Laboratorio 9 - Esercizi

Elementi di Informatica e Programmazione

Lab 9 – Es 1

Progettare una funzione che

- riceve due numeri interi (positivi) come parametri
- calcola e restituisce il massimo comune divisore (M.C.D.) dei due numeri ricevuti

Si ricorda che il M.C.D. di due numeri interi positivi è il più grande numero intero che è divisore di entrambi.

Per calcolare il M.C.D. di due numeri, **m** e **n**, si implementi la seguente definizione:

- se **n** è un divisore di **m**, allora n è il loro M.C.D.
- altrimenti, il M.C.D. di **m** e **n** è uguale al M.C.D. di **n** e del resto della divisione intera di **m** per **n**

Inserire la funzione nel programma **recursiveMCD.py** che contenga anche la funzione **main**, che chiede all'utente di fornire due numeri interi positivi e ne visualizza il M.C.D.

Lab 9 – Es 2 (1/3)

Scrivere il programma **recursiveAndIterativeFibonacci.py** che contenga due funzioni

- **recursiveFib**
- **iterativeFib**

Entrambe ricevono come parametro un numero intero non negativo, **n**, e restituiscono l'**n**-esimo numero **Fib(n)** nella sequenza di Fibonacci, così definita:

- **Fib(0) = 0**
- **Fib(1) = 1**
- **Fib(n) = Fib(n-2) + Fib(n-1)** per ogni **n > 1**

La funzione **recursiveFib** calcola il valore da restituire usando la ricorsione doppia, implementando direttamente la formula, mentre la funzione **iterativeFib** deve calcolare il valore da restituire senza usare la ricorsione e senza usare strutture dati di memorizzazione (cioé senza liste/tuple, ma usando soltanto variabili "semplici").

Lab 9 – Es 2 (2/3)

Inserire nel programma la funzione **main** che:

- chieda all'utente il numero **n** desiderato
- calcoli Fib(n) invocando **iterativeFib**, misurando il impiegato per il calcolo usando la funzione **perf_counter** del modulo **time**, illustrata nel seguito
- calcoli Fib(n) invocando **recursiveFib**, misurando il impiegato per il calcolo usando la funzione **perf_counter**
- visualizzi Fib(n) (oppure un messaggio d'errore se i due valori calcolati sono diversi...)
- visualizzi il tempo impiegato dalla funzione **iterativeFib**
- visualizzi il tempo impiegato dalla funzione **recursiveFib**

Osservare come, all'aumentare di **n**, il tempo impiegato dalla funzione iterativa rimanga pressoché trascurabile, mentre il tempo impiegato dalla funzione ricorsiva aumenti in modo esponenziale, seguendo approssimativamente la funzione 2^n, cioè il tempo richiesto per calcolare **Fib(n)** è circa il doppio di quello richiesto per calcolare Fib(n-1).

Lab 9 – Es 2 (3/3)

La funzione **perf_counter** del modulo **time** restituisce, ogni volta che viene invocata, un numero in virgola mobile che rappresenta il numero di secondi trascorsi fino all'istante in cui viene invocata, a partire da un evento predefinito (che dipende il sistema operativo, ma solitamente è l'istante iniziale del giorno 1 gennaio 1970, chiamato *the epoch*). Ovviamente ciò che interessa non è il valore di tale numero, bensì la differenza tra i valori restituiti da due invocazioni successive, che rappresentano la durata del corrispondente intervallo di tempo indipendentemente dall'istante di riferimento. Invocando tale funzione subito prima e subito dopo una sezione di codice, si ottiene il tempo impiegato dall'esecuzione della sezione stessa. Ad esempio:

```
from time import perf_counter
beginTime = perf_counter()
do_something()
endTime = perf_counter()
# tempo impiegato da do_something(), in secondi
elapsedTime = endTime - beginTime
```

Il tempo così calcolato è poco preciso per valori piccoli (inferiori a qualche secondo), ma è significativo per valori più elevati.

Lab 9 – Es 3

Scrivere il programma **recursiveIsPalindrome.py** che verifichi se una stringa, fornita dall'utente, è un **palindromo** oppure no.

Si ricorda che una stringa è un palindromo se è composta da una sequenza di caratteri (anche non alfabetici) che possa essere letta allo stesso identico modo anche al contrario (es. "radar", "anna", "inni", "xyz%u%zyx").

Il programma non deve utilizzare costrutti iterativi (cioè non deve avere nessun tipo di ciclo).

Verificare il corretto funzionamento del programma con:

- una stringa di lunghezza pari che sia un palindromo
- una stringa di lunghezza dispari che sia un palindromo
- una stringa di lunghezza pari che non sia un palindromo
- una stringa di lunghezza dispari che non sia un palindromo
- una stringa di lunghezza unitaria (che è ovviamente un palindromo)
- una stringa di lunghezza zero (che è ragionevole ritenere sia un palindromo, dato che niente la rende "non un palindromo"...)

Lab 9 – Es 4 (1/3)

In relazione al problema della generazione delle permutazioni di una stringa, scrivere il programma **stringPermutations.py** che, dopo aver acquisito la stringa fornita dall'utente (che non deve contenere caratteri duplicati), generi, in due modi diversi, tutte le permutazioni di tale stringa.

Innanzitutto, scrivere e capire la funzione **recursivePermutations** (sotto) che riceve una stringa e restituisce una lista contenente tutte le sue permutazioni.

```
def recursivePermutations(s) :
  # se la stringa contiene caratteri duplicati, la lista restituita
  # conterrà stringhe duplicate, ma saranno comunque presenti
  # tutte le permutazioni della stringa s
  if len(s) == 0 :
    return [] # la stringa vuota non ha permutazioni
  if len(s) == 1 :
    return [s] # la stringa è l'unica permutazione di se stessa
  lst = []
  c = s[0]
  lst2 = recursivePermutations(s[1:]) # tolgo il primo carattere per ogni
  # stringa della lista, aggiungo il primo carattere in tutte le posizioni possibili
  for ss in lst2 :
    for i in range(len(ss)+1) :
      lst.append(ss[:i] + c + ss[i:])
  return lst
```

Progettare, poi, la funzione **iterativePermutations**, che svolga la stessa elaborazione della funzione **recursivePermutations**.

