UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA

Edit Distance, Longest Palindromic Subsequence e DivCoins

Esercitazione del 17/05/2020

Francesco Cauteruccio, Ph.D.
cauteruccio@mat.unical.it
francescocauteruccio.info

Inside the Vault EDITION

UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA

Edit Distance, Longest Palindromic Subsequence e DivCoins

Esercitazione del 17/05/2020

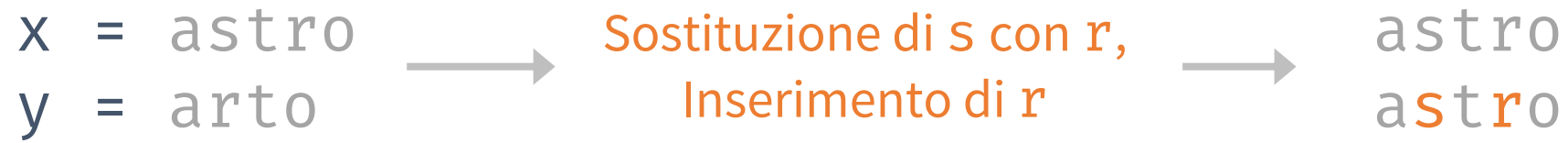
Francesco Cauteruccio, Ph.D.
cauteruccio@mat.unical.it
francescocauteruccio.info

Inside the Vault EDITION

Edit Distance

- Date due stringhe x e y, la **edit distance** tra x e y è definita come il numero minimo di *edit operation* da effettuare per trasformare x in y.
- Le edit operation sono:
  - **INSERTION**: inserire un simbolo in una qualunque posizione,
  - **DELETION**: cancellare un simbolo da una qualunque posizione,
  - **SUBSTITUTION**: sostituire un simbolo c con un simbolo c'.

- Esempio:



EditDistance(astro, arto) = 2

Edit Distance (algoritmo ricorsivo)

```
1. editDistanceRec(x, y, i, j):
2.   if i == 0:
3.     return j
4.
5.   if j == 0:
6.     return i
7.
8.   if x[i-1] == y[j-1]:
9.     editDistanceRec(x, y, i-1, j-1)
10.
11.  return 1 + min(
12.    editDistanceRec(x, y, i-1, j),
13.    editDistanceRec(x, y, i, j-1),
14.    editDistanceRec(x, y, i-1, j-1)
15.  )
16.


// chiamata alla funzione
editDistanceRec(x, y, x.len, y.len)
```

Caso riga 2 e 5:

- i == 0 indica che la prima stringa è vuota → inserimento di tutti i simboli della seconda nella prima
- j == 0 indica che la seconda stringa è vuota → cancello tutti i simboli della prima stringa

Caso riga 8:

- Gli ultimi due simboli sono uguali → non faccio alcunché e procedo con i restanti simboli

Caso riga 11:

- Gli ultimi due simboli non sono uguali → consideriamo le tre possibili operazioni, prendendo quella che porta al costo minimo (# di operazioni)

Edit Distance (con la programmazione dinamica)

- **Sottoproblemi**
  - I sottoproblemi sono rappresentati dalla edit distance tra i prefissi delle stringhe,
  - Bottom-up: **risolviamo prima i sottoproblemi semplici** (la edit distance tra due simboli), per poi **usarli nella risoluzione dei sottoproblemi più difficili** (la edit distance tra due prefissi)
- **Relazione di ricorrenza**
  - Come definiamo le relazioni tra i sottoproblemi? (*come uso la soluzione di un sottoproblema per un altro?*)
  - Siano `i` (risp. `j`) una posizione nelle stringhe x (risp. y), allora

$$\text{edit}(i, j) = \min \begin{cases} \text{edit}(i-1, j) + 1, \\ \text{edit}(i, j-1) + 1, \\ \text{edit}(i-1, j-1) + s \end{cases}$$

Costo sostituzione:
s = 0 se x[i] == y[j],
s = 1 altrimenti

Edit Distance (con la programmazione dinamica)

```
1. editDistance(x, y):
2.   d = new matrix[x.len+1][y.len+1]
3.
4.   for i = 0 to x.len:
5.     d[i][0] = i
6.   for j = 0 to y.len:
7.     d[0][j] = j
8.
9.   for i = 1 to x.len:
10.    for j = 1 to y.len:
11.      s = 0 if x[i-1] == y[j-1] else 1
12.
13.      d[i][j] = min(
14.        d[i-1][j] + 1,
15.        d[i][j-1] + 1,
16.        d[i-1][j-1] + s
17.      )
18.
19.  return d[x.len][y.len]
```

Variabili:

- d = matrice per la memoizzazione, la cella d[i][j] contiene la edit distance tra x[:i] e y[:j]

For riga 4 e 5:

- Sottoproblemi semplici (la edit distance tra il primo simbolo di una stringa e tutti i prefissi dell'altra)

I due for riga 9, 10:

- Implementazione della relazione di ricorrenza
- s indica il costo della sostituzione (0 se i simboli sono uguali)

Longest Palindromic Subsequence

- Esercizio *Palind* su domjudge
- Data una stringa x, determinare la lunghezza della sua **sottosequenza palindroma più lunga**
- Esempio:

```
x = scacciapensieri
    → scacciapensieri → saccas (lunghezza = 6)
```

