Strutture Dati e Algortimi

Algoritmi di ordinamento

Luca Di Gaspero, Università degli Studi di Udine

Il problema dell'ordinamento

Data una **sequenza** di n elementi e una loro relazione d'ordine $\leq$, disporre gli elementi **nell'array** in modo che risultino ordinati secondo la relazione $\leq$

Alcuni commenti:

- La relazione d'ordine non è necessariamente quella crescente e dipende dal tipo di dato contenuto nel vettore, per ora considereremo interi e la relazione $\leq$ su di essi
- La sequenza non è necessariamente contenuta in un array (ipotesi necessaria ora per le vostre conoscenze attuali)

Approccio genera e verifica

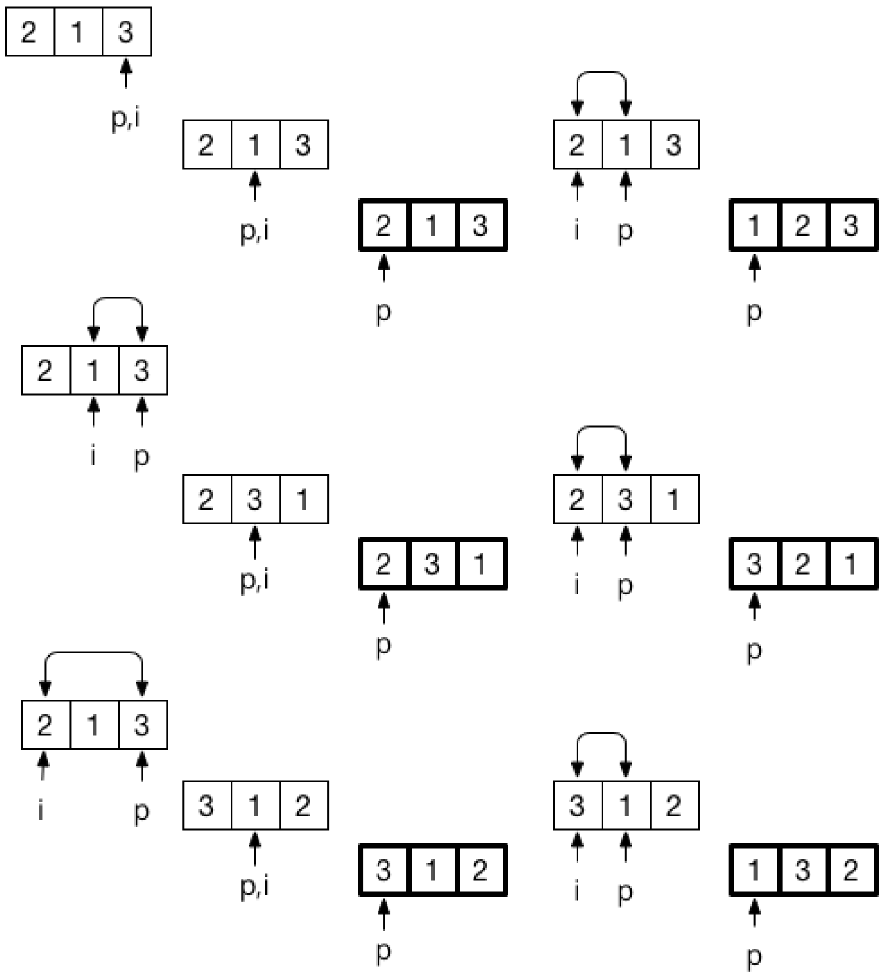
L'algoritmo *bovino* per l'ordinamento è quello che genera ogni possible permutazione dei valori dell'array e verifica se quella permutazione è ordinata

- Le permutazioni di un vettore di n elementi si possono ottenere calcolando le permutazioni del sottovettore di $n-1$ elementi e poi scambiando l'n-esimo elemento con ogni elemento del sottovettore

Generazione delle permutazioni di un vettore

Definendo p la posizione corrente, inzialmente pari all'indice dell'ultimo elemento del vettore

