Fabrizio Luccio. Appunti di algoritmica

4 Il problema della ricerca in un insieme

4.1 Ricerca del massimo

Consideriamo un insieme $R = \{r_0, r_1, \ldots, r_{n-1}\}$ di n elementi diversi tra loro per cui è definita una relazione di ordinamento totale indicata con i simboli $<, >$ (scriviamo $a < b$, o $b > a$). Un problema è quello di determinare il *massimo* elemento m di R, ovvero quello $>$ di tutti gli altri.

Ovviamente si tratta di un problema semplicissimo se si dispone di una operazione primitiva di confronto tra elementi, ma vogliamo investigarlo con rigore. In particolare ponendo che il confronto tra due elementi richieda tempo costante, e tale confronto sia l'operazione preminente nei possibili algoritmi che risolvono il problema, valuteremo la complessità di questi come numero di confronti $C(n)$ eseguiti in funzione di n, perché il tempo di calcolo sarà proporzionale a tale numero.

Anzitutto valutiamo un **limite inferiore** a $C(n)$ osservando che, a causa dell'estrema semplicità del problema, potremo presumibilmente calcolare tale limite in modo preciso anziché in ordine di grandezza.

1. Utilizziamo il criterio dell'*albero di decisione* già visto per il problema delle 12 monete. Il problema presente ha $s = n$ soluzioni: $m = r_0, m = r_1, \ldots, m = r_{n-1}$. Ogni confronto genera due possibilità, quindi i confronti successivi generano un *albero binario* che contiene le $s = n$ soluzioni del problema nelle foglie. L'albero binario con n foglie che minimizza il più lungo percorso radice-foglia (relativo al caso pessimo per una soluzione) è quello perfettamente bilanciato in cui tutti i percorsi contengono $\log_2 n$ nodi di diramazione. Ne segue un limite inferiore $C(n) \geq \log_2 n$.

2. Un ragionamento diverso è basato sulla necessità che tutti gli elementi di R entrino in almeno un confronto, altrimenti non potremmo asserire che un elemento mai confrontato non sia massimo. Poiché in Ogni confronto riguarda due elementi almeno $n/2$ confronti sono necessari, ovvero $C(n) \geq n/2$.

3. Un terzo ragionamento è basato sull'osservazione che la determinazione del massimo implica la certezza che gli altri $n - 1$ elementi di R non siano il massimo. Poiché per certificare che un elemento non è il massimo occorre che risulti minore di un altro in almeno un confronto, e da ogni confronto esce esattamente un perdente, sono necessari $n - 1$ confronti per dichiarare $n - 1$ perdenti, ovvero $C(n) \geq n - 1$.

Questi tre limiti inferiori sono tutti legittimi, ma il terzo è superiore agli altri due ed è quindi il più significativo. Concludiamo dunque per ora $C(n) \geq n - 1$, salvo determinare un limite superiore più alto di $n-1$ che lascerebbe aperto il problema di innalzare eventualmente il limite inferiore trovato. Vedremo ora che ciò non accade.

Valutiamo dunque il **limite superiore** a $C(n)$ che, come sappiamo, si ottiene come numero di confronti eseguito dal miglior algoritmo che saremo in grado di trovare. Proponiamo anzitutto un algoritmo in forma *iterativa*, che specifica la sequenza di operazioni in modo esplicito. Ovviamente formuleremo l'algoritmo come programma per un computer.

Poniamo che l'insieme sia memorizzato in un vettore $A[0...n-1]$ con $n > 1$ e supponiamo che il confronto tra elementi si esegua in tempo costante. Un algoritmo iterativo elementare di ricerca del massimo, detto MASSIMO-ITER, consiste nello scandire il vettore A per valori crescenti dell'indice, memorizzando in m, al passo i-esimo, il valore massimo trovato nella porzione del vettore tra $A[0]$ e $A[i]$. Il semplicissimo programma è:

MASSIMO-ITER(A, n, m)
// Ricerca del massimo elemento in un vettore $A[0...n-1]$.
input A, n; **output** m;
$m = A[0]$;
for $(i = 1; i \leq n-1; i++)$ **if** $(m < A[i])$ $m = A[i]$;

Un'analisi elementare dell'algoritmo mostra che questo è corretto ed esegue esattamente $n-1$ confronti. Ovvero un limite superiore è $C(n) \leq n-1$. Poiché questo valore è uguale al limite inferiore trovato prima l'algoritmo MASSIMO-ITER è ottimo e il problema è computazionalmente chiuso. Vogliamo però studiare anche un importante metodo diverso e ugualmente ottimo, basato sul paradigma *ricorsivo* (in alternativa al paradigma iterativo considerato sopra).