- Differenza tra *sottostringa* e *sottosequenza*:
  - `boh` è una sottostringa di `aibohphobia` (sequenza contigua di simboli),
  - `bop` è una sottosequenza di `aibohphobia` (sequenza [possibilmente] non contigua di simboli)

Longest Palindromic Subsequence (algoritmo ricorsivo)

```
1. LPS(s, i, j):
2.   if i == j:
3.     return 1
4.
5.   if s[i] == s[j] and i+1 == j:
6.     return 2
7.
8.   if s[i] == s[j]:
9.     return LPS(s, i+1, j-1) + 2
10.
11.  return max(
12.    LPS(s, i, j-1),
13.    LPS(s, i-1, j)
14.  )
15.
16.
17. // la prima chiamata
18. LPS(s, 0, s.len-1)
```

Caso riga 2:

- La lunghezza della LPS è 1.

Caso riga 5:

- Sottostringa del tipo *aa*, la lunghezza della LPS è 2.

Caso riga 8:

- Due simboli uguali a indice i e j, la lunghezza della LPS è 2 + la LPS della stringa più interna.

Caso riga 11:

- La lunghezza della LPS è la più grande tra le LPS delle stringhe s[i:j-1] e s[i-1:j].

Longest Palindromic Subsequence (rendiamolo più dinamico)

```
1. LPSd(i, j):
2.   if dp[i][j] != -1:
3.     return dp[i][j]
4.
5.   if i > j:
6.     return 0
7.
8.   if i == j:
9.     return 1
10.
11.  if i < j and s[i] == s[j]:
12.    dp[i][j] = 2 + LPSd(i+1,j-1)
13.    return dp[i][j]
14.
15.  if i < j and s[i] != s[j]:
16.    dp[i][j] = max(
17.      LPSd(i+1,j),
18.      LPSd(i,j-1)
19.    )
20.    return dp[i][j]
21.
22.  return 0
```

Matrice dp:

- Serve alla memoizzazione di “alcuni” sottoproblemi,
- Viene inizializzata all’esterno, ha dimensioni (s.len, s.len) e contiene inizialmente -1

Esercizio per casa:

Come si può completamente eliminare la ricorsione?

DivCoins

- Esercizio *DivCoins* su domjudge

- Dato un sacchetto contenente n monete, dove ogni moneta ha un valore positivo, determinare **la più equa divisione** delle monete tra **due persone**
  - Equa divisione: la **differenza** tra la *somma delle monete date alla prima persona* e la *somma delle monete date alla seconda persona* deve essere la **minima** possibile (tra tutte le possibili divisioni)

- Esempio: {10, 4, 6, 3, 8, 2}
  - una equa divisione è la seguente: {10, 6}, {8, 4, 3, 2} → |16 - 17| = 1

DivCoins (possibili approcci)

- Bruteforce
  - Provare tutte le possibili divisioni.
  - *Pro*: soluzione ottima assicurata,
    Con: complessità temporale enorme.
- Greedy
  - Uno stage dell'algoritmo greedy sceglie se inserire o meno la monete corrente in una partizione
  - Il criterio della scelta è **locale** (si basa solo sulle monete inserite fino a questo punto)
  - *Pro*: complessità lineare,
    Con: soluzione ottima non assicurata.
- Altri approcci?
  - Variazione del problema **subset sum** (dato un insieme `I` di numeri positivi e una soglia `T`, stabilire se esiste un sottoinsieme `I'` tale che *sum*(`I'`) ≤ `T` e sia la più grande possibile)
  - Variazione del problema **coin change** (dato un numero `N` e un insieme di monete `M`, in quanti modi possiamo ottenere `N` usando le monete di `M`?)

DivCoins (intuizione, basato su Coing Change)

- Due fasi

- **Fase 1: determinare tutte le possibili somme**
  - Quali sono le possibili somme che possiamo fare con tutte le monete?
  - Enumerare tutte le possibili somme in un modo intelligente.

- **Fase 2: qual è la somma, tra tutte le possibili, che ci permette di ottenere la differenza minima?**
  - Manteniamo una variabile `min` indicante la differenza minima possibile (inizialmente = T, dove T è la somma di tutte le monete),
  - Aggiorniamo `min` se troviamo una somma che ci permette di avere una differenza più piccola di `min`.

DivCoins (pseudo)

```
1. DivCoins(coins):
2.   n = coins.len
3.   T = sum(coins)
4.
5.   possibiliSomme = new array[false; T+1]
6.   possibiliSomme[0] = true
7.
8.   for i = 0 to n-1:
9.     for j = T - coins[i] to 0:
10.       if possibiliSomme[j]:
11.         possibiliSomme[j + coins[i]] = true
12.
13.  min = T
14.  for i = 0 to T:
15.    diff = |T - (i*2)|
16.    if possibiliSomme[i] and diff < min:
17.      min = diff
18.
19.  return min
```

Fase 1 (riga 5–11):

- possibiliSomme è un array lungo T+1 (da 0 a T) con tutti gli elementi inizialmente pari a false
- possibiliSomme[i] indica se è possibile ottenere la somma i
- Se possibiliSomme[j] è true, allora posso ottenere anche la somma j + coins[i]

Fase 2 (riga 13–17):

- Per ogni possibile somma i, diff è la differenza tra la somma delle monete e i*2
- Se i è una somma ottenibile e la differenza è più piccola di min, aggiorno min