Lab 9 – Es 4 (2/3)

Generare le permutazioni di una stringa senza utilizzare la ricorsione è abbastanza complicato e individuare un algoritmo non è per niente semplice... se, dopo averci pensato un po', non si riesce a risolvere il problema, implementare l'algoritmo seguente, che si basa sul principio di generare le permutazioni in ordine lessicografico crescente, per cui ogni nuova permutazione generata deve essere quella "minima" tra tutte quelle mancanti e "maggiori" di quella appena generata:

- si utilizza una lista **cl** contenente inizialmente i caratteri della stringa ordinati in senso lessicografico crescente (ad es. [‘a’, ‘b’, ‘c’, ‘d’])
- la prima permutazione si ottiene concatenando i caratteri presenti in **c**
- si calcola il numero **count** di permutazioni da generare (che è il fattoriale della lunghezza della stringa) e si esegue un ciclo a contatore **k** con **count - 1** iterazioni (perché la prima permutazione è già stata generata)
    1. procedendo a ritroso nella lista **cl**, partendo da **i** penultimo indice (i+1 quindi ultimo indice) si cerca la prima coppia tale che **cl[i] < cl[i+1]**
    2. Procedendo a ritroso nella lista **cl**, partendo dall’ultimo elemento **j**, si cerca il primo elemento **cl[j]** che è minore di **cl[i]**
    3. si scambia **c[i]** con **c[j]**
    4. si inverte il contenuto della porzione di lista che va dalla posizione **i** (esclusa) alla fine della lista
    5. La lista **cl** contiene una nuova permutazione

Lab 9 – Es 4 (3/3)

Prima di scrivere il codice, eseguire l'algoritmo a mano, su carta, con una stringa di 4 caratteri e cercare di capire come procede, passo dopo passo.

Infine, ordinare le due liste (con il metodo **sort**) e confrontarle con l'operatore **==**, verificando che siano identiche, e visualizzarne una.

Nel collaudo, attenzione ai casi particolari...

Quante permutazioni ha la stringa vuota?

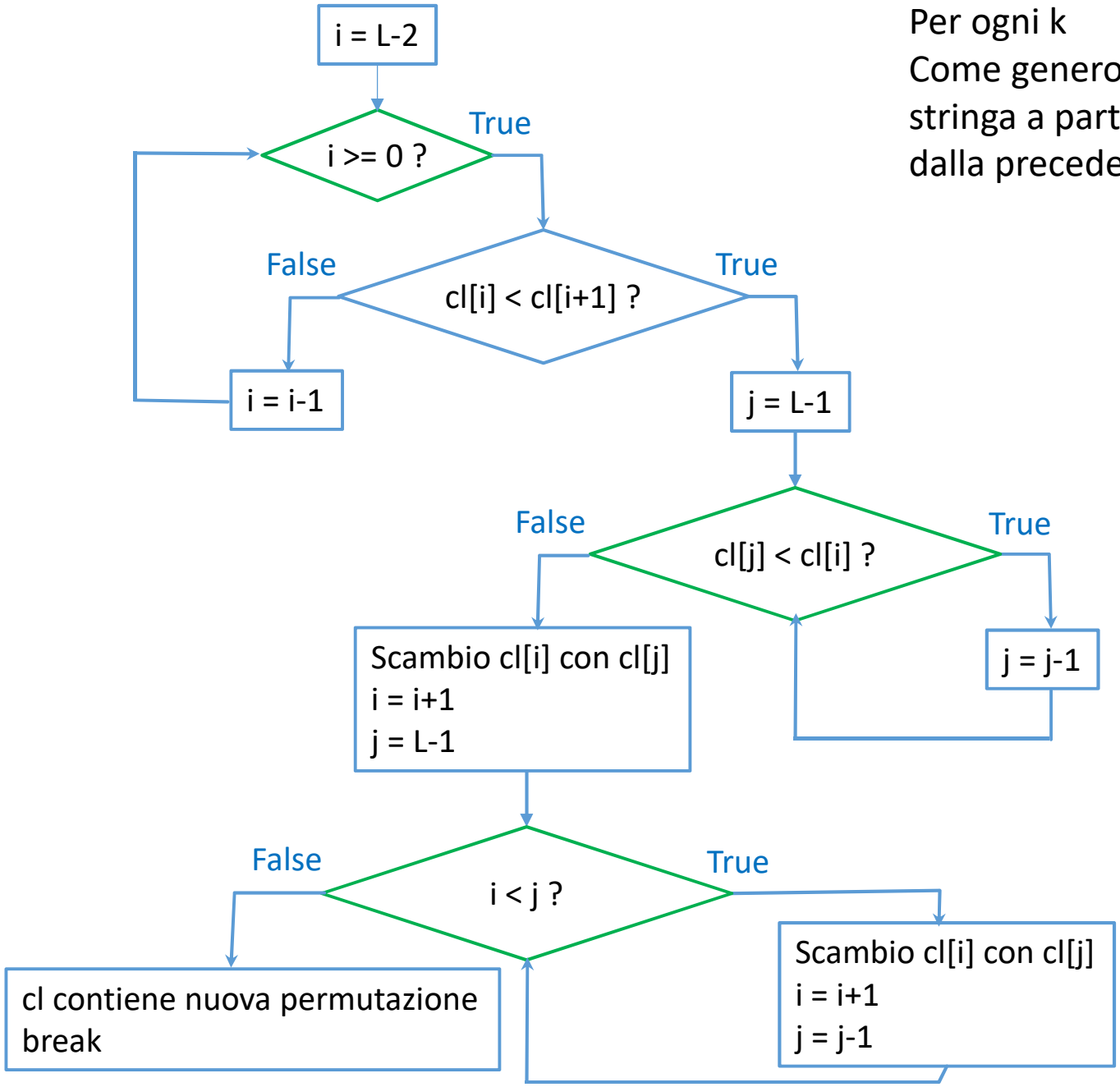
Quante e quali permutazioni ha una stringa di lunghezza uno?

