Università di Napoli Parthenope

Dipartimento di Scienze e Tecnologie

Corsi di Laurea in INFORMATICA

2016, release 1

E-BOOK: Risposte alle domande di esame di Programmazione 1 e Laboratorio

Giulio Giunta

L'e-book è ancora in fase di revisione ed è soggetto a modifiche anche sostanziali. Gli allievi sono invitati a comunicare al docente per email eventuali errori e imprecisioni rilevati nel testo.

L'e-book è stato sviluppato nell'ambito del Progetto di Ateneo "Blended Learning" ed è distribuito con licenza Creative Commons Attribuzione - non commerciale - non opere derivate (vedi http://creativecommons.org/licenses/by-nc-nd/3.0/it)

Sommario

Premessa ... 3
Come rispondere alla domanda: illustrare l'algoritmo di ordinamento per inserimento ... 4
Come rispondere alla domanda: illustrare l'algoritmo di ordinamento per selezione di minimo ... 7
Come rispondere alla domanda: illustrare l'algoritmo di ordinamento per selezione di massimo ... 10
Come rispondere alla domanda: illustrare l'algoritmo di fusione ... 14
Come rispondere alla domanda: illustrare l'algoritmo di string matching ... 18
Come rispondere alla domanda: che cosa è la complessità asintotica ... 21
Come rispondere alla domanda: illustrare l'algoritmo di ricerca binaria, versione iterativa ... 23
Come rispondere alla domanda: illustrare l'algoritmo di ricerca binaria, versione ricorsiva ... 27
Come rispondere alla domanda: illustrare l'algoritmo ricorsivo di somma array, approccio incrementale ... 31
Come rispondere alla domanda: illustrare l'algoritmo ricorsivo di somma array, approccio divide et impera ... 34

Premessa

L'e-book è costituito dalle "risposte alle domande di esame" più frequenti del Corso di Programmazione I e Laboratorio di Programmazione I, tenuto dai prof. Giulio Giunta e Angelo Ciaramella presso il Corso di Laurea in Informatica di UniParthenope.

Le "risposte alle domande di esame" contengono i punti essenziali di una possibile risposta alla corrispondente domanda.

Come tutto il materiale didattico del Corso (video-lezioni, pdf e pps delle slide, quiz online, esercizi, prove di esame, et cetera), le "sintesi" sono anche scaricabili in formato pdf dalla piattaforma di e-learning del DiST (http://e-dist.uniparthenope.it)

Come rispondere alla domanda: illustrare l'algoritmo di ordinamento per inserimento

RISPOSTA:
Lo scopo di un Algoritmo di Ordinamento è ordinare (in senso crescente) un array 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'AOINS ha come dati di input un array e il suo size e ha come dato di output lo stesso array con i dati ordinati. L'AOINS non fa uso di altri array o strutture dati e per questo motivo si dice che opera "in situ" o "in place".

L'AOINS è basato sull'applicazione dell'approccio incrementale al problema dell'ordinamento. Infatti, a ogni passo l'algoritmo risolve una istanza del problema dell'ordinamento, cioè l'ordinamento di una "porzione" dell'array A; all'aumentare del passo aumenta la dimensione dell'istanza del problema, fino ad arrivare alla dimensione n del problema dato.
Diamo una rapida analisi della meccanica dell'AOINS. Al generico passo `i`, essendo già stata ordinata nei passi precedenti la porzione `0..(i-1)` dell'array, l'AOINS ordinerà la porzione `0..i` dell'array A. E' importante osservare che tale ordinamento non è quello definitivo e che nei successivi passi tale ordinamento può essere modificato dall'inserimento di altri tra i rimanenti elementi dell'array.
L'idea di base è che se i primi `i-1` elementi sono ordinati, allora per ordinare i primi `i` elementi basta inserire l'`i`-simo elemento nella sua "giusta" posizione, cioè nella posizione in cui tale elemento avrà alla sua sinistra gli elementi minori o uguali e alla sua destra gli elementi uguali o maggiori.
Quindi, a ogni passo l'AOINS risolve il problema di "inserire" (di qui il nome dell'algoritmo) un elemento dell'array nella porzione già ordinata alla sua sinistra, in modo che la lunghezza della porzione ordinata aumenti di 1.
Un esempio aiuta a capire la dinamica dell'AOINS. Si consideri un array A di n=8 elementi interi

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

Nell'ottica incrementale, il primo elemento dell'array si trova nella posizione corretta: una porzione di lunghezza 1 è sempre ordinata!
Quindi, il primo passo (`i`=1) dell'AOINS consiste nell'ordinare i primi 2 elementi dell'array, cioè la porzione 0..1 dell'array A.

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

Ciò si ottiene considerando l'`i`-simo elemento, cioè l'elemento di indice 1 che è 41, e inserendolo nella "giusta" posizione, in modo che la porzione 0..1 sia ordinata. Poiché 41 è maggiore di 27, la sua posizione corretta è proprio quella in cui già si trova e, in questo caso, nessuno spostamento o scambio deve essere effettuato.
Al secondo passo (`i`=2) si devono ordinare i primi 3 elementi (porzione 0..2) sapendo che la porzione 0..1 è già ordinata. Ciò si ottiene considerando l'`i`-simo elemento, cioè l'elemento di indice 2 che è 36, e inserendolo nella "giusta" posizione, in modo che la porzione 0..2 risulti ordinata.

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

In generale, la “giusta” posizione viene determinata con un procedimento iterativo che procede a ritroso, esaminando successivamente le posizioni (i-1), (i-2),…. ed eventualmente fino alla prima pozione (indice 0). In tale procedimento l’elemento da inserire, 36 nell’esempio, viene confrontato con gli elementi alla sua sinistra. Se l’elemento alla sua sinistra è maggiore allora quell’elemento è spostato (*shiftato*) di una posizione a destra; altrimenti, l’elemento da inserire viene inserito nella posizione a destra dell’elementi che risulta essere minore. Nel nostro caso, si ha che, poiché 41 è maggiore di 36, allora 41 viene spostato a destra; poi, poiché 27 è minore di 36, allora 36 è inserito nella posizione 1, cioè la posizione immediatamente a destra di 27. Al termine del passo, l’array A ha i seguenti valori

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

Si noti che l’istanza 3 del problema, cioè l’ordinamento dei primi 3 elementi dell’array A, è stata risolta.
Al passo successivo, il passo i=3, l’elemento da inserire è 11. Poiché 11 è minore di tutti gli elementi alla sua sinistra, il procedimento a ritroso per l’individuazione della “giusta” posizione di 11 produce la traslazione a destra dei primi 3 elementi e l’inserimento di 11 nella prima posizione dell’array, quella di indice 0. Il nuovo valore dell’array A risulta

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

Si noti che l’istanza 4 del problema, cioè l’ordinamento dei primi 4 elementi dell’array A, è stata risolta.
Il passo successivo, i=4, ha lo scopo di inserire 28 nella giusta posizione in modo che la porzione 0..4 sia ordinata.
E così via, fino all’ultimo passo, i=7, in cui si determina la “giusta” posizione dell’elemento 31 in modo che la porzione 0..7, cioè tutto l’array A, risulti ordinata.

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

La struttura dell’algoritmo è costituita da un ciclo for per i passi e da un ciclo while innestato per il procedimento a ritroso di individuazione della “giusta” posizione.
Questa è l’implementazione in C dell’AOINS, per un array di tipo char:

```
void ord_inser(char array[],int n)
 {
  int i,j;
  char el_da_ins;
  for (i=1;i<n;i++)
  {
     el_da_ins = array[i];
     j = i-1;
     while(j >= 0  &&  el_da_ins < array[j])
         {
           array[j+1] = array[j];
           j--;
         }
     array[j+1] = el_da_ins;
  }
 }
```

Un breve commento al codice. Il `for` su `i` da 1 a n-1 gestisce gli n-1 passi dell'AOINS. Lo scopo del passo `i` è inserire nella giusta posizione l'elemento di indice `i`, che è salvato nella variabile **`el_da_ins`**, in modo che la porzione 0..`i` risulti ordinata.
Il processo iterativo a ritroso per l'individuazione della giusta posizione e il successivo inserimento di **`el_da_ins`** è realizzato mediante il ciclo `while`, il cui predicato di continuazione è **`j >= 0 && el_da_ins < array[j]`**.
Si noti che l'uscita dal ciclo `while` significa che è stata trovata la giusta posizione, quella di indice **`j+1`**, e che gli elementi maggiori di **`el_da_ins`** sono stati spostati a destra di tale posizione. L'ultima operazione del ciclo `for` consiste proprio nell'inserimento di **`el_da_ins`** nella posizione di indice **`j+1`**.

