Laboratorio 4

Elementi di Informatica e Programmazione

Esercizio 1: Sottostringa più lunga

Scrivere il programma `longestSubstring.py` che identifichi e visualizzi la più lunga sottostringa comune a due stringhe ricevute. Il programma deve contenere una funzione come la seguente.

```
def longest_substring(string1, string2):
    # your implementation
```

Le due stringhe saranno poi lette usando due chiamate alla funzione `input`.

Esempio: la più lunga sequenza di caratteri consecutivi (cioè sottostringa) comune alle due stringhe:

`PippoPluto2Paperino MinnieBambito2Dumbo`

è `to2`

Verificare che il programma gestisca correttamente la situazione in cui le due stringhe non hanno alcun carattere in comune e, quindi, la più lunga sottostringa appartenente a entrambe è la stringa vuota.

Provate due strategie:

1. Il programma non utilizza metodi che operano su stringhe, né l'operatore in.
2. Il programma utilizza metodi che operano su stringhe e l'operatore in.

Attenzione: le specifiche non sono complete… Cosa deve visualizzare il programma nel caso in cui le stringhe da elaborare siano PippoPluto e PippoMinniePluto ?

Fare ipotesi ragionevoli.

Soluzione

```
def longest_subsequence(s1, s2):
    longest = ""
    for i in range(len(s1)) :
        # i è l'indice di inizio della sottostringa in s1
        for j in range(len(s2)) :
            # j è l'indice di inizio della sottostringa in s2
            k = 0 # k è la lunghezza della sottostringa comune che sto cercando
            while i + k < len(s1) and j + k < len(s2) :
                if s1[i + k] != s2[j + k] :
                    break # la sottostringa comune è finita e ha lunghezza k (anche zero)
                k += 1
            if k > len(longest) :
                longest = s1[i : i + k] # oppure s2[j : j + k], è indifferente
                # se esistono più sottostringhe comuni aventi la stessa
                # lunghezza, nel qual caso le specifiche del programma sono
                # incomplete, questo programma sceglie quella che si trova
                # più a sinistra in s1
                #
                # scrivendo >= invece di > in questo if si sceglierebbe,
                # al contrario, quella più a destra in s1
    return longest

s1 = input("Prima stringa: ")
s2 = input("Seconda stringa: ")

longest = longest_substring(s1, s2)
print("La più lunga sottostringa comune a ", end="")
print('"' + s1 + '" e "' + s2 + '" è "' + longest + '"')
```

Ecco una seconda soluzione, più veloce. Come mai? Funziona?

```
def longest_subsequence(s1, s2):
    longest = ""
    i = 0
    while i < len(s1) - len(longest) : # qui è cambiato l'intervallo
        j = 0
        while j < len(s2) - len(longest) : # qui è cambiato l'intervallo
            # poi è tutto come prima...
            k = 0
            while i + k < len(s1) and j + k < len(s2) :
```

```
                if s1[i + k] != s2[j + k] :
                    break
                k += 1
            if k > len(longest) :
                longest = s1[i : i + k]
            j += 1
        i += 1
    return longest
```

perché non ho scritto i due cicli più esterni con un for ?

Esercizio 2: verifica password accettabili

Scrivere il programma newPassword.py che chieda all'utente una nuova password e la accetti soltanto se rispetta i requisiti di sicurezza, che, in questo esempio, sono:

- deve contenere almeno 8 caratteri
- deve contenere almeno una lettera maiuscola
- deve contenere almeno una lettera minuscola
- deve contenere almeno una cifra numerica
- deve contenere almeno un carattere che non sia una lettera né una cifra
- non deve contenere spazi
- non deve contenere caratteri ripetuti (cioè tutti i caratteri devono essere diversi)

Se la password è accettabile il programma termina, altrimenti visualizza un messaggio d'errore (che deve specificare quali delle regole sono state violate) e ripete la richiesta. Se la password viola più regole, queste vanno elencate tutte. Definire una funzione per ciascuna regola, che riceva la password e restituisca `True` se e solo se la regola corrispondente è rispettata (senza avere alcuna interazione con l'utente, né in input né in output). Definire, inoltre, una funzione che riceva la password e restituisca `True` se e solo se la password è valida, visualizzando un messaggio d'errore per ciascuna regola violata ma senza richiedere una nuova password.

