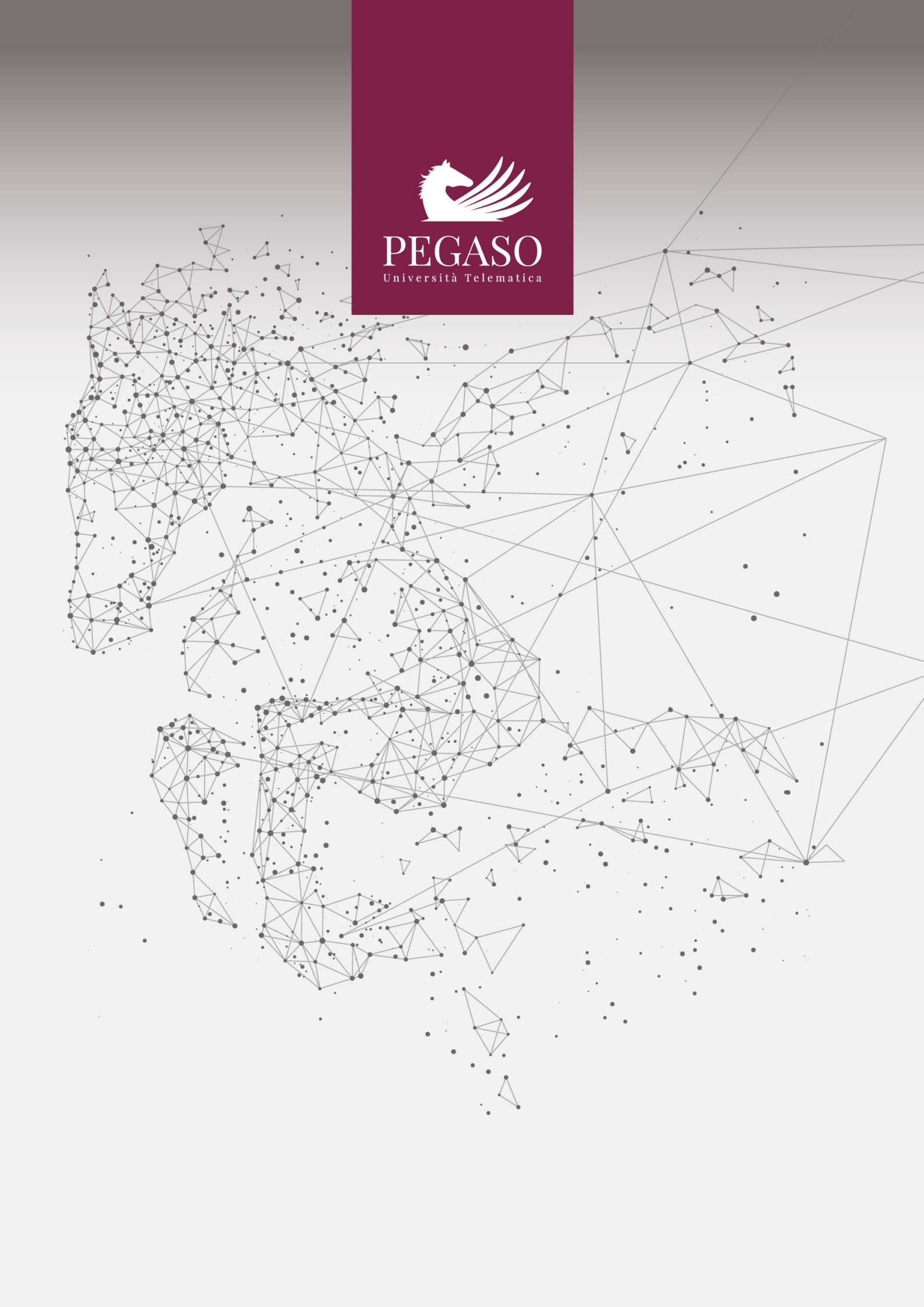
PEGASO
Università Telematica

Indice

1. Introduzione alla struttura dati

Quando parliamo di un array, possiamo immaginare una sorta di contenitore, le cui caselle sono dette celle (o elementi) dell'array stesso. Ciascuna delle celle si comporta come una variabile tradizionale: tutte le celle sono variabili di uno stesso tipo preesistente, detto tipo base dell'array (si parlerà perciò di tipi come "array di interi", "array di stringhe", "array di caratteri" ecc.).