Analizziamo la complessità dell'AOINS. La complessità di spazio è n, perché l'algoritmo opera completamente "in place".
Passiamo alla complessità di tempo. Negli algoritmi di ordinamento si considerano come operazioni dominanti l'operazione di confronto tra due elementi dell'array e l'operazione di scambio tra 2 elementi dell'array, o meglio, nel nostro caso, di spostamento a destra di un elemento dell'array.
L'operazione di confronto tra due elementi dell'array è quella che appare nel predicato del `while`.
Il ciclo `while` viene attivato n-1 volte, poiché il `while` è interno al ciclo `for`. Il numero di iterazioni di ogni attivazione del `while` non è costante, perché dipende sia dalla lunghezza della porzione dove si deve trovare la giusta posizione dell'elemento da inserire, sia dal valore degli elementi di tale porzione. Dobbiamo quindi fare un'analisi di complessità di caso peggiore. Il caso peggiore, cioè il massimo numero possibile di confronti, si ha quando si esce dal ciclo `while` sempre perché `j < 0` , ovvero quando la giusta posizione è sempre la prima posizione (`j = 0`) dell'array. Per inciso, questo accade quando l'array in input è ordinato nel senso decrescente.
Quindi, al primo passo, `i`=1, il massimo numero possibile di confronti nel `while` è 1; al secondo passo, `i`=2, il massimo numero possibile di confronti nel `while` è 2; e così via, fino all'ultimo passo, `i`=n-1, in cui il massimo numero possibile di confronti è n-1.
Per ottenere il numero totale di confronti bisogna sommare i numeri dei confronti di ogni passo, cioè (1 + 2 +…+ (n-2) + (n-1) . Ma tale numero è la somma dei primi (n-1) numeri naturali, che per la formula di Gauss è $\frac{n \cdot (n-1)}{2}$. Si tratta di una complessità di caso peggiore.
Per quanto concerne il numero di scambi, o meglio il numero di shift a destra, basta osservare che si ha uno spostamento a destra ogni volta che è vero il predicato del `while`. Pertanto, nel caso peggiore, il numero di shift a destra è uguale al numero dei confronti.
In conclusione, la complessità di tempo dell'AOINS è

$T(n) = \frac{n \cdot (n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$ confronti, al più

$T(n) = \frac{n \cdot (n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$ scambi (shift), al più.

Infine, è importante notare che il caso migliore si ha quando l'array A in input è già ordinato. In questa situazione l'AOINS effettua solo (n-1) confronti (uno per ogni iterazione del `for`, in quanto il blocco del `while` non viene mai eseguito) e 0 scambi/shift. Gergalmente, si dice che l'AOINS è in grado di "accorgersi" con (n-1) confronti del fatto che un array sia già ordinato.

Come rispondere alla domanda: illustrare l'algoritmo di ordinamento per selezione di minimo

RISPOSTA:
Lo scopo di un Algoritmo di Ordinamento è ordinare (in senso crescente) un array 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'AOSMIN ha come dati di input un array e il suo size e ha come dato di output lo stesso array con i dati ordinati. L'AOSMIN non fa uso di altri array o strutture dati e per questo motivo si dice che opera "in situ" o "in place".

Diamo una rapida analisi della meccanica dell'AOSMIN. L'algoritmo effettua un numero di passi (iterazioni) pari al size dell'array meno 1. Cioè, detto n il size dell'array che chiamiamo A, l'algoritmo effettua n-1 passi per ordinare l'array A.
Ad ogni passo l'algoritmo risolve il sottoproblema della ricerca dell'elemento minimo (e del suo indice) di una opportuna "porzione" dell'array A. Una volta trovato l'elemento minimo e la sua posizione, tale elemento viene scambiato con l'elemento che si trova al primo posto della porzione che si sta considerando.
Al passo successivo, si considera una nuova porzione dell'array, che avrà un numero di elementi diminuito di una unità rispetto al numero di elementi della porzione considerata precedentemente.
Un esempio aiuta a capire la dinamica dell'AOSMIN. Si consideri un array A di n=8 elementi interi

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

Chiamiamo i l'indice che conta i passi. Al primo passo (i=0), l'algoritmo determina il minimo elemento (e la sua posizione) dell'intero array, cioè della porzione che ha come primo elemento l'elemento di indice i (cioè di indice 0) e come ultimo elemento quello di indice n-1 (cioè indice 7). Si noti che tale sottoproblema della determinazione del minimo può essere risolto mediante una opportuna chiamata della function `min_val_ind`.
Nell'esempio, elemento minimo è 11 e il suo indice è 3. Ora, si deve scambiare tale elemento con il primo elemento della porzione che è 27 e che ha indice i=0. Si noti che tale operazione può essere effettuata mediante una opportuna chiamata della function `scambiare`. Dopo tale operazione di scambio l'array A ha i seguenti valori

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

Nel secondo passo si considera la porzione dell'array A di lunghezza n-1 (cioè 7) che va dal secondo elemento dell'array (indice i=1) fino all'ultimo elemento (cioè indice n-1=7)

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

e si determina l'elemento minimo (e il suo indice) di tale porzione di array, cioè la porzione 1..(n-1). Anche questo sottoproblema può essere risolto mediante una opportuna chiamata alla function `min_val_ind`.
Nell'esempio, l'elemento minimo della porzione 1..(n-1) è 13 e il suo indice (in A) è 5. Tale elemento deve essere scambiato con il primo elemento della porzione, che è 41 e ha indice i=1. Dopo tale operazione di scambio il secondo passo è terminato e l'array A ha i seguenti valori

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

Nel terzo passo, si considera la porzione 2..(n-1) dell'array (cioè i=2). Si noti che la porzione 0..1 dell'array è ordinata in modo definitivo e non sarà più modificata.
In generale, il modo migliore per sintetizzare il funzionamento dell'AOSMIN è quello di considerare la situazione al generico passo i. Nell'esempio, consideriamo i=4.

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

La porzione 0..(i-1), cioè 0..3, è ordinata nella sua forma definitiva e non viene più modificata. La porzione i..(n-1), cioè 4..7, è ancora disordinata.
Si determina l'elemento minimo, e il suo indice, della porzione i..(n-1). Tale elemento si scambia con l'elemento di indice i, cioè l'elemento che si trova all'inizio della porzione che si sta considerando. Nel nostro esempio 31 viene scambiato con 36, fornendo

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

La successiva porzione da considerare è la porzione 5..7. A ogni passo la lunghezza della porzione che viene considerata diminuisce di 1.
La struttura dell'algoritmo è costituita da un ciclo `for` per i passi. Questa è l'implementazione in C dell'AOSMIN, per un array di tipo `char`:

```
void ord_sel_min(char array[],int n)
 {
  char min_array;
  for (i=0;i<n-1;i++)
      {
         min_val_ind(&array[i],n-i,&min_array,&indice_min);
         scambiare_c(&array[i],&array[indice_min+i]);
      }
 }
```

Un breve commento al codice. Il for su i da 0 a n-2 gestisce gli n-1 passi dell'AOSMIN. Al passo i, la determinazione dell'elemento minimo (e del suo indice) della porzione i..(n-1)viene effettuata mediante la chiamata alla function `min_val_ind`, con 2 argomenti di input: l'indirizzo iniziale della porzione di array, cioè **`&array[i]`** , e il size della porzione, cioè **`n-i`** , e due argomenti di output, che essendo scalari devono essere passati per indirizzo: **`&min_array, &indice_min`**.
E' importante notare che il valore di `indice_min` restituito dalla function `min_val_ind` è il cosiddetto " valore locale" dell'indice, cioè si riferisce alla porzione che si sta considerando (i..(n-1))e non all'intero array. Per ricostruire il cosiddetto "valore globale" dell'indice, cioè per determinare la corretta posizione all'interno dell'array completo, è necessario effettuare l'operazione detta di "spiazzamento dell'indice locale" o "globalizzazione dell'indice locale", che consiste semplicemente nell'aggiungere al valore dell'indice locale il valore dell'indice di inizio della porzione, cioè i. In altre parole, l'indice globale dell'elemento minimo della porzione è **`indice_min+i`**.
Lo scambio tra tale elemento e l'elemento iniziale della porzione (cioè l'elemento di indice `i`) viene effettuato dalla function `scambiare_c`, che è chiamata con i due argomenti di input/output, necessariamente passati per indirizzo: **`&array[i]`** , **`&array[indice_min+i]`** .

