Algoritmi su liste

Lezione 5

1

Torre di Hanoi con 4 elementi: chiamate necessarie

```
hanoi(4, A, B, C)
    hanoi(3, A, C, B)
        hanoi(2, A, B, C)
            hanoi(1, A, C, B)
                    sposta(A, B)
            sposta(A, C)
            hanoi(1, B, A, C)
                sposta(B, C)
        sposta(A, B)
        hanoi(2, C, A, B)
            hanoi(1, C, B, A)
                    sposta(C, A)
            sposta(C, B)
            hanoi(1, A, C, B)
                sposta(A, B)
    sposta(A, C)
```

```
hanoi(3, B, A, C)
    hanoi(2, B, C, A)
            hanoi(1, B, A, C)
                    sposta(B, C)
            sposta(B, A)
            hanoi(1, C, B, A)
                    sposta(C, A)
    sposta(B, C)
    hanoi(2, A, B, C)
            hanoi(1, A, C, B)
                    sposta(A, B)
             sposta(A, C)
            hanoi(1, B, A, C)
                sposta(B, C)
```


2

programmi ricorsivi su liste

definizione di LISTA

- **NULL (sequenza vuota) è una LISTA**
- **un elemento seguito da una LISTA è una LISTA**

```
struct Elem {
        InfoType inf;
        Elem*    next;
};
```


programmi ricorsivi su liste

```
int length(Elem* p) {
   if (p == NULL) return 0;
return 1+length(p->next);
 }
```

```
int howMany(Elem* p, int x) {
 if (p == NULL) return 0;
 return (p->inf == x)+howMany(p->next, x);
 }
```

programmi ricorsivi su liste

```
int belongs(Elem *l, int x) {
   if (l == NULL) return 0;
   if (l->inf == x) return 1;
   return belongs(l->next, x);
 }

void tailDelete(Elem * & l) {
   if (l == NULL) return;
   if (l->next == NULL) {
       delete l;
       l=NULL;
   }
   else tailDelete(l->next);
 }
```


programmi ricorsivi su liste

```
void tailInsert(Elem* & l, int x) {
   if (l == NULL) {
       l=new Elem;
       l->inf=x;
       l->next=NULL;
   }
   else tailInsert(l->next,x);
 }
```

programmi ricorsivi su liste

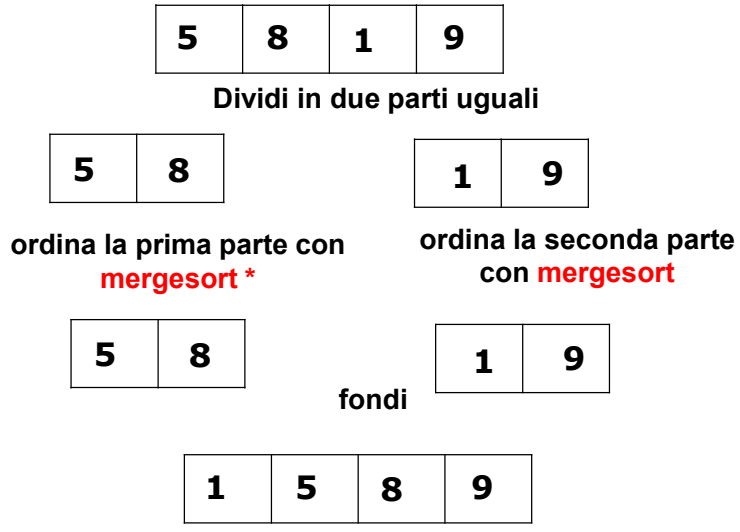
```
void append(Elem* & l1, Elem* l2) {
   if (l1 == NULL) l1=l2;
   else append(l1->next, l2);
 }


Elem* append(Elem* l1, Elem* l2) {
   if (l1 == NULL) return l2;
   l1->next=append( l1->next, l2 );
return l1;
 }
```


Mergesort

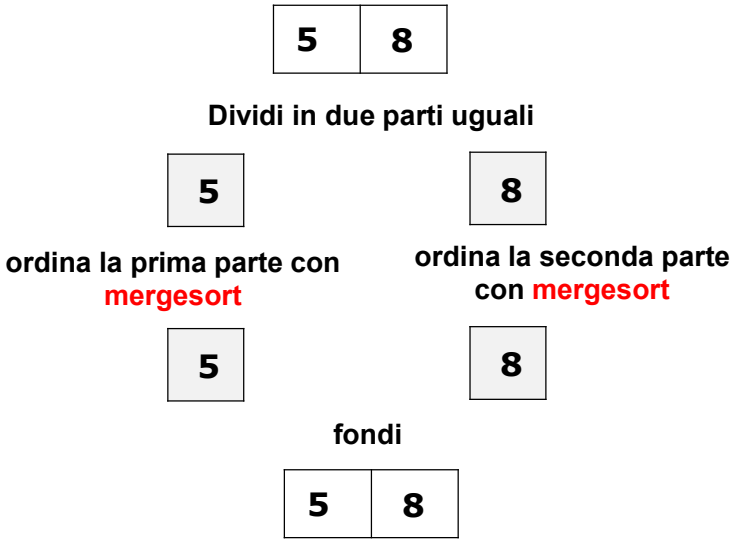
```
void mergeSort( sequenza S ) {
if ( |S|<= 1 )
        return;
else  {
    < dividi S in 2 sottosequenze S1 e S2 di uguale
                        lunghezza>;
     mergeSort(S1 );
     mergeSort(S2 );
    < fondi S1 e S2 >;
       }
 }
```

