B363: Bioinformatics Algorithms

Problem 1

Counting DNA Nucleotides

A Rapid Introduction to Molecular Biology

Making up all living material, the **cell** is considered to be the building block of life. The **nucleus**, a component of most **eukaryotic** cells, was identified as the hub of cellular activity 150 years ago. Viewed under a light microscope, the nucleus appears only as a darker region of the cell, but as we increase magnification, we find that the nucleus is densely filled with a stew of macromolecules called **chromatin**. During **mitosis** (eukaryotic cell division), most of the chromatin condenses into long, thin strings called **chromosomes**. See **Figure 1** for a figure of cells in different stages of mitosis.

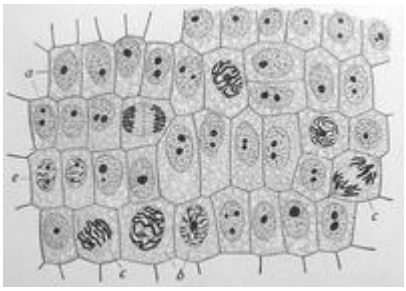

Figure 1. A 1900 drawing by Edmund Wilson of onion cells at different stages of mitosis. The sample has been dyed, causing chromatin in the cells (which soaks up the dye) to appear in greater contrast to the rest of the cell.

One class of the macromolecules contained in chromatin are called **nucleic acids**. Early 20th century research into the chemical identity of nucleic acids culminated with the conclusion that nucleic acids are **polymers**, or repeating chains of smaller, similarly structured molecules known as **monomers**. Because of their tendency to be long and thin, nucleic acid polymers are commonly called **strands**.

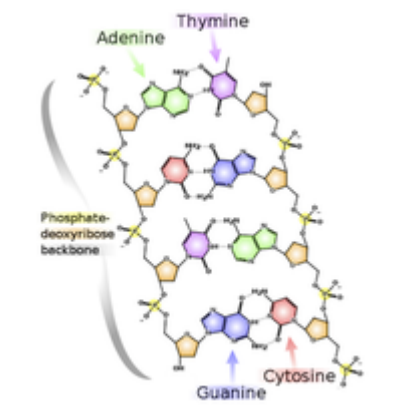

Figure 2. A sketch of DNA's primary structure.

The nucleic acid monomer is called a **nucleotide** and is used as a unit of strand length (abbreviated to nt). Each nucleotide is formed of three parts: a **sugar** molecule, a negatively charged **ion** called a **phosphate**, and a compound called a **nucleobase** ("base" for short). Polymerization is achieved as the sugar of one nucleotide bonds to the phosphate of the next nucleotide in the chain, which forms a **sugar-phosphate backbone** for the nucleic acid strand. A key point is that the nucleotides of a specific type of nucleic acid always contain the same sugar and phosphate molecules, and they differ only in their choice of base. Thus, one strand of a nucleic acid can be differentiated from another based solely on the *order* of its bases; this ordering of bases defines a nucleic acid's **primary structure**.

For example, **Figure 2** shows a strand of **deoxyribose nucleic acid** (DNA), in which the sugar is called **deoxyribose**, and the only four choices for nucleobases are molecules called **adenine** (A), **cytosine** (C), **guanine** (G), and **thymine** (T).

For reasons we will soon see, DNA is found in all living organisms on Earth, including bacteria; it is even found in many viruses (which are often considered to be nonliving). Because of its importance, we reserve the term **genome** to refer to the sum total of the DNA contained in an organism's chromosomes.

Problem

A **string** is simply an ordered collection of symbols selected from some **alphabet** and formed into a word; the **length** of a string is the number of symbols that it contains.

An example of a length 21 **DNA string** (whose alphabet contains the symbols 'A', 'C', 'G', and 'T') is "ATGCTTCAGAAAGGTCTTACG."

Given: A DNA string s of length at most 1000 nt.

