(16b) + 16a$

.

.

$T(n) = (n \log n)b + na$

**T(n) è O(nlogn)**

# quicksort

**La complessità nel caso medio è uguale a quella nel caso migliore: O(nlogn) (ma con una costante nascosta maggiore). Questo se tutti i possibili contenuti dell'array in input (tutte le permutazioni degli elementi) sono equiprobabili.**

**Per ottenere questo risultato indipendentemente dal particolare input, ci sono versioni "randomizzate" di quicksort in cui il perno ad ogni chiamata è scelto in modo casuale.**