Analisi tempo di esecuzione degli algoritmi

Una domanda

Accedere all'i-simo elemento di una lista in Python prende un tempo costante o dipendente dal numero degli elementi della lista?

1. **Su quale macchina?**

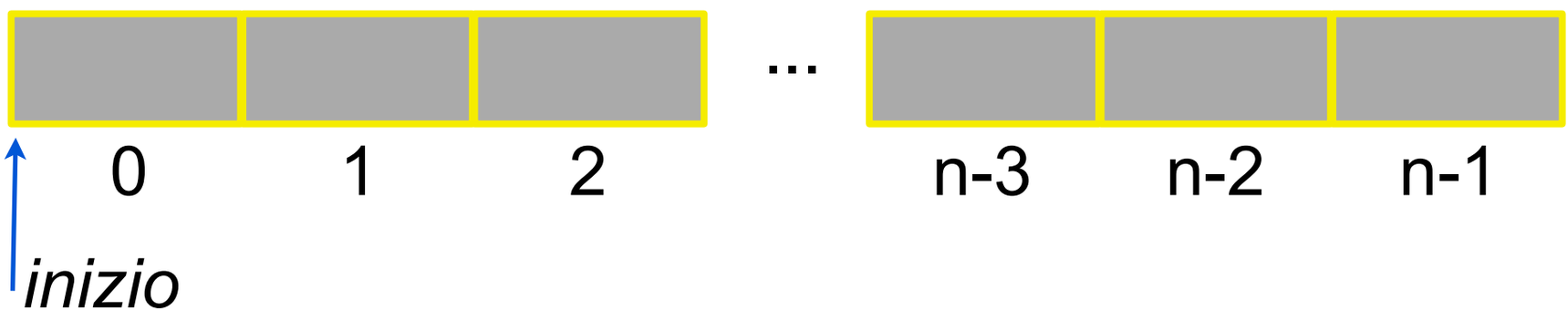
   Sul modello RAM

2. **Cosa bisogna sapere per rispondere?**

   Come è implementata l'operazione e come è rappresentata in memoria la lista.

Rappresentazione in memoria

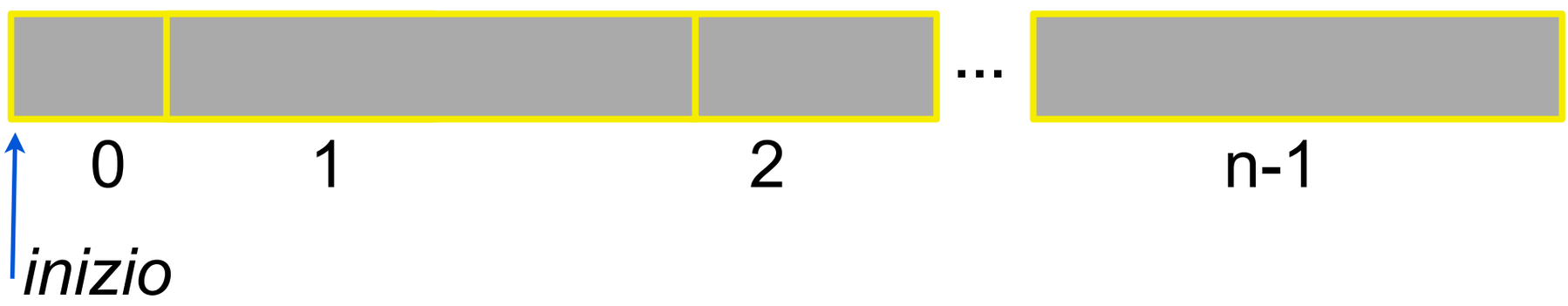
Supponiamo per ora che la lista contenga solo interi (il "vecchio" array) e che un intero sia rappresentato in memoria con 4 UM, dove con UM indico una Unità di Memoria (un byte o altro: non conta qui il dettaglio).
Una lista di interi occupa un'area contigua di memoria:



Qui *inizio* è l'indirizzo in memoria del primo elemento della lista.
L'elemento di indice i si trova all'indirizzo *inizio* + i*4UM
questa semplice espressione aritmetica è valutata in tempo costante, cioè indipendentemente dalla lunghezza della lista!