Soluzione

```
def main() :
    while True :
        s = input("Password: ")
        if is_valid(s) :
            print("OK")
            break
```

```
        print("Inserire una nuova password")

def is_valid(password) :
    ok = True
    if not rule1(password) :
        ok = False
        print("La password non contiene almeno 8 caratteri")
    if not rule2(password) :
        ok = False
        print("La password non ha almeno una lettera maiuscola")
    if not rule3(password) :
        ok = False
        print("La password non ha almeno una lettera minuscola")
    if not rule4(password) :
        ok = False
        print("La password non ha almeno una cifra numerica")
    if not rule5(password) :
        ok = False
        print("La password non ha almeno un carattere che non sia una lettera né una cifra
    if not rule6(password) :
        ok = False
        print("La password contiene almeno uno spazio")
    if not rule7(password) :
        ok = False
        print("La password contiene almeno un carattere ripetuto")
    return ok

def rule1(password) :
    return len(password) >= 8

def rule2(password) :
    return password != password.lower()

def rule3(password) :
    return password != password.upper()

def rule4(password) :
    for char in password :
        if char.isdigit() :
            return True
    return False
```

```
def rule5(password) :
    for char in password :
        if not char.isalnum() :
            return True
    return False

def rule6(password) :
    for char in password :
        if char == ' ' :
            return False
    return True

def rule7(password) :
    for i in range(len(password)) :
        for j in range(i + 1, len(password)) :
            if password[i] == password[j] :
                return False
    return True

main()
```

Esercizio 3: generazione di sottostringhe

Come visto nell'esercizio 1, data una stringa `s` le sue sottostringhe sono tutte e sole le sequenze di caratteri che compaiono consecutivamente in `s`. Esistono anche due sottostringhe "improprie": la stringa vuota `""` (che è una sottostringa di tutte le stringhe) e la stringa `s` stessa.

Create un modulo `mystringops.py` e scrivete una funzione `gen_substrings` con la seguente firma:

```
def gen_substrings(string):
    # your implementation
```

La funzione deve ritornare una lista contenente *tutte* le sottostringhe della stringa data, in ordine crescente di lunghezza.

Suggerimento: cominciate risolvendo un problema più piccolo. Quali sono tutte le sottostringhe di `string` di lunghezza 1? Quali quelle di lunghezza 2? Una volta pensata la soluzione a questo sottoproblema procedete a scrivere un loop su tutte le lunghezze di sottostringhe possibili.

Soluzione

```
def gen_substrings(s) :
    subs = [""] # prima sottostringa "impropria"
    # per ogni possibile lunghezza di sottostringa...
    for substring_length in range(1, len(s)) :
        # per ogni possibile posizione iniziale di
        # una sottostringa di lunghezza substring_length...
        for j in range(len(s) - substring_length + 1) :
            subs.append(s[j : j + substring_length])
    if len(s) > 0 :
        subs.append(s) # ultima sottostringa "impropria",
                       # solo se s non è vuota altrimenti risulta doppia
    return subs
```

Esercizio 4: numeri binari

Scrivere una funzione `as_binary` (nel modulo `mystringops`) che, dati x e n, restituisca la rappresentazione binaria di x con le seguenti caratteristiche:

- la codifica deve avere n cifre binarie (quindi bisognera aggiungere degli `0` per raggiungere la lunghezza desiderata)
- le cifre binarie devono essere poste in una lista
- la cifra meno significativa occupa la prima posizione della lista, la seconda meno significativa la seconda posizione e così via.

