Calcolare $x^n = x * x * \ldots x$ (n volte)

Abbiamo bisogno di:

- una variabile `ris` in cui ad ogni iterazione del ciclo si ha un risultato parziale, e che dopo l'ultima iterazione contiene il risultato finale;
- una variabile contatore `i`;

Eseguiamo l'istruzione `ris = 1;` dopo la sua esecuzione, subito prima di entrare nel ciclo, cioè dopo 0 iterazioni, la variabile `ris` contiene il valore x^0

Supponiamo che dopo `K` ripetizioni del ciclo la variabile `ris` contenga il valore x^K;

allora, se il corpo del ciclo è l'istruzione

```
ris = ris * x;
```

dopo `K+1` ripetizioni del ciclo, `ris` contiene il valore x^{K+1}.

Allora, per qualunque n, dopo n iterazioni `ris` conterrà x^n (per il principio di induzione).

Costruiamo una classe Mate di funzioni matematiche
iniziamo con exp(x,n) = x^n = x∗x∗ ... x (n volte)

```
public class Mate {
  private Mate() { } // non si possono costruire oggetti

  static double exp(double x, int n) {
    double ris = 1;
    for(int i=1; i<=n; i++) ris = ris * x;
      return ris;
    }
  }

public class UsaMate {
  public static void main(String args[]) {
    System.out.writeln(Mate.exp(5,30));
  }
}
```

esponenz. o potenza realizzata con ciclo discendente

```
static double exp(double x, int n) {
   double ris = 1;
   for( ; n>0; n--) ris = ris * x;
   return ris;
 }
```

chiamiamo N il contenuto (>= 0) di n, X il contenuto di x; nell'istante immediatamente precedente la prima esecuzione del corpo del ciclo si ha evidentemente:

ris: **1** n: N x: X

Si osservi che allora vale la seguente relazione matematica:

$$(\text{contenuto di } ris) \cdot (\text{contenuto di } x)^{(\text{contenuto di } n)} = X^N$$

che scriveremo in forma concisa come:

$$ris \cdot x^n = X^N$$

potenza o esponenziale con ciclo discendente

ris: **1** n: **N** x: **X** $ris \cdot x^n = X^N$

dopo la prima esecuzione del ciclo, cioè delle due istruzioni:

```
ris = ris * x;
n--;
```

ris: **X** n: **N - 1** x: **X** $ris \cdot x^n = X^N$

poi

ris: X^2 n: **N - 2** x: **X** $ris \cdot x^n = X^N$

... ...

ris: X^N n: **0** x: **X** $ris \cdot x^n = X^N$

I valori contenuti nelle variabili ris ed n cambiano ad ogni iterazione, ma il valore dell'espressione $ris \cdot x^n$ non varia.

Invarianti di ciclo

La proposizione

$$(\text{contenuto di } ris) \cdot (\text{contenuto di } x)^{(\text{contenuto di } n)} = X^N$$

cioè, in forma concisa:

$$ris \cdot x^n = X^N$$

è vera immediatamente prima di eseguire l'istruzione for,

è vera dopo ogni iterazione del ciclo,

è vera all'uscita dal for.

Tale **asserzione**, cioè tale proposizione riguardante i contenuti delle variabili della procedura, si dice che è un **invariante** del ciclo.

Nota: il simbolo rosso **=** è la vera uguaglianza matematica nel metalinguaggio; non confonderlo con i simboli Java = dell'assegnazione e == dell'operatore di confronto.

