Problem Set 6

All parts are due Thursday, May 5 at 11:59PM. Please download the .zip archive for this problem set, and read the comments in the gradient descent template file carefully to ensure you format your solutions correctly.

Part A

Problem 6-1. Gradient Descent [20 points]
In this problem, we will use gradient descent to locate the value of \( x \) that minimizes a parabolic function \( f(x) = ax^2 + bx + c \), where \( a > 0 \). We use the notation \( x^* \) to denote this “minimizing” value of \( x \). The idea is to start with an initial value \( x_0 \). Then for each successive value of \( i = 0, 1, \ldots \), we compute \( x_{i+1} = x_i - \eta f'(x_i) \), where \( \eta \) is some constant positive-real-valued step size.

Please provide explanations to your solutions wherever appropriate.

(a) [2 points] First consider the particular function \( f(x) = (2x - 3)^2 \), with starting value \( x_0 = 0 \). Find a specific value of \( \eta \) for which the sequence of \( x_i \) values fails to converge to the correct value \( x^* = \frac{3}{2} \). What is the sequence \( x_0, x_1, \ldots \) generated using this value of \( \eta \)?

**Solution:** We have \( f'(x) = 4(2x - 3) = 8x - 12 \). Consider \( \eta = 1/4 \). Then, \( x_1 = x_0 - (1/4)(f'(x_0)) = 0 - (1/4)(0 - 12) = 0 - (-3) = 3 \). Similarly, \( x_2 = 3 - (1/4)(f'(x_1)) = 3 - (1/4)(24 - 12) = 3 - 3 = 0 \). It’s clear that the sequence of \( x_i \) values will oscillate between 3 and 0, and therefore not converge.

Notice that \( f(x) = 4(x - \frac{3}{2})^2 \) and \( f'(x) = 8(x - \frac{3}{2}) \). Thus, \( f'(x) \) can be read as 8 times the distance from the optimum \( x^* \). Thus, any step size greater than \( \frac{1}{4} \) will diverge (equivalently, any step size greater than \( \frac{2}{f''(x)} = \frac{1}{4} \) for this specific quadratic).

(b) [2 points] For the same function \( f(x) = (2x - 3)^2 \), with the same starting value \( x_0 = 0 \), find another value of \( \eta \) for which the sequence does converge to \( x^* = \frac{3}{2} \). Try to choose \( \eta \) so that the algorithm converges quickly. In particular, consider an “accuracy” parameter \( \epsilon > 0 \), and define an index \( i \) to be \( \epsilon \)-good provided that \( |f(x_i) - f(x^*)| \leq \epsilon \) for every \( j \geq i \). Find \( \eta \) so that the smallest index \( i \) that is \( \epsilon \)-good is \( O(\log \frac{1}{\epsilon}) \).

**Solution:** Consider \( \eta = 1/16 \). Then, we claim that for \( i \geq 0 \), \( x_i = \frac{3}{2} - \frac{3}{2^i} \). We prove this by induction. For this choice of \( f \), we have \( f'(x) = 8x - 12 \). For \( i = 0 \), we have \( x_0 = 0 = \frac{3}{2} - \frac{3}{2} \), as needed. Now, suppose that, for some \( i \), \( x_i = \frac{3}{2} - \frac{3}{2^i} \). Then, \( x_{i+1} = x_i - \frac{1}{16}(f'(x_i)) = x_i - \frac{1}{16}(8x_i - 12) \). By inductive hypothesis, the right-hand
We claim this makes the smallest $\epsilon$-good index $O(\log \frac{1}{\epsilon})$. In particular, notice that the smallest $\epsilon$-good index is bounded above by $\lceil \log \frac{1}{\epsilon} \rceil$, where we note that for sufficiently close $x$ and $x^*$, $|f(x) - f(x^*)| \leq |x - x^*|$, so if $|x - x^*| \leq \epsilon$, so too will be $|f(x) - f(x^*)|$ for small $\epsilon$.

(c) [3 points] Generalize your work in part (b) to give a good step size for an arbitrary quadratic function of the form $f(x) = ax^2 + bx + c$, where $a > 0$, and with $x_0$ an arbitrary real number. In particular choose a step size $\eta$ so you approach the minimizer $x^*$ exponentially fast. In other words, choose $\eta$ such that the smallest index $i > 1$ that is $\epsilon$-good is equal to $c \log \left( \frac{|a|x_0 - x^*|}{\epsilon} \right)$ for some constant $c$ where $x^*$ is the minimizer of the function $f(x)$. Try to choose $\eta$ so that the smallest index $i$ that is $\epsilon$-good is $O(\log \left( \frac{|a|x_0 - x^*|}{\epsilon} \right))$, where $x^*$ is the minimizer of the function $f$.

Using your step size, give pseudocode for an algorithm that takes an arbitrary quadratic function of the given form and an arbitrary real number $x_0$, and outputs a value $x$ such that $|f(x) - f(x^*)| \leq \epsilon$.

**Solution:** Consider $\eta = \frac{1}{2a}$. Note that $f'(x) = 2ax + b$, and the minimizer $x^*$ can be computed analytically to be $-\frac{b}{2a}$. Without loss of generality, assume that $x_0 > x^*$; the other case follows by symmetry.

