**Lab/Homework Assignment #2**

**PART I: Local and global alignment**

**Exercise 1** *(20 points)*

Consider two following strings:

AATCCGGTACA

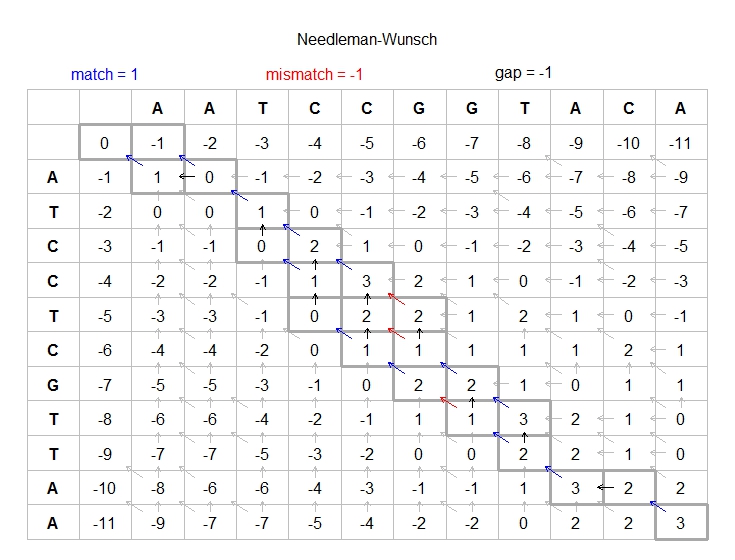
ATCCTCGTTAA

* 1. What is the total number of global alignments for these two strings?

|  |  |  |  |
| --- | --- | --- | --- |
| n | (n+r C n) | n C 3 | (n+r C n) \* n C 3 |
| 0 | 1 | 1 | 1 |
| 1 | 12 | 11 | 132 |
| 2 | 78 | 55 | 4290 |
| 3 | 364 | 165 | 60060 |
| 4 | 1365 | 330 | 450450 |
| 5 | 4368 | 462 | 2018016 |
| 6 | 12376 | 462 | 5717712 |
| 7 | 31824 | 330 | 10501920 |
| 8 | 75582 | 165 | 12471030 |
| 9 | 167960 | 55 | 9237800 |
| 10 | 352716 | 11 | 3879876 |
| 11 | 705432 | 1 | 705432 |
|  |  | SUM = | 45046719 |

**1.2.** Compute (i.e. layout and fill in) the dynamic programming alignment table that computes the value of the optimal alignment of these two strings; use the following objective function:

**maximize {#matches - #mismatches - #spaces}**



* 1. Now find at least one optimal trace-back path in the table and write out the associated optimal alignment. Use one of the tools (*see additional comments in the end*) to check that your alignment is in fact optimal. Turn in a print-out of the output for your alignment (from the program of your choice).

Length of sequence 1: 11 ->2014\_10\_23

Length of sequence 2: 11 ->2014\_10\_23

Aligned length: 11

Identical length: 5

Sequence identity: 0.455 (= 5/ 11)

AATCCGGTACA

: : :: :

ATCCTCGTTAA

12345678901

* 1. Find the local alignment with the highest value using the **Smith-Waterman algorithm** that we learned in class.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | A | A | T | C | C | G | G | T | A | C | A |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| T | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| C | 0 | 0 | 0 | 0 | 3 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| C | 0 | 0 | 0 | 0 | 1 | 4 | 0 | 0 | 0 | 0 | 1 | 0 |
| T | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| C | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| G | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 |
| T | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| T | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 1 |
| A | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

EMBOSS\_001 2 ATCCG-GTACA 11

||||. ||..|

EMBOSS\_001 1 ATCCTCGTTAA 11

How many local alignments of non-zero value do we have? = 4

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Exercise 2** *(10 points)*

Download Perl program needleman.pl. Be sure you have it running, and try to understand how it works. This is a Perl version of the **Needleman-Wunsch alignment algorithm** that we studied in class, using the dynamic programming recurrence relations, instead of an alignment graph. Even though you have not learned all the Perl constructs used in this program, you should be able to understand what the different parts of the program does, based on you prior exposure to some programming language.

