Complessità degli algoritmi di ricerca in strutture dati

Struttura di dato	**Algoritmo di ricerca applicabile**	**Complessità asintotica (caso peggiore)**
Tavola come array non ordinato (N elementi)	Ricerca sequenziale	O(N)
Tavola come array ordinato (N elementi)	Ricerca binaria	$O(\log_2 N)$
	Ricerca sequenziale	O(N)
File non ordinato (N elementi)	*Ricerca sequenziale*	*O(N)*
File ordinato (N elementi)	*Ricerca binaria*	*$O(\log_2 N)$*
	Ricerca sequenziale	*O(N)*
Lista collegata (anche se ordinata, N elementi)	Ricerca sequenziale	O(N)
Alberi binari (N nodi)	Ricerca esaustiva (una delle procedure visita)	O(N)
Alberi binari di ricerca (N nodi)	su campo chiave: Ricerca binaria	O(h), h altezza dell'albero (che vale $\log_2 N$ **solo se l'albero è bilanciato**)
	Su campo non chiave: Ricerca esaustiva (visita)	O(N)

FONDAMENTI DI INFORMATICA – 19 Dicembre 2018

Sia data la seguente funzione **fun** che riceve un carattere e una lista ordinata di caratteri:

```
int fun(char i, list L)
{  if (L==NULL) return 0;
   else   if (i==L->value) return 1;
          else   return fun(i,L->next);
}
```

Si indichi cosa fa la funzione **fun** e se e ricorsiva o iterativa. Nell'ipotesi che la lista abbia N elementi, si individui il caso peggiore e se ne valuti la complessità asintotica del numero di test (i==L->value), motivando adeguatamente.

Soluzione:
La funzione fun, rircorsiva, cerca l'elemento i nella lista L.

Che L sia ordinata non è rilevante, la ricerca è comunque sequenziale.

Nel caso peggiore, in cui i non compare in L, il test è eseguito per tutti gli elementi di L, quindi la complessità asintotica è O(N).

FONDAMENTI DI INFORMATICA – 12 Settembre 2011

Dato il seguente codice in linguaggio C, dove il tipo **list** rappresenta una lista di interi, **insord** la funzione di inserimento ordinato in senso crescente in una lista e **member** la funzione di ricerca binaria in un vettore ordinato di interi:

```
#include <stdio.h>
void main(void)
{ int i, c=0;
  int v[10] = {0,1,2,3,4,5,6,7,8,9};
  list L1 = NULL;

  scanf("%d",&i);
  do  { L1=insord(i,L1);
        scanf("%d",&i);   }
  while (i>0);

  while (L1!=NULL)
     { if ( member(L1->value,v) ) c++;
       L1=L1->next; }
  printf("\n%d\n", c); }
```

a) Si indichi cosa fa questo frammento di programma;

b) Se ne indichi la complessità in termini di numero di test condizionali eseguiti dalla funzione **member**, ipotizzando che da input siano stati inseriti N valori distinti (da 1 a N) seguiti da 0. Si distinguano caso migliore, peggiore e medio (il vettore **v** è dato e di dimensione 10) motivando opportunamente.

Soluzione: Il programma crea una lista di N+1 interi (N positivi, e 0). Poi conta quanti elementi di questa lista appartengono al vettore **v**, ed infine stampa a video tale valore.

Caso migliore: tutti gli N valori positivi sono uguali al mediano di **v**, la complessità è N volte un confronto:

$$N*1$$

Caso peggiore: tutti gli N valori positivi sono diversi da qualsiasi elemento di **v**, la complessità è N volte * $\log_2(10)$:

$$(N+1)*\log_2(10)$$

Caso medio: la ricerca binaria costa nel caso medio come nel caso peggiore:

$$N*\log_2(10)$$

In tutti e tre i casi, la ricerca binaria dell'N+1-esimo valore della lista (lo 0), paga un costo log2(10), da sommarsi alle formule precedenti, ma di fatto trascurabile.