We show that the distance from our current estimate $x_i$ to $x^*$ gets halved at every step. Specifically, we show that $x_i = \frac{-b}{2a} + \frac{x_0 + \frac{b}{2a}}{2^i}$. We proceed by induction. For the base case, $i = 0$, simple algebra yields that $x_0 = \frac{-b}{2a} + \frac{x_0 + \frac{b}{2a}}{2^0}$. For the inductive step, assume that $x_i = \frac{-b}{2a} + \frac{x_0 + \frac{b}{2^i}}{2^i}$. Then we have

$$x_{i+1} = x_i - \frac{1}{4a} (2ax_i + b) = \frac{-b}{2a} + \frac{x_0 + \frac{b}{2^i}}{2^i} - \frac{1}{4a} (2a\left(\frac{-b}{2a} + \frac{x_0 + \frac{b}{2^i}}{2^i}\right) + b) = \frac{-b}{2a} + \frac{x_0 + \frac{b}{2^i}}{2^{i+1}}$$

The argument regarding the time complexity is similar to part (b). At each step, we halve the distance $|x_i - x^*|$. Now we continue until this distance is less than $\frac{\epsilon}{2}$. Since $|f(x) - f(x^*)| \leq a|x - x^*|$ for $x$ sufficiently close to the minimizer $x^*$, this ensures $|f(x) - f(x^*)| \leq \epsilon$. 
Problem Set 6


GRADIENT-DESCENT-MINIMIZER\((a, b, c, x_0, \epsilon)\)

1. Initialize \(x = x_0\)
2. while \(|f(x) - f(\frac{-b}{2a})| > \epsilon\)
   3. \(x = x - \frac{1}{4a}(2ax + b)\)
4. return \(x\)

Pseudocode based on recitation notes is ok too:

GRADIENT-DESCENT-MINIMIZER\((a, b, c, x_0, \epsilon)\)

1. Initialize \(x = x_0\)
2. while \(|\nabla f(x(t))| > \epsilon\)
   3. \(x = x - \frac{1}{4a}(2ax + b)\)
4. return \(x\)

The above is a good approximation for the simplified version of gradient descent we presented. Also note that the above is good specially close to the minimizer (i.e. when distance is less than 1).

\(\textbf{(d) [3 points]}\) Again consider an arbitrary quadratic function of the form \(f(x) = ax^2 + bx + c\), where \(a > 0\). Now consider a different strategy for finding an approximation to \(x^*\), based on binary search. (Note that this strategy works well for single-variable functions, but does not extend to multi-variable functions, whereas gradient descent still works in such cases.) This time, suppose we are given two values \(x_0\) and \(x_1\), such that \(f'(x_0) < 0\) and \(f'(x_1) > 0\). We are also given an accuracy parameter \(\epsilon > 0\).

Describe in words and pseudocode a simple algorithm that outputs a value \(x\) such that \(|f(x) - f(x^*)| \leq \epsilon\). Your algorithm should take time \(O(\log(\frac{a|x_0 - x_1|}{\epsilon}))\).

\textbf{Solution:} We maintain \(x_l\) and \(x_u\) with the invariant \(x_l \leq x^* \leq x_u\). The stopping condition is similar to the above, where for sufficiently close \(x\) to \(x^*\), we have \(|f(x) - f(x^*)| < |ax - ax^*|\); thus, we wish for \(|x - x^*| < \epsilon/a\) as the stopping condition. However, if we make our guess either of \(x_l\) or \(x_u\), and we have \(|x_u - x_l| < \epsilon/a\), this is also sufficient because \(x^*\) is in between these two values.

Initialize \(x_l = x_0, x_u = x_1\). Then, iterate \(x_{temp} = (x_l + x_u)/2\), and if \(f'(x_{temp}) > 0\), update \(x_u = x_{temp}\); otherwise, \(x_l = x_{temp}\). This gets the guaranteed runtime because it’s a binary search on a space originally of size \(|x_0 - x_1|\). Similarly, binary search halves at each step the distance to \(x^*\), thus, the time complexity argument follows in a similar manner as in the previous part.
**Binary-Search-Minimizer** \((f, f', x_0, x_1)\)

1. Initialize \(x_u = x_1, x_l = x_0\)
2. while \(|f(x_u) - f(x_l)| > \epsilon\)
   3. \(x_{temp} = \frac{x_u + x_l}{2}\)
   4. if \(f'(x_{temp}) > 0\)
      5. \(x_u = x_{temp}\)
   6. else
      7. \(x_l = x_{temp}\)
8. return \(x_u\)

(e) [4 points] Show that for this very special case (very simple single variable convex function) Gradient Descent can find the optimal minimizer \(x^*\) in one single step. In particular find a better setting of the step size \(\eta\) such that one can decrease the objective function \(f(x)\) as much as possible with the current estimate of the gradient for any Gradient Descent update. Briefly, comment on why your method works as well as it does for this type of function and why it might not work for other more complicated functions.

**Hint:** consider the Taylor expansion of \(f(x)\) and the error term considered in lecture and choose a step size that causes the greatest decrease in \(f(x)\) while still approaching \(f(x)\)

**Solution:** Recall the Taylor expansion for a function:

\[
f(x + \epsilon) = f(x) + \epsilon f'(x) + \frac{\epsilon^2}{2!} f''(x) + \frac{\epsilon^3}{3!} f'''(x) + \cdots + \frac{\epsilon^k}{k!} f^{(k)}(x) + \cdots\]

recall that we defined the error term as follow \(Err(\epsilon, x) = \frac{\epsilon^2}{2!} f''(x) + \frac{\epsilon^3}{3!} f'''(x) + \cdots + \frac{\epsilon^k}{k!} f^{(k)}(x) + \cdots\). We can actually make the analysis from lecture exact for quadratic functions because \(f^{(k)}(x) = 0\) for all \(k \geq 3\) (i.e. derivatives of 3rd degree or higher are zero for quadratic functions). Therefore we have \(Err(\epsilon, x) = \frac{\epsilon^2}{2!} f''(x)\) and we have \(f(x + \epsilon) = f(x) + \epsilon f'(x) + Err(\epsilon, x) = f(x) + \epsilon f'(x) + \frac{\epsilon^2}{2!} f''(x)\). Let \(\epsilon = -\eta f'(x)\). Then the equation becomes: \(f(x - \eta f'(x)) = f(x) + -\eta f'(x)^2 + Err(-\eta f'(x), x) = f(x) + -\eta f'(x)^2 + \frac{\eta^2 f'(x)^2}{2} f''(x)\). To have any decrease in the update we need: \(-\eta f'(x)^2 + Err(-\eta f'(x), x) < 0\). Thus we consider: \(-\eta f'(x)^2 + \frac{\eta^2 f'(x)^2}{2} f''(x) < 0\) \(\implies -\eta + \frac{\eta^2}{2} f''(x) = \eta \left(\frac{\eta}{2} f''(x) - 1\right) < 0\) and since we are only considering \(\eta > 0\) (because we want a decrease in the steepest descent) we want a step-size in the following range \(0 < \eta < \frac{2}{f''(x)}\). However, to get the value that gets the largest decrease we want the most negative value of \(\frac{\eta^2 f'(x)^2}{2} f''(x) - \eta f'(x)^2\). In other words, the goal is to make the decrease due to \(-\eta f'(x)^2\) as large as possible compared to the error term \(Err(-\eta f'(x), x) = \frac{\eta^2 f'(x)^2}{2} f''(x)\). This is achieved for the following value (which can be easily found using calculus):
\[ \eta = \frac{1}{f''(x)} = \frac{1}{2a} \]

