POLITECNICO DI MILANO

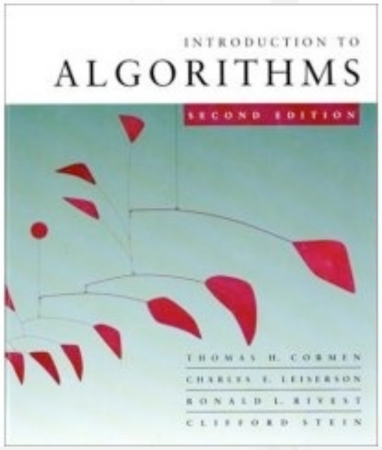
6. Divide et Impera

Informatica 3

Andrea Mocci, mocci@elet.polimi.it

Ringraziamenti

- Parte di queste slide proviene da vecchie slide di Paolo Costa e di altre slide preparate dal Prof. Lanzi per il corso di Algoritmi e Strutture Dati…
- A loro volta tratte da slide
del corso Introduction to Algorithms (2005-fall-6046) tenuto dal Prof. Leiserson all'MIT (http://people.csail.mit.edu/cel/)
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein Introduction to Algorithms, Second Edition, The MIT Press, Cambridge, Massachusetts London, England McGraw-Hill Book Company



Algoritmi di Sorting

- Esempio: Algoritmi di Sorting
- Insertion Sort

```
INSERTION-SORT (A, n) ⊳ A[1 . . n]
   for j ← 2 to n
        do key ← A[ j]
            i ← j − 1
            while i > 0 and A[i] > key
                 do A[i+1] ← A[i]
                     i ← i − 1
            A[i+1] = key
```

Complessità dell'Insertion Sort

- Nel caso peggiore:
  - n volte n scambi
  - $T(n) = O(n^2)$

- Va bene quando n è piccolo
- Esistono soluzioni migliori quando n è grande

Verso un sorting più efficiente

- L'idea: Suddividere il problema in sottoproblemi più piccoli, chiamando ricorsivamente l'algoritmo sui problemi più piccoli ***(Divide-et-impera)***

- In genere, in maniera ricorsiva fino ad un problema veramente semplice da risolvere, o già risolto
  - In questo caso: ordinare due vettori di dimensione 1

- Ricombinare i problemi più semplici per riottenere il problema a livello superiore ***(Ricombinazione)***
  - Per esempio: riunire due vettori ***ordinati*** di dimensione 1 in un vettore ***ordinato*** di dimensione 2

