Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 6

Ugo Vaccaro

La tecnica Divide-et-Impera

Gli algoritmi basati sulla tecnica Divide-et-Impera hanno, generalmente, la seguente struttura ricorsiva:

```
Algoritmo D&I(x)
IF (l'input x é sufficientemente piccolo o semplice) {

        RETURN adhoc(x) (% ovvero risolvi x direttamente)

        } ELSE { decomponi l'istanza di input x in k istanze piú piccole x_1, x_2, ..., x_k

                }

  s1= Algoritmo D&I(x_1)

  s2= Algoritmo D&I(x_2)

    ⋮

  sk= Algoritmo D&I(x_k)

componi le sottosoluzioni s1, s2, ..., sk alle istanze x_i per ottenere una soluzione globale
s alla istanza completa x

RETURN(s)
```

Quando analizzeremo un generico algoritmo $\mathcal{A}$ progettato in accordo alla tecnica Divide-et-Impera, il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ sarà, in generale, esprimibile mediante un'equazione di ricorrenza, del tipo:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ kT(f(n)) + g(n) & \text{altrimenti} \end{cases}$$

dove

- c_0 è una costante che conta il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ quando l'input è di dimensione "piccola", ovvero di dimensione $\leq n_0$
- k è il numero di chiamate ricorsive di $\mathcal{A}$
- $f(n)$ è (una limitazione superiore al)la dimensione dell'input di ogni chiamata ricorsiva di $\mathcal{A}$
- $g(n)$ è il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ al di fuori della ricorsione, ovvero il numero di operazioni elementari per la suddivisione dell'istanza di input in "sotto-istanze", più il tempo

per la composizione delle "sotto-soluzioni" alle sotto-istanze in una soluzione globale all'istanza completa di partenza.

Ricordiamo il seguente risultato per la risoluzione delle più comuni equazioni di ricorrenza:

Teorema. La soluzione alla ricorrenza

$$T(n) = \begin{cases} d & \text{se } n \leq 1 \\ aT(n/c) + bn^k & \text{altrimenti} \end{cases}$$

per a, c, b, k costanti, è

$$T(n) = \begin{cases} O(n^k) & \text{se } a < c^k \\ O(n^k \log n) & \text{se } a = c^k \\ O(n^{\log_c a}) & \text{se } a > c^k \end{cases}$$

Come primo e classico esempio di applicazione della tecnica Divide-et-Impera, rideriviamo l'algoritmo della Ricerca Binaria in un array ordinato. In input abbiamo quindi una coppia (a, k), dove a è array $a = a[0]a[1] \ldots a[n-1]$ di numeri, ordinato in senso non decrescente, ovvero supponiamo che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, e k è un arbitrario numero. L'output che desideriamo è un valore $i \in \{0, 1, \ldots, n-1\}$, se $k = a[i]$, desideriamo invece la risposta `‘‘non c’è’’` se $k \neq a[i]$, per ogni $i \in \{0, 1, \ldots, n-1\}$. L'algoritmo della Ricerca Binaria può essere scritto nel seguente modo. Esso prende in input l'array $a = a[0]a[1] \ldots a[n-1]$ e il numero k, ed effettua la ricerca nel sottoarray $a[s] \ldots a[d]$. La risoluzione del problema avverrà chiamando l'algoritmo con parametri $s = 0$ e $d = n - 1$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ‘‘non c’è’’
    } }
  c = (s + d)/2
  IF (k ≤ a[c]) {
    RETURN(RicercaBinaria(a, k, s, c)
  } ELSE {
    RETURN(RicercaBinaria(a, k, c + 1, d)
  }
```

Come è ben noto, detta $T(n)$ la complessità di `RicercaBinaria`$(a, k, 0, n-1)$, si ha che

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per costanti c_0 e c opportune. Come è altrettanto ben noto, la soluzione dell'equazione di ricorrenza sopra riportata è $T(n) = O(\log n)$.

Vediamo un'applicazione della ricerca binaria. Supponiamo che il docente del corso di Progettazione di Aforismi, Prof. I.N. Geniere, abbia collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova. Supponiamo inoltre che gli elementi di a siano stati ordinati, ovvero valga che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che *solo* gli studenti che hanno ottenuto una votazione maggiore di un certo valore k possano sostenere l'esame. Il Prof. Geniere vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:

Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Il Prof. Geniere propone la seguente soluzione al problema: si effettui una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k.

```
CONTAPROMOSSI1(a, k)
1. c = 0
2. FOR(i = 0; i < n; i = i + 1) {
3.     IF (a[i] > k) { c = c + 1
         }
    }
RETURN c
```

La complessità dell'algoritmo è chiaramente $\Theta(n)$.

Vediamo una soluzione (marginalmente) piu efficiente. Sfruttando il fatto che l'array a è ordinato, si può evitare la scansione completa e ci si può fermare al primo elemento il cui valore è $> k$; infatti si sà che tutti gli elementi successivi saranno anch'essi maggiori di k.

