Solutions for Homework Assignment #3

Answer to Question 1.

In what follows, if T is a tree constructed during the execution of Huffman's algorithm, $f(T)$ denotes the sum of the frequencies of the leaves of T.

a. An example is $\Gamma = \{A, B, C, D\}$, with $f(A) = 2/5$ and $f(B) = f(C) = f(D) = 1/5$. One possible execution of Huffman's algorithm is to

- first create a tree T_1 containing B and C, with frequency $f(T_1) = 2/5$;
- then create a tree T_2 containing A and D, with frequency $f(T_2) = 3/5$; and
- finally create a tree T with subtrees T_1 and T_2, with frequency $f(T) = 1$.

In the resulting tree, all codewords have length 2. Note that there is another possible execution of the algorithm resulting in a tree where A has a codeword of length 1.

b. Assume, for contradiction, that some symbol, say A, has frequency greater than $2/5$, yet Huffman's algorithm constructs a tree in which no codeword has length 1. Consider the step t when A is first merged with another tree T. (It is possible that T consists of a single node.) Let T_1 be the tree that results from merging T and A.

How many trees other than T_1 are there after step t? There must be at least one such tree, say T_2: otherwise, t would be the last step of Huffman's algorithm and the codeword of A would have length 1. In every step we merge two trees of minimum frequency. Since one of the trees involved in step t was A, $f(T_2) \geq f(A) > 2/5$. If after step t there was also a third tree T_3 (in addition to T_1 and T_2), we would have $f(T_3) > 2/5$ for exactly the same reason why $f(T_2) > 2/5$. But then $f(T_1) + f(T_2) + f(T_3) > 6/5$. This is impossible since at the end of each step, the frequencies of all trees must add up to 1.

Thus, after step t there are exactly two trees left, T_1 and T_2. From this we conclude two facts:

(i) $f(T_1)+f(T_2) = 1$. Since T_1 resulted from merging T and A, we have $f(T) = 1-\big(f(A)+f(T_2)\big) < 1/5$. Thus, at the start of step t there is a tree, namely T, of frequency less than $1/5$.

(ii) The algorithm ends after step $t+1$, producing a single tree by merging T_1 and T_2. This implies that T_2 does not consist of a single node: otherwise, the algorithm would produce a codeword of length 1. So, T_2 has two subtrees, say T_{21} and T_{22}.

Consider now the step $t' < t$ in which T_2 was formed, by merging T_{21} and T_{22}. During that step there were certainly trees with frequency less than $1/5$. This is because as we just saw, at the start of a later step, namely t, there is still a tree, namely T, such that $f(T) < 1/5$. Thus, either T itself or a tree rooted at some node of T, of frequency even less than that of T, is available at step t'.

Since at each step we merge two trees of minimum frequency, and in step t' we merged T_{21} and T_{22}, it follows that the frequency of each of these trees is less than $1/5$. But then $f(T_2) = f(T_{21}) + f(T_{22}) < 2/5$, contradicting that, as previously shown, $f(T_2) > 2/5$.

Answer to Question 2.

a. The suggested algorithm proceeds in $k-1$ stages. In stage 1 it merges two piles of length n each, producing a pile of length $2n$. In stage i, $1 \leq i < k$, it merges the pile produced in stage $i-1$ (of length $i \cdot n$) and a pile of length n, producing a pile of length $(i+1)n$. Thus, for all i such that $1 \leq i < k$, stage i requires time proportional to $(i+1)n$. The overall running time (over all $k-1$ stages) is therefore $\Theta\big(\sum_{i=1}^{k-1} n(i+1)\big) = \Theta(nk^2)$.

b. A better algorithm is to (a) divide the k piles into two groups, each consisting of (about) $k/2$ lists; (b) recursively merge the piles in each group; and (c) merge the resulting two piles (each of size $kn/2$ into a single list. The recursion continues until $k = 1$, in which it simply returns the input list (there is nothing to merge).

The running time of this algorithm is therefore described by the following recurrence:

$$T(k, n) = \begin{cases} 2T(k/2, n) + kn, & \text{if } k > 1 \\ 1, & \text{if } k = 1 \end{cases}$$

(Here we are assuming, for simplicity, that k is a power of 2. If not, the general term of the recurrence becomes $T(k, n) = T(\lceil k/2 \rceil, n) + T(\lfloor k/2 \rceil, n) + kn$, and the solution is within a constant factor of the solution for the special case.) Unwinding the recurrence using standard techniques (e.g., from CSCB36) yields $T(k, n) = \Theta(nk \log k)$, which is better than $\Theta(nk^2)$.

