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.

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

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

```
def pow(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 pow(a, n):
    if n == 0:
        return 1
    x = pow(a, n//2)
    if n % 2:
        return x*x*a
    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 $\Theta(\log n)$

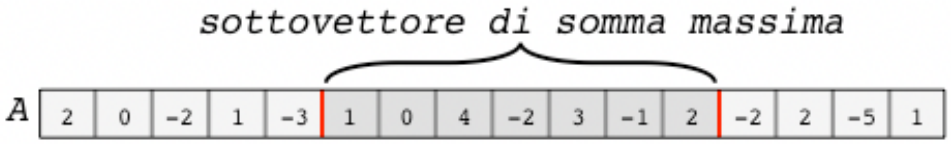
Esempio:

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 es(A):
    '''restituisce il valore massimo tra quello dei sottovettori di A'''
    n = len(A)
    val = A[0]
    for i in range(n):
        for j in range(i,n):
            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 $[i...j-1]$ più $lista[j]$.

Implementazione 2:

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

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

- La complessità ottenuta è $O(n^2)$
- Possiamo fare di meglio ??????
- Le sottoliste possibili sono:

$$\frac{n(n-1)}{2} + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

- 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.

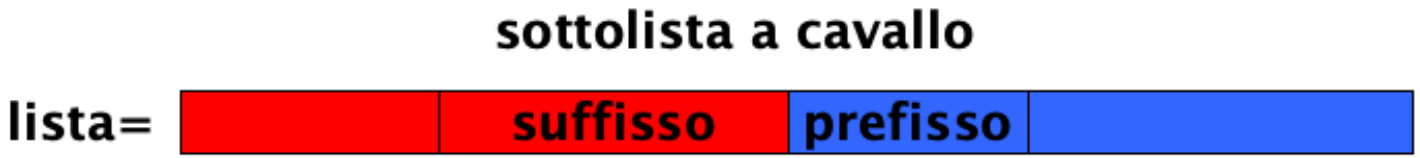
Qual'è la sua complessità di questo algoritmo? La risposta dipende da come effettuiamo il calcolo delle somme delle sottoliste a cavallo. Se lo facciamo in modo diretto cioè, calcoliamo la somma della sottolista $[i \ldots j]$ per ogni coppia di indici i e j con i nella prima metà e j nella seconda, 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 $T(n) = \Theta(n^2)$

- 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 es(A):
    '''restituisce il valore massimo tra quello dei sottovettori di A'''
    if len(A) == 1: return A[0]
    m = len(A)//2
    rs = es(A[:m])
    rd= es(A[m:])
    pmax = x =  A[m-1]
    for i in range(m-2,-1,-1):
        x += A[i]
        pmax = max(pmax, x)
    smax = x = A[m]
    for i in range(m+1, len(A)):
        x += A[i]
        smax = max(smax, x)
    return max(rs, rd, smax + pmax)


>>> A=[3, -5, 6, -5, 8, -3, -4, 5]
>>> es(A)
>>> 9
>>> A=[2, 0, -2, 1, -3, 1, 0, 4, -2, 3, -1, 2, -2, -2, -5, 1]
>>> es(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 A costa $\Theta(n)$
- il calcolo del valore del prefisso massimo della seconda metà di A 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 à $\Theta(n \log n)$

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 $A[m :]$) e per ovviare a questo problema indicheremo i sottoproblemi di A con il loro punto di inizio i ed il loro punto di fine j.

2. Non possiamo permetterci di pagare $\Theta(n)$ in fase di combinazione delle risposte fornite dai due sottoproblemi. Per ovviare a questo problema faremo si che ogni sottoproblema restituisca ulteriore informazione che aiuti nella combinazione.

Facciamo in modo che la procedura ricorsiva per A con la soluzione *sol* al problema restituisca anche altri tre valori (che semplificheranno poi il costo della combinazione):

- *tot:* la somma dei valori di A
- *pm:* il valore massimo per i prefissi di A
- *sm:* il valore massimo per i suffissi di A

dai valori *solS, totS pmS* e *smS* restituiti dal primo sottoproblema e dai valori *solD, totD pmD* e *smD* restituiti dal secondo problema è possibile calcolare in $\Theta(1)$:

- $sol = max\{solS,\ solD,\ smS + pmD\}$
- $tot = totS + totD$
- $pm = \max\{pmS,\ solS + pmD\}$
- $sm = \max\{smD,\ smS + totD\}$
-

$A =$ 

implementazione:

```
def es(A, i, j ):
    '''per A[i:j] restituisce:
    sol = il valore della soluzione,
    tot = il totale dei suoi elementi
    pm = il valore massimo per i prefissi
    sm = il valore massimo per i  suffiss'''
    if i==j:
        return A[i], A[i], A[i], A[i]
    m= (i+j)//2
    sols, tots, pms, sms = es(A, i, m)
    sold, totd, pmd, smd = es(A, m+1, j)
    sol = max(sols, sold, sms + pmd)
    tot = tots + totd
    pm = max(pms, tots + pmd)
    sm = max(smd, totd + sms)
    return sol, tot, pm, sm

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

>>> A=[2, 0, -2, 1, -3, 1, 0, 4, -2, 3, -1, 2, -2, -2, -5, 1]
>>> es(A, 0, len(A)-1)
(7, -3, 5, 1)
```

La relazione di ricorrenza per il tempo di calcolo di questa implementazione è:

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$ che risolta dà $\Theta(n)$.

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

Ad esempio:

- per $S = '0011'$ restituisce 4 (infatti le sottostringhe sono $001\_$, 0011, $\_01\_\_$, $\_011$)
- per $S = '0101'$ restituisce 3 (infatti le sottostringhe 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 sottostringhe ricevendo come soluzione ts e td

· restituisco il valore $ts + td + c$ dove c è il numero di sottostringhe che iniziano nella sottostringa a destra e terminano nella sottostringa a sinistra.

  · per calcolare c basta contare il numero z degli zeri nella sottostringa di sinistra ed il numero u degli uni nella sottostringa a destra (vale infatti $c = z * u$)

IMPLEMENTAZIONE:

```
def es(S):
    '''conta le sottostringhe di S che cominciano con 0 e terminano con 1'''
    if len(S) == 1:
        return 0
    m = len(S)//2
    ts = es(S[:m])
    td = es(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 t 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 $\Theta(1)$ il numero di sottostringhe che cominciano nella sottostringa di sinistra e finiscono nella sottostringa di destra.
Sapendo infatti zs (il numero di zeri della sottostringa di sinistra) e ud (il numero di uni della sottostringa di destra) il numero di sottosequenze che iniziano alla sinistra del taglio e terminano alla destra del taglio sono $zs * ud$.

```
def es(S):
    '''conta le sottostringhe di S che cominciano con 0 e terminano con 1'''
    if len(S) == 1:
        if S[0]=='0': return (1,0,0)
        else: return (0,1,0)
    m = len(S)//2
    zs, us, ts = es(S[:m])
    zd, ud, td = es(S[m:])
    return zs + zd, us + ud, ts + td + 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 le sottoliste da passare tramite indici di inizio e fine.

IMPLEMENTAZIONE:

```
def es(S, i, j):
    '''conta le sottostringhe di S che cominciano con 0 e terminano con 1'''
    if i == j:
        if S[i]=='0': return (1,0,0)
        else: return (0,1,0)
    m = (i+j)//2
    zs, us, ts = es(S, i, m)
    zd, ud, td = es(S, m+1, j)
    return zs + zd, us + ud, ts + td + zs * ud

>>> S = '0011'
>>> es(S, 0, len(S)-1)
(2, 2, 4)
>>> S = '0101'
>>> es(S, 0, len(S)-1)
(2, 2, 3)
>>> S = '1100'
>>> es(S, 0, len(S)-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 $\Theta(n)$.