COMP 3711H – Fall 2016
Tutorial 5 – Sketch Solutions

Note. The answers given below might not be complete. Some are sketches or hints, sometimes missing details.

1. Huffman Coding
   From the book Problems on Algorithms, *by Ian Parberry, Prentice-Hall, 1995.*
   Build a Huffman Tree on the frequencies $\{1, 3, 5, 7, 9, 11, 13\}$.
   For this problem, follow the rule that if two items are combined in a merge, the smaller one goes to the left subtree (in case of ties *within* a merge you can arbitrarily decide which goes on the left).
   Are there any *ties* in the Huffman Construction process, i.e., are there times when the merge procedure can choose between different choices of items?
   How many *different* Huffman Trees can be built on this frequency set?

   There is one possible tie. There is a time when the smallest items are $7, 9$ *and there are two possible* 9*s to choose from.*

   That is the only *time when the algorithm can make an arbitary choice, so only two different Huffman codes can be built off of this frequency set.*

2. A Huffman Coding Variant

   (a) Recall that in each step of Huffman's algorithm, we merge two trees with the lowest frequencies. Show that the frequencies of the two trees merged in the ith step are at least as large as the frequencies of the trees merged in any previous step.

   We will show that the frequencies merged at the $(i+1)$*st step are at least as large as the frequencies merged at the* i*'th step. The proof will folllow.*

   Let $a \leq b \leq c \leq d$ *be the values of the four smallest frequencies in the priority queue immediately before the start of the* i*th step. The* i*th step merges* a, b *and creates a new frequency* $z = a + b$ *that is inserted into the priority queue.*

   In the $(i+1)$*st step, only two possibilities can occur*

   i. $z \geq d$: *In this case the two frequencies merged are values* c, d *and the statement is trivially correct.*

   ii. $z < d$: *In this case, the two values merged are* c, z *and the statement is still correct.*

   (b) Suppose that you are given the n input characters, already sorted according to their frequencies. Show how you can now construct the Huffman code in $O(n)$ time. (*Hint:* You need to make clever use of the property given in part (a). Instead of using a priority queue, you will find it advantageous to use a simpler data structure.)

   From part (a) we know that the new frequencies created by the Huffman algorithm are created in non-decreasing order.

   The algorithm is as follows:

   - *Create two* queues *(not priority queues). Recall that a queue is a linked list that permits checking and/or removing its "head" item in* $O(1)$ *time and adding an item to its "tail" in* $O(1)$ *time.*

- *Add the original items in sorted order to the 1st queue in $O(n)$ time. Note that removing items from the head of this queue, one at a time, will remove them in sorted order.*
  In the algorithm, we will only remove items from this queue but never add new ones.
- *Every time a new frequency is created, add it to the tail of the 2nd queue in $O(1)$ time. Note that, from (a), we know that the items will be added in nonincreasing order to the queue (so they will be sorted in THAT queue).*
- *At every step of the Huffman algorithm the current set of frequencies is stored in the 2 queues. The items are* sorted *in each queue. Find the item with smallest frequency by comparing the items at the heads of the two queues. Note that this can be done in $O(1)$ time because the head of each queue always hold the smallest item in that queue. Remove that smallest item from its corresponding queue. Now repeat the $O(1)$ operation to find the smallest remaining frequency in the two queues and remove it.*
- *add the newly created frequency to the tail of the 2nd queue.*

This takes the two smallest items off at every step and adds the new item so it is exactly implementing the Huffman algorithm. It uses $O(1)$ time per step instead of $O(n)$ so it is an $O(n)$ algorithm in total.

3. (CLRS–16.2-5) A *unit-length closed interval* on the real line is an interval $[x, 1 + x]$. Describe an $O(n)$ algorithm that, given input set $X = \{x_1, x_2, \ldots, x_n\}$, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct. You should assume that $x_1 < x_2 < \cdots < x_n$.

   Keep the points in an array. Walk through the array as follows.

   (a) Set $x = x_1$.

   (b) Walk through the points in increasing order until finding the first j such that $x_j > x + 1$. (if no such point then stop)

   (c) Output $[x, 1 + x]$

   (d) If there was no such j in (b) then stop. Otherwise, set $x = x_j$.

   (e) Go to Step (b)

   Note that each point is seen only once so this is an $O(n)$ algorithm.

   Let $Greedy(i, k)$ be the algorithm run on the Array $[i..j]$. Note that it can be rewritten as

   (a) Output $[x_i, 1 + x_i]$

   (b) Find min j such that $x_j > x_i + 1$.

   (c) If such a j does not exist, stop, else return $Greedy(j, k)$.

   We will prove correctness by induction on the number of points $|X|$.