Now plugging in that step size to Gradient Descent gives that we converge to \( x^* = -\frac{b}{2a} \) in one single step for any initial \( x^{(0)} \):

\[ x^{(1)} = x^{(0)} - \frac{1}{2a} (2ax^{(0)} + b) = x^{(0)} - x^{(0)} - \frac{b}{2a} = -\frac{b}{2a} = x^* \]

The reason this method worked as cleanly as it did was because quadratic functions are simple and we can ignore a lot of complicated high degree terms in the error function due to \( f^{(k)}(x) = 0 \). Also, the derivatives (1st and 2nd) for quadratics have closed analytic forms that are easy to compute, which makes the analysis easy to follow. Furthermore, the global minimizer was easy to find in closed analytic form, so it was easy to check if we got to the true global minimum. All of the above are not generally true.

However, notice that even if the function was easier to handle, its not guaranteed that a step of gradient descent will get us exactly to the local minimum in general. In reality, we might only be able to get as close as we can to the local minimum given that we only know our previous guess \( x^{(t)} \) and the direction of steepest descent \(-\nabla f(x)\).

(f) [4 points] In the previous question one was able to find an extremely good step-size \( \eta \) in part because quadratic functions are very simple convex functions. In general that is not the case, however, one can still get a good step size in general. In particular given the trajectory of greatest change of \( f(x) \) one can try to compute the step-size \( \eta \) that gives us the greatest decrease in our objective function \( f(x) \) while still approaching some local minimum of \( f \) without overshooting it. Concretely, one could solve the following optimization problem at each gradient descent iteration to get the ideal step-size:

\[ \eta^{(t+1)} = \arg \min_{\eta \in \mathbb{R}} f(x^{(t+1)}), \text{ subject to } x^{(t+1)} = x^{(t)} - \eta \nabla f(x^{(t)}) \]

Describe an efficient algorithm to compute an approximation to the optimal step-size if you are given an upper bound \( \eta_{max} \) on the best \( \eta^* \). Also provide the update step for Gradient Descent such that it incorporates this method to choose the step size (notice that you may break ties arbitrarily).

[Hint 1: notice the above optimization is over a single variable]

[Hint 2: use part (d) as inspiration and assume \( f(x) \) is easy to compute]

**Solution:**

To ease the description of the solution let:

\[ h(\eta) = f \left( x^{(t)} - \eta \nabla f(x^{(t)}) \right) \]
One can basically solve the above with a line search. There are two ways to do this. One way to do it is with binary search in the range interval $(0, \eta_{\text{max}}]$. Or to do repeated doubling until you find you can’t decrease $h(\eta)$ anymore. However, any reasonable solution to this should guarantee that whatever step size we return is better than our initial estimates.

Notice that one can re-write the optimization procedure to obtain the optimal $\eta$ as follow:

$$\eta^{(t+1)} = \arg\min_{\eta \in \mathbb{R}} f(x^{(t)} - \eta \nabla f(x^{(t)})) = \arg\min_{\eta \in \mathbb{R}} h(\eta)$$

The basic idea is to have a window from $\eta_l$ to $\eta_u$ and do binary search in that interval in a similar fashion as in part (d). The idea is to narrow down $\eta_l$ to $\eta_u$ small enough so to nail down $\eta^*$ as much as possible according to some pre-specified $\epsilon$ or pre-specify the number of iterations $T$ one is willing to wait. Finding $\eta$ is a single variable minimization with $h(\eta) = f(x^{(t)} - \eta \nabla f(x^{(t)}))$, thus, if the conditions of part (d) apply, one can basically re-use the algorithm in that part. The only caveat might be that $h'(\eta)$ might be difficult to derive in an analytic form for complicated functions. However, if $h(\eta)$ is easy to compute, one can numerically estimate it and since its only one variable its not too expensive to approximate $h'(\eta)$ using $h(\eta)$ (and the definition of a derivative). Using that idea and if $h(\eta)$ has similar condition to (d), one could re-use (d).

In the case where the above does not hold one can instead just update the upper bound or the lower bound when one find values of $\eta$ that have smaller values of $h(\eta)$. In particular you can implement the following algorithm (students do not have to provide pseudocode but this is what it would be):

**Binary-Search-Minimizer-Step-Size**($f, \nabla f, x^{(t)}, \eta_0, \eta_1, T$)

1. Define the function $h(\eta) = f(x^{(t)} - \eta \nabla f(x^{(t)}))$
2. Initialize $\eta_u = \eta_1, \eta_l = \eta_0$
3. For $i = \{1, \ldots, T\}$
4. \hspace{1em} $\eta_{\text{temp}} = \frac{\eta_u + \eta_l}{2}$
5. \hspace{1em} If $h(\eta_u) \leq h(\eta_{\text{temp}})$
6. \hspace{2em} $\eta_u = \eta_{\text{temp}}$
7. \hspace{1em} Else
8. \hspace{2em} $\eta_l = \eta_{\text{temp}}$
9. Return $\eta_l$

Notice that this algorithm relies that we maintain the invariant $h(\eta_l) \leq h(\eta_u)$ so that we always return a step size better than our initial estimates.