Merge Sort

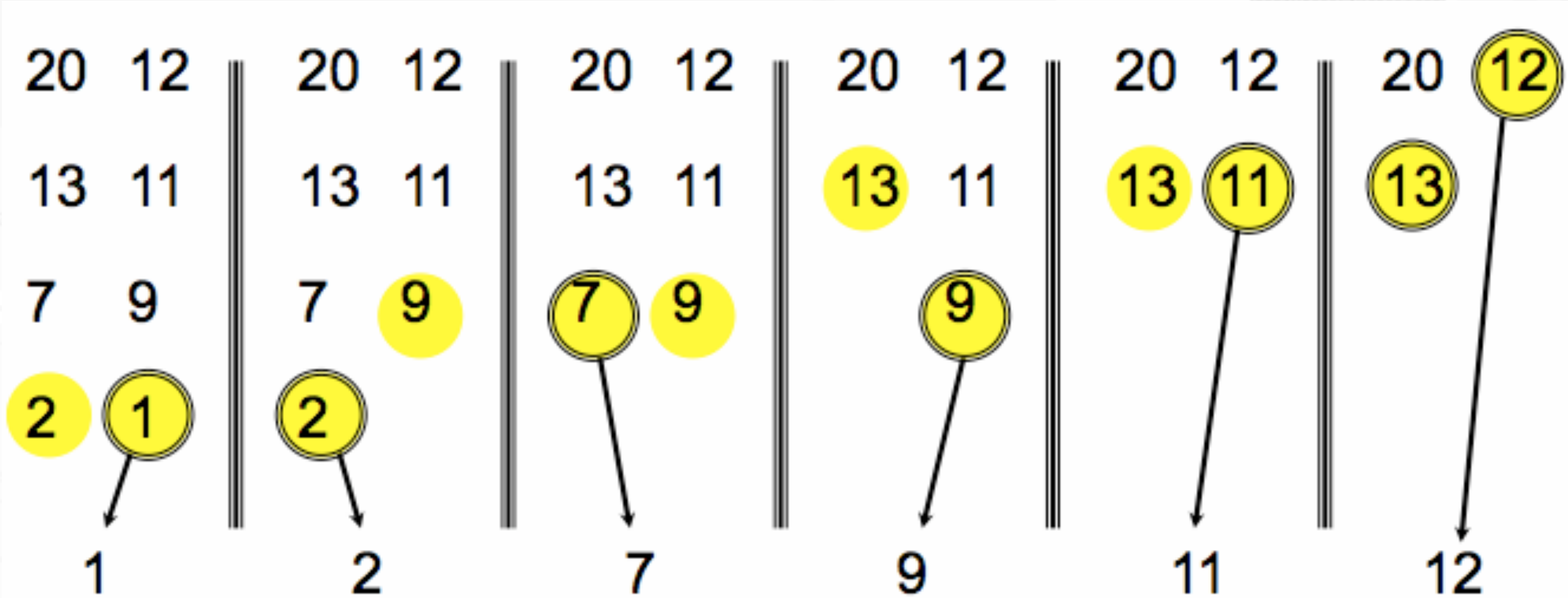
- Approccio divide-et-impera
- Divido l'array in due array di ugual dimensione
- Chiamo l'algoritmo sugli array
- Ricombino per ottenere la soluzione al problema di livello superiore

MERGE-SORT A[1 . . n]

1. If n = 1, done.
2. MERGE-SORT A[1 . . n/2]
3. MERGE-SORT A[n/2 +1 . . n]
4. merge the two sorted arrays
   A[1 . . n/2] and A[n/2 +1 . . n]

