DIPARTIMENTO DI MATEMATICA E INFORMATICA

UNIVERSITÀ degli STUDI di CATANIA

Algoritmi di ordinamento ricorsivi

Alessandro Ortis
Image Processing Lab - iplab.dmi.unict.it

alessandro.ortis@unict.it
www.dmi.unict.it/ortis/

IMAGE PROCESSING LABORATORY

Ricorsione

quando una funzione chiama se stessa, sia direttamente che tramite altre funzioni, essa viene detta *ricorsiva*

- es: il fattoriale è una funzione intrinsecamente ricorsiva:

$$n! = n*(n - 1)!$$

- ogni funzione ricorsiva può avere un'implementazione iterativa:

```
// implementazione ricorsiva
int fattoriale(int n)
  { if (n == 0) return 1;
    else return n * fattoriale(n - 1);
}

// implementazione iterativa
int fattoriale = 1;
for (int contatore = n; contatore >= 1; contatore --)
   fattoriale *= contatore ;
```

Esempio: Prodotto di due numeri naturali

Prodotto di due numeri naturali a e b

- Soluzione iterativa

$$\text{prod}(a,b) = \underbrace{a+a+a+\ldots+a}_{\text{b volte}}$$

- Soluzione ricorsiva

$$\text{prod}(a,b) = a \qquad \text{se } b=1$$

$$\text{prod}(a,b) = a + \text{prod}(a, b-1) \qquad \text{altrimenti}$$

Ricorsione

Si può usare l'induzione matematica per convincersi che un programma ricorsivo si comporta correttamente:

- Caso base: calcola direttamente $0! = 1$
- Altrimenti: assumendo che il programma calcoli $k!$ per $k<N$ (ipotesi induttiva), allora esso calcola $N!$

```
int fattoriale(int n)
  { if (n == 0) return 1;
    else return n * fattoriale(n - 1);
}
```

In pratica, il legame con l'induzione matematica ci dice che le funzioni ricorsive devono soddisfare due requisiti fondamentali:

1. Risolvere in modo esplicito il caso base
2. Ogni chiamata ricorsiva deve avere come argomenti valori più piccoli

Ricorsione

Possiamo dimostrare la correttezza della seguente funzione *puzzle*?

```
int puzzle(int N){
   if (N == 1) return 1;
   if (N % 2 == 0)
      return puzzle(N/2);
   else
      return puzzle(3*N+1);
}
```

Ricorsione

Possiamo dimostrare la correttezza della seguente funzione *puzzle*?

```
int puzzle(int N){
   if (N == 1) return 1;
   if (N % 2 == 0)
      return puzzle(N/2);
   else
      return puzzle(3*N+1);
}
```

Se N è dispari la funzione chiama se stessa sull’argomento 3N+1, mentre se N è pari la funzione chiama se stessa su N/2. Non possiamo dimostrare per induzione che questo programma termina perché non tutte le chiamate ricorsive hanno come argomento valori più piccoli di quello dato.

Ricorsione e iterazione

- Qualunque problema risolvibile ricorsivamente può essere risolto con un algoritmo iterativo;
  - per ogni funzione ricorsiva se ne può trovare un'altra che fa la stessa cosa attraverso un ciclo (senza richiamare se stessa)

- la ricorsione spesso produce soluzioni concettualmente più semplici
  - la corrispondente soluzione iterativa sarà normalmente più efficiente, sia in termini di occupazione di spazio di memoria che in termini di tempo di computazione.

Svantaggi della Ricorsione

- Spreco di tempo
  - Ogni chiamata della funzione richiede per se un tempo di esecuzione (indipendente da cosa farà la funzione)

- Spreco di memoria
  - Ad ogni chiamata bisogna memorizzare nello stack una serie di registri e parametri
    - Es. indirizzo dell'istruzione da seguire quando la funzione terminerà la sua esecuzione.
    - Argomenti della funzione
    - Variabili locali

Vantaggi della Ricorsione

- i programmi sono più chiari, più semplici, più brevi e più facili da capire delle corrispondenti versioni iterative

- il programma riflette la strategia di soluzione del problema

- spesso la soluzione trovata può poi trasformarsi in una soluzione iterativa equivalente ma più efficiente

Esempio: elevamento a potenza

- Soluzione ricorsiva: Pot(b,n)