La funzione può avere la seguente firma:

```
def as_binary(x, n):
    # your implementation here
```

A titolo di esempio, la funzione quando chiamata con input `x=3` e `n=3` deve generare questo output:

```
[1, 1, 0]
```

Soluzione

```
def as_binary(x, n):
    out = []
```

```
    # Implementiamo l'algoritmo per convertire un numero da base
    # 10 a base 2, accumulando le cifre nella lista `out`
    while x > 0:
        rem =  x % 2
        x //= 2
        out.append(rem)

    # Aggiungiamo `0` fino a ottenere una lista di lunghezza `n`.
    # Cosa succede se la lista ha già lunghezza n?
    while len(out) < n:
        out.append(0)
    return out
```

Esercizio 5: generare sottosequenze

Le sottosequenze di una stringa sono simili alle sottostringhe, ma i caratteri non sono necessariamente contigui. In altre parole, una sottosequenza è un sottoinsieme dei caratteri della stringa di input, nello stesso ordine in cui appaiono nell'input.

Ad esempio, le *sottostringhe* di `"ciao"` sono:

```
['', 'c', 'i', 'a', 'o', 'ci', 'ia', 'ao', 'cia', 'iao', 'ciao']
```

mentre le sottosequenze sono:

```
['', 'c', 'i', 'ci', 'a', 'ca', 'ia', 'cia', 'o',
 'co', 'io', 'cio', 'ao', 'cao', 'iao', 'ciao']
```

Notate che `''cao'` è una sottosequenza di `'ciao'`, ma non è una sottostringa.

Scopo dell'esercizio è creare, nel modulo `mystringops.py` una funzione `gen_substrings` che generi una lista di tutte le sottosequenze di una stringa data.

Generare sottosequenze è più difficile che generare sottostringhe, quindi procederemo passo passo partendo da una osservazione. Se immaginiamo che la sequenza sia memorizzata in una lista, come possiamo descrivere i valori che appartengono a una specifica sottosequenza? Possiamo farlo mediante le posizioni, all'interno della lista, dei caratteri che appartengono alla sottosequenza: tali posizioni sono, ovviamente, un sottoinsieme delle posizioni valide all'interno della lista (i due sottoinsiemi impropri corrispondono alle sottosequenze improprie: nessuna posizione e tutte le posizioni). Che alternative abbiamo per specificare un sottoinsieme delle posizioni di una sequenza? Possiamo generare una nuova sequenza contenente, come valori, tutti e soli gli indici corrispondenti alle posizioni che fanno parte del sottoinsieme. Oppure,

possiamo utilizzare una sequenza di valori booleani, avente la stessa lunghezza della sequenza che stiamo elaborando, dove ciascun valore corrisponde al fatto che il valore della sequenza che si trova nella posizione corrispondente faccia parte del sottoinsieme (oppure no): una lista di questo tipo viene solitamente chiamata "vettore caratteristico" del sottoinsieme che vuole rappresentare.

Esempio:

```
sequenza = "abcdef"
sottosequenza = "acd"
vettore caratteristico = [True, False, True, True, False, False]
```

A volte, per semplicità, invece di una lista di valori booleani si usa una lista di valori numerici zero o uno (dove, ad esempio, zero corrisponde a False). L'esempio precedente diventa:

```
[1, 0, 1, 1, 0, 0]
```

Il primo valore 1 corrisponde al fatto che la lettera "a" (la prima della sequenza) fa parte della sottosequenza. Il primo valore 0 corrisponde al fatto che la lettera "b" (la seconda della sequenza) NON fa parte della sottosequenza. Osserviamo che, ovviamente, questa modalità di descrizione delle sottosequenze è valida anche quando la sequenza contiene eventuali elementi replicati, perché non fa riferimento ai valori contenuti nella sequenza, bensì alle loro posizioni. Dopo aver osservato ciò, come possiamo generare tutte le sottosequenze di una sequenza? Generiamo tutti i vettori caratteristici che le descrivono! Quali sono tali vettori? Sono tutte (e sole!) le liste di lunghezza uguale alla lunghezza della sequenza e contenenti valori 0 e 1, in tutte le combinazioni possibili.

Quante sono le possibili sequenze di n valori, ciascuno dei quali viene scelto tra due possibilità? Sono 2^n. Come le genero? Sono le rappresentazioni binarie posizionali dei numeri interi che vanno da zero a $2^n - 1$.