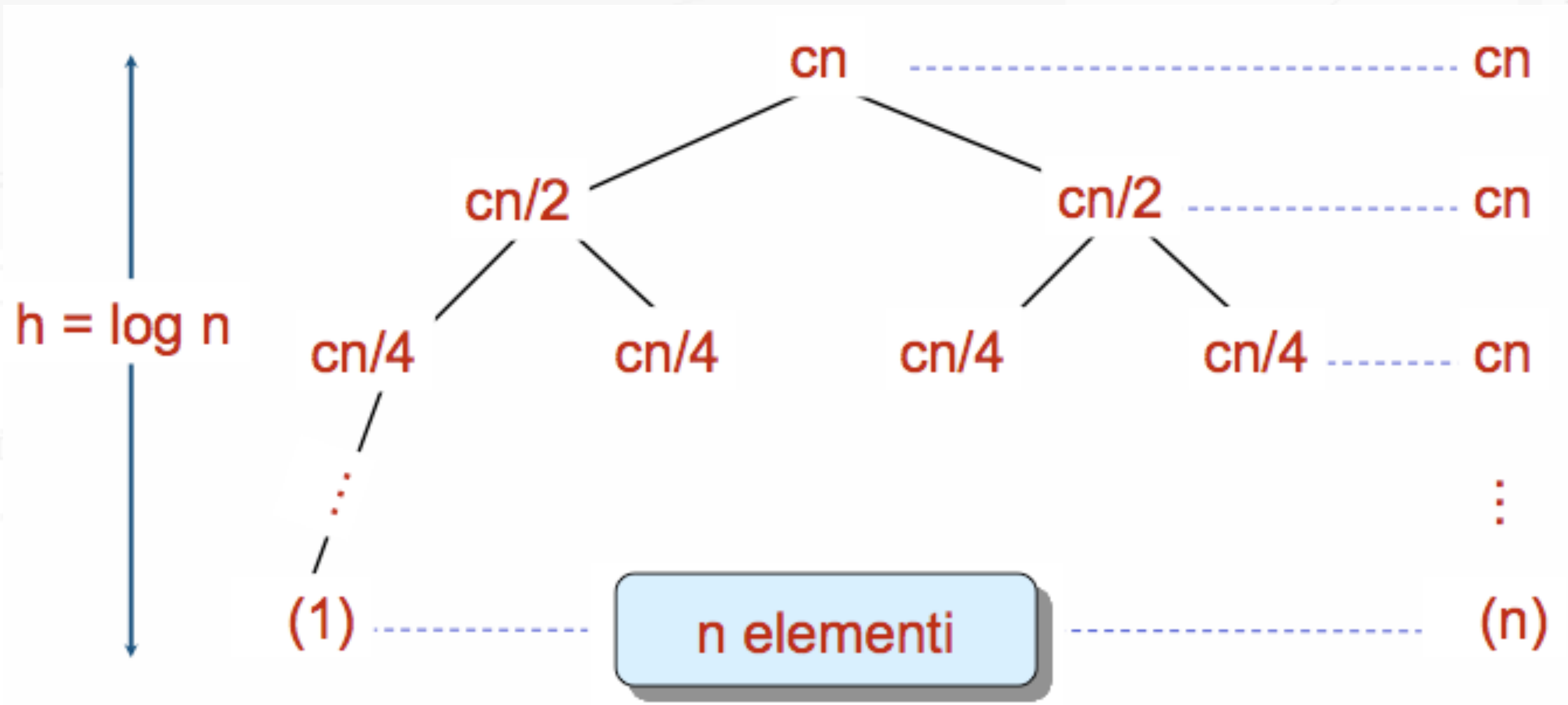
Ricombinazione

- Il merge è $\Theta(n)$

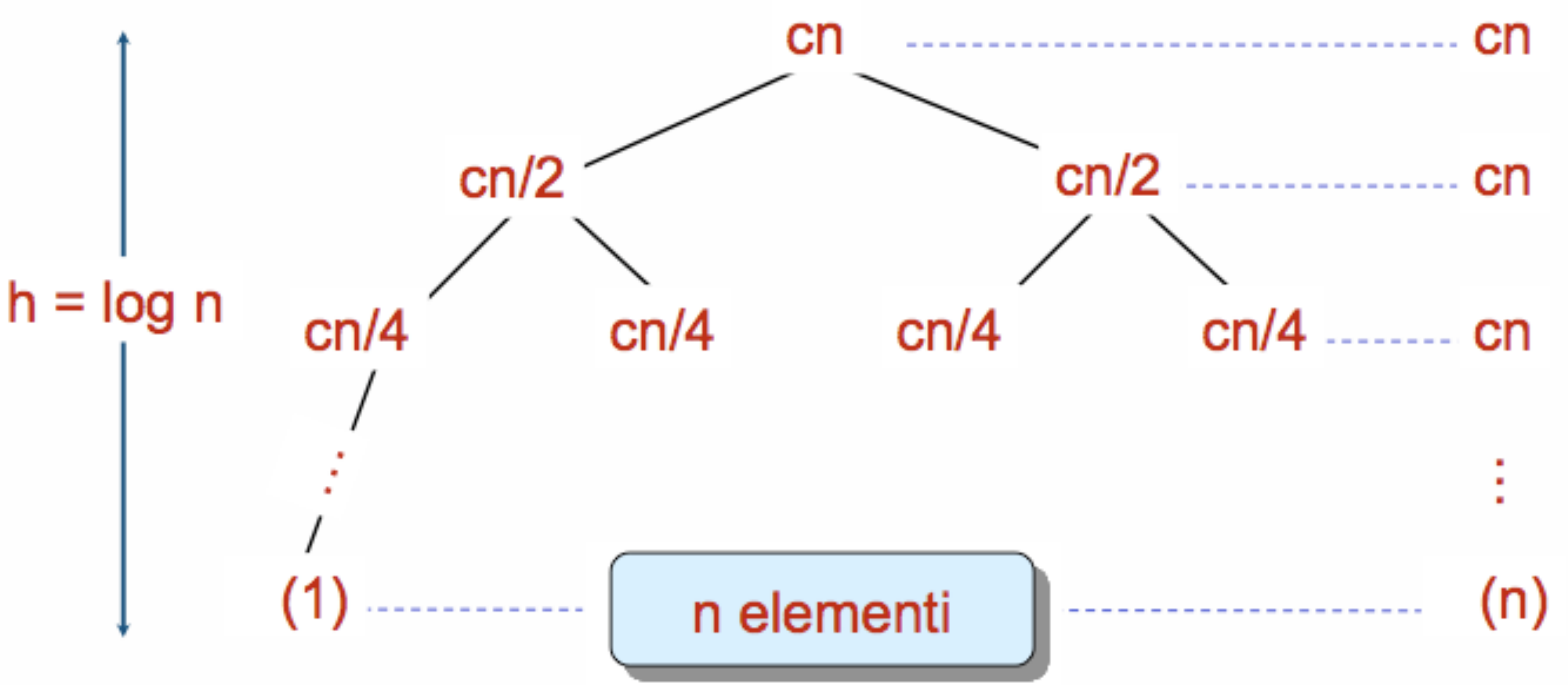


Analisi della complessità

- Occorre calcolare la complessità di una espressione ricorsiva del tipo:
- $T(n) = 2\,T(n/2) + cn$, con $c > 0$ costante



Analisi della complessità



- Complessità: $\Theta(n \log n)$

Esempio: Merge Sort

- Divide: divide il vettore di lunghezza n in due sottovettori di n/2 elementi
- Impera: Ordina ricorsivamente i due sottovettori
- Ricombina le soluzioni dei sottoproblemi in tempo lineare

$$T(n) = \underbrace{2}_{\#\ \text{sottoproblemi}}T(\underbrace{n/2}_{\text{grandezza dei sottoproblemi}}) + \underbrace{cn}_{\text{costo suddivisione e ricombinazione}}$$

- Come calcoliamo T(n)?

Analisi della Complessità per algoritmi ricorsivi

- Supponiamo di considerare forme ricorsive del tipo:
- $T(n) = a\,T(n / b) + f(n)$
- Dove $a >= 1$, $b > 1$, f asintoticamente positiva
- La complessità di $T(n)$ dipende da a, b, $f(n)$
  - Dipende in sostanza dal fatto che domini la suddivisione in sottoproblemi o la ricombinazione (o che crescano in maniera simile)
  - Si confronta $f(n)$ con $n^{log\_b(a)}$

Analisi della Complessità per algoritmi ricorsivi

- Tre casi possibili:
  - f(n) cresce più lentamente di $n^{\log_b(a)}$
  - Domina la suddivisione ($n^{\log_b(a)}$)

  - f(n) cresce in maniera simile a $n^{\log_b(a)}$
  - La complessità dipende da entrambi i fattori

  - f(n) cresce più velocemente di $n^{\log_b(a)}$
  - Domina f(n)

Analisi di Algoritmi Ricorsivi: Master Method (o Master Theorem)

- $T(n) = aT(n/b) + f(n)$
  dove $a \geq 1$, $b > 1$, ed f è asintoticamente positiva

- Caso 1: $f(n) = O(n^{\log_b a - \varepsilon})$ per una costante $\varepsilon > 0$
  - Soluzione: $T(n) = \Theta(n^{\log_b a})$

- Caso 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$ per una costante $k \geq 0$
  - Soluzione: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

- Caso 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per una costante $\varepsilon > 0$
  - $f(n)$ soddisfa a $f(n/b) \leq c\, f(n)$ per una costante $c < 1$
  - Soluzione: $T(n) = \Theta(f(n))$