Rappresentazione in memoria

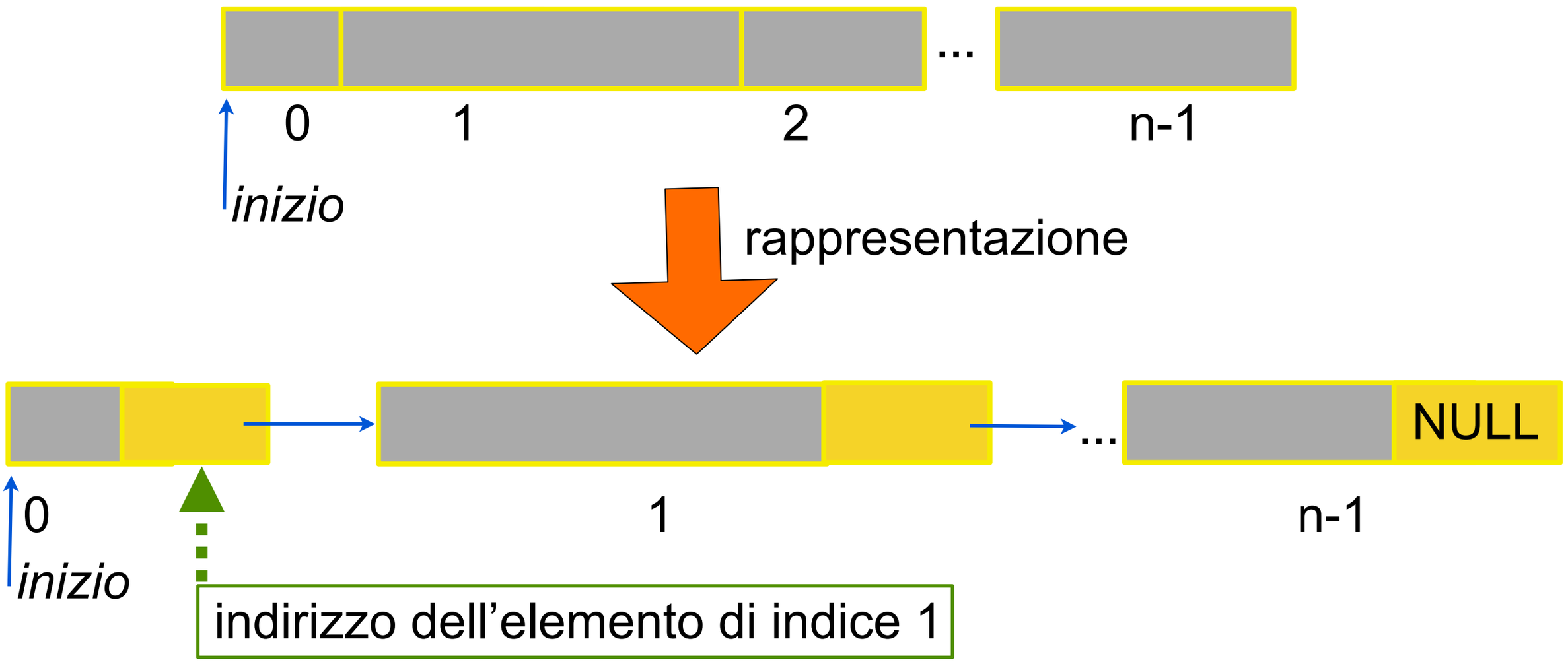
Ma una Lista in Python può contenere dati di tipo diverso, tra cui anche altre liste:



L’ elemento di indice i si trova calcolando ind(i)=?

Rappresentazione in memoria

Si potrebbe usare una lista concatenata:
si dota ogni elemento della lista di un campo *riferimento*
(il "vecchio" *puntatore)*, che contiene l'indirizzo
dell'elemento successivo:



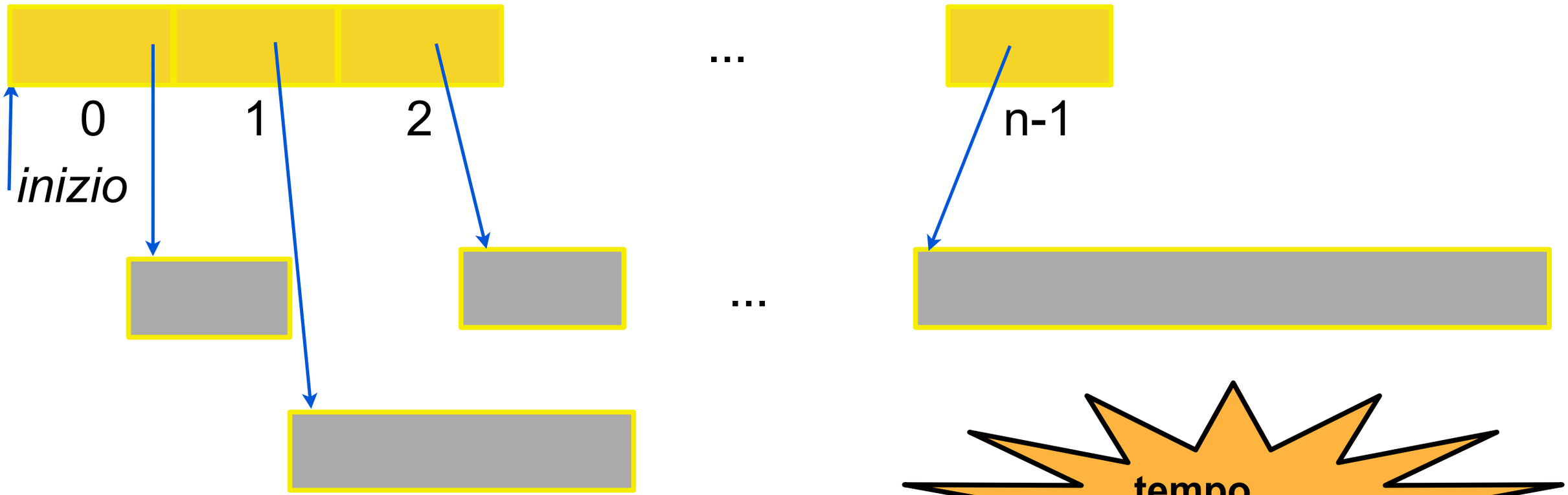
Rappresentazione in memoria

L'elemento di indice i è ottenuto seguendo il riferimento (l'indirizzo di memoria) contenuto nel campo riferimento dell'(i-1)-simo elemento.
Ma quindi per ottenerlo bisogna "seguire" i riferimenti, a partire dal primo!
Dunque il numero di passi necessari non è costante, ma dipende dal valore di i, che varia tra 0 e n-1.

