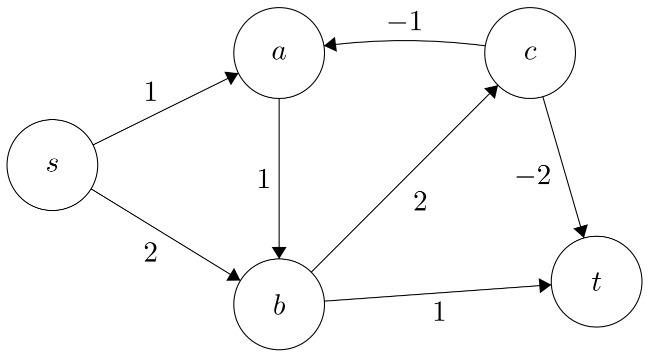
Homework 1

Solution Key

Problem 1



The following is a solution following Wikipedia's version of Bellman-Ford. However, when grading, we of course accepted solutions that followed the version covered in class (see the 2nd lecture notes). We are going to find the shorted path from s to t in the given graph by executing Bellman-Ford by hand.

For each vertex in the graph we store a "best distance found so far from s" and a "predecessor on the shortest path found so far from s". We initialize this table in the straightforward way, without any paths, by setting the distance (from s) to s to be 0, and the distances (from s) to all other nodes to be ∞; we set all predecessors to be "null".

distance	predecessor
a.dist = ∞	a.pred = null
b.dist = ∞	b.pred = null
c.dist = ∞	c.pred = null
s.dist = 0	s.pred = null
t.dist = ∞	t.pred = null

We then iterate through the edges in the order below, and for each edge (u, v), if the best path found so far to u plus the length of (u, v) is better than the best path found so far to v, then we update the distance to v to reflect this, and set the predecessor of v to u. The table below describes the *updates* this procedure makes to the above table, with the updates listed in chronological order.

edge	distance	predecessor
(s,a)	a.dist = 1	a.pred = s
(s,b)	b.dist = 2	b.pred = s
(a,b)	[no update]	[no update]
(b,c)	c.dist = 4	c.pred = b
(b,t)	t.dist = 3	t.pred = b
(c,a)	[no update]	[no update]
(c,t)	t.dist = 2	t.pred = c

These updates lead to the following final state of our tables:

distance	predecessor
a.dist = 1	a.pred = s
b.dist = 2	b.pred = s
c.dist = 4	c.pred = b
s.dist = 0	s.pred = null
t.dist = 2	t.pred = c

Additional iterations through the edges lead to no more updates, which means that we have found the shortest paths, and that none of these paths involve negative-weight cycles. (Had we iterated through the edges in a different order, it might have taken more than one pass through the edges.)

Thus the shortest path to t has weight 2, and we can find the path by tracing back the predecessor pointers: $t \leftarrow c \leftarrow b \leftarrow s$, leading to a path of $s \rightarrow b \rightarrow c \rightarrow t$.

(Note that s $\rightarrow$ a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ t is also a valid solution, having the same minimum weight of 2, and you may have found that solution if you iterated through edges in a different order.)

Problem 2

The recurrence for `editDistance` from the problem statement is:

$$S(i,j) = \begin{cases} S(i-1,j-1) & \text{if } A[i] = B[j] \\ \min(S(i-1,j), S(i,j-1), S(i-1,j-1)) + 1 & \text{if } A[i] \neq B[j] \end{cases}$$

Recall the *meaning* of $S(i,j)$: it is the edit distance between the first i characters of the first string A and the first j characters of the second string B. Thus we want to compute $S(A.length, B.length)$, which we do via dynamic programming, filling in all entries $S(i,j)$ for i between 0 and $A.length$ and j between 0 and $B.length$, leading to a 2-dimensional array with dimensions $(A.length + 1) \times (B.length + 1)$

The base cases for our algorithm encode the fact that the edit distance between the empty string and a string of length i (or j) is i (or j respectively):

$$S(i,0) = i, \text{ for } 0 \leq i \leq A.length$$
$$S(0,j) = j, \text{ for } 0 \leq j \leq B.length$$

Here is an example Java implementation of what we have just discussed:

```
public static int editDistance(String s1, String s2){
    int[][] table = new int[s1.length() + 1][s2.length() + 1];
    for (int i = 0; i < s1.length() + 1; i++) {
        table[i][0] = i;
    }
    for (int j = 0; j < s2.length() + 1; j++) {
        table[0][j] = j;
    }
    for (int row = 1; row < s1.length()+1; row++) {
        for (int col = 1; col < s2.length() + 1; col++) {
            if (s1.charAt(row-1) == s2.charAt(col-1)) {
                table[row][col] = table[row-1][col-1];
            } else {
                table[row][col] = 1 + Math.min(table[row-1][col-1],
                                      Math.min(table[row-1][col],
                                               table[row][col-1]));
            }
        }
    }
    return table[s1.length()][s2.length()];
}
```

Problem 3

See Wikipedia!