If one doesn’t know the value $\eta_{\text{max}}$ one can instead do repeated doubling from a minimum estimate of $\eta$. Notice that we always have a good idea of the minimum value of
η because a negative value of η points in the direction opposite of the gradient. Thus, negative values are not considered. Starting with a minimum estimate of the step size at some point, say η_l = 0 + ϵ where ϵ is a small value, one can repeatedly double η_l until the function stops decreasing h(η) (or increases). When this happens halt and return η_l:

DOUBLING-LINE-SEARCH(x_t, ∇f, η_min, T)
1 define the function h(η) = \( f(x(t) - η\nabla f(x(t))) \)
2 Initialize \( η_{current} = η_{min} \)
3 for i = \{1, ⋯, T\}
4 \( η_{current} = 2\eta_{current} \)
5 if h(η_l) < h(η_{current})
6 return η_l
7 return η_l

Now for the gradient descent update with this new step size choice will be now:

1 \( \eta(t+1) = \arg\min_{\eta \in R} h(\eta) = \arg\min_{\eta \in R} f(x(t) - η\nabla f(x(t))) \)
2 \( x(t+1) = x(t) - \eta(t+1)\nabla f(x(t)) \)

Notice that the step size is changed at every iteration of Gradient Descent now.

Students could come up with many variants of these. This is fine as long as their algorithm actually produces a better step size than the ones already stated in the question (i.e. better than just η = 0 or η = η_max and that the runtime isn’t exponential.

An example that is acceptable is to uniformly sample different step-sizes between (0, η_max] and return the one that produces the best step size. Or even, do binary search or repeated doubling from each of them. This solution is not a bad idea because it has the advantage that it may find better local minimums than the first methods proposed but is more computationally expensive. Notice that technically one should also worry about getting stuck in saddle points, however, this is going a beyond what we asked and expected.

(g) [2 points] Now we consider gradient search again, but now for a cubic function, say \( f(x) = x^3 \). Let the step size η be 1. For which values of \( x_0 \) does gradient search converge? (You don’t need to identify the value of x to which it converges.)

Solution: We have \( f'(x) = 3x^2 \) and \( x_{i+1} = x_i - 3x_i^2 \). Notice that if any \( x_i \) is negative, then \( x_{i+1} \) will be a negative number of even larger magnitude, so the update sizes from that point onward will be larger and larger, and thus will not converge. In particular, \( x_1 \) can only be nonnegative if \( x_0 \) is in the range \([0, \frac{1}{3}]\).

For these choices of \( x_0 \), we are guaranteed convergence, because the sequence \( x_0, x_1, \ldots \) is decreasing monotonically and bounded (by the monotone convergence theorem). To prove this, if \( x_i \in [0, \frac{1}{3}] \), then \( x_{i+1} \) is too, and the update \( 3x_i^2 \) is nonnegative, so it’s monotonically decreasing and bounded below by 0, as desired.
Problem 6-2. [20 points] Optimal Merging

Fodder’s, Inc. is trying to finalize its 2016 rankings of Boston restaurants. It has two lists of restaurants, say \( A_1, A_2, \ldots, A_m \) and \( B_1, B_2, \ldots, B_n \), each of which is already ranked from highest to lowest. (For example, \( A \) might contain relatively new restaurants, whereas the \( B \) restaurants have been around awhile.) All Fodder’s must do is to merge the lists into a single list, preserving the order of the two sublists.

It turns out that some of the \( A \) restaurants care very much that they appear ahead of certain \( B \) restaurants on the merged list, and vice versa. In fact, they care enough that they offer to donate money to a local food bank if they obtain that ordering.

Formally, the \( A \) restaurants’ proposed donations appear in a matrix \( M_A \), where \( M_A_{i,j} \) represents the (nonnegative integer) number of dollars that \( A_i \) will donate if it is ranked ahead of \( B_j \). Similarly, the \( B \) restaurants’ proposed donations appear in a matrix \( M_B \), where \( M_B_{i,j} \) represents the number of dollars that \( B_j \) will donate if it is ranked ahead of \( A_i \).

The value of a merge is the total amount of money that would be donated as a result of that merge. Formally, that is the sum of all the entries \( M_A_{i,j} \) for pairs such that \( A_i \) precedes \( B_j \) in the merged list, plus all the entries \( M_B_{i,j} \) such that \( B_j \) precedes \( A_i \) in the merged list. Fodder’s is interested in determining the maximum possible value of any merge of the two lists.

In all parts of this problem, provide both a clear explanation and pseudocode.

(a) [5 points] Describe a recursive algorithm to solve the optimal merge problem. The inputs are:

- A pair of lists \( A_1, \ldots, A_m \) and \( B_1, B_2, \ldots, B_n \).
- Matrices \( M_A \) and \( M_B \) of integers; both have dimensions \( m \) by \( n \).

The output should be the largest value of a list that can be obtained by merging lists \( A_1, \ldots, A_m \) and \( B_1, \ldots, B_n \), where “value” is defined above.

Analyze the time complexity of your algorithm.

Solution: For convenience, we will create two pre-processed matrices, \( M_A' \) and \( M_B' \), which we will refer to in the remainder of the solution. In particular, let the \( ij \)th entry of \( M_A' \) be the total value contributed by \( A_i \) if it is before \( B_j \) but after \( B_{j-1} \). This makes \( M_A'_{i,j} = \sum_{k \leq j} M_A_{i,k} \). Similarly, define \( M_B'_{i,j} = \sum_{i \leq k \leq n} M_B_{k,j} \).