non va bene

Rappresentazione in memoria

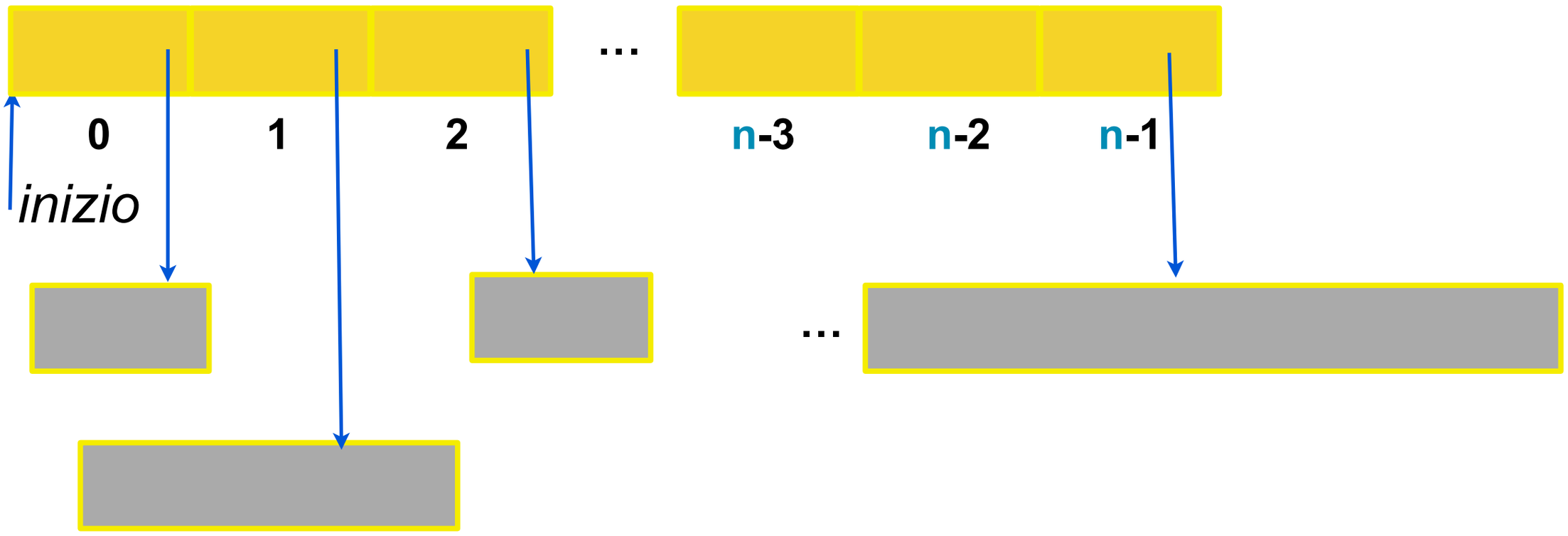
Usiamo un array i cui elementi sono i riferimenti alla locazione di memoria iniziale degli elementi:



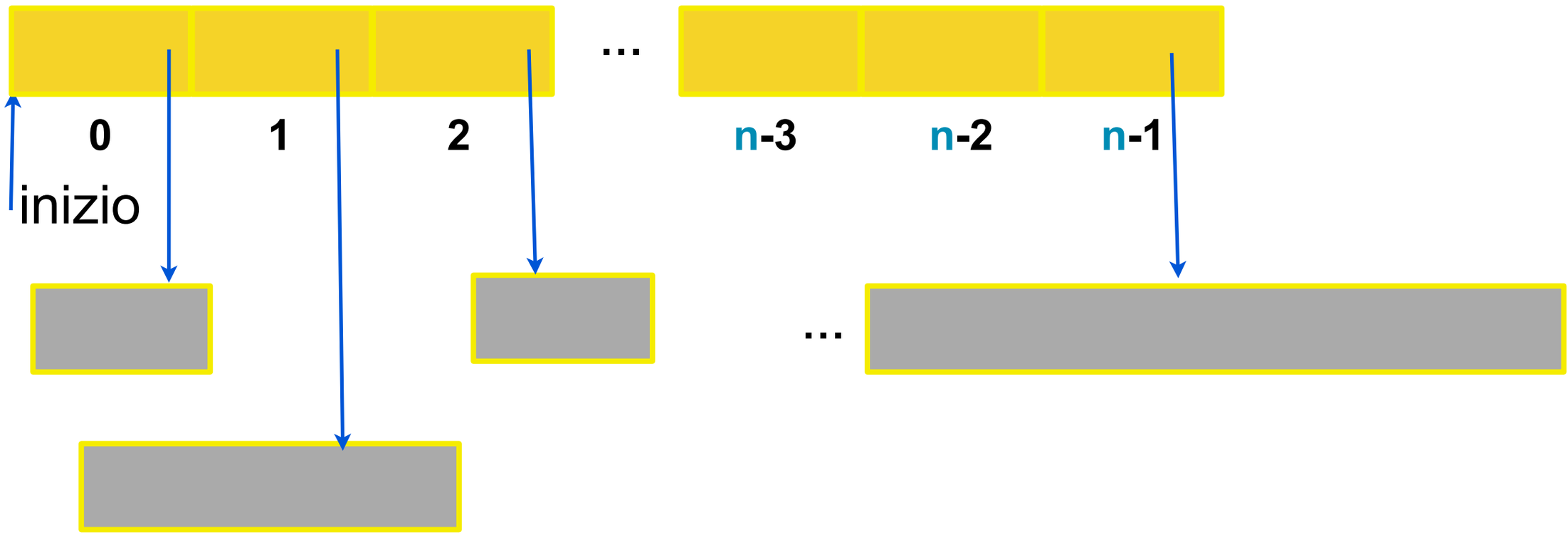
Un riferimento occupa 4UM, quindi l' elemento di indice i è contenuto nella locazione il cui indirizzo è *inizio*+i*4UM

