Corso di laurea in Informatica
Progettazione d'algoritmi

La tecnica del Divide et Impera

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Divide et Impera

Un programma sviluppato secondo questa tecnica è sostanzialmente diviso in tre parti:

- Divide: in questa parte si procede alla suddivisione dei problemi in problemi di dimensione minore;
- Impera: nella seconda parte i problemi vengono risolti in modo ricorsivo. Quando i sottoproblemi arrivano ad avere una dimensione sufficientemente piccola, essi vengono risolti direttamente tramite il caso base;
- Combina: l'ultima fase del paradigma prevede di ricombinare l'output ottenuto dalle precedenti chiamate ricorsive al fine di ottenere il risultato finale.

Quali sono i vantaggi offerti da questa tecnica?

- Di solito, quando la tecnica è applicabile, non è difficile vedere come applicarla. Questo perchè, tipicamente, è facile intuire come si potrebbe spezzare l'istanza.
- Inoltre, è abbastanza facile, una volta che un algoritmo di Divide et Impera è stato ideato, analizzarne sia la correttezza che la complessità.
- In particolare, la complessità di un algoritmo Divide et Impera, essendo un algoritmo ricorsivo, può essere determinata risolvendo un'opportuna relazione di ricorrenza relativa alla funzione $T(n)$ che dà il massimo tempo di calcolo dell'algoritmo su istanze di dimensione n.

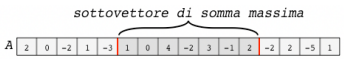
Esempio:

Data una lista di interi A, una *sottolista* è una sequenza consecutiva di valori della lista. Il *valore della sottolista* è dato dalla somma degli elementi che comprende.

Problema: Data una lista di interi vogliamo trovare il valore massimo delle sue sottoliste.

È chiaro che il problema è banale se i valori della lista sono tutti nonnegativi o tutti nonpositivi.

Il problema risulta interessante se la lista contiene valori sia positivi che negativi. Ecco un esempio:



Un algoritmo diretto per il problema, basato sulla ricerca esaustiva, consiste nel considerare le somme di tutte le possibili sottoliste e prenderne il massimo:

Implementazione 1:

```
def es2(A):
    # restituisce il valore massimo tra quello dei sottovettori di A
    val=A[0]
    for i in range(len(A)):
        for j in range(len(A)):
            val=max(sum(A[i:j+1]),val)
    return val
```

- L'algoritmo è molto inefficiente: la complessità è $O\left(n^3\right)$.
- Possiamo abbassare la complessità osservando che la somma di una sottolista $[i \ldots j]$ è uguale alla somma della sottolista $[i \ldots j-1]$ più $lista[j]$.

Implementazione 2:

```
def es2(A):
    # restituisce il valore massimo tra quello dei sottovettori di A
    val=0
    for i in range(len(A)):
        s=0
        for j in range(i,len(A)):
            s+=A[j]
            val=max(s,val)
    return val
```

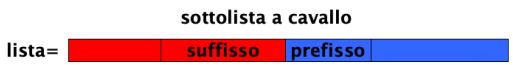
- L'algoritmo ha ora complessità $\Theta\left(n^2\right)$.

- La complessità ottenuta è $O\left(n^2\right)$.
- Possiamo fare di meglio????
- La sottoliste possibili sono $\frac{n^2}{2} = \Theta\left(n^2\right)$
- Ma è realmente necessario esaminarle tutte per risolvere il problema????

Proviamo ad applicare la tecnica del *Divide et Impera*:

- La prima cosa che viene in mente è di dividere a metà la lista, trovare il massimo in ciascuna delle due metà e poi calcolare il massimo delle somme delle sottoliste a cavallo delle due metà. Alla fine restituiamo il massimo dei tre massimi ottenuti.
- Sicuramente l'algoritmo è corretto perché considera tutte le possibili sottoliste. Infatti, una sottolista o è contenuta interamente nella prima metà della lista o interamente nella seconda o è cavallo delle due metà.
- qual'è la sua complessità di questo algoritmo? La risposta dipende da come effettuiamo il calcolo delle somme delle sottoliste a cavallo. Se lo facciamo in modo diretto cioè, calcoliamo la somma della sottolista $[i \dots j]$ per ogni coppia di indici i e j con i nella prima metà e j nella seconda, questo richiede tempo $\Theta(n^2)$ (dato che il numero di sottoliste a cavallo è $\frac{n}{2}\frac{n}{2} = \Theta(n^2)$).
  Ovviamente questo implica che la complessità $T(n)$ del nostro algoritmo Divide et Impera è determinata dalla relazione
  $$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^2)$$
- che risolta dà $T(n) = O(n^2)$

