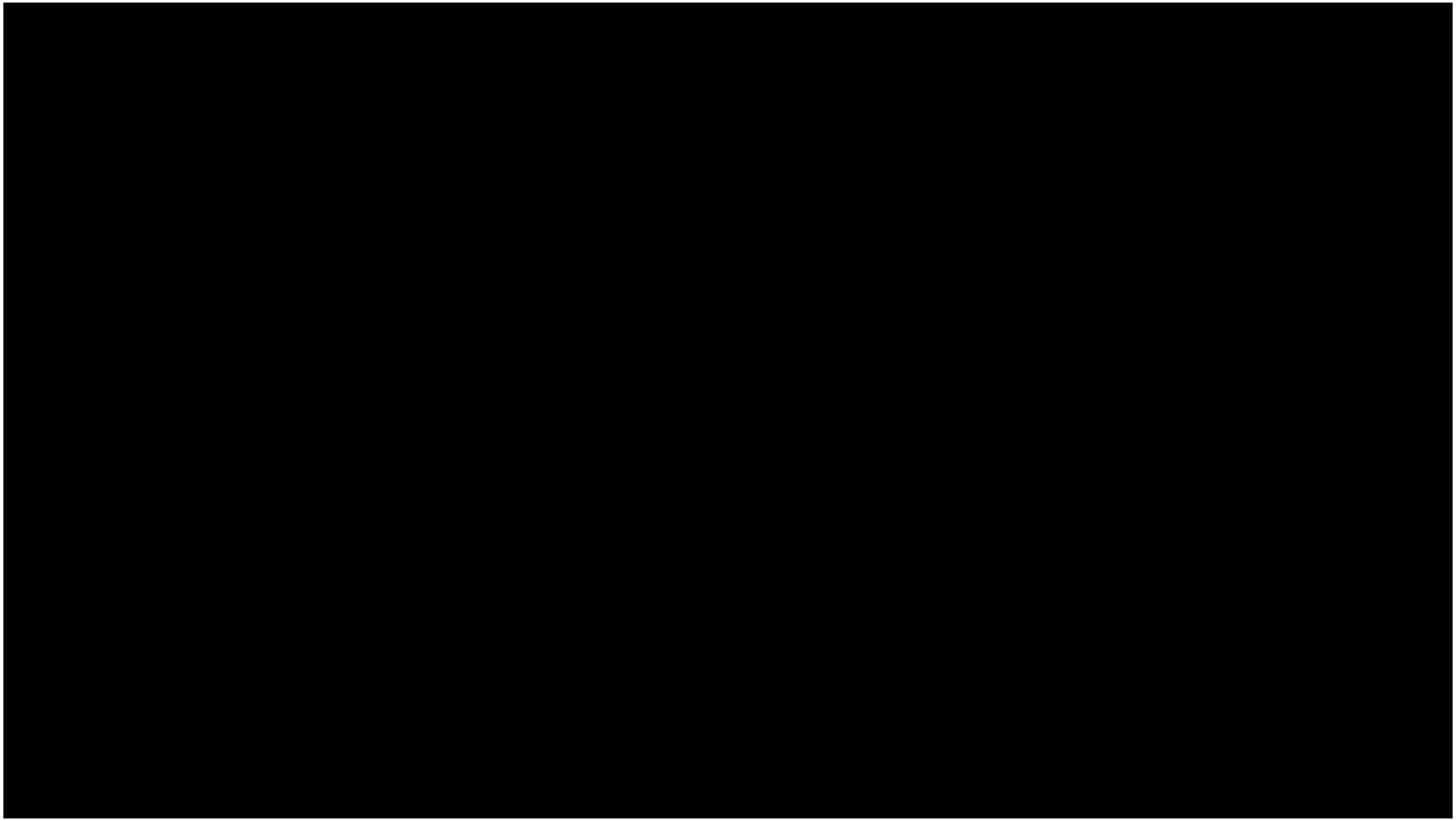
Laboratorio di Algoritmi e Strutture Dati I

Esercitazione 1

Esercitazione 1: Selection Sort

- Algoritmo di ordinamento con complessità $O(n^2)$;