   If $|X| = 1$ the algorithm is obviously correct. Otherwise, suppose $|X| = n$ and we know that the algorithm is correct for all problems with size $< n$.

   Let $O[i, j]$ to be the minimum number of intervals needed to cover $\{x_i, \ldots, x_j\}$ and $G[i, j]$ the number of intervals Greedy uses to cover them.

We assume that $[x_1, 1+x_1]$ does not cover all of X because, if it did, Greedy would return that one interval solution which is optimal.

Let j be the smallest index such that $x_j > 1 + x_1$. Note that Greedy returns $[x, 1 + x]$ concatenated with the greedy solution for $\{x_j, \ldots, x_n\}$ and, by induction, its Greedy solution for $\{x_j, \ldots, x_n\}$ is optimal. So, it uses $1 + O(j, n)$ intervals.

Now, suppose that there is a solution OPT different than the Greedy one. Let $[x, 1+x]$ be the interval with the leftmost starting point in OPT. Note that $x \leq x_1$ because otherwise x_1 would not be covered by any interval in OPT. Let k be the minimum index such that $x_k > 1+x$. After removing $[x, 1+x]$ the remaining intervals in OPT must form an optimal solution for $\{x_k, \ldots, x_n\}$ (otherwise we could build a solution using fewer intervals). So the total number of intervals used by OPT is $1 + O(k, n)$.

The main observation is that because $x \leq x_1$, $j \geq k$. Thus the optimal solution for $\{x_k, \ldots, x_n\}$ is A solution for $\{x_j, \ldots, x_n\}$ so it has at least as many intervals as the optimal solution for $\{x_j, \ldots, x_n\}$, i.e., $O(k, n) \geq O(j, n)$.

Combining the pieces yields

$$\begin{aligned} G(1,n) &= 1 + G(j,n) \\ &= 1 + O(j,n) \\ &\leq 1 + O(k,n) \\ &= O(1,n) \end{aligned}$$

which means that Greedy must be optimal for X.

4. Huffman Coding and Mergesort

   Recall that Mergesort can be represented as a tree,
   with each internal node corresponding to a merge of two lists.

   The weight of a leaf is 1,
   with the weight of an internal node being the sum of the weights of its two children,
   or equivalently, the number of leaves in its subtrees.

   The cost of a single Merge is the number of items being merged,
   so the cost of Mergesort is the sum of the weights of the tree's internal nodes.

   (a) Prove that the cost of Mergesort can be rewritten as the weighted external path length of its associated tree, when all leaves have weight 1

   This is not specific to mergesort. We wil prove that it is always true that if we define the weight of an internal node to be the number of leaves in its subtrees then the sum of the weights of the internal nodes is the weighted external path length of its associated tree, when all leaves have weight 1.

   In what follows, let T be a tree, $h(t)$ its height, $L(t)$ the number of leaves in the tree, $W(T)$ the sum of the weights of the internal nodes in the tree and $EPL(T)$ the weighted external path length of the tree when all leaves have weight 1. We want to prove that $W(T) = EPL(T)$ for all trees T.

   This statement is proven by induction on the height h of the tree. It is obviously true when $h = 1$ (and the tree has one or two leaves).

Suppose that it is true for all trees of height $< h$. Let T be a tree with $h(T) = h$ and x be the root of T.

If x has only one child let T_1 be the subtree falling off of x. Then by definition $W(T) = L(T) + W(T_1)$. On the other hand, every node in T is exactly one level deeeper than it was in T_1 so $EPL(T) = EPL(T_1) + L(T_1)$. Since $h(T_1) = h(T) - 1 = h - 1 < h$ the proof follows by the induction hypothesis and noting

$$W(T) = W(T_1) + L(T) = EPL(T_1) + L(T_1) = EPL(T).$$

If x has two children then let T_1 and T_2 be the left and right tree falling off of x. Since $h(T_1), h(T_2) < h(T) = h$ the induction hypothesis tells us that $W(T_1) = EPL(T_1)$ and $W(T_2) = EPL(T_2)$.

By definition $W(T) = W(T_1) + W(T_2) + L(T)$. On the other hand, every node in T_1 and T_2 is exactly one level deeper in T than it was in T_1 or T_2. So

$$EPL(T) = EPL(T_1) + L(T_1) + EPL(T_2) + L(T_2) = EPL(T_1) + EPL(T_2) + L(T)$$

and the proof again follows from the induction hypothesis and noting

$$W(T) = W(T_1) + W(T_2) + L(T) = EPL(T_1) + EPL(T_2) + L(T) = EPL(T).$$

(b) Prove that the recursive Mergesort studied in class has height $h = \lceil \log_2 n \rceil$, with $x = 2^h - n$ leaves on level $h - 1$ and $n - x$ leaves on level h.