- Possiamo migliorare il calcolo del massimo delle somme delle sottoliste a cavallo delle due metà?
- Osserviamo che ogni sottolista a cavallo è formato da un $suffisso$ della prima metà della lista ed un $prefisso$ della seconda metà della lista



- La sottolista a cavallo di somma massima ha come valore il valore del $suffisso$ di somma massima della prima metà della lista più il valore del $prefisso$ di somma massima della seconda metà della lista

Implementazione 3:

```
def es2(A):
    "restituisce il valore massimo tra quelli dei suoi sottovettori"
    if len(A)==1: return A[0]
    m=len(A)//2
    vs=es2(A[:m])
    vd=es2(A[m:])
    #calcolo il prefisso massimo per la stringa a cavallo
    pm=s=A[m-1]
    for i in range(m-2,-1,-1):
        s+=A[i]
        pm=max(s,pm)
    #calcolo il suffisso massimo per la stringa a cavallo
    sm=s=A[m]
    for i in range(m+1,len(A)):
        s+=A[i]
        sm=max(s,sm)
    return max(vs,vd,pm+sm)

 >>> A=[3, -5, 6, -5,8,-3,-4,5]
>>> es2(A)
9
>>> A=[2,0,-2,1,-3,1,0,4,-2,3,-1,2,-2-2,-5,1]
>>> es2(A)
7
```

- il calcolo di $A[:m]$ e di $A[m:]$ costa $\Theta(n)$
- il calcolo del valore del suffisso massimo della prima metà di $lista$ costa $\Theta(n)$
- il calcolo del valore del prefisso massimo della seconda metà di $lista$ costa $\Theta(n)$
- la ricorrenza che cattura la complessità $T(n)$ della funzione è
  $$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
  che risolta dà $\Theta(n \log n)$

Abbiamo ora un algoritmo di complessità $\Theta(n \log n)$.
Si può fare di meglio????

- La risposta è SI!. Faremo ora vedere una migliore implementazione dell'algoritmo basato sul *divide et impera* che presenterà una complessità $\Theta(n)$.

- Nota che l'algoritmo $\Theta(n)$ sarà ottimo per il problema (il lower bound $\Omega(n)$ segue dalla banale osservazione che per risolvere il problema non si può fare a meno di scorrere tutti gli n elementi della lista).

1. Non possiamo permetterci di pagare $\Theta(n)$ in fase di invocazione della funzione sui due sottoproblemi (per la creazione di $A[: m]$ e $A[m :]$)

2. Non possiamo permetterci di pagare $\Theta(n)$ in fase di combinazione delle risposte fornite dai due sottoproblemi.

Per risolvere il problema del costo $\Theta(n)$ in fase di combinazione dei due sottoproblemi facciamo in modo che la procedura ricorsiva oltre a restituire la soluzione *solu* al problema restituisca anche nell'ordine questi altri tre valori (che semplificheranno poi il costo della combinazione):

- $pref$ il valore massimo per i prefissi di A
- $suff$ il valore massimo per i suffissi di A
- tot la somma di tutti i valori di A

- Facciamo in modo che la procedura ricorsiva per A oltre a restituire la soluzione *solu* al problema restituisca anche questi altri tre valori (che semplificheranno poi il costo della combinazione):
  - $pref$ il valore massimo per i prefissi di A
  - $suff$ il valore massimo per i suffissi di A
  - tot il valore dell'intera A
- dai valori $prefS$, $suffS$, $totS$ restituiti dal primo sottoproblema e dai valori $prefD$, $suffD$, $totD$ restituiti dal secondo problema è possibile calcolare in $O(1)$:
  - **il valore ottimo della sottolista a cavallo (è $suffS + prefD$)**
  - i valori di $IntS$, $intD$ e int da restituire per l'intera lista:
    * $pref = \max(prefS,\ totS + prefD)$
    * $suff = \max(suffD,\ totD + suffS)$
    * $tot = totS + totD$

$A =$ 

implementazione:

```
def es2(A):
    '''restituisce il valore massimo tra quelli dei suoi sottovettori,
    il prefisso massimo, il suffisso massimo, e il suo valore'''
    if len(A)==1: return (A[0],A[0],A[0],A[0])
    m=len(A)//2
    soluS, prefS, suffS, totS = es2(A[:m])
    soluD, prefD, suffD, totD = es2(A[m:])
    return max(soluS,soluD,suffS+prefD), max(prefS,totS+prefD), max(suffD,totD+suffS), totS+totD

 >>> A=[3, -5, 6, -5,8,-3,-4,5]
>>> es2(A)
(9, 7, 7, 5)
>>> A=[2,0,-2,1,-3,1,0,4,-2,3,-1,2,-2-2,-5,1]
>>> es2(A)
(7, 1, 5, -3)
```

