Problem 1)

| Symbol | Frequency |
| --- | --- |
| A | 0.35 |
| B | 0.2 |
| C | 0.15 |
| D | 0.15 |
| E | 0.08 |
| F | 0.07 |



First Huffman Code



Second Huffman Code

Expected bits per symbol for First Huffman Code

* A = 1 bits | B = 3 bits | C = 3 bits | D = 3 bits | E = 4 bits | F = 4 bits
* 1\*0.35 + 3\*0.2 + 3\*0.15 + 3\*0.15 + 4\*0.08 + 4\*0.07
* 2.45 bits per symbol on average

Expected bits per symbol for Second Huffman Code

* A = 2 bits | B = 2 bits | C = 3 bits | D = 3 bits | E = 3 bits | F = 3 bits
* 2\*0.35 + 2\*0.2 + 3\*0.15 + 3\*0.15 + 3\*0.08 + 3\*0.07
* 2.45 bits per symbol on average

Problem 2)

Greedy Strategy 1

* Consider (1, 2), (1, 10), (11, 13) for S = 0 and F = 13
* This algorithm will select (1, 2), since it finishes the earliest
* It will discard (1, 10) because that overlaps with (1, 2)
* So its gap intervals will be (0, 1) and (2, 11), or a total gap time of 10
* The optimal solution is to select (1, 10) and (11, 13)
* This leaves gap intervals (0, 1) and (10, 11), or a total gap time of 2

Greedy Strategy 2

* Consider (1, 3), (2, 10), (11, 13) for S = 0 and F = 13
* This algorithm will select (1, 3), since it starts the earliest
* It will discard (2, 10) because that overlaps with (1, 3)
* So its gap intervals will be (0, 1) and (3, 11), or a total gap time of 9
* The optimal solution is to select (2, 10) and (11, 13)
* This leaves gap intervals (0, 2) and (10, 11), or a total gap time of 3

Greedy Strategy 3

* Consider (1, 3), (3, 5), (4, 13) for S = 0 and F = 13
* This algorithm will select (1, 3) and (3, 5), since these have the smallest gap between them (a gap time of 0)
* (4, 13) is discarded since it overlaps with (3, 5)
* There is no need for further recursive calls, as all intervals have been either selected or discarded
* Its gap intervals will be (0, 1) and (5, 13), or a total gap time of 9
* The optimal solution is to select (1, 3) and (4, 13)
* This leaves gap intervals (0, 1) and (3, 4), or a total gap time of 2

Problem 3)

My algorithm exploits the fact that, given three evenly-spaced collinear points, the second point is the midpoint of the other two. The actual algorithm itself is pretty simple, therefore, as it simply calculates the midpoint between each pair of points and checks whether that midpoint is a point on the graph. If so, it’s a triple.

The first step of the algorithm is storing the input points in a list of tuples in the form [(x1, y1), (x2, y2), … (xn, yn)]. Those points are then sorted in ascending order first by x-coordinate and then by y-coordinate. For instance, the list [(6, 4), (6, 3), (1, 8)] would become [(1, 8), (6, 3), (6, 4)]. The algorithm then constructs a dictionary with x-coordinates as keys and tuples of the form (start, end) where start is the starting index of that x-coordinate in the points list and end is the ending index of that x-coordinate in the points list. For instance, the x-coordinate dictionary for [(1, 8), (6, 3), (6, 4)] would look like {1: (0, 0), 6: (1, 2)}. The fact that this is a dictionary isn’t important running-time wise, it’s just convenient for code readability.

The next step of the algorithm is to loop through every pair of points (so O(n2)) and calculate their midpoint. Then, it uses an adjusted binary search to search the original list of points to check if that midpoint is in there. If it is, the “triple\_exists” variable is set to True, the loop breaks, and the program prints “YES”. If the loop gets to the end without finding a midpoint in the list, the program prints “NO”.

The adjusted binary search iterates logarithmically through the list of points based on x-coordinate like normal until it finds an x-coordinate that matches the x-coordinate in the midpoint. Then, it performs binary search again on the y-coordinates that are paired with the found x-coordinate, using the x-coordinate dictionary to set the high and low based on where every coordinate with the matching x-coordinate exists in the list of points. Overall, this still runs in O(log(n)) time, even though it is performing binary search twice. In the absolute worst case, both sections of the adjusted binary search take O(log(n)), which means the whole thing takes O(2log(n)), which is the same as O(log(n)).

Building the list of points runs in O(n) time, sorting that list runs in O(nlog(n)) time, building the x-coordinate dictionary runs in O(n) time, and looping through every pair of coordinates and performing binary search for each pair runs in O(n2log(n)) time. This makes the whole algorithm O(n + nlog(n) + n + n2log(n)), which is just O(n2log(n)).

Problem 4)

Precise verbal description of the meaning of the dynamic programming array:

* The “decodings” list is the dynamic programming array in my program. The ith element in the list holds the number of decodings from i bits and back, which is calculated starting with the previous element (element i - 1). If the previous 2 characters combined form a double bit encoding (either 10 or 01), then add element i - 2 to the current decoding. If the previous 3 characters combined form a triple bit encoding (either 111 or 011), then add element i - 3 to the current decoding.
* The value of element 0 is 1 at the start of the program, since there is exactly 1 decoding by default for the first bit, whether a 0 or a 1. Every time a new decoding can be taken from a next bit or sequence of bits, this recursively increases the total number of decodings starting with the previous element and going backwards. That is what the i - 1 element represents in the dynamic programming array: the result of that recursive counting back to element 0. For a double bit encoding, it begins at i - 2 because there’s 2 characters, and for a triple bit encoding, it begins at i - 3 because there’s 3 characters. All of these get added together because they’re all unique decodings.

A mathematical formula that describes how to compute the value of each cell of the array:



Return value of the algorithm

* The total number of unique character strings the given bit string could be decoded into