DOMANDA: Illustrare l'Algoritmo di Ricerca Binaria, versione iterativa (ARBINi)

RISPOSTA:
L'algoritmo di Ricerca Binaria è un algoritmo di ricerca di una chiave in un array ORDINATO di dati.
I dati possono essere numeri, caratteri, stringhe di caratteri o qualsiasi altro insieme di dati su cui sia definita una relazione di ordine.
L'algoritmo di Ricerca Binaria è un algoritmo ottimale per tale problema e la sua complessità di tempo asintotica è logaritmica.

L'ARBINi ha come dati di input la chiave di ricerca, un array e il suo size, e ha come dato di output un dato scalare intero che è la posizione della prima occorrenza della chiave nell'array oppure il valore convenzionale -1 nel caso in cui la chiave non appartenga all'array.
L'ARBINi non fa uso di altri array o strutture dati e per questo motivo si dice che opera "in situ" o "in place".

L'ARBINi è basato sull'applicazione dell'approccio divide et impera al problema della ricerca in array ordinati. Per tale problema, l'approccio divide et impera risulta particolarmente efficiente, in quanto consente a ogni passo non solo di suddividere una data istanza in due istanze di dimensioni dimezzate, ma soprattutto di eliminare una delle due istanze continuando la ricerca solo su una delle due porzioni dimezzate. Infatti, a ogni passo l'algoritmo risolve una istanza del problema della ricerca in un array ordinato, cioè su una "porzione" dell'array A, e a ogni passo la dimensione della "porzione" si dimezza, fino ad arrivare eventualmente alla dimensione 1, che è la dimensione dell'istanza banale del problema della ricerca.
Diamo una rapida analisi della meccanica dell'ARBINi. Si indicheranno con gli indici `primo` e `ultimo` rispettivamente la posizione iniziale e la posizione finale della porzione di array su cui si effettua la ricerca. Al primo passo, `primo` vale 0 e `ultimo` vale (n-1). Al generico passo di iterazione, si deve effettuare la ricerca sulla porzione `primo..ultimo` dell'array. Le azioni di un passo sono: la determinazione della posizione "centrale" della porzione (indice `mediano`); il confronto a tre vie tra la chiave e l'elemento dell'array che si trova nella posizione "centrale": se i due valori sono uguali, allora l'algoritmo termina restituendo tale posizione, altrimenti se la chiave è minore la ricerca proseguirà al passo successivo sulla semi-porzione "di sinistra", cioè la porzione `primo..(mediano-1)`, mentre se la chiave è maggiore la ricerca proseguirà al passo successivo sulla semi-porzione "di destra", cioè la porzione `(mediano+1)..ultimo`.
Se la chiave non appartiene all'array, il processo iterativo terminerà quando la porzione su cui agire è vuota, ovvero costituita da nessun elemento, situazione questa indicata dal fatto che l'indice `primo` ha un valore maggiore di quello dell'indice `ultimo`.
Un esempio aiuta a capire la dinamica dell'ARBINI. Si consideri la chiave 52 e un array A di n=8 elementi interi. La posizione centrale è (0+7)/2, cioè 3

11	13	27	28	31	36	41	52
primo=0	1	2	mediano=3	4	5	6	**ultimo= n-1=7**

Quindi, al primo passo dell'ARBINi la chiave è confrontata con 28: poiché 52 è diverso da 28 l'algoritmo deve continuare, e poiché 52 è maggiore di 28 la ricerca deve proseguire sulla semi-porzione di destra 4..7 dell'array A.

11	13	27	28	31	36	41	52
0	1	2	3	**primo=4**	mediano=5	6	**ultimo= n-1=7**

Al secondo passo dell'ARBINi la chiave è confrontata con 36: poiché 52 è diverso da 36 l'algoritmo deve continuare, e poiché 52 è maggiore di 36 la ricerca deve proseguire sulla semi-porzione di destra 6..7 dell'array A.

11	13	27	28	31	36	41	52
0	1	2	3	4	5	**primo=6** mediano=6	**ultimo= n-1=7**

Al terzo passo dell'ARBINi la chiave è confrontata con 41: poiché 52 è diverso da 41 l'algoritmo deve continuare e poiché 52 è maggiore di 41 la ricerca deve proseguire sulla semi-porzione di destra 7..7 dell'array A.

11	13	27	28	31	36	41	52
0	1	2	3	4	5	6	**primo=7 ultimo= n-1=7** mediano=7

Al quarto passo dell'ARBINi la chiave è confrontata con 52 e poiché sono uguali, l'algoritmo termina restituendo come dato di output 7.

Questa è l'implementazione in C dell'ARBINi, per una chiave e un array di tipo `char`:

```
int ric_bin(char chiave,char elenco[],int n)
{
 int mediano, primo = 0, ultimo = n-1;
 while(primo <= ultimo)
 {
  mediano = (primo + ultimo)/2;
  if(chiave == elenco[mediano])
     return mediano;
   else if(chiave < elenco[mediano])
            ultimo = mediano-1;
          else
            primo = mediano+1;
 }
 return -1;
}
```

Un breve commento al codice. Il `while` gestisce i passi dell'ARBINi. Il predicato di permanenza nel ciclo **`primo <= ultimo`** indica che il processo iterativo continua se la porzione di array su cui effettuare la ricerca è costituita da almeno un elemento.
La nuova porzione su cui agire viene denotata modificando il valore di `ultimo`, se si dovrà operare sulla semi-porzione di sinistra, mentre invece modificando il valore di `primo`, se si dovrà operare sulla semi-porzione di destra.
Si noti che l'uscita normale dal ciclo `while` significa che la chiave non è stata trovata, e quindi in tal caso si restituisce il valore convenzionale -1.