La complessità asintotica di questa nuova implementazione è ancora $\Theta(n \log n)$ a causa del costo $\Theta(n)$ che paghiamo per la creazione delle istanze dei due sottoproblemi.
Questo problema è facilmente risolvibile utilizzando due indici i e j che ci dicono dove inizia e dove finisce il sottovettore di A su cui ricorsivamente invochiamo la funzione.

implementazione:

```
def es2(A,i,j):
    '''restituisce il valore massimo tra quelli dei suoi sottovettori,
    il prefisso massimo, il suffisso massimo, e il suo valore'''
    if i==j: return (A[i],A[i],A[i],A[i])
    m=(i+j)//2
    soluS, prefS, suffS, totS = es2(A,i,m)
    soluD, prefD, suffD, totD = es2(A,m+1,j)
    return max(soluS,soluD,suffS+prefD), max(prefS,totS+prefD), max(suffD,totD+suffS), totS+totD


>>> A=[3, -5, 6, -5,8,-3,-4,5]
>>> es2(A,0,len(A)-1)
(9, 7, 7, 5)
>>> A=[2,0,-2,1,-3,1,0,4,-2,3,-1,2,-2-2,-5,1]
>>> es2(A,0,len(A)-1)
(7, 5, 1, -3)
```

- La relazione di ricorrenza per il tempo di calcolo di questa implementazione è
$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$
che risolta dà $\Theta(n)$.

ESERCIZIO: Progettare un algoritmo che, data una lista ordinata A di n interi ed un intero x, restituisce il numero di occorrenze di x nella lista. La complessità dell'algoritmo deve essere $O(\log n)$

Ad esempio: se $A = [3, 3, 3, 3, 5, 5, 6, 6, 6, 6, 6, 8, 9]$
- per $x = 3$ la risposta è 4
- per $x = 4$ la risposta è 0
- per $x = 8$ la risposta è 1

Un possibile algoritmo è il seguente :

def *conta_occorrenze(lista, x) :*
  a = *la posizione della prima occorrenza di x in A*
    (−1 se x non e nella lista)
  if $a == -1$: **return** 0
  b = *la posizione dell'ultima occorrenza di x in A*
  return $b - a + 1$

Nota: per garantire un tempo logaritmico bisognerà ricercare la prima e l'ultima occorrenza tramite ricerca binaria.

Implementazione:

```
def conta_occorrenze(A,x):
    a= trova_primo(A,0,len(A)-1,x)
    if a==-1: return 0
    b= trova_ultimo(A,0,len(A)-1,x)
    return b-a+1
```

```
def trova_primo(A,i,j,x):
    if i>j: return -1
    m=(i+j)//2
    if A[m]==x and (m==0 or A[m-1]!=x): return m
    if A[m]>=x: return trova_primo(A,i,m-1,x)
    return trova_primo(A,m+1,j, x)
```

```
def trova_ultimo(A,i,j,x):
    if i>j: return -1
    m=(i+j)//2
    if A[m]==x and (m==len(A)-1 or A[m+1]!=x): return m
    if A[m]> x: return trova_ultimo(A,i,m-1,x)
    return trova_ultimo(A,m+1,j, x)
```

Complessità:
- le due procedure *trova_primo* e *trova_ultimo* hanno complessità $O(\log n)$ che è di conseguenza anche il costo dell'algoritmo.

ESERCIZIO: Dati due interi a ed n progettare un algoritmo che calcoli a^n in tempo $O(\log n)$ (le uniche operazioni aritmetiche permesse sono $+, *$ e //)

- un semplice algoritmo che calcola a^n utilizzando $n-1$ prodotti è il seguente:

```
def pow1(a,n):
    s=1
    for _ in range(n):
        s*= a
    return s
```

- Per applicare il *divide et impera* osserva che:

$$a^n = a^{\lfloor \frac{n}{2} \rfloor} \cdot a^{\lceil \frac{n}{2} \rceil}$$

Ad esempio: $a^{15} = a^7 \cdot a^8$ e $a^{16} = a^8 \cdot a^8$

- inoltre, $a^{\lceil \frac{n}{2} \rceil} = \begin{cases} a^{\lfloor \frac{n}{2} \rfloor} & \text{se } n \text{ è pari} \\ a^{\lfloor \frac{n}{2} \rfloor} \cdot a & \text{se } n \text{ è dispari} \end{cases}$

- quindi, $a^n = \begin{cases} a^{\lfloor \frac{n}{2} \rfloor} \cdot a^{\lfloor \frac{n}{2} \rfloor} & \text{se } n \text{ è pari} \\ a^{\lfloor \frac{n}{2} \rfloor} \cdot a^{\lfloor \frac{n}{2} \rfloor} \cdot a & \text{se } n \text{ è dispari} \end{cases}$

