Algoritmi (modulo di laboratorio)

Corso di Laurea in Matematica

Roberto Cordone

DI - Università degli Studi di Milano

Lezioni:	Martedì 8.30 - 10.30 in aula 8 e Z4	Mercoledì 10.30 - 13.30 in aula 2 e Z4
	Giovedì 15.30 - 18.30 in aula 2 e Z4	Venerdì 10.30 - 12.30 in aula 400 e Z4
Ricevimento:	su appuntamento (Dipartimento di Informatica)	
E-mail:	roberto.cordone@unimi.it	
Pagina web:	http://homes.di.unimi.it/~cordone/courses/2022-algo/2022-algo.html	
Sito Ariel:	https://mgoldwurmasd.ariel.ctu.unimi.it	

Lezione 6: Tabelle e algoritmi di ordinamento quadratici Milano, A.A. 2021/22

Tabelle: struttura dati astratta

I vettori hanno una dimensione fissata una volta per tutte

Spesso occorre raccogliere un numero di informazioni

- non noto a priori
- variabile durante l'elaborazione

ma di cui si conosce una stima per eccesso

Una tabella T di dimensione k su un insieme U è definita come una n-upla ordinata $(v_1, \ldots, v_n)$ di elementi di U con $n \in \{0, \ldots, k\}$

La tabella ha una cardinalità n scelta a piacere e può anche essere vuota

La struttura dati astratta è definita come

- l'insieme $\mathcal{T}_{k,U}$ di tutte le possibili tabelle di dimensione k su U

$$\mathcal{T}_{k,U} = \bigcup_{n=0}^{k} U^n$$

Tabelle: operazioni

Le tabelle

- ammettono le operazioni di proiezione $\pi_i(T)$ e sostituzione $\sigma_i(T, u)$ ma occorre verificare che l'indice i sia in $\{1, \ldots, n\}$
- possono ammettere altre operazioni:
  - cardinalità $\mathrm{card}(T)$ che associa a una tabella il numero di elementi

$$\mathrm{card} : \mathcal{T}_{k,U} \to \{0, \ldots, k\}$$

  - inserimento $\mathrm{ins}(T, u)$ che associa a una tabella e a un elemento la tabella ottenuta aggiungendo l'elemento in posizione terminale

$$\mathrm{ins} : \mathcal{T}_{k,U} \times U \to \mathcal{T}_{k,U}$$

Occorre verificare che la cardinalità n non ecceda la soglia k

  - cancellazione $\mathrm{canc}(T, i)$ che associa a una tabella e a un indice la tabella ottenuta cancellando l'elemento associato all'indice

$$\mathrm{canc} : \mathcal{T}_{k,U} \times \{1, \ldots, k\} \to \mathcal{T}_{k,U}$$

Occorre verificare che l'indice i non ecceda la cardinalità n

Tabelle: implementazione in C

In C una tabella si può realizzare con una struttura contenente

- un vettore di k elementi di tipo U
- il valore intero k, che rappresenta la dimensione allocata, costante
- il valore intero n, che rappresenta la cardinalità, variabile

Una tabella T di oggetti di tipo U si dichiara come segue:

```
struct _tabella {
   U *V;
   int k;
   int n;
};
typedef struct _tabella tabella;

tabella T;
```

Per poterla usare, non bisogna dimenticare le procedure per la

- creazione, cioè per l'allocazione del campo `V`
- distruzione, cioè per la deallocazione del campo `V`

In realtà spesso si tengono i tre dati separati senza accorparli in un record

Tabelle: costi delle operazioni

Il costo spaziale della tabella è ovviamente lineare in k ($\Theta(k)$)

I costi temporali delle operazioni sono tutti costanti

- per la cardinalità, si restituisce il valore di n

```
int card (tabella *pT)
{
  return pT->n;
}
```

- per l'inserimento, si incrementa n e si assegna l'elemento

```
void ins (tabella *pT, U u)
{
  if (pT->n >= pT->k) exit(EXIT_FAILURE);
  (pT->n)++;
  pT->V[pT->n] = u;
}
```

Passiamo la tabella per indirizzo solo per efficienza (non è necessario)

Tabelle: costi delle operazioni

I costi temporali delle operazioni sono tutti costanti

- per la cancellazione, si sovrascrive l'elemento indicato con l'ultimo e si decrementa *n*

```
void canc (tabella *pT, int i)
{
  if ( (i <= 0) || (i > pT->n) ) exit(EXIT_FAILURE);
  pT->V[i] = pT->V[pT->n];
  (pT->n)--;
}
```

L'implementazione assume che l'ordine degli elementi non sia fissato

Se l'ordine va conservato, la cancellazione passa da $\Theta(1)$ a $\Theta(k)$ perché si scalano un passo indietro gli elementi che seguono quello cancellato