Analizziamo la complessità dell'AOSMIN. La complessità di spazio è n, perché l'algoritmo opera completamente "in place".
Passiamo alla complessità di tempo. Negli algoritmi di ordinamento si considerano come operazioni dominanti l'operazione di confronto tra due elementi dell'array e l'operazione di scambio tra 2 elementi dell'array. Cominciamo, per semplicità, col determinare il numero di operazioni di scambio. In pratica, si tratta di contare il numero di chiamate della function `scambiare_c`. L'attivazione della `scambiare_c` viene fatta a ogni iterazione del ciclo `for`, che va da 0 a (n-2), cioè per un totale di (n-1) iterazioni. Quindi l'AOSMIN effettua sempre (n-1) scambi; è una complessità cosiddetta assoluta, perché indipendente dal valore degli elementi dell'array.
Determiniamo ora il numero delle operazioni di confronto tra due elementi dell'array. Poiché né nel testo della function di ordinamento né nella function `scambiare_c` compaiono operazioni di confronto tra due elementi dell'array, allora i soli confronti sono quelli contenuti nella function `min_val_ind`. Ricordiamo che la complessità di tempo dell'algoritmo di determinazione del minimo di un array è data dal size dell'array -1 operazioni di confronto. Tale complessità è assoluta, cioè indipendente dal valore dei dati. Nel nostro caso la function `min_val_ind` è chiamata (n-1) volte, una per ogni iterazione del ciclo `for`, ma il size della porzione su cui agisce la function non è costante, perché diminuisce di 1 a ogni iterazione.
Alla prima iterazione, la porzione passata alla function `min_val_ind` coincide con l'intero array e ha lunghezza n; quindi la function effettua (n-1) confronti. Alla seconda iterazione, la porzione passata alla function `min_val_ind` ha una lunghezza diminuita di uno rispetto alla precedente, cioè ha lunghezza n-1; quindi la `min_val_ind` effettua n-2 confronti.
E così via, fino all'ultimo passo, in cui alla `min_val_ind` viene passata una porzione di lunghezza 2, per cui il numero di confronti è 1.
Per ottenere il numero totale di confronti bisogna sommare i numeri dei confronti di ogni passo, cioè (n-1) + (n-2) + (n-3) +…+ 2 + 1 . Ma tale numero è la somma dei primi (n-1) numeri naturali, che per la formula di Gauss è $\frac{n \cdot (n-1)}{2}$. Tale complessità è assoluta.
In conclusione, la complessità di tempo dell'AOSMIN è
$T(n) = \frac{n \cdot (n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$ confronti,
$T(n) = n = O(n)$ scambi.
Infine, poiché la complessità di tempo dell'AOSMIN non dipende dal valore dei dati, l'AOSMIN assume lo stesso costo anche nel caso in cui l'array sia già ordinato. Gergalmente, si dice che l'AOSMIN non si "accorge" del fatto che un array sia già ordinato.

Come rispondere alla domanda: illustrare l'algoritmo di ordinamento per selezione di massimo

RISPOSTA:
Lo scopo di un Algoritmo di Ordinamento è ordinare (in senso crescente) un array 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'AOSMAX ha come dati di input un array e il suo size e ha come dato di output lo stesso array con i dati ordinati. L'AOSMAX non fa uso di altri array o strutture dati e per questo motivo si dice che opera "in situ" o "in place".

Diamo una rapida analisi della meccanica dell'AOSMAX. L'algoritmo effettua un numero di passi (iterazioni) pari al size dell'array meno 1. Cioè, detto n il size dell'array che chiamiamo A, l'algoritmo effettua n-1 passi per ordinare l'array A.
Ad ogni passo l'algoritmo risolve il sottoproblema della ricerca dell'elemento massimo (e del suo indice) di una opportuna "porzione" dell'array A. Una volta trovato l'elemento massimo e la sua posizione, tale elemento viene scambiato con l'elemento che si trova all'ultimo posto della porzione che si sta considerando.
Al passo successivo, si considera una nuova porzione dell'array, che avrà un numero di elementi diminuito di una unità rispetto al numero di elementi della porzione considerata precedentemente.
Un esempio aiuta a capire la dinamica dell'AOSMAX. Si consideri un array A di n=8 elementi interi

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

Per comodità, chiamiamo i l'indice che individua il termine della porzione di array che si sta considerando (tale indice decrescerà a ogni passo). Al primo passo (i=n-1=7), l'algoritmo determina il massimo elemento (e la sua posizione) dell'intero array, cioè della porzione che ha come primo elemento l'elemento di indice 0 e come ultimo elemento quello di indice i (cioè indice 7). Si noti che tale sottoproblema della determinazione del massimo può essere risolto mediante una opportuna chiamata della function `max_val_ind`.
Nell'esempio, elemento massimo è 52 e il suo indice è 6. Ora, si deve scambiare tale elemento con l'ulyimo elemento della porzione che è 31 e che ha indice i=7. Si noti che tale operazione può essere effettuata mediante una opportuna chiamata della function `scambiare`. Dopo tale operazione di scambio l'array A ha i seguenti valori

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

Nel secondo passo si considera la porzione dell'array A di lunghezza n-1 (cioè 7) che va dal primo elemento dell'array (indice 0) fino al penultimo elemento (cioè indice i=6)

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

e si determina l'elemento massimo (e il suo indice) di tale porzione di array, cioè la porzione 0..6. Anche questo sottoproblema può essere risolto mediante una opportuna chiamata alla function `max_val_ind`.

Nell'esempio, l'elemento massimo della porzione 0..6 è 41 e il suo indice (in A) è 1. Tale elemento deve essere scambiato con l'ultimo elemento della porzione, che è 31 e ha indice i=6. Dopo tale operazione di scambio il secondo passo è terminato e l'array A ha i seguenti valori

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

Nel terzo passo, si considera la porzione 0..5 dell'array (cioè i=5). Si noti che la porzione 6..7 dell'array è ordinata in modo definitivo e non sarà più modificata.
In generale, il modo migliore per sintetizzare il funzionamento dell'AOSMAX è quello di considerare la situazione al generico passo i. Nell'esempio, consideriamo i=4.

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

La porzione (i+1)..(n-1), cioè 5..7, è ordinata nella sua forma definitiva e non viene più modificata. La porzione 0..i, cioè 0..4, è ancora disordinata.
Si determina l'elemento massimo, e il suo indice, della porzione 0..4. Tale elemento si scambia con l'elemento di indice i, cioè l'elemento che si trova all'ultimo posto della porzione che si sta considerando. Nel nostro esempio 31 viene scambiato con 28, fornendo

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

La successiva porzione da considerare è la porzione 0..3. A ogni passo la lunghezza della porzione che viene considerata diminuisce di 1.
La struttura dell'algoritmo è costituita da un ciclo `for` per i passi. Questa è l'implementazione in C dell'AOSMAX, per un array di tipo `char`:

```
void ord_sel_max(char array[],int n)
{
  int i,indice_max;
  char max_array;
  for (i=n-1;i>0;i--)
      {
       max_val_ind(&array[0],i+1,&max_array,&indice_max);
       scambiare_c(&array[i],&array[indice_max]);
      }
 }
```

Un breve commento al codice. Il for su i da n-1 a 1 gestisce gli n-1 passi dell'AOSMAX. Al passo i, la determinazione dell'elemento massimo (e del suo indice) della porzione 0..i viene effettuata mediante la chiamata alla function `max_val_ind`, con 2 argomenti di input: l'indirizzo iniziale della porzione di array, cioè **`&array[0]`**, e il size della porzione, cioè **`i+1`** , e due argomenti di output, che essendo scalari devono essere passati per indirizzo: **`&max_array, &indice_max`**.
Si noti che il valore di `indice_max` restituito dalla function `max_val_ind` è il cosiddetto " valore locale" dell'indice, cioè si riferisce alla porzione che si sta considerando (0..i) ma coincide con il cosiddetto "valore globale" dell'indice, cioè quello che determina la corretta posizione all'interno dell'array completo, e quindi non si deve effettuare alcuna operazione di "spiazzamento dell'indice locale", che invece è necessaria nel caso dell'AOSMIN.

Lo scambio tra tale elemento e l'elemento finale della porzione (cioè l'elemento di indice `i`) viene effettuato dalla function `scambiare_c`, che è chiamata con i due argomenti di input/output, necessariamente passati per indirizzo: **`&array[i]`** , **`&array[indice_max]`** .
Se si avesse a disposizione una function che determina solo l'indice dell'elemento massimo di un array, chiamiamola **`max_ind`**, allora la function **`ord_sel_max`** assumerebbe la seguente forma:

```
void ord_sel_max(char array[],int n)
 {
  int i;
  for (i=n-1;  i>0; i--)
    scambiare_c(&array[i],&array[max_ind(&array[0],i+1)]);
 }
```

Con

```
int max_ind(char a[],int n)
 {
  int i,i_max;   i_max = 0;
  for (i=1;  i<n; i++)
       if (a[i] > a[i_max])
            i_max = i;
  return i_max;
 }
```

