Problem 1 -> Sequence Alignment algorithm

In [29]:

```
import biopython as bp
from Bio import pairwise2
from Bio.pairwise2 import format_alignment
```

In [30]:

```
# """
# Pairwise sequence alignment using a dynamic programming algorithm.

# This provides functions to get global and local alignments between two sequences. A g
lobal alignment finds the best concordance between all characters in two sequences. A l
ocal alignment finds just the subsequences that align the best.

# When doing alignments, you can specify the match score and gap penalties. The match s
core indicates the compatibility between an alignment of two characters in the sequence
s. Highly compatible characters should be given positive scores, and incompatible ones
 should be given negative scores or 0. The gap penalties should be negative.

# The names of the alignment functions in this module follow the convention <alignment
 type>XX where <alignment type> is either "global" or "local" and XX is a 2 character c
ode indicating the parameters it takes. The first character indicates the parameters fo
r matches (and mismatches), and the second indicates the parameters for gap penalties.

# The match parameters are:

# CODE  DESCRIPTION
# x     No parameters. Identical characters have score of 1, otherwise 0.
# m     A match score is the score of identical chars, otherwise mismatch
#       score.
# d     A dictionary returns the score of any pair of characters.
# c     A callback function returns scores.
# The gap penalty parameters are:

# CODE  DESCRIPTION
# x     No gap penalties.
# s     Same open and extend gap penalties for both sequences.
# d     The sequences have different open and extend gap penalties.
# c     A callback function returns the gap penalties.
# All the different alignment functions are contained in an object align. For example:
# """
# will return a list of the alignments between the two strings. For a nice printout, us
e the format_alignment method of the module:
```

In [31]:

```
# All alignment functions have the following arguments:

# Two sequences: strings, Biopython sequence objects or lists. Lists are useful for sup
plying sequences which contain residues that are encoded by more than one letter.
# penalize_extend_when_opening: boolean (default: False). Whether to count an extension
 penalty when opening a gap. If false, a gap of 1 is only penalized an "open" penalty,
 otherwise it is penalized "open+extend".
# penalize_end_gaps: boolean. Whether to count the gaps at the ends of an alignment. By
 default, they are counted for global alignments but not for local ones. Setting penali
ze_end_gaps to (boolean, boolean) allows you to specify for the two sequences separatel
y whether gaps at the end of the alignment should be counted.
# gap_char: string (default: '-'). Which character to use as a gap character in the ali
gnment returned. If your input sequences are lists, you must change this to ['-'].
# force_generic: boolean (default: False). Always use the generic, non-cached, dynamic
 programming function (slow!). For debugging.
# score_only: boolean (default: False). Only get the best score, don't recover any alig
nments. The return value of the function is the score. Faster and uses less memory.
# one_alignment_only: boolean (default: False). Only recover one alignment.
# The other parameters of the alignment function depend on the function called. Some ex
amples:

# Find the best global alignment between the two sequences. Identical characters are gi
ven 1 point. No points are deducted for mismatches or gaps.
```

In [32]:

```
alignments = pairwise2.align.globalxx("ACCGT", "ACG")
print(format_alignment(*alignments[0]))
```

```
ACCGT
| ||
A-CG-
  Score=3
```

In [33]:

```
for a in pairwise2.align.globalxx("ACCGT", "ACG"):
    print(format_alignment(*a))
```

```
ACCGT
| ||
A-CG-
  Score=3

ACCGT
|| |
AC-G-
  Score=3
```

In [34]:

```
# Same thing as before, but with a local alignment.
for a in pairwise2.align.localxx("ACCGT", "ACG"):
    print(format_alignment(*a))
```

```
ACCGT
| ||
A-CG-
  Score=3

ACCGT
|| |
AC-G-
  Score=3
```

In [35]:

```
# Do a global alignment. Identical characters are given 2 points, 1 point is deducted f
or each non-identical character. Don't penalize gaps.
for a in pairwise2.align.globalmx("ACCGT", "ACG", 2, -1):
    print(format_alignment(*a))
```

```
ACCGT
| ||
A-CG-
  Score=6

ACCGT
|| |
AC-G-
  Score=6
```

In [41]:

```
# Define two sequences to be aligned
X = "ACGGGT"
Y = "ACG"

# Get a list of the global alignments between the two sequences ACGGGT and ACG
# No parameters. Identical characters have score of 1, else 0.
# No gap penalties.
alignments = pairwise2.align.globalxx(X, Y)

# Use format_alignment method to format the alignments in the list
for a in alignments:
    print(format_alignment(*a))
```

```
ACGGGT
||  |
AC--G-
  Score=3

ACGGGT
|| |
AC-G--
  Score=3

ACGGGT
|||
ACG---
  Score=3
```

Problem 2 -> Random Shuffling of DNA Sequences

In [92]:

```
# NOTE: One cannot use function "count(s,word)" to count the number
# of occurrences of dinucleotide word in string s, since the built-in
# function counts only nonoverlapping words, presumably in a left to
# right fashion.

import numpy as np
import sys,string,random
from random import *

def computeCountAndLists(s):
  #WARNING: Use of function count(s,'UU') returns 1 on word UUU
  #since it apparently counts only nonoverlapping words UU
  #For this reason, we work with the indices.

  #Initialize lists and mono- and dinucleotide dictionaries
  List = {} #List is a dictionary of lists
  List['A'] = []; List['C'] = [];
  List['G'] = []; List['T'] = [];
  nuclList   = ["A","C","G","T"]
  s       = s.upper()
  s       = s.replace("T","T")
  nuclCnt    = {}  #empty dictionary
  dinuclCnt  = {}  #empty dictionary
  for x in nuclList:
    nuclCnt[x]=0
    dinuclCnt[x]={}
    for y in nuclList:
      dinuclCnt[x][y]=0

  #Compute count and lists
  nuclCnt[s[0]] = 1
  nuclTotal     = 1
  dinuclTotal   = 0
  for i in range(len(s)-1):
    x = s[i]; y = s[i+1]
    List[x].append( y )
    nuclCnt[y] += 1; nuclTotal  += 1
    dinuclCnt[x][y] += 1; dinuclTotal += 1
  assert (nuclTotal==len(s))
  assert (dinuclTotal==len(s)-1)
  return nuclCnt,dinuclCnt,List


def chooseEdge(x,dinuclCnt):
  numInList = 0
  for y in ['A','C','G','T']:
    numInList += dinuclCnt[x][y]
  z = random.random()
  denom=dinuclCnt[x]['A']+dinuclCnt[x]['C']+dinuclCnt[x]['G']+dinuclCnt[x]['T']
  numerator = dinuclCnt[x]['A']
  if z < float(numerator)/float(denom):
    dinuclCnt[x]['A'] -= 1
    return 'A'
  numerator += dinuclCnt[x]['C']
  if z < float(numerator)/float(denom):
    dinuclCnt[x]['C'] -= 1
    return 'C'
  numerator += dinuclCnt[x]['G']
  if z < float(numerator)/float(denom):
```

```
    dinuclCnt[x]['G'] -= 1
    return 'G'
  dinuclCnt[x]['T'] -= 1
  return 'T'


def connectedToLast(edgeList,nuclList,lastCh):
  D = {}
  for x in nuclList: D[x]=0
  for edge in edgeList:
    a = edge[0]; b = edge[1]
    if b==lastCh: D[a]=1
  for i in range(2):
    for edge in edgeList:
      a = edge[0]; b = edge[1]
      if D[b]==1: D[a]=1
  ok = 0
  for x in nuclList:
    if x!=lastCh and D[x]==0: return 0
  return 1


def eulerian(s):
  nuclCnt,dinuclCnt,List = computeCountAndLists(s)
  #compute nucleotides appearing in s
  nuclList = []
  for x in ["A","C","G","T"]:
    if x in s: nuclList.append(x)
  #compute numInList[x] = number of dinucleotides beginning with x
  numInList = {}
  for x in nuclList:
    numInList[x]=0
    for y in nuclList:
      numInList[x] += dinuclCnt[x][y]
  #create dinucleotide shuffle L
  firstCh = s[0]  #start with first letter of s
  lastCh  = s[-1]
  edgeList = []
  for x in nuclList:
    if x!= lastCh: edgeList.append( [x,chooseEdge(x,dinuclCnt)] )
  ok = connectedToLast(edgeList,nuclList,lastCh)
  return ok,edgeList,nuclList,lastCh


def shuffleEdgeList(L):
  n = len(L); barrier = n # n = randmax here in our program and we will run for loop fo
r n iterations.
  for i in range(n):
    z = int(np.random.rand() * barrier)
    print("z",z)
    tmp = L[z]
    L[z]= L[barrier-1]
    L[barrier-1] = tmp
    barrier -= 1
    print("barrier",barrier) #barrier runs till n i.e randmax
  return L
```

In [93]:

```
computeCountAndLists('a')
```

Out[93]:

```
({'A': 1, 'C': 0, 'G': 0, 'T': 0},
 {'A': {'A': 0, 'C': 0, 'G': 0, 'T': 0},
  'C': {'A': 0, 'C': 0, 'G': 0, 'T': 0},
  'G': {'A': 0, 'C': 0, 'G': 0, 'T': 0},
  'T': {'A': 0, 'C': 0, 'G': 0, 'T': 0}},
 {'A': [], 'C': [], 'G': [], 'T': []})
```

In [94]:

```
eulerian('a')
```

Out[94]:

```
(1, [], [], 'a')
```

In [96]:

```
shuffleEdgeList(['a','b','g','f','t','a','f','f','d'])
```

```
z 5
barrier 8
z 0
barrier 7
z 5
barrier 6
z 3
barrier 5
z 2
barrier 4
z 0
barrier 3
z 2
barrier 2
z 1
barrier 1
z 0
barrier 0
```

Out[96]:

```
['f', 'b', 't', 'f', 'g', 'f', 'd', 'a', 'a']
```