Altra Formulazione del Master Method

- Si applica a forme ricorsive del tipo,

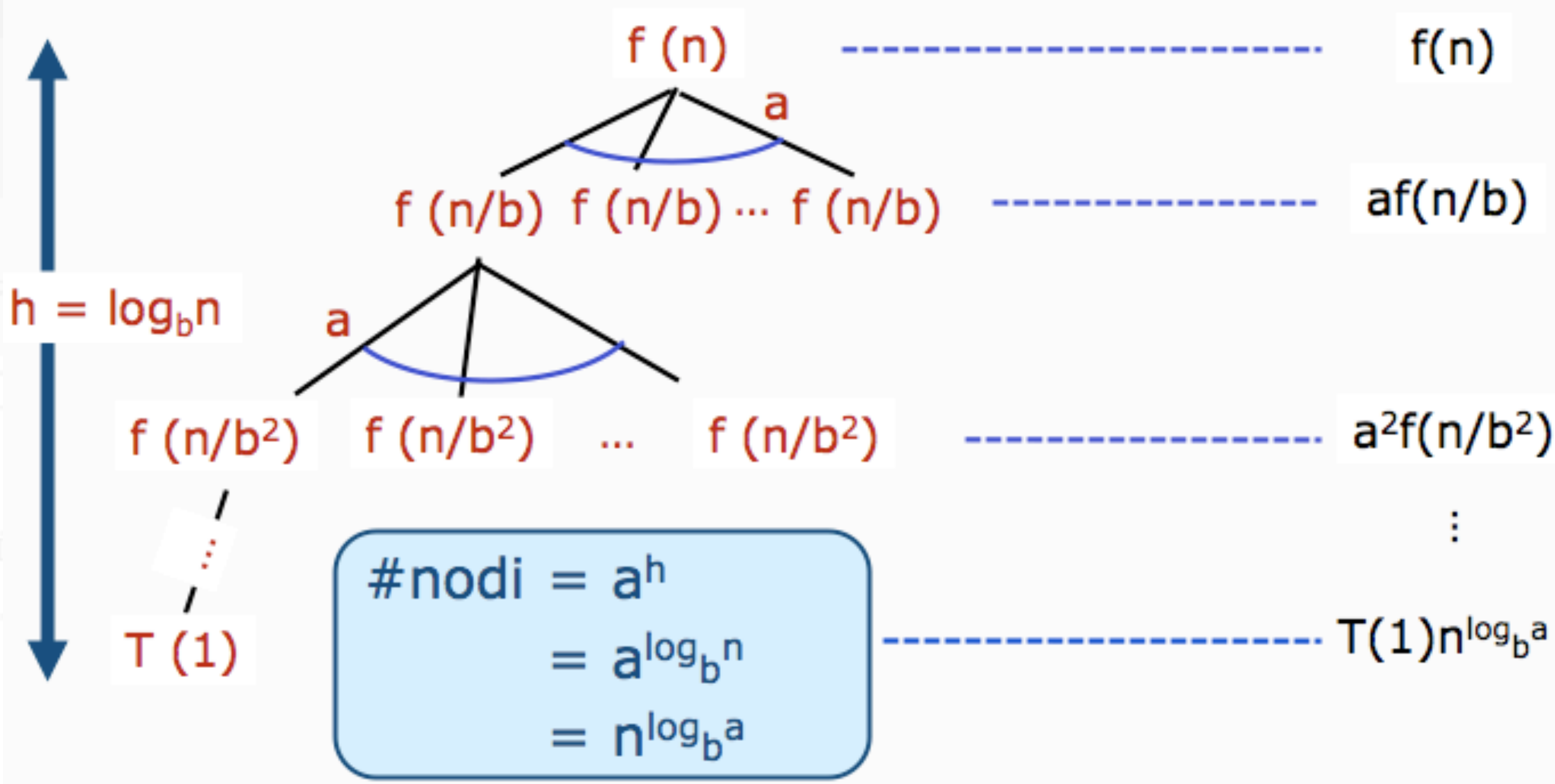
  $T(n) = aT(n/b) + cn^k$

  dove $a \geq 1$, $b > 1$ e $k \geq 0$

- Ancora tre casi,
  - $T(n) = \Theta(n^{\log_b a})$ se $a > b^k$
  - $T(n) = \Theta(n^k \log n)$ se $a = b^k$
  - $T(n) = \Theta(n^k)$ se $a < b^k$

