Fondamenti di Informatica (Elettronici)

Algoritmi fondamentali – Lezione 31 (Paoluzzi)

16 dicembre 2020

Algoritmi fondamentali: ordinamento

1. Problema dell'ordinamento
2. Ordinamento per inserzione (insertion sort)
3. Analisi di complessità
4. Ordinamento per selezione (selection sort)
5. Analisi di complessità
6. Ordinamento a bolle (Bubble Sort)
7. Esempio
8. Analisi di complessità.

Section 1

Problema dell'ordinamento

Sort (Ordinamento)

Problema fondamentale in moltissime applicazioni

Vedremo alcuni algoritmi di ordinamento "interno", ovvero in memoria centrale, supponendo che le informazioni da ordinare siano memorizzate in un array, e che il campo chiave (rispetto a cui effettuare l'ordinamento) sia dello stesso tipo dell'indice dell'array

Nelle applicazioni reali il problema dell'ordinamento riguarda normalmente grandi archivi memorizzati su memorie di massa. Il sort è anche un componente fondamentale di molti altri algoritmi.

Gli algoritmi che studieremo possono essere utilizzati in entrambi i casi.

Struttura dati: esempio

Nello studio degli algoritmi di ordinamento supporremo di dover ordinare array di informazioni eterogenee (record) appartenenti a un tipo struct prefissato

```
struct record
    key        ::Int
    nome           ::String
    indirizzo      ::String
    codice_fiscale::String
end

table1 = Array{record,1}()  # inizializzazione array vuoto
# caricamento dei dati nell'array
push!(table1, record(101, "adalberto", "via d.manin", "mnndnl123"))
push!(table1, record(567, "gilberto", "via p.albertelli", "lbrtpl012"))
push!(table1, record(232, "roberto", "via c.cavour", "cvrcml456"))
push!(table1, record(133, "alberto", "via g.garibaldi", "grbgsp789"))
```

E' ovvio che ridefinendo la struttura precedente potremo applicare gli algoritmi implementati all'ordinamento di archivi differenti.

Section 2

Ordinamento per inserzione (insertion sort)

Ordinamento per inserzione

- Iniziare con la mano vuota

Ordinamento per inserzione

- Iniziare con la mano vuota
- Inserire una carta nella giusta posizione della mano già ordinata

Ordinamento per inserzione

- Iniziare con la mano vuota
- Inserire una carta nella giusta posizione della mano già ordinata
- Continua fino a quando tutte le carte sono inserite e ordinate

Ordinamento per inserzione

- Iniziare con la mano vuota
- Inserire una carta nella giusta posizione della mano già ordinata
- Continua fino a quando tutte le carte sono inserite e ordinate

Ordinamento per inserzione

- Iniziare con la mano vuota
- Inserire una carta nella giusta posizione della mano già ordinata
- Continua fino a quando tutte le carte sono inserite e ordinate

```
function insertionsort!(A)
    for j=2:length(A)
        rec = A[j]
        i = j-1
        while i>0 && A[i].key > rec.key
            A[i+1] = A[i]
            i = i-1
        end
        A[i+1] = rec
    end
    return A
end
```

5	1	6	2	4	3
1	5	6	2	4	3
1	5	6	2	4	3
1	2	5	6	4	3
1	2	4	5	6	3
1	2	3	4	5	6

Struttura dati: esempio

Applichiamo l'algoritmo `insertionsort!` all'array `table1`.

```
julia> insertionsort!(table1)   # l'array di record viene ordinato
4-element Array{record,1}:
 record(101, "adalberto", "via d.manin", "mnndnl123")
 record(133, "alberto", "via g.garibaldi", "grbgsp789")
 record(232, "roberto", "via c.cavour", "cvrcml456")
 record(567, "gilberto", "via p.albertelli", "lbrtpl012")
```

Struttura dati: esempio

Applichiamo l'algoritmo `insertionsort!` all'array `table1`.

```
julia> insertionsort!(table1)   # l'array di record viene ordinato
4-element Array{record,1}:
 record(101, "adalberto", "via d.manin", "mnndnl123")
 record(133, "alberto", "via g.garibaldi", "grbgsp789")
 record(232, "roberto", "via c.cavour", "cvrcml456")
 record(567, "gilberto", "via p.albertelli", "lbrtpl012")
```

- I record dopo l'ordinamento con `insertionsort!` ne mostrano gli effetti sull'array.