```
CONTAPROMOSSI2(a, k)
1. i = 0
2. WHILE ((i < n)&&(a[i] ≤ k)) {
3.       i = i + 1
    }
4. RETURN n − i
```

Benchè l'algoritmo abbia complessità $\Theta(n)$ nel caso peggiore, e ciò corrisponde al caso in cui nessuno degli studenti abbia ricevuto un voto $> k$, l'algoritmo `CONTAPROMOSSI2` potrebbe terminare anche dopo un solo passo, e ciò nel caso in cui $a[0] > k$. Abbiamo qui un caso in cui il comportamento di un algoritmo nel caso *peggiore* può differire significativamente dal suo comportamento nel caso *migliore*.

Vediamo infine la soluzione efficiente proposta dallo studente I. N. Formatico per contare quanti elementi dell'array a risultano avere valore $> k$. La soluzione efficiente è basata sulla ricerca binaria. La funzione CONTAPROMOSSI3(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \dots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

```
CONTAPROMOSSI3(a, k, i, j)
IF (i > j) {
  RETURN 0
  } ELSE {
      m = ⌊(i + j)/2)⌋
      IF { (a[m] ≤ k) {
      RETURN CONTAPROMOSSI3(a, k, m + 1, j)
          } ELSE {
        RETURN (j − m + 1)+ CONTAPROMOSSI3(a, k, i, m − 1)
      }
    }
```

L'algoritmo viene inizialmente invocato con CONTAPROMOSSI3$(a, k, 0, n-1)$ Ad ogni passo:

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo è zero;
- se $i \leq j$ come prima cosa determiniamo la posizione m dell'elemento centrale, esattamente come si fa con la ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \dots a[j]$
  - se $a[m] > k$, l'elemento centrale è sopra la soglia. Quindi, essendo l'array ordinato, tutti gli $(j-m+1)$ elementi in $a[m] \dots a[j]$ risultano *sopra* la soglia. Il numero di tali elementi, sommato al conteggio di quelli sopra soglia in $a[i] \dots a[m-1]$ (che viene calcolato dalla chiamata ricorsiva) produce il risultato.

Detta $T(n)$ la complessità dell'algoritmo CONTAPROMOSSI3$(a, k, 0, n-1)$, essa soddisfa ovviamente la equazione di ricorrenza

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per costanti c_0 e c opportune. La soluzione dell'equazione di ricorrenza sopra riportata è $T(n) = O(\log n)$[1].

◇

Consideriamo ora un altro esempio di applicazione della tecnica Divide-et-Impera.

Problema: Dato un numero a ed un intero positivo n, calcolare a^n usando il minor numero di moltiplicazioni.

Algoritmo semplice

[1] Lo studente Formatico superò poi l'esame con il massimo dei voti

```
SlowPower(a, n)
  x = a
  FOR(i = 2; i < n + 1; i = i + 1){
    x = x × a
    }
  RETURN(x)
```

Il numero di moltiplicazioni effettuate da `SlowPower`(a, n) è chiaramente pari a $n - 1$. Vediamo come migliorare l'algoritmo usando la tecnica Divide-et-Impera.

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.

Sulla base di queste osservazioni, possiamo pensare il seguente algoritmo.

```
FastPower(a, n)
  IF(n == 1) {
    RETURN(a)
    } ELSE {
      x = FastPower(a, ⌊n/2⌋)
        IF (n é pari) RETURN(x × x)
          ELSE RETURN(x × x × a)
     }
```

Sia $T(n)$ il numero di moltiplicazioni effettuate dall'algoritmo `FastPower`(a, n). É ovvio che

$$T(n) \leq \begin{cases} 0 & \text{se } n = 1 \\ T(n/2) + 2 & \text{altrimenti} \end{cases}$$

da cui ne discende che $T(n) = O(\log n)$, migliorando considerevolmente rispetto all'algoritmo `SlowPower`.

◇

Consideriamo ora il seguente problema.

Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$

Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.

Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo `CoppieIdentiche`$(a, 0, n-1)$

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] == a[i + 1]) {RETURN(1)
                  ELSE RETURN(0)
      } ELSE k = ⌊(i + j)/2⌋
        RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k, j))
```

Sia $T(n)$ il numero di operazioni effettuate dall'algoritmo `CoppieIdentiche`$(a, 0, n-1)$. É ovvio che

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 2 \\ 2T(n/2) + c & \text{altrimenti} \end{cases}$$

per opportune costanti c_0 e c, da cui ne discende che $T(n) = O(n)$.

◇

Altro esercizio:

Input: numero intero $x \geq 0$

Output: $\lfloor \sqrt{x} \rfloor$

Un algoritmo per la risoluzione dell'esercizio, usando la tecnica Divide et Impera potrebbe essere il seguente.

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
```

```
}
  IF (m × m < x) {
  s = m + 1, r = m
  } ELSE {d = m − 1
  }
}
RETURN(r)
```

Complessità: $T(n) = T(n/2) + c \Rightarrow T(n) = O(\log n)$