Evaluation du pouvoir prédictif, modèle de Nussinov

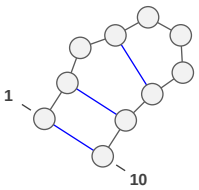
Parsing des structures secondaires

On souhaite extraire une liste de paires de bases à partir de la notation parenthésée utilisée classiquement pour dénoter une structure secondaire. Dans une telle notation, des positions sont appariées si elles apparaissent comme des parenthèses correspondantes.

On implémentera une fonction `parseSS(struct)` qui prend en argument une chaîne de caractère `struct` et renvoi une liste de paires de bases ainsi que le nombre de positions dans la structure.

Exemple :

Entrée		Sortie
`parsePK("((.(...)))")`	⟶	`([(0,9),(1,8),(3,7)],10)`



Échauffement

On codera ensuite une fonction `displaySecStr` qui prend en argument une liste `l` de paires de bases et une taille `n`, et renvoie la séquence bien parenthésée associée. On numérotera les bases en partant de 0.

Exemple :

Entrée		Sortie
`displaySS([(2,8),(3,7)],10)`	⟶	`"..((...))."`

On vérifiera la correction de cette fonction grâce à sa fonction *inverse* `parseSS` implémentée précédemment. En particulier, pour toute séquence bien parenthésée `seq`, on devrait avoir `displaySS(parseSS(seq))=seq`.

Comptage

On codera, à partir d'une version simplifiée (On fixera [1] $\theta = 1$) de l'algorithme de Nussinov, une fonction `countSS` permettant de compter le nombre de structures compatibles avec une séquence d'ARN. La fonction prendra deux arguments : la séquence `r` d'ARN et un booléen `debug` optionnel. Si `debug` est à `False`, alors une paire de base ne sera autorisée à se former que si ses bases sont comprises dans `[(C,G),(A,U),(G,U)]`. Si `debug` est à `True`, alors toute paire de base sera autorisée (Seule la taille `len(r)` compte). Exemple :

Entrée		Sortie
`countSS("A")`	⟶	1
`countSS("CAG")`	⟶	2
`countSS("CAGU")`	⟶	3
`countSS("AAAA",True)`	⟶	4
`countSS("AAAAAAAA",True)`	⟶	82

On remarque que le mode `debug==True`, en levant la contrainte de compatibilité des bases appariées, peut être réinterprété comme une énumération les structures secondaires combinatoires. Or ces dernières ont été comptées par Waterman en 1978, et vous pourrez donc vérifier la correction de votre implémentation en comparant les nombres obtenus pour des séquences de taille `len(r)` croissante avec les nombres `S` ci-dessous :

`len(r)`	1	2	3	4	5	6	7	8	9	10	11	12
`S`	1	1	2	4	8	17	37	82	185	423	978	2283

Vous utiliserez ces nombres pour vérifier votre implémentation.