Esponenziale (o potenza) veloce.

```
static double expVeloce(double x, int n) {
  // precondizione: n >= 0
  double ris = 1;
   while(n > 0) {
     if(n%2 == 0) {  // n pari
       x = x*x;
       n = n/2;
     }
     else {              // n dispari
       ris = x*ris;
       n--;
     }
   }
   return ris;
} // postcondizione: restituisce x^n
```

Traccia di dimostrazione di correttezza di expVeloce

Lemma: Se X ed N sono i contenuti iniziali risp. di x ed n, dopo qualunque numero di ripetizioni del corpo del ciclo while vale l'asserzione:

$$(\text{contenuto di } x)^{(\text{contenuto di } n)} \cdot (\text{contenuto di } ris) = X^N$$

Dimostrazione per induzione:

Base: Dopo 0 ripetizioni del ciclo (cioè immediatamente prima di eseguire il while) l'asserzione vale.

Dimostrazione della base: ovvia, perché il contenuto iniziale di ris è 1, quelli di x ed n sono rispettivamente X ed N, quindi:

$$(\text{contenuto di } x)^{(\text{contenuto di } n)} \cdot (\text{contenuto di } ris) = 1 \cdot X^N = X^N$$

Traccia di dimostrazione di correttezza di expVeloce (cont.)

Passo induttivo:

Ipotesi induttiva: Dopo k iterazioni del ciclo si ha:

x: Y n: M ris: R

(contenuto di x)$^{\text{(contenuto di n)}}$ · (contenuto di ris) = $Y^M \cdot R = X^N$

Tesi induttiva: Dopo k+1 iterazioni si ha:

x: Y' n: M' ris: R'

(contenuto di x)$^{\text{(contenuto di n)}}$ · (contenuto di ris) = X^N

Traccia di dimostrazione di correttezza di expVeloce (cont.)

Dimostrazione del passo induttivo:

x: Y n: M ris: R

(contenuto di x)$^{\text{(contenuto di n)}}$ · (contenuto di ris) = $Y^M \cdot R = X^N$

caso M pari: vengono eseguite le istruzioni

```
x = x*x;
n = n/2;
```

allora si ha: x: Y^2 n: $M/2$ ris: R

$(Y^2)^{M/2} \cdot R = Y^M \cdot R = X^N$

caso M dispari: vengono eseguite le istruzioni

```
ris = x*ris;
n = n - 1;
```

allora si ha: x: Y n: $M-1$ ris: $Y \cdot R$

$Y^{M-1} \cdot (Y \cdot R) = Y^M \cdot R = X^N$

Traccia di dimostrazione di correttezza di expVeloce (cont.)

Teorema: Se X ed N sono i contenuti iniziali risp. di x ed n, all'uscita dal ciclo si ha:

$$\text{contenuto di } ris = X^N$$

Dimostrazione: Per il lemma, all'uscita dal ciclo si ha:

$$(\text{contenuto di } x)^{(\text{contenuto di } n)} \cdot (\text{contenuto di } ris) = X^N$$

Ma all'uscita dal ciclo si ha anche (perché? vedi slide succ.):

$$\text{contenuto di } n = 0$$

Quindi:

$$(\text{contenuto di } x)^0 \cdot (\text{contenuto di } ris) = X^N$$

cioè:

$$1 \cdot (\text{contenuto di } ris) = X^N$$

cioè

$$\text{contenuto di } ris = X^N$$

Traccia di dimostrazione di correttezza di expVeloce (fine)

Abbiamo così dimostrato che se il ciclo while termina, e se all'uscita il contenuto di n è 0, allora la procedura restituisce il risultato corretto.

Per completare la dimostrazione di correttezza bisogna quindi ancora dimostrare che effettivamente dal ciclo si esce sempre, e sempre con n = 0.

Ma ciò è abbastanza ovvio:

inizialmente deve essere n **>=** 0;

il ciclo viene eseguito solo se n > 0; ma se n > 0 allora si ha:

0 < n/2 < n, 0 **<=** n-1 < n

Quindi ad ogni passo il contenuto di n diminuisce pur restando non negativo: allora dopo un numero finito di passi assumerà per forza il valore 0 (e quindi ris conterrà il risultato voluto).