Esercizio 1_1: Selection Sort iterativo

- Implementare il **Selection Sort** nella sua versione iterativa.
- Dichiarare un array di N elementi seguendo le seguenti direttive:
  1. Popolare l'array con elementi ordinati (1, 2, 3, 4, 5…);
  2. Con elementi inversamente ordinati (…5, 4, 3, 2, 1);
  3. Con elementi parzialmente ordinati (1, 2, 3, 4, 5, 43, 7, 123, 12, 0, 97…)
  4. Con elementi random (… 79, 43, 99, 1, 67…)
- Testare l'algoritmo con N = {100, 1.000, 10.000, 100.000, 200.000 e 500.000}
- Misurare i tempi di esecuzione per ogni dimensione N e per ogni tipologia di array indicata e stampare il tempo richiesto.

```
algoritmo selectionSort(array A, int n)

for i ← 0 to n-2 do
   min ← i
   for j ← i+1 to n-1 do
        if  (A[j] < A[min]) then  min ← j
   scambia A[min] con A[i]
```

Esercizio 1_1: Funzioni utili

- Funzione che genera l'array della dimensione e della tipologia desiderata:

```
typedef enum{ORDINATO, QUASI_ORDINATO, INV_ORDINATO, CASUALE} inputType;

algoritmo genera_array(int dimensione, inputType tipo_schema) → int *
```

- Calcolare i tempi di esecuzione per ciascun test (questo NON è pseudo-codice):

```
#include <time.h>

...

clock_t  start, end;
double t;

...

start = clock();
<chiamata all'algoritmo di ordinamento>
end = clock();
t = ((double) (end - start)) / CLOCKS_PER_SEC;

printf("tempo impiegato: %lf secondi", t);
```

- Una funzione di swap

Esercizio 1_2: difficoltà++

- Creare e popolare un array di strutture che rappresentano delle ricette.
- Una Ricetta dovrà essere così definita:

```
typedef struct
{
      char nome[DIM_NOME];
      double tempo;
      int difficolta;
} Ricetta;
```

Nella quale:

- **Nome** rappresenta il titolo della ricetta;
- **Tempo** rappresenta i minuti necessari per la sua realizzazione
- **Difficoltà** è rappresentata da un intero compreso tra 1 (molto semplice) e 10 (estremamente complessa).

Esercizio 1_2: difficoltà++

- Riscrivere il Selection Sort per ordinare gli elementi di tipo Ricetta in base alla loro durata (dalla più breve alla più lunga) e, a parità di durata, in base alla loro difficoltà (dalla più semplice alla più complessa).

- Definire una funzione ***compare*** per il confronto tra Ricette in questo modo:

$$compare\ (r1, r2) = \begin{cases} 0 & se\ r1.tempo < r2.tempo \\ 0 & se\ r1.tempo = r2.tempo\ AND\ r1.difficolta < r2.difficoltà \\ 1 & altrimenti \end{cases}$$

- Definire una funzione swap che scambia due ricette.

- I dati relativi alle ricette devono essere inseriti in input dall'utente.

- In questa versione l'array può essere STATICO.