1. On rappelle que θ correspond au nombre minimal de bases non-appariées entre deux bases appariées.

Nussinov

On adaptera (après sauvegarde) le code de la fonction `countSS` en une fonction `fillMatrix`, qui précalcule l'énergie de la structure d'énergie minimale. Au passage, on prendra en compte une valeur générale pour θ. Cette fonction prendra en paramètre une séquence `r` et un paramètre optionnel `theta` (Défaut : 3), et renverra une liste de liste `tab` telle que `tab[i][j]` est l'énergie minimale d'un repliement sur l' intervalle `[i,j]`.
Exemple :

```
Entree : fillMatrix(CCCCUUUUGGGGG,3)
Sortie :
tab = [[ 0.  0.  0.  0.  0.  0.  0.  0. -1. -2. -3. -4. -5.]
       [ 0.  0.  0.  0.  0.  0.  0.  0. -1. -2. -3. -4. -4.]
       [ 0.  0.  0.  0.  0.  0.  0.  0. -1. -2. -3. -3. -4.]
       [ 0.  0.  0.  0.  0.  0.  0.  0. -1. -2. -2. -3. -3.]
       [ 0.  0.  0.  0.  0.  0.  0.  0. -1. -1. -2. -2. -3.]
       [ 0.  0.  0.  0.  0.  0.  0.  0.  0. -1. -1. -2. -2.]
       [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1. -1. -2.]
       [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1. -1.]
       [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
       [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
       [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
       [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
       [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

On implémentera en outre une fonction `traceback` qui reconstruit les paires de bases de la structure d'énergie minimale. Cette fonction prendra en argument une liste de liste `tab` spécifiée ci-dessus, et renverra une liste de paires de bases correspondant au repliement d'énergie minimale.
Exemple :

Entrée		Sortie
`traceback(tab)`	$\longrightarrow$	`[(0,12),(1,11),(2,10),(3,9),(4,8)]`
`displaySS(traceback(tab))`	$\longrightarrow$	`"(((((...)))))"`

On combinera ces deux fonctions en une fonction `nussinov` qui prendra une séquence d'ARN en argument et renverra la structure d'énergie minimale dans le modèle de Nussinov.

Interlude

On souhaite comparer les performances prédictives de notre notre algorithme de Nussinov, utilisant un modèle d'énergie simple(-iste), à l'implémentation la plus récente de l'algorithme `MFold` vu en cours. Pour cela on va utiliser le Vienna package, une bibliothèque d'outils dédiés à l'ARN et maintenue à Vienne par Ivo Hofacker, pour évaluer la précision. On y trouve entre autres `RNAEval`, qui permet de calculer l'énergie d'un couple séquence/structure secondaire selon des paramètres récents du modèle de Turner, et `RNAFold` qui calcule par programmation dynamique la structure d'énergie minimale dans ce même modèle.
On met à votre disposition deux *wrappers*, c'est à dire des fonctions `runRNAEval` et `runRNAFold` telles que :

- `runRNAFold(seq)` prend en argument une séquence `seq` et renvoie un couple `(mfe,E)`, où `mfe` est le repliement d'énergie minimale pour `seq` sous la forme d'une liste de paire de base, et `E` son énergie.
- `runRNAEval(seq,str)` prend en argument une séquence `seq` et une structure secondaire `str`, et renvoie l'énergie associée.

Vous téléchargerez ces interfaces à l'adresse :

http ://www.lix.polytechnique.fr/~ponty/enseignement/ViennaWrapper.py

Différences des modèles

Implémentez une fonction `delta`, qui prend en argument une séquence d'ARN ω et renvoie la différence entre les énergies des structures respectivement prédites par votre implémentation `nussinov` et par `RNAFold`.
Expliquez pourquoi celle-ci est toujours positive ?

Performances

On codera tout d'abord une fonction `compareSS`, qui prend en argument deux structures secondaires S et S' (Listes de paires de bases), et renvoie le nombre de paires de bases communes à S et S'.

On utilisera cette fonction pour implémenter une fonction `benchmark`, qui prend une séquence d'ARN ω en entrée et sa structure native (connue) S, et renvoie la proportion des paires de bases correctement prédites par les algorithmes `nussinov` et `RNAFold` sur toutes les séquences présente dans une base de structures réunies par D. H. Mathews, téléchargeable à l'URL :

http ://www.lix.polytechnique.fr/~ponty/enseignement/MathewsRNASorted.faa

Rappels sur le langage `python`

Avant propos : Indentation

En `python`, les espaces son **significatifs**, c'est à dire qu'ils interviennent dans la syntaxe du langage. De nombreuses erreurs de compilations, voir des bugs, que vous rencontrerez dans vos codes seront dus à cet aspect un peu déroutant de `python`, qui a pour unique mérite d'imposer une discipline dans l'indentation.

Plus précisément, l'appartenance d'instructions consécutives à un même **bloc** exige un même niveau d'indentation.