- Per $i = p-1, \ldots, 1, 0$:
  - scambia ciascun $a[i]$ con l'elemento correntemente in ultima posizione (ovvero $a[p]$)
  - **permuta ricorsivamente** i primi $p-1$ elementi di a nello stesso modo
  - scambia nuovamente l'ultimo elemento $a[p]$ con $a[i]$
    (per rimetterli a posto e provare altre permutazioni)




```c
bool GeneraPermutazioni(int a[], int n, int p) {
  int i;
  if (p == 0)
    // la permutazione è completa e va verificata
    return VerificaOrdinamento(a, n);
  else {
    for (i = p; i >= 0; i--) {
      // prova tutti gli scambi dell'elemento p
      Scambia(&a[i], &a[p]);
      // genera ricorsivamente tutte
      // le permutazioni sul sottovettore a sinistra
      if (GeneraPermutazioni(a, n, p - 1))
        // se la funzione ha trovato un ordinamento
        // termina l'esecuzione (e le chiamate ricorsive)
        return true;
      Scambia(&a[i], &a[p]);
    }
    // nessuna delle permutazioni generate era ordinata
    return false;
  }
}

bool VerificaOrdinamento(int a[], int n) {
  int i;
  // verifico che tutti gli elementi siano in ordine crescente
  for (i = 0; i < n - 1; i++)
    if (a[i] > a[i + 1])
      return false;
  return true;
}

// chiamata iniziale: GeneraPermutazioni(a, n, n - 1)
```

Esecuzione su PythonTutor

Generazione delle permutazioni di un vettore

La complessità dell'algoritmo, nel caso pessimo, è data dalle chiamate ricorsive:

$$T(n,p) = \begin{cases} O(n) & \text{se}\, p < 1 \\ p \cdot \big(T(n, p-1) + O(1)\big) & \text{se}\, p \geq 1 \end{cases}$$

- Dunque, per la chiamata originaria, usando il metodo di espansione:

$$T(n, n-1) = \overbrace{(n-1)}^{p}\big(T(n, \overbrace{n-2}^{p-1}) + \overbrace{c}^{O(1)}\big) = (n-1) \cdot \Big((n-2) \cdot \big(T(n, n-3) + c\big)\Big) =$$

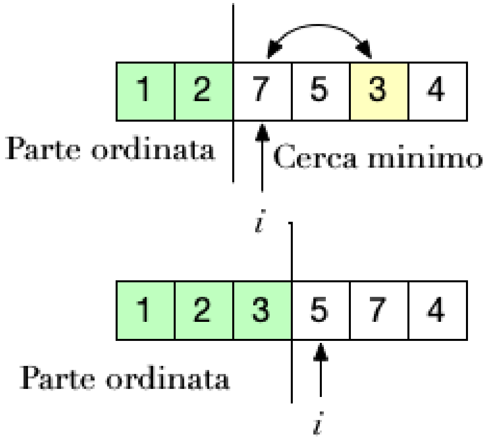
$$= (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 1 \cdot (\overbrace{T(n,0)}^{O(n)} + c)$$
$$= (n-1)! \cdot (cn + c) = (n-1)! \cdot (n+1) \cdot c \geq O(n!) \equiv O(n^n)$$

Anche intuitivamente, nel caso peggiore è necessario generare tutte le $n!$ permutazioni

Approssimazione di Stirling $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n = O(n^n)$

Selection sort

Idea: al passo i seleziona l'elemento di rango $(i+1)$ ossia il minimo tra i rimanenti $n-i$ elementi e scambialo con l'elemento in posizione i



Nota: il rango di un elemento del vettore è costituito dal numero di elementi più piccoli di esso che corrisponde alla sua posizione nel vettore ordinato

```
void SelectionSort(int a[], int n) {
    int i, indice_minimo;
    for (i = 0; i < n - 1; i++) {
        indice_minimo = MinimoAPartireDa(a, n, i);
        Scambia(&a[i], &a[indice_minimo]);
    }
}

int MinimoAPartireDa(int a[], int n, int i) {
    int j, m = i;
    for (j = i + 1; j < n; j++) {
        if (a[j] <  a[m])
            m = j;
    }
    return m;
}
```