In un algoritmo ricorsivo una procedura su dati di dimensione n chiama sé stessa su sottoinsiemi di tali dati, finché questi sottoinsiemi raggiungono una dimensione costante (e piccola) per cui il problema è risolto in modo diretto. Tale formulazione, che si presenta a volte come la più spontanea per molti problemi, è anche assai utile perché consente di dimostrare per induzione la correttezza del programma, e si presta a uno studio matematico della complessità in tempo. Per contro diviene però spesso difficile comprendere la sequenza di operazioni effettivamente eseguita dal programma.

Per il problema della ricerca del massimo un algoritmo ricorsivo effettua il calcolo secondo la nota struttura di un torneo a eliminazione diretta: infatti il confronto tra due elementi per determinare il maggiore può essere visto come un confronto sportivo in cui il più forte vince ed elimina l'avversario. Ammettendo per semplicità che sia $n = 2^t$, gli elementi si confrontano in coppie e restano in gara $n/2$ vincitori dopo $n/2 = 2^{t-1}$ confronti; i vincitori si confrontano a coppie, riducendo i partecipanti a 2^{t-2} dopo altrettanti confronti; il torneo prosegue fino alla finale tra $2^1 = 2$ partecipanti, da cui emerge il massimo. Secondo una formula già vista gli incontri sono $\sum_{i=0}^{t-1} 2^i = 2^t - 1 = n - 1$. Dunque anche questo algoritmo è ottimo, anche se è bene notare che gli $n-1$ confronti sono eseguiti in ordine completamente diverso da quelli dell'algoritmo precedente.

Il metodo del torneo è puramente ricorsivo perché il massimo è ottenuto dal confronto finale tra i massimi precedentemente ottenuti tra le due metà dell'insieme R (o tra le due metà del tabellone nella interpretazione sportiva): gli n dati sono cioé divisi in due sottoinsiemi di $n/2$ elementi cui si applica lo stesso algoritmo che "ricorsivamente" divide le due metà dell'insieme in quarti e così via, finché i sottoinsiemi contengono esattamente due elementi e il massimo si determina in altro modo,

cioè con un confronto diretto. Dunque a questo punto iniziano i confronti mentre il resto dell'algoritmo serve a organizzare il calcolo. Il programma MASSIMO-RICOR realizza l'algoritmo specificando come parametri d'ingresso le due posizioni i, j del vettore A che delimitano il sottoinsieme in ogni chiamata della procedura (si ha sempre $i < j$). Il programma è innescato dall'esterno con la chiamata MASSIMO-RICOR$(A, 0, n-1, m)$ sull'intero vettore. La condizione limite in cui un sottoinsieme è ridotto a due elementi si verifica quando $i = j - 1$.

Strettamente parlando l'algoritmo si applica ai valori $n = 2^t$ e richiede una piccola modifica se n non è una potenza di 2. Il caso $n = 2^t$ meglio indica il senso del paradigma ricorsivo.

MASSIMO-RICOR(A, i, j, m)
// Ricerca ricorsiva del massimo in un sotto-vettore di A tra le posizioni i, j su cui lavora la particolare "chiamata" della procedura. Il calcolo nell'intero vettore A è innescato dai valori iniziali $i = 0$, $j = n - 1$, comunicati dall'esterno.

input A, i, j; **output** m;
if $(j - i == 1)$ {**if** $(A[i] < A[j])$ $m = A[j]$ **else** $m = A[i]$;
return m};
$k = \lfloor (i + j)/2 \rfloor$; *//Calcolo dell'indice di mezzo tra i e j*
MASSIMO-RICOR(A, i, k, m_1);
MASSIMO-RICOR$(A, k + 1, j, m_2)$;
if $(m_1 < m_2)$ $m = m_2$ **else** $m = m_1$;

La correttezza dell'algoritmo si dimostra impiegando il principio d'induzione. La base è per $n = 2$ (quindi $i = 0, j = 1$): in questo caso il massimo m è individuato dalla frase **if** iniziale. Per $n = 2^t$ con $t > 1$, poniamo che la procedura calcoli correttamente il massimo per ogni valore $n' < n$: si ha quindi per ipotesi che le due chiamate ricorsive di MASSIMO-RICOR calcolano correttamente i massimi m_1, m_2 nelle due metà del vettore e il calcolo di m è quindi corretto come mostra la frase **if** finale.