```
for i in [1,2,3]:
  x = i
print x
# Sortie : 3
```

```
for i in [1,2,3]:
  x = i
  print x
# Sortie : 1,2,3
```

De plus, comme `python` ne peut pas deviner quel va être le style d'indentation utilisé, il exige que toute instruction précédant un bloc (i.e. finissant par le caractère `:`) soit effectivement suivie d'un changement dans l'indentation.

```
for i in [1,2,3]:
x = i
print x
# Erreur de compilation
```

Dans des cas exceptionnels, il peut s'avérer utile de créer un bloc sans instruction associée [2]. Il est alors possible d'insérer l'**instruction vide** `pass` afin de satisfaire les exigences syntaxiques du compilateur sans modifier l'exécution du programme.

```
for i in [1,2,3]:
  pass
x = i
print x
# Passe la compilation
# Sortie : 3
```

Instructions conditionnelles

`if then else`

On retrouve en `python` le mécanisme classique des `if then else`.

2. En général, cela révélera plutôt une erreur dans la conception de l'algorithme (Ex : Instruction `if... then ... else` sans instruction dans le `if`), on examinera donc soigneusement le code plutot que d'abuser de la commande `pass`

```
msg = ""
if b:
  msg = "OK"
else:
  msg = "KO"
print msg
# Sortie : "OK" si b==True  (ou b!=0)
#          "KO" si b==False (ou b==0)
```

La clause `else` peut être omise si nécessaire.

```
msg = "KO"
if b:
  msg = "OK"
print msg
# Sortie : "OK" si b==True  (ou b!=0)
#          "KO" si b==False (ou b==0)
```

Enfin, les clauses `if then else` peuvent être enchaînées sans rajouter de niveau d'indentation grâce à l'instruction `elif`.

```
if a==1:
  ...
else:
  if a==2:
    ...
  else:
    if a==3:
      ...
    else:
      ...
```

```
if a==1:
  ...
elif a==2:
  ...
elif a==3:
  ...
else:
  ...
```

Boucles

while

Commençons par la boucle `while`, dont la syntaxe est très légère.

```
i = 0
while i<=10:
  print i
  i += 1
# Sortie : 1 2 3 4 5 6 7 8 9 10
```

for

La syntaxe des boucles `for` est un peu plus insolite, puisque l'itération se fait uniquement sur les valeurs d'une liste. On ne manipule donc pas directement, la plupart du temps, les

indices d'une boucle, ce qui a souvent le mérite d'éviter pas mal d'erreurs de bornes.

```
bases = ["A","C","G","U"]
for b in bases:
  print b
# Sortie : A C G U
```

range

Évidemment, il est des situations où l'on ne pourra pas facilement se passer d'une itération sur des indices. Dans ce cas, on créera une liste contenant les indices sur lesquels itérer avec la commande `range`, qui adopte différents comportements selon le nombre d'arguments. Pour un unique argument `n`, la commande `range(n)` retourne une liste composée des entiers de 0 à $n-1$.

```
for b in range(10):
  print b
# Sortie : 0 1 2 3 4 5 6 7 8 9
```

Pour deux arguments `l` et `h`, la commande `range(l,h)` retourne une liste composée des entiers de l à $h-1$.

```
for b in range(2,10):
  print b
# Sortie : 2 3 4 5 6 7 8 9
```

Enfin, on pourra spécifier un troisième argument `step` afin de définir l'incrément de la boucle, c'est à dire la valeur ajoutée au compteur de boucle après chaque itération.

```
for b in range(2,10,2):
  print b
# Sortie : 2 4 6 8
```

Listes

Les listes sont LA structure de donnée centrale à `python`, au point qu'elle remplacent par défaut les tableaux, qui ne sont accessibles que grace à l'inclusion de bibliothèque spécialisées.

.1 Initialisation

Les listes sont initialisées de façon très souples grace à la syntaxe `[...]`.

```
ln = [1,4,8,32]
ls = ["bla","bli","blu"]
ll = [ln,ls]
print ll
# Sortie : [[1,4,8,32],["bla","bli","blu"]]
```

Pour initialiser une liste de façon compacte, afin par exemple de simuler un tableau, on pourra utiliser la syntaxe *fonctionnelle* suivante.