Return: Four integers (separated by spaces) counting the respective number of times that the symbols 'A', 'C', 'G', and 'T' occur in s.

Sample Dataset

```
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
```

Sample Output

```
20 12 17 21
```

Problem 2

Find the Most Frequent Words in a String

We say that *Pattern* is a **most frequent *k*-mer** in *Text* if it maximizes *Count*(*Text*, *Pattern*) among all k-mers. For example, "ACTAT" is a most frequent 5-mer in "ACAACTATGCATCACTATCGGGAACTATCCT", and "ATA" is a most frequent 3-mer of "CGATATATCCATAG".

Frequent Words Problem

Find the most frequent k-mers in a string.

Given: A DNA string *Text* and an integer *k*.

Return: All most frequent *k*-mers in *Text* (in any order).

Sample Dataset

```
ACGTTGCATGTCGCATGATGCATGAGAGCT
4
```

Sample Output

```
CATG GCAT
```

Extra Datasets

Click for an extra dataset

Click for debug datasets

Problem 3

Find a Position in a Genome Minimizing the Skew

Define the **skew** of a DNA string *Genome*, denoted *Skew*(*Genome*), as the difference between the total number of occurrences of 'G' and 'C' in *Genome*. Let $Prefix_i$ (*Genome*) denote the **prefix** (i.e., initial substring) of *Genome* of length *i*. For example, the values of *Skew*($Prefix_i$ ("CATGGGCATCGGCCATACGCC")) are:

0 -1 -1 -1 0 1 2 1 1 1 0 1 2 1 0 0 0 0 -1 0 -1 -2

Minimum Skew Problem

Find a position in a genome minimizing the skew.

Given: A DNA string *Genome*.

Return: All integer(s) *i* minimizing *Skew*($Prefix_i$ (*Text*)) over all values of *i* (from 0 to |*Genome*|).

Sample Dataset

```
CCTATCGGTGGATTAGCATGTCCCTGTACGTTTCGCCGCGAACTAGTTCACACGGCTTGATGGCAAATGGTTTTTC
CGGCGACCGTAATCGTCCACCGAG
```

Sample Output

```
53 97
```

Extra Datasets

Click for an extra dataset

Click for debug datasets

Problem 4

Implement GreedyMotifSearch with Pseudocounts

We encountered *GreedyMotifSearch* in "Implement GreedyMotifSearch". In this problem, we will power it up with pseudocounts.

Implement *GreedyMotifSearch* with Pseudocounts

Given: Integers *k* and *t*, followed by a collection of strings *Dna*.

Return: A collection of strings *BestMotifs* resulting from running *GreedyMotifSearch*(*Dna*, *k*, *t*) with pseudocounts. If at any step you find more than one *Profile*-most probable *k*-mer in a given string, use the one occurring first.

Sample Dataset

```
3 5
GGCGTTCAGGCA
AAGAATCAGTCA
CAAGGAGTTCGC
CACGTCAATCAC
CAATAATATTCG
```

Sample Output

```
TTC
ATC
TTC
ATC
TTC
```

Extra Dataset

Click for an extra dataset

Click for debug datasets

Problem 5

Construct the De Bruijn Graph of a Collection of k-mers

Given an arbitrary collection of *k*-mers *Patterns* (where some *k*-mers may appear multiple times), we define *CompositionGraph*(*Patterns*) as a graph with |*Patterns*| isolated edges. Every edge is labeled by a *k*-mer from Patterns, and the starting and ending nodes of an edge are labeled by the prefix and suffix of the *k*-mer labeling

that edge. We then define the de Bruijn graph of *Patterns*, denoted *DeBruijn*(*Patterns*), by gluing identically labeled nodes in *CompositionGraph*(*Patterns*), which yields the following algorithm.