1. 'abcd'
2. 'abdc'
3. 'acbd'
4. 'acdb'
5. 'adbc'
6. 'adcb'
7. 'bacd'
8. 'badc'
9. 'bcad'
10. 'bcda'
11. 'bdac'
12. 'bdca'
13. 'cabd'
14. 'cadb'
15. 'cbad'
16. 'cbda'
17. 'cdab'
18. 'cdba'
19. 'dabc'
20. 'dacb'
21. 'dbac'
22. 'dbca'
23. 'dcab'
24. 'dcba'

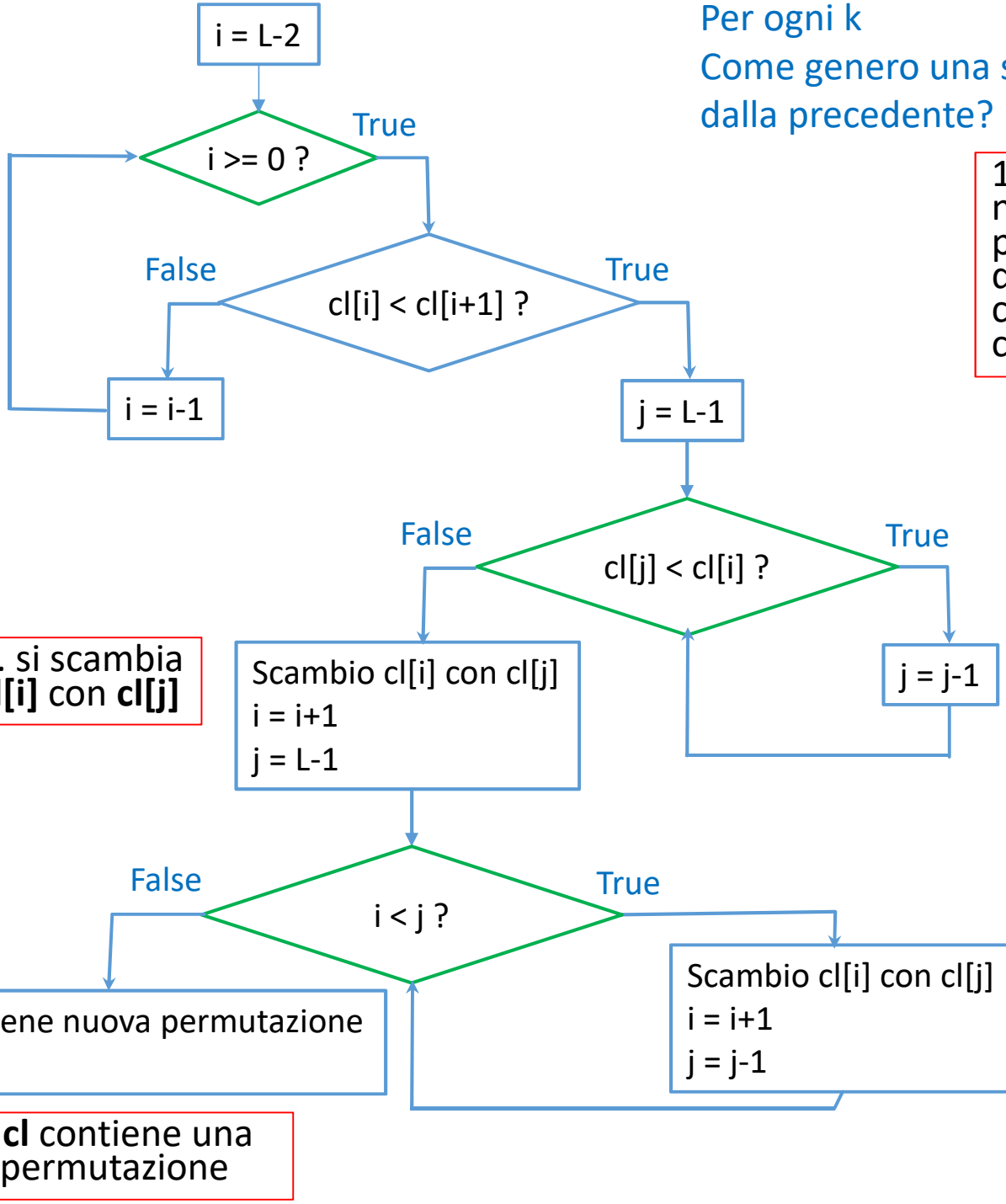
- Strategia: Generare le permutazioni in ordine lessicografico crescente, per cui ogni nuova permutazione generata deve essere quella "minima" tra tutte quelle mancanti e "maggiori" di quella appena generata
- La stringa non deve contenere caratteri ripetuti e deve avere lunghezza >1 (gli altri casi vanno gestiti come casi particolari)
- In tutto ci sono fact(L) permutazioni (con L lunghezza della stringa)
- Detta cl la lista di caratteri della stringa corrente devo modificare cl in modo di ottenere la nuova lista che, concatenata, darà la nuova stringa
- Per ogni k, come genero una stringa a partire dalla precedente?

1. 'abcd'
2. 'abdc'
3. 'acbd'
4. 'acdb'
5. 'adbc'
6. 'adcb'
7. 'bacd'
8. 'badc'
9. 'bcad'
10. 'bcda'
11. 'bdac'
12. 'bdca'
13. 'cabd'
14. 'cadb'
15. 'cbad'
16. 'cbda'

...



Per ogni k
Come genero una stringa a partire dalla precedente?