L’inserimento in una lista?

Il metodo nome_lista.insert(*i*,el) inserisce l’ elemento el nella posizione *i* nella lista nome_lista. E’ eseguito in un tempo costante o dipendente dalla dimensione (=numero degli elementi) della lista?



L’inserimento in una lista



Occorrerà (a parte i controlli sui valori della chiamata, di costo costante)

1. aumentare di uno la lunghezza, n, della lista e allocare nuova memoria per la lista, se possibile - tempo costante

2. spostare a destra di una posizione tutti i puntatori da *i* a n - tempo dipendente da *i*, che può essere 0 o n

3. inserire l’indirizzo di el in *i* - tempo costante

Analisi inserimento in una lista

Il metodo nome_lista.insert(*i*,el) inserisce l' elemento el nella posizione *i* nella lista nome_lista.

Occorrerà (a parte i controlli sui valori della chiamata, di costo costante)
1. aumentare di uno la lunghezza, n, della lista e allocare nuova memoria per la lista, se possibile - tempo costante
2. spostare a destra di una posizione tutti i puntatori da *i* a n - tempo dipendente da *i*, che può essere 0 o n
3. inserire el in *i* - tempo costante

Possiamo dire che
1. nel caso migliore *i* = n e allora tutta l'operazione è eseguita in tempo costante
2. nel caso peggiore *i* = *0* e allora l'operazione è eseguita in c * n + d passi per un certo c > 0, che dà conto dei costi delle operazioni eseguite circa n volte e d > 0 che dà conto del costo costante delle operazioni diverse dagli spostamenti
3. in generale che l'operazione è eseguita in un numero di passi superiormente limitato da c * n + d

Analisi inserimento in una lista

Il metodo nome_lista.insert(*i*,el) inserisce l' elemento el nella posizione *i* nella lista nome_lista.

Occorrerà (a parte i controlli sui valori della chiamata, di costo costante)
1. aumentare di uno la lunghezza, n, della lista e allocare nuova memoria per la lista, se possibile - tempo costante
2. spostare a destra di una posizione tutti i puntatori da *i* a n - tempo dipendente da *i*, che può essere 0 o n
3. inserire el in *i* - tempo costante

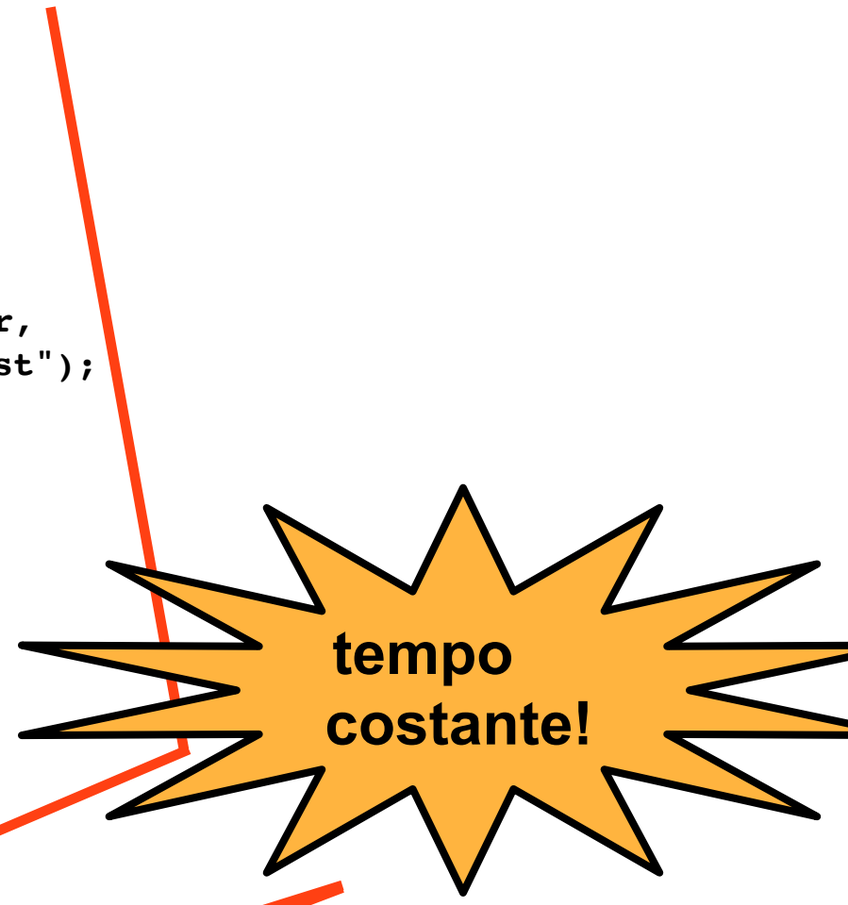
Quindi se chiamiamo $T_{ins}(n)$ la funzione che esprime il tempo di calcolo del metodo su un input di dimensione n, possiamo dire che $T_{ins}(n) \leq c * n + d$.
Oppure se chiamiamo $TMAX_{ins}(n)$ la funzione che esprime il tempo di calcolo del metodo nel caso peggiore su un input di dimensione n, possiamo dire che $TMAX_{ins}(n) = c * n + d$.

