*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology November 22, 2016
Professors Erik Demaine, Debayan Gupta, and Ronitt Rubinfeld Problem Set 5

# Problem Set 5

**All parts are due on December 8, 2016 at 11:59PM.** Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

---

Note that due to MIT policy on end of semester assignments, **you may use at most 1 slack day on this problem set**, even if you have more than one remaining.

# Part A

**Problem 5-1.** [35 points] **Fantastic Beasts**

You are aiding Newt Scamander, the Wizarding World's pre-eminent magizoologist, in his quest to discover and understand magical creatures. You have managed to sequence a gene, which is a string of a 4 letter alphabet $\{A, T, G, C\}$, from a Hippogriff, and you want to answer the age-old question of whether a Hippogriff is more eagle or more horse by comparing this gene to the existing and well-understood horse and eagle genes.

You will measure how "close" two gene sequences are by computing the optimal "score" of an "alignment" of the two genes. An ***alignment*** of two strings $a$ and $b$ is a pair of alignment strings $a^+$ and $b^+$ which contain gap characters '_' such that

1. $|a^+| = |b^+|$;
2. for each $i$ with $0 \leq i \leq |a^+| - 1$, at most one of $a^+[i]$ and $b^+[i]$ equals '_';
3. removing all '_' symbols from $a^+$ yields $a$; and
4. removing all '_' symbols from $b^+$ yields $b$.

For example, if we have two genes $a = ATAGCATGC$ and $b = ACTAGCTGC$, then one possible alignment of these genes is as follows:

$$a^+ = ATAG\_CAT\_GC,$$
$$b^+ = A\_CTAG\_CTGC.$$

The ***score*** of a particular alignment $\{a^+, b^+\}$ of two genes is the sum of the "character-match scores" for each index of the alignment:

$$\texttt{score}(a^+, b^+) = \sum_{i=0}^{|a^+|-1} w(a^+[i], b^+[i]),$$

where the ***character-match scores*** $w(i, j)$ are given by the following table:

| | $A$ | $T$ | $G$ | $C$ | '_' |
|---|---|---|---|---|---|
| $A$ | 0 | 1 | 2 | 3 | 4 |
| $T$ | 1 | 0 | 3 | 2 | 4 |
| $G$ | 2 | 3 | 0 | 1 | 4 |
| $C$ | 3 | 2 | 1 | 0 | 4 |
| '_' | 4 | 4 | 4 | 4 | n/a |

Different alignments of the same two genes can produce different scores. We define an ***optimal alignment*** of two genes to be an alignment of minimum possible score. For example, the example alignment above has a score of 25, while the optimal alignment of the two genes $ATAGCATGC$ and $ACTAGCTGC$ actually has score 8:

$$a^+ = A\_TAGCATGC,$$
$$b^+ = ACTAGC\_TGC.$$

**(a)** [2 points] Design a dynamic programming algorithm that, given two genes $a$ and $b$, each of length $\Theta(n)$, finds the score of their optimal alignment in $O(n^2)$ time and $O(n^2)$ space.

**Solution:** Let the notation $S(a_i, b_j)$ denote the optimal matching score between the first $i$ nucleotides of $a$, and the first $j$ nucleotides of $b$. The base case of $S(0, 0)$ is trivially 0, since there's nothing to match. View the problem as a matrix with $|a|$ rows and $|b|$ columns. Then the location in the matrix at row $i$ and column $j$ has the value$S(a_i, b_j)$. To solve for the value of a given sub-problem, $S$, you need to find the value in the corresponding matrix location. The following recursion is used: $S(a_i, b_j) = \min(S(a_{i-1}, b_{j-1}) + w(a[i], b[j]), S(a_i, b_{j-1}) + w(\_, b[j]), S(a_{i-1}, b_j) + w(a[i],\_)$ , which corresponds to the location on top, to the left, and to the top-left in the matrix. If there are no such locations, exclude the corresponding argument in the $\min$ portion of the recurrence, since that sub-problem is not a thing. The upper-leftmost sub-problem is $0$ since we had earlier noted that the base case of $S(0, 0) = 0$. This formulation of the recurrence will henceforth be referred to as the Prefix-formulation, so named because you align the prefixes of both sequences and work your way to the right. Storing the entire matrix is $O(n^2)$.

**(b)** [3 points] Describe how to modify your algorithm from (a) to use only $O(n)$ space.

**Solution:** Note that the answer to each sub-problem depends only on the row above it, and the item immediately to it's left in the matrix. It suffices to only store the most recently completed row of sub-problems and the current row of sub-problems you are working to compute. This takes O(n) space. (A similar argument can be made for columns and diaonals that go from the bottom left to the top-right).