Analizziamo la complessità dell'AOSMAX. La complessità di spazio è n, perché l'algoritmo opera completamente "in place".
Passiamo alla complessità di tempo. Negli algoritmi di ordinamento si considerano come operazioni dominanti l'operazione di confronto tra due elementi dell'array e l'operazione di scambio tra 2 elementi dell'array. Cominciamo, per semplicità, col determinare il numero di operazioni di scambio. In pratica, si tratta di contare il numero di chiamate della function `scambiare_c`. L'attivazione della `scambiare_c` viene fatta a ogni iterazione del ciclo `for`, che va da (n-1) a 1, cioè per un totale di (n-1) iterazioni. Quindi l'AOSMAX effettua sempre (n-1) scambi; è una complessità cosiddetta assoluta, perché indipendente dal valore degli elementi dell'array.
Determiniamo ora il numero delle operazioni di confronto tra due elementi dell'array. Poiché né nel testo della function di ordinamento né nella function `scambiare_c` compaiono operazioni di confronto tra due elementi dell'array, allora i soli confronti sono quelli contenuti nella function `max_val_ind` . Ricordiamo che la complessità di tempo dell'algoritmo di determinazione del massimo di un array è data dal size dell'array -1 operazioni di confronto. Tale complessità è assoluta, cioè indipendente dal valore dei dati. Nel nostro caso la function `max_val_ind` è chiamata (n-1) volte, una per ogni iterazione del ciclo `for`, ma il size della porzione su cui agisce la function non è costante, perché diminuisce di 1 a ogni iterazione.
Alla prima iterazione, la porzione passata alla function `max_val_ind` coincide con l'intero array e ha lunghezza n; quindi la function effettua (n-1) confronti. Alla seconda iterazione, la porzione passata alla function `max_val_ind` ha una lunghezza diminuita di uno rispetto alla precedente, cioè ha lunghezza n-1; quindi la `max_val_ind` effettua n-2 confronti.
E così via, fino all'ultimo passo, in cui alla `max_val_ind` viene passata una porzione di lunghezza 2, per cui il numero di confronti è 1.
Per ottenere il numero totale di confronti bisogna sommare i numeri dei confronti di ogni passo, cioè (n-1) + (n-2) + (n-3) +…+ 2 + 1 . Ma tale numero è la somma dei primi (n-1) numeri naturali, che per la formula di Gauss è $\frac{n \cdot (n-1)}{2}$. Tale complessità è assoluta.
In conclusione, la complessità di tempo dell'AOSMAX è

$T(n) = \frac{n \cdot (n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$ confronti,

$T(n) = n = O(n)$ scambi.

Infine, poiché la complessità di tempo dell'AOSMAX non dipende dal valore dei dati, l'AOSMAX assume lo stesso costo anche nel caso in cui l'array sia già ordinato. Gergalmente, si dice che l'AOSMAX non si "accorge" del fatto che un array sia già ordinato.

Come rispondere alla domanda: illustrare l'algoritmo di fusione

RISPOSTA:
L'Algoritmo di Fusione risolve il cosiddetto problema del "merging", ovvero la fusione di due array ORDINATI di dati (dello stesso tipo) in un terzo array ORDINATO che ha come elementi gli elementi dei due array, e quindi ha un size uguale alla somma dei size dei due array. I dati possono essere numeri, caratteri, stringhe di caratteri o qualsiasi altro insieme di dati su cui sia definita una relazione di ordine.

L'AFUS ha come dati di input due array ordinati, che chiamiamo `A` e `B`, e i loro size che chiamiamo rispettivamente `n_A` e `n_B`, e ha come dato di output il terzo array che chiamiamo `C`, il cui size è noto ed è `n_A+n_B`.

L'AFUS è basato sull'idea di effettuare un numero di passi uguale al size del terzo array `C`, cioè `n_A+n_B` passi; a ogni passo, si determina un elemento dell'array `C`.
Detto `i_C` l'indice per accedere agli elementi dell'array `C`, al generico passo `i_C` lo scopo è determinare l'elemento di `C` di indice `i_C`.
Come si costruisce l'elemento `C(i_C)` ?
Per prima cosa, chiamiamo `i_A` e `i_B` i due indici per accedere rispettivamente agli elementi dell'array `A` e agli elementi dell'array `B`. Al primo passo `i_A`, `i_B` e `i_C` indicano il primo elemento (dei rispettivi array).
Al passo `i_C`, ci sono tre possibili scenari:
Il primo è quello per così dire "normale", cioè c'è almeno un elemento di `A` e un elemento di `B` da considerare, ovvero non ancora inseriti nell'array `C`; gli indici `i_A` e `i_B` indicano l'elemento di `A` e l'elemento di `B` che devono essere considerati al fine di determinare quale deve essere inserito in C; è facile convincersi che il più piccolo tra `A(i_A)` e `B(i_B)` deve essere inserito in `C`, nella posizione `i_C` (nel caso gli elementi `A(i_A)` e `B(i_B)` fossero uguali, si inserirebbe uno qualunque dei due); L'indice dell'array il cui elemento viene inserito in `C` deve essere incrementato.
Il secondo scenario è quello in cui tutti gli elementi dell'array `A` sono stati già inseriti in `C` e rimangono da considerare solo gli elementi dell'array `B`; in tal caso, l'elemento `B(i_B)` deve essere inserito in `C`, nella posizione `i_C`, e l'indice `i_B` deve essere incrementato.
Il terzo scenario è quello in cui tutti gli elementi dell'array `B` sono stati già inseriti in `C` e rimangono da considerare solo gli elementi dell'array `A`; in tal caso, l'elemento `A(i_A)` deve essere inserito in `C`, nella posizione `i_C`, e l'indice `i_A` deve essere incrementato.
Si noti che, per una data istanza del problema, solo uno tra lo scenario 2 e lo scenario 3 può verificarsi.

Un esempio aiuta a capire la dinamica dell'AFUS. Si consideri un array `A` ordinato di `n_A`=7 elementi interi e un array `B` ordinato di `n_B`=4 elementi interi

27	36	41	48	52	66	68
i_A=0	1	2	3	4	5	6

31	33	49	50
i_B=0	1	2	3

L'array `C` avrà 11 (cioè 7+4) elementi

0	0	0	0	0	0	0	0	0	0	0
i_C=0	1	2	3	4	5	6	7	8	9	10

Poiché 27 è minore di 31, allora 27 è assegnato a `C(0)` e gli indici `i_A` e `i_C` incrementati

27	0	0	0	0	0	0	0	0	0	0
0	**i _C=1**	2	3	4	5	6	7	8	9	10

27	36	41	48	52	66	68
0	**i _A=1**	2	3	4	5	6

31	33	49	50
i _B=0	1	2	3

Al secondo passo, poiché 31 è minore di 36 allora 31 è assegnato a `C(1)` e gli indici `i_B` e `i_C` incrementati

27	31	0	0	0	0	0	0	0	0	0
0	1	**i _C=2**	3	4	5	6	7	8	9	10

27	36	41	48	52	66	68
0	**i _A=1**	2	3	4	5	6

31	33	49	50
0	**i _B=1**	2	3

Al terzo passo, poiché 33 è minore di 36, allora 33 è assegnato a `C(2)` e gli indici `i_B` e `i_C` incrementati

27	31	33	0	0	0	0	0	0	0	0
0	1	2	**i _C=3**	4	5	6	7	8	9	10

27	36	41	48	52	66	68
0	**i _A=1**	2	3	4	5	6

31	33	49	50
0	1	**i _B=2**	3

E così via, fino a quando tutti gli elementi di uno dei due array sono stati inseriti in `C` (nell'esempio, l'array `B`)

27	31	33	36	48	49	50	0	0	0	0
0	1	2	3	4	5	6	**i _C=7**	8	9	10

27	36	41	48	52	66	68
0	1	2	3	**i _A=4**	5	6

31	33	49	50	
0	1	2	3	**i _B=4**

A questo punto non è più necessario effettuare alcun confronto tra elementi di A e di B, ma si deve solo assegnare 52 a C(1) e incrementare gli indici i_A e i_C.
L'algoritmo termina quando viene inserito l'elemento nella posizione 10 dell'array C.

Questa è l'implementazione in C dell'AFUS, per due array di tipo char:

```
void fusioneC(char a[],int n_a,char b[],int n_b,char c[])
{
  int i_a=0,i_b=0,i_c;
  for (i_c=0; i_c < n_a+n_b; i_c++)
    {
     if(i_a<n_a && i_b<n_b)
      {
       if(a[i_a] < b[i_b]){
         c[i_c] = a[i_a];
         ia++;}
       else {
         c[i_c] = b[i_b];
         ib++;}
      }
      else if(i_b >= n_b) {
         c[i_c] = a[i_a];
         ia++;}
       else {
         c[i_c] = b[i_b];
         ib++;}
    }
}
```

Un breve commento al codice. Il for su i_c da 0 a n_a+n_b-1 gestisce i passi dell'AFUS. Lo scopo del passo i_c è inserire nella posizione i_c dell'array C uno tra i due elementi A(i_a) o B(i_b). La selezione **if(i_a<n_a && i_b<n_b)** individua il primo scenario, mentre la selezione **if(i_b >= n_b) .. else ..** individua gli altri due scenari.