Tabelle: implementazione come vettori con terminatore

Un'implementazione alternativa (poco usata) impiega

- un vettore di $k + 1$ elementi di tipo U
- un valore intero k, che rappresenta la dimensione allocata, costante
- un terminatore, cioè un elemento esterno ad U che non rappresenta un'informazione effettiva, ma indica il termine della tabella

Si risparmia l'intero n, ma si spende lo spazio occupato dal terminatore

Gli svantaggi sono:

- non si può usare il terminatore come informazione effettiva
- cardinalità, inserimento e cancellazione richiedono tempo lineare, perché richiedono di individuare il terminatore scorrendo la tabella

E allora perché ne parliamo?

Stringhe: implementazione in C

In C, le stringhe sono rappresentate come

- vettori di caratteri (`char s[N+1];`)
- terminati dal carattere *null* (`'\0'`), detto terminatore, il quale ha codifica binaria interamente nulla

Se la stringa `s` vale "`pro`", significa che contiene 4 caratteri:

- `s[0]` vale `'p'`
- `s[1]` vale `'r'`
- `s[2]` vale `'o'`
- `s[3]` vale `'\0'`

anche se lo spazio allocato è più lungo:

`'p'`	`'r'`	`'o'`	`'\0'`	`'v'`	`'a'`	`'\0'`	`vale ''pro''`
0	1	2	3	4	5	6	

Non occorre specificare la dimensione di una stringa: un vettore di `N+1` caratteri può rappresentare stringhe di qualsiasi dimensione da 0 a `N`
Però non c'è controllo che una stringa contenga il carattere `'\0'`

Relazioni d'ordine

Un preordine su un insieme U è una relazione binaria $\preceq$ su U che gode delle proprietà

1. riflessiva: $u \preceq u$ per ogni $u \in U$
2. transitiva: se $u_1 \preceq u_2$ e $u_2 \preceq u_3$, allora $u_1 \preceq u_3$ per ogni $u_1, u_2, u_3 \in U$

Una relazione d'ordine parziale è un preordine che gode della proprietà

- antisimmetrica: se $u_1 \preceq u_2$ e $u_2 \preceq u_1$, allora $u_1 = u_2$ per ogni $u_1, u_2 \in U$

Una relazione d'ordine debole è un preordine che gode della proprietà

- di completezza: se $u_1 \not\preceq u_2$, allora $u_2 \preceq u_1$ per ogni $u_1, u_2 \in U$

Una relazione d'ordine totale è un preordine che gode di ambo le proprietà

Il problema dell'ordinamento

Sia U un insieme dotato di un ordine debole $\preceq$ *(si ammettono ex-aequo)*

Il problema dell'ordinamento ha come

- istanza: qualsiasi vettore V su U
- soluzione: un vettore V' permutazione di V tale che

$$V[i] \preceq V[j] \text{ per ogni } i \leq j$$

Esempio:

$V =$

5	2	8	4	7	1	3	6

$V' =$

1	2	3	4	5	6	7	8

Ordinamento per inserimento

Convenzione: dato un vettore V, indichiamo con
$V[s, d]$ il sottovettore degli elementi di V con indici compresi fra s e d

InsertionSort gestisce la soluzione come una tabella ordinata T

- inizialmente T contiene solo il primo elemento di V
- ogni elemento $V[j]$ (con $j = 2, \ldots, n$) viene inserito in T in ordine:
  - scalando gli elementi $> V[j]$ nella posizione di indice successivo
  - inserendo $V[j]$ nella posizione liberata

La tabella T viene rappresentata con il sottovettore $V[1, j-1]$

- gli elementi vanno scalati partendo da $V[j-1]$ per j decrescenti
  (altrimenti ognuno cancellerebbe il successivo)
- bisogna salvare $V[j]$ a parte per prima cosa
  (altrimenti $V[j-1]$ lo cancellerebbe)

InsertionSort: pseudocodice ed esempio

```
InsertionSort(V,n)
{
  for (j = 2; j <= n; j++)
  {
    x = V[j];
    InserisceOrdinato(x,V,j-1);
  }
}


InserisceOrdinato(x,V,n)
{
  for (i = n; (i > 0)&&(V[i] > x); i--)
    V[i+1] = V[i];
  V[i+1] = x;
}
```

5	2	8	4	7	1	3	6

$x = V[2] \Rightarrow x = 2$

5	x	8	4	7	1	3	6
x	5	8	4	7	1	3	6
2	5	8	4	7	1	3	6

$x = V[3] \Rightarrow x = 8$

2	5	x	4	7	1	3	6
2	5	x	4	7	1	3	6
2	5	8	4	7	1	3	6

$x = V[4] \Rightarrow x = 4$

2	5	8	x	7	1	3	6
2	x	5	8	7	1	3	6
2	4	5	8	7	1	3	6

...