L'idea del Master Method

Albero di ricorsione



Esempi

- Esempio 1: $T(n) = 4\,T(n/2) + n$
  - $a = 4;\ b = 2$
  - $n^{\log_b a} = n^2;\ f(n) = n$
  - Caso 1: $T(n) = \Theta(n^2)$
- Esempio 2: $T(n) = 4\,T(n/2) + n^2$
  - $a = 4;\ b = 2\ ;\ n^{\log_b a} = n^2;\ f(n) = n^2$
  - Caso 2: $f(n) = \Theta(n^2 \lg^0 n);\ k = 0$
  - $T(n) = \Theta(n^2 \lg^0 n)$
- Esempio 3: $T(n) = 4\,T(n/2) + n^3$
  - $a = 4;\ b = 2\ \ n^{\log_b a} = n^2;\ f(n) = n^3$
  - Caso 3: $T(n) = \Theta(n^3)$

Ricerca Binaria
Codifica in C++

```
bool binary_find(int v[], int n, int x)
{
  int l = -1;
  int r = n; // l, r are beyond array bounds
  while (l+1 != r) { // Stop when l, r meet
    int i = (l+r)/2;                   // Check middle
    if (x < array[i]) r = i;           // Left half
    if (x == array[i]) return true; // Found it
    if (x > array[i]) l = i;           // Right half
  }
  return false; // Search value not in array
}
```

Esempio: Ricerca Binaria

- Divide: confronta l'elemento centrale
- Impera: ricerca l'elemento in uno dei due sottovettori
- Non c'è ricombinazione

$$T(n) = 1T(n/2) + c$$

sottoproblemi

grandezza dei sottoproblemi

costo suddivisione e ricombinazione

- Per calcolare T(n), $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$, ovvero, il secondo caso con k=0
- $T(n) = \Theta(\lg n)$

Calcolo di a^n

- Compute a^n, where $n \in \mathbb{N}$
- L'algoritmo tipico è $\Theta(n)$, possiamo fare di meglio?

- Approccio Divide-et-Impera

$$a^n = \begin{cases} a^{n/2} \times a^{n/2} & \text{se } n \text{ è pari} \\ a^{(n-1)/2} \times a^{(n-1)/2} \times a & \text{se } n \text{ è dispari} \end{cases}$$

- $T(n) = T(n/2) + \Theta(1) \;\Rightarrow\; T(n) = \Theta(\lg n)$

Moltiplicazione fra Matrici

Input: $A = [a_{ij}],\ B = [b_{ij}].$

Output: $C = [c_{ij}] = A \times B.$

$$\left.\begin{array}{l} A = [a_{ij}],\ B = [b_{ij}]. \\ C = [c_{ij}] = A \times B. \end{array}\right\} \quad i, j = 1, 2, \dots, n.$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

Algoritmo Standard

for $i \leftarrow 1$ **to** n
    do for $j \leftarrow 1$ **to** n
        do $c_{ij} \leftarrow 0$
            for $k \leftarrow 1$ **to** n
                do $c_{ij} \leftarrow c_{ij} + a_{ik} \times b_{kj}$

Complessità $\Theta(n^3)$

Algoritmo Divide-et-Impera

matrice nxn = 2x2 matrice
di (n/2)x(n/2) sottomatrici

$$\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \times \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]$$

$$C = A \times B$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\}$$

8 multiplicazioni di matrici (n/2)×(n/2)
4 somme di matrici (n/2)×(n/2)

Algoritmo Divide-et-Impera

$$T(n) = 8T(n/2) + \Theta(n^2)$$

moltiplicazioni

Su matrici n/2

Somme su matrici n/2xn/2

$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow$ Primo caso $\Rightarrow T(n) = \Theta(n^3)$

Nessun miglioramento!