Working with decreasing \( i \) and \( j \), we can compute these matrices in \( O(mn) \) time.

For \( 1 \leq i \leq m + 1, 1 \leq j \leq n + 1 \), let \( V_{i,j} \) be the optimal solution to the problem of merging the suffix list \( A_i, \ldots, A_m \) with \( B_j, \ldots, B_n \), that is, the largest value of a list that can be obtained by merging these two suffixes. We wish to compute \( V_{1,1} \).

The base cases are \( V_{i,n+1} = V_{m+1,j} = 0 \) for all \( i, j \), because trivially there exist no pairs which contribute donations.

We then have the following recursion for \( V_{i,j} \), where \( 1 \leq i \leq m, 1 \leq j \leq n \). Either the first term in the merge is \( A_i \) or \( B_j \), and the first entry will contribute its full value; thus, \( V_{i,j} = \max(V_{i,j+1} + M_B'_{i,j}, V_{i+1,j} + M_A'_{i,j}) \).
For time complexity, we have the recurrence $T(m, n) = T(m, n - 1) + T(m - 1, n) + O(1)$ with base cases $T(0, 0) = O(1), T(m, 0) = O(1)$ and $T(0, n) = O(1)$. Choose $c$ to be the maximum of the constant terms in the four equations. We show by induction that $T(m, n) = T(m, n - 1) + T(m - 1, n) + c$ is upper bounded by $c'(2^{m+n+1} - 1)$ for some sufficiently large $c'$.

**Base cases:** For $m = n = 0$, we have $2^{m+n+1} - 1 = 2^{0+0+1} - 1 = 1$, so $T(0, 0) \leq c \leq c(2^{m+n+1} - 1)$, as needed. For $m \geq 1$, $n = 0$, we have $2^{m+n+1} - 1 \geq 2^{0+0+1} - 1 = 1$, so $T(m, 0) \leq c \leq c(2^{m+n+1} - 1)$, as needed. Similarly, for $m = 0$, $n \geq 1$, we have $T(0, n) \leq c(2^{m+n+1} - 1)$.

**Inductive step:** We must show that $T(m, n) \leq c(2^{m+n+1} - 1)$. We know that $T(m, n) \leq T(m, n - 1) + T(m - 1, n) + c$. By the inductive hypothesis, the right-hand side is

$$\leq c(2^{m-1+n+1} - 1) + c(2^{m+n-1+1} - 1) + c = 2c(2^{m+n} - 1) + c = c(2^{m+n+1} - 1),$$

as required.

One can also show that the runtime is exponential in at least one of the variables in the worst case as follows: We have $T(m, n) = T(m - 1, n) + T(m, n - 1) + c$. Let $k = \min(m, n)$. Then we can lower bound the recurrence by:

$$T(m, n) \geq T(k, k) \geq T(k - 1, k) + T(k, k - 1) + c \geq 2T(k - 1, k - 1) + c.$$

This yields an exponential lower bound to the runtime with respect to one of the variables in the worst case.

**Preprocess-Matrices($A, B, MA, MB$)**

```python
1 Initialize $MA', MB'$ to be $m$ by $n$ matrices
2 for $i = \text{len}(A)$ to 1
3     for $j = \text{len}(B)$ to 1
4         if $j == \text{len}(B)$
5             $MA'_{i,j} = 0$
6         else
7             $MA'_{i,j} = MA_{i,j} + MA'_{i,j+1}$
8     for $j = \text{len}(B)$ to 1
9     for $i = \text{len}(A)$ to 1
10        if $i == \text{len}(A)$
11           $MB'_{i,j} = 0$
12        else
13           $MB'_{i,j} = MB_{i,j} + MB'_{i+1,j}$
14 return $MA', MB'$
```
OPTIMAL-MERGE($A, B, MA, MB, s, t$)
1  if $s == \text{len}(A) + 1$ or $t == \text{len}(B) + 1$
2      return 0
3  $MA', MB' = \text{PREPROCESS-MATRICES}(A, B, MA, MB)$
4  return $\max(\text{OPTIMAL-MERGE}(A, B, MA, MB, s, t + 1) + MB'_{s,t},$
5  $\text{OPTIMAL-MERGE}(A, B, MA, MB, s + 1, t) + MA'_{s,t})$

(b) [5 points] Use Dynamic Programming with memoization to make your recursive algorithm from part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

Solution: Subproblems in this case correspond simply to the intermediate values $V_{i,j}$. So, we memoize all previously computed values.

Computing a new value $V_{i,j}$ is a comparison between $MA'_{i,j} + V_{i+1,j}$ and $MB'_{i,j} + V_{i,j+1}$, both of which we can compute in $O(1)$ due to memoization. Thus, the recursion now costs $O(mn)$ due to the $mn$ subproblems. Combining with the $O(mn)$ cost from preprocessing, the algorithm is $O(mn)$. In the code below, assume we have a globally initialized table $V$ keyed by $i, j$.

OPTIMAL-MERGE-DP($A, B, MA, MB, s, t$)
1  if $s == \text{len}(A) + 1$ or $t == \text{len}(B) + 1$
2      $V[i, j] = 0$
3      return 0
4  $MA', MB' = \text{PREPROCESS-MATRICES}(A, B, MA, MB)$
5  if $s, t + 1 \in V\text{.keys()}$
6      $\text{merge}1 = V[s, t + 1]$
7  else
8      $\text{merge}1 = \text{OPTIMAL-MERGE-DP}(A, B, MA, MB, s, t + 1)$
9      $V[s, t + 1] = \text{merge}1$
10     if $s + 1, t \in V\text{.keys()}$
11        $\text{merge}2 = V[s + 1, t]$
12     else
13        $\text{merge}2 = \text{OPTIMAL-MERGE-DP}(A, B, MA, MB, s + 1, t)$
14        $V[s + 1, t] = \text{merge}2$
15  return $\max(\text{merge}1 + MB'_{s,t}, \text{merge}2 + MA'_{s,t})$

(c) [5 points] Now use Dynamic Programming with bottom-up calculation to make your recursive algorithm from part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

Solution: We can use any iterative calculation order which corresponds to a topological sort on the dependency graph for the $V_{i,j}$ values, and proceed with the calculation of the $V_{i,j}$ values in that iterative order. The dependency graph has vertices corresponding to the values $i, j$ and has an edge to $(i, j)$ from $(i, j + 1)$ and $(i + 1, j)$ for
all values $1 \leq i \leq m, 1 \leq j \leq n$. For example, we can proceed by iterating through decreasing $i$, and for each $i$ iterating through decreasing $j$, after computing the base cases. It’s clear that this represents an order consistent with the dependencies.