You have now decided that you would like to know not only which gene is more closely related, but also the actual optimal alignment of base pairs, and not just the scores.

**(c)** [5 points] Describe how to modify your algorithm from (a) to compute the optimal alignment in $O(n^2)$ time and $O(n^2)$ space.

**Solution:** We modify the Prefix algorithm in part a as follows: Previously, we were only storing the optimal score, the solution to $S(a_i, b_j)$ in the corresponding location in the matrix. Since we now want to return the optimal alignment, we want to return the optimal set of choices we made in addition to the score. If we achieved our current score by relying on a sub-problem directly above or directly to the left in the matrix, we have inserted a gap. Otherwise, we have matched two base pairs, $a[i-1]$ and $b[j-1]$. Thus we can derive what optimal alignment we chose, based on which sub-problems we relied on. So in addition to keeping the score in each matrix location, we also keep a pointer to the sub-problem we relied on to get there. Storing the entire matrix takes $O(n^2)$, and each sub-problem takes constant time to compute as before, yielding an $O(n^2)$ run-time as well.

**(d)** [15 points] Describe how to modify your algorithm from (b) to compute the optimal alignment in $O(n^2)$ time and $O(n)$ space.

*Hint:* To solve this problem, you will simultaneously need to work forwards (aligning prefixes of the strings) and backwards (aligning suffixes of the strings), until they meet on the main diagonal of the DP matrix. Then divide and conquer in two of the submatrices, to obtain the recurrence $T(p) = 2T(p/4) + O(p)$ where $p$ is the total number of entries in the matrix (i.e., the product of the lengths of $a$ and $b$).

**Solution:** From part b, we know that we can get the optimal score in linear space. We do this until we get to the middle column of the matrix, while only storing the most recent row and thus taking $O(n)$ space as before. We stop when we get an optimal answer for some prefix of $a$ alighing with $b[0 : midPoint]$ where $midPoint = |b|/2$. So store all the values in the $|b|/2$ column of the matrix also. Store all the parent pointers for the locations in column $|b|/2$ only.

Now we do the same thing in reverse, the suffix problem. Here we align the suffixes of $a$ and $b$ and build from right to left. The new recurrence is $Q(a_i, b_j) = \min(Q(a_{i+1}, b_{j+1}) + w(a[i], b[j]), Q(a_i, b_{j+1}) + w(\_, b[j]), Q(a_{i+1}, b_j) + w(a[i], \_)$, where $Q(a_i, b_j)$ represents the optimal alignment between the $a[i : end]$ and $b[j : end]$, the suffixes of both sequences. It's the same problem, just in the other direction. Like we did in part b, we can also only store the most recently completed row and the current row for this procedure, which we will call the Suffix Algorithm. This is O(n) space. Keep doing this until you have the optimal alignment for some suffix of a and $b[midpoint : end]$. So as before, store all the values for the $|b|/2$ column of the table. Store all the parent pointers for the locations in column $|b|/2$ only.

Add the values from the prefix algorithm to the correspoinding values from the suffix algorithm at column $|b|/2$ of the table. The minimum of these points is the optimal point.The optimal alignment goes through this location.Thanks to the parent pointers, we also know which location in the matrix is used to get to the minimum from both columns $|b/2+1$ and $|b|/2-1$. Let's say this minium location is in row $i$ and of course column $|b|/2$.

We have now created two sub-problems. In the first we need to find the optimal alignment between $a[0 : i]$ and $b[0 : |b|/2]$, and in the second, we need to find the optimal alignment between $a[i : end]$ and $b[|b|/2 : end]$. We use the steps described in the previous three paragraphs on each of these sub-problems, and continue recursing. When we're done, we have parent pointers along the optimal path, and that gives us the optimal alignment in linear space.

When reasoning about run-time, let's use the notation $T(A, B)$ to represent aligning sequences with length $A$ and length $B$. Then we get the recursion $T(A, B) = \theta(AB)+ T(i, B/2)+T(A-i, B/2)$ where $i \leq A$. In the $T(A, B)$ problem, we do $\theta(AB)$ work before recursing. In $T(A - i, B/2)$, we do $\theta(\frac{AB-Bi}{2})$ work, and in $T(i, B/2)$ we do $\theta(\frac{Bi}{2})$ work. Thus in the second layer we do a sum of $\theta(\frac{AB}{2})$ work, which is half the work we did in the first layer of the recursion. Thus the work we do at each successive layer halves. This leads to the total amounf of work done being $\theta(AB \times (1 + \frac{1}{2} + \frac{1}{4} + \cdots)) = \theta(AB)$ work. Thus our run-time is still $O(n^2)$.