Struttura dati: esempio

Applichiamo l'algoritmo `insertionsort!` all'array `table1`.

```
julia> insertionsort!(table1)   # l'array di record viene ordinato
4-element Array{record,1}:
 record(101, "adalberto", "via d.manin", "mnndnl123")
 record(133, "alberto", "via g.garibaldi", "grbgsp789")
 record(232, "roberto", "via c.cavour", "cvrcml456")
 record(567, "gilberto", "via p.albertelli", "lbrtpl012")
```

- I record dopo l'ordinamento con `insertionsort!` ne mostrano gli effetti sull'array.
- Notare l'identificatore della funzione, che termina con il carattere `!`.

Ordinamento di un array di interi

```
function insertionsort!(A)
    println(A) # stampa l'input
    for j=2:length(A)
        pivot = A[j]
        i = j-1
        while i>0 && A[i] > pivot
            A[i+1] = A[i]
            i = i-1
        end
        A[i+1] = pivot
        # stampa dopo ogni scambio
        println(A)
    end
    return A
end
```

```
julia> using Random

julia> A = Random.randperm(9)'
1×9 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:
 8  1  7  4  2  3  5  6  9

julia> insertionsort!(A)
[8 1 7 4 2 3 5 6 9]
[1 8 7 4 2 3 5 6 9]
[1 7 8 4 2 3 5 6 9]
[1 4 7 8 2 3 5 6 9]
[1 2 4 7 8 3 5 6 9]
[1 2 3 4 7 8 5 6 9]
[1 2 3 4 5 7 8 6 9]
[1 2 3 4 5 6 7 8 9]
[1 2 3 4 5 6 7 8 9]
[1 2 3 4 5 6 7 8 9]
1×9 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:
 1  2  3  4  5  6  7  8  9
```

Section 3

Analisi di complessità

Analisi di complessità

Ci sono $n-1$ iterazioni del ciclo. Ad ogni iterazione vengono spostati in avanti tutti i termini dal punto di inserzione fino al pivot.

Nel caso peggiore (pivot ultimo) $n-1$ termini. L'esecuzione peggiore dell'algoritmo su dati di misura n richiede $(n-1)+(n-2)+\cdot+1$ spostamenti (scambi).

L'algoritmo equivale al calcolo di una permutazione di n elementi attraverso scambi a partire dall'identità.

Quindi:

Complessità asintotica:

$$T(n) = n\,(\frac{n-1}{2} + 3) = O(n^2).$$

Section 4

Ordinamento per selezione (selection sort)

Ordinamento per selezione

Ingresso Un array A non ordinato sui valori della chiave

Ordinamento per selezione

Ingresso Un array A non ordinato sui valori della chiave

Uscita Lo stesso array A ordinato sui valori della chiave

Ordinamento per selezione

Ingresso Un array A non ordinato sui valori della chiave

Uscita Lo stesso array A ordinato sui valori della chiave

Metodo Ad ogni iterazione viene selezionato un elemento (in posizione `imin`) e posto nella sua posizione finale `i`.

Ordinamento per selezione

Ingresso Un array A non ordinato sui valori della chiave

Uscita Lo stesso array A ordinato sui valori della chiave

Metodo Ad ogni iterazione viene selezionato un elemento (in posizione `imin`) e posto nella sua posizione finale `i`.

Primo raffinamento

Ordinamento per selezione

Ingresso Un array A non ordinato sui valori della chiave

Uscita Lo stesso array A ordinato sui valori della chiave

Metodo Ad ogni iterazione viene selezionato un elemento (in posizione `imin`) e posto nella sua posizione finale `i`.