Questa è l'implementazione in C dell'ARBINi, per una chiave e un array di tipo stringa di `char`:

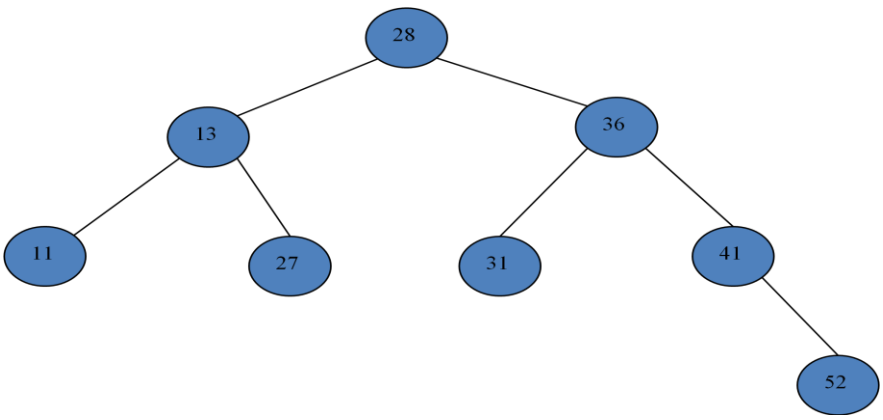
```
int ric_bin_S(char chiave[],char *elenco[],int n)
{
```

```
 int mediano, primo = 0, ultimo = n-1;
 while(primo <= ultimo)
 {
  mediano = (primo+ultimo)/2;
  if(strcmp(chiave,elenco[mediano]) == 0)
     return mediano;
   else if(strcmp(chiave,elenco[mediano]) < 0)
             ultimo = mediano-1;
           else
             primo = mediano+1;
 }
 return -1;
}
```

Si noti che la struttura dell'algoritmo è inalterata e sono cambiate solo le modalità di confronto tra chiave ed elemento dell'array. Si ricorda che la function **`strcmp`** restituisce 0 se le due stringhe sono uguali, restituisce un numero negativo se la prima stringa precede la seconda nell'ordine alfabetico, restituisce un numero positivo se la prima stringa segue la seconda nell'ordine alfabetico.

Analizziamo la complessità dell'ARBINi. La complessità di spazio è n, perché l'algoritmo opera completamente "in place".
Passiamo alla complessità di tempo. Negli algoritmi di ricerca l'operazione dominante è l'operazione di confronto tra la chiave e un elemento dell'array. Nel caso dell'algoritmo di ricerca binaria l'operazione dominante è il **confronto a tre vie** tra la chiave e un elemento dell'array.
L'operazione di confronto a tre vie è l' `if-else-if` che appare nel `while`.
Come per tutti gli algoritmi di ricerca, la complessità di tempo dipende dal valore dei dati oltre che dalla dimensione computazionale e quindi si deve effettuare un'analisi di complessità di caso peggiore.
Il modo più semplice per determinare la complessità di tempo dell'algoritmo di ricerca binaria consiste nell'esaminare il cosiddetto albero binario delle decisioni associato all'esecuzione dell'algoritmo per un dato problema.
I nodi di tale albero sono gli elementi dell'array e il livello in cui appaiono nell'albero corrisponde al passo dell'algoritmo in cui sono confrontati con la chiave. In altre parole, la radice dell'albero è l'elemento che è confrontato con la chiave al primo passo dell'algoritmo, i due nodi del livello 1 sono i due elementi che possono essere confrontati con la chiave al secondo passo, ovvero l'elemento in posizione centrale della semi-porzione di sinistra oppure l'elemento in posizione centrale della semi-porzione di destra, e così via.

11	13	27	28	31	36	41	52
0	1	2	3	4	5	6	7



E’ chiaro che, poiché l’algoritmo a ogni passo effettua la ricerca solo una delle due semi-porzioni del passo precedente, a ogni livello corrisponde una sola operazione di confronto a tre vie.
Quindi si può concludere che il numero totale di confronti a tre vie è uguale alla profondità (cioè il numero di livelli) del corrispondente albero delle decisioni.
Qual è la profondità di un tale albero? L’albero ha un esattamente n nodi, dove n è il size dell’array. Se l’albero fosse completo (cioè ogni nodo diverso da una foglia ha esattamente due nodi figli) la sua profondità sarebbe $\log_2(n+1)$. E’ facile convincersi che l’albero delle decisioni dell’algoritmo di ricerca binaria, in generale, è quasi-completo, nel senso che solo il livello delle foglie può essere incompleto, e quindi la sua profondità è $\lfloor \log_2(n) \rfloor + 1$, dove $\lfloor a \rfloor$, detto il floor di a, indica il più grande intero minore o uguale di a.
In conclusione, la complessità di tempo dell’algoritmo di ricerca binaria è $T(n) = \lfloor \log_2(n) \rfloor + 1$ al più.
Si noti che il caso peggiore è quello in cui la chiave non appartiene all’array, oppure quello in cui la chiave è uguale a una delle foglie del corrispondente albero delle decisioni.