[10 points] Unfortunately, you find that your solution to part (c) is taking too long to run on the DNA sequences (whose lengths are in the millions). You decide to modify your program to approximate the optimal alignment by restricting the number of gaps allowed in each alignment string $a^+$ and $b^+$ to $100$. Describe how to modify your algorithm from part (c) to satisfy this additional constraint. What is the running time of the resulting algorithm?

**Solution:** For each $i \in \{0, 1, \ldots, 100\}$, $j \in \{0, 1, \ldots, 100\}$, and $k \in \{0, 1, \ldots, n\}$ define a subproblem $T(i, j, k)$ to be the minimum score of an alignment of the first $k$ characters of the first sequence that uses $i$ gaps in the first sequence and $j$ gaps in the second. Note that this determines how many characters of the second sequence are used, and we don't care about anything that uses more than 100 gaps in either sequence. We can calculate each subproblem from 3 other subproblems, and there are $100^2n = O(n)$ subproblems, so the total time is $O(n)$, as desired.

**Problem 5-2.** [20 points] **Rye Elections**

Prof. Caufield has decided to run in the 2020 U.S. presidential race, and has appointed you as his campaign manager. He wants to campaign in the state of Massachusetts, which (thanks to rising oceans) has become a line of $n$ cities, $c_0, c_1, \ldots, c_{n-1}$. Prof. Caufield's campaign consists of running television advertisements in specific cities. Each city $c_i$ charges a price $p_i$ to broadcast an advertisement. Furthermore, the residents of Massachusetts love nothing more than to talk amongst each other and gossip, and so if Holden runs a television ad in city $c_i$, its $k$ neighboring cities on each side $(c_{i-k}, \ldots, c_{i-1}, c_i, c_{i+1}, \ldots, c_{i+k})$ all hear about Prof. Caufield's exciting policies.

Can you design an algorithm to choose which cities to advertise in, to reach all of the cities in Massachusetts, for the minimum total price?

**(a)** [5 points] Let's first consider the case where $k = 1$, and thus running an ad in a city reaches itself and both of its neighboring cities. Prof. Caufield comes up with the following algorithm: until all cities have been reached, select the cheapest unreached city and run an advertisement there. Does this algorithm minimize the cost of reaching all of the $n$ cities? Prove correctness or provide a counterexample.

**Solution:** Prof. Caufield's algorithm does not minimize the cost of reaching all cities. Consider the line of three cities $\{c_1, c_2, c_3, c_4\}$, with broadcasting prices $\{p_1 = \$10, p_2 = \$5, p_3 = \$6, p_4 = \$10$. Prof. Caufield's algorithm will choose to run an advertisement first in $c_2$, and then $c_4$, as it's the cheapest unreached city, when in fact the cheapest campaign would be to broadcast at cities $c_2$ and $c_3$.

**(b)** [10 points] For the $k = 1$ case, design an algorithm that computes the subset of cities in which Prof. Caufield should run advertisements that minimizes the total cost and still reaches every city, using $O(n)$ time.

**Solution:** We present a dynamic programming solution based on suffixes. First, let's define our subproblems to be $T(i, b)$, the minimum cost of reaching all cities from $c_i$ to $c_{n-1}$, inclusive, where $b$ is `True` if $c_i$ has already been reached, and `False` if not. Our base cases are then that $T(k, b) = 0$ for $k \geq n$ and $b =$ `True` or `False`, since there are no cities left to be reached. To solve the general subproblem, we'll guess whether or not to run an advertisement at $c_i$. This gives us the following recurrence:

$$T(i, b) = \begin{cases} \min(p_i + T(i+1, \texttt{True}), T(i+1, \texttt{False})) & \text{if } b \text{ is } \texttt{True} \\ \min(p_i + T(i+1, \texttt{True}), p_{i+1} + T(i+2, \texttt{True})) & \text{if } b \text{ is } \texttt{False} \end{cases}$$

There are a total of $2n$ subproblems, and so we will compute in order first the subproblems $T(n-1, b)$ for both $b$, then $T(n-2, b)$ for both $b$, and so on, storing each subproblem solution in a $2 \times n$ table and looking up the solutions when you need them. Solving each of these subproblems simply requires looking up two solutions in the table, addition, and taking a minimum of two elements, which can all be done in

$O(1)$, and so we can solve all subproblems in $2n \cdot O(1) = O(n)$. Finally, the solution to the minimum cost required to reach all cities is stored in $T(0, \texttt{False})$.