Il numero $C(n)$ di confronti tra elementi è soggetto alla seguente *relazione di ricorrenza* che si ottiene immediatamente da un esame del programma:

$$C(n) = 1, \qquad \text{per } n = 2;$$
$$C(n) = 2\,C(n/2) + 1, \quad \text{per } n > 2.$$

Infatti se $n = 2$ il programma esegue un solo confronto senza che le chiamate ricorsive entrino in azione. Per $n > 2$ la procedura richiama se stessa su due insiemi grandi $n/2$ (onde il termine $2\,C(n/2)$) e poi esegue un confronto nella frase **if** finale. Per esprimere esplicitamente $C(n)$ come funzione di n possiamo effettuare lo sviluppo seguente, ove la formula di $C(n)$ è via via applicata a insiemi di dimensioni 1/2, 1/4, 1/8 ecc. su cui operano le chiamate ricorsive, fino a un insieme finale di due elementi cui si applica il valore di $C(2) = 1$. Ponendo $n = 2^t$ si ha:

$$
\begin{aligned}
C(2^t) &= 2\,C(2^{t-1}) + 1 \\
&= 2\,(2\,C(2^{t-2}) + 1) + 1 \\
&\quad = 4\,C(2^{t-2}) + 2 + 1 \\
&= 4\,(2\,C(2^{t-3}) + 1) + 2 + 1 \\
&\quad = 8\,C(2^{t-3}) + 4 + 2 + 1 \\
&= \cdots\cdots\cdots \\
&\quad = 2^{t-1}(C(2^{t-(t-1)}) + 2^{t-2} + 2^{t-3} + \ldots + 1 \\
&\quad = 2^{t-1} + 2^{t-2} + 2^{t-3} + \ldots + 1 \\
&\quad = 2^t - 1 = n - 1.
\end{aligned}
$$

Otteniamo anche in questo caso il limite superiore $n-1$ per $C(n)$: anche questo algoritmo è ottimo benché completamente diverso dal primo.

Esercizio 1. Dato un insieme R di $n = 2^t$ interi positivi diversi tra loro e non ordinati, con $t \geq 1$, si devono determinare il massimo m e il secondo massimo s (cioè il massimo in $R - \{s\}$). L'algoritmo dovrà essere organizzato in due modi:

1. **Facile**. Scandire l'insieme mantenendo a ogni passo i valori di m, s tra gli elementi fin lì esaminati. Calcolare poi il numero di confronti $C(n)$ effettuati dall'algoritmo nel caso pessimo (dovrà risultare $C(n) = 2n - 3$). Qual'è il caso pessimo?

2. **Difficile**. Dividere l'insieme in due metà e risolvere ricorsivamente il problema determinando i valori di m, s per tali metà e confrontandoli per ottenere m, s per l'intero insieme. Calcolare poi il numero di confronti $C(n)$ effettuati dall'algoritmo nel caso pessimo (dovrà risultare $C(n) = 3n/2 - 2$). Qual'è il caso pessimo?

Nota. Non è richiesto di formalizzare i due algoritmi in un linguaggio di programmazione. È sufficiente e più istruttivo immaginare semplicemente come ciascun algoritmo debba essere strutturato: si vedrà che il calcolo di $C(n)$ può essere eseguito anche considerando la sola struttura dell'algoritmo. Scelto così il migliore dei due si potrà affrontare la sua realizzazione pratica sotto forma di un programma.

4.2 Ricerca di un elemento

La ricerca di un dato k in un insieme di n dati è uno dei problemi fondamentali nella costruzione di algoritmi. In particolare desideriamo individuare, se esiste, un dato uguale a k perché tale dato normalmente rappresenta un "nome" cui è associata un'informazione aggiuntiva: l'esempio più ovvio è la ricerca di un nome k in una rubrica, per estrarre da questa i dati associati a quel nome.