  Pot(b,0) = 1

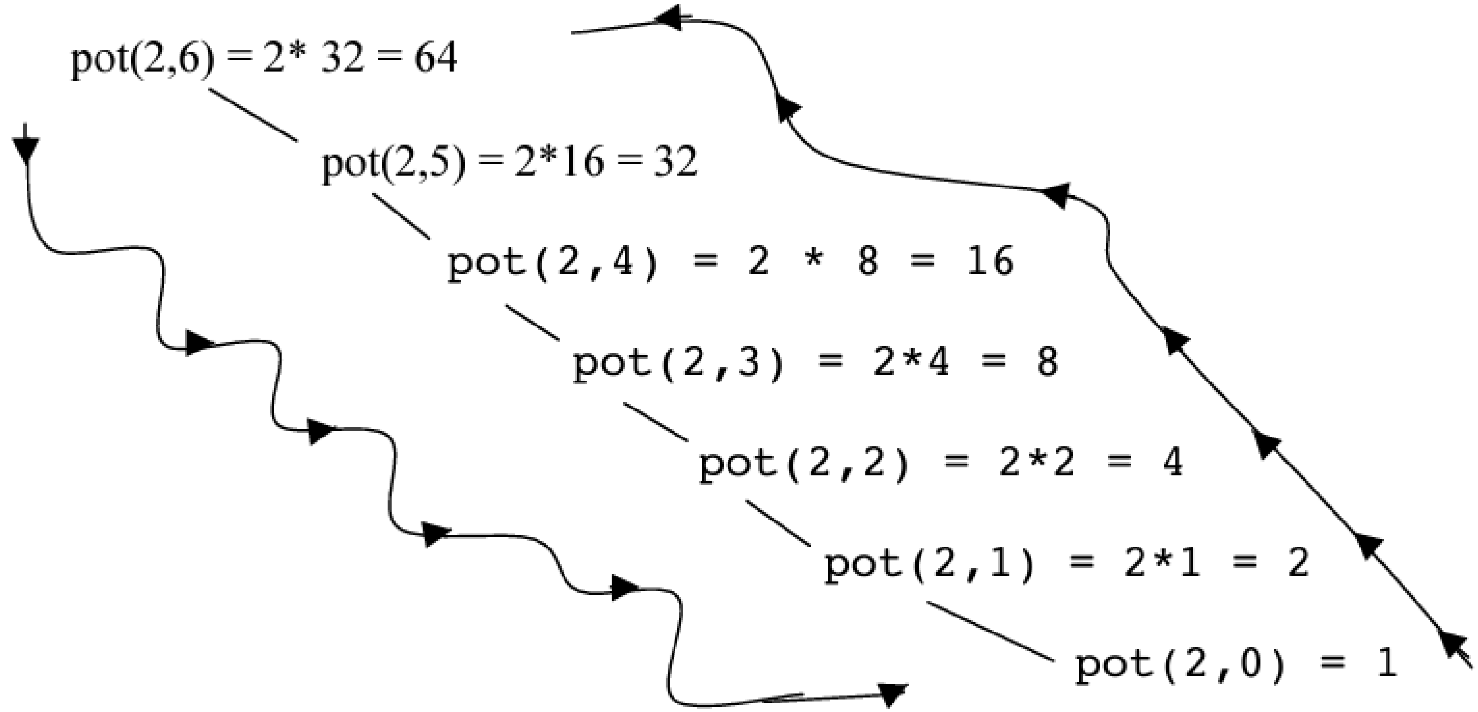
  Pot(b,n) = b * Pot(b,n-1))

Esempio: elevamento a potenza

Quanto costa calcolare ricorsivamente la potenza n-sima di un numero b?

Esempio: elevamento a potenza

Quanto costa calcolare ricorsivamente la potenza n-sima di un numero b?



Si calcolano n prodotti e si occupa uno spazio di memoria proporzionale a n, perché si deve considerare lo spazio per le chiamate in sospeso.

Esempio: elevamento a potenza

La versione iterativa equivalente invece comporta l’esecuzione di n prodotti ma in spazio costante:

```
int potIter(int b, int n){
        int ris = 1;
        for (;0<n;n--)
                ris = ris*b;
        return ris;
}
```

Esercizio: definire una funzione ricorsiva per determinare il massimo in un array di N elementi

Esercizio: definire una funzione ricorsiva per determinare il massimo in un array di N elementi