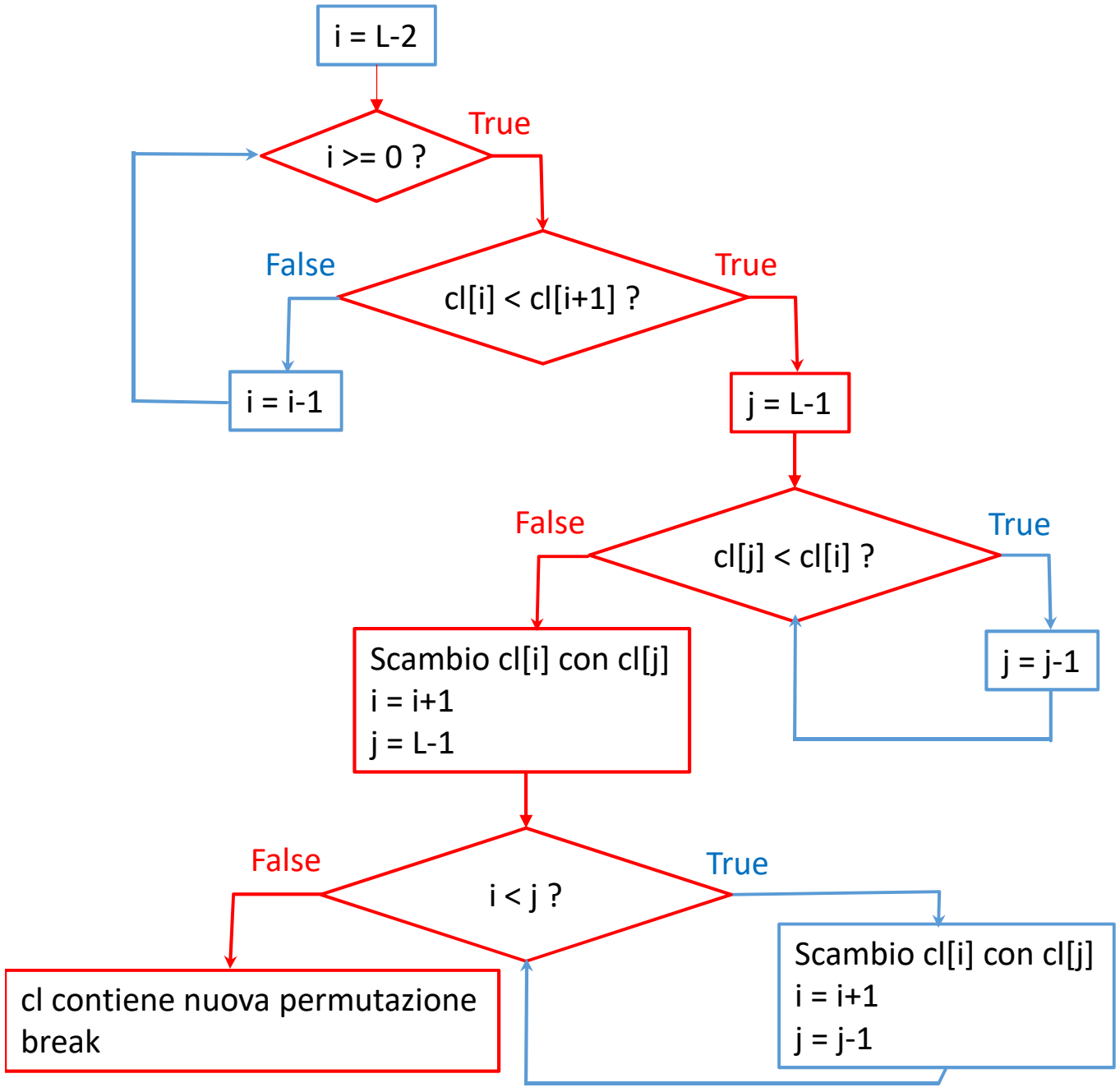
Per ogni k
Come genero una stringa a partire dalla precedente?

1. procedendo a ritroso nella lista **cl**, partendo da i penultimo indice (i+1 quindi ultimo indice) si cerca la prima coppia tale che cl[i] < cl[i+1]

2. Procedendo a ritroso nella lista **cl**, partendo dall'ultimo elemento j, si cerca il primo elemento **cl[j]** che è minore di **cl[i]**

3. si scambia **cl[i]** con **cl[j]**

4. si inverte il contenuto della porzione di lista che va dalla posizione **i** (esclusa) alla fine della lista



k=0
cl =['a', 'b', 'c', 'd']