Answer to Question 3.

Suppose we sort the list of given intervals by their left endpoint. Let the sorted intervals be $A = I_1, I_2, \ldots, I_n$, and let A_L be the sublist of A consisting of the first $\lfloor n/2 \rfloor$ intervals, and A_R be the sublist of A consisting of the remaining intervals.

Let $I = (\ell, r)$ and $I' = (\ell', r')$ be two intervals in A with longest intersection; without loss of generality, assume $\ell \leq \ell'$. There are three possibilities: (1) both I and I' are on A_L; (2) both I and I' are on A_R; and (3) I is on A_L and I' is on A_R. In case (3) we can assume, without loss of generality, that I is any interval in A_L with maximum right endpoint. For, suppose I is not an interval in L with maximum right endpoint, and let $\hat{I} = (\hat{\ell}, \hat{r})$ be any interval in A_L with maximum right endpoint. Then we have $\max(\hat{\ell}, \ell') = \max(\ell, \ell') = \ell'$ (since, by assumption, I and $\hat{I}$ are in A_L and I' is in A_R); and $\min(\hat{r}, r') \geq \min(r, r')$ (since, by assumption, $\hat{I}$ has a greater right endpoint than I). Thus the intersection of $\hat{I}$ and I' is at least as long as the intersection of I and $\hat{I}$, and so, by definition of I and I', the intersection of $\hat{I}$ and I' is the longest intersection between any two distinct intervals.

This observation immediately leads to the following divide-and-conquer algorithm: We recursively find the pair of intervals I_L and I'_L on A_L with the longest intersection, and the pair of intervals I_R and I'_R on A_R with the longest intersection. We then scan the $\lfloor n/2 \rfloor$ intervals on A_L to find an interval I among them with the maximum right endpoint. This clearly takes $O(n)$ time. Next we scan the $\lceil n/2 \rceil$ intervals on A_R to find an interval I' among them with the longest intersection with I. This takes $O(n)$ time, since computing the length of the intersection of two intervals can be done in $O(1)$ time. Finally, we return whichever of the three pairs of intervals (I_L, I'_L), (I_R, I'_R), and (I, I') has the longest intersection. This is shown in pseudocode in Figure 1. The algorithm takes as input a list A of $n \geq 1$ nonempty intervals, sorted by their left endpoints. It returns the pair $(\varnothing, \varnothing)$ if $n = 1$, and a pair (I, I') of intervals in A with the longest intersection if $n > 1$. If I and I' are intervals, $|I \cap I'|$ denotes the length of the intersection of I and I'; clearly this can be computed in constant time.

To find the pair of intervals in a list A of $n > 2$ intervals given in arbitrary order, we first sort the intervals in A by their left endpoint, and then we return the pair returned by LONGESTINTERSECTION(A). The correctness of the algorithm follows from the observation above.

The running time of LONGESTINTERSECTION(A), where A is a list of length n, is described by the following recurrence, assuming n is a power of 2:

$$T(n) = \begin{cases} 2T(n/2) + cn, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

where c is some constant. Using the Master Theorem, we observe that in our case we have $a = b = 2$ and $d = 1$, so $a = b^d$, and the solution of the recurrence is $\Theta(n^d \log n) = \Theta(n \log n)$.

The algorithm to find a pair of intervals with the longest intersection among n intervals given in arbitrary order consists of sorting the given intervals followed by a call to LONGESTINTERSECTION. Since sorting can be done in $O(n \log n)$ time, the entire algorithm takes $O(n \log n) + O(n \log n) = O(n \log n)$ time, as wanted.

LongestIntersection(A)

▷ the nonempty list of intervals A is sorted by left endpoint
if A consists of only one interval **then return** $(\varnothing, \varnothing)$
else
  $A_L :=$ first half of A
  $A_R :=$ second half of A
  $(I_L, I'_L) :=$ LongestIntersection(A_L)
  $(I_R, I'_R) :=$ LongestIntersection(A_R)
  $I :=$ interval in A_L with maximum right endpoint
  $I' :=$ interval in A_R such that $|I \cap I'|$ is maximum
  ▷ return the pair among (I_L, I'_L), (I_R, I'_R), and (I, I') with longest intersection
  if $|I_L \cap I'_L| > |I \cap I'|$ **then** $(I, I') := (I_L, I'_L)$
  if $|I_R \cap I'_R| > |I \cap I'|$ **then** $(I, I') := (I_R, I'_R)$
  return (I, I')

Figure 1: Pseudocode for divide-and-conquer algorithm to find intervals with longest intersection