mergesort

mergesort *



mergesort *



Mergesort su liste

```
void mergeSort(Elem*& s1) {
        if (s1 == NULL || s1->next == NULL)  return;
        Elem* s2 = NULL;

        split (s1, s2);

        mergeSort (s1);

        mergeSort (s2);

        merge (s1, s2);
}
```

$T(0) = T(1) = d$

$T(n) = bn + 2T(n/2)$

$T(n) \in O(n \log n)$

Mergesort : split

```
void split (Elem* & s1, Elem* & s2) {
        if (s1 == NULL || s1->next == NULL)
            return;
        Elem* p = s1->next;
        s1-> next = p-> next;
        p-> next = s2;
        s2 = p;
        split (s1-> next, s2);
}
```

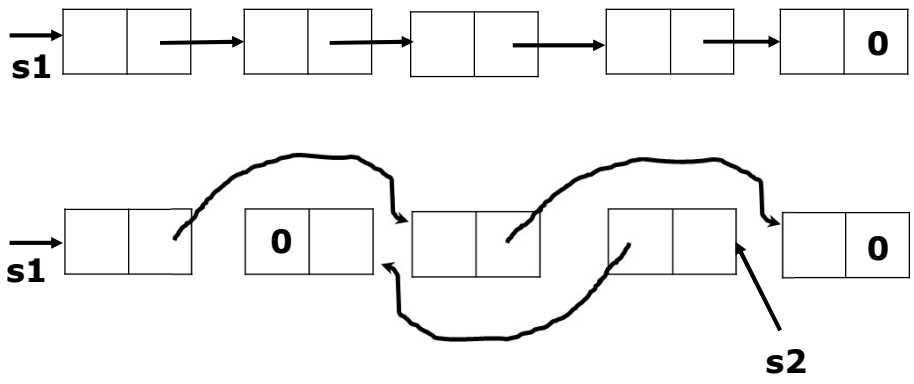
$T(0) = T(1) = d$

$T(n) = b + T(n-2)$

$T(n) \in O(n)$

Mergesort : split

Divide la lista in due liste mettendo gli elementi di posizione pari in una e quelli di posizione dispari nell'altra (rovesciati)



Mergesort : merge

```
void merge (Elem* & s1, Elem* s2) {
        if (s2 == NULL)
            return;
        if (s1 == NULL) {
            s1 = s2;
            return;
        }
        if (s1->inf <= s2->inf)
            merge (s1-> next, s2);

        else {
            merge (s2-> next, s1);
            s1 = s2;
        }
}
```