Quello che si ottiene dichiarandolo è dunque un contenitore statico ed omogeneo di valori, variabili o oggetti:

1. statico: i suoi elementi non variano di numero a tempo d'esecuzione;
2. omogeneo: i suoi elementi sono tutti dello stesso tipo.

Ciascuna delle celle dell'array è identificata da un valore di indice. L'indice è generalmente numerico è parte dallo 0: si potrà quindi parlare della cella di indice 0, di indice 1, e, in generale, di indice N, dove N è un intero compreso fra 0 e il valore massimo per gli indici dell'array.

La possibilità di accedere agli elementi attraverso un indice è la principale caratteristica di un array.

È possibile accedere singolarmente ad una sua generica posizione ("accesso casuale", come per la memoria), oltre a scorrerlo sequenzialmente in entrambe le direzioni tramite un ciclo iterativo in tutti i suoi elementi o a partire da alcuni di essi.

Gli elementi di un array sono memorizzati consecutivamente e individuati attraverso la loro posizione: l'indirizzo di ciascuno di essi è determinato dalla somma dell'indirizzo del primo elemento dell'array e dell'indice che individua l'elemento in questione all'interno dell'array.

Poiché l'accesso a ciascun elemento di un array è diretto, l'operazione di lettura o modifica del valore di un elemento di un array ha complessità asintotica $O(1)$ rispetto al numero di elementi dell'array.

2. Problema della ricerca – analisi in C++

La formulazione del "problema della ricerca" in un array è la seguente:

dati un array e un valore, stabilire se il valore è contenuto in un elemento dell'array, riportando in caso affermativo l'indice di tale elemento

Lo specifico elemento da ricercare è tipicamente chiamato "chiave".

Iniziamo con una "visita" dell'array:

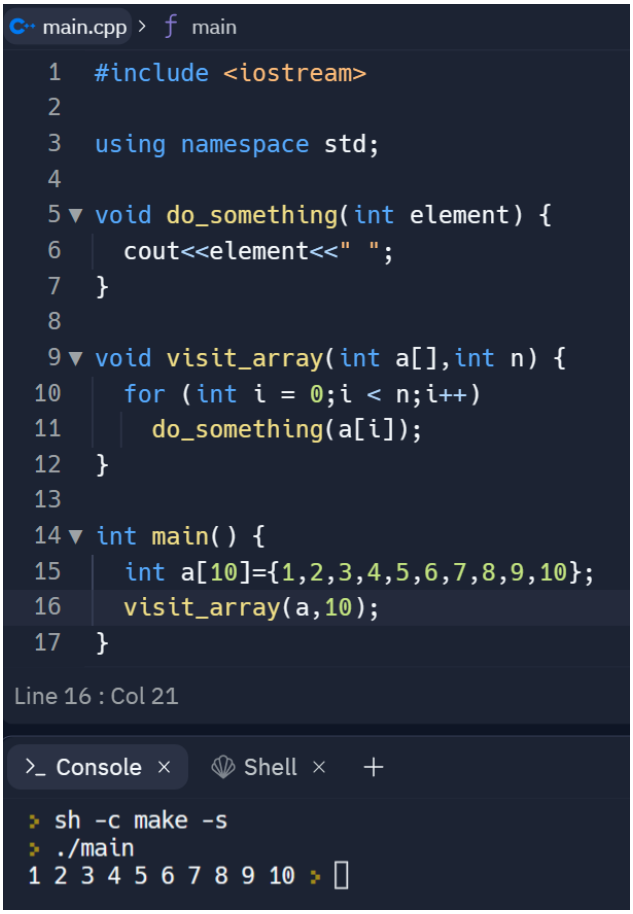

Figura 1 *- Visita dell'array*

In questo esempio la visita dell'array prevede che per ogni elemento analizzato venga semplicemente eseguito un output su console (questo è quanto gestito all'interno della funzione $do\_something$), tuttavia si può generalizzare ed immaginare che l'elaborazione dell'elemento abbia un costo d.

$$T(n) = n \cdot (1 + d + 1) = n \cdot (d + 2) = O(n)$$

La complessità è pertanto lineare.