L'inserimento in una lista,il codice in C:

http://svn.python.org/view/*checkout*/python/tags/r271/Objects/listobject.c?content-type=text%2Fplain

L'assegnamento è ripetuto per i da n fino a *where*, cioè per n - *where* +1 volte. Quindi detto c il costo dell'assegnamento e dei confronti, si ha un costo di esecuzione del ciclo for pari a c(n - *where* +1). $TMAX_{ins}(n) = c * n + d$, dove d è il costo costante delle operazioni al'esterno del ciclo.

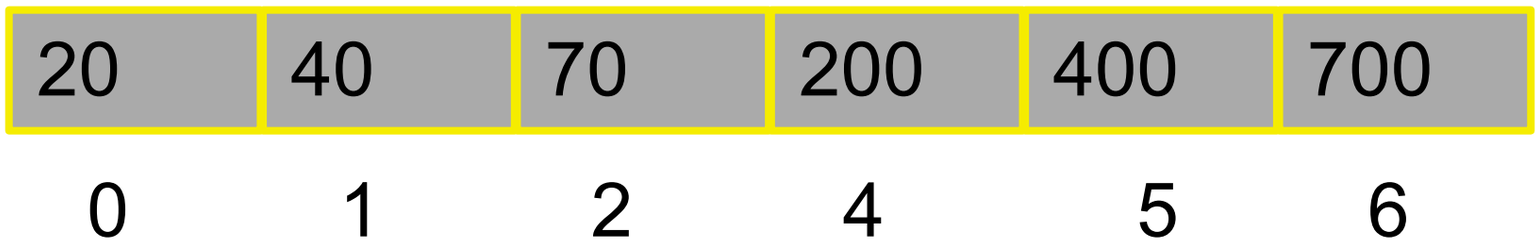
```
static int
ins1(PyListObject *self, Py_ssize_t where, PyObject *v)
{
    Py_ssize_t i, n = Py_SIZE(self);
    PyObject **items;
    if (v == NULL) {
        PyErr_BadInternalCall();
        return -1;
    }
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
            "cannot add more objects to list");
        return -1;
    }

    if (list_resize(self, n+1) == -1)
        return -1;

    if (where < 0) {
        where += n;
        if (where < 0)
            where = 0;
    }
    if (where > n)
        where = n;
    items = self->ob_item;
    for (i = n; --i >= where; )
        items[i+1] = items[i];
    Py_INCREF(v);
    items[where] = v;
    return 0;
}
```



Altro esempio di analisi della complessità: Inserimento in un array ordinato

Dato un array A di interi ordinato in ordine crescente, cioè in modo tale che A[i] ≤ A[i+1], per 0≤i<n-1, vogliamo inserire un elemento, in modo da ottenere ancora un array ordinato.

Esempio: inserimento di 80 in :



Inserimento in un array ordinato: soluzione 1

A è un array di interi ordinato in ordine crescente, cioè in modo tale che A[i] ≤ A[i+1], per 0≤i<n-1. Vogliamo inserire un elemento, in modo da mantenere la proprietà che l'array sia ordinato. Cominciamo con l'aggiungerlo in coda.

Esempio: inserimento di 80 nell'array:

20	40	70	200	400	700	
0	1	2	3	4	5	6

80

Confrontiamo l'elemento in coda all'array con il penultimo e se sono nell'ordine sbagliato li scambiamo e proseguiamo con i confronti e gli scambi con gli altri elementi finché il nuovo elemento è maggiore o uguale di quello con cui è confrontato.

Inserimento in un array ordinato: soluzione 1

Confrontiamo l'elemento in coda all'array con il penultimo e se sono nell'ordine sbagliato li scambiamo e proseguiamo con i confronti e gli scambi con gli altri elementi finché il nuovo elemento è maggiore o uguale di quello con cui è confrontato.