Problem 4

The following are the longest palindromic subsequences found in the given strings:

1. a → a, of length 1
2. abcacb → bcacb, of length 4
3. dabcacbd → dbcacbd, of length 7
4. youshallnotpass → sallas, of length 6
5. thereandbackagain → aacaa or aakaa, of length 5

Dynamic Programming Algorithm

We describe a dynamic programming approach for computing the length of the longest palindromic subsequence in a string of characters. We will define a recurrence in terms of $T(i, j)$, representing the *length of the longest palindromic subsequence consisting of characters from i to j inclusive.* We now derive a recurrence relation for T.

There are two cases to the recurrence. Either the first and last character of the substring from i to j match, or they do not:

- If they match, then the length of the longest palindromic subsequence from i to j must be two more than the length of the longest palindromic subsequence from $i + 1$ to $j - 1$.
- If they do not match, then no palindromic subsequence can both start with the ith character and end with the jth character, so the longest palindromic subsequence between i and j must be *either* the longest palindromic subsequence between i and $j-1$, or the longest palindromic subsequence between $i + 1$ and j.

Written explicitly, the recursion is:

$$T(i,j) = \begin{cases} T(i+1, j-1) + 2 & \text{if } s[i] = s[j] \\ \max(T(i+1,j), T(i,j-1)) & \text{if } s[i] \neq s[j] \end{cases}$$

Since the recurrence only makes use of entries with greater first index, or smaller second index, we can fill in the table with two nested loops, where one loop decreases the first index and the other loop increases the second index.

It is fairly clear that the following two base cases are enough to define the rest of the values in T:

- Strings of length 1: for all i between 0 and $n-1$, let $T(i,i) = 1$.
- Strings of length 0: for all i between 1 and $n-1$, let $T(i,i-1) = 0$.

The following is the pseudocode for our algorithm, which accepts as input a string s and outputs the length of the longest palindromic subsequence:

```
PALINDROME($s$)
 1  Set $n \leftarrow$ LENGTH($s$)
 2  Create 2-dimensional array $T[n][n]$
 3  for $i \leftarrow 0$ to $n-1$
 4        do T(i,i)=1
 5  for $i \leftarrow 1$ to $n-1$
 6        do T(i,i-1)=0
 7  for $i \leftarrow n-1$ down to 0
 8        do for $j \leftarrow i+1$ to $n-1$
 9                 do if $s(i) = s(j)$
10                       then $T(i,j) \leftarrow T(i+1,j-1)+2$
11                       else  $T(i,j) \leftarrow \max(T(i+1,j), T(i,j-1))$
12  Return $T(0,n-1)$
```

Proof Of Correctness

As we derived the pseudocode above, we introduced all the elements needed for a proof of correctness. We need only assemble the pieces:

We argued above that the recurrence for $T(i,j)$ correctly expresses the length of the longest palindromic subsequence between locations i and j in terms of $T(i+1,j-1)$, $T(i+1,j)$, and $T(i,j-1)$. The order in which we fill the table defines each $T(i,j)$ in terms of values that have already been computed. And finally, the base cases, strings of length 0 and 1, clearly have longest palindromic subsequences of lengths 0 and 1 respectively. Therefore, induction on the number of times line 9 in the pseudocode is executed yields a proof of correctness of the values of the table T, and in particular, guarantees that the answer $T(0,n-1)$ returns the length of the longest palindromic subsequence of the entire string.

Running Time Analysis

We fill half of a table that contains $O(n^2)$ elements. In filling each element, we do constant work. Thus, the runtime of our algorithm is $\boldsymbol{O(n^2)}$.

Problem 5

Given a paragraph of text, we want to find the alignment of words that minimizes our penalty function. Our penalty function is defined as the square of the sum of the distances between the end of the each line and the margin. The last line does not contribute to the penalty, but the last line must not extend beyond the margin. Given a text of n words, the position of the margin m, and our penalty function, we use a dynamic programming algorithm to find the optimal alignment, and a slight amount of post-processing to recover what we need.