Analizziamo ora la ricerca "lineare". In questo caso gli elementi dell'array vengono attraversati uno dopo l'altro nell'ordine in cui sono memorizzati finché il valore cercato non viene trovato o la fine dell'array non viene raggiunta:

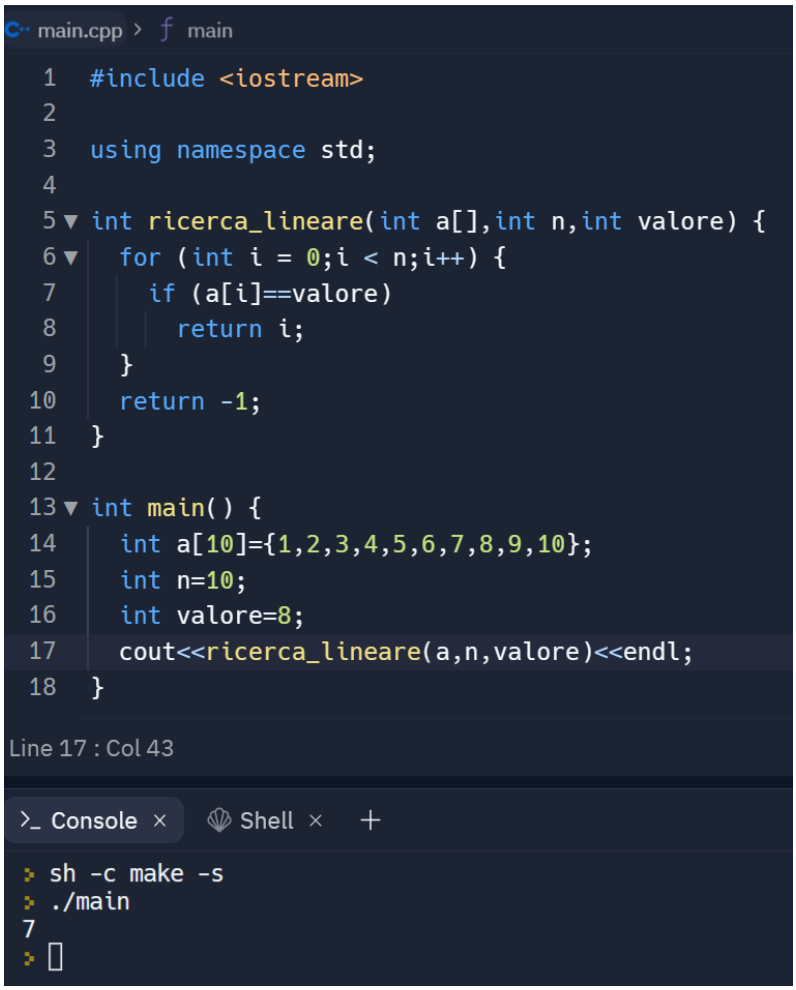

Figura 2 *- Ricerca Lineare*

Anche in questo caso la complessità è lineare e possiamo distinguere tra il caso ottimo, in cui la chiave è al primo elemento e dunque: $T(n) = O(1)$ oppure il caso peggiore in cui si deve scorrere tutto l'array perché l'elemento non è presente o è all'ultimo posto: $T(n) = O(n)$

Analizziamo ora il codice relativo alla ricerca binaria, cioè alla ricerca di una chiave all'interno di un array ordinato:

```
main.cpp > ƒ main
12
13 ▼ int ricerca_binaria_non_ricorsiva(int a[],int n,int valore) {
14     int sx=0;
15     int dx=n-1;
16     int mx=(sx+dx)/2;
17 ▼   while (sx!=dx) {
18       if (a[mx]==valore)
19         return mx;
20       if (a[mx]<valore)
21         sx=mx+1;
22       else
23         dx=mx-1;
24       mx=(sx+dx)/2;
25     }
26
27     return (a[mx]==valore)?mx:-1;
28   }
29
30 ▼ int main() {
31     int a[11]={1,2,3,4,5,6,7,8,9,10,11};
32     int n=11;
33     int valore=8;
34     cout<<ricerca_binaria_non_ricorsiva(a,n,valore)<<endl;
35   }
```