Moltiplicazione fra Matrici: Metodo di Strassen

- La moltiplicazione fra due matrici richiede solo 7 chiamate ricorsive, non 8 come abbiamo appena visto

$P_1 = a \times (f - h)$

$P_2 = (a + b) \times h$

$P_3 = (c + d) \times e$

$P_4 = d \times (g - e)$

$P5 = (a + d) \times (e + h)$

$P6 = (b - d) \times (g + h)$

$P7 = (a - c) \times (e + f\ \ )$

7 moltiplicazioni
18 somme/sottrazioni

$r = P_5 + P_4 - P_2 + P_6$

$s = P_1 + P_2$

$t = P_3 + P_4$

$u = P_5 + P_1 - P_3 - P_7$

Moltiplicazione fra Matrici: Metodo di Strassen

- Divide: suddivide A and B in sottomatrici di dimensione (n/2)x(n/2) utilizzati per creare i termini che devono essere moltiplicati usando + e –
- Impera: esegue 7 moltiplicazioni di sottomatrici (n/2)x(n/2)
- Ricombina: crea C usando somme e sottrazioni di sottomatrici (n/2)x(n/2)

Il migliore è $\Theta(n^{2.37\ldots})$

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow T(n) = \Theta(n^{\lg 7})$$

2.81 sembra poco ma appare all'esponente!

TdE 5 Luglio 2004

- In un array a contenente n elementi distinti, si dice che la coppia (a[i], a[j]) costituisce un'inversione se i<j e a[i]>a[j]. Il numero di inversioni presenti in a *si definisce come il numero di coppie (a[i], a[j]), con 0≤i<j≤n-1 che costituiscono un'inversione. Per esempio, per l'array contenente gli elementi [3, 7, 2, 9, 5], il numero di inversioni in a è 4, corrispondente alle coppie (3,2), (7,2), (7,5), (9,5).*

TdE 5 Luglio 2004

- **Quesito 1.** Indicare la relazione tra la complessità di calcolo dell'algoritmo di ordinamento per inserzione e il numero di inversioni presenti nell'array che esso riceve in ingresso. Fornire una breve spiegazione.

Codice dell'Algoritmo

- Facendo riferimento al codice dell'algoritmo:

```
static void inssort(Elem[] array) {
  for (int j=1; j<array.length; j++)
    for (int i=j; (i>0) && (array[i].key()<array[i-1].key()); i--)
      DSutil.swap(array, i, i-1);
}
```

- Il numero di invocazioni a swap sia N_{INV}, ossia il numero di inversioni
- Come faccio ad esemprimere la complessità dell'algoritmo in funzione di N_{INV}?

Complessità ed Inversioni (1)

```
static void inssort(Elem[] array) {
  for (int j=1; j<array.length; j++)
     for (int i=j; (i>0) && (array[i].key()<array[i-1].key()); i--)
        DSutil.swap(array, i, i-1);
}
```

- Se l'array iniziale consiste di n elementi ordinati, quanti confronti vengono effettuati?
- Risposta: un confronto per ciascun j, quindi n – 1 confronti
- Il corpo del secondo ciclo infatti non viene mai eseguito

Complessità ed Inversioni (2)

```
static void inssort(Elem[] array) {
 for (int j=1; j<array.length; j++)
    for (int i=j; (i>0) && (array[i].key()<array[i-1].key()); i--)
       DSutil.swap(array, i, i-1);
}
```

- Se l'array iniziale consiste di n elementi ordinati in senso decrescente, quanti confronti vengono effettuati?
- Risposta: per ciascun j, il ciclo interno viene sempre eseguito: n (n – 1) / 2

Complessità ed Inversioni (3)

```
static void inssort(Elem[] array) {
  for (int j=1; j<array.length; j++)
     for (int i=j; (i>0) && (array[i].key()<array[i-1].key()); i--)
        DSutil.swap(array, i, i-1);
}
```

- Qual è il rapporto tra il numero di inversioni e il numero dei confronti?
- Risposta: Per ogni ciclo interno: o eseguo uno swap (inversione) o faccio un confronto che non è uno swap ed esco dal ciclo. Quindi il numero totale di confronti è compreso tra N_{inv} e $N_{inv} + (n-1)$ (supponendo che ogni volta il ciclo interno termini per una violazione del confronto)

Complessità insertion sort

- Quindi posso sempre trovare tre costanti $k_{1,}$ k_2, k_3, tali per cui la complessità dell'algoritmo di insertion sort è:
- $T(N_{inv,}\ n) = k_1 N_{inv} + k_2 n + k_3 = O(N_{inv} + n)$

- Ciò spiega anche perché l'algoritmo di ordinamento per inserzione è particolarmente efficiente su array "quasi ordinati", che cioè hanno un ridotto numero di inversioni (quasi lineare!!!!!)

