DOMANDA: Illustrare l'Algoritmo di Ricerca Binaria, versione ricorsiva (ARBINric)

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'ARBINric 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 vale 1 (vero) se la chiave appartiene all'array, oppure il valore 0 (falso) se la chiave non appartiene all'array.
L'ARBINric non fa uso di altri array o strutture dati e per questo motivo si dice che opera "in situ" o "in place".
L'ARBINric 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 autoattivazione 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 autoattivazione si crea un processo che risolve una istanza del problema della ricerca in un array ordinato, cioè su una "porzione" dell'array A, e tale processo agisce su una "porzione" di dimensione dimezzata rispetto alla precedente, 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' ARBINric. Alla prima attivazione si crea un processo che agisce sull'intero array A. Alla generica autoattivazione, si crea un processo che effettua la ricerca su una porzione dell'array A. Tale porzione è individuata dall'indirizzo base della porzione e dal size della porzione.
Il caso base della ricorsione è costituito da due distinte situazioni: la prima è che il size della porzione è 0, e in tal caso si deve restituire 0, cioè falso, in quanto la chiave non appartiene alla porzione e quindi non appartiene all'array; la seconda è che l'elemento in posizione "centrale" della porzione (non vuota) che si sta considerando è uguale alla chiave, e in tal caso si deve restituire 1, cioè vero.
Se l'istanza che il processo sta risolvendo non rientra nel caso base, allora si deve confrontare la chiave e l'elemento dell'array che si trova nella posizione "centrale": se la chiave è minore la ricerca proseguirà con l'autoattivazione sulla semi-porzione "di sinistra", cioè la porzione che ha lo stesso indirizzo base della porzione in input e size dimezzato, mentre se la chiave è maggiore la ricerca proseguirà con l'autoattivazione sulla semi-porzione "di destra", cioè la porzione che ha come indirizzo base l'indirizzo della posizione immediatamente a destra della posizione centrale e size dimezzato.
Un esempio aiuta a capire la dinamica dell' ARBINric. Si consideri la chiave 52 e un array A di n=8 elementi interi.
Poiché il size 8 è diverso da 0, non siamo nella prima situazione del caso base. La posizione centrale è (0+7)/2, cioè 3, e poiché la chiave 52 è diversa da 28 non siamo neanche nella seconda situazione del caso base.

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

Quindi, 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, cioè si deve effettuare una autoattivazione che crea un processo che agisce sulla porzione di inizio 4 e size 4.
Il nuovo processo "vede" la seguente porzione di array (si faccia attenzione agli indici locali)

31	36	41	52
0	mediano=1	2	3

Poiché il size 4 è diverso da 0, non siamo nella prima situazione del caso base. La posizione centrale è (0+3)/2, cioè 1, e poiché la chiave 52 è diversa da 36 non siamo neanche nella seconda situazione del caso base.
Quindi, l'algoritmo deve continuare, e poiché 52 è maggiore di 36 la ricerca deve proseguire sulla semi-porzione di destra 2..3 della porzione che si sta considerando, cioè si deve effettuare una autoattivazione che crea un processo che agisce sulla porzione di inizio 2 e size 2.
Il nuovo processo "vede" la seguente porzione di array (si faccia attenzione agli indici locali)

41	52
mediano=0	1

Poiché il size 2 è diverso da 0, non siamo nella prima situazione del caso base. La posizione centrale è (0+1)/2, cioè 0, e poiché la chiave 52 è diversa da 41 non siamo neanche nella seconda situazione del caso base.
Quindi, l'algoritmo deve continuare, e poiché 52 è maggiore di 41 la ricerca deve proseguire sulla semi-porzione di destra 1..1 della porzione che si sta considerando, cioè si deve effettuare una autoattivazione che crea un processo che agisce sulla porzione di inizio 1 e size 1.
Il nuovo processo "vede" la seguente porzione di array (si faccia attenzione agli indici locali)

52
mediano=0

Poiché il size 1 è diverso da 0, non siamo nella prima situazione del caso base. La posizione centrale è (0+0)/2, cioè 0, e poiché la chiave 52 è uguale a 52, siamo nella seconda situazione del caso base e quindi il processo termina restituendo il valore 1, cioè vero.
Si noti che tale valore viene restituito al processo top dello stack dei processi sospesi, e così via, fino al processo iniziale che agiva sull'intero array, restituendo al programma chiamante la soluzione.

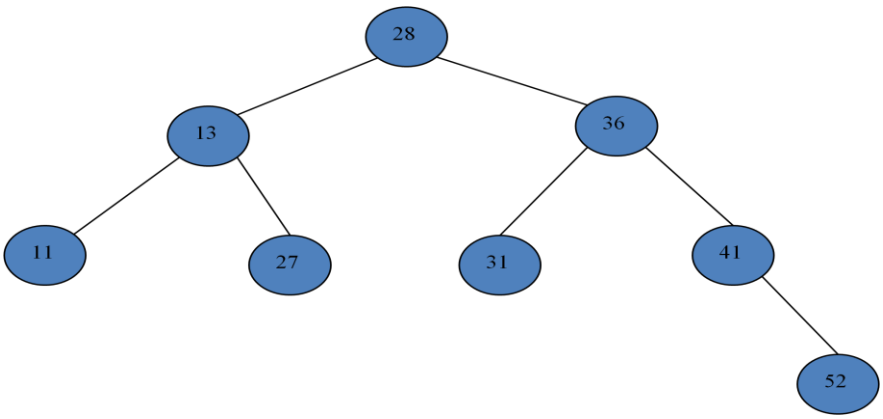
Questa è l'implementazione in C dell'ARBINric, per una chiave e un array di tipo `char`:

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

Un breve commento al codice. La function ha la classica struttura delle function ricorsive, cioè un `if-the-else` che distingue il caso base dalle istanza non banali.
Si noti che il primo `if` si riferisce alla prima situazione del caso base, e il secondo `if` alla seconda situazione del caso base.
Il blocco `else` gestisce l'autoattivazione. La prima autoattivazione agisce sulla semi-porzione di sinistra, che ha lo stesso indirizzo base della porzione ricevuta in input e size uguale al valore dell'indice **`mediano`**; la seconda autoattivazione agisce sulla semi-porzione di destra, che ha come indirizzo base quello della posizione immediatamente a destra della posizione centrale della porzione ricevuta in input e size uguale al valore **`n-mediano-1`**.

Analizziamo la complessità dell'ARBINric. 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 corpo della function.
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 ricorsivo 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 livello dell'autoattivazione in cui sono confrontati con la chiave. In altre parole, la radice dell'albero è l'elemento che è confrontato con la chiave alla prima attivazione dell'algoritmo, i due nodi del livello 1 sono i due elementi che possono essere confrontati con la chiave al secondo livello di autoattivazione (che può essere sulla semi-porzione di sinistra o sulla semi-porzione di destra), 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 autoattivazione genera un processo che effettua la ricerca solo una delle due semi-porzioni della porzione dell'autoattivazione 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 ricorsivo 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.