Then, we compute the recursion sequentially using the topological sort order. Similar to the reasoning in part b, this will take $O(mn)$ time.

**OPTIMAL-MERGE-BOTTOM-UP**($A, B, MA, MB$)

1. $MA', MB' = \text{PREPROCESS-MATRICES}(A, B, MA, MB)$
2. Initialize $V$ to be $m + 1$ by $n + 1$ matrix
3. for $i = 1$ to $\text{len}(A) + 1$
   4. $V_{i,n+1} = 0$
4. for $j = 1$ to $\text{len}(B) + 1$
   5. $V_{m+1,j} = 0$
6. for $i = \text{len}(A)$ to 1
   7. for $j = \text{len}(B)$ to 1
      8. $V_{i,j} = \max(V_{i,j+1} + MB_{i,j}', V_{i+1,j} + MA_{i,j}')$
9. return $V_{1,1}$

(d) [5 points] Modify one of your efficient algorithms to produce an actual optimal merge of the two input lists. Analyze the time complexity of the merge algorithm.

**Solution:** We will modify the algorithm from part (c). What we need to do here is in each $\max$ computation, to further store (augment the entry with) which of the two values was used, in the form $A, i$ or $B, j$.

Upon doing this, looking at $V_{1,1}$, we will be able to determine which is the first entry in the merge, and also which of $V_{2,1}$ or $V_{1,2}$ is the relevant subproblem to figure out the remaining path from. The augmentation is $\Theta(1)$, and the path tracing is proportional to the number of entries, thus $O(m + n)$. The overall algorithm remains runnable in $O(mn)$. 
Problem 6-3. [20 points] Optimal Graph Paths

Here is another problem about finding “best paths” in the directed graphs. This time, we are given a weighted directed graph $G = (V, E, w)$ with designated start node $s$ and designated target node $t \neq s$. The weights are nonnegative integers. We are also given a positive integer $k$ and a particular vector $w = (w_1, w_2, \ldots, w_k)$ of nonnegative integer weights, representing the desirable number of edges in the path and the desirable weights for the edges (in order on the path). Our job is to find a $k$-edge path from $s$ to $t$ whose weight vector is closest to the vector $w$, in the sense of
minimizing the sum of squares of the differences of the edge weights. That is, we need a vector \( x = (x_1, x_2, \ldots, x_k) \) for which the sum \( \sum_{i=1}^{n} (x_i - w_i)^2 \) is minimum.

(a) [5 points] Give pseudocode for a recursive algorithm to determine the minimum achievable cost of a path, in the problem described above. The inputs should be the directed graph \( G \), the number \( k \) and the vector \( w \). You may assume that \( G \) is given in the usual adjacency-list format, and may also assume an adjacency-list representation for the “reverse” of \( G \)—the same graph in which all the edges are reversed (that might help). The output should be the smallest cost for a path from \( s \) to \( t \) consisting of exactly \( k \) edges, where the cost of the path is the sum of squares of differences in edge weights, as described above. If no \( k \)-edge path from \( s \) to \( t \) exists, then your algorithm should output \( \infty \).

Analyze the time complexity of your algorithm.

Solution: Define, for vertex \( v \) and natural number \( i \), the value \( C(v, j) \) to be the minimum cost of a path consisting of exactly \( j \) edges from \( s \) to \( v \). Here the cost is measured as \( \sum_{i=1}^{j} (x_i - w_i)^2 \), where \( (x_1, \ldots, x_j) \) is the vector of weights of the edges on the path and \( (w_1, \ldots, w_j) \) is the length-\( j \) prefix of the given desirable weight vector. The desired result is \( C(t, k) \).

The base case of the recursion is \( C(s, 0) = 0 \), and \( C(v, 0) = \infty \) for \( v \neq s \). Then, we have the following recursion. For a path of length \( j \), we consider all possible paths of \( j - 1 \) edges ending at a vertex with an edge to \( v \). Thus, \( C(v, j) = \min_{u,v} \in E(C(u, j - 1) + (w_{u,v} - w_j)^2) \). This matches the specification; if no path of length \( j \) exists to \( v \), then by default the result must be \( \infty \).

The complexity of this algorithm is \( |V|^k \).

```
MIN-COST-PATH(E, w, k, s, t)
1   if k == 0
2     if t == s
3       return 0
4     else
5       return \infty
6   min_cost = \infty
7   for (v, t) \in E
8     min_cost = min(min_cost, MIN-COST-PATH(E, w_{1:k-1}, k - 1, s, v) + (w_{v,t} - w_k)^2)
9   return min_cost
```

(b) [4 points] Use Dynamic Programming with memoization to make your recursive algorithm from part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

Solution: The subproblems of this dynamic programming are the \( C(v, j) \) for all \( v \in V, 0 \leq j \leq k \). The algorithm is the same as the above, but the calls to the previously solved subproblems are \( O(1) \) due to memoization. We can compute the
runtime summing over the cost for each subproblem, by noting that the total runtime for the subproblems \( C(v, j) \) for a particular \( j \) over all vertices \( v \) is \( \Sigma_v (\text{indegree}(v) + 1) = |V| + |E| \). Then, summing over all such \( j \), we have a runtime of \( O(k(|V| + |E|)) \).