Proponiamo anzitutto un algoritmo di soluzione che permette di stabilire un limite superiore alla complessità in tempo del problema, formulando l'algoritmo come programma in forma iterativa. Poniamo che gli elementi dell'insieme siano memorizzati nelle celle $A[0], \ldots, A[n-1]$ di un vettore $A[0...n]$ ove la cella $A[n]$ sia usata per controllo, e supponiamo che il confronto tra elementi, le operazioni aritmetiche e di controllo degli indici si eseguano in tempo costante. Studieremo

così il problema tenendo conto di tutte le operazioni eseguite (ciè non soltanto dei confronti).

Un algoritmo *iterativo* elementare di ricerca, detto RICERCA-SEQUENZIALE, consiste nel confrontare in sequenza i dati contenuti in $A[0], \ldots, A[n\text{-}1]$ con k fino a individuarlo se è presente nell'insieme, o raggiungere l'ultimo elemento con esito negativo e stabilire dunque che k non è presente. Ponendo che la risposta dell'algoritmo sia fornita attraverso un parametro intero r, con $0 \leq r \leq n-1$, che indica la posizione di k nell'insieme se tale valore è presente nella cella $A[r]$, e $r = -1$ se k non è presente nell'insieme, l'algoritmo può essere espresso come:

RICERCA-SEQUENZIALE(k, A, r)
// *Ricerca dell'elemento* k *nelle posizioni* $A[0], \ldots, A[n\text{-}1]$ *di un vettore* $A[0...n]$
input k, A; **output** r;
$A[n] = k$;
$i = 0$;
while $(k \neq A[i])$ $i = i + 1$;
if $(i \leq n-1)$ $r = i$ **else** $r = -1$;

La memorizzazione iniziale di k nella cella $A[n]$ garantisce la terminazione del ciclo while su $k = A[n]$ anche nel caso che k non sia compreso nell'insieme: l'algoritmo è in genere riportato in forma un po' diversa nei libri di algoritmica, e la presente ne costituisce una versione migliorata.

RICERCA-SEQUENZIALE si arresta dopo $i+1$ ripetizioni del ciclo se k appare in $A[i]$ con $i \leq n-1$, altrimenti richiede $n+1$ iterazioni se k non è contenuto nell'insieme. Il caso pessimo è dunque quest'ultimo. Poiché tutte le operazioni in ogni iterazione richiedono tempo costante, il tempo $T(n)$ richiesto dall'algoritmo è proporzionale al numero di iterazioni, ovvero $T(n) = O(n)$.

Un metodo alternativo per eseguire la ricerca utilizza il paradigma *ricorsivo*. Un esempio fondamentale è rappresentato dal seguente algoritmo RICERCA-BINARIA che prevede che sia definita una relazione di ordinamento tra i dati (indicata con il simbolo $\leq$), e lavora su un insieme ordinato in ordine crescente. L'algoritmo costituisce una formulazione in termini precisi del metodo usato manualmente per reperire un dato in una rubrica come per esempio l'elenco del telefono. Si inizia in un punto ragionevole dell'elenco (il computer inizia esattamente nel centro); se il dato d ivi contenuto coincide con l'elemento k cercato ci si arresta con successo, altrimenti si ripete l'operazione nella metà sinistra dell'elenco se $k < d$, o nella metà destra se $k > d$, fino a ridursi a un sotto-elenco vuoto che indica che k non è presente. Realizzando l'algoritmo in un programma l'insieme è contenuto in un vettore $A[0...n-1]$ i cui elementi sono disposti in ordine crescente, e il risultato r ha lo stesso significato visto per la ricerca sequenziale.

RICERCA-BINARIA(k, A, i, j, r)
// *Ricerca dell'elemento k in un vettore $A[0...n\text{-}1]$ ordinato in ordine crescente. I parametri i, j indicano gli estremi del sotto-vettore $A[i...j]$ su cui la particolare "chiamata" della procedura deve lavorare: il calcolo è innescato dai valori iniziali $i = 0$, $j = n - 1$, comunicati dall'esterno assieme a k.*

input k, A, i, j; **output** r;
if $(i > j)$ $\{r = -1$; **stop**;$\}$
else $\{m = \lfloor (i + j)/2 \rfloor$;
if $(k == A[m])$ $\{r = m$; **stop**;$\}$
if $(k < A[m])$ RICERCA-BINARIA$(k, A, i, m - 1, r)$
else RICERCA-BINARIA$(k, A, m + 1, j, r)$;
$\}$

