DIVIDE ET IMPERA

(divide and conquer)

- Un tecnica algoritmica molto naturale. Si basa sull'idea che un problema possa essere suddiviso in SOTTOPROBLEMI e la soluzione viene ottenuta COMBINANDO le soluzioni dei sottoproblemi

Abbiamo visto già qualche esempio

- **Algoritmo di Strassen** per moltiplicare matrici $n \times n$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

A_{ij} B_{ij} C_{ij} matrici $\frac{n}{2} \times \frac{n}{2}$

è possibile ottenere i valori di C_{ij} facendo solo 7 prodotti di matrici $\frac{n}{2} \times \frac{n}{2}$

Ricorrenza
$$\underbrace{T(n)}_{\text{costo } n \times n} := 7 \cdot \underbrace{T\left(\frac{n}{2}\right)}_{\text{costo } \frac{n}{2} \times \frac{n}{2}} + \underbrace{\Theta(n^2)}_{\text{tempo per ricomporre le sotto soluzioni}}$$

da cui:
$$T(n) = \Theta(n^{\log 7}) \ll n^3$$
n^3 ← Algoritmo banale

- **Mergesort**: ordinamento di un array di n elementi

| 1 | 9 | 7 | 2 | -2 | 4 | 1 | 5 |

| 1 | 9 | 7 | 2 | | -2 | 4 | 1 | 5 |

Mergesort n/2 elementi ↓   ↓ Mergesort n/2 elementi

| 1 | 2 | 7 | 9 | | -2 | 1 | 4 | 5 |

Merge

| -2 | 1 | 1 | 2 | 4 | 5 | 7 | 9 |

Ricorrenza $T(n) = 2T(n/2) + \Theta(n)$

da cui $T(n) = \Theta(n \log n)$

Altri algoritmi basati su divide et impera che conoscete già?

Metodologia Un algoritmo che risolve un problema utilizzando questa tecnica normalmente si divide in 3 fasi.

- **Divide**: il problema è SUDDIVISO in sottoproblemi più piccoli
- **Impera**: i sottoproblemi sono risolti. Se sono molto piccoli sono risolti direttamente, altrimenti questi vengono sottoposti alla stessa procedura di Divide et Impera, solitamente in modo RICORSIVO
- **Combina**: le soluzioni dei sottoproblemi sono RICOMBINATE per produrre la soluzione del problema principale

Perché usare questa tecnica: Quando è utilizzabile, è relativamente semplice capire come usare le soluzioni di problemi parziali per ottenere la soluzione totale, invece di costruire la soluzione totale direttamente

Tipicamente la COMPLESSITÀ dell'algoritmo può essere analizzata tramite equazioni di RICORRENZA che sono facilmente associabili ad uno schema RICORSIVO.

Caso di Studio: Sottovettore di Somma Massima

- Dato un array di numeri trovare l'intervallo dell'array la cui somma sia massima.

BANALE NEL CASO: tutti valori ≥ 0 → tutto l'array
tutti valori ≤ 0 → intervallo vuoto

Ad esempio:

2	0	-2	1	-3	1	0	4	-2	3	-1	2	-2	2	-5	1

somma massima

Input del problema

Un array A di lunghezza n, contenente NUMERI

Output Una coppia (i,j) che indica che l'intervallo dalla posizione i alla posizione $j-1$ è quello massimo

estremo sx incluso
estremo dx escluso

```python
def sottovettore1(A):
    n = len(A)
    risultato=(0,0)
    somma=0
    for i in range(n):
        for j in range(i+1,):
            t = 0
            for k in range(i,j):
                t += A[k]
            if t>somma:
                risultato = (i,j)
                somma = t
    return risultato
```

n+1

Ricerca Esaustiva

- Prova tutti gli intervalli (i,j) con $i<j$
- Per ognuno calcola la somma dell'intervallo

Complessità $\Theta(n^3)$

Q: Cosa fa l'algoritmo per un array tutto negativo?