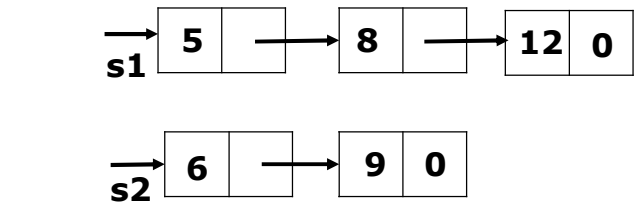
$T(0) = d$

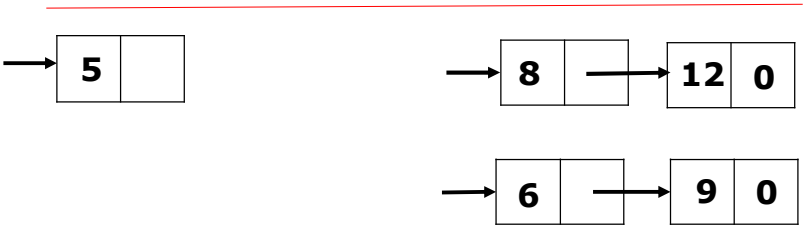
$T(n) = b + T(n-1)$

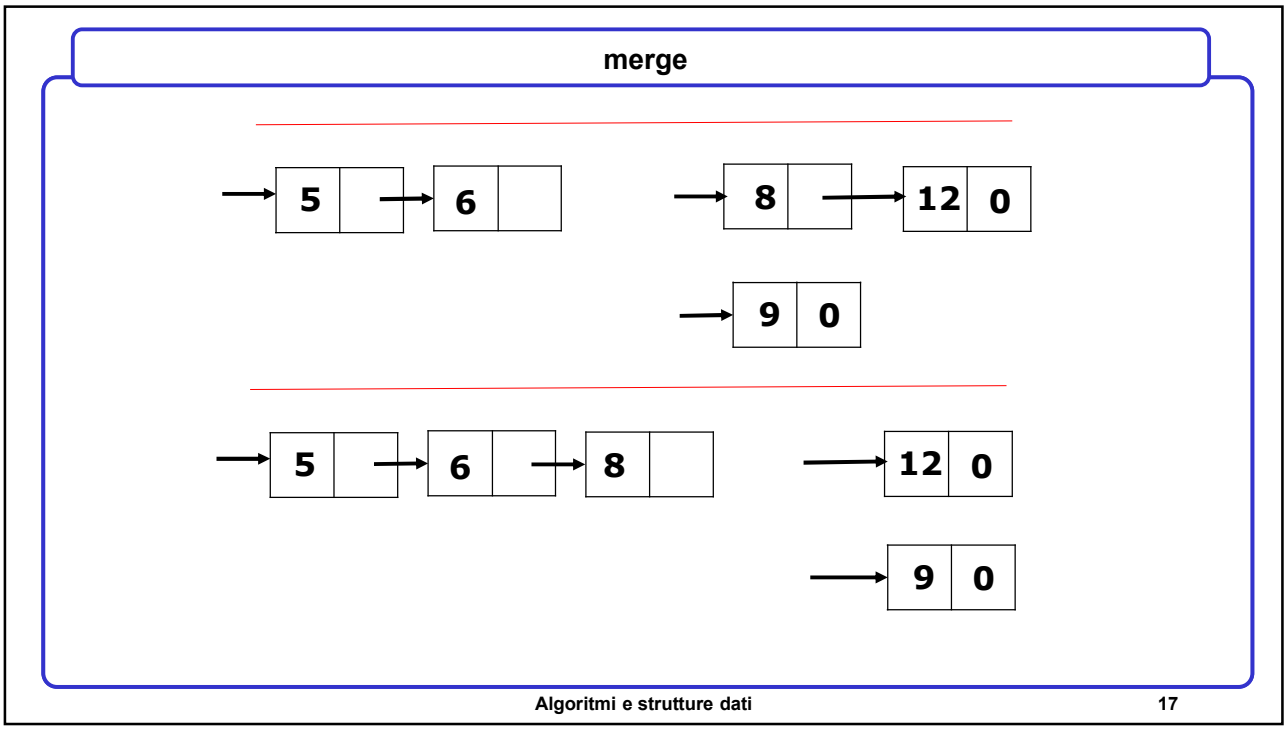
$T(n) \in O(n)$

Scorre in parallelo le due liste confrontando i loro primi elementi e scegliendo ogni volta il minore fra i due