Per dimostrare la correttezza della procedura RICERCA-BINARIA notiamo prima di tutto che se k non è presente nel vettore la ricerca giungerà a confrontare k con un sotto-vettore composto da un solo elemento $A[s]$ attraverso la chiamata ricorsiva RICERCA-BINARIA(k, A, s, s, r) che produce $m = s$, nella quale il confronto **if** $(k < A[m])$ innescherà la chiamata ricorsiva RICERCA-BINARIA$(k, A, s, s - 1, r)$ o RICERCA-BINARIA$(k, A, s + 1, s, r)$ che arrestano la procedura restituendo il valore $r = -1$.

Se invece k è contenuto nel vettore la dimostazione procede per induzione su n.

- La base d'induzione è $n = 1$, cioè A contiene il solo valore $A[0] = k$: in questo caso la chiamata iniziale RICERCA-BINARIA$(k, A, 0, 0, r)$ produce $m = 0$ e si chiude con il confronto positivo if$(k == A[0])$.
- Per $n > 1$, posto che la procedura sia corretta per ogni $n' \leq n - 1$, cioè che le due chiamate ricorsive interne alla procedura individuino la posizione di k se presente nel relativo sotto-vettore, la correttezza complessiva deriva immediatamente dall'osservazione che o il test **if** $(k == m)$ individua k, o lo individuerà una delle successive chiamate interne.

Per quanto riguarda la valutazione della complessità in tempo dalla procedura RICERCA-BINARIA, il tempo $T(n)$ da essa richiesto è soggetto alla seguente *relazione di ricorrenza* che si ottiene immediatamente da un esame del programma:

$T(n) = c_1$, con c_1 costante, per $n = 1$;

$T(n) \leq T(n/2) + c_2$, con c_2 costante, per $n > 1$.

Infatti se $n = 1$ il programma richiede un numero costante c_1 di passi per confrontare k con $A[0]$, senza che le chiamate ricorsive entrino in azione. Per $n > 1$ il caso pessimo si ha quando k non è presente in A. In questo caso, oltre a un numero costante c_2 di operazioni di test e calcolo degli indici, si richiama la procedura su un insieme grande $\lfloor n/2 \rfloor$, onde il relativo termine addizionato nella funzione. Per esprimere esplicitamente $T(n)$ come funzione di n possiamo effettuare lo sviluppo

seguente, ove la formula di $T(n)$ è via via applicata a insiemi di dimensioni 1/2, 1/4, 1/8 ecc. su cui operano le chiamate ricorsive, fino a un insieme finale di un unico elemento cui si applica il valore di $T(1)$. Nell'ipotesi semplificativa che sia $n = 2^t$ per un opportuno intero t si ha:

$$\begin{aligned} T(n) &\leq T(n/2) + c_2 \\ &= (T(n/4) + c_2) + c_2 = T(n/4) + 2c_2 \\ &= (T(n/8) + c_2) + 2c_2 = T(n/8) + 3c_2 \\ &= ..... \\ &= (T(n/2^t) + c_2) + (t-1)c_2 = T(1) + tc_2 \\ &= c_1 + tc_2. \end{aligned}$$

Invertendo la relazione $n = 2^t$ abbiamo $t = \log_2 n$, dunque l'algoritmo ha complessità logaritmica $T(n) = O(\log n)$ (si ricordi che la base del logaritmo non appare nell'ordine di grandezza perché i logarimni di un numero calcolati in diverse basi sono correlati tra loro da una costante moltiplicativa). Se n non è una potenza di due il calcolo sopra è approssimato perché i termini $T(n/2^i)$ non sono interi, ma l'ordine di grandezza di $T(n)$ non cambia.

Vediamo ora come stabilire un limite inferiore alla complessità in tempo del problema. Se non vi è alcuna relazione tra gli elementi dell'insieme e il modo come essi vengono presentati (in particolare se non sono ordinati) non è possibile per esempio stabilire che k non appartiene all'insieme senza esaminare tutti gli n elementi di questo, il che ci consente di stabilire un limite inferiore di complessità del problema pari a $\Omega(n)$. Poiché la procedura RICERCA-SEQUENZIALE vista all'inizio ha complessità $O(n)$ e quindi eguaglia (in ordine di grandezza) il limite inferiore trovato, possiamo concludere che il semplice ragionamento eseguito per determinare il limite inferiore è del tutto soddisfacente e quella procedura è ottima.