Figura 3 *- Ricerca binaria non ricorsiva*

È possibile costruire un algoritmo ricorsivo, tuttavia in questa fase vogliamo limitarci a considerare una soluzione non ricorsiva: si può procedere dimezzando lo spazio di ricerca ogni volta. L'idea è di confrontare il valore cercato col valore dell'elemento che sta nella posizione di mezzo dell'array:

1. se l'elemento nel mezzo è la chiave, si restituisce l'indice altrimenti si prosegue al punto 2;
2. se non si può più suddividere l'array si restituisce -1;
3. nel caso in cui l'elemento nel mezzo è più piccolo del valore ricercato si ricercherà nella metà a destra ripartendo dal punto 1;
4. nel caso in cui l'elemento nel mezzo è più grande del valore ricercato si ricercherà nella metà a sinistra ripartendo dal punto 1;

Il caso ottimo si verifica quando il valore cercato è presente nell'elemento che sta nella posizione di mezzo dell'array; in questo caso:

$$T(n) = O(1)$$

Nel caso pessimo lo spazio di ricerca viene ripetutamente diviso a metà fino a restare con un unico elemento da confrontare con il valore cercato.

Se indichiamo con k il numero di iterazioni, questo valore coincide con il numero di dimezzamenti dello spazio di ricerca e nel caso pessimo $k = \log_2 n$ pertanto:

$$T(n) = \log_2 n$$

L'algoritmo di ricerca binario è dunque più efficiente di quello di ricerca lineare ma è meno generale, in quanto può essere applicato solo ad array ordinati.

3. Problema della ricerca – analisi in Python

Analizziamo lo stesso problema costruendo i corrispondenti algoritmo in Python, usando le Python List.

Iniziamo con la ricerca lineare:

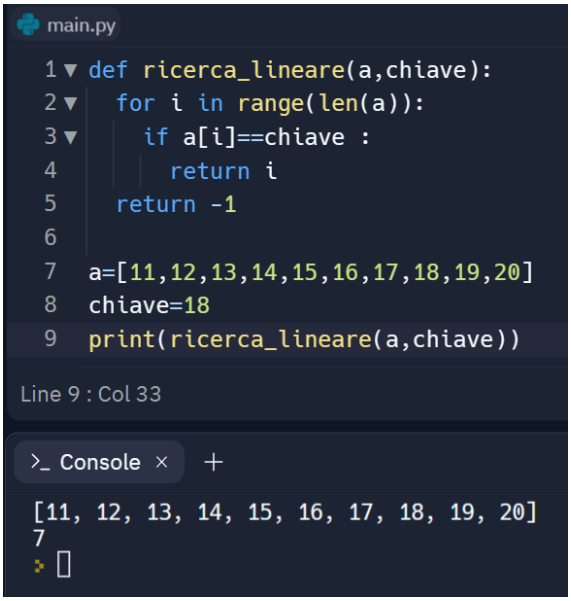

Figura 4 *- Ricerca Lineare in Python*

Anche in questo caso possiamo verificare che la complessità è la stessa.

Proseguendo con la ricerca binaria non ricorsiva si può verificare facilmente che valgono le stesse considerazioni fatte per l'implementazione in C++:

- Il caso ottimo si verifica quando il valore cercato è presente nell'elemento che sta nella posizione di mezzo dell'array; in questo caso: $T(n) = O(1)$
- Nel caso pessimo lo spazio di ricerca viene ripetutamente diviso a metà fino a restare con un unico elemento da confrontare con il valore cercato. Se indichiamo con k il numero di iterazioni, questo valore coincide con il numero di dimezzamenti dello spazio di ricerca e nel caso pessimo $k = \log_2 n$ pertanto: $T(n) = \log_2 n$

```python
def ricerca_binaria_non_ricorsiva(a,valore):
    sx=0
    dx=len(a)
    mx=(sx+dx)//2

    while True:
        if mx<0 or mx>=len(a):
            return -1
        if a[mx]==valore:
            return mx
        if a[mx]<valore:
            sx=mx+1
        else:
            dx=mx-1

        mx=(sx+dx)//2

a=[11,12,13,14,15,16,17,18,19,20]
chiave=25
print(ricerca_binaria_non_ricorsiva(a,chiave))
```

Figura 5 *- Ricerca binaria non ricorsiva in Python*

Uno dei vantaggi che si ha in Python è la possibilità di costruire degli unit tests per poter testare tutte gli input "interessanti":