Q: Quanti intervalli esplora l'algoritmo

Tentativo 2: vediamo come evitare di ricalcolare la somma

- Osservate che nel secondo ciclo annidato si esplorano gli intervalli $(i,i+1)$ $(i,i+2)$ $(i,i+3)$, ... (i,j) ... (i,n)

- Calcolata la somma dell'intervallo $(i,j-1)$ la somma di (i,j) è

$$\text{SOMMA}(i,j) = \text{SOMMA}(i,j-1) + A[j-1]$$

Quindi possiamo evitare il terzo ciclo interno

```python
def sottovettore2(A):
    n = len(A)
    risultato=(0,0)
    somma=0
    for i in range(n):
        t = 0
        for j in range(i+1,n+1):
            t += A[j-1]
            if t>somma:
                risultato = (i,j)
                somma = t
    return risultato
```

- Per ogni intervallo (i,j) l'algoritmo fa $O(1)$ operazioni

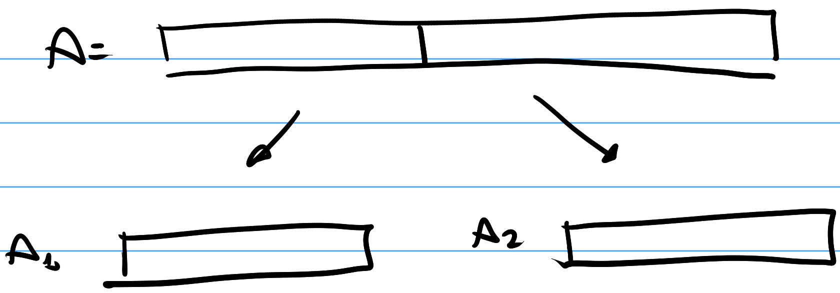
Complessità $\Theta(n^2)$

Entrambi i tentativi tentano tutti i possibili intervalli

(i,j) con $0 \leq i < j \leq n$ $\rightarrow$ $\binom{n+1}{2} = \frac{(n+1)\cdot n}{2} \approx \frac{n^2}{2}$

È NECESSARIO ESPLORARLI TUTTI?

Tentativo banale di divide et impera



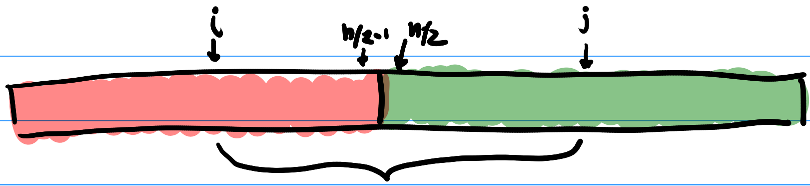
- trovo l'intervallo di somma massima (i_1, j_1) in A_1
- trovo l'intervallo di somma massima (i_2, j_2) in A_2
- MANCANO GLI INTERVALLI CON i in $[0, -n/2)$ e j in $[n/2 - n)$ che sono $\frac{n}{2} \times \frac{n}{2}$ e che devono essere controllati

È semplice quindi descrivere un algoritmo ricorsivo con ricorrenza

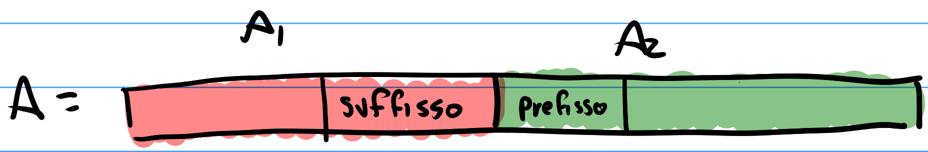
$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2)$$

che porta ad una soluzione di complessità $T(n) = \Theta(n^2)$

CHIARAMENTE ABBIAMO BISONGNO DI UN'IDEA PER GESTIRE GLI INTERVALLI A CAVALLO DELLE DUE METÀ



- Ad esempio (come in figura) l'intervallo ottimo potrebbe essere (i,j) tuttavia non è detto che $(i, \frac{n}{2})$ e $(n/2, j)$ siano gli intervalli ottimi delle due sottoliste.