i = 2

2 >= 0 True

c < d True

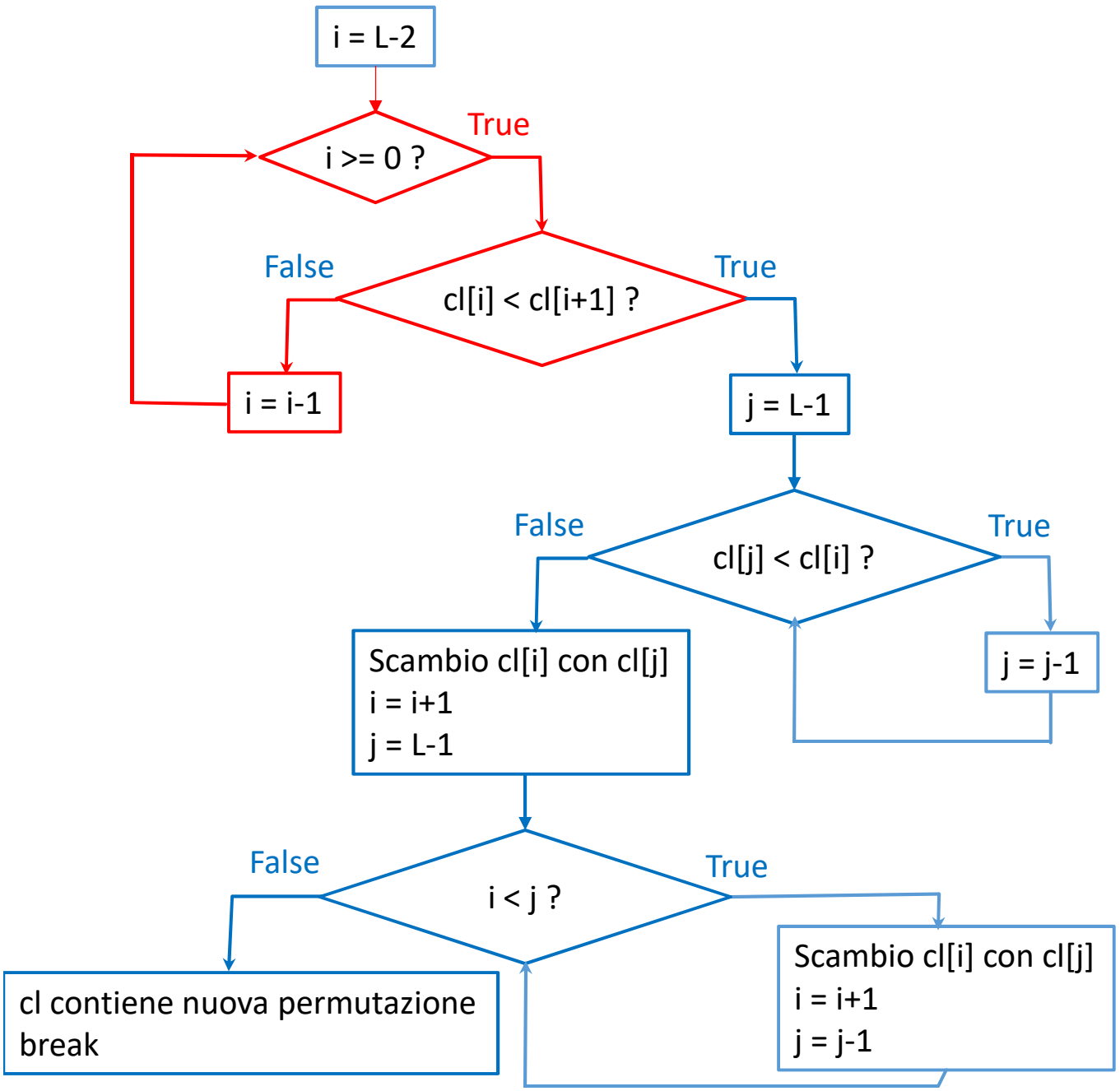
j = 3

d < c False

cl =['a', 'b', 'd', 'c']
i = 3
j = 3

i < j False

cl =['a', 'b', 'd', 'c']



k=5

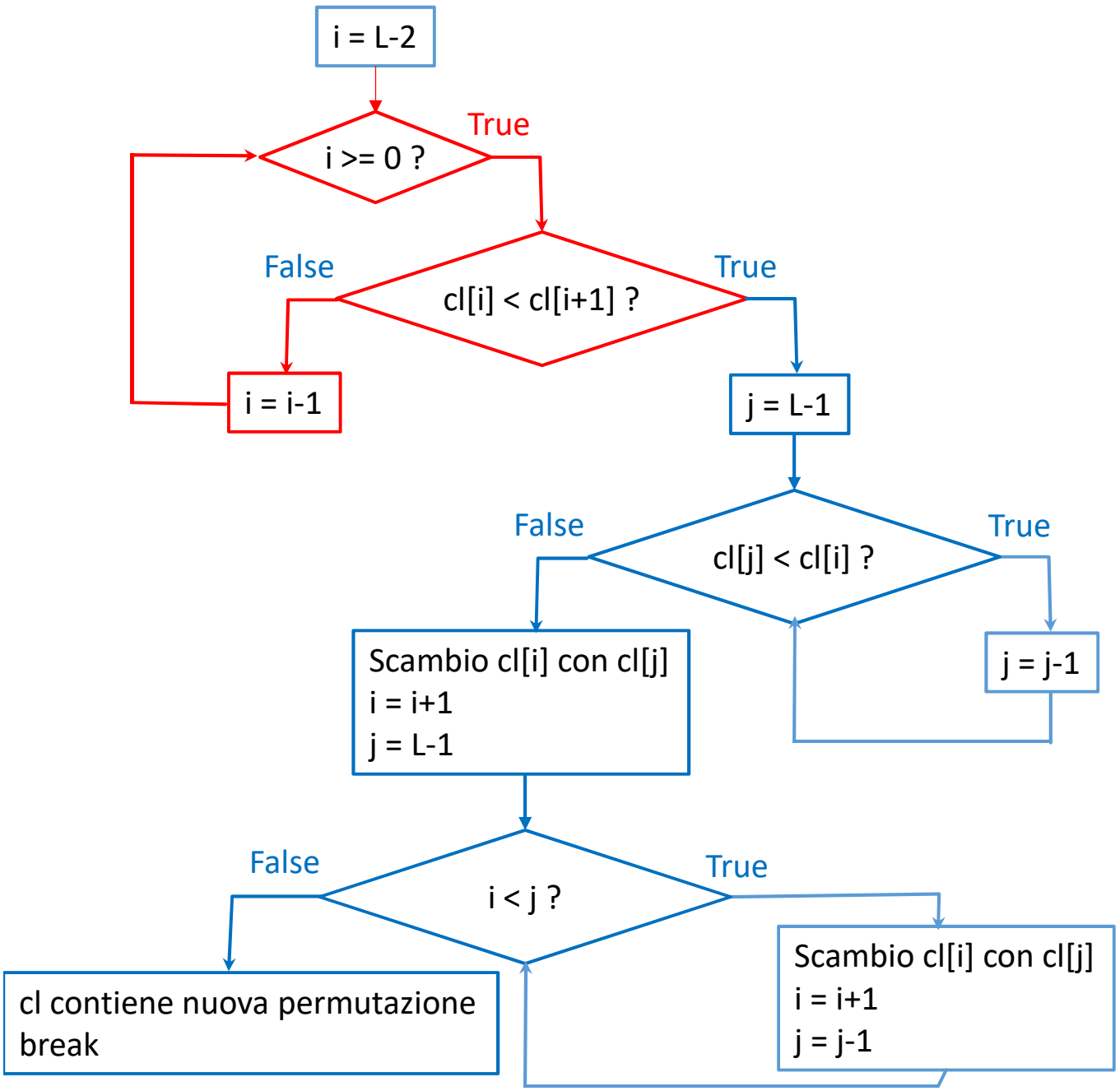
cl =['a', 'd', 'c', 'b']