InsertionSort: pseudocodice ed esempio

```
InsertionSort(V,n)
{
  for (j = 2; j <= n; j++)
  {
    x = V[j];
    InserisceOrdinato(x,V,j-1);
  }
}
```

```
InserisceOrdinato(x,V,n)
{
  for (i = n; (i > 0)&&(V[i] > x); i--)
    V[i+1] = V[i];
  V[i+1] = x;
}
```

2	4	5	8	7	1	3	6

$x = V[5] \Rightarrow x = 7$

2	4	5	8	x	1	3	6
2	4	5	x	8	1	3	6
2	4	5	7	8	1	3	6

$x = V[6] \Rightarrow x = 1$

2	4	5	7	8	x	3	6
x	2	4	5	7	8	3	6
1	2	4	5	7	8	3	6

$x = V[7] \Rightarrow x = 3$

1	2	4	5	7	8	x	6
1	2	x	4	5	7	8	6
1	2	3	4	5	7	8	6

...

InsertionSort: pseudocodice ed esempio

```
InsertionSort(V,n)
{
  for (j = 2; j <= n; j++)
  {
    x = V[j];
    InserisceOrdinato(x,V,j-1);
  }
}
```

1	2	3	4	5	7	8	6

$x = V[8] \Rightarrow x = 6$

1	2	3	4	5	7	8	x
1	2	3	4	5	x	7	8
1	2	3	4	5	6	7	8

```
InserisceOrdinato(x,V,n)
{
  for (i = n; (i > 0)&&(V[i] > x); i--)
    V[i+1] = V[i];
  V[i+1] = x;
}
```

InsertionSort: correttezza

L'algoritmo funziona per induzione matematica

- Al principio, $j = 2$ e la tabella $T = V[1, j-1] = V[1, 1]$
  1. è in corrispondenza biunivoca con un sottoinsieme di $j-1$ elementi di V
  2. è ordinata
- Ad ogni iterazione, j cresce di 1 e la tabella T
  1. include un nuovo elemento di V
  2. lo inserisce in posizione ordinata

Dunque conserva le due proprietà

Al termine, $j = n+1$ e la tabella $T = V[1, j-1] = V[1, n]$

- è in corrispondenza biunivoca con un sottoinsieme di $j-1 = n$ elementi di V (cioè tutti)
- è ordinata

In altre parole, per qualsiasi n e per qualsiasi vettore V,
al termine dell'algoritmo la tabella T è una permutazione ordinata di V

InsertionSort: complessità

```
InsertionSort(V,n)
{
  for (j = 2; j <= n; j++)
  {
    x = V[j];
    InserisceOrdinato(x,V,j-1);
  }
}
```

$\sum_{j=2}^{n} (\ldots)$

$\Theta(1)$

$f(j)$

```
InserisceOrdinato(x,V,n)
{
  for (i = n; (i > 0)&&(V[i] > x); i--)
    V[i+1] = V[i];
  V[i+1] = x;
}
```

$f(n) = \ldots \qquad (\text{con } n = j - 1)$

$\sum_{i=p_x}^{n} (\ldots)$

$\Theta(1)$

$\Theta(1)$

con p_x = indice finale di x in V

InsertionSort: complessità

Riassumendo la precedente analisi dettagliata

$$T(n) = \sum_{j=2}^{n} \left(\Theta(1) + f(j)\right) = \sum_{j=2}^{n} \left(\Theta(1) + \sum_{i=p_x(j)}^{j-1} \Theta(1) + \Theta(1) \right)$$

da cui

$$T(n) \in \Theta\left(\sum_{j=2}^{n} 1 + \sum_{j=2}^{n} (j - p_x(j)) \right)$$

Ha costanti asintotiche piccole: è l'algoritmo migliore per istanze piccole

Per istanze grandi, la complessità dipende dal valore (incognito) di p_x

- caso pessimo, cioè $p_x(j) = 1$ sempre: $T(n) \in \Theta(n^2)$
- caso medio (per opportune distribuzioni): $T(n) \in \Theta(n^2)$
- caso ottimo, cioè $p_x(j) = j$ sempre: $T(n) \in \Theta(n)$

Il caso ottimo è interessante: corrisponde a vettori già ordinati (*o quasi*)

Ordinamento per selezione

SelectionSort gestisce due tabelle

- i dati non ordinati come una tabella che si svuota progressivamente
- la soluzione come una tabella T ordinata che si riempie via via

Si procede in questo modo:

- inizialmente T è vuota
- ogni passo estrae l'elemento massimo da V e lo inserisce in cima a T

Per inserire in cima, la tabella è un vettore con indice iniziale decrescente

Rappresentiamo la tabella T con il sottovettore $V[j+1,n]$ e
la tabella dei dati residui con il sottovettore $V[1,j]$

- decrementando j, si sposta l'elemento $V[j]$ da V a T
- per spostare l'elemento massimo, basta prima scambiarlo con $V[j]$

SelectionSort: pseudocodice ed esempio

```
SelectionSort(V,n)
{
  for (j = n; j > 1; j--)
  {
    i = TrovaIndiceMassimo(V,j);
    Scambia(&V[i],&V[j]);
  }
}


TrovaIndiceMassimo(V,n)
{
  iMax = 1;
  for (i = 2; i <= n; i++)
    if (V[i] > V[iMax]) iMax = i;
  return iMax;
}
```

5	2	8	4	7	1	3	6

$j = 8 \Rightarrow i = 3$

5	2	x	4	7	1	3	x
5	2	6	4	7	1	3	8

$j = 7 \Rightarrow i = 5$

5	2	6	4	x	1	x	8
5	2	6	4	3	1	7	8

$j = 6 \Rightarrow i = 3$

5	2	x	4	3	x	7	8
5	2	1	4	3	6	7	8

$j = 5 \Rightarrow i = 1$

x	2	1	4	x	6	7	8
3	2	1	4	5	6	7	8

...

SelectionSort: pseudocodice ed esempio

```
SelectionSort(V,n)
{
  for (j = n; j > 1; j--)
  {
    i = TrovaIndiceMassimo(V,j);
    Scambia(&V[i],&V[j]);
  }
}


TrovaIndiceMassimo(V,n)
{
  iMax = 1;
  for (i = 2; i <= n; i++)
    if (V[i] > V[iMax]) iMax = i;
  return iMax;
}
```

3	2	1	4	5	6	7	8

$j = 4 \Rightarrow i = 4$

3	2	1	x	5	6	7	8
3	2	1	4	5	6	7	8

$j = 3 \Rightarrow i = 1$

x	2	x	4	5	6	7	8
1	2	3	4	5	6	7	8

$j = 2 \Rightarrow i = 2$

1	x	3	4	5	6	7	8
1	2	3	4	5	6	7	8

SelectionSort: correttezza

L'algoritmo funziona per induzione matematica

- Al principio, $j = n$ e la tabella $T = V[j+1, n] = V[n+1, n]$
  1. è in corrispondenza biunivoca
     con un sottoinsieme di $n - j = 0$ elementi di V (*vuoto!*)
  2. gli elementi di T sono tutti $\geq$ agli elementi residui di V
  3. è ordinata
- Ad ogni iterazione, j cala di 1 e la tabella T
  1. include l'elemento massimo di V
  2. lo inserisce in posizione iniziale
  3. tale elemento è $\leq$ a tutti gli altri elementi di T

Dunque conserva le tre proprietà

Al termine, $j = 0$ e la tabella $T = V[j+1, n] = V[1, n]$

- è in corrispondenza biunivoca con un sottoinsieme di n elementi di V
- è ordinata

In altre parole, per qualsiasi n e per qualsiasi vettore V,
al termine dell'algoritmo la tabella T è una permutazione ordinata di V

SelectionSort: complessità

```
SelectionSort(V,n)
{
  for (j = n; j > 1; j--)
  {
    i = TrovaIndiceMassimo(V,j);
    Scambia(&V[i],&V[j]);
  }
}
```

$\sum_{j=2}^{n} (\ldots)$

$f(j)$

$\Theta(1)$

```
TrovaIndiceMassimo(V,n)
{
  iMax = 1;
  for (i = 2; i <= n; i++)
    if (V[i] > V[iMax]) iMax = i;
  return iMax;
}
```

$f(n) = \ldots$ (con $n = j$)

$\Theta(1)$

$\sum_{i=2}^{n} (\ldots)$

$\Theta(1)$

SelectionSort: complessità

Riassumendo la precedente analisi dettagliata

$$T(n) = \sum_{j=2}^{n} (f(j) + \Theta(1)) = \sum_{j=2}^{n} \left(\Theta(1) + \sum_{i=2}^{j} \Theta(1) + \Theta(1) \right)$$

da cui

$$T(n) \in \Theta\left(\sum_{j=2}^{n} 1 + \sum_{j=2}^{n} (j-1) \right) \Rightarrow T(n) \in \Theta\left(n^2\right)$$

La complessità è sempre quadratica, senza casi fortunati e sfortunati

Vedremo che una variante consente di abbattere la complessità