Primo raffinamento

Ordinamento per selezione

Ingresso Un array A non ordinato sui valori della chiave

Uscita Lo stesso array A ordinato sui valori della chiave

Metodo Ad ogni iterazione viene selezionato un elemento (in posizione `imin`) e posto nella sua posizione finale `i`.

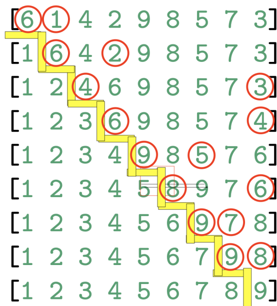
Primo raffinamento

```
function selectionsort!(A)
    n = length(A)
    for i=1:n-1
        # ricerca min [A[i], A[n]]
        # scambia A[i] e A[imin])
    end
    return A
end
```

Esempio: Ordinamento per selezione

Osserviamo il funzionamento dell'algoritmo confrontando la posizione delle chiavi nei passi successivi del ciclo più esterno (di indice i)

```
julia> A = Random.randperm(9)'
1×9 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:
 6  1  4  2  9  8  5  7  3

julia> selectionsort!(A)
1×9 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:
 1  2  3  4  5  6  7  8  9
```



Ordinamento per selezione

```
function selectionsort!(A)
    println(A)
    n = length(A)
    for i=1:n-1
        # ricerca min{A[i], A[n]}
        imin = i
        for j=i+1:n
            if A[j] < A[imin]
                imin = j
            end
        end
        # scambia A[i] e A[imin]
        A[i], A[imin] = A[imin], A[i]
        println(A)
    end
    return A
end
```

Section 5

Analisi di complessità

Analisi di complessità

L'istruzione dominante

```
if A[j] < A[imin]
    imin = j
end
```

viene eseguita (nel caso peggiore) $n-1$ volte quando i vale 1; $n-2$ volte quando i vale 2; $n-(n-1)=1$ volte quando i vale $n-1$.

Sommando si avrà

$$d(n) = (n-1) + (n-2) + \cdots 2 + 1 = n(n-1)/2$$

Pertanto avremo, sulla base delle ipotesi assunte nel nostro modello di costo di un programma:

$$T(n) = c_1(n(n-1)/2) + c_2$$

Quindi la complessità del `selectionsort` è $O(n^2)$.

Section 6

Ordinamento a bolle (Bubble Sort)

Ordinamento a bolle

Si dice “Bubble sort” perchè il funzionamento somiglia all’emergere di bolle d’aria dal fondo di un liquido.

Ad ogni passo almeno un elemento “leggero” si sposta raggiungendo la sua posizione finale.

Lo spostamento può coinvolgere contemporaneamente numerosi elementi, che si avvicinano alla posizione finale, consentendo spesso all’algoritmo di terminare prima che nel caso peggiore

Ordinamento a bolle

Metodo Ad ogni iterazione viene “fatto emergere” almeno un elemento, che raggiunge la sua posizione finale.

```
function bubblesort!(A)
    n = length(A)
    println(A)
    for _=2:n # ripeti finche' emergono elementi
        for j=1:n-1 # eventuale scambio di A[j] e A[j-1]
            if A[j] > A[j+1]
                A[j], A[j+1] = A[j+1], A[j]
                println(A)
            end
        end
    end
    return A
end
```

Bubble Sort (Julia idiomatic version)

from https://rosettacode.org/wiki/Sorting_algorithms/Bubble_sort#Julia

```
julia> function bubblesort!(arr::AbstractVector)
           for _ in 2:length(arr), j in 1:length(arr)-1
               if arr[j] > arr[j+1]
                   arr[j], arr[j+1] = arr[j+1], arr[j]
               end
           end
           return arr
       end
bubblesort! (generic function with 2 methods)

julia> v = rand(-10:10, 7)'
1×10 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:
 -5, -1, -1, -2, -1, 1, 0, -10, -9, 2

julia> println("# unordered: $v\n -> ordered: ", bubblesort!(v))
# unordered: [-5, -1, -1, -2, -1, 1, 0, -10, -9, 2]
 -> ordered: [-10, -9, -5, -2, -1, -1, -1, 0, 1, 2]
```

Section 7

Esempio

Esempio 1/2

Si noti come in ogni passo vari elementi si spostino avvicinando la loro posizione finale.