i = 2

2 >= 0 True

c < b False

i = 1



k=5

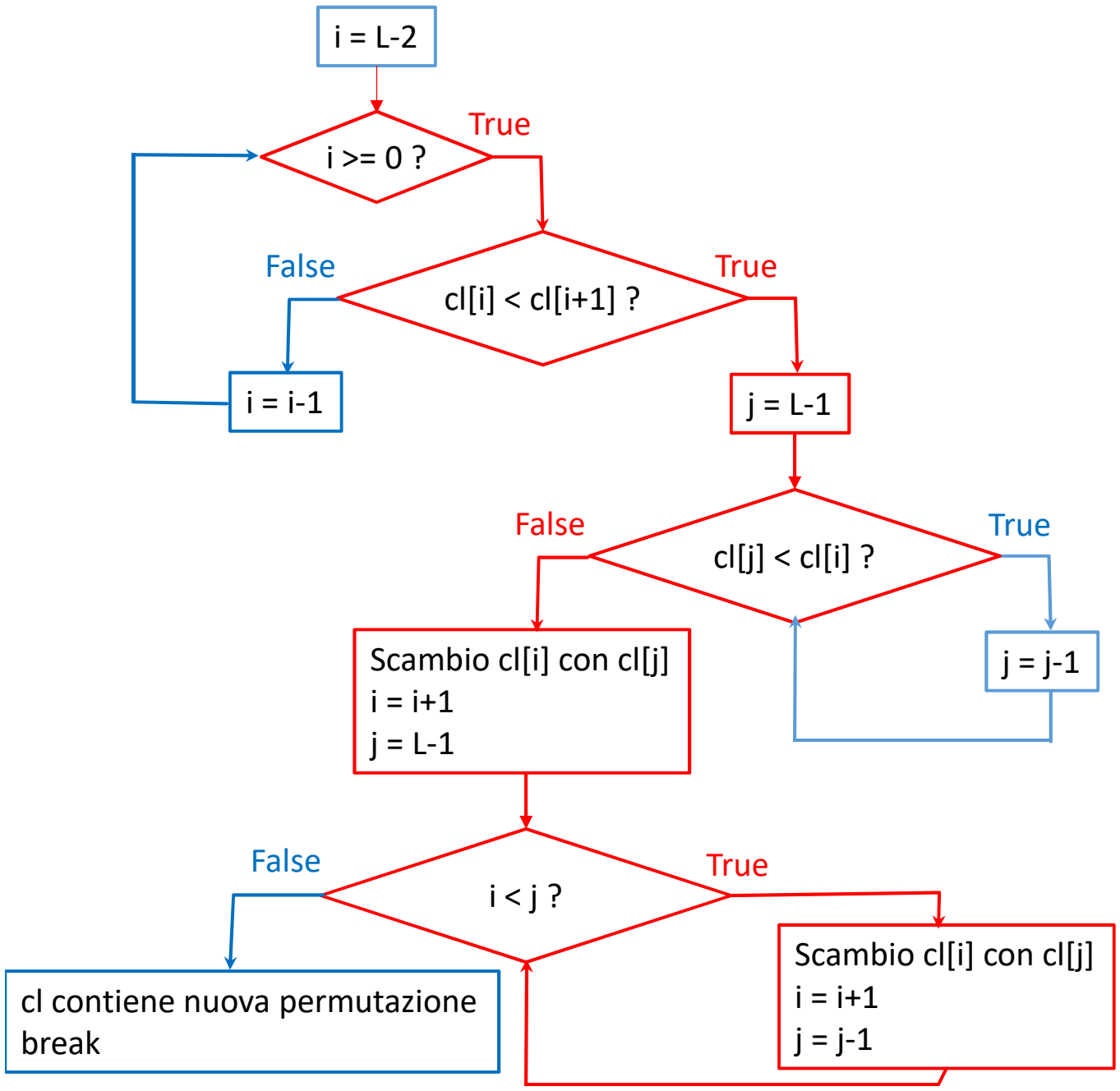
cl =['a', 'd', 'c', 'b']

i = 1

1 >= 0 True

d < c False

i = 0



k=5

cl =[‘a’, ‘d’, ‘c’, ‘b’]

i = 0

0 >= 0 True

a < d True

j = 3

b < a False

cl =[‘b’, ‘d’, ‘c’, ‘a’]