Dunque una soluzione ottima per A può essere uno di tre casi

- una sol. ottima per A_1
- una sol. ottima per A_2
- una soluzione "a cavallo" costituita

suffisso di somma massima in A_1 + prefisso di somma massima in A_2

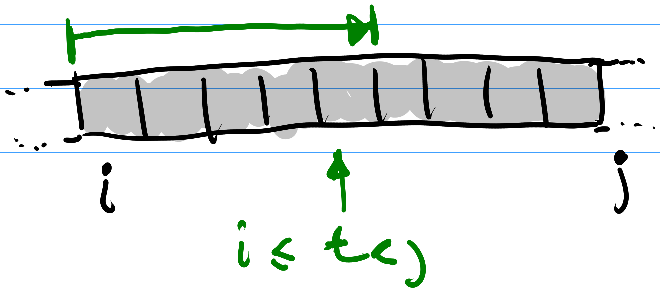
?: perché di somma massima? È possibile che una coppia suffisso+prefisso non massimi diano luogo ad una soluzione "a cavallo" migliore?

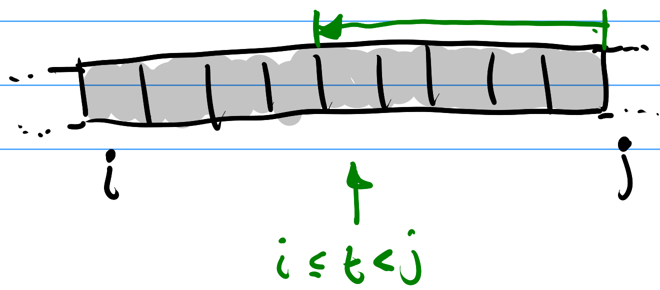
Definiamo: prefisso$(A, i, j) \to (t, v)$ tale che (i, t) sia il prefisso di somma massima v.

suffisso$(A, i, j) \to (t, v)$ tale che (t, j) sia il suffisso di somma massima v

```python
def prefisso(A,i,j):
    end, best = i,0
    v =0
    for t in range(i,j):
        v += A[t]
        if v>best:
            best = v
            end  = t+1
    return end,best
```

```python
def suffisso(A,i,j):
    start, best = j,0
    v =0
    for t in range(j-1,i-1,-1):
        v += A[t]
        if v>best:
            best  = v
            start = t
    return start,best
```





- Prefisso e Suffisso hanno complessità $O(j-i)$
- L'algoritmo ricorsivo, dato $(A, start, end)$ produce la soluzione (v, i, j) dove $A[i], \ldots, A[j-1]$ ha somma v e $start \leq i \leq j \leq end$
- La soluzione del problema si calcola su $(A, 0, n)$

```python
def sottovettore3(A,start=0,end=None):
    if end is None:
        end=len(A)

    # casi base
    if start==end:
        return (0,start,end)
    elif start+1==end and A[start]<0:
        return (0,start,start)
    elif start+1==end and A[start]>=0:
        return (A[start],start,end)

    # Dividi
    mid = (end+start) // 2

    # soluzioni ottime nelle due metà
    v1,i1,j1 = sottovettore3(A,start,mid)
    v2,i2,j2 = sottovettore3(A,mid,end)

    # soluzione a cavallo
    left, vl = suffisso(A,start,mid)
    right,vr = prefisso(A,mid,end)

    # Combina
    return max([(v1,i1,j1),
                (v2,i2,j2),
                (vl+vr,left,right)])
```

- start e end predefiniti a 0 e len(A)

Base della ricorsione: lunghezze 0 e 1

→ Divide

Chiamate ricorsive

→ Combina le soluzioni

Complessità $\Theta(n)$

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

$$\hookrightarrow T(n) = \Theta(n \log n)$$

Teorema principale (Master Theorem)

- Serve a risolvere alcuni tipi di ricorrenze semplici della forma

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

allora

1. $f(n) = O(n^c)$ con $c < \log_b a \rightarrow T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^c)$ con $c = \log_b a \rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log n)$
3. $f(n) = \Omega(n^c)$ con $c > \log_b a$ ed esiste $c<1$, $c\,f(n) > a \cdot f\left(\frac{n}{b}\right)$ $\rightarrow T(n) = \Theta(f(n))$