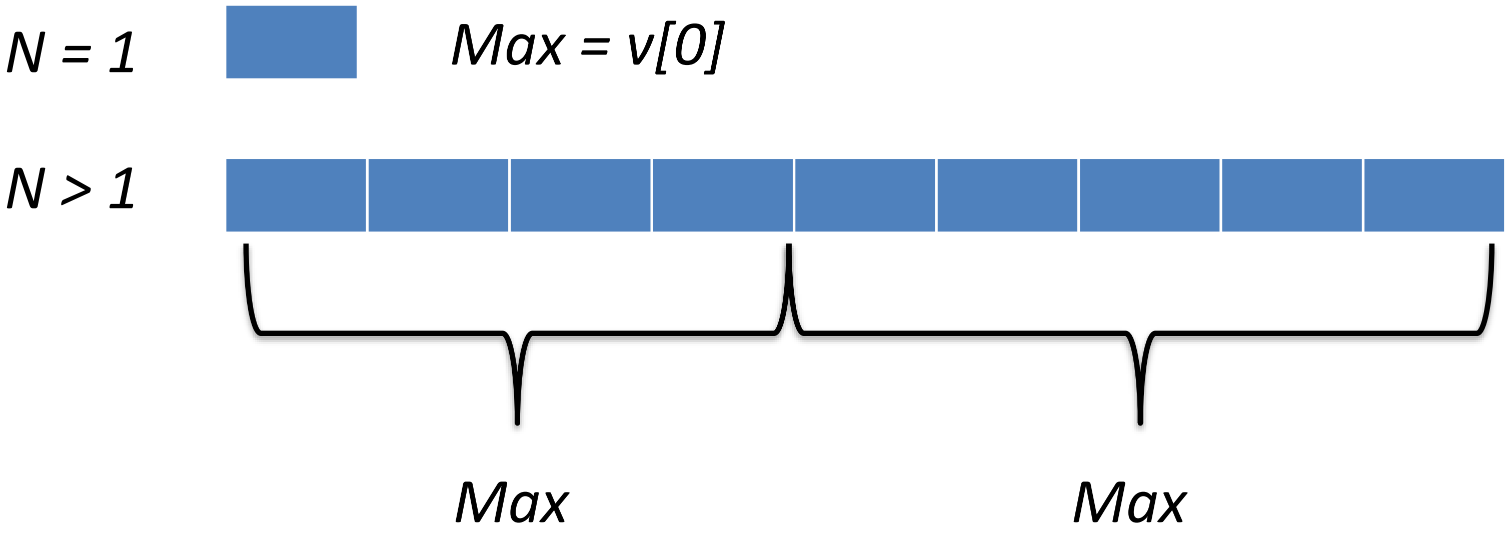
N = 1 *Max = v[0]*

Esercizio: definire una funzione ricorsiva per determinare il massimo in un array di N elementi

N = 1  *Max = v[0]*

N > 1 

Esercizio: definire una funzione ricorsiva per determinare il massimo in un array di N elementi



Esercizio: definire una funzione ricorsiva per determinare il massimo in un array di N elementi

Dimostrazione induttiva:

- Caso base: se N = 1 allora max=v[0]
- Altrimenti: se N>1 dividi l'array in due sottoarray di dimensioni inferiori ad N, trova il max tra i due sottoarray e restituisci il più grande dei due valori.

Merge Sort

Questo algoritmo implementa il paradigma *divide et impera:*

- L'input di dimensione *n* viene partizionato in due parti di lunghezza *n/2*.

- Le due sottosequenze vengono ordinate in maniera ricorsiva fino a quando si ottengono delle sequenze composte da un solo elemento.

- A questo punto la procedura *merge* unisce due sottosequenze ordinate.

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

38	27	43	3	9	82	10

Merge Sort

	Caso Migliore	Caso Medio	Caso Peggiore
MergeSort	O(nLogn)	O(nLogn)	O(nLogn)

La complessità è la medesima in tutti e tre i casi perché l’algoritmo divide sempre le sequenze a metà impiegando un tempo O(log n) e le unisce impiegando un tempo lineare.

Quick Sort

Il Quicksort è l'algoritmo di ordinamento più efficiente. Si basa sulla divisione del vettore in tre partizioni:

- Centrale: contenente un solo elemento detto *pivot*
- Sinistra: contenente tutti gli elementi minori del *pivot*
- Destra: contenente tutti gli elementi maggiori del *pivot*

Come conseguenza avremo che tutti gli elementi della partizione sinistra saranno minori del più piccolo della partizione di destra.

Si applica ricorsivamente l'algoritmo sulle partizioni sinistra e destra fino ad ordinare tutto il vettore.

Il *pivot* può essere scelto a caso.

Quick Sort

44	12	55	42	94	18

Quick Sort

Possiamo definire una procedura *partition* che si occupa di effettuare la partizione e restituire la posizione del pivot.