i = 1

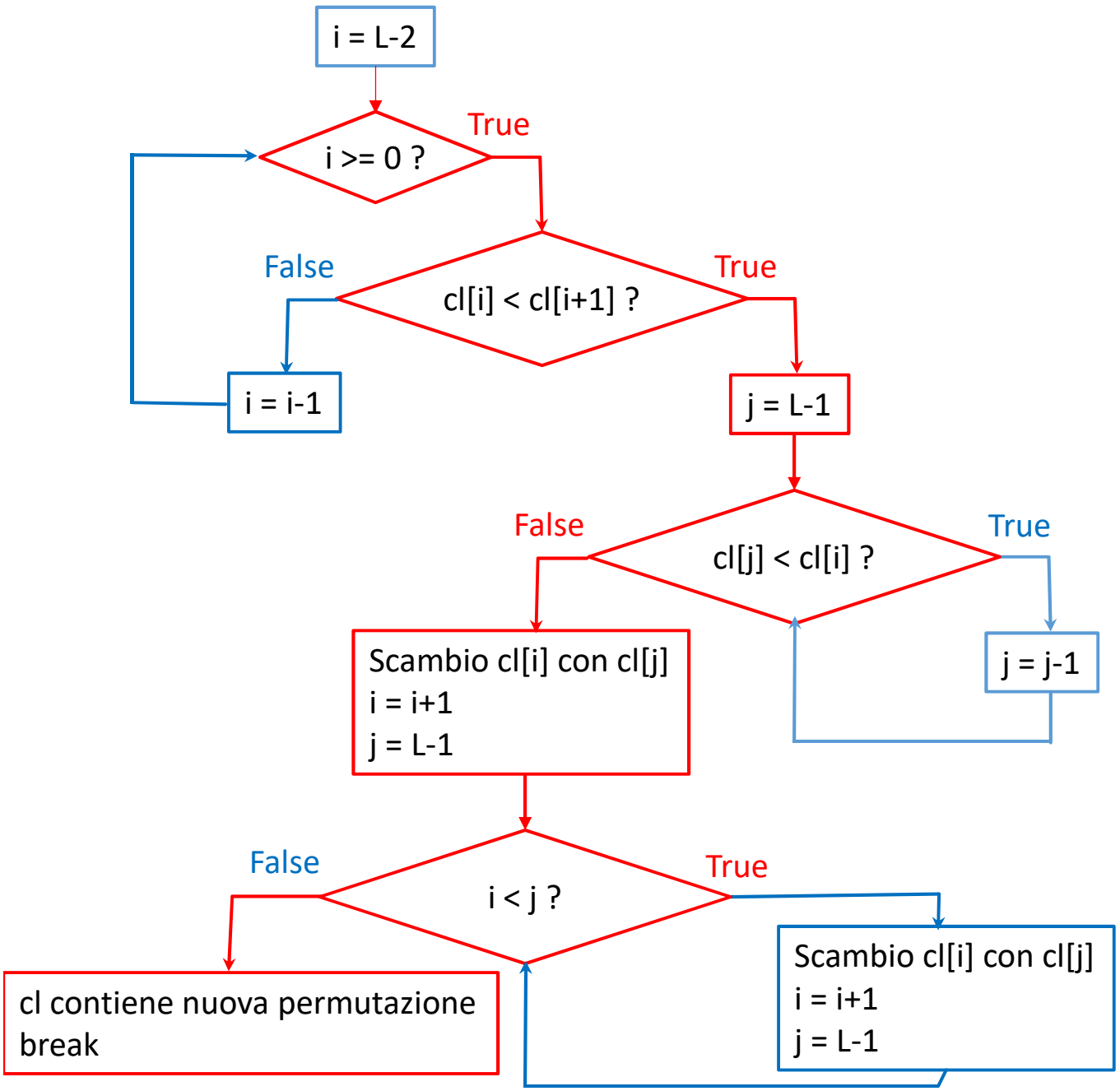
j = 3

i < j True

cl =[‘b’, ‘a’, ‘c’, ‘d’]

i = 2

j = 2



k=5
cl =[‘a’, ‘d’, ‘c’, ‘b’]

cl =[‘b’, ‘a’, ‘c’, ‘d’]
i = 2
j = 2
i < j False

cl =[‘b’, ‘a’, ‘c’, ‘d’]

Lab 9 – Es 5 (1/2)

Scrivere il programma **LCS2.py** che, dopo aver chiesto all'utente di fornire due stringhe (in generale, quindi, due righe di testo), identifichi la più lunga sequenza di caratteri appartenente alle due stringhe. Una sequenza di caratteri appartiene a una stringa se la stringa contiene i caratteri della sequenza nell'ordine corretto, anche se con altri caratteri interposti. Si tratta di un problema classico della bioinformatica (https://en.wikipedia.org/wiki/Longest_common_subsequence_problem).

Esempio: la più lunga sequenza di caratteri comune alle due stringhe **PippoPluto2Paperino** e **MimoAiuzzto2Zo** è **iouto2o**.

Suggerimento: identificare un algoritmo ricorsivo (a ricorsione doppia).

Attenzione perché il problema, apparentemente semplice, richiede tempi di calcolo relativamente lunghi... per risolvere l'esempio qui riportato, su un computer del nostro ufficio servono quasi 2 minuti e la funzione ricorsiva viene invocata... più di 340 milioni di volte!

Lab 9 – Es 5 (2/2)

Rendere, poi, più generale il programma in modo che identifichi la più lunga sequenza di caratteri appartenente a tre stringhe fornite dall'utente, scrivendo il programma **LCS3.py**.

Rendere, poi, ancora più generale il programma in modo che identifichi la più lunga sequenza di caratteri appartenente a tutte le stringhe di un insieme di dimensione qualunque, scrivendo il programma **LCSmany.py**, che acquisisce righe di testo come singole stringhe e continua finché non viene letta una riga vuota, senza visualizzare alcun messaggio di richiesta.

Ovviamente questo nuovo programma sostituisce i precedenti, in quanto funziona anche con due o tre stringhe.

Lab 9 – Es 6

Scrivere un programma che legga il file ***sample_mbox.txt*** (disponibile sulla pagina moodle del corso) e indichi quanti messaggi vengono ricevuti nelle diverse ore (senza considerare i minuti quindi) del giorno.

Il programma cerca le righe che cominciano con ‘From’ (non 'From:'') e crea un dizionario che memorizza le ore e quante volte sono apparse. Una volta che il dizionario è stato creato, ordinare in base alle ore e utilizzare un ciclo for per stampare il numero di messaggi ricevuti per i diversi orari.