Ricerca sequenziale realizzata senza break

```
public static boolean ricerca(double x, double[] a) {
    int i = 0;
    int n = a.length;
    while(i < n && a[i] != x) i++;
    return i < n;
}
```

alla fine di ogni ripetizione del ciclo sappiamo che

x non compariva in a[0 .. i-1], e che i <= n

il ciclo può terminare con:

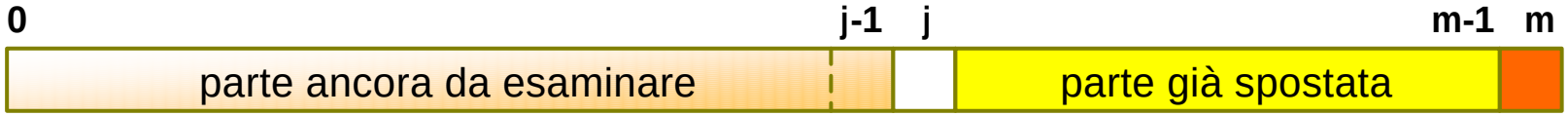
i < n, a[i] = x: il valore x compare nell'array;

i = n: per i < n il valore x non compariva, quindi x non c'è in a.

Inserimento di un valore x in un array ordinato a

situazione iniziale: array ordinato, con m elementi occupati

situazione al passo generico:



è x < a[j+1] <= a[j+2] <= a[j+3] <= ... <= a[m]

a[j] è il posto libero (che si è creato spostando a destra di uno tutti gli elementi successivi;

il ciclo termina se j = 0 oppure se (j > 0 e) x >= a[j-1]:

in entrambi i casi il posto libero ha raggiunto la posizione giusta, e il valore x va inserito in tale posto;

altrimenti l'elemento a[j-1] va spostato in a[j], il che equivale a spostare a sinistra di uno il posto libero

Si esce dal ciclo per **j = 0 oppure x >= a[j-1];**

quindi, anche ricordando de Morgan,

si continua per **j > 0 and x < a[j-1];**

In conclusione:

```
while(j > 0 && x < a[j-1]) {
  a[j] = a[j-1];
  j--;
}
a[j] = x;
```

Inizializzazione: l'array era occupato fino ad a[m-1] compreso,

quindi la posizione iniziale del posto libero è m

```
j = m;
```

testo della procedura completa

presi come parametri:

un intero **x;** un array **a** parzialmente riempito;

la lunghezza **m** della parte occupata di **a**, che deve essere ORDINATA

(e l'array **a** deve avere lunghezza almeno m+1)

inserisce l'intero **x** al posto giusto nell'array ordinato **a**

```
private static void inserisci(int x, int[] a, int m) {
  int j = m;
  while( j > 0 && x < a[j-1]) {
    a[j] = a[j-1];
    j--;
  }
  a[j] = x;
}
```

Tempo di calcolo della procedura inserisci

caso migliore: il valore da inserire è maggiore o uguale all'ultimo dei presenti nell'array (e quindi a tutti);

il calcolo richiede **un solo passo**;

caso peggiore: il valore da inserire è minore di tutti i presenti,

quindi esso sarà confrontato con tutti gli elementi dell'array,

i quali dovranno essere tutti spostati di uno;

il numero di passi di calcolo è **proporzionale a n**

(ossia **lineare in n**)

in media il valore da inserire sarà a metà dell'array, quindi il numero di passi sarà circa proporzionale a n/2, cioè ancora **proporzionale a n** (o **lineare in n**)

Creazione di una copia ordinata di un array

Dato un array arbitrario, per creare un nuovo array contenente tutti gli elementi dell'array di partenza ma disposti in ordine, basta (copiare il primo elemento nel nuovo array, e poi) inserire via via nel nuovo array gli elementi dell'array originale, per mezzo della procedura inserisci (che mantiene l'ordine).

```
public static int[] copiaOrdinata(int[] a) {
  int n = a.length;
  int[] b = new int[n]; // crea un array di lunghezza uguale
  b[0] = a[0];          // copia il primo elemento
  for(int i = 1; i < n; i++) {
    inserisci(a[i],b,i);  // inserisce a[i] nell'array b
  }                       // contenente i elementi occupati,
  return b;               // cioè pieno fino a b[i] escluso;
}
```

Tempo di calcolo della procedura copiaOrdinata

Il corpo della procedura è costituito da un ciclo for di n passi, senza uscite forzate. Tuttavia all'interno del ciclo viene invocata la procedura inserisci il cui tempo di calcolo non è costante, ma in generale dipende dal parametro i.