Se invece è possibile eseguire una elaborazione preventiva sull'insieme il ragionamento precedente sul limite inferiore non è più valido. In particolare se gli elementi dell'insieme sono ordinati vale la proprietà transitiva della relazione "$\leq$", secondo la quale se abbiamo per esempio scoperto che $k < A[i]$ sappiamo per transitività che $k < A[j]$ per ogni $j > i$, senza dover eseguire questi confronti. Possiamo quindi determinare un limite inferiore alla complessità del problema considerando il numero minimo $min(n)$ di confronti tra elementi, che devono essere necessariamente eseguiti nel caso pessimo per individuare un dato k in un insieme ordinato, o stabilire che non vi è contenuto. Il valore di $min(n)$ costituirà così un limite inferiore per il problema, salvo dimostrare che altre operazioni siano più significative in questo caso, ed esista un limite inferiore più alto. A tale scopo utilizziamo un albero di decisione

Il confronto tra k e un elemento $A[i]$ può generare le tre risposte $k<A[i]$, $k=A[i]$, o $k>A[i]$, quindi aprire tre percorsi di computazione nei quali successivi confronti moltilicheranno per tre, a ogni passo, le sitauzioni possibili: in effetti se $k=A[i]$ il percorso si arresta subito con l'individuazione della soluzione per il particolare insieme considerato. Dunque dopo t confronti possono essersi aperte al massimo 3^t

vie diverse, al termine delle quali saranno allocate le diverse soluzioni del problema per ogni possibile valore dei dati d'ingresso. Notando che:

- le possibili soluzioni distinte del problema sono $s = n+1$ e cioè: k corrisponde al primo, secondo, ... n-esimo dato dell'insieme, o k non è contenuto nell'insieme;
- come sappiamo ciascuna soluzione deve trovarsi all'estremo di un percorso di compitazione eseguito da un qualsiasi algoritmo di soluzione del problema;
- tali estremi sono al massimo 3^t e deve risultare $3^t \geq s$, quindi $3^t > n$;

applicando il logaritmo a base 3 nella disuguaglianza $3^t > n$, e ricordando che t è un numero di confronti e deve essere approssimato come intero, risulta $t \geq \lceil \log_3 n \rceil$.

Ricordando infine che t è il numero di confronti nel più lungo percorso di computazione (cioè nel caso pessimo: alcuni percorsi potrebbero terminare prima), cioè $min(n) = t$, si ha che $min(n)$ è di ordine $\Omega(\log n)$, che costituisce un limite inferiore alla complessità del problema. Poiché questo limite coincide con il limite superiore $O(\log n)$ stabilito dalla procedura RICERCA-BINARIA, concludiamo che quella procedura è ottima e che il calcolo del limite inferiore basato sui confronti è sufficiente.

Esercizio 2. Dato un insieme R di $n \geq 2$ interi positivi diversi tra loro e *ordinati*, e un intero k, si devono determinare tutti gli elementi x con $k - 10 \leq x \leq k$. L'algoritmo dovrà essere organizzato scandendo l'insieme, oppure utilizzando una ricerca simile a quella binaria (come?). Calcolare la complessità in tempo dei due algoritmi nel caso pessimo.

Nota. Nel secondo algoritmo si tratta di applicare l'algoritmo di ricerca binaria a $y = k - 10$ e $y = k$, in una semplice trasformazione che gli consenta di trovare l'elemento s immediatamente successivo a y, e l'elemento p immediatamente precedente a y, nel caso che y non sia contenuto nel vettore. A tale proposito dimostrare la seguente cruciale affermazione:

Se y non è contenuto nel vettore, l'algoritmo standard di ricerca binaria si ferma su un sottoinsieme vuoto e l'ultimo elemento precedentemente visitato è p (o rispettivamente s) se l'ultimo passo è stato eseguito verso destra (o rispettivamente sinistra).

L'algoritmo di ricerca binaria dovrà quindi essere modificato tenendo nota della direzione presa all'ultimo passo.

Come per l'esercizio 1 non è richiesto di formalizzare i due algoritmi in un linguaggio di programmazione ma è sufficiente immaginarne la struttura e calcolare la complessità in tempo da essa.