Esercizio 1_2: difficoltà++

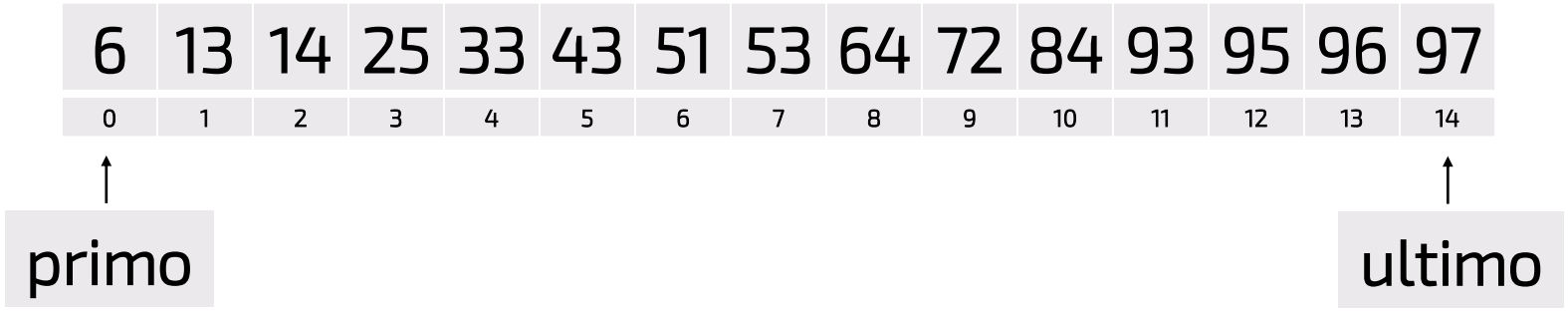
- Esempio:

```
Prima:

nome: Pasta alle vongole
tempo: 20.00
diff.: 3

nome: Pollo alle mandorle
tempo: 30.00
diff.: 5

nome: Zuppa di pesce
tempo: 60.00
diff.: 7

nome: Pasta ai carciofi
tempo: 30.00
diff.: 2
```

```
Dopo:

nome: Pasta alle vongole
tempo: 20.00
diff.: 3

nome: Pasta ai carciofi
tempo: 30.00
diff.: 2

nome: Pollo alle mandorle
tempo: 30.00
diff.: 5

nome: Zuppa di pesce
tempo: 60.00
diff.: 7
```

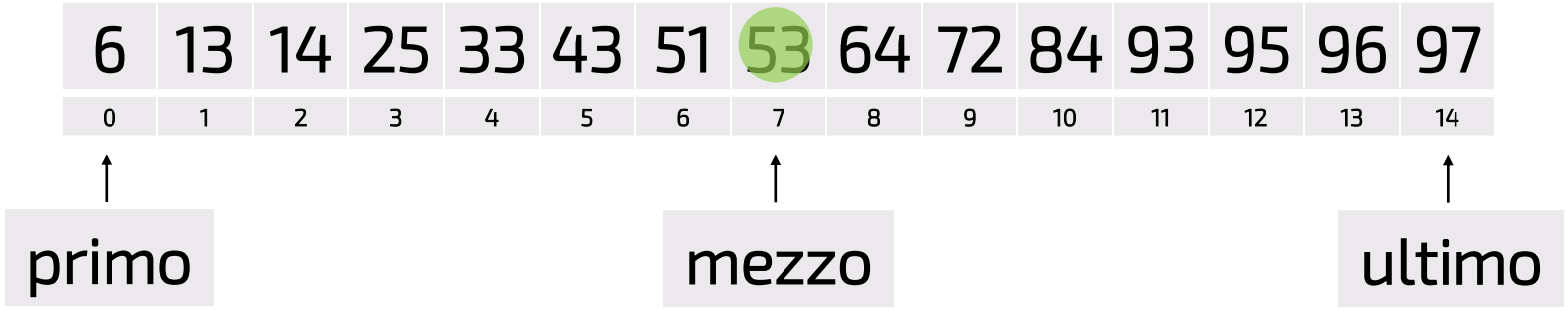
Lezione 1: Ricerca binaria

- Algoritmo di ricerca fondamentale in Computer Science.
- **Prerequisito**: la sequenza di valori, in cui si effettua la ricerca di un elemento, deve essere **ordinata**.
- L'algoritmo lavora su uno "spazio di ricerca": inizialmente l'intera sequenza, che si riduce di metà a ogni passo.
  - Complessità O(log n).
- Esempio: supponiamo di voler cercare il valore **33.**