Dynamic Programming Algorithm

The "special treatment" of the last line is a bit odd, so we will first construct a dynamic programming algorithm for the version of the problem that ignores this aspect, treating every line the same; we will post-process the result to yield the desired answer to the original problem.

We will define a recurrence relation for $T(j)$ representing *the cost of the optimal way of arranging only words* 0 *to* $j-1$ (with no special treatment of the last line).

Let m denote the width of the margin, and let $len(i)$ denote the number of letters in word i.

We create the table $c(i,j)$ to contain the total number of characters in a line consisting of the words between words i and j inclusive. We note that we can fill in this table in time $O(n^2)$ by, for each i, defining $c(i,i) = len(i)$, and then iteratively defining $c(i,j+1) \leftarrow c(i,j) + 1 + len(j+1)$, meaning that adding word $j+1$ contributes one letter for the space, plus the number of letters in word $j+1$.

Having defined c, we now define the recurrence relation for $T(j)$, representing the fact that the optimal cost for arranging words 0 through $j-1$ equals the minimum, over all possible words i with which to *start* the last line, of the optimum cost of arranging words 0 through $i-1$, plus the cost of putting the words i through $j-1$ on a single line:

$$T(j) = \begin{cases} \min\limits_{i \in \{0,\ldots,j-1\}} \left[T(i) + (m - c(i,j-1))^2\right] & \text{if } j > 0 \\ 0 & \text{if } j = 0 \end{cases}$$

As expressed in the recurrence relation, the base case of the recurrence is $T(0) = 0$, which makes the above recurrence well-defined.

We now have an algorithm to efficiently compute the optimum cost of arranging line breaks, ignoring the special treatment of the last line. However, it is easy to see that, in fact, the optimal way to choose the last line break for the original problem can be found from our tables by choosing the i with minimal $T(i)$ from all i such that $c(i, n-1) \leq m$.

Finally, we note that, along with T, we can construct a table of "back-pointers" P: given that we have decided to insert a line break between words $j-1$ and j, $P(j)$ tells the optimal index before which to end the *previous* line. Evaluating the recurrence for T automatically gives us the values $P(j)$: this is just the value of i at which the expression is minimized.

The following is the pseudocode for our algorithm, which takes as input an array *len* describing the lengths of a sequence of words, along with the margin width m, and outputs b, a set of indices of words before which to insert line breaks:

```
WORDWRAP($len, m$)
  Let $n$ be the number of words, namely the number of entries in $len$
  Create 2-dimensional array $c[n][n]$
  Create arrays $T[n+1]$ and $P[n+1]$
  Initialize empty set $b$
  for $i \leftarrow 0$ to $n-1$
      do $c(i,i) \leftarrow len(i)$
         for $j \leftarrow i+1$ to $n-1$
             do $c(i,j+1) \leftarrow c(i,j) + 1 + len(j+1)$
  Initialize $T(0) \leftarrow 0$ and $P(0) = 0$
  for $j \leftarrow 1$ to $n$
      do $T(j) \leftarrow \min_{i \in \{0,\ldots,j-1\}} \left[T(i) + (m - c(i,j-1))^2\right]$ and $P(j)$ gets the $i$ that minimizes this expression
  Let $w \leftarrow \operatorname{argmin}_{\{i : c(i,n-1) \leq m\}} T(i)$        ▷ "argmin" returns the index which minimizes the expression
  while $w \neq 0$
      do Add $w$ to the list $b$
         $w \leftarrow P(w)$
  Return $b$
```

Proof of Correctness

In the introduction to the algorithm, we argued that $c(i,j)$ correctly stores the length of a line between words i and j, and justified that recursive expression for $T(j)$ correctly represents the optimal cost of arranging words 0 through j (with no special treatment for the last line), with $P(j)$ storing the location of the previous linebreak in an optimal such scheme. Further, we noted that $\operatorname{argmin}_{\{i:c(i,n-1)\leq m\}} T(i)$ computes the optimal location for the start of the last line under the original rules. To recover an optimal set of linebreaks, we iterate back through the table P, starting from this location.

Running Time Analysis

The double "for" loop to initialize c clearly takes $O(n^2)$ time; the min is computed n times, and each time involves $O(n)$ work; the final steps clearly take linear time. This leads to a total time of $O(n^2)$.