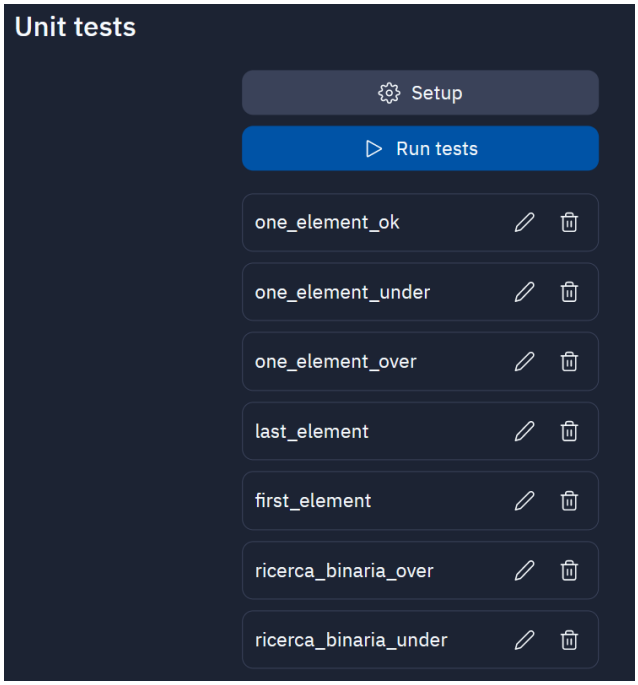

Figura 6 *- Unit Tests*

Analizziamo di seguito un esempio di Unit test costruito per verificare che l'output sia corretto nel momento in cui si stia ricercando un valore in un array che possiede l'elemento ricercato nell'ultima posizione:

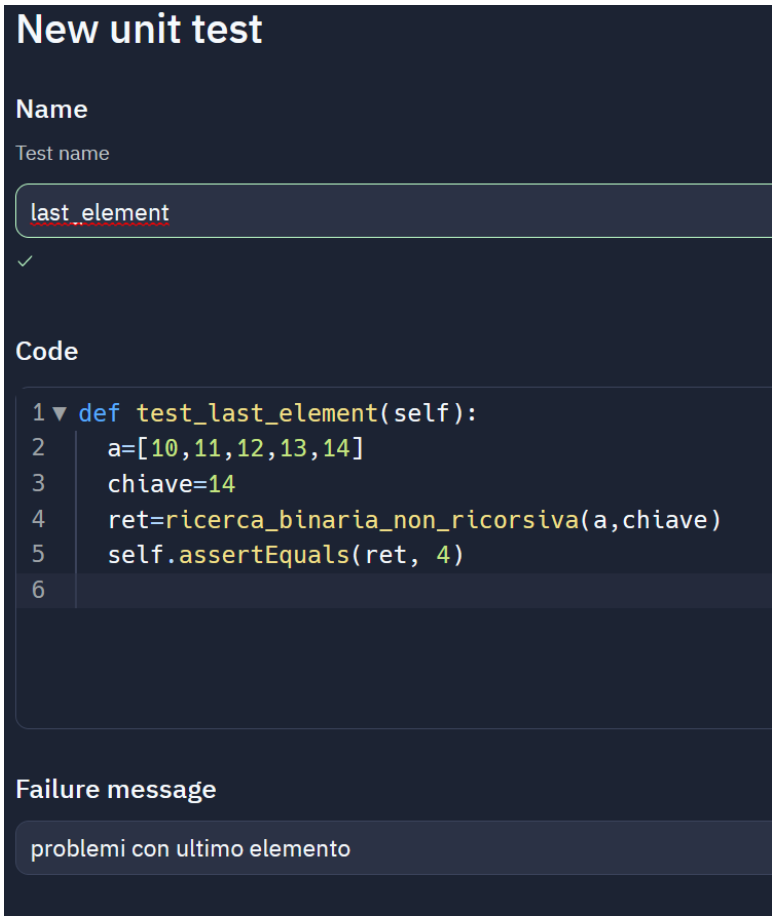

Figura 7 *- Unit Test per elemento in ultima posizione*

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley
- Sedgewick: Algoritmi in C, Pearson, 2015
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione