APA Modulo 1 Esercizi 2

Elena Zucca

7 aprile 2020

Design e analisi di algoritmi iterativi

- coffee can problem (barattolo di caffè) dovuto a David Gries
- C un barattolo con chicchi bianchi o neri, $N(C)$ = numero di chicchi
- numero illimitato di chicchi neri a disposizione fuori
- //Pre: $N(C) \geq 2$

```
while (N(C)>1)
  estrai due chicchi da C
  se hanno lo stesso colore
    eliminali e inserisci un chicco nero in C
  altrimenti
    ri-inserisci il chicco bianco in C
    elimina il nero
```

- provare che l'algoritmo termina
- a ogni passo il numero di chicchi nel barattolo diminuisce di uno, e il numero è limitato inferiormente da uno

Design e analisi di algoritmi iterativi

```
//Pre: N(C) ≥ 2
while (N(C)>1)
  estrai due chicchi da C
  se hanno lo stesso colore
    eliminali e inserisci un chicco nero in C
  altrimenti
    ri-inserisci il chicco bianco in C
    elimina il nero
```

- provare con opportuna invariante:
  ultimo chicco bianco $\Leftrightarrow$ numero chicchi bianchi iniziale dispari
- numero chicchi bianchi iniziale dispari $\Rightarrow$
  si mantiene dispari a ogni iterazione $\Rightarrow$
  ultimo chicco bianco
- analogamente numero chicchi bianchi iniziale pari $\Rightarrow$ ultimo chicco nero

Versione un po' più formale

- *odd-white*(C) = numero chicchi bianchi in C dispari
- //Pre: $N(C) \geq 2 \wedge \mathtt{odd\text{-}white}(C)$
  ```
  while (N(C)>1)
  //Inv: odd-white(C) ∧ N(C) ≥ 1
    estrai due chicchi da C
    se hanno lo stesso colore
       eliminali e inserisci un chicco nero in C
    altrimenti
      ri-inserisci il chicco bianco in C
      elimina il nero
  //Post: odd-white(C) ∧ N(C) = 1
  ```
- invariante si preserva e insieme alla condizione di uscita $(N(C) \leq 1)$ implica postcondizione
- funzione di terminazione $N(C)$ limitata inferiormente da 1 se vale *Inv*
- prova analoga se precondizione $\neg$*odd-white*(C)

Design e analisi di algoritmi iterativi

- dare algoritmo che, prese in input k liste ordinate, restituisca la "fusione" di tali liste, generalizzando *merge* di *mergesort*, in tempo $O(n \log k)$
- giustificare correttezza e complessità
- idea: heap per memorizzare tutti i primi elementi delle liste
  minimo in tempo $O(\log k)$
- assumiamo `first` restituisca il primo senza toglierlo, `getFirst` lo tolga, `add` aggiunga in fondo

Pseudocodice

```
merge(list_1 ... list_k)
  Q = heap di lunghezza k
  list = lista vuota
  for each (i in 1..k)
    if (list_i non vuota) Q.add(i,list_i.first())
  while (Q non vuoto)
    i = Q.getMin()
    list.add(list_i.getFirst())
    if (list_i non vuota) Q.add(i,list_i.first())
  return list
  Post: list ordinata, list = tutti e soli gli elementi
```

Invariante?

- `list` è ordinata
- `list` $\leq$ `list_1`, ..., `list_k` sono tutti e soli gli elementi
- `Q` $=$ primi elementi di `list_1`, ..., `list_k`

Pseudocodice con annotazioni

```
merge(list_1 ... list_k)
  Q = heap di lunghezza k
  list = lista vuota
  for each (i in 1..k)  //costo k log k
    if (list_i non vuota) Q.add(i,list_i.first())
  while (Q non vuoto) // costo n log k
    //INV: list ordinata
    //list<=list_1 ...  list_k = tutti e soli gli elementi
    //Q = primi elementi di list_1 ...  list_k
    i = Q.getMin()
    list.add(list_i.getFirst())
    if (list_i non vuota) Q.add(i,list_i.first())
  return list
  //Post:  list ordinata, list = tutti e soli gli elementi
```