```
l = [2*i for i in range(5)]
print l
# Sortie : [0,2,4,6,8]
```

`len`

On récupèrera la taille d'une liste grâce à la commande `len`.

```
l = [1,2,3,8]
print len(l)
# Sortie : 4
```

Concaténation

L'opérateur d'addition `+` est naturellement surchargé sur les listes en une concaténation.

```
l1 = [1,2]
l2 = [3,4]
l  = l1 + l2
print l
# Sortie : [1,2,3,4]
```

Extraction

L'extraction d'un élément est réalisée grâce à la très classique notation `[...]`. Comme en C, les indices d'une liste de taille n vont de 0 à $n-1$

```
l = ["a","b","c","d","e"]
print l[0],l[2]
# Sortie : a c
```

Par ailleurs, on pourra utiliser des indices négatifs afin d'adresser avec une syntaxe allégée les derniers éléments de la liste. Par exemple, `l[-1]` et `l[len(l)-1]` représentent toutes deux le dernier élément de la liste.

```
l = ["a","b","c","d","e"]
print l[-1],l[-2]
# Sortie : e d
```

Sous-listes

L'extraction d'un sous-ensemble d'éléments consécutifs dans la liste est réalisée grâce à une notation `l[a:b]`, qui renvoie une copie de la liste composée des élements de `l` d'indices a à $b-1$.

```
l = ["a","b","c","d","e"]
print l[1:4]
# Sortie : ["b","c","d"]
```

Chacune des deux bornes de la sous-listes peuvent être ignorées, auquel cas la sous-liste sera un préfixe, un suffixe ou une copie de la liste de départ selon le cas.

```
l = ["a","b","c","d","e"]
print l[:3], l[3:], l[:]
# Sortie : ["a","b","c"] ["d","e"] ["a","b","c","d","e"]
```

De même que pour l'extraction de coefficients, on pourra utiliser des indices négatifs afin, par exemple, de filtrer des éléments de la queue de liste.

```
l = ["a","b","c","d","e"]
print l[:-1]
# Sortie : ["a","b","c","d"]
```

Ajout

Il est possible d'ajouter un élément `e` à une liste `l` grace à l'évocation de la méthode `l.append(e)`.

```
l = ["a","b"]
l.append("c")
print l
# Sortie : ["a","b","c"]
```

Suppression du dernier élément

On peut facilement utiliser une liste `l` pour simuler une pile à travers l'évocation de la méthode `pop()`, qui renvoie puis supprime le dernier élément de la liste.

```
l = ["a","b"]
l.append("c")
print l
# Sortie : ["a","b","c"]
```

T-uplets

Les t-uplets sont des structures de données légères et souples qui simplifieront la manipulation de couples, triplets, ...

Initialisation

On initialise un t-uplet grâce à la syntaxe `(..,..)`.

```
c = (1,2)
print l
# Sortie : (1,2)
```

A noter qu'il est aussi possible, grâce à cette syntaxe, d'affecter plusieurs variables simultanément, économisant ainsi des affectations multiples.

```
(i,j) = ("a",3)
print i,j
# Sortie : a 3
```

Fonctions/procédures

def

Les fonctions et procédures sont définies par la commande `def`.

```
def f():
  print "a"
f()
# Sortie : a
```

Une valeur de retour peut être spécifiée à travers la commande `return`.

```
def f():
  return "a"
print f()
# Sortie : a
```

Des arguments (sans type) peuvent être spécifiés.

```
def f(i):
  return i*(i-1)
print f(10)
# Sortie : 90
```

Certains des arguments peuvent se voir associer une valeur par défaut.

```
def f(i,j=0):
  return i*j
print f(10),f(10,1)
# Sortie : 0 10
```

Bibliothèques

Il existe deux mécanismes pour importer du code extérieur au code source actuel.

`import`

Permet l'importation d'un fichier, qui rend visible un objet bibliothèque. Les méthodes ne sont pas directement visibles, et nécessite d'être préfixées par l'objet bibliothèque correspondant.