Una implementazione più elegante dell'AFUS è la seguente, in cui ognuno dei tre scenari è realizzato da un ciclo while

```
void fusioneC(char a[],int n_a,char b[],int n_b,char c[])
{
  int i_a=0,i_b=0,i_c=0;
  while (i_a < n_a && i_b <n_b)
    {
     if(a[i_a] < b[i_b])
         c[i_c++] = a[i_a++];
       else
         c[i_c++] = b[i_b++];
     }
  while (i_a < n_a)
     c[i_c++] = a[i_a++];
  while (i_b < n_b)
     c[i_c++] = b[i_b++];
}
```

Analizziamo la complessità dell'AFUS. Chiamiamo n la dimensione computazionale (n = n_A+n_B) del problema. La complessità di spazio è 2 n, perché l'algoritmo costruisce un terzo array di size n.
Passiamo alla complessità di tempo. L'operazione dominante è l'operazione di confronto tra un elemento dell'array `A` e un elemento dell'array `B`.
L'AFUS effettua n = n_A+n_B passi, cioè quanti sono gli elementi di `C` che devono essere determinati. A ogni passo si costruisce un elemento dell'array `C`.
Se si è nello scenario 1, allora è necessario esattamente un confronto per determinare un elemento di `C`.
Se si è nello scenario 2 o nello scenario 3, allora non si effettua alcun confronto.
Quindi si può concludere che il numero totale di confronti è al più n = n_A+n_B.
Si noti che si tratta di una complessità di caso peggiore.
In conclusione la complessità di tempo dell'AFUS è
$T(n) = T(n_A, + n_B) = n = O(n)$ confronti, al più; con $n = n_A + n_B$.

Come rispondere alla domanda: illustrare l'algoritmo di string matching

RISPOSTA:
L'Algoritmo di String Matching risolve il cosiddetto problema del "matching", ovvero l'individuazione di una certa stringa di caratteri (detta "stringa chiave") all'interno di un'altra data stringa di caratteri (detta "stringa testo"). Da un punto di vista astratto, il problema è assimilabile a un problema di ricerca ("searching"). Ci sono diverse varianti del problema del matching, che differiscono per l'output che viene richiesto: per esempio, l'indice di inizio della prima occorrenza, il numero delle occorrenze, la sostituzione dell'occorrenza della stringa chiave nella stringa testo con un'altra stringa assegnata (il classico "trova e sostituisci" dei text editor), etc.

L'ASM ha come dati di input due stringhe, cioè la "stringa chiave" e la "stringa testo", e ha come dato di output il numero delle volte (detto numero delle occorrenze) in cui la stringa chiave compare come sottostringa all'interno della stringa testo.

L'ASM è basato sull'idea di effettuare un numero di passi uguale alla lunghezza della stringa testo (più precisamente, alla lunghezza della stringa testo meno la lunghezza della stringa chiave +1, per evitare inutili confronti nelle iterazioni finali), risolvendo a ogni passo il problema di determinare l'uguaglianza tra la stringa chiave e la sottostringa della stringa testo che ha come posizione inziale il passo e come lunghezza la lunghezza della stringa chiave.

Un esempio aiuta a capire la dinamica dell'ASM. Si consideri una stringa `testo` di `m`=7 caratteri e una stringa `chiave` di `n`=2 caratteri

testo	G	T	G	A	T	G	T
	i =0	1	2	3	4	5	6
chiave	T	G					
	0	1					

Al primo passo (`i`=0), poiché la sottostringa di inizio 0 e lunghezza 2 di `testo` è diversa dalla stringa `chiave`, allora non c'è alcuna occorrenza e bisogna continuare il processo iterativo

testo	G	T	G	A	T	G	T
	0	i =1	2	3	4	5	6
chiave		T	G				
		0	1				

Al secondo passo (`i`=1), poiché la sottostringa di inizio 1 e lunghezza 2 di `testo` è uguale alla stringa `chiave`, allora bisogna incrementare il contatore delle occorrenze e poi continuare il processo iterativo

testo	G	T	G	A	T	G	T
	0	1	i =2	3	4	5	6
chiave			T	G			
			0	1			

Al terzo passo (`i`=2), poiché la sottostringa di inizio 2 e lunghezza 2 di `testo` è diversa dalla stringa `chiave`, allora non c'è alcuna occorrenza e bisogna continuare il processo iterativo

testo	G	T	G	A	T	G	T
	0	1	2	i =3	4	5	6
chiave				T	G		
				0	1		

Al quarto passo (i=3), poiché la sottostringa di inizio 3 e lunghezza 2 di testo è diversa dalla stringa chiave, allora non c'è alcuna occorrenza e bisogna continuare il processo iterativo

testo	G	T	G	A	T	G	T
	0	1	2	3	i =4	5	6
chiave					T	G	
					0	1	

Al quinto passo (i=4), poiché la sottostringa di inizio 4 e lunghezza 2 di testo è diversa dalla stringa chiave, allora non c'è alcuna occorrenza e bisogna continuare il processo iterativo

testo	G	T	G	A	T	G	T
	0	1	2	3	4	i =5	6
chiave						T	G
						0	1

Al sesto passo (i=5), poiché la sottostringa di inizio 5 e lunghezza 2 di testo è uguale alla stringa chiave, allora bisogna incrementare il contatore delle occorrenze. L'algoritmo termina perché è stata eseguita l'ultima iterazione (la (m-n+1)-sima, cioè la sesta iterazione, con l'indice di ciclo i che ha raggiunto il valore 5, cioè 7-2).

Questa è l'implementazione in C dell'ASM:

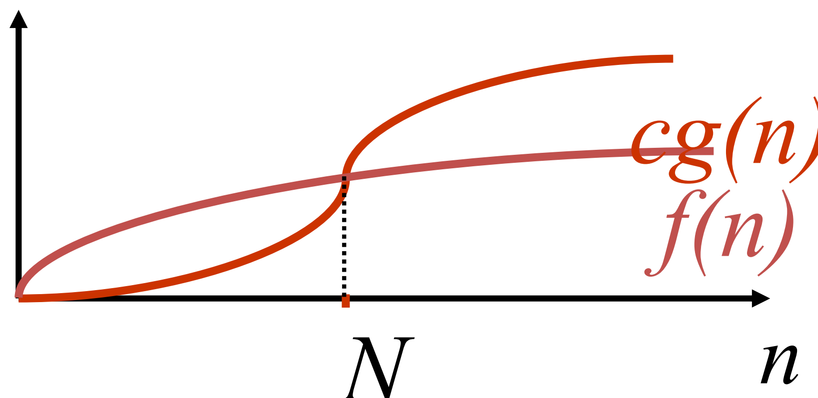
```
int string_matching(char chiave[],char testo [])
{
  int n, m, i, conta_chiave;
  n = strlen(chiave);
  m = strlen(testo);
  conta_chiave = 0;
  for (i=0; i <= m-n; i++)
     if(strncmp(chiave,&testo[i],n) == 0)
        conta_chiave++;
  return conta_chiave;
}
```

Un breve commento al codice. Le lunghezze delle due stringhe in input sono determinate dalla function **strlen** della libreria string. Il for su i da 0 a m-n gestisce i passi dell'ASM. Lo scopo del passo i è risolvere il problema di determinare l'uguaglianza tra la (sotto)stringa della stringa chiave di lunghezza n e la sottostringa della stringa testo che ha come posizione inziale i e come lunghezza n. Si noti che poiché è coinvolta una sottostringa della stringa testo, si deve usare la function della libreria string che effettua il confronto tra due sottostringhe, ovvero la **strncmp** (non si può usare la function **strcmp**, che invece confronta due stringhe).
Si ricorda che la function **strncmp** ha tre parametri di input: la prima sottostringa (più precisamente, il puntatore all'indirizzo iniziale della prima sottostringa), che è proprio l'indirizzo base della stringa chiave; la seconda sottostringa (più precisamente, il puntatore all'indirizzo iniziale della seconda

sottostringa), che è l'indirizzo dell'elemento in posizione `i` della stringa `testo`; la lunghezza delle due sottostringhe da confrontare, che nel nostro caso è `n`. La function **`strncmp`** ha un unico parametro di output: un numero intero, che vale 0 se le due sottostringhe di input sono uguali, vale un numero negativo se la prima sottostringa precede la seconda nell'ordine alfabetico, mentre vale un numero positivo se la prima sottostringa segue la seconda nell'ordine alfabetico.

Analizziamo la complessità dell'ASM. La dimensione computazionale è individuata dalle due quantità m e n, rispettivamente la lunghezza della stringa testo e la lunghezza della stringa chiave. La complessità di spazio è $m+n$, perché l'algoritmo lavora "in place".
Passiamo alla complessità di tempo. L'operazione dominante è l'operazione di confronto tra un elemento della stringa testo e un elemento della stringa chiave. Tali confronti sono eseguiti all'interno della function **`strncmp`**. Poiché la **`strncmp`** determina l'uguaglianza di due sottostringhe, cioè di due array, il numero di confronti per ogni chiamata della function **`strncmp`** è al più la lunghezza n delle sottostringhe in input. La function **`strncmp`** è chiamata una sola volta per ogni passo del ciclo `for`, che effettua $(m\text{-}n+1)$ passi. Quindi si può concludere che il numero totale di confronti è al più $(m\text{-}n+1)\ n$.
Si noti che si tratta di una complessità di caso peggiore.
In conclusione, la complessità di tempo dell'ASM è
$T(m,n) = (m-n+1)\cdot n = O(m\cdot n)$ confronti, al più.