In the code below, assume we have a globally initialized table \( C \) keyed by \( v, j \).

**MIN-COST-PATH-DP**(\( V, E, w, k, s, t \))

```python
1  if \( k == 0 \)
2      if \( t == s \)
3          \( C[k, t] = 0 \)
4          return 0
5  else
6      \( C[k, t] = \infty \)
7      return \( \infty \)
8  \( \text{min\_cost} = \infty \)
9  for \((v,t) \in E\)
10     if \( v, k - 1 \in C\.keys() \)
11        \( \text{subpath\_cost} = C[v, k - 1] \)
12     else
13        \( \text{subpath\_cost} = \text{MIN-COST-PATH}(E, w_{1:k-1}, k - 1, s, v) \)
14        \( C[v, k - 1] = \text{subpath\_cost} \)
15        \( \text{min\_cost} = \min(\text{min\_cost}, \text{subpath\_cost} + (w_{v,t} - w_k)^2) \)
16  return \( \text{min\_cost} \)
```

(c) [4 points] Now use Dynamic Programming with bottom-up calculation to make your recursive algorithm from part (a) more efficient. Analyze the time complexity of the resulting improved algorithm.

**Solution:** We can use any iterative calculation order which corresponds to a topological sort on the dependency graph for the \( C(v, j) \) values, and proceed with the calculation of the \( C(v, j) \) values. The dependency graph has vertices corresponding to the pairs \((v, j)\) and has an edge from \((u, j - 1)\) to \((v, j)\) for all edges \((u, v)\) and all values of \( j \). For example, the topological sort might order all the \( C(v, 0) \) for all choices of \( v \) first, then all the \( C(v, 1) \), and so on.

The runtime analysis is the same as in the previous part of the problem, since the recursion is set up so the subproblems will be given for free. The runtime is then \( O(k(|V| + |E|)) \).
MIN-COST-PATH-BOTTOM-UP($V, E, w, k, s, t$)
1 Initialize $C$ to be $|V|$ by $k+1$ matrix
2 for $v \in V$
3 if $v == s$
4 $C_{v,0} = 0$
5 else
6 $C_{v,0} = \infty$
7 for $i = 1$ to $k$
8 for $v \in V$
9 $\text{min}_\text{cost} = \infty$
10 for $(u, v) \in E$
11 $\text{min}_\text{cost} = \min(\text{min}_\text{cost}, C_{u,i-1} + (w_{u,v} - w_{i})^2)$
12 return $C_{t,k}$

(d) [3 points] Describe a situation in which the memoized version of the algorithm is more efficient than the bottom-up version.

Solution: The memoized algorithm is better when it is not necessary to compute every subproblem. For example, suppose the graph can be decomposed into “layers”, where every edge connects a node in one layer to a node in the next layer. Then, for vertices $v$ in the $j^{th}$ layer, only the subproblems $C(v, j)$ are necessary, saving $\Theta(k)$ other computations on that layer. The bottom-up algorithm considers all the subproblems iteratively, so it does not skip these calculations.

General note: The implementation of a recursive algorithm incurs extra overhead for maintaining the stack of recursive calls. So for practical use, we should also take such overhead into account.

(e) [4 points] Modify one of your efficient algorithms to produce an actual best path. Analyze the time complexity of the path-finding algorithm.

Solution: We modify the solution from part (c), augmenting each memoized result with additional information about the optimal $u$ it came from in the $\text{min}$ calculation. This augmentation takes $\Theta(1)$ at every step. Then, with this information the calculation of the best path is straightforward; from $C(t, k)$, we find the optimal vertex in the $k-1$ layer, and so on. This will yield a path in $O(k)$ time, because each augmentation takes $O(1)$ to look up and traverse to the next node in the path.
**Problem Set 6**

**MIN-COST-PATH-DP**($V, E, w, k, s, t$)

1. Initialize $C$ to be $|V|$ by $k + 1$ matrix
2. Initialize $T$ to be $|V|$ by $k$ matrix
3. for $v \in V$
   4. if $v == s$
   5. $C_{v,0} = 0$
   6. else
   7. $C_{v,0} = \infty$
8. for $i = 1$ to $k$
9. for $v \in V$
10. $\text{min}_\text{cost} = \infty$
11. for $(u, v) \in E$
12. if $C_{u,i-1} + (w_{u,v} - w_i)^2 \leq \text{min}_\text{cost}$
13. $\text{min}_\text{cost} = C_{u,i-1} + (w_{u,v} - w_i)^2$
14. $T_{v,i} = u$
15. return $C_{t,k}, T$

**TRACEBACK**($k, s, t, T$)

1. Initialize $path = [t]$
2. $v = t$
3. for $i = k$ to $1$
4. $path$prepend($T_{v,i}$)
5. $v = T_{v,i}$
6. $i -= 1$
7. return $path$
Part B

Problem 6-4. [40 points] Campaign Madness

Ben Bitdiddle, starting from his humble beginnings at MIT, is now running for president of the United States under the "Fewer Psets Party". In order to most effectively campaign, Ben wants to identify swing voters, that is voters who have a decent chance of voting for Ben given some more directed campaigning. To this end, Ben has examined polling data very closely, trying to figure out how factors such as age or income influence how positively voters view him. Using this data, he wants to come up with a prediction how much a previously-unseen voter may approve of Ben.

Explicitly, Ben is given a $n \times m$ matrix, $\theta$ which contains $m$ different pieces of demographic information about $n$ individuals. For instance, the first row of $\theta$ will consist of $m$ different non-negative real numbers that correspond to some quantitative information (like age, income, number of children, years of formal education, hand size, etc.) about a particular person. For instance, if Ben polled three voters about just their age and handsizes in centimeters, the matrix might look like this:

$$
\begin{bmatrix}
21 & 20 \\
44 & 15 \\
64 & 17
\end{bmatrix}
$$

Ben also has a length $n$ vector $y$ of real numbers, the approval scores, which represent how much a person approves of Ben. The more positive the score, the more a person approves of Ben. For instance, the $i^{th}$ element of $y$ corresponds to the approval score of the $i^{th}$ person.