To generate the actual subset of cities, we can also maintain a $2 \times n$ table of parent pointers $\pi$, in which $\pi_{i,b}$ stores a boolean value of which decision we made for the $T(i, b)$ subproblem. We can then follow these parent pointers from $\pi_{0,\texttt{False}}$ all the way back to one of the base cases and reconstruct the actual subset of cities where advertisements should be run.

**(c)** [5 points] Describe how to modify your algorithm for the case of general $k$, while running in $O(nk)$ time.

**Solution:** We modify our subproblems to be $T(i, b)$, where $b$ is now an integer in the range $[1, k + 1]$. Each subproblem now represents the minimum cost of reaching all cities from $c_i$ to $c_{n-1}$, inclusive, where the largest $j < i$ such that an advertisement has already been run on $c_j$ is $c_{i-b}$. Here, $b = k + 1$ represents the special case where no advertisements were run in the $k$ cities before $c_i$. Just as before, our base cases are then $T(k, b) = 0$ for any $k \geq n$ and any $b$, since there are no cities left to be reached. Our recurrence is then

$$T(i,b) = \begin{cases} \min(p_i + T(i+1,1), T(i+1,b+1)) & \text{if } b \leq k \\ \min_{j \in [i,i+k]}(p_j + T(j+1,1)) & \text{if } b = k+1 \end{cases}$$

There are a total of $n(k+1) = O(nk)$ subproblems. However, all subproblems are not created equally, and we note that $nk$ of these $n(k + 1)$ subproblems (whenver $b \leq k$) can be solved in $O(1)$, while the remaining $n$ (whenever $b = k + 1$) require $O(k)$ time to compare all $k$ possible choices of running the next advertisement and taking the minimum of them. Thus, we can solve all subproblems in $O(nk) \cdot O(1) + O(n) \cdot O(k) = O(nk)$. We will fill out our table and our parent pointers in exactly the same way as before, and follow the parent pointers back from our overall subproblem stored at $T(0, k + 1)$, just as before.

# Part B

**Problem 5-3.** [45 points] **Halloween**

Prof. Neko is really excited about Google's Halloween Doodle this year.[1] She now wants to beat the world record. In particular, Prof. Neko needs to solve the following problem.

There are $g$ ghosts on the screen, labeled $1, 2, \ldots, g$. Each ghost has a sequence of game moves needed to eliminate it. There is a small constant number $m$ of possible game moves, which we denote $A, B, C, \ldots$ below. (In the game, they are —, |, $\vee$, $\wedge$, and $\lightning$.) We denote the required move

[1] Try it out yourself! `https://www.google.com/doodles/halloween-2016`

sequence of ghost $i$ by $s_i = (s_i[0], s_i[1], \ldots, s_i[\ell_i - 1])$ where $\ell_i$ is the length of the sequence $s_i$. Note that different ghosts can have sequences of different length. Prof. Neko now needs to find a sequence of moves so that all of the $g$ ghosts disappear. A ghost ***disappears*** when all of the moves in its sequence have been played, in that order, potentially with other moves in between.

It is not hard to come up with a sequence that makes all ghosts disappear. We can simply play the moves of the first ghost, then the moves of the second ghost, etc. Note that this can take up to $\sum_{i=1}^{g} \ell_i$ moves in total, but there might be much shorter move sequences. In particular, consider the following example:

Ghost 1: $(A, B, B, B)$
Ghost 2: $(C, B, B, B, B)$

First playing all the moves of Ghost 1 followed by the moves of Ghost 2 takes nine moves in total. Instead, we could play $(A, C, B, B, B, B)$, which is only six moves and still makes all ghosts disappear. After making this realization, Prof. Neko now wonders how to find the *shortest* possible move sequence for a given set of ghosts.

**(a)** [20 points] First, we consider the $g = 2$ case. Implement a dynamic programming algorithm that computes the shortest possible move sequence that makes both ghosts disappear in the `double_kill` method. The algorithm should run in $O(\ell_1 \cdot \ell_2)$ time.

**Solution:** We can solve the problem via dynamic programming. The state space is the pair of positions $(i_1, i_2)$ in the move sequences of the two ghosts. For a given pair $(i_1, i_2)$, we keep track of the smallest number of moves we can play in order to cover the prefixes $s_1[1], \ldots, s_1[i_1]$ and $s_2[1], \ldots, s_2[i_2]$.

In the recursive update formula, we have to distinguish two cases:

- $s_1[i_1] = s_2[i_2]$: In this case, playing the symbol $s_i[i_1]$ (or equivalently, $s_2[i_2]$) covers the current move in both ghosts. So we can use the value of the sub-solutoin $(i_1 - 1, i_2 - 1)$ and add 1 for the current move.
- Otherwise, we can only cover one of the two current symbols with the current move. We play the symbol that leads to the smaller sub-solution, i.e., the minimum of the solution values corresponding $(i_1 - 1, i_2)$ and $(i_1, i_2 - 1)$. As before, we then add 1 for the current move.