- Il sottoproblema da risolvere è dunque $a^{\lfloor \frac{n}{2} \rfloor}$

- $a^n = \begin{cases} a^{\lfloor \frac{n}{2} \rfloor} \cdot a^{\lfloor \frac{n}{2} \rfloor} & \text{se } n \text{ è pari} \\ a^{\lfloor \frac{n}{2} \rfloor} \cdot a^{\lfloor \frac{n}{2} \rfloor} \cdot a & \text{se } n \text{ è dispari} \end{cases}$

```
def pow2(a,n):
    if n==0: return 1
    x= pow2(a,n//2)
    if n%2 :
        #n è dispari
        return x*x*a
    # n è pari
    return x*x
```

- la ricorrenza da studiare per la complessità dell'algoritmo (e per il numero di prodotti richiesti) è

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

che risolta dà $\Theta(\log n)$

ESERCIZIO: Data una stringa binaria S di n bit progettare un algoritmo che calcoli il numero di sottosequenze di S che cominciano con 0 terminano con 1.

Ad esempio:
per $S =' 0011'$ restituisce 4 (le sottosequenze sono $001\_$, 0011, $\_01\_\_$, $\_011$)
per $S =' 0101'$ restituisce 3 (le sottosequenze sono 01__, 0101, __01)
per $S =' 1100'$ restituisce 0

Progettare due algoritmi basati sulla tecnica del Divide et Impera, uno a complessità $\Theta(n \log n)$ e l'altro a complessità $\Theta(n)$

IDEA:

- divido la stringa in due sottostringhe di lunghezza circa $\frac{n}{2}$
- risolvo il problema in ciascuna delle due sottostringe ricevendo come soluzione n_1 e n_2
- restituisco il valore $n_1 + n_2 + n_3$ dove n_3 è il numero di sottostringe che iniziano nella sottostringa di destra e terminano nella sottostringa di sinistra.
  - per calcolare n_3 basta contare il numero $zeri$ degli zeri nella sottostringa di sinistra ed il numero uni della sottostringa di destra (vale infatti $n_3 = zeri * uni$)

IMPLEMENTAZIONE:

```
def es1(S):
    '''conta le sottostringhe che cominciano con 0 e finiscono
    con 1 nella stringa binaria A'''
    if len(S)==1: return 0
    m=len(S)//2
    ts=es1(S[:m])
    td=es1(S[m:])
    return ts+td + S[:m].count('0')*S[m:].count('1')
```

- La relazione di ricorrenza per il tempo di calcolo di questa implementazione è
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
che risolta dà $\Theta(n \log n)$.

IDEA2: facciamo in modo che la funzione ricorsiva sulla stringa S oltre a restituire il numero tot delle sottosequenze che cominciano con zero e finiscono con uno restituisca anche il numero z dei suoi zeri e il numero u dei suoi uni. Con queste informazioni da parte delle due chiamate ricorsive la funzione è in grado di calcolare in $O(1)$ il numero di sottosequenze che cominciano nel sottoproblema di sinistra e finiscono nel sottoproblema di destra.
Sapendo infatti zs (il numero di zeri della sottostringa di sinistra) e ud (il numero di zeri della sottostringa di destra) il numero di sottosequenze che iniziano alla sinistra del taglio e terminano alla destra del taglio sono $zs * ud$.

```
def es1(S):
    '''conta le sottostringhe che cominciano con 0 e
    finiscono con 1 nella stringa binaria A'''
    if len(S)==1:
        if S[0]=='0': return (1,0,0)
        else: return (0,1,0)
    m=len(S)//2
    zs,us,tots= es1(S[:m])
    zd,ud,totd=es1(S[m:])
    return zs+zd,us+ud,tots+totd+zs*ud
```

- La relazione di ricorrenza per il tempo di calcolo di questa implementazione è ancora
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
a causa del costo $\Theta(n)$ necessario per la creazione delle due istanze da invocare.

Il problema è facilmente risolvibile specificando i sottoinsiemi da passare tramite indici di inizio e fine dei sottoarray.

IMPLEMENTAZIONE:

```
def es1(A,i,j):
    "conta le sottostringhe che cominciano con 0 e finiscono con 1 nella stringa binaria A"
    if i==j:
        if A[i]=='0': return (1,0,0)
        else: return (0,1,0)
    m=(i+j)//2
    zs,us,tots= es1(A,i,m)
    zd,ud,totd=es1(A,m+1,j)
    return zs+zd,us+ud,tots+totd+zs*ud

>>> A='0011'
>>> es1(A,0,len(A)-1)
(2, 2, 4)
>>> A='0101'
>>> es1(A,0,len(A)-1)
(2, 2, 3)
>>> A='1100'
>>> es1(A,0,len(A)-1)
(2, 2, 0)
```

- La relazione di ricorrenza per il tempo di calcolo di questa implementazione è
$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$
che risolta dà $\Theta(n)$.