In order to predict the approval score of a new voter, Ben wants to construct a model that generates a guess, $\hat{y}$ based on the demographic information of that voter. He does this for the $i^{th}$ voter as

$$
\hat{y}(\theta_i) = \sum_{j=1}^{m} \theta_{ij} x_j = \theta_i \cdot x
$$

Where $\theta_i$ denotes the $i^{th}$ row of $\theta$ and $\theta_{ij}$ the element in the $j^{th}$ column of the $i^{th}$ row. In other words, we compute the approval score as the inner product between a voter’s attributes and some vector of parameters $x$. Note that for our input we will set all $\theta_{i1} = 1$, so that the $x_1$ term functions as a constant offset for our model.

Ben wants to build the best model possible and wants to do so by picking the vector of parameters $x$ that minimize the mean-squared error, $J(x)$, that is he wants to find a $x^*$ that minimizes

$$
J(x) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \theta_i \cdot x)^2 = \frac{1}{n} (y - \theta x)^\top (y - \theta x)
$$

(2)
Once Ben finds the optimal $x$ for this cost function, he will generate guesses of the approval score of brand new voters and in turn decide whether or not to devote resources campaigning for those people. You won’t have to make the decision of where Ben has to campaign, but he needs your help to build this model!

(a) [5 points] In order to perform the gradient descent algorithm, we first need an update equation. Analytically derive the update step for the gradient descent algorithm. You may leave the step size as $\eta$ for now.

**Solution:** This is a straightforward matter of taking the gradient and plugging it into the general update for gradient descent. In particular, we have

$$x^{t+1} = x^t - \eta \cdot \nabla J(x^t)$$  \hspace{1cm} (3)

Where the $j^{th}$ component of the gradient is

$$\nabla J(x^t)_j = -\frac{2}{n} \sum_{i=1}^{n}(y_i - \theta_i \cdot x)\theta_{ij}$$  \hspace{1cm} (4)

(b) [10 points] Implement gradient descent as specified in the template for the function `gradient_descent`. Your input will be a $n \times m$ 2-d array representing $m$ different pieces of demographic information for $n$ different voters, a length $n$ array for those voter’s approval scores for Ben, a length $m$ vector representing your initial guess for the parameters of this model, and a float for the step size you should use. You should return the vector of nearly optimal $x = [x_1, ..., x_m]$ as a list. Also, consider stopping the gradient descent procedure when the mean-squared value of the vector $x^t - x^{t-1}$ is below a threshold. In other words, once the update stops changing the parameters much, we should stop. See the code template for more details.

(c) [10 points] Ben has managed to poll an incredible amount of people, but realizes that if he picks a random subset of people on which to perform his update to the gradient, the overall algorithm will converge to an optimal $x$ much quicker. Specifically, if he chooses a subset of ten voters, $S$, at random, he can perform an update with the gradient of a modified cost function for just these voters.

$$J(x)^{mini} = \frac{1}{10} \sum_{i \in S} (y_i - \theta_i \cdot x)^2$$  \hspace{1cm} (5)

He can then iterate through all the voters in sets of ten following some random order. Given the same set of inputs as part (b), implement `minibatch_gradient_descent` by choosing a random subset of 10 voters on which to perform a subgradient update on.

**Hint:** This is a slightly new algorithm! The comments in the code template should be helpful in figuring out how to adapt your algorithm from part (b).
(d) [10 points] Ben realizes something else as he’s refining his algorithm. The error is convex in the parameters $x$. In other words, when traveling along a trajectory determined by the gradient, there is a single minimum value. Starting with a minimum estimate of the step size at some point and the gradient vector, he decides to use a repeated doubling procedure to find the step size that gives us a new $x$ along that trajectory at iteration of his algorithm. Your task is to implement a search along the direction of the gradient following this pseudocode:

\begin{verbatim}
DOUBLING-LINE-SEARCH($x_{start}, \nabla f, \eta_{min}, \eta_{max}$)
1 Initialize $\eta_{current} = \eta_{min}, x_{current} = x_{start} - \eta_{current} \nabla f(x_{start})$
2 while $\eta_{current} < \eta_{max}$
3 $x_{temp} = x_{current} - \eta_{current} \nabla f(x_{start})$
4 if $J(x_{temp}) < J(x_{current})$
5 $x_{current} = x_{temp}$
6 $\eta_{current} = 2\eta_{current}$
7 else break
8 return $x_{current}$
\end{verbatim}

where $J$ is our error function. Implement this in the function `line_search`. Furthermore, implement two other simple helper functions, `compute_gradient` (which is exactly the same as in part (b)) and `prediction_error`. These functions will be used as subroutines in an otherwise complete `gradient_descent_complete` which is the function that will be tested.

(e) [5 points] Alyssa P. Hacker, Ben’s longtime friend and data science consultant, notes that there is a single global minimum for $J(x)$ and that he could have just come up with a closed-form solution for $x$. Your last task is to find the closed form for $x$ in terms of the matrix $\theta$ and vector $y$.

**Hint:** It may be useful to know that for a matrix $A$ and vector $v$ that $\frac{\partial A v}{\partial v} = A^\top$ and that $\frac{\partial v^\top A}{\partial v} = A$.

**Solution:** Take the derivative of the cost with respect to the vector $x$ and set that equal to 0.

\[
J(x) = \frac{1}{n}(y^\top y - 2y^\top \theta x + x^\top \theta^\top \theta x)
\]

\[
\frac{\partial J(x)}{\partial x} = 0 = -2\theta^\top y + 2\theta^\top \theta x
\]

\[
x = (\theta^\top \theta)^{-1} \theta^\top y
\]