Suggerimento:

Quando trovate una riga che comincia con ‘From ’ (ATTENZIONE: non ‘From:’) come la riga

```
From 157977224092161647 8@xxx Thu Sep 28 08:43:31 +0000 2017
```

usate **split()** e …

Lab 9 – Es 7

Scrivere, usando una o più funzioni, un programma che legga un file con la struttura del file ***sample_mbox.txt*** e trovi chi ha scritto il maggior numero di messaggi. Il programma cerca le righe che cominciano con 'From:' e crea un dizionario che memorizza gli indirizzi dei mittenti racchiusi tra i simboli < > e quante volte sono apparsi nel file. Una volta che il dizionario è stato creato, il programma lo legge e trova il mittente che ha inviato il maggior numero di email.

Suggerimento: usate le regular expression

Lab 9 – Es 8

Scrivere il programma **findFile.py** che

- chiede all'utente una stringa da cercare
- chiede all'utente il nome di una cartella del filesystem (che può essere un percorso assoluto o relativo) in cui iniziare la ricerca; se l'utente fornisce la stringa vuota, la cartella sarà quella in cui il programma viene eseguito (esattamente come se l'utente avesse scritto il nome speciale ".")
- visualizza il percorso completo (cartella + nome del file) di tutti i file il cui nome contiene la stringa da cercare

Lab 9 – Es 9

Scrivere il programma **findEmptyDir.py** che

- chiede all'utente il nome di una cartella del file system (che può essere un percorso assoluto o relativo) in cui iniziare la ricerca; se l'utente fornisce la stringa vuota, la cartella sarà quella in cui il programma viene eseguito (esattamente come se l'utente avesse scritto il nome speciale ".")
- visualizza i nomi delle cartelle che non contengono altre cartelle né file (cioè sono cartelle vuote)

Lab 9 – Es 10

Scrivere il programma **spellcheck.py** che implementi un correttore ortografico per un file di testo in inglese:

- chiede all'utente il nome del file di cui occorre verificare l'ortografia
  - se l'utente scrive la stringa vuota, si usi il testo **American Fairy Tales by L. Frank Baum**, reperibile all'indirizzo http://www.gutenberg.org/cache/epub/4357/pg4357.txt
  - se l'utente scrive una stringa che inizia con **http://** oppure **https://**, il file deve essere scaricato da Internet
  - se il file si trova nel file system locale (la stringa può essere il solo nome, un percorso relativo o assoluto)
- chiede all'utente il nome del file contenente il dizionario di parole valide (il formato del file deve essere "una parola per riga")
  - se l'utente scrive la stringa vuota, si usi il dizionario reperibile all'indirizzo https://users.cs.duke.edu/~ola/ap/linuxwords
  - se l'utente scrive una stringa che inizia con **http://** oppure **https://**, il file deve essere scaricato da Internet
  - se il file si trova nel file system locale (la stringa può essere il solo nome, un percorso relativo o assoluto)
- tutte le parole lette dal dizionario vanno convertite in minuscolo
- dopo aver convertito il testo in minuscolo, isola le singole parole, definite come sequenze di lettere dell'alfabeto inglese separate da almeno un carattere che non sia una lettera, e le inserisce in una lista
- ordina lessicograficamente la lista, in modo che eventuali (e probabili) parole duplicate si dispongano in posizioni consecutive della lista
- a partire dalla lista ordinata di parole, genera una nuova lista che sia priva di parole duplicate, in modo che contenga un solo esemplare di ciascuna parola presente nel testo
- visualizza tutte e sole le parole di tale lista che non sono presenti nel dizionario delle parole valide: sono i probabili errori di ortografia

Lab 9 – Es 11

SENZA SCRIVERE CODICE, progettare un algoritmo che consenta di disporre valori in una lista in modo da rendere massimo il numero di scambi richiesti dall'algoritmo di ordinamento per selezione (individuandone, così, un "caso pessimo"), quando questo venga utilizzato, appunto, per ordinare la lista (detto in altre parole, l'algoritmo da progettare deve "mescolare" gli elementi nella lista in modo che sia massimo il tempo richiesto per il suo ordinamento mediante selection sort). Per semplicità, si supponga che i valori presenti nella lista siano tutti distinti.

Si osservi che il caso pessimo per l'ordinamento per selezione NON è quello in cui i dati sono originariamente ordinati in senso decrescente all'interno della lista: in tal caso, infatti, il numero di scambi necessario per ordinare una lista di dimensione n è n/2 (se n è pari) oppure (n-1)/2 (se n è dispari). Perché?

Lab 9 – Es 12 (1/2)

Scrivere il programma **letterfreq.py** che calcoli la frequenza delle singole lettere dell'alfabeto all'interno di un file di testo (un'informazione utile nell'ambito della crittografia):

- chiede all'utente il nome del file da elaborare
  - se l'utente scrive la stringa vuota, si usi il testo **American Fairy Tales by L. Frank Baum**, reperibile all'indirizzo http://www.gutenberg.org/cache/epub/4357/pg4357.txt
  - se l'utente scrive una stringa che inizia con **http://** oppure **https://**, il file deve essere scaricato da Internet
  - altrimenti, il file si trova nel file system locale (e la stringa può essere il solo nome, oppure un percorso relativo o assoluto)
- nel testo da elaborare, dopo averlo convertito in maiuscolo, conta le occorrenze delle singole lettere (ignorando tutti i caratteri che non sono lettere)
- visualizza una tabella di due colonne: nella prima colonna vanno riportate le lettere dell'alfabeto inglese, in ordine lessicografico (cioè, in questo caso, alfabetico), nella seconda colonna la percentuale di lettere del testo che sono uguali a quella indicata in tale riga, riportando i dati con due cifre decimali nella parte frazionaria, allineati a destra

Lab 9 – Es 12 (2/2)

Esempio:

```
A   1.32%
B  12.20%
C   0.01%
D   3.00%
...
```

Osservate come solitamente nei testi in lingua inglese la lettera che ricorre più frequentemente sia la **e**, mentre nei testi in lingua italiana la "competizione" sia accesa tra le lettere **e** e **a**.