FONDAMENTI DI INFORMATICA II– 2013 Maggio 23

Dato il seguente codice in linguaggio C, dove il tipo **list** rappresenta una lista di interi (non ordinata):

```
int p(list  A, list L)
{ int c=0;
  while (A!=NULL)
          {if  (search(A->value, L)) c++;
           A=A->next;}
 return c; }

int search(int i, list L)
{ if (L!=NULL)
         {if (i==L->value) return 1+search(i,L->next);
          else return search(i, L->next);
          }
  else return 0;  }
```

a) Si descriva cosa restituisce la funzione **p** e che tipo di ricerca fa la funzione **search**.
b) Si discuta la complessità del codice per la chiamata **p(l1,l2)** sapendo che la lista **l1** contiene **M** elementi e la lista **l2** contiene **N** elementi, per la chiamata **p(l1,l2)** (contare le esecuzioni complessive del test sottolineato nella funzione **search**).

Soluzione:

La funzione **p** conta gli elementi comuni alle due liste. Per ogni elemento della prima lista, chiama la funzione **search** per cercarlo nella seconda, con ricerca sequenziale.

Se la lista **l1** contiene **M** elementi e la lista **l2** contiene **N** elementi, la complessità della chiamata **p(l1,l2)** data dal numero di esecuzioni complessive del test sottolineato nella funzione **search** è: **M x N**

FONDAMENTI DI INFORMATICA - 27 Marzo 2013

Dato il seguente codice in linguaggio C, dove il tipo **tree** rappresenta un albero binario di interi e **ordins** la funzione di inserimento in un albero binario di ricerca:

```
#include <stdio.h>
int fun(int e, tree T)
{ if (T==NULL) return 0;
  else   if (e==T->value) return 1;
         else  if (e<T->value) return fun(e,T->left);
               else return fun(e,T->right);   }

void main(void)
{ int i, sum=0;
  tree T =NULL;
  scanf("%d",&i);
  while (i>0)  { T=ordins(i,T);
                 scanf("%d",&i); }
  for(i=0; i<10; i++)
      if ( fun(i,T) )  sum=sum+1;
  printf("%d",sum);   }
```

- Si indichi cosa fa il programma e la funzione **fun** (cosa viene stampato dal main?);
- Se ne indichi la complessità in termini di numero di test condizionali dell'istruzione **if** (test sottolineato) nella funzione **fun**, ipotizzando che da input siano stati dati **M** valori **(nessuno uguale a 0,1, ..9)** che, inseriti nell'albero **T** , hanno prodotto un albero perfettamente bilanciato.

Soluzione:
Il programma crea un albero binario di ricerca di interi, leggendo i valori da input. Cerca poi con la funzione **fun** i primi 10 valori interi da 0 a 9, e stampa quanti di questi compaiono nell'albero.
fun esegue una ricerca binaria nell'albero.

Se da input sono stati dati **M** valori **(nessuno uguale a 0,1, ..9 – caso peggiore)** che, inseriti nell'albero **T** , hanno prodotto un albero perfettamente bilanciato, in ciascuna delle dieci chiamate della funzione, la complessità è $\log_2 M$, quindi in complesso:

$$10 * \log_2 M$$

FONDAMENTI DI INFORMATICA – modulo B – 2015 Gennaio 13

Dato il seguente codice in linguaggio C, dove il tipo **tree** rappresenta un albero binario di caratteri e **ord_ins** la funzione di inserimento per alberi binari di ricerca:

```
#include <stdio.h>
. . .
int m (char i, tree T)
{ if (T==NULL) return 0;
  else  if (i==T->value) return (1 + m(i,T->left));
        else
           if (i<T->value) return m(i,T->left);
           else return m(i,T->right); }

void main(void)
{ char i;
  tree T=NULL;
  scanf("%c",&i);
  do { T=ord_ins(i,T);
       scanf("%c",&i); }
  while ((i>='A') and (i<='Z'));
  scanf("%c",&i);
  printf("%d",m(i,T)); }
```

a) Si indichi cosa fa questo frammento di programma e la funzione **m** in particolare.

b) Si supponga che il ciclo **do** legga da input N caratteri compresi tra 'A' e 'Z', poi '*'. Si valuti la complessità come numero di attivazioni della funzione **m**, nel caso in cui il valore letto successivamente all'uscita dal ciclo **do** non sia presente nell'albero. Si discutano i casi migliore e peggiore e li si motivi adeguatamente.