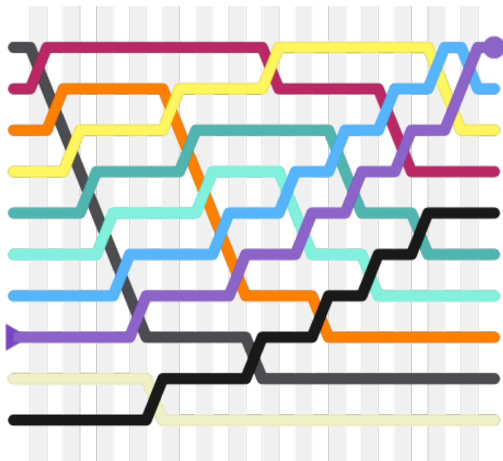
```
julia> v = [1 2 3 4 5 6]
1×6 Array{Int64,2}:
 1  2  3  4  5  6

julia> bubblesort!(v)
[1 2 3 4 5 6]
1×6 Array{Int64,2}:
 1  2  3  4  5  6

julia> w = rand(-10:10, 7)'
1×7 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:
 -8  5  10  -4  -7  9  0
```

```
julia> bubblesort!(w);
[-8 5 10 -4 -7 9 0]
[-8 5 -4 10 -7 9 0]
[-8 5 -4 -7 10 9 0]
[-8 5 -4 -7 9 10 0]
[-8 5 -4 -7 9 0 10]
[-8 -4 5 -7 9 0 10]
[-8 -4 -7 5 9 0 10]
[-8 -4 -7 5 0 9 10]
[-8 -7 -4 5 0 9 10]
[-8 -7 -4 0 5 9 10]
```

Quando l'input è ordinato (caso migliore) la complessità è $\Omega(n)$!!

Esempio 2/2

Una visualizzazione statica del Bubble Sort



Section 8

Analisi di complessità.

Analisi di complessità

La complessità dipende dai dati di ingresso (è una funzione di n).

Una sola iterazione e' sufficiente quando i dati sono già ordinati. In questo caso la complessità di tempo è $\Omega(n)$, comunque necessario per lo I/O di n dati.

Il caso peggiore si verifica quando i dati sono immessi nell'ordinamento opposto. Nel caso peggiore sono necessarie numerose iterazioni; in ognuna emerge un elemento e tutti i precedenti vengono traslati di una posizione.

Il numero dei confronti è:

$n-1$ nella prima iterazione;
$n-2$ nella seconda iterazione;
... ...
$n-i$ nella i-esima iterazione;

Quindi il numero totale di confronti è $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}=O(n^2)$.
Il caso medio ha la stessa complessità.

Esempi di esecuzione

Nell'ordine si vedono, per varie esecuzioni del programma bubblesort!, gli array dei valori (input) immessi dall'utente, e le stampe delle chiavi non ordinate e ordinate

```
julia> u = [25 -22 0 16 88 24 -12 47 3 16];

julia> println("# unordered: $u\n -> ordered: ", bubblesort!(u))
# unordered: [25 -22 0 16 88 24 -12 47 3 16]
 -> ordered: [-22 -12 0 3 16 16 24 25 47 88]

julia> v = [55 12 -13 48 26 84 -3 -9811 -13 48 26];

julia> println("# unordered: $v\n -> ordered: ", bubblesort!(v))
# unordered: [55 12 -13 48 26 84 -3 -9811 -13 48 26]
 -> ordered: [-9811 -13 -13 -3 12 26 26 48 48 55 84]

julia> w = [19 1817 -16 -255 1413 1211 10 -22 346 1 0];

julia> println("# unordered: $w\n -> ordered: ", bubblesort!(w))
# unordered: [19 1817 -16 -255 1413 1211 10 -22 346 1 0]
 -> ordered: [-255 -22 -16 0 1 10 19 346 1211 1413 1817]
```