Earlier in the semester, we introduced Huffman's algorithm which finds an optimal binary prefix-free code for a set of n symbols $s_1, \ldots, s_n$ which occur $o_1, \ldots, o_n$ times respectively in some document. The procedure creates a set of trees – one for each symbol – and initializes the weight w_i of each tree to o_i. It then selects two trees with minimum weight, and replaces them with a new tree composed of the subtrees. This new tree has weight equal to the sum of weights of its subtrees. This procedure is repeated until only a single tree remains. The resulting tree can be interpreted as a description of a set of codewords for the symbols used.

The cost of a Huffman tree is equivalent to the size of the document when it's encoded using the code defined by the tree. If symbol s_i appears o_i times and s_i is a leaf at depth d_i in the tree then each occurrence of s_i in the document is encoded with d_i bits. As a result, s_i contributes cost $o_i \times d_i$ to the total cost of the encoded document. In general the total cost of the encoded document (and hence the tree) is

$$o_1 \times d_1 + o_2 \times d_2 \cdots + o_n \times d_n \tag{1}$$

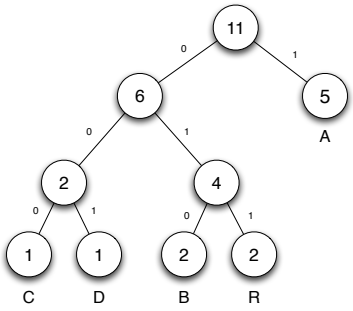
For example, consider the following document (of length 11) composed of 5 symbols:

ABRACADABRA

The table below lists each symbol and its number of occurrences.

Symbol	A	B	C	D	R
Occurrences	5	2	1	1	2

Running Huffman's algorithm on the table might produce the following Huffman tree:



Encoding the document according to the code defined by the Huffman tree yields

10100111000100110100111

which has a length of 23. In other words, the cost of the tree is 23. We can verify this is indeed the cost of the code by noting that in the tree, A has depth 1 and the remaining symbols have depth 3. Hence, using just the tree, we can also arrive at the cost by using Equation 1.

$$1 \times 3 + 1 \times 3 + 2 \times 3 + 2 \times 3 + 5 \times 1 = 3 + 3 + 6 + 6 + 5 = 23$$

Surprisingly, one does not need to actually create a tree to compute the cost of a Huffman code. To see how, imagine that we are creating the Huffman tree shown above. Initially, our running total is $T = 0$. Every time we merge two trees, the codewords for the symbols in those trees increase by 1 bit. Our procedure works by adding in this additional codeword cost at the time of a merge.

- We begin by merging C and D. We now know that each occurrence of C and D will be replaced by codewords with length at least 1, so each occurrence of C and each occurrence of D will contribute cost at least 1 to the overall cost. Since C and D each occur only once, $T = T + 2 = 0 + 2 = 2$.
- Next we merge B and R. Since B and R will be replaced by codewords with length at least 1, each occurrence of B and each occurrence of R will contribute cost at least 1 to the overall cost. Since B and R each occur twice they will contribute an additional cost of $2 + 2 = 4$ to T. Hence, $T = 6$.

- Next we merge the CD tree with the BR tree. The CD tree has weight 2. This means C and D occur 2 times collectively in the document. The BR tree has weight 4. This means B and R occur 4 times collectively in the document. Since we are merging these two nodes, we know the codewords for C and D will be 1 bit longer in the document. The same is true for B and R. Since C and D occur twice collectively, this merge adds a cost of 2 to the total. Similarly, since B and R occur 4 times collectively, the merge adds a cost of 4 to the total. Hence the total is now $T = T + 2 + 4 = 6 + 2 + 4 = 12$.
- Finally, we merge the CDBR node with the A node. Each occurrence of A will be encoded with a codeword of length 1 and the symbols in the CDBR node each require an additional bit in their keywords. Since A occurs 5 times and C,D,B, and R occur 6 times collectively, our total is $T = T + 5 + 6 = 12 + 5 + 6 = 23$.

Note that while we discussed computing the running total in terms of tree merges, we don't actually need to create or merge any trees; we just need to find the two smallest weights and replace them with their sum in the current list of weights. This means each merge operation is really just a sum operation. Each sum operation has an associated update to the running total so it makes sense to talk about the value of the running total after each sum operation.

The series of steps that would be performed when applying this tree-less version of the algorithm to the occurrence counts for the ABRACADABRA example are shown below. In each step, the value in the list of occurrence counts that has been produced by merging two of the smallest occurrence counts in the previous list is shown in bold face. It is this value that is added to "Running total" at each step.

occurrences = | 5 | 2 | 1 | 1 | 2 |	Running total = 0
occurrences = | 5 | 2 | **2** | 2 |	Running total = 2
occurrences = | 5 | **4** | 2 |	Running total = 6
occurrences = | 5 | **6** |	Running total = 12
occurrences = | **11** |	Running total = 23

Question 1. Consider a document using 8 symbols whose occurrence counts are shown in the following table:

Symbol	A	B	C	D	E	F	G	H
Occurrences	2	5	20	7	3	2	24	2

Fill in the cells in the tables provided below to show how the algorithm described above would unfold when applied to a message with these occurrence counts. To help you know whether you did this correctly, we have filled in the final values.

occurrences = | 2 | 5 | 20 | 7 | 3 | 2 | 24 | 2 |	Running total = 0
occurrences = | | | | | | | |	Running total =
occurrences = | | | | | | |	Running total =
occurrences = | | | | | |	Running total =
occurrences = | | | | |	Running total =
occurrences = | | | |	Running total =
occurrences = | | |	Running total =
occurrences = | |	Running total =
occurrences = | **65** |	Running total = 157