Correttezza

- invariante vale all'inizio: `list` è vuota quindi ordinata, tutti i primi elementi delle liste (se esistenti) sono in `Q`
- si mantiene: spostiamo il minimo degli elementi in `Q` (minimo dei primi elementi di `list_1 ... list_k`) in `list`
  spostiamo in `Q` il nuovo primo elemento (se esistente) della lista di cui questo minimo era il primo elemento
- invariante + condizione di uscita implica postcondizione: se `Q` vuoto `list_1 ... list_k` non hanno primo elemento, ossia sono vuote

Complessità

```
merge(list_1 ... list_k)
  Q = heap di lunghezza k
  list = lista vuota
  for each (i in 1..k)  //costo k log k
    if (list_i non vuota) Q.add(i,list_i.first())
  while (Q non vuoto) //costo n log k
     i = Q.getMin()
    list.add(list_i.getFirst())
    if (list_i non vuota) Q.add(i,list_i.first())
  return list
```

Design e analisi di algoritmi ricorsivi

- sequenza $a_1, \ldots, a_n$ $(n \geq 1)$ bitonica se esiste k, $1 \leq k \leq n$, tale che $a_1, \ldots, a_k$ ordinata in modo strettamente crescente $a_k, \ldots, a_n$ è ordinata in modo strettamente decrescente
- dare un algoritmo che, presa in input una sequenza bitonica, restituisca tale indice k in tempo $O(\log n)$
- giustificare correttezza e complessità dell'algoritmo

Pseudocodice

```
get_index(a,inf,sup)
  mid = (inf+sup)/2
  if (mid<sup)
    if (a[mid]<a[mid+1])
      return get_index(a,mid+1,sup)
    else
      return get_index(a,inf,mid)
 return mid
```

- correttezza per induzione su lunghezza 2^n
- $n = 0$ unico elemento, mid$=$ inf$=$ sup, restituisce correttamente mid
- $2^n + 1$, sicuramente mid $<$ sup, si divide in due sequenze di 2^n elementi
- se l'elemento a destra di mid è maggiore, l'indice può essere solo $>$ mid
- analogamente se l'elemento a destra di mid è minore
- complessità: come ricerca binaria
  $T(1) = 0$, $T(2^{n+1}) = T(2^n) + \Theta(1)$

Programmazione dinamica: problema del resto

- problema del resto di dimensione n: trovare il numero minimo di monete necessarie a dare resto n assumendo tagli $t[1..k]$
- sia $S(n)$ soluzione = sequenza di monete (tagli)
  $C(n)$ il suo costo = numero monete = lunghezza di $S(n)$
- definire induttivamente $C(n)$ giustificando la correttezza
- dare un algoritmo di programmazione dinamica che calcoli $C(n)$ e renda anche possibile ricostruire $S(n)$
- valutare la complessità

Soluzione

- idea: scelto taglio $t[j]$, si deve risolvere il sottoproblema $n - t[j]$
- occorre prendere il minimo tra tutte le scelte della prima moneta
- quindi:
  $C(n) = \min\{1 + C(n - t[j]) \mid t[j] \leq n, j \in 1..k\}$ per $n > 0$
  assumiamo minimo insieme vuoto $= \infty$
  $C(0) = 0$

Algoritmo di programmazione dinamica

```
C[0]=0
for (i=1;i<=n;i++)
  C[i]=∞
  for (j=1;j<=k;j++]
    if (t[j]<=i && 1+C[n-t[j]] < C[i])
      C[i] = 1+C[n-t[j]]
```

- complessità? $O(nk)$

Per ricostruire $S(n)$ la sequenza di monete

- $S[n] =$ prima moneta (taglio) utilizzata per resto n

```
C[0]=0
for (i=1;i<=n;i++)
  C[i]=∞
  for (j=1;j<=k;j++]
    if (t[j]<=i && 1+C[n-t[j]] < C[i])
      C[i] = 1+C[n-t[j]]
      S[i]=j
```