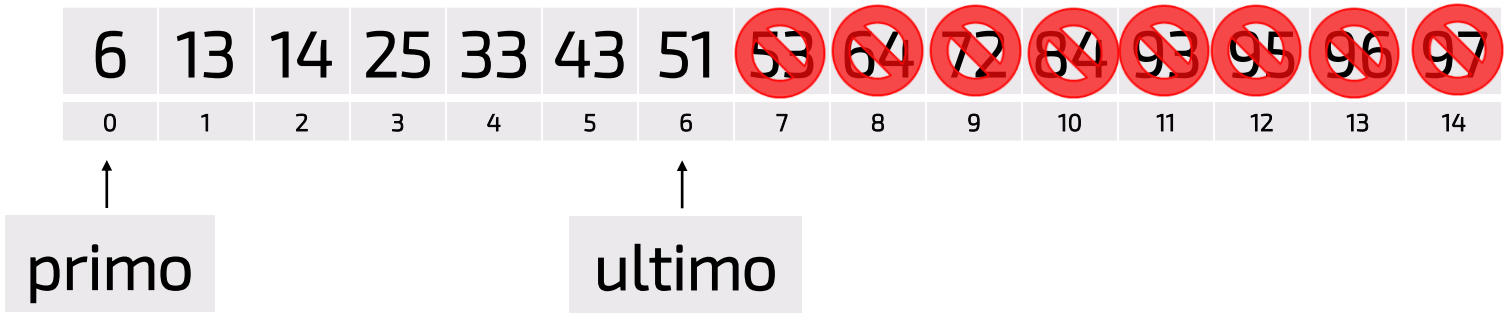
Lezione 1: Ricerca binaria

- Esempio: supponiamo di voler cercare il valore **33.**
  - Accediamo all'elemento **centrale** della sequenza considerata (53)



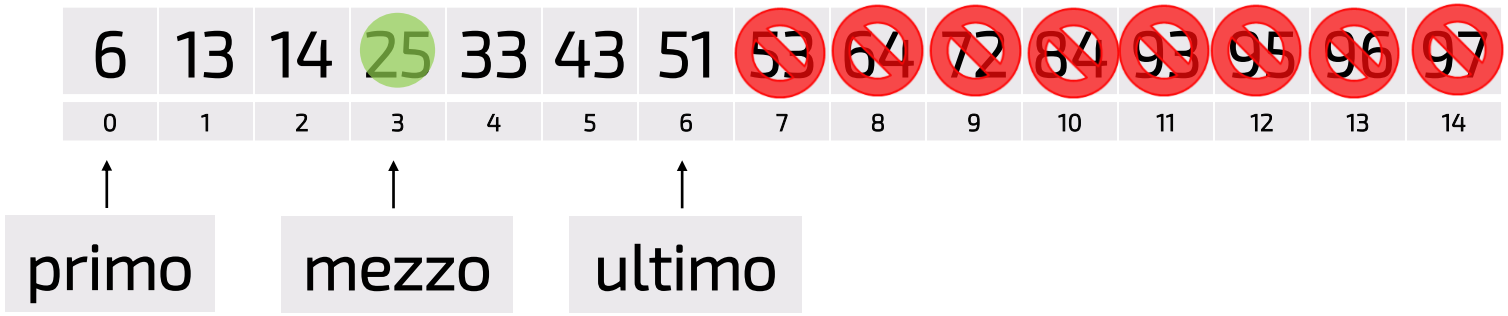
Lezione 1: Ricerca binaria

- Esempio: supponiamo di voler cercare il valore **33.**
  - Accediamo all'elemento **centrale** della sequenza considerata (53)
  - 33 != 53 quindi proseguiamo
  - 33 < 53 quindi **ultimo** diventa **mezzo-1** (ultimo punta a 51)