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

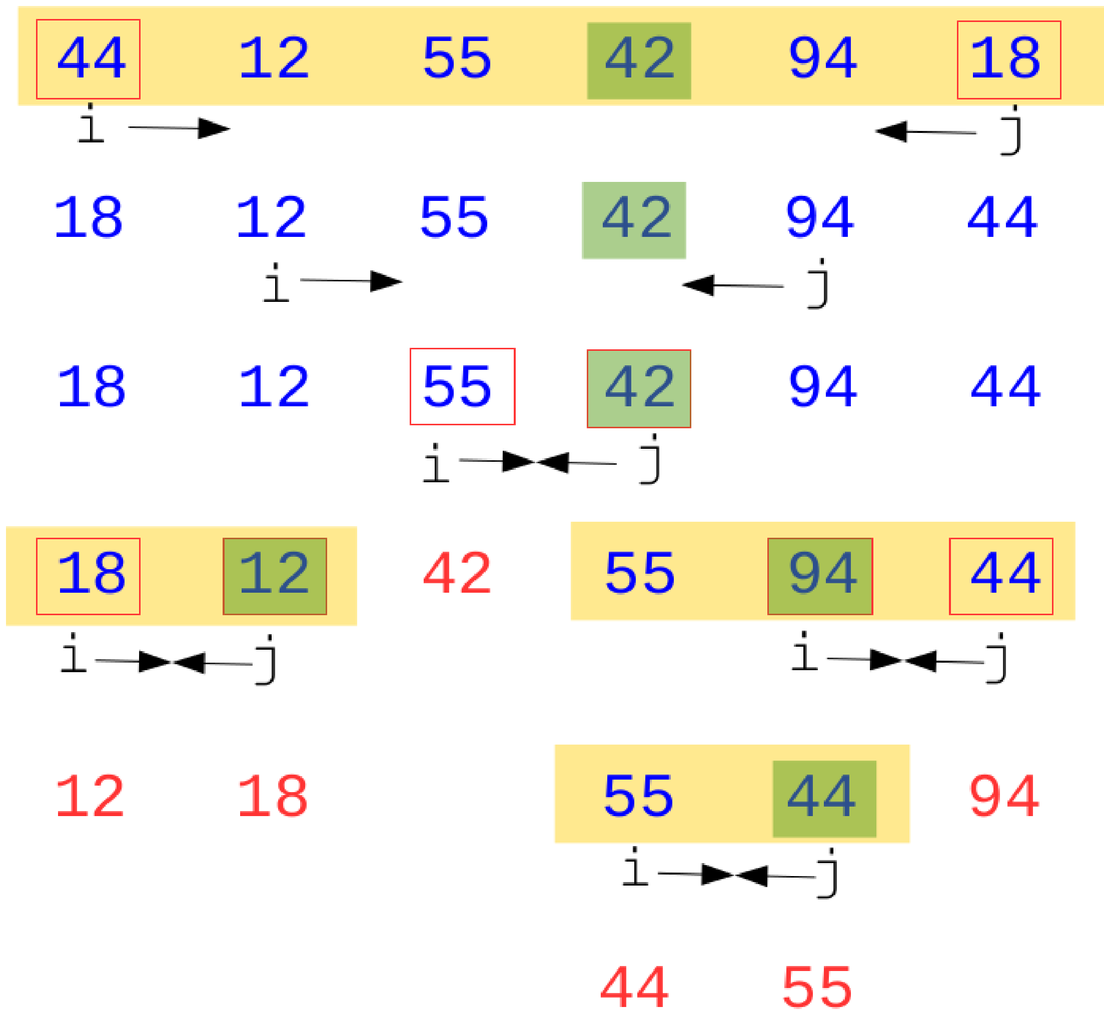
```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

Quick Sort

Implementazione alternativa senza la procedura partition.

```
void QuickSort(el* v, int n) {
   QuickSort(v, 0, n-1);}
void QuickSort(el v[], int s, int d)
{
    int i = s, j = d;
    el tmp;
    el pivot = v[(s + d) / 2];
    while (i <= j) {                  // PARTIZIONE
        while (v[i] < pivot) i++;
        while (v[j] > pivot) j--;
        if (i <= j) {
            tmp = v[i];
            v[i] = v[j];
            v[j] = tmp;
            i++;
            j--;
        }
    };
    if (s < j)          // RICORSIONE
    QuickSort(v, s, j);
    if (i < d)
    QuickSort(v, i, d);
}
```

Quick Sort



44 12 55 42 94 18
i j
18 12 55 42 94 44
i j
18 12 55 42 94 44
i j
18 12 42 55 94 44
i j i j
12 18 55 44 94
i j
44 55

Quick Sort

	Caso Migliore	Caso Medio	Caso Peggiore
QuickSort	O(nLogn)	O(nLogn)	$O(n^2)$

- **Caso peggiore:** quando le due partizioni sono formate da 0 ed n-1 elementi. In questo caso il partizionamento costa O(n) e se questo caso si verifica ad ogni chiamata ricorsiva avremo un costo totale di $O(n^2)$.

- **Caso migliore:** bilanciamento massimo. Si verifica quando i due sottoproblemi hanno dimensione circa n/2. In questo caso il costo è O(nLogn)

- **Caso medio:** è possibile dimostrare che anche con una ripartizione sproporzionata ad ogni livello di ricorsione, il quicksort viene eseguito nel tempo O(nLogn). Questo perché qualsiasi ripartizione con proporzionalità costante produce una ricorsione di profondità O(logn) il cui costo unitario è O(n).