A meno di componenti di costo costante, al passo i-esimo il costo del corpo del `for` è pari al costo $t(i, n)$ della chiamata della funzione `MinimoAPartireDa()`

- esso non è costante ma dipende anche da i (oltre che da n)
- dunque il costo totale è pari a $\sum_{i=0}^{n-2} t(i, n) + O(1)$

```
int MinimoAPartireDa(int a[], int n, int i) {
    int j, m = i;
    for (j = i + 1; j < n; j++) {
        if (a[j] <  a[m])
            m = j;
    }
    return m;
}
```

Il costo della funzione $t(i, n)$ in dipendenza di i è proporzionale al numero di iterazioni del ciclo (più l'assegnamento esterno) quindi $t(i, n) = O(n - i)$

Pertanto:

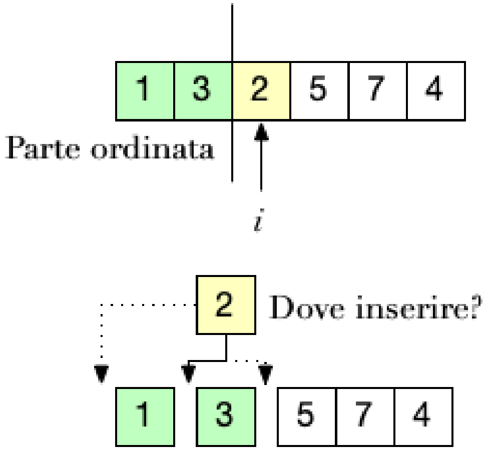
$$T(n) = \sum_{i=0}^{n-2} O(n - i) + O(1)$$

$$= O(\sum_{i=0}^{n-2} n - i) = O(\sum_{i=1}^{n-1} i) = O(n^2)$$

A causa del calcolo del minimo fra gli elementi rimasti anche la complessità nel caso migliore è $O(n^2)$ (e quindi anche nel caso medio)

Insertion Sort

Idea: al passo i-esimo inserisci l'elemento in posizione i al posto giusto tra i primi i elementi (già ordinati)



```
void InsertionSort(int a[], int n) {
  int i, j, prossimo;
  for (i = 1; i < n; i++) {
    // estrae il prossimo elemento da inserire
    prossimo = a[i];
    // j è la posizione candidata all'inserimento
    j = i;
    // verifica se la posizione corrente
    // è qeuella giusta
    while (j > 0 && a[j - 1] > prossimo) {
      // altrimenti fai spazio
      a[j] = a[j - 1];
      j = j - 1;
    }
    a[j] = prossimo;
  }
}
```

Per poter inserire in un qualunque punto fra gli elementi già ordinati devo fare spazio (non posso modificare un vettore creando un elemento in un punto qualunque)

Il ciclo `while`, se necessario, sposta gli elementi verso destra per fare spazio al prossimo elemento da inserire

Insertion Sort

Al passo i del `for` esterno il costo è, praticamente, dominato dal costo $t(i)$ del ciclo `while` interno

- Il ciclo `while` richiede al massimo $i + 1$ iterazioni, ciascuna di costo costante
  - $t(i) = O(i + 1)$ per il ciclo `while`

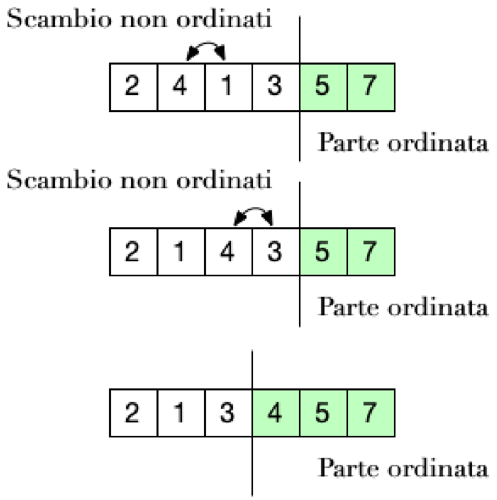
    In totale: $\sum_{i=0}^{n-1} O(i + 1) = O(\sum_{i=0}^{n-1} i + 1) = O(\frac{n(n+1)}{2}) = O(n^2)$