Soluzione:
Il programma conta e stampa quante volte il carattere letto compare nell'albero, tramite la funzione **m**.

Il caso migliore si ha quando il carattere è la radice dell'albero, costo: 1
Il caso peggiore, quando il carattere non appartiene all'albero, costo pari all'altezza dell'albero.
Casi particolari:

- se albero **completamente sbilanciato**, l'altezza diventa N+1 (N, numero dei nodi),
- se albero **bilanciato**, l'altezza è $\log_2(N+1)$.

FONDAMENTI DI INFORMATICA – 2016 Febbraio 18

Esercizio n. 1

Dato il tipo **`tree`** per un albero binario di interi, sia **`ordins(int,tree)`** la funzione che inserisce un intero in un albero binario di ricerca. Si consideri il codice seguente:

```
int  common(tree A, tree B)
{ if ((A!=NULL)&&(B!=NULL))
       { if ( A->value == B->value )          /* istr. dominante */
            return 1+common(A->left,B->left)+common(A->right,B->right);
         else return 0; }
   else return 0;
}

void main(void)
{ int i;
  tree T1=NULL;
  tree T2=NULL;
  scanf("%d",&i);
  while (i>0)
       {  T1=ordins(i,T1);
          T2=ordins(i,T2);
         scanf("%d",&i);   }
 printf("%d",common(T1,T2));  }
```

a) Si descriva cosa fa la funzione **`common`**;

b) Si discuta la complessità del codice sapendo che **`T1`**, e **`T2`** costruiti nel **`main`**, hanno **M** elementi. Individuare la complessità come numero di esecuzioni dell'istruzione dominante nella funzione **`common`** (codice sottolineato), motivando opportunamente.

La funzione **`common`** verifica se gli alberi sono identici (stessi valori, stesso profilo). Se non lo sono, restituisce 0, se lo sono restituisce il numero dei nodi.
Di fatto per come sono creati, lo sono.
La funzione esegue una visita in pre-ordine, e tocca tutti i nodi (di ciascun albero): la complessità (asintotica) è pari al numero dei nodi, O(**M**).

FONDAMENTI DI INFORMATICA – 19 Febbraio 2013

Dato il seguente codice in linguaggio C, dove il tipo **list** rappresenta una lista di interi, e **tree** un tipo albero binario (di ricerca). Le variabili **L** di tipo **list** e **T** di tipo **tree** sono state create come lista ordinata e albero binario di ricerca.

```
#include <stdio.h>
list crea_lista(list);
tree crea_tree(tree);

int same(list l, tree t)
{ int trovato=0;
  tree aux=t;
  while (l!=NULL)
    { t=aux; trovato=0;
      while ((t!=NULL) && (!trovato))
         if (l->value==t->value) trovato=1;
         else if (l->value < t->value) t=t-> left;
              else t=t->right; }
      if (trovato) l=l->next;
      else l=NULL;
    }
   return trovato; }

void main(void)
{  list L=NULL;
   tree T=NULL;
   L=crea_lista(L);
   T=crea_tree(T);
   printf("%d", same(L,T));
 }
```

a) Si indichi cosa fa questo frammento di programma e la funzione **`same`** in particolare;

b) Se ne indichi la complessità in termini di numero di esecuzioni del test condizionale sottolineato, ipotizzando che da input siano stati inseriti N valori in lista e M valori nell'albero e che gli N valori in lista siano tutti presenti anche nell'albero T.

Il programma crea lista L e albero T, poi con la funzione **`same`** in particolare controlla che ogni elemento della lista L sia in T. Questa funzione scandisce sequenzialmente L e cerca ogni elemento di L in T applicando una ricerca binaria.

La complessità in termini di numero di esecuzioni del test condizionale sottolineato, ipotizzando che da input siano stati inseriti N valori in lista e M valori nell'albero e che gli N valori in lista siano tutti presenti anche nell'albero T:

- nel caso peggiore è di N*h dove h è l'altezza dell'albero (tutti gli elementi in lista sono uguali a alcune delle foglie di T; h vale $\log_2 M$ solo se T è perfettamente bilanciato; se completamente sbilanciato h vale M);
- nel caso migliore è N*1 (tutti gli elementi in lista sono uguali alla radice di T)