In order to reconstruct the actual move sequence at the end, we also have to keep track of parent pointers.

We have to make sure we handle the boundary cases ($i_1 = 0$ or $i_2 = 0$) correctly. If one of the positions is 0, we can simply play the move sequence of the other ghost. So at the beginning we can set the solution value of the boundary cases $(i_1, 0)$ to $i_1$ and the solution values of $(0, i_2)$ to $i_2$.

Overall, we have $O(\ell_1 \cdot \ell_2)$ states in our dynamic program. Moreover, each state takes constant time to compute. Hence we get the desired time complexity of $O(\ell_1 \cdot \ell_2)$.

**(b)** [10 points] Next, extend your algorithm to the $g = 3$ case in the `triple_kill` method. This algorithm should run in $O(\ell_1 \cdot \ell_2 \cdot \ell_3)$ time.

**Solution:** As before, we use dynamic programming. This time, the state space consists of a 3-tuple of current positions $(i_1, i_2, i_3)$. As before, we keep track of the best solution for a given prefix of the move sequences. The overall idea is the same as in the previous part. However, the update step becomes more complicated. We now have three cases:

1. All three ghosts have the same symbol at the current positions.
2. Exactly two of the three ghosts have the same symbol at the current positions.
3. The three ghosts have pairwise distinct symbols at the current positions.

In the first case, we play the common move. In the second case, we have to choose whether the common move or the move of the single ghost is better to play. We can make this decision based on the respective sub-solutions we have computed before. In the third case, we have to decide between all three possible moves.

We now have $O(\ell_1 \cdot \ell_2 \cdot \ell_3)$ states in our dynamic program. As before, each state takes constant time to compute. This leads to the desired time complexity.

Computing the boundary states (one or more of the indices are 0) also requires additional care. If two of the three ghosts are at index 0, we can use the index of the remaining ghost as solution value. If only one of the ghosts is at index 0, we can invoke the solution from the previous part to find the best move sequence for the two ghosts with non-empty prefixes.

**(c)** [5 points] Another way to view this problem is as a graph. Describe how you would formulate the problem from part (b) as a shortest-paths problem in a graph. Your algorithm should still run in $O(\ell_1 \cdot \ell_2 \cdot \ell_3)$ time.

**Solution:** Our graph consists of one node per state of the dynamic program. To be precise, we have one node per 3-tuple $(i_1, i_2, i_3)$ of possible positions in the three move sequences. Hence the total number of nodes is $O(\ell_1 \cdot \ell_2 \cdot \ell_3)$.

We now add a directed edge between nodes $v_1$ and $v_2$ if playing a game move can bring us from state $v_1$ to $v_2$. The number of edges per node is constant, so overall we have $O(V)$ edges. In this new graph, the shortest path from a node $u$ to another node $v$ corresponds to the shortest possible move sequence bringing us from state $u$ to state $v$. So running a BFS from node $(0, 0, 0)$ computes the shortest path to $(\ell_1, \ell_2, \ell_3)$ in time $O(V + E)$, which is $O(\ell_1 \cdot \ell_2 \cdot \ell_3)$. This move sequence is an optimal solution to the original problem.

**(d)** [5 points] For either the graph or the dynamic programming approach, extend your algorithm so that it applies to an arbitrary, fixed number of ghosts $g$, and runs in $O(\prod_{i=1}^{g} \ell_i)$ time.

**Solution:** Our state space is now is the set of positions in all $g$ ghosts. Hence the

size of the state space is $O(\prod_{i=1}^{g} \ell_i)$. For a given state, the number of relevant sub-solutions (and the number of edges per node in the graph formulation) now depends on $g$. For fixed $g$, we can still consider this number as a constant. So overall the time complexity of the algorithm is $O(\prod_{i=1}^{g} \ell_i)$.

**(e)** [5 points] In the game, there is a special symbol $\lightning$ that has a different behavior from the other symbols: it can only be played when one of the ghosts currently has it at the next symbol in its sequence, and when played, it eliminates the next symbol of *all* $g$ ghosts. Describe how to modify your dynamic program from part (d) to handle this behavior of $\lightning$.

**Solution:** We modify our recursive update formula as follows. If we are in a state where one of the ghosts has $\lightning$ as its current symbol, we can execute the special move. In our dynamic program, this corresponds to considering the state $(i_1 - 1, i_2 - 1, \ldots, i_g - 1)$ as a possible sub-solution.