In media, inserisci farà un numero di passi circa uguale a i/2, cioè 1/2 la prima volta, 2/2 la seconda, 3/2 la terza, ...;

approssimativamente, il numero totale di passi sarà allora:

$1/2\ (1 + 2 + 3 + ... + n) = 1/2\ (n(n+1)/2) = (n^2 + n)/4$

Il tempo di calcolo cresce in modo **quadratico** rispetto alla lunghezza dell'array da copiare/ordinare, cioè è circa proporzionale al quadrato di tale lunghezza: se la lunghezza raddoppia, il tempo si quadruplica; se la lunghezza si triplica, il tempo si moltiplica per 9; e così via.

Tempo di calcolo di copiaOrdinata (cont.)

caso peggiore: l'array di partenza è ordinato inversamente; allora ogni chiamata della procedura inserisci esegue esattamente k passi, con k successiv. uguale a 1, 2, ...,n (verificare simulando a mano l'esecuzione); il numero totale di passi è quindi:

$1+2+3+ \ldots + n = n(n+1)/2 = (n^2 + n)/2$

quadratico

caso migliore: l'array di partenza è già ordinato; allora ogni inserimento, inserendo un elemento che è maggiore o uguale dei precedenti, fa un solo passo; il numero totale di passi è quindi

1+1+1+ ... (n volte)

cioè **proporzionale a n** (o **lineare in n**)

Tempo di calcolo di copiaOrdinata (cont.)

Riassumendo:

- caso peggiore: tempo quadratico (rispetto alla lunghezza)
- caso medio: tempo quadratico
- caso migliore: tempo lineare

Ordinamento per inserimento

Nella procedura copiaOrdinata (come in qualunque procedura di copia), ad ogni istante la parte occupata nel nuovo array ha ovviamente la stessa lunghezza della parte già esaminata (e appunto copiata) dell'array di partenza. Allora, se non interessa mantenere anche la versione originale non ordinata, invece di creare un nuovo array'si può mettere la parte ordinata nell'array stesso passato come argomento, cioè far coincidere, per così dire, l'array di partenza con quello di arrivo.

In questo modo si ottiene una procedura che, a differenza di copiaOrdinata, non restituisce alcun array né alcun risultato, ma invece modifica l'array-argomento, ordinandolo.

È l'algoritmo di ordinamento per inserimento, che non richiede quindi array aggiuntivi, e che ha ovviamente complessità temporale quadratica nei casi peggiore e medio, lineare nel caso migliore (array già ordinato o quasi ordinato).

```
public static int[] copiaOrdinata(int[] a) {
    int n = a.length;
    int[] b = new int[n];
    b[0] = a[0];
    for(int i = 1; i < n; i++) {
      inserisci(a[i],b,i);
    }
    return b;
}
```

```
public static void      Ordina   (int[] a) {
   int n = a.length;

   a[0] = a[0];
   for(int i = 1; i < n; i++) {
     inserisci(a[i],a,i);
   }

}
```

```
public static void      Ordina  (int[] a) {
   int n = a.length;


   for(int i = 1; i < n; i++) {
    inserisci(a[i],a,i);
   }

}
```

Procedura di ordinamento per inserimento

```
public static void ordinaIns(int[] a) {
    int n = a.length;
    for(int i = 1; i < n; i++) {
      inserisci(a[i],a,i);
    }
}
```