RISPOSTA:
La complessità di un algoritmo è una misura astratta della quantità di risorse computazionali necessarie per la sua esecuzione.
In particolare, si parla di complessità di spazio e di complessità di tempo, che si riferiscono alle due principali risorse computazionali: lo spazio di memoria e il tempo dell'esecutore.
Formalmente, la complessità di spazio e la complessità di tempo, indicate rispettivamente con S(n) e con T(n), sono funzioni della dimensione computazionale n del problema da risolvere.
La dimensione computazionale di un problema è una misura astratta del numero dei dati, o in generale di informazioni, che definiscono il problema, o meglio una particolare *istanza* del problema.
S(n) è il size di tutte le strutture dati (di input, locali e di output) che devono essere utilizzate dall'algoritmo durante la sua esecuzione per risolvere una istanza di dimensione n del problema.
T(n) è il numero di operazioni dominanti (di input, locali e di output) che devono essere eseguite dall'algoritmo per risolvere una istanza di dimensione n del problema.
Il termine complessità asintotica si riferisce al comportamento asintotico, cioè al tendere di n all'infinito, delle funzioni S(n) e T(n). Si noti che S(n) e T(n), escludendo il caso di algoritmi che implementano una formula chiusa e la cui complessità è quindi costante, sono funzioni che tendono a +infinito per n che tende a +infinito.
La notazione asintotica delle funzioni di complessità ha lo scopo di evidenziare in modo sintetico il loro comportamento asintotico, indicando solo i termini che crescono più velocemente rispetto agli altri eventuali termini che costituiscono l'espressione delle funzioni S(n) e T(n).
La notazione asintotica dell' "O" (che si legge "O grande") è formalmente definita nel seguente modo.
Siano f(n) e g(n) due funzioni definite (dominio) sui numeri naturali e a valori (codominio) sui numeri naturali; f(n) e g(n) sono funzioni non decrescenti, cioè sono costanti oppure sono crescenti:
allora f(n) = O(g(n)) significa che esistono due costanti (cioè non dipendenti da n) c e N_0, non negative, tali che f(n) <= c g(n) per n >= N_0.
In altre parole, con la notazione f(n) = O(g(n)) intendiamo affermare che la funzione f(n) tende all'infinito con la stessa rapidità o con rapidità inferiore rispetto a una scalatura opportuna della funzione g(n). Il termine scalatura di una funzione indica semplicemente il prodotto della funzione per un numero non negativo.
L'interpretazione grafica di tale definizione è immediata: il grafico della funzione f(n) non è al di sopra del grafico della funzione scalata c g(n) per tutti i valori di n a partire da N_0 in poi.



Vediamo alcuni esempi riferiti alla funzione complessità di tempo.
$T(n) = 1453 \quad \Rightarrow \quad T(n) = O(1)$ cioè la complessità di tempo è costante

$T(n) = 10n + 1000 \Rightarrow T(n) = O(n)$ cioè la complessità di tempo è lineare
$T(n) = 100000\,(n+1) \Rightarrow T(n) = O(n)$ cioè la complessità di tempo è lineare
$T(n) = 30n^2 + 100n + 233 \Rightarrow T(n) = O(n^2)$ cioè la complessità di tempo è quadratica
$T(n) = \frac{n \cdot (n-1)}{2} = O(n^2)$

In generale, se T(n) è un polinomio di grado k, allora $T(n)=O(n^k)$
$T(n) = \frac{1}{2}n + 1000 \log(n) \Rightarrow T(n) = O(n)$ cioè la complessità di tempo è lineare
$T(n) = 1.5^n + n^{1000} \Rightarrow T(n) = O(1.5^n)$ cioè la complessità di tempo è esponenziale
$T(n) = 1.5^n + n^{1000} \Rightarrow T(n) = O(1.5^n) = O(2^n) = O(10^n)$ cioè la complessità di tempo è esponenziale.
.

Come rispondere alla domanda: illustrare l'algoritmo di ricerca binaria, versione iterativa

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.

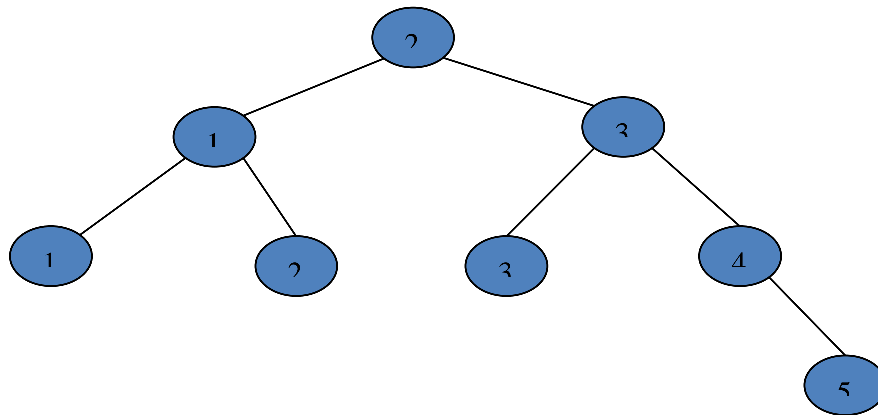
Questa è l'implementazione in C dell'ARBINi, per una chiave e un array di tipo stringa di `char`:

```
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.

Come

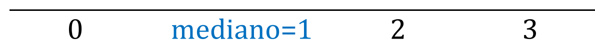
rispondere alla domanda: illustrare l'algoritmo di ricerca binaria, versione ricorsiva

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.

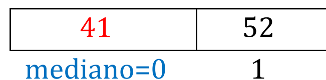
11	13	27	28	31	36	41	52
0	1	2	mediano=3	4	5	6	7

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



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)



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)

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`:

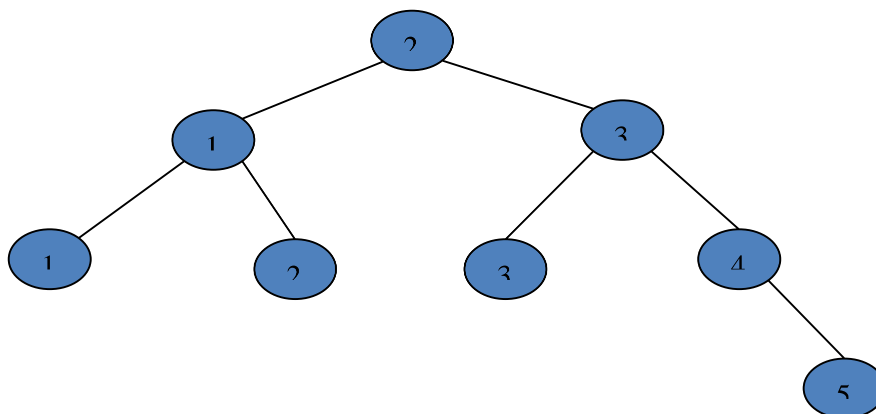
```
int ric_bin_ricTF(char chiave,char elenco[],int n)
{
 int mediano;
 if(n == 0)
    return 0;
 mediano = (n-1)/2;
 if(chiave == elenco[mediano])
   return 1;
  else if(chiave < elenco[mediano])
          return ric_bin_ricTF(chiave,elenco,mediano);
         else
          return ric_bin_ricTF(chiave,elenco+mediano+1,n-mediano-1);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

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.

Come rispondere alla domanda: illustrare l'algoritmo ricorsivo di somma array, approccio incrementale