Esempio: inserimento di 80 nell'array:

20	40	70	200	400	700	80
0	1	2	3	4	5	6

Inserimento in un array ordinato: soluzione 1

Possiamo evitare di fare uno scambio ogni volta che l'elemento risulta minore di quello con cui è confrontato. Salviamo l'ultimo elemento in una variabile x

Esempio: inserimento di 80 nell'array A:

x = 80

20	40	70	200	400	700	80
0	1	2	3	4	5	6

Poiché x = 80 < A[5] = 700, sposto A[5] di una posizione a destra per fare spazio a 80

Poiché x = 80 < A[4] = 400, sposto A[4] di una posizione a destra per fare spazio a 80

Poiché x = 80 < A[3] = 200, sposto A[3] di una posizione a destra per fare spazio a 80

Ora x= 80 ≥ A[2]=70, quindi inserisco 80 in A

Inserimento in un array ordinato: pseudocodice

```
Ins(A,lo,hi)
input:A è un array di interi, lo e hi due interi non negativi
prec: A[lo:hi-1] è  ordinato
output: l'array A[lo:hi] ordinato, estendendo l'ordinamento
all'ultimo elemento
x = A[hi]
i = hi-1
while x < A[i] do
      A[i+1] = A[i] // spostamento di una posizione a destra
      i = i - 1
A[i+1] = x
  // All'uscita x ≥ A[i] e x < A[i+1]  e poichè A è ordinato k<i =>
  A[k] ≤ A[i] ≤ x e x≤ A[k] per i< k ≤ j e quindi i+1 è il posto
  giusto per x in A
```

Inserimento in un array ordinato: analisi soluzione 1

```
Ins(A,lo,hi)
x = A[hi]
i = hi-1
while x < A[i] do
      A[i+1] = A[i]
      i = i -1
 A[i+1] = x
```

tempo costante

Gli assegnamenti sono ripetuti finchè vale x < A[i]. Sia n il numero di elementi di A[lo:hi], c il costo (costante e positivo) dei confronti e degli assegnamenti all'interno del ciclo while e d quello (costante e positivo) delle operazioni al di fuori del ciclo:

1. nel caso migliore non si entra nel *ciclo while* perché $x \geq A[hi-1]$, A[hi] si trova già nella posizione giusta e allora tutta l'operazione è eseguita in tempo costante.
2. nel caso peggiore $x < A[lo]$ e allora l'operazione è eseguita in $c * n + d$ passi, cioè la complessità di tempo nel caso peggiore su un input di dimensione n, $TMAX_{linInsOrd}(n)$ è pari a $c * n + d$.
3. in generale che l'operazione è eseguita in un numero di passi superiormente limitato da $c * n + d$, cioè che la complessità di tempo su un input di dimensione n $T_{linInsOrd}(n) \leq c * n + d$.

Inserimento in un array ordinato: soluzione 2

Dato un array A di interi ordinato in ordine crescente, cioè in modo tale che A[i] ≤ A[i+1], per 0≤i<n-1, vogliamo inserire un elemento, in modo da mantenere l'ordine.

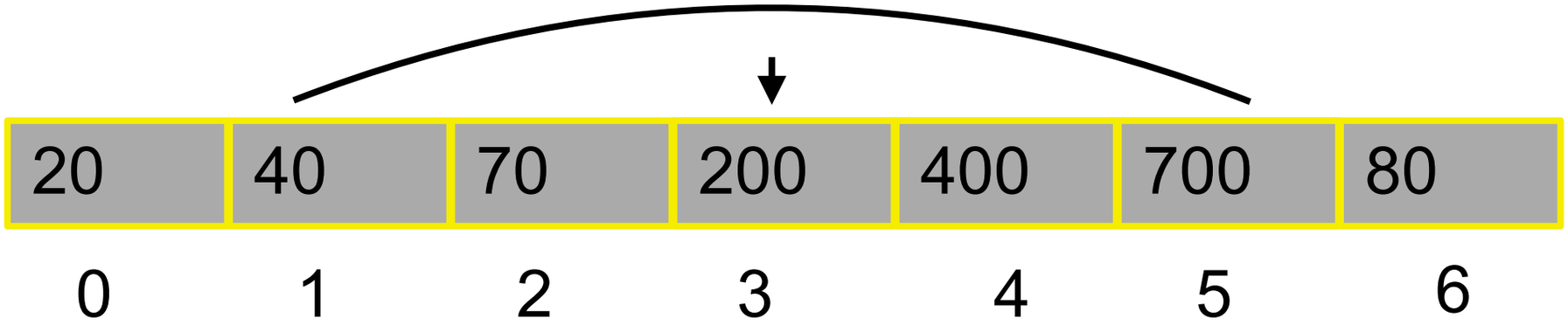
Possiamo usare il metodo bisect_ritgh di Python, qui semplificato, e che trovate seguendo questo link http://hg.python.org/cpython/file/2.7/Lib/bisect.py per determinare la posizione di inserimento dell'elemento.
Si tratta di una variante dell'algoritmo della ricerca binaria o dicotomica!

Inserimento in un array ordinato: la posizione

```
RBisect( A,x)
input: A è un array di  n interi, x è un intero
prec: A è ordinato
output: se x non è presente dà la posizione di inserimento
ordinato di x, altrimenti dà la posizione immediatamente a
destra dell'ultima occorrenza di  x in A.
lo=0
hi = A.size
while lo < hi do
se l'elemento cercato è presente ha un indice in [lo,hi)
        m = midpoint(lo,hi)
        if   x < A[m] then hi = m  else lo = m + 1
return lo
```

All'uscita dal ciclo lo=hi e gli elementi A[0],…,A[lo-1], dell'intervallo A[0,lo), sono minori o uguali a x, mentre gli elementi A[lo],…,A[n-1], dell'intervallo A[lo,n), sono maggiori di x. Quindi lo è la posizione giusta per x in A in ogni caso.