The proof will be by induction on i. We will prove, for every i the statement is true for all $n \leq 2^i$.

It is obviously true for all $n \leq 2^1 = 2$. Now suppose that it is true for all $n \leq 2^{i-1}$.

Let $2^i < n \leq 2^{i+1}$. We need to show that it is true for all such n.

There are two cases, n odd and n even.

(a) n even: Then $n = 2n'$ with $n' \leq 2^i$. The algorithm splits the n items into two sets, each of size n', building a tree on each of them. By the induction hypothesis, each of those trees has height $h' = \lceil \log_2 n' \rceil$, with $x' = 2^{h'} - n'$ leaves on level $h' - 1$ and $n' - x'$ leaves on level h'.

Note that, by definition the height of the final tree is

$$h' + 1 = \lceil \log_2 n' \rceil + 1 = \lceil \log_2 2n' \rceil = \lceil \log_2 n \rceil$$

proving the first part of the statement.

Note that the leaves of the final tree are exactly the leaves of the subtree but pushed one level deeper. this means that the subtree has

$$2x' = 2(2^{h'} - n) = 2^{h'+1} - 2n' = 2^h - n$$

leaves on level $h' - 1 + 1 = h - 1$ and $n - x$ leaves on level $h' + 1 = h$.

(a) n odd: The proof is similar but needs a little bit of extra work when $n = 2^{i-1} + 1$.

(c) Show that an optimal Huffman tree for n items, all with the same frequency 1, will have the property that the tree will have height $h = \lceil \log_2 n \rceil$, with $x = 2^h - n$ leaves on level $h - 1$ and $n - x$ leaves on level h.

First note that the Huffman tree T must (by definition) have minimal weighted external path length among all trees with n nodes (with all leaves having weight 1).

Note that this immediately implies that all leaves must be on the bottom two levels of the Huffman tree. Suppose not and the tree has height h. Let (i) x be a leaf on level $d \leq h-2$, (ii) u be a leaf on level h and (iii) p be u's parent on level $h-1$.

Let v be the sibling of v (which must exist because the tree is full). Now make p be a leaf (associated with the character that used to be associated with x. This removes u, v.

Further, make x be an internal node with two children, associated with the characters that used to be associated with u, v. We have now created a new prefix-free coding tree for the same set of characters which has SMALLER weighted external path length, contradicting the optimality of T.

We now know that T is full (i.e., every internal node has two children; this fact was proven in class) and has all of its children at depths h and $h-1$. There are two cases

(a) ALL of the leaves are on one level, i.e., h. This means that all of the 2^{h-1} nodes on level $h-1$ are internal nodes. Since each of them must have two children and both of those children must be leaves there are $2 \cdot 2^h$ leaves (all on level h in the tree) so $n = 2^h$ and the statement is trivially correct.

(b) Leaves exist on BOTH level h and $h-1$. Note that, by the fullness, T has 2^{h-2} internal nodes at depth $h-2$ and thus 2^{h-1} nodes on level $h-1$. Suppose it has $y > 0$ leaves on level $h-1$. Note that because there are also leaves on level h, we must have $0 < y < 2^{h-1}$ and all of the non-leaf nodes on level $h-1$ have two children on level h. This implies that there are $2^{h-1} - y$ internal nodes on level $h-1$ and $2(2^{h-1} - y) = 2^h - 2y$ leaves on level h. This means $n = 2^h - 2y + y = 2^h - y$ so $2^{h-1} < n < 2^h$ and $h = \lceil \log_2 n \rceil$. Furthermore, the number of leaves on level $h-1$ is $y = 2^h - n$ and the number in level h is $n - y$ proving the 2nd part of the statement.

(d) Use the above facts to prove that recursive mergesort is *optimal*, i.e., that there is no other merge pattern for merging n items that has lower total cost.

(d) follows directly from (a), (b) and (c).

The Huffman algorithm constructs a tree with minimum weighted external path length. If all characters have weight 1, (c) immediately implies that this tree has cost

$$(h-1)(2^h - n) + h(2n - 2^h)$$

where $h = \lceil \log_2 n \rceil$. By the proof of optimality of Huffman codes, we know that this is the mimimum weighted external path length such a tree could have.

From (a) we know that the cost of a Mergesort is equal to the weighted external path length of its tree. From the previous paragraph we know that this can not be less than

$$(h-1)(2^h - n) + h(2n - 2^h)$$

From (b) we know that the tree for recursive mergesort has exact cost

$$(h-1)(2^h - n) + h(2n - 2^h)$$

which therefore implies it must be optimal.