Osservazione: l'algoritmo richiede solo $O(n)$ operazioni quando l'array è già ordinato

- In generale, è possibile provare che l'algoritmo richiede tempo $O(nk)$ se ciascun elemento si trova al più a distanza k dalla sua posizione nell'array ordinato (quindi parecchio efficiente per array quasi ordinati)

Bubble Sort

Idea: confrontare gli elementi a coppie e fare salire i valori più grandi verso la fine dell'array (e scendere quelli più piccoli verso l'inizio)



```
void BubbleSort(int a[], int n) {
  int i, k = n - 1;
  bool scambio = true;
  while (scambio) {
    scambio = false;
    for (i = 0; i < k; i++)
      if (a[i] > a[i + 1]) {
        Scambia(&a[i], &a[i + 1])
        scambio = true;
      }
    k = k - 1;
  }
}
```

All'iterazione $i = 1, \ldots, n$ del ciclo `while` l'elemento di posizione $n - i = k$ sarà salito nella sua posizione definitiva

- la porzione di vettore compresa fra gli indici k e $n - 1$ è ordinata
- Al passo k del ciclo `while` il costo del suo corpo $t(k)$ è dominato dal ciclo for interno che richiede tempo $O(k)$

Bubble Sort

Il corpo del ciclo `while` può essere eseguito al più n volte (per $k = n-1, \ldots, 0$) quindi in totale

$$T(n) = \sum_{k=0}^{n-1} t(k) = \sum_{k=0}^{n-1} O(k) = O(\sum_{k=0}^{n-1} k) = O(\frac{n(n-1)}{2}) = O(n^2)$$

Nel caso di un array già ordinato, il numero di operazioni svolte è $O(n)$, corrispondenti all'esecuzione del ciclo `for` che determina che nessuno scambio è necessario e in tal caso il ciclo `while` viene eseguito una volta sola.

Riepilogo degli algoritmi visti

Algoritmo	Idea	Caso Pessimo	Caso Ottimo
Genera e verifica	Genero tutte le permutazioni e verifico quale di esse corrisponde al vettore ordinato	$O(n!)$	$O(n)$
Selection sort	Cerco (seleziono) l'elemento minimo fra quelli rimasti da ordinare e lo scambio con l'elemento corrente	$O(n^2)$	$O(n^2)$
Insertion sort	Cerco di inserire l'elemento corrente fra quelli precedenti, già ordinati	$O(n^2)$	$O(n)$
Bubble sort	Effettuo più passaggi facendo *affiorare* gli elementi più grandi, finché non sono necessari più scambi	$O(n^2)$	$O(n)$

- È possibile fare di meglio?

Limite inferiore di complessità dell'ordinamento

Qual è, nel caso peggiore, il **numero minimo di operazioni** richieste da un qualunque algoritmo di ordinamento basato sul confronto di elementi?

- Il cuore degli algoritmi di ordinamento sono le operazioni di confronto, contiamo quindi tali operazioni
- Consideriamo un qualunque algoritmo di ordinamento $\mathcal{A}$ che usa confronti tra coppie di elementi

Limite inferiore di complessità dell'ordinamento

In t confronti (*passi, operazioni*), $\mathcal{A}$ può discernere al più 2^t situazioni distinte:

- sono infatti possibili due risposte per ogni confronto: $a_i \le a_j$, oppure $a_i > a_j$

Il numero di possibili ordinamenti di n elementi è $n!$ (tutte le loro permutazioni)

Poiché $\mathcal{A}$ deve discernere tra $n!$ possibili situazioni, deve valere $2^t \ge n!$, pertanto, risolvendo in t:

$$t = \log 2^t \ge \log \underbrace{n!}_{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = \log O(n^n) = O(\log n^n) = O(n \log n)$$

Dunque $t = \Omega(n \log n)$ (perché $t \ge O(f(n)) \Rightarrow t = \Omega(f(n))$) è un limite inferiore alla complessità di qualunque algoritmo di ordinamento basato sui confronti