Calcolo di midpoint

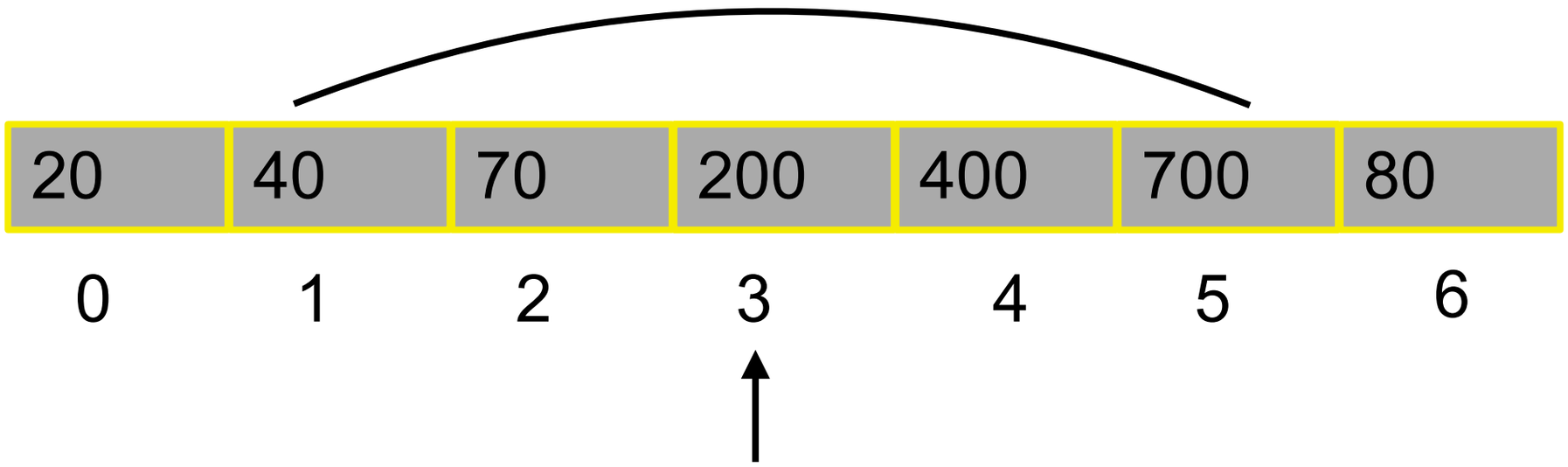


L'intervallo che consideriamo è [lo,hi) cioè è l'insieme degli elementi di indice lo,lo+1, …, ,hi-1.
Per esempio lo = 1 e hi = 5
Il numero degli elementi è la differenza tra gli indici: 5 - 1 e l'indice si ottiene aggiungendo questa quantità divisa per due all'indice iniziale, 1 ne nostro caso, otteniamo quindi 1+2=3.

La regola è (hi - lo)/2 + lo.
Notiamo che (hi - lo)/2 + lo = (hi - lo + 2lo)/2 = (hi + lo)/2

midpoint(lo,hi) può essere implementata in modi diversi:
midpoint(lo,hi) = (lo + hi)/2 se si è sicuri di non incorrere in overflow, midpoint(lo,hi) = lo + (hi-lo)/2 altrimenti