TdE 5 Luglio 2004

- **Quesito 2**: Scrivere un semplice algoritmo per contare il numero di inversioni presenti in un array, che abbia complessità $\Theta(n^2)$, fornendo una sintetica argomentazione del fatto che esso calcola effettivamente il numero delle inversioni e del fatto che è quadratico.

Quesito 2: Risposta

- ```
  static int invCount(Elem[] array) {
    int c = 0;
      for (int i=0; i<array.length-1; i++)
          for (int j=i+1; (j<array.length); j++)
              if (a[i]>a[j]) c++;
    return c;
  }
  ```
- L'algoritmo ha complessità quadratica perché il ciclo for interno viene eseguito n-1 volte e il suo corpo viene ripetuto per un numero decrescente di volte, partendo da n-1 e arrivando a 1. Nel far ciò l'algoritmo considera tutte le coppie (a[i], a[j]) con $0<i<j\leq a.length$, e quindi effettivamente calcola il numero delle inversioni.
  ```

TdE: 5 Luglio 2004

- **Quesito 3:** Definire un algoritmo che calcoli il numero di inversioni in un array in tempo $\Theta(n \cdot \log n)$ nel caso pessimo, fornendo un'opportuna, sintetica spiegazione della sua correttezza e una valutazione della sua complessità. Suggerimento: si applichi il metodo *divide et impera, ispirandosi all'algoritmo di ordinamento per fusione (mergesort)*.

Quesito 3: risposta

- Consideriamo la fase di ricostruzione dell'algoritmo di merge sort.
- Posso contare le inversioni nel seguente modo:
  - Durante la ricombinazione (merge), quando inserisco un elemento del secondo sottoarray nel primo, sto implicitamente eliminando delle inversioni:
  - [1 9 10 11] [2 3 4 5]
  - Quando inserisco il 2, elimino le inversioni (9, 2) (10, 2) (11, 2); quando inserisco il 3, elimino (9, 3) (10, 3) (11, 3), etc
  - Elimino ad ogni passo un numero di inversioni pari al numero di elementi nel primo sotto-array che rimangono

Quesito 3: soluzione

- Posso riscrivere l'algoritmo di mergesort semplicemente riscrivendo la parte di merge, in modo che restituisca il numero di inversioni eliminate

```
static int mergesortCountInv(Elem[] array, Elem[] temp,
                             int l, int r) {
  if (l == r) return 0;
  int mid = (l+r)/2;
  int c1 = mergesortCountInv(array, temp, l, mid);
  int c2 = mergesortCountInv(array, temp, mid+1, r);
  return c1 + c2 + mergeCountInv(array, temp, l, mid, mid+1,
r);
}
```

Quesito 3: soluzione

```
static int mergeCountInv(Elem[] array, Elem[] temp,
                         int l1, int r1, int l2, int r2) {
  for (int i=l1; i<=r2; i++)
    temp[i] = array[i];
     int countInv = 0; int i1 = l1; int i2 = l2;
     for (int curr=l1; curr<=r2; curr++) {
       if (i1 > r1)
         array[curr] = temp[i2++];
       else if (i2 > r2)
         array[curr] = temp[i1++];
       else if (temp[i1].key() < temp[i2].key())
         array[curr] = temp[i1++];
       else {
         //temp[i2] scavalca gli r1-i1+1 elementi
         //ancora presenti nel primo segmento
         countInv += r1-i1+1;
         array[curr] = temp[i2++];
       }
  } return countInv;
}
```