Esempi

Mergesort (caso 2) $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \rightarrow T(n) = \Theta(n \log n)$

Ricerca binaria (caso 2) $T(n) = T\left(\frac{n}{2}\right) + \Theta(1) \rightarrow T(n) = \Theta(\log n)$

Tentativo a pag 5 (caso 3) $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^2) \rightarrow T(n) = \Theta(n^2)$

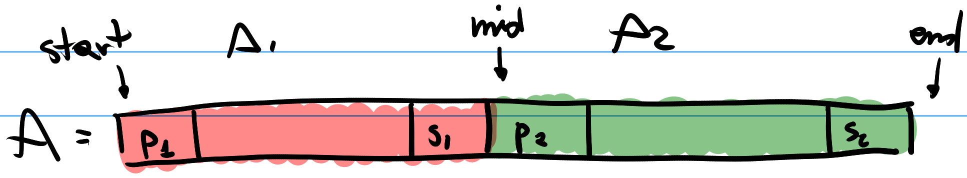
Strassen (caso 1) $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) \rightarrow T(n) = \Theta(n^{\log_2 7})$

Challenge: come ottenere un algoritmo $O(n)$?

L'algoritmo $O(n)$ sarebbe OTTIMO!!

- Il problema è il costo $\Theta(n)$ per la ricombinazione, cioé per calcolare suffissi e prefissi ottimi

NON ABBIAMO USATO Divide et Impera per Prefisso e suffisso!!



$mid = \frac{start + end}{2}$

Supponiamo di avere per A_b con $b = \{1, 2\}$

- intervallo con somma massimo (i_b, j_b)
- prefisso e suffisso ottimo p_b e s_b posizioni degli estremi

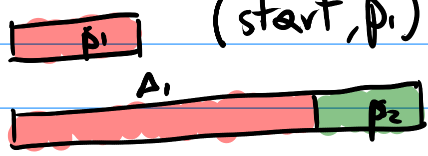
Definiamo $\Sigma(A, i, j) := \sum_{k=i}^{j-1} A[k]$

- Intervallo di somma massima di A

  il migliore tra (i_1, j_1) (i_2, j_2) (s_1, p_2)

- Prefisso migliore per A

  il migliore tra

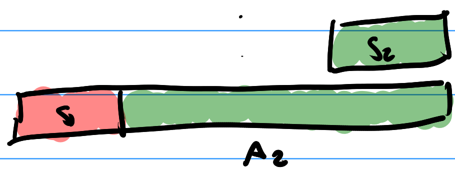


$(start, p_1)$

$(start, p_2)$

- Suffisso migliore per A

  il migliore tra



(s_2, end)

(s_1, end)

Complessità: il passo di combinazione adesso costa $O(1)$

quindi $T(n) = 2T(n/2) + \Theta(1)$

implica $T(n) = \Theta(n)$

Esercizio Completare l'algoritmo che risolva il problema dell'intervallo di somma massima come descritto a pag 7. Ma utilizzando l'algoritmo ricorsivo di complessità $\Theta(n)$

Esercizio Dato un array ORDINATO di numeri A, di lunghezza n, e dato un valore x

Calcolare in $\Theta(\log(n))$ passi quante volte x è presente in A

Esercizio Considerate in input

a : numero reale

n : numero intero ≥ 0

Assumendo che le operazioni $+, \cdot, //$ abbiano complessità $\Theta(1)$

è possibile calcolare a^n con l'algoritmo di complessità $\Theta(n)$

```
t = 1
for i = 1 ... n
    t := t · a
return t
```

Trovate un algoritmo di complessità $\Theta(\log n)$

INDIZI

- $a^{n_1+n_2} = a^{n_1} \cdot a^{n_2}$
- $a^{27} = a^{16} \cdot a^{8} \cdot a^{2} \cdot a^{1}$
- $a^{2n} = a^n \cdot a^n$