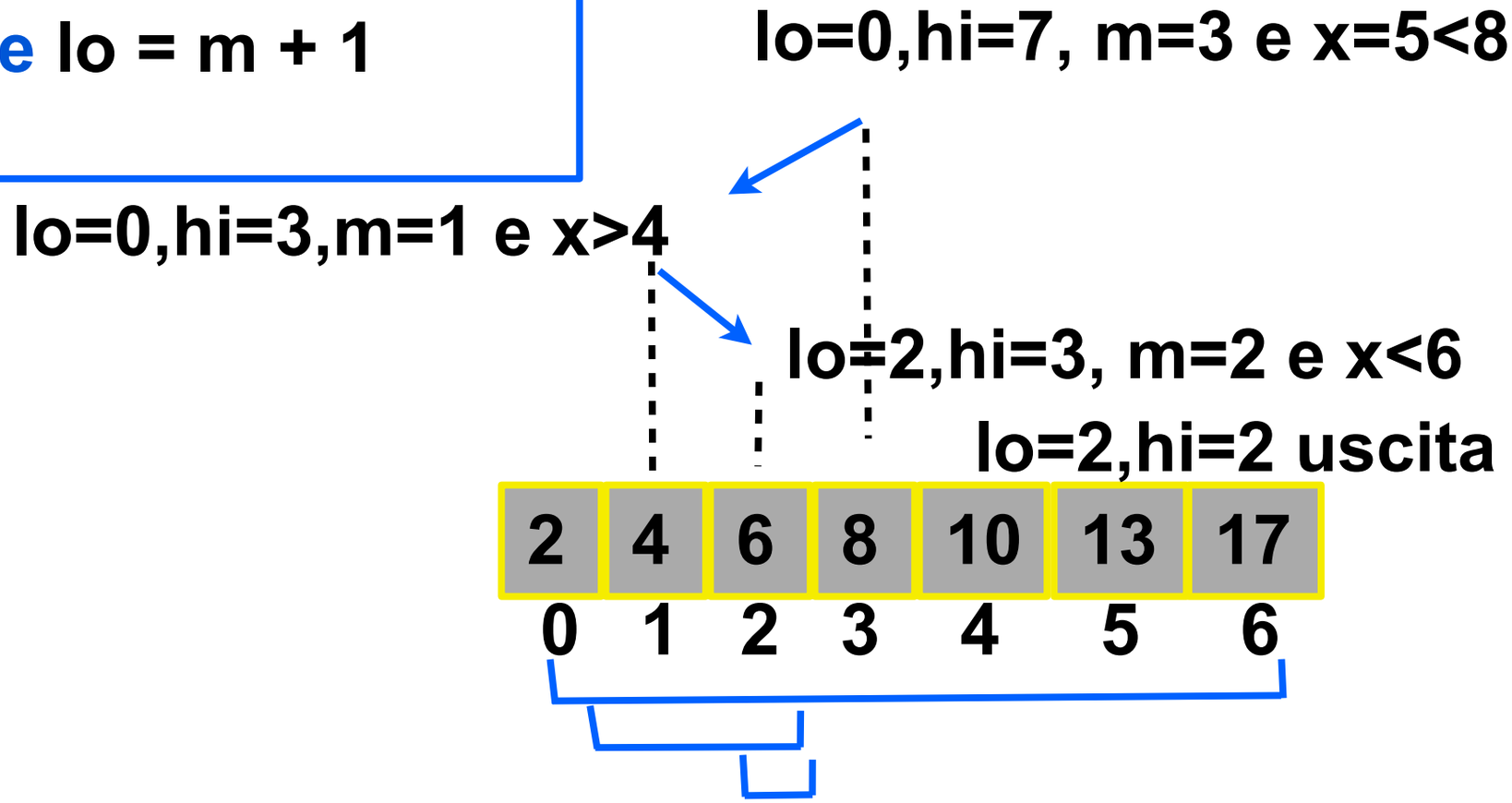
Calcolo dell'indice intermedio



Se avessimo dovuto considerare l'intervallo [lo,hi] cioè è l'insieme degli elementi di indice lo,lo+1, …, ,hi, allora il numero degli elementi è hi - lo + 1
Per esempio lo = 1 e hi = 5
Il numero degli elementi è la differenza tra gli indici aumentata di uno, nell'esempio, 5 - 1 + 1 e l'indice si ottiene aggiungendo questa quantità divisa per due all'indice iniziale, 1 ne nostro caso, otteniamo quindi 1 + 2 = 3.
La regola è (hi - lo + 1)/2 + lo.
Notiamo che (hi - lo + 1)/2 + lo =
(hi - lo +1 + 2lo)/2 = (hi + lo + 1)/2

RBisect: Esempio

```
RBisect( A, x)
lo=0
hi=len(A)
while lo < hi do
      m = (lo+hi)/2
      if x < A[m] then hi = m
                  else lo = m + 1
return lo
```



RBisect: analisi

```
RBisect( A, x)
lo=0
hi=len(A)
while lo < hi do
      m = (lo+hi)/2
      if x < A[m] then hi = m
                  else lo = m + 1
return lo
```

Gli assegnamenti e i confronti all'interno del ciclo sono ripetuti finchè vale lo < hi. Detto c_r il costo dei confronti e degli assegnamenti all'interno del ciclo while e d_r quello delle operazioni al di fuori, ci chiediamo quante volte viene eseguito il ciclo.

A ogni ripetizione il numero, n, degli elementi si dimezza. Quante volte posso dividere n per 2 prima di ottenere un risultato ≤ 1?

Se $n = 2^k$ allora dopo k divisioni per 2 si ottiene 1. Altrimenti $2^k \le n < 2^{k+1}$ allora e ancora dopo k divisioni intere per 2 si ottiene 1. Quindi concludiamo che il ciclo viene ripetuto $\lceil \log_2 n \rceil$ volte.

Abbreviazione per il logaritmo più usato: $\lg n = \log_2 n$

Inserimento in un array ordinato: soluzione 2

Trovata la posizione, bisogna inserire l'elemento. Si può usare la nome_lista.insert(*i*,el) che abbiamo già analizzato.
Nel caso peggiore per questa operazione si ha:

$TMAX_{insPos}(n)=c_i * n + d_i$ **, dove il pedice i sta per inserimento.**

A $TMAX_{insPos}(n)$ **si deve sommare il costo della ricerca della posizione**

$TMAX_{RBisect}(n)=c_r * \lg n + d_r$**, dove il pedice r sta per ricerca, ottenendo**

$$TMAX_{insOrd}(n) = TMAX_{RBisect}(n) + TMAX_{insPos}(n)= c_r * \lg n + d_r + c_i * n + d_i$$

Un'osservazione e una domanda:

1. **Espressione per** $TMAX_{insOrd}$ **è troppo complicata**

2. Quale tra le due soluzioni è migliore per l'inserimento in un array ordinato?