Vediamo una descrizione dell'algoritmo in pseudocodice:

```
n = len(s) # s è la stringa di input
subs = [ ] # lista vuota
for x in range(2**n) :
    converti x in binario con n bit (usare la funzione as_binary
          creata per l'esercizio precedente)
    sub = ""
    for j in range(n) :
        se il bit j-esimo della conversione è 1
        sub += s[j]
    subs.append(sub)
```

La firma della funzione può essere

```
def gen_subsequences(string):
    # your code here
```

Soluzione

```
def get_subsequences(string):
    n = len(string)

    # Inizializziamo la lista di output
    out = []

    # Per ogni numero da 0 a (n**2 - 1)
    for x in range(n**2):
        # Otteniamo la lista di 0 e 1 corrispondente
        # al numero x, da usare come selettore
        selector = as_binary(x, n)
        # Inizializziamo la sottosequenza vuota
        subsequence = ""
        # Per ognuno dei caratteri della stringa, guardiamo
        # se è "attivo" nella lista selettore
        for i in range(n):
            if selector[i] == 1:
                subsequence += string[i]
        # Aggiungiamo all'output la sottosequenza
        out.append(subsequence)

    return out
```

Esercizio 6: modulo `mystringops`

Arricchire il modulo `mystringops.py` (che già contiene `gen_substrings` e `gen_subsequences`) con le seguenti funzioni:

- `num_substrings(s)` restituisce la lunghezza della lista che verrebbe restituita da getAllSubstrings(s), senza costruire le sottostringhe né invocare getAllSubstrings(s); ad esempio, `num_substrings("") = 1`, `num_substrings("tu") = 4`, `num_substrings("casa") = 11`, ecc.

- `are_unique(ss)` riceve una lista di stringhe (eventualmente vuota) e restituisce `True` se e solo se i suoi elementi sono tutti diversi
- `is_sorted_by_increasing_length(ss)` riceve una lista di stringhe (eventualmente vuota) e restituisce `True` se e solo se i suoi elementi sono disposti in ordine di lunghezza crescente (o, meglio, non decrescente) al crescere dell'indice
- `is_sorted_by_decreasing_length(ss)` riceve una lista di stringhe (eventualmente vuota) e restituisce `True` se e solo se i suoi elementi sono disposti in ordine di lunghezza decrescente (o, meglio, non crescente) al crescere dell'indice
- `is_forward_sorted(ss)` riceve una lista di stringhe (eventualmente vuota) e restituisce `True` se e solo se i suoi elementi sono disposti, al crescere dell'indice, secondo l'ordinamento imposto dal criterio lessicografico
- `is_backward_sorted(ss)` riceve una lista di stringhe (eventualmente vuota) e restituisce `True` se e solo se i suoi elementi sono disposti, al crescere dell'indice, secondo l'ordinamento inverso di quello imposto dal criterio lessicografico
- `is_sorted(ss)` svolge la stessa elaborazione svolta dalla funzione `is_forward_sorted` (risolvere il problema senza copiare il codice di quella funzione)
- `is_substring(s1, s2)` riceve due stringhe e restituisce True se e solo se s1 è una sottostringa (eventualmente impropria) di s2 (è ammesso l'uso dell'operatore `in`)
- `is_subsequence(s1, s2)` riceve due stringhe e restituisce True se e solo se s1 è una sottosequenza (eventualmente impropria) di s2 (come visto nell'Esercizio 4-6, una stringa s1 è una sottosequenza di un'altra stringa s2 se e solo se tutti i caratteri di s1 sono presenti in s2 nello stesso ordine, anche se in posizioni diverse e non consecutive)
- `num_subsequences(s)` restituisce la lunghezza della lista che verrebbe restituita da `get_subsequences(s)`, senza costruire le sottostringhe né invocare `get_subsequences(s)`; ad esempio, `num_subsequences("") = 1`, `num_subsequences("tu") = 4`, `num_subsequences("casa") = 16`, ecc.

