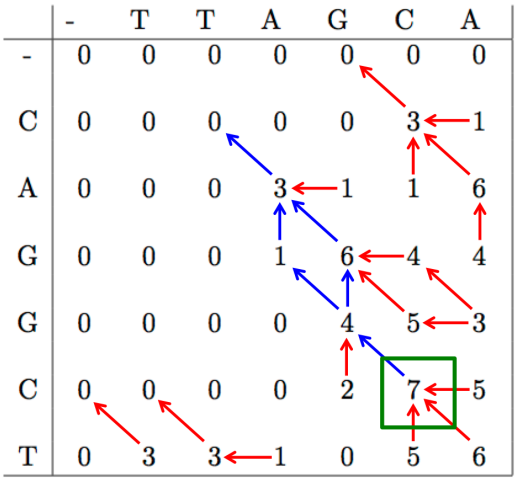
1. Fill in the dynamic programming table and find all maximal *local* alignments for the strings $X = CAGGCT$ and $Y = TTAGCA$, using a scoring function $\sigma(x, y) = 3$ if $x = y$, $\sigma(x, y) = -1$ if $x \neq y$, and $\sigma(x, -) = \sigma(-, y) = -2$ for a gap.

   Solution:



	-	T	T	A	G	C	A
-	0	0	0	0	0	0	0
C	0	0	0	0	0	3	1
A	0	0	0	3	1	1	6
G	0	0	0	1	6	4	4
G	0	0	0	0	4	5	3
C	0	0	0	0	2	7	5
T	0	3	3	1	0	5	6

   The optimal local alignment score is $V(5, 5) = 7$. Backtracing from this cell, there are two paths (terminating when we hit a 0) which correspond to two different alignments: $A - GC$ aligned to $AGGC$ and $AG - C$ aligned to $AGGC$.

2. [modified from Gusfield, Section 11.9, Problem 35] The sequence of a gene is made up of *introns* and *exons*. During transcription, the gene sequence is copied into RNA, but the introns are spliced out, leaving only the exons. Exome sequencing is becoming more and more common as a way of cheaply looking for variants in coding sequences. Describe a dynamic programming algorithm that would align an RNA transcript (just the exons) T to a reference genome G, in such a way that there are no gaps in the reference (potentially unrealistic) and no penalties for unaligned sequence at the beginning and end of the reference. Most introns start with the dinucleotide GT and end with AC. Modify your algorithm to enforce this constraint.

   Solution: (sketch) Considering the case without the GT start and AC end constraints, we want an alignment that does not penalize "free" (unaligned) ends in the reference G, but then after we start aligning the transcript T, we don't want any gaps in G. Create a DP table V where $V(i, j)$ represents the optimal alignment of $T[1...i]$ and $G[1...j]$. Let $|T| = m$ and $|G| = n$. Then the base case and recursion are

$$\begin{aligned} V(0,0) &= 0 \\ V(i,0) &= -\infty, \quad 1 \leq i \leq m \\ V(0,j) &= 0, \quad 1 \leq j \leq n \\ V(i,j) &= \max \begin{cases} V(i-1, j-1) + \sigma(t_i, g_j) \\ V(i, j-1) + \sigma(-, g_j) \end{cases} \end{aligned}$$

where σ reasonably penalizes gaps and mismatches. To find the final cell of the best alignment, look at the bottom row of the DP table:

$$V(m, k^*) = \arg\max_k \ V(m, k),$$

then traceback to find the optimal alignment. There are a few different ways to modify the algorithm to ensure that introns start with GT and end with AC. One way is to create a boolean for whether we are in an intron or an exon. If we are in an intron, we should not penalize unaligned reference sequence, whereas if we are in an exon, we should be mapping [most] characters of the transcript. Each time we see a GT, we have the option of starting an intron, and each time we see an AC we have the option of ending the intron and starting an exon.

3. [BWT revisited] Fill in the table below for $\pi^{\text{sorted}}(S)$, for $S = ccabcbcabc$. What are $M[a], M[b], M[c]$? Describe how to (exactly) match the pattern $P = abc$ to S using the recursions from lecture. Why might it be difficult to extend BWT to handle indels and mismatches?

Solution:

i	F	L = BWT	occ(a,i)	occ(b,i)	occ(c,i)
1	$ c c a b c b c a b c		0	0	1
2	a b c $ c c a b c b c		0	0	2
3	a b c b c a b c $ c c		0	0	3
4	b c $ c c a b c b c a		1	0	3
5	b c a b c $ c c a b c		1	0	4
6	b c b c a b c $ c c a		2	0	4
7	c $ c c a b c b c a b		2	1	4
8	c a b c $ c c a b c b		2	2	4
9	c a b c b c a b c $ c		2	2	5
10	c b c a b c $ c c a b		2	3	5
11	c c a b c b c a b c $		2	3	5

$M[a] = 2$, $M[b] = 4$, $M[c] = 7$. To match P to S, we begin at the end of P and work backwards:

$$\begin{aligned} sp(c) &= M[c] = 7 \\ ep(c) &= M[c] + occ(c, |S| + 1) = 11 \end{aligned}$$

$$\begin{aligned} sp(bc) &= M[b] + occ(b, sp(c) - 1) = 4 + 0 = 4 \\ ep(bc) &= M[b] + occ(b, ep(c)) - 1 = 4 + 3 - 1 = 6 \end{aligned}$$

$$\begin{aligned} sp(abc) &= M[a] + occ(a, sp(bc) - 1) = 2 + 0 = 2 \\ ep(abc) &= M[a] + occ(a, ep(bc)) - 1 = 2 + 2 - 1 = 3 \end{aligned}$$

Thus there are two occurrences of P in S, starting at positions $A_S[2] = 8$ and $A_S[3] = 3$, where A_S is the suffix array of S.

To handle mismatches or indels using this type of method, the number of options we would have to keep track of would grow very quickly. As we worked backward through the pattern P, the recursive step would have to be performed for different character (or gap) options. This would quickly become too expensive, which is why dynamic programming provides a better alternative in many cases.