DeBruijn(*Patterns*)
  represent every *k*-mer in *Patterns* as an isolated edge between its prefix and suffix
  glue all nodes with identical labels, yielding the graph *DeBruijn*(*Patterns*)
return *DeBruijn*(*Patterns*)

De Bruijn Graph from *k*-mers Problem

Construct the de Bruijn graph from a collection of k-mers.

Given: A collection of *k*-mers *Patterns*.

Return: The de Bruijn graph *DeBruijn*(*Patterns*), in the form of an adjacency list.

Sample Dataset

```
GAGG
CAGG
GGGG
GGGA
CAGG
AGGG
GGAG
```

Sample Output

```
AGG -> GGG
CAG -> AGG,AGG
GAG -> AGG
GGA -> GAG
GGG -> GGA,GGG
```

Extra Dataset

Click for an extra dataset

Problem 6

Find an Eulerian Cycle in a Graph

A cycle that traverses each edge of a graph exactly once is called an **Eulerian cycle**, and we say that a graph containing such a cycle is **Eulerian**. The following algorithm constructs an Eulerian cycle in an arbitrary directed graph.

```
EULERIANCYCLE(Graph)
    form a cycle Cycle by randomly walking in Graph (don't visit the same edge twice!)
    while there are unexplored edges in Graph
        select a node newStart in Cycle with still unexplored edges
        form Cycle' by traversing Cycle (starting at newStart) and then randomly walking
        Cycle ← Cycle'
    return Cycle
```

Eulerian Cycle Problem

Find an Eulerian cycle in a graph.

Given: An Eulerian directed graph, in the form of an adjacency list.

Return: An Eulerian cycle in this graph.

Sample Dataset

```
0 -> 3
1 -> 0
2 -> 1,6
3 -> 2
4 -> 2
5 -> 4
6 -> 5,8
7 -> 9
8 -> 7
9 -> 6
```

Sample Output

```
6->8->7->9->6->5->4->2->1->0->3->2->6
```

Extra Dataset

Click for an extra dataset

Problem 7

Generate the Theoretical Spectrum of a Linear Peptide

Given an amino acid string *Peptide*, we will begin by assuming that it represents a *linear* peptide. Our approach to generating its theoretical spectrum is based on the assumption that the mass of any subpeptide is equal to the difference between the masses of two prefixes of *Peptide*. We can compute an array *PrefixMass* storing the masses of each prefix of *Peptide* in increasing order, e.g., for *Peptide* = NQEL, *PrefixMass* = (0, 114, 242, 371,

484). Then, the mass of the subpeptide of *Peptide* beginning at position *i* + 1 and ending at position *j* can be computed as *PrefixMass*(*j*) – *PrefixMass*(*i*). For example, when *Peptide* = NQEL,

$$Mass(QE) = PrefixMass(3) - PrefixMass(1) = 371 - 114 = 257.$$

G	A	S	P	V	T	C	I	L	N	D	K	Q	E	M	H	F	R	Y	W
57	71	87	97	99	101	103	113	113	114	115	128	128	129	131	137	147	156	163	186

Figure 1. The arrays AminoAcid (top row) and AminoAcidMass (bottom row) corresponding to the integer mass table.

The pseudocode shown on the next step implements this idea. It also represents the alphabet of 20 amino acids and their integer masses as a pair of 20-element arrays *AminoAcid* and *AminoAcidMass*, corresponding to the top and bottom rows of the following integer mass table, respectively.

Figure 1

```
LinearSpectrum(Peptide, AminoAcid, AminoAcidMass)
    PrefixMass(0) ← 0
    for i ← 1 to |Peptide|
        for j ← 1 to 20
            if AminoAcid(j) =  i-th amino acid in Peptide
                PrefixMass(i) ← PrefixMass(i − 1) + AminoAcidMass(j)
    LinearSpectrum ← a list consisting of the single integer 0
    for i ← 0 to |Peptide| − 1
        for j ← i + 1 to |Peptide|
            add PrefixMass(j) − PrefixMass(i) to LinearSpectrum
    return sorted list LinearSpectrum
```

Linear Spectrum Problem

Generate the ideal linear spectrum of a peptide.