Modify the program so that it asks the user for a match value *V*, a mismatch cost *Cm*, and an indel cost *Im*. It should read the inputs from the keyboard and assigns them to variables. Modify the program so that it finds the maximum value of any possible alignments of the two input strings, where the objective function is **maximize{*V* \* (#matches) - *Cm* \* (#mismatches) - *Im* \* (#indels)}**

open (OUT, '> outer'); #Open a file called 'outer' for outputing.

print "Input string 1 \n";

$line = <>;

chomp $line;

@string1 = split(//, $line); #split up the line into individual characters

#and place the characters into a list, whose

#first index is 0 (here Perl has inherited

#one of the worst features of C) See Johnson

# 4.3 for an introduction to lists. Look up

# split in the index and read it.

print "Input string 2 \n";

$line = <>;

chomp $line;

@string2 = split(//, $line);

print "Please enter the match value (V):\n";

$match = <>;

chomp $match;

print "Please enter the mismatch cost (Cm):\n";

$mismatch = <>;

chomp $mismatch;

print "Please enter the indel cost (Im):\n";

$indel = <>;

chomp $indel;

$n = @string1; #assigning a list to a scalar just assigns the

#number of elements in the list to the scalar.

$m = @string2;

print "The lengths of the two strings are $n, $m \n"; # Just to make sure this works.

$V[0][0] = 0; # Assign the 0,0 entry of the V matrix

for ($i = 1; $i <= $n; $i++) { # Assign the column 0 values and print

# String 1 See section 5.2 of Johnson

# for loops

$V[$i][0] = -1\*$indel \* $i;

print OUT "$string1[$i-1]"; # Note the -1 here because array indexes start at 0 (ug!)

}

print OUT "\n\n";

for ($j = 1; $j <= $m; $j++) { # Assign the row 0 values and print String 2

$V[0][$j] = -1\*$indel \* $j;

print OUT "$string2[$j-1]";

}

for ($i = 1; $i <= $n; $i++) { # follow the recurrences to fill in the V matrix.

for ($j = 1; $j <= $m; $j++) {

print OUT "$string1[$i-1], $string2[$j-1]\n"; # This is here for debugging purposes.

if ($string1[$i-1] eq $string2[$j-1]) {

$t = $match ; }

else {

$t = -$mismatch ;

}

$max = $V[$i-1][$j-1] + $t;

print OUT "For $i, $j, t is $t \n"; # Another debugging line.

if ($max < $V[$i][$j-1] -$indel) {

$max = $V[$i][$j-1] -$indel;

}

if ($V[$i-1][$j] -$indel > $max) {

$max = $V[$i-1][$j] -$indel;

}

$V[$i][$j] = $max;

print OUT "V[$i][$j] has value $V[$i][$j]\n";

}

}

print OUT "\n The similarity value of the two strings is $V[$n][$m]\n";

close(OUT);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Exercise 3** *(10 points)*

Extend the needleman.pl program of exercise 2 to incorporate gaps. Note that you will have to modify the way spaces are handled.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PART II: Perl and Regular Expressions**

**Exercise 4** *(25 points)*

Please read Introduction\_to\_Perl\_2.pdf *by D. Gusfield and K. Stevens*. Do exercises 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.14 (Please number them as 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.14 in your report ). Read about lists and arrays in *Johnson* or elsewhere.

# 4.1

print "Input a string: \n";

$string1 = <>;

if ($string0 =~ m/TA+GG/) { print "Match with + \n"; }

if ($string0 =~ m/TA\*GG/) { print "Match with \* \n"; }

print "Input a string: \n";

$string2 = <>;

if ($string1 =~ m/TA+GG/) { print "Match with + \n"; }

if ($string1 =~ m/TA\*GG/) { print "Match with \* \n"; }

#Input a string:

#ATATAAAGGGA

#Match with +

#Match with \*

#Input a string:

#ATATGGGA

#Match with \*

#Explanation:

#ATATAAAGGGA has matches with + and \* because it contains the substring: #TAAAGG

#ATATGGGA has only a match with \* because TGG is a substring with no A after T

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 4.2

$string2 = 'TATATAGG';

if ($string2 =~ m/(TA)+GG/) { print "Match!\n"; } #matches

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 4.3

#Liver exterases: H(T|V|I)EL

#ER: (H(T|V|I)EL)|((K|R|H|Q|A)(D|E|N|Q)EL)

print "Input ER:\n";