```
1 import math
2 print math.sqrt(4)
3 # Sortie : 2.0
```

Notez que les fonctions définies par math ne sont pas accessibles directement.

```
1 import math
2 print sqrt(4)
3 # Erreur de compilation
```

`from ... import`

Permet l'importation et la mise à disposition de fonctions définies dans une bibliothèque externe.

```
1 from math import sqrt, trunc
2 print trunc(sqrt(3))
3 # Sortie : 1
```

A noter qu'il est aussi possible (mais déconseillé, sous peine d'encombrer l'espace des noms) d'importer toutes les fonctions d'une bibliothèque via la commande `from ... import *`.

```
1 from math import *
2 print exp(log(5))
3 # Sortie : 4.999999999999991
```

Bibliothèque `sys`

Il est recommandé d'importer la bibliothèque `sys` via la commande `import sys`. Elle met à disposition, outre les fonctions utiles comme `sys.exit`, les paramètres de la ligne de commande, ce qui est utile pour les exécutables.

`sys.argv`

Liste des arguments de la ligne de commande. Par convention, le premier élément de cette liste est le nom du fichier exécutable. L'exemple ci-dessous est typique du rapatriement d'options pour un exécutable.

```
1 if __name__ == '__main__':
2   index = 1
3   while index<len(sys.argv):
4     if sys.argv[index] == "-a":
5       ...
6     elif sys.argv[index] == "-b":
```

```
      ...
    else:
      ...
    index += 1
```

sys.stdout/sys.stderr

Permet une écriture direct sur les sorties standard et d'erreur. Permet un formattage précis des données émises en évitant les retour-chariots et espaces ajoutés automatiquement par la commande `print`.

```
import sys
for i in range(1000001):
  sys.stdout.write("\r%s/%s    "%(i,1000000))
# Sortie : Décompte sur une ligne de 1 à 1000000
```

sys.exit

Permet d'interrompre le cours du programme et quitter l'interpréteur `python`. Il est possible de spécifier une valeur entière de retour à la fonction, auquel cas les conventions exigent une valeur de retour nulle si et seulement si l'exécution s'est déroulée correctement.

```
if __name__ == '__main__':
  if len(sys.argv)<1:
    print "Missing argument"
    sys.exit(1)
  else:
    # Fonctionnement normal
    ...
    sys.exit(0)
```

Exécutable

__main__

Toute commande contenue dans le bloc principal d'un fichier `python` (Bloc d'indentation nulle) est exécuté automatiquement lors du chargement. Cependant, il est fortement recommandé de ne pas utiliser ce bloc pour les instructions contenues dans votre programme principal (*main*). En effet, celui ci serait alors inutilisable en temps que bibliothèque de fonction par la suite.
Il existe donc un test permettant de distinguer le fichier exécuté des fichiers librairie. Plus précisément, une variable `__name__` se verra affecter la valeur `"__main__"` uniquement si le fichier courant est celui passé en argument du compilateur.

joe.py

```
def f():
  return "hi"
print "oh"
if __name__ == '__main__':
  print "ah"
# python joe.py => oh ah
```

jim.py

```
import joe
if __name__ == '__main__':
  joe.f()
# python jim.py => oh hi
```

Exemple complet

```
import os, sys

# Conversion notation parenthesee/paires de bases
def parseSS(secstr):
  result = []
  pile = []
  for i in range(len(secstr)):
    c = secstr[i]
    if c == "(":
      pile.append(i)
    elif c == ")":
      j = pile.pop()
      result.append((j,i))
    else:
      pass
  return result

if __name__ == '__main__':
  if len(sys.argv)<=1:
    print "Error: Missing argument!"
    sys.exit(1)
  secstr = sys.argv[1]
  bps = parseSS(secstr)
  print bps
  sys.exit(0)
```