Problem Set 9

All parts are due on December 8, 2017 at 11:59PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on alg.csail.mit.edu .

Problem 9-1. [20 points] **Programming Jumble**

Gill Bates wrote generic Python pseudocode to solve dynamic programming problems, using both top-down and bottom-up approaches. Unfortunately, the lines of his code got jumbled, and he needs your help to reconstruct his functions. Below are his two Python function definitions, as well as some lines of additional Python code. Help Gill by indicating the sequence of lines that completes each function. It may help to write out each function line by line, but a correct answer will simply be a sequence numbers corresponding to the lines that complete each function.

```
 1.  else:
 2.  j = i
 3.  n = i
 4.  i = j
 5.  i = n
 6.  return memo[n]
 7.  for i in range(n):
 8.  if i in base_cases:
 9.  top_down_dp(i, memo)
10.  top_down_dp(j, memo)
11.  for i in dependencies(j):
12.  for j in dependencies(i):
13.  memo[i] = solve_base_case(i)
14.  memo[i] = solve_from_subproblem_memo(i, memo)
```

Each line of code above may be used in both, only one, or neither function, and each function should be completed using only the lines provided (do not write additional code).

(a) [10 points]

```
def top_down_dp(n, memo = {}):
    # compute subproblems top down
```

(b) [10 points]

```
def bottom_up_dp(n, memo = {}):
    # compute subproblems bottom up
```

Solution:

```
def top_down_dp(n, memo = {}):
    # compute subproblems top down
5   i = n
8   if i in base_cases:
13      memo[i] = solve_base_case(i)
1   else:
12      for j in dependencies(i):
10          top_down_dp(j, memo)
14      memo[i] = solve_from_subproblem_memo(i, memo)
6   return memo[n]

def bottom_up_dp(n, memo = {}):
    # compute subproblems bottom up
7   for i in range(n):
8       if i in base_cases:
13          memo[i] = solve_base_case(i)
1       else:
14          memo[i] = solve_from_subproblem_memo(i, memo)
5   i = n
14  memo[i] = solve_from_subproblem_memo(i, memo)
6   return memo[n]
```

Here we note that Gill's code is correct, but his top-down function does not run in polynomial time. Because he never checks whether the answer to a subproblem has already been recorded in the memo, his top-down function makes an exponential number of recursive calls (assuming that `dependencies(i)` consists of more than one subproblem). However, since no line of code is provided to check for inclusion in the memo, full points will be given to any implementation of the top-down function that is correct but inefficient. Gill apologies for his inefficiencies.

Here is efficient top-down pseudocode, adding one additional line that was not provided. If you addressed the fact that an efficient top-down function does not exist, +2 points extra credit.

```
def top_down_dp(n, memo = {}):
    # compute subproblems top down
5   i = n
X   if i not in memo:
8       if i in base_cases:
13          memo[i] = solve_base_case(i)
1       else:
12          for j in dependencies(i):
10              top_down_dp(j, memo)
14          memo[i] = solve_from_subproblem_memo(i, memo)
6   return memo[n]
```

Rubric:

- 10 points for correct line orderings
- -2 points per mistake
- +2 points for catching Gill's error

Problem 9-2. [40 points] **Coding Practice**

Alice wants to encode the message `yummybanana` using a small number of bits. Currently each letter is encoded using ASCII[1], a fixed-width coding scheme mapping characters to combinations of 7-bits. As such, Alice's message encoded in ASCII requires $7 \times 11 = 77$ bits to store. Since there are only six types of letters in her message, $\mathcal{X} = \{\texttt{a}, \texttt{b}, \texttt{m}, \texttt{n}, \texttt{u}, \texttt{y}\}$, she thinks she should be able to construct a lossless binary code to represent each letter using no more than 3 bits.

(a) [15 points] The Kraft inequality states that if character $x_i \in \mathcal{X}$ is represented using l_i bits in a uniquely decodable code, then $\sum_{x_i \in \mathcal{X}} 2^{-l_i} \leq 1$. Show there exists an assignment of bit lengths to letters in $\mathcal{X}$ that satisfies the Kraft inequality such that no codeword uses more than 3 bits. For such a code, what is the maximum number of characters that can be assigned codewords using either one or two bits respectively?

Solution:

The sum in the Kraft inequality decreases as encoded lengths increase. The Kraft inequality holds if we simply assign all characters to 3 bit strings, because $6 \times 2^{-3} = 6/8 < 1$. If any character is assigned 1 bit, the Kraft inequality will not hold even if all other characters are encoded using 3 bits, i.e. $2^{-1} + 5 \times 2^{-3} = 9/8 > 1$. So no valid encoding can use a single bit for any character. Alternatively, two characters can be encoded using 2 bits, which will bring Kraft to equality, i.e. $2 \times 2^{-2} + 4 \times 2^{-3} = 1$.

Rubric:

- 5 points for assignment of bit lengths satisfying Kraft
- 5 points for number of characters that can be assigned 1 bit codewords
- 5 points for number of characters that can be assigned 2 bit codewords
- Partial credit may be awarded

(b) [10 points] Give a prefix-free code mapping each character in $\mathcal{X}$ to a codeword represented using exactly 3 bits. Draw a prefix tree associated with your code, with zeros branching left and ones branching right.

Solution: Any tree where all characters are leaves at depth three is valid. For example:

```
       _______0|1_______
   ___0|1___          ___0|
 _0|1_   _0|1_   _0|1_
 a   b   m   n   u   y
000 001 010 011 100 101
```

Rubric:

- 5 points for code
- 5 points for drawing of tree

[1]`https://en.wikipedia.org/wiki/ASCII`

- Partial credit may be awarded

(c) [15 points] Your code from part (b) stores Alice's message using only $3 \times 11 = 33$ bits, a significant improvement! However, Alice heard that she might be able to compress her message even further by taking into account how often each character appears in her message. For each character in $\mathcal{X}$, calculate the frequency it appears in the string, i.e. (# occurrences / length of string). Then construct a binary Huffman code on the characters $\mathcal{X}$ using your frequencies, and draw its prefix tree. How many bits are needed to encode Alice's message using your Huffman code?

Solution:

Characters b and u each occur once with frequency $1/11$, character a occurs three times with frequency $3/11$, and the remaining characters each occur twice with frequency $2/11$. A Huffman code will repeatedly link the lowest frequency characters together, e.g. `(b,u)`, then `(m, (b,u))`, then `(n,y)`, then `(a, (m, (b,u)))`, then `((a, (m, (b,u))), (n,y))`. Two Huffman prefix tree types are equally valid, one using at most 3 bits for each character, the other using at most 4. Both codes encode Alice's message using 28 bits.

```
       _______0|1_________                  ______ 0|1_______
    __0|1___              _0|1_          __0|1__         __0|1__
    a    ____0|1__        n    y         a    _0|1_   _0|1_   m
   00   m      _0|1_     10   11        00    n    y  b    u  11
       010    b    u                         001 010 011 100
           0100 0101
```

Rubric:

- 4 points for character frequencies
- 4 points for Huffman code
- 4 points for drawing of tree
- 3 points for encoded length
- Partial credit may be awarded

Problem 9-3. [20 points] **Expensive Ones**

Consider a source sequence of characters from a set $\mathcal{X}$, with each character $x \in \mathcal{X}$ occuring with probability $p(x)$.

(a) [10 points] Describe an algorithm to construct a binary prefix-free code that minimizes the expected number of ones in its encoding of the source; the bit lengths of codewords are not constrained.

Solution:

If we are not constrained by the number of bits per code word, it is possible to assign each character a codeword with at most one 1. By the prefix property, at most one codeword can contain only zeros, so an optimal code will assign a codeword with no ones to the most probable character, and a codeword with a single one to every other character. Let x be the most character that occurs with the highest probability, and assign each of the other characters in $x_i \in \mathcal{X} \setminus x$ to a unique integer from $a_i \in [0, |\mathcal{X}| - 2]$. Then assign to x a codeword containing $|\mathcal{X}| - 1$ zeros, and to each x_i the codeword containing a_i zeros followed by a one. The prefix tree for this code is a caterpillar with $|\mathcal{X} - 1|$ right leaves and one left leaf. This algorithm runs in linear time $O(|\mathcal{X}|)$.

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

(b) [10 points] Huffman's algorithm constructs an *optimal* binary prefix-free code, that minimizes the expected number of bits to encode the source. Modify Huffman's algorithm to return a code from among all optimal prefix-free codes that minimizes the expected number of ones in its encoding of the source.

Solution:

As in the previous algorithm, we will want to maximize left leaves over right leaves as much as possible. In Huffman, we take the two least probable characters (or recursively combined sets of character) and connect them to a common branch node in some order. We now specify this order when the probabilities of the two characters are not equal, connecting the more probable character as a left child, and the less probable character as a right child. This modification will find from among all prefix codes constructable by the Huffman algorithm, the code containing the fewest number of ones.

Note that the Huffman algorithm cannot produce all codes of optimal length. For example, if a source contains as, bs, cs, and ds with probabilities $0.27, 0.26, 0.24, 0.23$ respectively, any optimal binary code will assign a two bit codeword to every letter. Any Huffman code will pair c and d as children of the same node, while the following is an optimal length code that is not reachable by Huffman.

```
  ___0|1___
_0|1_   _0|1_
a   c   b   d
```

However, any optimal length code that minimizes the number of 1s is a Huffman code. To prove this, we first prove that in some code that minimizes the number of 1s (where 0s branch left and 1s branch right), leaf characters at the same depth (i.e. with the same length codeword) appear from left to right in descending order of their probabilities. To show this, consider an optimal code that minimizes the number of 1s. Characters from left to right at the same depth in the prefix tree weakly monotonically increase in the number of 1s contained in their codeword. Of two characters at the same depth, if the left one has higher probability than the other, they must have the same number of 1s in their codewords, or else swapping them would decrease the number of 1s in the encoding, contradicting optimality. So we can swap them without affecting the number of 1s in the encoding. Repeating this process results in an optimal code that minimizes the number of 1s and has characters at the same depth appearing in descending order of probabilities.

Further, the two least probable characters occur at the lowest depth in any optimal code; if they did not, swapping a character at the lowest depth for one of them would decrease the length of the encoding, contradicting optimality. Thus in some optimal code that minimizes the number of 1s, two least probable characters are at the lowest depth and appear as the two rightmost characters, so it is constructable by Huffman. This proof is a bit beyond the scope of this class, so we will give most of the points for describing a correct modification of Huffman without a full proof of correctness.

Rubric:

- 10 points for a correct algorithm
- Up to 9 points if no proof of correctness
- Partial credit may be awarded

Problem 9-4. [20 points] **Dependent Sources**

Consider the following five binary sources, each providing a temporally independent bit to you every second:

- X_1 is always 0;
- X_2 is randomly distributed, with a 1 occurring with probability 0.2;
- X_3 is randomly distributed, with a 1 occurring with probability 0.3;
- X_4 is the symbol-wise sum of X_1, X_2, and X_3 modulo 2; and
- X_5 is symbol-wise equal to either X_1 or X_2, each with probability 0.5.

You would like to compress one or more of these sources for future lossless recovery. Because some of these sources are not mutually independent, coding two of them together may provide higher compression than coding them separately, and Slepian-Wolf can be used to jointly encode them efficiently.

Recall that the **entropy** of a Bernoulli random variable X with $\Pr\{X = 1\} = p$ is given by $H(X) = -p\log_2 p - (1-p)\log_2(1-p)$, and that joint entropy is given by

$$H(A,B) = -\sum_{a\in A}\sum_{b\in B}\Pr\{A=a, B=b\}\log_2 \Pr\{A=a, B=b\} = H(A|B) + H(B).$$

(a) [5 points] State the individual entropy $H(X_i)$ for each source, for $i \in [1,5]$.

Solution:

- $H(X_1) = \lim_{p\to 0}(-p\log_2 p - (1-p)\log_2(1-p)) = 0$
- $H(X_2) = -0.2\log_2 0.2 - (1-0.2)\log_2(1-0.2) = \log_2 \frac{5}{4^{0.8}} \approx 0.7219$
- $H(X_3) = -0.3\log_2 0.3 - (1-0.3)\log_2(1-0.3) = \log_2 \frac{10}{3^{0.3}7^{0.7}} \approx 0.8813$
- $H(X_4)$ can be thought of as a Bernoulli random variable with probability of a one equal to the probability of a one in exactly one of X_2 or X_3, i.e. $p = 0.2 * 0.7 + 0.3 * 0.8 = 0.38$, so

$$H(X_4) = -0.38\log_2 0.38 - (1-0.38)\log_2(1-0.38) = \log_2 \frac{50}{19^{0.38}13^{0.62}} \approx 0.9580$$

- $H(X_5)$ can be thought of as a Bernoulli random variable with probability of a one equal to the average probabilities in X_1 and X_2, i.e. $p = 0.1$, so

$$H(X_5) = -0.1\log_2 0.1 - (1-0.1)\log_2(1-0.1) = \log_2 \frac{10}{9^{0.9}} \approx 0.4690$$

Rubric:

- 1 point for each entropy calculation

(b) [15 points] State the joint entropy $H(X_i, \ldots, X_j)$ for the following five sets of sources, in terms of the individual source entropies $H(X_i)$. Which sets can theoretically be encoded at a lower bit-rate when encoded jointly rather than separately?

$$\{X_1, X_2\}, \{X_2, X_3\}, \{X_2, X_5\}, \{X_3, X_4\}, \{X_3, X_4, X_5\}$$

Solution:

$H(X_1, X_2) = H(X_1) + H(X_2|X_1) = H(X_1) + H(X_2) = H(X_2)$, same bit-rate

$H(X_2, X_3) = H(X_2) + H(X_3|X_2) = H(X_2) + H(X_3)$, same bit-rate

$$\begin{aligned}
H(X_2, X_5) =& -\sum_{i=0}^{1}\sum_{j=0}^{1} \Pr\{X_2 = i, X_5 = j\} \log_2 \Pr\{X_2 = i, X_5 = j\} \\
=& -\sum_{i=0}^{1}\sum_{j=0}^{1} \Pr\{X_5 = j|X_2 = i\} \Pr\{X_2 = i\} \log_2(\Pr\{X_5 = j|X_2 = i\} \Pr\{X_2 = i\}) \\
=& -\sum_{i=0}^{1} \Pr\{X_2 = i\} \log_2 \Pr\{X_2 = i\} \sum_{j=0}^{1} \Pr\{X_5 = j|X_2 = i\} \\
& -\sum_{i=0}^{1} \Pr\{X_2 = i\} \sum_{j=0}^{1} \Pr\{X_5 = j|X_2 = i\} \log_2 \Pr\{X_5 = j|X_2 = i\} \\
=& H(X_2) - \sum_{i=0}^{1} \Pr\{X_2 = i\} \sum_{j=0}^{1} \Pr\{X_5 = j|X_2 = i\} \log_2 \Pr\{X_5 = j|X_2 = i\} \\
=& H(X_2) - (0.8(1)\log_2 1 + 0.8(0)\log_2 0 + 0.2(0.5)\log_2 0.5 + 0.2(0.5)\log_2 0.5) \\
=& H(X_2) + 0.2, \text{ lower bit-rate than } H(X_2) + H(X_5)
\end{aligned}$$

$H(X_3, X_4) = H(X_3) + H(X_4|X_3) = H(X_3) + H(X_2)$, lower bit-rate than $H(X_3) + H(X_4)$

$$\begin{aligned}
H(X_3, X_4, X_5) = & H(X_3) + H(X_4|X_3) + H(X_5|X_3, X_4) \\
=& H(X_3) + H(X_2) + H(X_5|X_2) = H(X_3) + H(X_2, X_5) \\
=& H(X_3) + H(X_2) + 0.2, \text{ lower bit-rate than } H(X_3) + H(X_4) + H(X_5)
\end{aligned}$$

Rubric:

- 3 points for each entropy calculation
- Partial credit may be awarded