Given: An amino acid string *Peptide*.

Return: The linear spectrum of *Peptide*.

Sample Dataset

```
NQEL
```

Sample Output

```
0 113 114 128 129 242 242 257 370 371 484
```

Extra Dataset

Click for an extra dataset

Problem 8

Find a Highest-Scoring Alignment of Two Strings

Global Alignment Problem

Find the highest-scoring alignment between two strings using a scoring matrix.

Given: Two amino acid strings.

Return: The maximum alignment score of these strings followed by an alignment achieving this maximum score. Use the BLOSUM62 scoring matrix and indel penalty σ = 5. (If multiple alignments achieving the maximum score exist, you may return any one.)

Sample Dataset

```
PLEASANTLY
MEANLY
```

Sample Output

```
8
PLEASANTLY
-MEA--N-LY
```

Extra Dataset

Click for an extra dataset

Problem 9

Compute the 2-Break Distance Between a Pair of Genomes

2-Break Distance Problem

Find the 2-break distance between two genomes.

Given: Two genomes with circular chromosomes on the same set of synteny blocks.

Return: The 2-break distance between these two genomes.

Sample Dataset

```
(+1 +2 +3 +4 +5 +6)
(+1 -3 -6 -5)(+2 -4)
```

Sample Output

```
3
```

Extra Dataset

Click for an extra dataset

Problem 10

Find a Shortest Transformation of One Genome into Another by 2-Breaks

2-Break Sorting Problem

Find a shortest transformation of one genome into another by 2-breaks.

Given: Two genomes with circular chromosomes on the same set of synteny blocks.

Return: The sequence of genomes resulting from applying a shortest sequence of 2-breaks transforming one genome into the other.

Sample Dataset

```
(+1 -2 -3 +4)
(+1 +2 -4 -3)
```

Sample Output

```
(+1 -2 -3 +4)
(+1 -2 -3)(+4)
(+1 -2 -4 -3)
(+1 +2 -4 -3)
```

Extra Dataset

Click for an extra dataset

Problem 11

Compute Limb Lengths in a Tree

Limb Length Problem

Find the limb length for a leaf in a tree.

Given: An integer *n*, followed by an integer *j* between 0 and *n* - 1, followed by a space-separated additive distance matrix *D* (whose elements are integers).

Return: The limb length of the leaf in *Tree*(*D*) corresponding to row *j* of this distance matrix (use 0-based indexing).

Sample Dataset

```
4
1
0   13  21  22
13  0   12  13
21  12  0   13
22  13  13  0
```

Sample Output

```
2
```

Extra Dataset

Click for an extra dataset

Problem 12

Implement AdditivePhylogeny

The following recursive algorithm, called **AdditivePhylogeny**, finds the simple tree fitting an *n* x *n* additive distance matrix *D*. We assume that you have already implemented a program **Limb**(*D*, *j*) that computes *LimbLength*(*j*) for a leaf *j* based on the distance matrix *D*. Rather than selecting an arbitrary leaf *j* from *Tree*(*D*) for trimming, **AdditivePhylogeny** selects leaf *n* (corresponding to the last row and column of *D*).

```
AdditivePhylogeny(D, n)
    if n = 2
        return the tree consisting of a single edge of length D1,2
    limbLength ← Limb(D, n)
    for j ← 1 to n - 1
        Dj,n ← Dj,n - limbLength
        Dn,j ← Dj,n
(i,n,k) ← three leaves such that Di,k = Di,n + Dn,k
x ← Di,n
remove row n and column n from D
T ← AdditivePhylogeny(D, n - 1)
v ← the (potentially new) node in T at distance x from i on the path between i and k
add leaf n back to T by creating a limb (v, n) of length limbLength
return T
```

Additive Phylogeny Problem

Construct the simple tree fitting an additive matrix.

Given: *n* and a tab-delimited *n* x *n* additive matrix.

Return: A weighted adjacency list for the simple tree fitting this matrix.

Note on formatting: The adjacency list must have consecutive integer node labels starting from 0. The *n* leaves must be labeled *0, 1, ..., n-1* in order of their appearance in the distance matrix. Labels for internal nodes may be labeled in any order but must start from *n* and increase consecutively.

Sample Dataset

```
4
0   13  21  22
13  0   12  13
21  12  0   13
22  13  13  0
```

Sample Output

```
0->4:11
1->4:2
2->5:6
3->5:7
4->0:11
4->1:2
4->5:4
5->4:4
5->3:7
5->2:6
```

Extra Dataset

Click for an extra dataset

Problem 13

Implement the Neighbor Joining Algorithm

The pseudocode below summarizes the neighbor-joining algorithm.

NeighborJoining(*D*,*n*)
  if *n* = 2
    T ← tree consisting of a single edge of length $D_{1,2}$
    return T
  D' ← neighbor-joining matrix constructed from the distance matrix *D*
find elements *i* and *j* such that $D'_{i,j}$ is a minimum non-diagonal element of *D*'
Δ ← ($TotalDistance_D(i)$ - $TotalDistance_D(j)$) /(*n* - 2)
$limbLength_i$ ← (1/2)($D_{i,j}$ + Δ)
$limbLength_j$ ← (1/2)($D_{i,j}$ - Δ)
add a new row/column *m* to *D* so that $D_{k,m} = D_{m,k}$ = (1/2)($D_{k,i}$ + $D_{k,j}$ - $D_{i,j}$) for any *k*
remove rows *i* and *j* from *D*
remove columns *i* and *j* from *D*
T ← *NeighborJoining*(*D*, *n* - 1)
add two new limbs (connecting node *m* with leaves *i* and *j*) to the tree *T*
assign length $limbLength_i$ to *Limb*(*i*)
assign length $limbLength_j$ to *Limb*(*j*)
return *T*

Neighbor Joining Problem

Construct the tree resulting from applying the neighbor-joining algorithm to a distance matrix.

Given: An integer *n*, followed by a space-separated *n* x *n* distance matrix.

Return: An adjacency list for the tree resulting from applying the neighbor-joining algorithm. Edge-weights should be accurate to two decimal places (they are provided to three decimal places in the sample output below).

Note on formatting: The adjacency list must have consecutive integer node labels starting from 0. The *n* leaves must be labeled *0, 1, ..., n-1* in order of their appearance in the distance matrix. Labels for internal nodes may be labeled in any order but must start from *n* and increase consecutively.

Sample Dataset

```
4
0   23  27  20
23  0   30  28
27  30  0   30
20  28  30  0
```

Sample Output

```
0->4:8.000
1->5:13.500
2->5:16.500
3->4:12.000
4->5:2.000
4->0:8.000
4->3:12.000
5->1:13.500
5->2:16.500
5->4:2.000
```

Extra Dataset

Click for an extra dataset

Problem 14

Implement the Lloyd Algorithm for k-Means Clustering

The **Lloyd algorithm** is one of the most popular clustering heuristics for the *k*-Means Clustering Problem. It first chooses *k* arbitrary points *Centers* from *Data* as centers and then iteratively performs the following two steps:

- **Centers to Clusters:** After centers have been selected, assign each data point to the cluster corresponding to its nearest center; ties are broken arbitrarily.
- **Clusters to Centers:** After data points have been assigned to clusters, assign each cluster's center of gravity to be the cluster's new center.

We say that the Lloyd algorithm has **converged** if the centers (and therefore their clusters) stop changing between iterations.

Implement the Lloyd algorithm

Given: Integers *k* and *m* followed by a set of points *Data* in *m*-dimensional space.

Return: A set *Centers* consisting of *k* points (centers) resulting from applying the Lloyd algorithm to *Data* and *Centers*, where the first *k* points from *Data* are selected as the first *k* centers.

Sample Dataset

```
2 2
1.3 1.1
1.3 0.2
0.6 2.8
3.0 3.2
1.2 0.7
1.4 1.6
1.2 1.0
```

```
1.2 1.1
0.6 1.5
1.8 2.6
1.2 1.3
1.2 1.0
0.0 1.9
```

Sample Output

```
1.800 2.867
1.060 1.140
```

Extra Dataset

Click for an extra dataset

Problem 15

Construct the Burrows-Wheeler Transform of a String

Our goal is to further improve on the memory requirements of the suffix array introduced in "Construct the Suffix Array of a String" for multiple pattern matching.

Given a string *Text*, form all possible **cyclic rotations** of *Text*; a cyclic rotation is defined by chopping off a suffix from the end of *Text* and appending this suffix to the beginning of *Text*. Next — similarly to suffix arrays — order all the cyclic rotations of *Text* lexicographically to form a |*Text*| × |*Text*| matrix of symbols that we call the **Burrows-Wheeler matrix** and denote by *M*(*Text*).

Note that the first column of *M*(*Text*) contains the symbols of *Text* ordered lexicographically. In turn, the second column of *M*(*Text*) contains the second symbols of all cyclic rotations of *Text*, and so it too represents a (different) rearrangement of symbols from *Text*. The same reasoning applies to show that any column of *M*(*Text*) is some rearrangement of the symbols of *Text*. We are interested in the last column of *M*(*Text*), called the **Burrows-Wheeler transform** of *Text*, or *BWT*(*Text*).

Burrows-Wheeler Transform Construction Problem

Construct the Burrows-Wheeler transform of a string.

Given: A string *Text*.

Return: *BWT*(*Text*).

Sample Dataset

```
GCGTGCCTGGTCA$
```

Sample Output

```
ACTGGCT$TGCGGC
```

Extra Dataset

Click for an extra dataset

> **Note**
>
> Although it is possible to construct the Burrows-Wheeler transform in O(|*Text*|) time and space, we do not expect you to implement such a fast algorithm. In other words, it is perfectly fine to produce BWT(*Text*) by first producing the complete Burrows-Wheeler matrix M(*Text*).

Problem 16

Implement BWMatching

We are now ready to describe **BWMatching**, an algorithm that counts the total number of matches of *Pattern* in *Text*, where the only information that we are given is *FirstColumn* and *LastColumn* = *BWT*(*Text*) in addition to the Last-to-First mapping (from "Generate the Last-to-First Mapping of a String"). The pointers *top* and *bottom* are updated by the green lines in the following pseudocode.

BWMatching(*FirstColumn*, *LastColumn*, *Pattern*, *LastToFirst*)

```
top ← 0
bottom ← |LastColumn| – 1
while top ≤ bottom
  if Pattern is nonempty
    symbol ← last letter in Pattern
    remove last letter from Pattern
    if positions from top to bottom in LastColumn contain an occurrence of symbol
      topIndex ← first position of symbol among positions from top to bottom in LastColumn
      bottomIndex ← last position of symbol among positions from top to bottom in LastColumn
      top ← LastToFirst(topIndex)
      bottom ← LastToFirst(bottomIndex)
    else
      return 0
  else
    return bottom – top + 1
```

Implement *BWMatching*

Given: A string *BWT*(*Text*), followed by a collection of strings *Patterns*.

Return: A list of integers, where the *i*-th integer corresponds to the number of substring matches of the *i*-th member of *Patterns* in *Text*.

Sample Dataset

```
TCCTCTATGAGATCCTATTCTATGAAACCTTCA$GACCAAAATTCTCCGGC
CCT CAC GAG CAG ATC
```

Sample Output

```
2 1 1 0 1
```

Extra Dataset

Click for an extra dataset