$num = 0;

$string3 = <>;

if ($string3 =~ m/(H(T|V|I)EL)|((K|R|H|Q|A)(D|E|N|Q)EL)/){

print "YES!\n";

$num++;

}

print "$num match(es)\n";

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 4.4

#serine: TC[TCAG]|AG[TC]

#arginine or serine: CG[ATCG]|TC[TCAG]|AG[TCAG]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 4.5

$varstring = 'helloCACACACACAhelloCACACACACACACACA';

$varstring =~ s/(CA){5}(CA)\*/X/g;

print "$varstring\n";

#helloXhelloX

#It substitutes when there is at least 5 consecutive instances of CA with an X.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 4.6

No, because there is no OR operator in between the two repetition modifiers which means it could have 1 capital letter followed by 6 digits or 2 capital letters followed by 5 digits; which is not standard Genbank accession numbers.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 4.7

print "Enter potential BGl1 sequence:\n";

$varstring = <>;

if ($varstring =~ m/GCC[ATCG]{5}GCC/)

{print "YES, BGl1";}

#Enter potential BGl1 sequence:

#GCCATCGAGCC

#YES, BGl1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# 4.14

print "Enter potential RNA polymerase II promoter sequence:\n";

$varstring = <>;

if ($varstring =~ m/TATA[AT]A[AT].{17}[CT][CT]CA[AG][AG]/)

{print "YES, RNA polymerase II promoter sequence";}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Exercise 5** *(15 points)*

In the next exercises you will work with short regular expression programs. Run the first three programs (ex5\_program1.pl, ex5\_program2.pl, and ex5\_program3.pl) and understand how the regular expression works in them. In the starting comments of each program, you are asked to run the program in certain ways. Be sure you do those, but please **DON'T turn in scripts of those executions**.

**5.1.** Program accession.pl is very similar to ex5\_program3.pl, but has some extensions. Modify accession.pl so that it only prints the accession numbers it finds, and prints each one on a separate line. That is, it prints a list of accession numbers it finds in the input file, but does not print anything else. Use ex5\_testfile.txt as the test input file.

print "type the input file name please\n";

open (INFO, <>); # open the input file and assign the handle INFO

print "type the output file name please\n";

$myoutfile = <>; # Read in the file name for the output.

open (OUTFO, "> $myoutfile"); # open the output file and assign the handle

#OUTFO

while (<INFO>) {

foreach $word (m/(\w+)/g) {

if ($word =~ m/[A-Z]{1,2}\d{3,7}/){

print OUTFO "$word\n";

}

}

}

**5.2.** The meaning of [ ,.;:?] in the regular expression is that the digits of an accession number must be followed by any ONE of the six characters listed between the brackets [...]. So this is an OR of the six characters. Now in the program ex5\_program2.pl (it looks for a DNA string in an input line) we used “|” to indicate an OR. That regular expression had (A|T|C|G|a|t|c|g). Replace that with [ATCGatcg] to see if the program works. Script the result. Use ex5\_testfile.txt as the test input file.

$file = 'ex5\_testfile.txt';

open (INFO, $file);

$line = <INFO>;

if ($line =~ m/(^| |=|>)([ATCGatcg]{3,})/ ) {

print "The input line does contain what looks like a dna sequence of length at least three, and it is $2 \n";

}

else {

print "The input line does not contain a dna sequence\n";

}

print "The input line is \n$line\n";

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PART III: Longest Common Subsequence**

**6.1.** By setting *V* to 1, and *Cm* and *Im* to zero, the program needleman.pl (*see Exercise 2*) will produce the length of the **longest common subsequence** (LCS) between the two sequences. That is the alignment that simply maximizes the number of matches that can be obtained, without regard for how many space and mismatches are involved. The LCS between two strings is sometimes taken as a measure of the similarity of two strings. By letting your figures make up "random" strings of length 20 say, and computing the LCS of those two strings, try to find in this way the expected length of the LCS of two random strings.

open (OUT, '> outer6'); #Open a file called 'outer' for outputing.

my @chars = ("A".."Z");

my $string1;

my $string2;

print "String 1: \n";

$random1 .= $chars[rand @chars] for 1..20;

print "$random1\n";

print "String 2: \n";

$random2 .= $chars[rand @chars] for 1..20;

print "$random2\n";

@string1 = split(//, $random1);

@string2 = split(//, $random2);

$n = @string1; #assigning a list to a scalar just assigns the

#number of elements in the list to the scalar.

