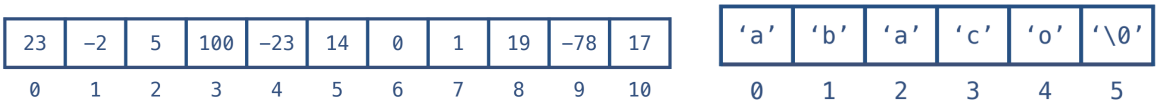
2 Array

Una struttura dati molto conosciuta e chiamata array.

Definizione 2.1 (Array)**.** *Gli array sono delle strutture dati omogenea, statiche e lineari implementate mediante un gruppo di celle contigue di memoria dello stesso tipo.*

Di seguito due esempi grafici di array uno di interi ed uno di stringhe, da notare sotto la posizione degli elementi nell'array che si conta partendo dallo 0.



(a) (b)

Figure 2: In (a) un array lungo 11 di interi, in (b) un array lungo 6 di caratteri

Note 2.0.1. *Nota che nell'array di caratteri sopra nell'ultima posizione c'è sempre* \0 *(Null).*

Negli array si accede mediante l'indice della posizione nella sequenza. Si possono inoltre effetturare sugli elementi tutte le operazioni definte sul tipo corrispondete agli elementi dell'array.

Esempio 2.1. Alcuni esempi di accesso ed operazioni su gli arrey sopra:

- a[6] == 0 a[3] == 100 b[2] == 'a'
- a[4] = a[5] + a[7] (a[5] == 14, a[7] == 1, quindi il risultato sarà 14 + 1 = 15)

Inoltre possiamo dire che gli array sono allocati in memoria quando il controllo del flusso a tempo di esecuzione entra nel blocco in cui sono definiti e sono distrutti quando il controllo esce dal blocco.

Il nome dell'array è una variabile che contiene la locazione di memoria in cui è memorizzata la prima cella. Essendo che le celle sono contigue e hanno tutte lo stesso tipo basta infatti conoscere la posizione della prima cella per poi, tramite una semplice operazione algebrica di somma, accedere a quelle successive. In generale possiamo scrivere che:

$$a[i] \;=\; \sigma(\rho(a) + size(type(a)) \times i)$$

Esempio 2.2. Se abbiamo un array di lunghezza 11, ed chiamiamo la prima locazione (quella dove è contenuto il primo elemento dell'array) loc1, per raggiungere la posizione numero 10 basterà eseguire l'operazione loc1 + 32*10.

Questo consente l'accesso diretto agli elementi degli array con una sola operazione indipendentemente dalla lunghezza dell'array (consto di accesso costante).

2.1 BinarySearch

Problema: Dato un elemento (o chiave) k, determinare se esiste all'interno di un array ordinato A di n elementi. Se l'elemento esiste, si restituisce la sua posizione, altrimenti -1. Soluzione con ricerca binaria.
Proprietà: $\forall i \in [0..n-1]\,.\,A[i] \leq A[i+1]$
Questa proprietà dice che l'array A deve essere obbligatoriamente ordinato, sennò la ricerca binaria non potrà esser fatta.

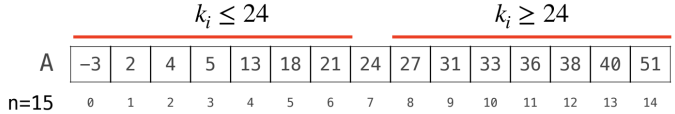


Figure 3: Array A in BinarySearch

2.1.1 Codice dell'algoritmo

```
1  function binSearch(k,A) {
2      var pos:Int = -1;
3      var sin:int = 0;
4      var dx:Int = n - 1;
5      while(sin <= dx && pos == -1){
6          const cen:int = (sin + dx)/2;
7          if (A[cen] == k) {pos = cen}
8          else if (k < A[cen]) {dx = cen - 1}
9          else {sin = cen + 1}
10     }
11     return pos;
12 }
```

Listing 2: Codice BinarySearch

Di seguito una spiegazione del funzionamento dell'algoritmo:

- **Righe 2-4:** Andiamo ad inizializzare 3 variabili: "pos" che indicherà la posizione dell'elemento da cercare, viene inizializzata a -1 perché nel caso non si trovasse ritorna così -1.
  "Sin" che indica il capo sinistro della posizione che stiamo analizzando, e "dx" che indica il capo destro, sono entrambi inizialmente inizializzati come gli estremi dell'array.
- **Riga 5:** La condizione del while dice in sintesi che finché non abbiamo trovato il valore (pos == -1) e finchè "sin" e "dx" non si scambiano (che vorrebbe dire che abbiamo finito le iterazioni possibili), continuare a ciclare.
- **Righe 6-9:** All'interno del while quello che andiamo a fare e prendere il centro della porzione dell'array che stiamo considerando (inizialmente il centro dell'interno array) e vedere se il valore che dobbiamo cercare si trova in quella posizione, e in tal caso finiamo, è minore, e quindi si troverà alla sinistra del centro, o maggiore, in tal caso si troverà alla destra; nel caso non si sia trovato ci spostiamo ad analizzare la parte destra o sinistra asseconda del risultato. Eseguiamo questa operazioni finché è consentito dal ciclo.

Note 2.1.1. *Nota che a noi non ci importa se la porzione è pari o dispari, quello che ci ritornerà esclude il resto.*

Esempio 2.3. Esempio con l'array in figura 3 cercando il valore 18.

pos	sin	dx	cen	A[cen]
-1	0	14	7	24
-1	0	6	3	5
-1	4	6	5	**18**

Iterazioni	Dimensione A
1	$n = n/2^0$
2	$n/2 = n/2^1$
3	$n/4 = n/2^2$
...	...

Table 1: Esempio di funzionamento dell'algoritmo a sinistra e numero iterazione a destra

2.1.2 Calcolo caso pessimo e migliore

Per calcolare il caso pessimo partiamo guardando la tabella sopra, notiamo che in questo algoritmo verranno eseguite $n/2^i$ operazioni, quindi il massimo possibile dipende da quanto è grande i. Per andare a trovare i basta:

$$n/2^i = 1 \qquad n = 2^i \qquad \log_2 n = \log_2 2^i \qquad i = \log_2 n \in O(\log_n)$$

Questo caso è o quando k si trova agli estremi o quando k non c'è nell'array, e quindi ritorna -1.