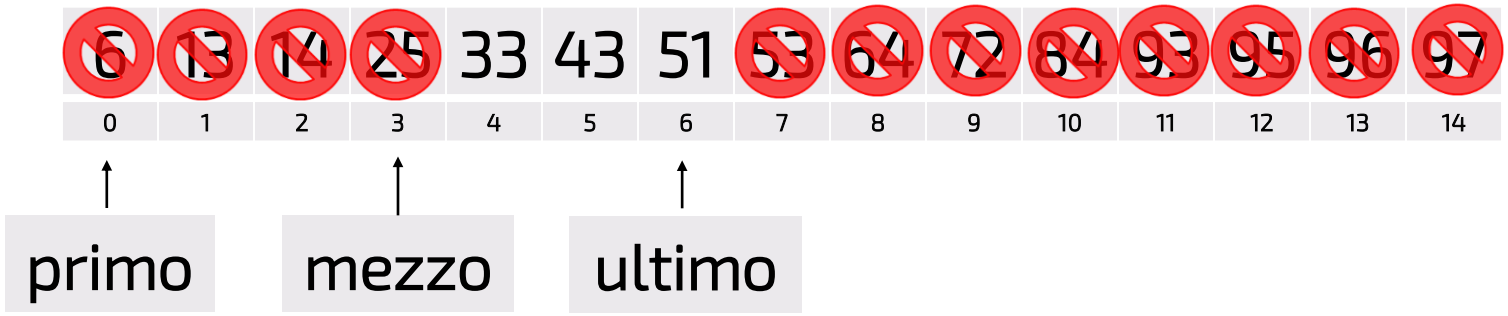
Lezione 1: Ricerca binaria

- Esempio: supponiamo di voler cercare il valore **33.**
  - Accediamo all'elemento **centrale** della sequenza considerata (25)



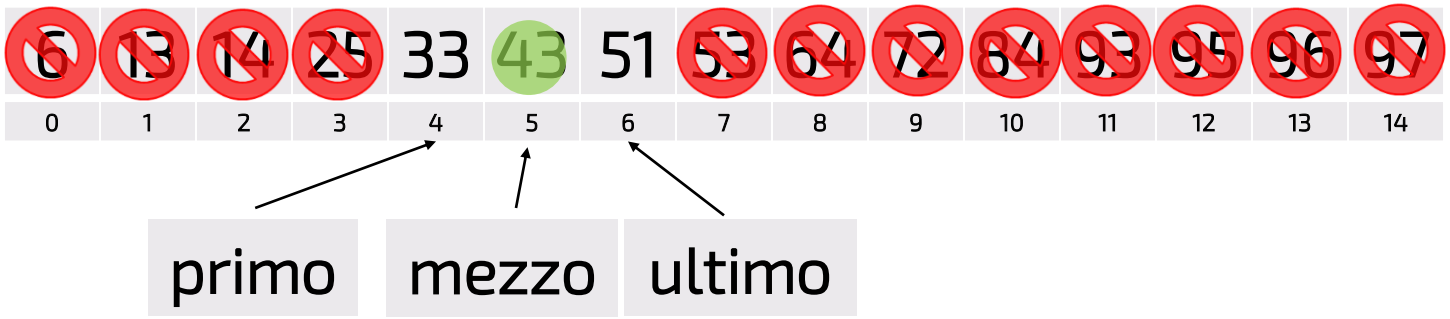
Lezione 1: Ricerca binaria

- Esempio: supponiamo di voler cercare il valore **33.**
  - Accediamo all'elemento **centrale** della sequenza considerata (25)
  - 33 != 25 quindi proseguiamo
  - 33 > 25 quindi **primo** diventa **mezzo+1** (primo punta a 33)



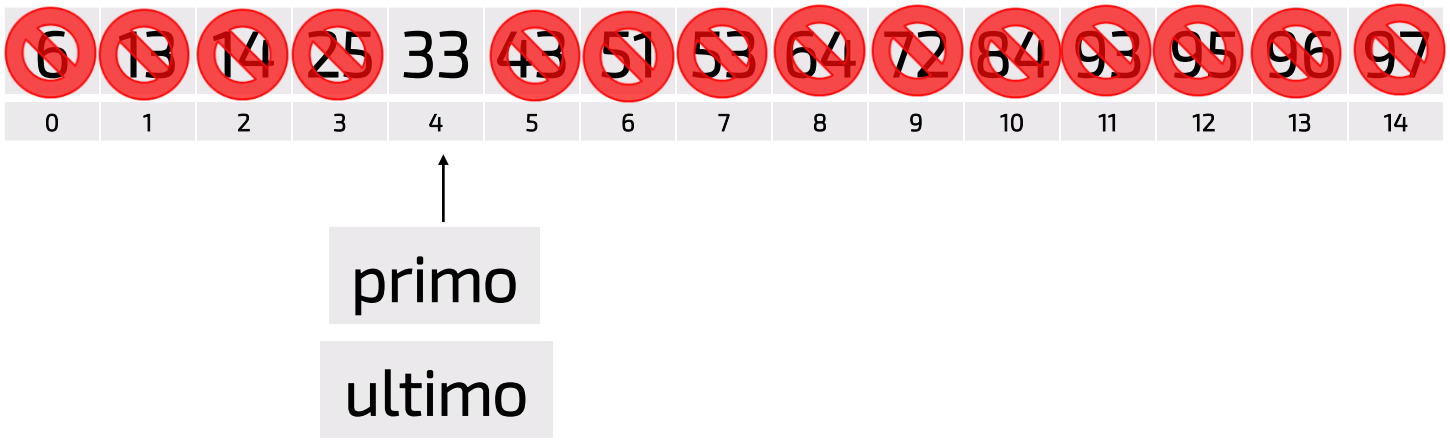
Lezione 1: Ricerca binaria

- Esempio: supponiamo di voler cercare il valore **33.**
  - Accediamo all'elemento **centrale** della sequenza considerata (43)
  - 33 != 43 quindi proseguiamo
  - 33 < 43 quindi **ultimo** diventa **mezzo-1** (ultimo punta a 33)



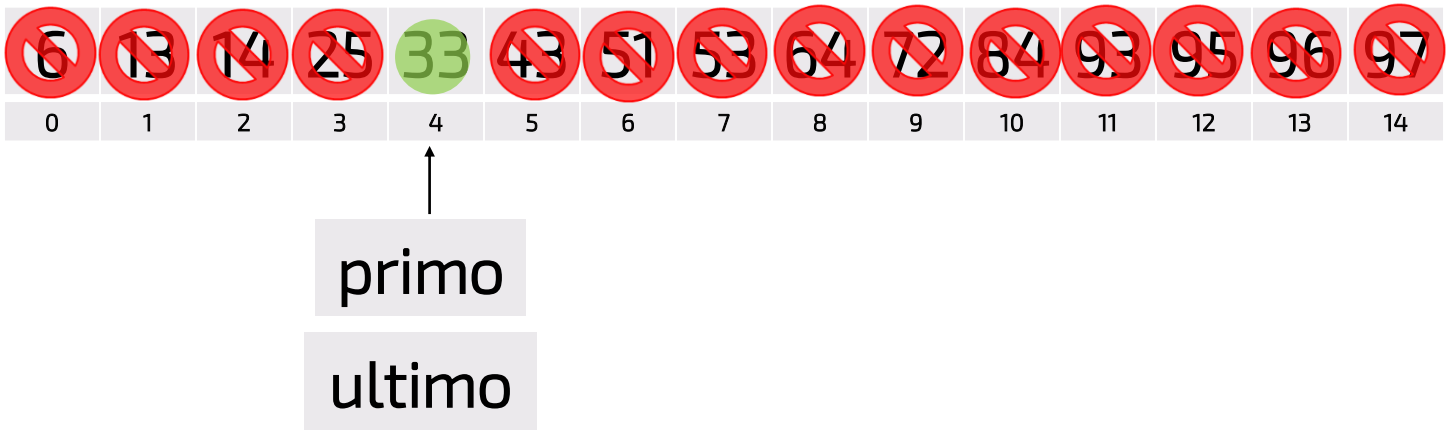
Lezione 1: Ricerca binaria

- Esempio: supponiamo di voler cercare il valore **33**.
  - **primo** e **ultimo** puntano allo stesso elemento
  - Abbiamo trovato il numero cercato e ci fermiamo



Lezione 1: Ricerca binaria

- Esempio: supponiamo di voler cercare il valore **33**.
  - **primo** e **ultimo** puntano allo stesso elemento
  - Abbiamo trovato il numero cercato e ci fermiamo



Lezione 1: Ricerca binaria

- Esempio: supponiamo di voler cercare il valore **33**.
  - **primo** e **ultimo** puntano allo stesso elemento
  - Abbiamo trovato il numero cercato e ci fermiamo

Ricerche a confronto

- Create un nuovo progetto.
- Generare con la funzione *genera_array(...)* dell'esercizio precedente degli array di interi **random** prevedendo le seguenti dimensioni: 10, 50, 100, 1.000, 10.000, 100.000 e 500.000.
- Ordinare gli array utilizzando il Selection Sort
- Testare la ricerca di un elemento (intero) all'interno degli array con gli algoritmi di ricerca presenti nelle prossime slide.
- Contare il numero di elementi dell'array analizzati per trovare la posizione del numero cercato (o per capire che esso non è presente).
  - ATTENZIONE: Considerare SOLO gli elementi dell'array nel conteggio (non gli indici utilizzati per scorrerlo).
  - Le variabili utilizzate per i conteggi possono essere globali.

Esercizio 1_3: Ricerca lineare (opzionale)

- Implementare una ricerca banale (la cosiddetta ricerca lineare o sequenziale).
  - Complessità O(n).

- Sfrutteremo questo algoritmo esclusivamente per fare i confronti (in termini di efficienza) con la ricerca binaria.

- Modificare la ricerca per contare il numero di elementi dell'array analizzati per trovare la posizione del numero cercato (o per capire che esso non è presente). Utilizzare una variabile intera "contL".
- NB: Misurare il tempo (oltre che il numero di elementi verificati).

```
algoritmo ricercaBanale(array A, intero numric, intero dim) → intero

while (pos < dim and A[pos] ≤ numric) do
    if (numric == A[pos])
          then return pos
    else
          incrementa pos
return -1
```

Esercizio 1_4: Ricerca binaria iterativa

- Implementare la ricerca binaria iterativa.

- Modificare la ricerca per contare il numero di elementi dell'array analizzati per trovare la posizione del numero cercato (o per capire che esso non è presente). Utilizzare una variabile intera "contBI".
- NB: Misurare il tempo (oltre che il numero di elementi verificati).

```
algoritmo ricercaBinariaIter(array A, intero numric, intero dim) → intero

primo ← 0
ultimo ← dim - 1
while (primo ≤ ultimo) do
    mezzo ← (primo + ultimo)/2
    if (numric < A[mezzo])
          then ultimo ← mezzo-1
    else if (numric == A[mezzo])
          then return mezzo
    else
          primo ← mezzo+1
return -1
```

Esercizio 1_5: Ricerca binaria ricorsiva

- Implementare la ricerca binaria ricorsiva.

- Modificare la ricerca per contare il numero di elementi dell'array analizzati per trovare la posizione del numero cercato (o per capire che esso non è presente). Utilizzare una variabile intera "contBR".
- NB: Misurare il tempo (oltre che il numero di elementi verificati).

```
algoritmo ricBinRic(array A, intero numric, intero primo, intero ultimo) → intero

if (primo > ultimo)
    then return -1

mezzo ← (primo + ultimo)/2
if (A[mezzo] == numric)
    then return mezzo
else if (A[mezzo] < numric)
    then return ricBinRic(A, numric, mezzo+1, ultimo)
else
    return ricBinRic(A, numric, primo, mezzo-1)
```

Ricerche a confronto

- Osservare i risultati ottenuti, in termini di numero di confronti e tempi, con i vari algoritmi di ricerca e riflettere su:

- Qual è la ricerca più efficiente in termini di tempo?

- Qual è la ricerca più efficiente in termini di numero di elementi dell'array analizzati?

- Che differenze esistono tra la versione iterativa e quella ricorsiva della Ricerca Binaria?

Lezione 1 : Ricapitolando

Implementare:

- Funzione genera_array(…)

- E1_1: Selection Sort iterativo, con misurazione tempi di esecuzione
- E1_2: Selection Sort per ricette

- E1_3: Ricerca Lineare, con conteggi e tempi di esecuzione (opzionale)
- E1_4: Ricerca Binaria Iterativa, con conteggi e tempi di esecuzione
- E1_5: Ricerca Binaria Ricorsiva, con conteggi e tempi di esecuzione

- Ragionare su quanto osservato!

Esercitazione 1

end().