$m = @string2;

print "The lengths of the two strings are $n, $m \n"; # Just to make sure this works.

$V[0][0] = 0; # Assign the 0,0 entry of the V matrix

for ($i = 1; $i <= $n; $i++) { # Assign the column 0 values and print

# String 1 See section 5.2 of Johnson

# for loops

$V[$i][0] = -$i;

print OUT "$string1[$i-1]"; # Note the -1 here because array indexes start at 0 (ug!)

}

print OUT "\n\n";

for ($j = 1; $j <= $m; $j++) { # Assign the row 0 values and print String 2

$V[0][$j] = -$j;

print OUT "$string2[$j-1]";

}

for ($i = 1; $i <= $n; $i++) { # follow the recurrences to fill in the V matrix.

for ($j = 1; $j <= $m; $j++) {

#print OUT "$string1[$i-1], $string2[$j-1]\n"; # This is here for debugging purposes.

if ($string1[$i-1] eq $string2[$j-1]) {

$t = 1; }

else {

$t = 0 ;

}

$max = $V[$i-1][$j-1] + $t;

#print OUT "For $i, $j, t is $t \n"; # Another debugging line.

if ($max < $V[$i][$j-1] -0) {

$max = $V[$i][$j-1] -0;

}

if ($V[$i-1][$j] -0 > $max) {

$max = $V[$i-1][$j] -0;

}

$V[$i][$j] = $max;

print OUT "V[$i][$j] has value $V[$i][$j]\n";

}

}

print OUT "\n The expected LCS value of the two strings is $V[$n][$m]\n";

close(OUT);

**6.2.** In this exercise, you will determine as best you can what is the correct expected length of the LCS is. Randomdna.pl is a Perl program to produce random DNA strings. Get this program running and make sure you understand how it works. Then modify it so that it asks the user how many random strings to produce, and then it generates that many strings and writes them into a file. The main modification is to put a loop around part of the existing program.

Now generate gobs (*that is a technical term for a number that is at least 10*) of random strings of length 10 each, and using your modified Needleman-Wunsch from Exercise 6.1, compute the length of the LCS between the first of your strings and each of the other ones. Compute the average LCS length obtained. Then repeat with strings of length 20, then 50, then 100, 200. What do you observe about the average LCS length? If you are comfortable enough with Perl and programming at this point, you may want to put everything together into a single program that generates the random strings, computes the LCS lengths, computes and reports the averages. This may save you a lot of effort compared to running programs over and over again. In fact, if you do have a single program, then for each string length you should generate at least 100 random strings.

#generate a random DNA strings and print it to file specified by the user.

print "What is the name of the output file?\n";

$rstrings = <STDIN>;

open (OUT, ">$rstrings");

print "How many nucleotides for the string?\n";

$n = <>;

print "How many random strings to produce?\n";

$count = <>;

for ($x=1; $x <= $count; $x++){

$numstring = ''; # start with the empty string;

$i = 0;

while ($i < $n) {

$numstring = int(rand(4)) . $numstring; # generate a new random integer

# between 0 and 3, and concatenate

# it with the existing $numstring,

# assigning the result to $numstring.

$i++; # increase the value of $i by one.

}

$dnastring = $numstring; # unneeded, but good for clarity.

$dnastring =~ tr/0123/actg/; # translate the numbers to DNA characters.

print OUT "$dnastring\n";

}

close (OUT);

my $file = "$rstrings";

open(FH, "< $file") or die;

my @line;

while (<FH>){

chomp;

push (@line, $\_)

}

close FH or die;

print join ("\n", @line);

for ($a=0; $a < $count; $a++){

my @strings = split(//, @line[a]);

for each my $str (@strings){

print "$str\n";

}

#$firststring = @strings[a];

#print "@first";

print "\nLength = $firststring";

}