Soluzione

```
def num_substrings(s) :
   return 1 + len(s) * (len(s) + 1) // 2

def are_unique(ss) :
   for i in range(len(ss)) :
      for j in range(i + 1, len(ss)) :
         if ss[i] == ss[j] :
            return False
   return True

def is_sorted_by_increasing_length(ss) :
   for i in range(1, len(ss)) :
```

```
      if len(ss[i - 1]) > len(ss[i]) :
         return False
   return True

def is_sorted_by_decreasing_length(ss) :
   for i in range(1, len(ss)) :
      if len(ss[i - 1]) < len(ss[i]) :
         return False
   return True

def is_forward_sorted(ss) : # lexicographically
   for i in range(1, len(ss)) :
      if ss[i - 1] > ss[i] :
         return False
   return True

def is_backward_sorted(ss) : # lexicographically
   for i in range(1, len(ss)) :
      if ss[i - 1] < ss[i] :
         return False
   return True

def is_sorted(ss) :
   return is_forward_sorted(ss)

def is_substring(s1, s2) :
   return s1 in s2

def is_subsequence(s1, s2) :
   k = j = 0
   while j < len(s1) and k < len(s2) :
      if s2[k] == s1[j] :
         j += 1
      k += 1
   return j == len(s1)

def num_subsequences(s) :
   return 2 ** len(s)
```

Esercizio 7: test del modulo `mystringops`

Scrivere, infine, un programma di collaudo, `test_substring.py`, che svolga le seguenti elaborazioni dopo aver importato il modulo `mystringops`:

- per ogni stringa appartenente alla lista `["", "a", "ab", "abc", "abcdefghilm"]` genera tutte le sottostringhe invocando `get_substrings` e ottenendo la lista `ss`, poi:
- visualizza la lunghezza di `ss` prevista dalla funzione `num_substrings`
- visualizza la lunghezza effettiva di `ss`, evidenziando un errore se i due valori differiscono
- passa `ss` alla funzione areUnique e segnala un errore se viene restituito il valore `False`
- ordina la lista `ss` usando il metodo sort
- passa `ss` alla funzione `is_forward_sorted` e segnala un errore se viene restituito il valore `False`
- inverte il contenuto della lista `ss` usando la funzione `reverse`, da progettare all'interno del programma (tale funzione riceve una lista e ne inverte il contenuto)
- passa `ss` alla funzione `is_backward_sorted` e segnala un errore se viene restituito il valore `False`
- per ogni stringa appartenente alla lista `["", "a", "ab", "abc", "abcdefghilm"]` genera tutte le sottosequenze invocando `get_subsequences` e ottenendo la lista `ss`, poi:
- visualizza la lunghezza di `ss` prevista dalla funzione `num_subsequences`
- visualizza la lunghezza effettiva di `ss`, evidenziando un errore se i due valori differiscono
- passa `ss` alla funzione `are_unique` e segnala un errore se viene restituito il valore `False`

Soluzione

```
from mystringops import *

def reverse(s) :
   for i in range(len(s) // 2) :
      temp = s[i]
      s[i] = s[-i-1]
      s[-i-1] = temp

for s in ["", "a", "ab", "abc", "abcdefghilm"] :
   print('"' + s + '"')
   ss = gen_substrings(s)
   print("Sottostringhe, valore previsto:", num_substrings(s))
   print("Sottostringhe, valore effettivo:", len(ss))
   if len(ss) != num_substrings(s) :
      print("      ERRORE !!")
   if are_unique(ss) :
```

```
   print("Le sottostringhe sono tutte diverse")
else :
   print("ERRORE !! Le sottostringhe NON sono tutte diverse")
ss.sort()
if not is_forward_sorted(ss) :
   print("ERRORE !! Il metodo isForwardSorted non funziona")
reverse(ss)
if not is_backward_sorted(ss) :
   print("ERRORE !! Il metodo isBackwardSorted non funziona")
ss = gen_subsequences(s)
print("Sottosequenze, valore previsto:", num_subsequences(s))
print("Sottosequenze, valore effettivo:", len(ss))
if len(ss) != num_subsequences(s) :
   print("      ERRORE !!")
if are_unique(ss) :
   print("Le sottosequenze sono tutte diverse")
else :
   print("ERRORE !! Le sottosequenze NON sono tutte diverse")
```