- sequenza monete $S(n) = S[n]$ e poi $S(n - t[j])$

Problema dell'esistenza di una sottosequenza comune

- $X[1..n]$, $Y[1..n]$ due sequenze
  $\exists CS[i,j,k]$, con $0 \leq i,j,k \leq n =$ esiste sottosequenza comune di $X[1..i]$ e $Y[1..j]$ di lunghezza (almeno) k
- definire induttivamente $\exists CS[i,j,k]$ giustificando la correttezza
- dare un algoritmo di programmazione dinamica
  $\exists CS$ matrice a valori booleani
- valutare la complessità
- descrivere come ottenere anche una sottosequenza di lunghezza (almeno) k, se ne esistono

Definizione induttiva: base

- $\exists CS[i,j,0] = T$ per ogni $0 \leq i,j \leq n$
  la sequenza vuota è sempre una sottosequenza comune
- $\exists CS[i,j,k] = F$ per ogni $k > 0$ se $i = 0$ oppure $j = 0$
  se una delle due stringhe è vuota non ci sono sottostringhe comuni di lunghezza > 0

Passo induttivo

- per ogni $0 < i, j, k \leq n$
  $$\exists CS[i,j,k] = \begin{cases} \exists CS[i-1,j-1,k-1] & \text{se } X[i] = Y[j] \\ \exists CS[i-1,j,k] \vee \exists CS[i,j-1,k] & \text{se } X[i] \neq Y[j] \end{cases}$$
- analogamente a quanto visto per LCS
- correttezza per induzione forte
  le chiamate ricorsive sono su triple (i,j,k) strettamente minori

Algoritmo di programmazione dinamica

```
for (i = 0; i <= n; i++)
  for (j = 0; i <= n; i++) Exists[i,j,0] = true

for (k = 1; k <= n; k++)
  for (j = 1; j <= n; j++) Exists[0,j,k] = false
  for (i = 1; i <= n; j++) Exists[i,0,k] = false

for (k = 1; k <= n; k++)
  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
      if (X[i]=Y[j]) Exists[i,j,k]= Exists[i-1,j-1,k-1]
      else Exists[i,j,k]=
        Exists[i-1,j,k]||Exists[i,j-1,k]
```

- complessità costruisce matrice tridimensionale quindi $O(n^3)$.
- per ottenere anche una delle sottosequenze tecnica analoga a quella vista per LCS

Massima sottosequenza palindroma

- data una sequenza $X[1..n]$ trovare la lunghezza della massima sottosequenza palindroma di X
- per esempio, se $X = [A, B, C, D, C, E, A]$, la massima lunghezza di una sottosequenza palindroma è 4, corrispondente alla sottosequenza $[A, C, C, A]$.
- come definire i sottoproblemi?
- $P[i, j]$, con $1 \leq i, j \leq n =$ sottoproblema lunghezza massima sottosequenza palindroma di $X[i..j]$
- definire induttivamente $P[i, j]$ giustificando la correttezza
- dare un algoritmo di programmazione dinamica

Definizione induttiva: base

- massima sottosequenza palindroma di una sequenza vuota o di un elemento è la sequenza stessa
- $P[i,j] = 0$ per $i, j \in 1..n, j < i$
- $P[i,i] = 1$ per $i \in 1..n$

Passo induttivo

- per sequenza di almeno due elementi, due casi
- se primo e ultimo elemento uguali:
- $P[i, j] = P[i + 1, j - 1] + 2$ se $X[i] = X[j]$, $j > i$
- se diversi:
- $P[i, j] = \max\{P[i, j - 1], P[i + 1, j]\}$ se $X[i] \neq X[j]$, $j > i$

Algoritmo di programmazione dinamica

```
//P[1..n,1..n]
for (i=1; i<=n; i++)
  for (j=1; j<i; j++) P[i,j]=0
for (i=1; i<=n; i++) P[i,i]=1
for (i=1; i<n; i++)
  for (j=i+1; j<=n; j++)
    if (X[i]=X[j]) P[i,j]=2+P[i+1,j-1]
    else P[i,j]=max(P[i,j-1],P[i+1,j])
```

- complessità?
- costruisce matrice $n \times n$, quindi complessità (temporale e spaziale) $\Theta(n^2)$