RISPOSTA:
L'algoritmo ricorsivo di somma di un array, approccio incrementale, calcola la somma degli elementi di un array di dati. I dati possono essere numeri interi, numeri reali o qualsiasi altro insieme di dati su cui sia definita una operazione di somma.
L'ASOMRICAI ha come dati di input un array e il suo size, e ha come dato di output un dato scalare, che è il valore della somma di tutti gli elementi dell'array.
L'ASOMRICAI non fa uso di altri array o strutture dati.
L'ASOMRICAI è basato sull'applicazione del classico approccio incrementale al problema della somma degli elementi di un array.
L'approccio incrementale è una metodologia di progetto di algoritmi in cui la soluzione del problema viene determinata risolvendo iterativamente, a partire dall'istanza banale, istanze di dimensione crescente (incrementata ogni volta di 1) del problema, fino ad arrivare all'istanza in input. A ogni iterazione, la soluzione dell'istanza è calcolata utilizzando la soluzione (già calcolata) dell'istanza precedente.
La tecnica di programmazione ricorsiva, basata sull'autoattivazione di una function, consente di descrivere facilmente un algoritmo basato sull'approccio incrementale, usando la cosiddetta "tail recursion".
Nel caso del problema della somma degli elementi di un array, la soluzione del problema, cioè la somma, viene calcolata come la somma tra l'ultimo elemento dell'array e la somma degli elementi della porzione dell'array che va dal primo al penultimo elemento.
Si noti che si tratta di una sorta di "variante" rispetto all'approccio Divide et impera: qui, la porzione "di sinistra" invece di avere size dimezzato (come nel DI) ha size diminuito di 1 e la porzione "di destra" invece di avere size dimezzato (come nel DI) è costituita da un unico elemento (in particolare, l'ultimo elemento). Di qui il nome "tail recursion".
Ogni autoattivazione suddivide una data istanza in due istanze: una di size 1, la cui soluzione è proprio l'unico elemento che la costituisce, e una di dimensione diminuita di 1; crea un processo che risolve l'istanza del problema della somma sulla porzione di size diminuito di 1; e quindi restituisce la somma tra il valore dell'ultimo elemento della porzione e il valore ottenuto da tale processo. A ogni autoattivazione, il processo creato agisce su una "porzione" di size diminuito di 1 rispetto al precedente, fino ad arrivare al size 1, che è il size dell'istanza banale del problema della somma.
Diamo una rapida analisi della meccanica dell' ASOMRICAI. Alla prima attivazione si crea un processo che agisce sull'intero array A, di size n; esso crea a sua volta un processo che effettua la somma sulla porzione dell'array A costituita dai primi (n-1) elementi; il valore calcolato da questo processo è sommato all'ultimo elemento e il risultato è restituito. La porzione è individuata dall'indirizzo base e dal size (n-1).
Il caso base della ricorsione è costituito dalla porzione di size 1, e in tal caso si deve restituire il valore dell'unico elemento della porzione.
Se l'istanza che il processo sta risolvendo non rientra nel caso base, allora si deve autoattivare sulla porzione 0..(n-2), cioè la porzione che ha lo stesso indirizzo base della porzione in input e size diminuito di 1, e sommare il risultato di tale autoattivazione con l'elemento in ultima posizione della porzione in input, restituendo tale somma al chiamante.
Un esempio aiuta a capire la dinamica dell'ASOMRICAI. Si consideri un array A di n=8 elementi interi.
Poiché il size 8 è diverso da 1, non siamo nel caso base.

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

Quindi, l'algoritmo deve continuare, cioè deve effettuare una autoattivazione che crea un processo che agisce sulla porzione di inizio 0 e size 7 (cioè la porzione 0..6); il valore restituito da tale autoattivazione deve essere sommato al valore dell'elemento in ultima posizione, cioè il valore 13, che è il valore dell'elemento in posizione 7.
Il nuovo processo "vede" la seguente porzione di array
Processo 1

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

Poiché il size 7 è diverso da 1, non siamo nella prima situazione del caso base. Si attiva un nuovo processo che agisce sulla porzione di inizio 0 e size 6 (cioè la porzione 0..5); il valore restituito da tale autoattivazione deve essere sommato al valore dell'elemento in ultima posizione, cioè il valore 11, che è il valore dell'elemento in posizione 6.
Il nuovo processo "vede" la seguente porzione di array
Processo 2

41	31	27	28	52	36
0	1	2	3	4	5

Poiché il size 6 è diverso da 1, non siamo nella prima situazione del caso base. Il processo attiva un nuovo processo che agisce sulla porzione di inizio 0 e size 5 (cioè la porzione 0..4); il valore restituito da tale autoattivazione deve essere sommato al valore dell'elemento in ultima posizione, cioè il valore 36, che è il valore dell'elemento in posizione 5.
Il nuovo processo "vede" la seguente porzione di array
Processo 3

41	31	27	28	52
0	1	2	3	4

Poiché il size è 5, non siamo nella prima situazione del caso base. Si crea un nuovo processo e così via, fino al raggiungimento del caso base.
A questo punto ognuno dei processi sospesi 7,6,5,4,3,2,1 somma il valore ricevuto con elemento in ultima posizione della porzione di loro competenza e restituisce tale somma; al termine, il processo sospeso iniziale ottiene il valore dal processo 1, effettua la somma con l'ultimo elemento dell'array A e restituisce la soluzione del problema.

Questa è l'implementazione in C dell'ASOMRICAI, per un array di tipo `int`:

```
int somma_a_ricAI(int a[],int n)
{
 if (n == 1)
     /* soluzione del caso base */
       return a[0];
    else
     /* autoattivazione */
       return a[n-1] + somma_a_ricAI(a,n-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 istanze non banali.
Il blocco `then` gestisce il caso base, restituendo il valore dell'unico elemento di una porzione di size 1.

Il blocco `else` gestisce l'autoattivazione. L'autoattivazione agisce sulla porzione che ha lo stesso indirizzo base della porzione ricevuta e size uguale a **n-1**, cioè un size diminuito di 1 rispetto al size **n** della porzione ricevuta. La soluzione dell'istanza è calcolata sommando il valore dell'ultimo elemento della porzione ricevuta con il valore restituito dall'autoattivazione, ed è restituita al chiamante.
L'esecuzione dell'ASOMRICAI dà luogo alla seguente sequenza di attivazione/sospensione dei processi nello stack (ogni colonna "fotografa" lo stack dopo ogni autoattivazione):

							P(0)			
						P(0..1)	P(0..1)	P(0..1)		
					P(0..2)	P(0..2)	P(0..2)	P(0..2)	P(0..2)	
				P(0..3)	P(0..3)	P(0..3)	P(0..3)	P(0..3)	P(0..3)	P(0..3)
			P(0..4)	P(0..4)	P(0..4)	P(0..4)	P(0..4)	P(0..4)	P(0..4)	P(0..4)
		P(0..5)	P(0..5)	P(0..5)	P(0..5)	P(0..5)	P(0..5)	P(0..5)	P(0..5)	P(0..5)
	P(0..6)	P(0..6)	P(0..6)	P(0..6)	P(0..6)	P(0..6)	P(0..6)	P(0..6)	P(0..6)	P(0..6)
P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)

P(0..4)			
P(0..5)	P(0..5)		
P(0..6)	P(0..6)	P(0..6)	
P(0..7)	P(0..7)	P(0..7)	P(0..7)

Analizziamo la complessità dell'ASOMRICAI. La complessità di spazio è n, perché l'algoritmo opera completamente "in place".
Passiamo alla complessità di tempo. L'operazione dominante è l'operazione di somma tra due numeri.
C'è un'unica operazione di somma nell'ASOMRICAI, che è effettuata nel blocco `else` a ogni autoattivazione, con esclusione di quella relativa al caso base. Poiché il numero totale di attivazioni, esclusa quella del caso base, è n-1, è immediato concludere che il numero di totale di somme è n-1.
In conclusione, la complessità di tempo dell'ASOMRICAI è $T(n) = n - 1$ somme.
Si noti che si tratta di una complessità assoluta, cioè indipendente dal valore degli elementi dell'array, e che tale complessità è identica a quella del classico algoritmo non ricorsivo di somma degli elementi di un array basato sull'approccio incrementale.
In particolare, in entrambi gli algoritmi (ricorsivo e iterativo) l'ordine secondo cui sono sommati i vari numeri è esattamente lo stesso. In altre parole, i due algoritmi descrivono la medesima sequenza computazionale di operazioni.

Come rispondere alla domanda: illustrare l'algoritmo ricorsivo di somma array, approccio divide et impera

RISPOSTA:
L'algoritmo ricorsivo di somma di un array, approccio Divide et impera, calcola la somma degli elementi di un array di dati. I dati possono essere numeri interi, numeri reali o qualsiasi altro insieme di dati su cui sia definita una operazione di somma.
L'ASOMRICDI ha come dati di input un array e il suo size, e ha come dato di output un dato scalare, che è il valore della somma di tutti gli elementi dell'array.
L'ASOMRICDI non fa uso di altri array o strutture dati.
L'ASOMRICDI è basato sull'applicazione dell'approccio Divide et impera al problema della somma degli elementi di un array.
L'approccio Divide et impera è una metodologia di progetto di algoritmi in cui la soluzione del problema viene determinata suddividendo ricorsivamente il problema in due istanze di dimensione dimezzata (fase Divide) e poi esprimendo la soluzione combinando opportunamente le due soluzioni delle istanze dimezzate (fase Impera).
La tecnica di programmazione ricorsiva, basata sull'autoattivazione di una function, consente di descrivere in modo naturale un algoritmo basato sull'approccio Divide et impera.
Nel caso del problema della somma degli elementi di un array, la soluzione del problema, cioè la somma, viene calcolata come la somma tra la somma degli elementi della porzione "di sinistra" e la somma degli elementi della porzione "di destra" in cui si può suddividere l'array.
Ogni autoattivazione suddivide una data istanza in due istanze di dimensioni dimezzate, crea un processo che risolve l'istanza del problema della somma sulla "porzione di sinistra" e un processo che risolve l'istanza del problema della somma sulla "porzione di destra" della porzione ricevuta in input e quindi restituisce la somma dei valori ottenuti dai due processi. A ogni autoattivazione, i due processi creati agiscono ognuno su una "porzione" di size dimezzato rispetto al precedente, fino ad arrivare al size 1, che è il size dell'istanza banale del problema della somma.
Diamo una rapida analisi della meccanica dell' ASOMRICDI. Alla prima attivazione si crea un processo che agisce sull'intero array A; esso crea a sua volta un processo che effettua la somma sulla porzione di sinistra dell'array A e un processo che effettua la somma sulla porzione di destra dell'array A; i valori restituiti da questi due processi sono sommati e restituiti. Le porzioni sono individuate dall'indirizzo base e dal size.
Il caso base della ricorsione è costituito dalla porzione di size 1, e in tal caso si deve restituire il valore dell'unico elemento della porzione.
Se l'istanza che il processo sta risolvendo non rientra nel caso base, allora si deve calcolare la posizione "centrale" della porzione; poi si deve autoattivare sulla porzione "di sinistra", cioè la porzione che ha lo stesso indirizzo base della porzione in input e size dimezzato, e autoattivare sulla porzione "di destra", cioè la porzione che ha come indirizzo base l'indirizzo della posizione immediatamente a destra della posizione centrale e size dimezzato; i due risultati di tali auto attivazioni devono essere sommati e restituiti al processo chiamante.
Un esempio aiuta a capire la dinamica dell'ASOMRICDI. Si consideri un array A di n=8 elementi interi.
Poiché il size 8 è diverso da 1, non siamo nel caso base.

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

Quindi, l'algoritmo deve continuare, cioè deve effettuare una prima autoattivazione che crea un processo che agisce sulla porzione di inizio 0 e size 4 e una seconda autoattivazione che crea un processo che agisce sulla porzione di inizio 4 e size 4; i valori restituiti da tali autoattivazioni devono essere sommati.

I due nuovi processi "vedono" rispettivamente le seguenti porzioni di array (si faccia attenzione agli indici locali)

Processo 1

41	31	27	28
0	1	2	3

Processo 2

52	36	11	13
0	1	2	3

Poiché il size 4 è diverso da 1, non siamo nella prima situazione del caso base. Il primo processo calcola la posizione centrale è (0+3)/2, cioè 1, e attiva due nuovi processi che agiscono rispettivamente sulla porzione 0..1 e sulla porzione 2..3 della propria porzione ricevuta. Analogamente, Il secondo processo calcola la posizione centrale è (0+3)/2, cioè 1, e attiva due nuovi processi che agiscono rispettivamente sulla porzione 0..1 e sulla porzione 2..3 della propria porzione ricevuta.

I quattro nuovi processi "vedono" rispettivamente le seguenti porzioni di array (si faccia attenzione agli indici locali)

Processo 1.1

41	31
0	1

Processo 1.2

27	28
0	1

Processo 2.1

52	36
0	1

Processo 2.2

11	13
0	1

Poiché il size 2 è diverso da 1, non siamo nella prima situazione del caso base. Il primo processo calcola la posizione centrale è (0+1)/2, cioè 0, e attiva due nuovi processi che agiscono rispettivamente sulla porzione 0 e sulla porzione 1 della propria porzione ricevuta. Analogamente, il secondo, il terzo e quarto processo calcolano la posizione centrale è (0+1)/2, cioè 0, e attivano ognuno due nuovi processi che agiscono rispettivamente sulla porzione 0 e sulla porzione 1 della propria porzione ricevuta.

Gli otto nuovi processi "vedono" rispettivamente le seguenti porzioni di array (si faccia attenzione agli indici locali)

Processo 1.1.1

41
0

Processo 1.1.2

31
0

Processo 1.2.1

27
0

Processo 1.2.2

28
0

Processo 2.1.1

52
0

Processo 2.1.2

36
0

Processo 2.2.1

11
0

Processo 2.2.2

13
0

Poiché il size è 1, siamo nella prima situazione del caso base. Quindi tutti gli otto processi restituiscono il valore dell'unico elemento della propria porzione ricevuta.
A questo punto i processi sospesi 1.1, 1.2, 2.1, 2.1 sommano i valori ricevuti e restituiscono tali somme ai processi sospesi 1 e 2, che a loro volta sommano i valori ricevuti e restituiscono tali somme al processo sospeso iniziale, che effettua la somma e restituisce la soluzione del problema.

Questa è l'implementazione in C dell'ASOMRICDI, per un array di tipo `int`:

```
int somma_a_ricDI(int a[],int n)
{
 int mediano;
/* soluzione del caso base */
 if (n == 1)
     return a[0];
   else
    /* autoattivazioni */
     {
      mediano = (n-1)/2;
      return somma_a_ricDI(a,mediano+1) + ...
             somma_a_ricDI(a+mediano+1,n-mediano-1);
      }
}
```

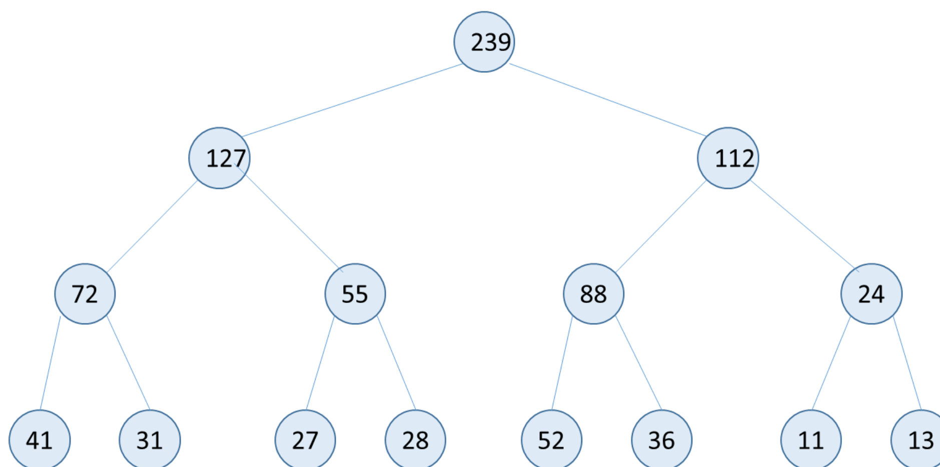
Un breve commento al codice.
La function ha la classica struttura delle function ricorsive, cioè un `if-the-else` che distingue il caso base dalle istanze non banali.
Il blocco `else` gestisce l'autoattivazione. La prima autoattivazione agisce sulla semi-porzione di sinistra, che ha lo stesso indirizzo base della porzione ricevuta in input e size uguale al valore dell'indice **mediano+1**; 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**. La soluzione dell'istanza è calcolata sommando i valori restituiti dalle due autoattivazioni ed è restituita al chiamante.
Poiché una espressione è eseguita da "sinistra verso destra", l'autoattivazione sulla porzione di sinistra (che appare come primo addendo nell'espressione **somma_a_ricDI(a,mediano+1) + somma_a_ricDI(a+mediano+1,n-mediano-1)**) è eseguita prima dell'autoattivazione sulla porzione di destra. Quindi l'esecuzione dell'ASOMRICDI dà luogo alla seguente sequenza di attivazione/sospensione dei processi nello stack (per semplicità si usano gli indici globali; ogni colonna "fotografa" lo stack dopo ogni autoattivazione):

		P(0)	P(1)		P(2)	P(3)				
		P(0..1)	P(0..1)		P(2..3)	P(2..3)				P(4..5)
	P(0..3)	P(0..3)	P(0..3)	P(0..3)	P(0..3)	P(0..3)	P(0..3)		P(4..7)	P(4..7)
P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)

P(4)	P(5)		P(6)	P(7)		
P(4..5)	P(4..5)		P(6..7)	P(6..7)		
P(4..7)	P(4..7)	P(4..7)	P(4..7)	P(4..7)	P(4..7)	
P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)	P(0..7)

Analizziamo la complessità dell'ASOMRICDI. La complessità di spazio è n, perché l'algoritmo opera completamente "in place".
Passiamo alla complessità di tempo. L'operazione dominante è l'operazione di somma tra due numeri.
Il modo più semplice per determinare la complessità di tempo dell'ASOMRICDI consiste nell'esaminare il cosiddetto albero binario delle operazioni associato all'esecuzione dell'algoritmo per un dato problema.
I nodi foglia di tale albero sono gli elementi dell'array, i nodi interni contengono i valori delle somme delle varie porzioni in cui l'array viene suddiviso durante le successive autoattivazioni, e il livello in cui appaiono nell'albero corrisponde al livello di autoattivazione. In altre parole, la radice dell'albero contiene la soluzione del problema, ottenuta sommando dal basso verso l'alto le varie coppie di numeri.

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



E' chiaro esaminando l'albero che per ottenere il valore di ciascuno dei nodi interni (cioè tutti i nodi escluse le foglie) è necessario effettuare esattamente una somma; quindi il numero totale di somme è uguale al numero dei nodi interni. Poiché il numero delle foglie è n, in base a una delle proprietà degli alberi binari (completi), si ha che il numero dei nodi interni è n-1.
In conclusione, la complessità di tempo dell'ASOMRICDI è $T(n) = n - 1$ somme.

Si noti che si tratta di una complessità assoluta, cioè indipendente dal valore degli elementi dell'array, e che tale complessità è identica a quella del classico algoritmo non ricorsivo di somma degli elementi di un array basato sull'approccio incrementale.