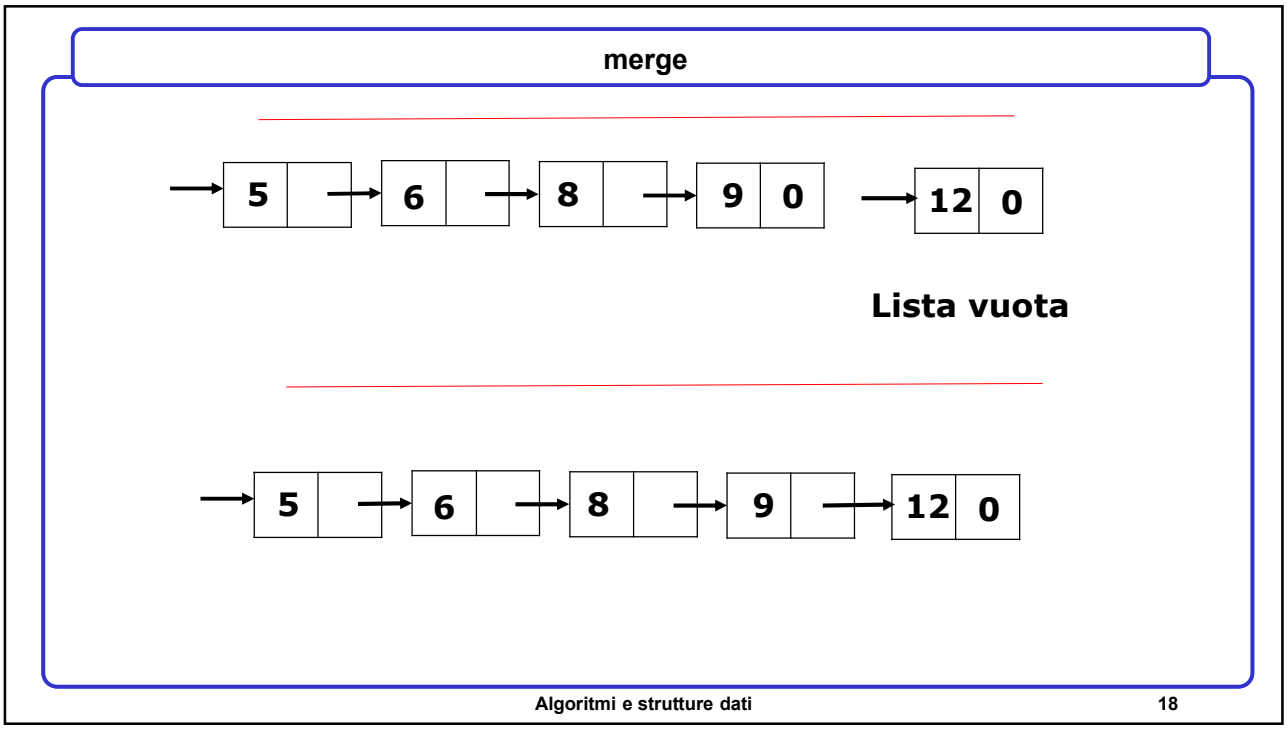
merge






merge
5
6
8
12
0
9
0
5
6
8
12
0
9
0
Algoritmi e strutture dati
17

17


merge
5
6
8
9
0
12
0
Lista vuota
5
6
8
9
12
0
Algoritmi e strutture dati
18

18

Esempio mergesort

```
mergeSort( [2,1,3,5] )
    dividi( [2,1,3,5] )
    mergeSort( [2,3] )
        dividi( [2,3] )
        mergeSort( [2] )        [2]
        mergeSort( [3] )        [3]
        fondi( [2] , [3] )      [2,3]
    mergeSort( [5,1] )
        dividi( [5,1] )
        mergeSort( [5] )        [5]
        mergeSort( [1] )        [1]
        fondi( [5] , [1] )      [1,5]
    fondi( [2,3], [1,5] )       [1,2,3,5]
```

19

Quale delle seguenti affermazioni è vera?
(a) log n = O(n)
(b) n = O(log n)
(c) log n = O(log log n)
(d) Nessuna delle precedenti è vera

Risposta Esatta:

Perchè è utile la notazione asintotica?
(a) Per ridurre il tempo di esecuzione dei nostri algoritmi
(b) Per calcolare anche l'occupazione di memoria oltre che il tempo di esecuzione
(c) Perchè ci consente di stimare gli ordini di grandezza delle funzioni che vogliamo calcolare, ignorando i fattori costanti ed altri dettagli ininfluenti
(d) Perchè le linee di codice di un programma non possono essere calcolate

Risposta Esatta:

20

Come misuriamo l'efficienza di un algoritmo?
a) In base al tempo necessario a scrivere un programma basato sull'algoritmo
b) In base al tempo necessario ad eseguire l'algoritmo su un'architettura di riferimento
c) In base al numero di linee di codice di un programma basato sull'algoritmo
d) In base alla quantità di risorse (tempo, spazio) richieste alla sua esecuzione

Risposta Esatta:

Quali sono gli algoritmi più efficienti, quelli ricorsivi o quelli iterativi?
a) Quelli ricorsivi
b) Quelli iterativi
c) Non può essere stabilito in maniera generale
d) Nessuno dei due: sono gli algoritmi scritti in C ad essere i più efficienti

Risposta Esatta:

Quale delle seguenti affermazioni è vera?
a) $\log n = \Omega(n)$
b) $n = \Omega(\log n)$
c) $\log n = O(\log \log n)$
d) Nessuna delle precedenti è vera

Risposta esatta:

Quanti confronti vengono eseguiti dalla ricerca binaria nel caso migliore?
(a) 1
(b) $O(\log \log n)$
(c) $O(\log n)$
(d) $O(n)$

Risposta esatta:

Quanti confronti esegue la ricerca sequenziale nel caso medio?
(a) 3n/4 se l'elemento è presente nell'insieme, n se l'elemento non è presente nell'insieme
(b) O(log n) se l'elemento è presente nell'insieme, n se l'elemento non è presente nell'insieme
(c) (n + 1)/2 se l'elemento è presente nell'insieme, n se l'elemento non è presente nell'insieme
(d) sempre n confronti, in ogni caso

Risposta esatta:

Quanti confronti vengono effettuati dall'algoritmo quickSort?
(a) O(n) nel caso medio
(b) $O(n^2)$ nel caso peggiore
(c) O(n log n) nel caso peggiore
(d) O(n log2 n) nel caso medio

Risposta esatta: