CS:3330 Exam 2 Solution, Spring 2017

1. (a) $\{1, 8, 12, 3, 16, 10\}$

   (b) 1. $t \leftarrow -\infty$; $S \leftarrow \emptyset$
   2. **while** there is an interval that starts at or after t **do**
   3. **if** there is no interval I such that $start(I) \leq t < finish(I)$ **then**
   4. $t \leftarrow$ start time of earliest interval that starts after t
   5. $I \leftarrow$ interval that contains t and has the largest finish time
   6. Add I to S
   7. $t \leftarrow$ finish time of I 8. textbfreturn S

   (c) Let $A = \{a_1, a_2, \ldots, a_p\}$ be the solution returned by the algorithm, arranged in increasing order of start times. Note that no two intervals in A can have the same start time. Similarly, let $B = \{b_1, b_2, \ldots, b_q\}$ be an optimal solution, also arranged in increasing order of start times. Again, note that no two intervals in B can have the same start time.

   Now note that the start times of a_1 and b_1 are identical and equal to the start time of the earliest starting interval in the input. Because of the "greedy" choice (Line 5) of a_1, the finish time of a_1 is at least as large as the finish time of b_1. Therefore, a_1 "covers" everything that b_1 "covers" and the set

   $$O_1 = \{a_1, b_2, \ldots, b_q\}$$

   is a tiling path and since it has size q, it is also optimal. Also note that the intervals in O_1 are listed in increasing order of start time.

   As an inductive hypothesis, suppose that the set

   $$O_j = \{a_1, a_2, \ldots, a_j, b_{j+1}, b_{j+2}, \ldots, b_q\}$$

   is an optimal tiling path with intervals listed in increasing order of start times. Therefore, interval b_{j+1} starts after a_j starts. If interval b_{j+1} ends at or before interval a_j ends, then it would be entirely "covered" by a_j. We could then drop b_{j+1} and get a tiling path of size $q - 1$ – contradicting the optimality of O. So we can assume that b_{j+1} ends after a_j ends. After a_j has been picked by the algorithm, the variable t is set to the finish time of a_j in Line 7. Now there are two cases depending on whether there exists an interval I such that $start(I) \leq t < finish(T)$.

   – If there is an interval I such that $start(I) \leq t < finish(I)$, then both a_{j+1} and b_{j+1} start at or before t and finish after t. Among all such intervals, due to the "greedy" choice, a_{j+1} has the highest finish time. Therefore, the intervals $\{a_1, a_2, \ldots, a_j, a_{j+1}\}$ together cover everything that the intervals $\{a_1, a_2, \ldots, a_j, b_{j+1}\}$ cover. Therefore, the set

   $$O_{j+1} = \{a_1, a_2, \ldots, a_j, a_{j+1}, b_{j+2}, \ldots, b_q\}$$

   obtained by replacing b_{j+1} by a_{j+1} in O_{j+1} is an optimal tiling path.

   – If there is no interval I such that $start(I) \leq t < finish(I)$, then t is moved forward to the earliest start time of an interval after t (Line 4) and both a_{j+1} and b_{j+1} start at this new t. Among all such intervals, due to the "greedy" choice, a_{j+1} has the highest finish time. Therefore, the intervals $\{a_1, a_2, \ldots, a_j, a_{j+1}\}$ together cover everything that the intervals $\{a_1, a_2, \ldots, a_j, b_{j+1}\}$ cover. Therefore, the set

   $$O_{j+1} = \{a_1, a_2, \ldots, a_j, a_{j+1}, b_{j+2}, \ldots, b_q\}$$

   obtained by replacing b_{j+1} by a_{j+1} in O_{j+1} is an optimal tiling path.

   Therefore, by induction, $O_q = \{a_1, a_2, \ldots, a_j, a_{j+1}, a_{j+2}, \ldots, a_q\}$ is an optimal tiling path. This means that after the algorithm has added intervals $a_1, a_2, \ldots, a_q$ to S, there are no intervals left to add. Hence, $p = q$ and the set returned by the algorithm is optimal.

2. (a) The optimal Huffman encoding for the given set of frequencies is the following–

   a: 0000000
   b: 0000001
   c: 000001
   d: 00001
   e: 0001
   f: 001
   g: 01
   h: 1

   Of course, the binary code tree corresponding to the above encoding is also a valid answer.

   (b) The optimal binary Huffman encoding of one letter will be $n-1$ zeros and the other letter will be $n-2$ zeros followed by a one.

   (c) The algorithm to follow here is to take the three least frequent letters and merge them to form one letter. But we need to add an extra letter with frequency zero to make the number of letters odd before executing the algorithm.

   The reason for this is that since we are removing three letters and adding one (the combination of the three), we are effectively removing two letters from the input in each iteration. We don't want to be in a situation where there are two letters to combine at the last iteration of the algorithm. This is because the resulting ternary code tree will not be "full" (in a full ternary tree, every non-leaf node has exactly three children).[1]

   The optimal ternary code is –

   a: 0
   b: 20
   c: 21
   d: 1
   e: 220
   f: 221

   And there will be an extra letter with encoding 222 having frequency zero.

3. In this problem, we follow the convention used in class that if a vertex is already in the bag when we try to insert it, then the distance value of the vertex is updated instead of inserting another copy in the bag.

   The version given in Prof. Jeff Erickson's notes does not do this so the execution where we have two copies of a in the stack after iteration 2 is also correct.

No.	Contents of Stack	Vertex out of Stack	Relaxed edges	New $dist(\cdot)$ values
1.	s	s	$s \to a$, $s \to b$	$a = 4$, $b = -3$
2.	a, b	b	$b \to a$, $b \to d$	$a = 3$, $d = 9$
3.	a, d	d	$d \to c$	$c = 14$
4.	a, c	c	$c \to e$	$e = 13$
5.	a, e	e	None	None
6.	a	a	$a \to c$	$c = 5$
7.	c	c	$c \to e$	$e = 4$
8.	e	e	$e \to d$	$d = 6$
9.	d	d	None	None

[1] Since this was not pointed out in the problem, students were given full credit if they happen to miss this issue. Good job if you saw or fixed this issue!

4. For this problem we will maintain a counter $hops(\cdot)$ for each vertex in the graph where $hops(v)$ denotes the number of hops that the shortest path from s to v takes. The definition of a tense edge will need to be modified –

   An edge $e = (u, v)$ is tense if $dist(v) > dist(u)+w(e)$ or if $dist(v) = dist(u)+w(e)$ and $hops(u)+1 < hops(v)$.

   When we relax an edge (u, v) we update $hops(v) = 1+hops(u)$ in addition to updating the distance value and predecessor pointer.

   The priorities in the priority queue are also modified to break ties in favour of nodes that have a lower $hops(\cdot)$ value. That is if $dist(u) = dist(v)$ and $hops(u) < hops(v)$ then the node u will have higher priority.

5. (a) The running time of the implementation of Prim's algorithm discussed in class is $O(m \log n)$. To improve this running time to $O(m)$, we replace the min-heap priority queue data structure that holds edges leaving the component C, by a size-10 array $A[1 \ldots 10]$ of sets, where $A[j]$ is the set of all edges of weight j leaving the component C. In each iteration of Prim's algorithm, we look in A for a safe edge for C and on finding a safe edge $\{u, v\}$, $u \in C$, $v \notin C$, we add vertex v to C and then process all edges incident on v, deleting some of these edges from A and adding some of these edges to A.

   Since A has size 10, it takes $O(1)$ time to scan A left-to-right and find the first non-empty set in A. This yields a lightest edge in A, which in turn is a safe edge for C. Assume that the safe edge that was found was $\{u, v\}$, $u \in C$, $v \notin C$. After adding v to C, we process every edge $\{v, w\}$ incident on v and if $w \in C$, we delete $\{v, w\}$ from A and otherwise we add $\{v, w\}$ to A. Both the DELETE and ADD operations take $O(1)$ time because A is of size 10 and therefore it takes $O(1) + O(degree(v))$ time to find the safe edge $\{u, v\}$ and process edges incident on v. Summing this over all vertices v, we get a total running time of $O(n + m) = O(m)$.

   (b) We initialize a size-10 array $A[1 \ldots 10]$ of edge-sets and consider each edge e in the input graph and add it to the (initially empty) set $A[weight(e)]$. This sorts the edges by weight and we now consider edges in $A[1]$ first, then edges in $A[2]$, then edges in $A[3]$, etc. For the rest of Kruskal's algorithm we use the shallow threaded tree with union by size implementation of the DISJOINT SET UNION FIND data structure.

   Sorting the edges in the manner described above takes $O(m)$ time because each edge can be inserted into the array A in $O(1)$ time. (This algorithm is sometimes called *bucket sort*.) The shallow threaded tree with union by size implementation of the DISJOINT SET UNION FIND data structure supports MAKESET in $O(1)$ time, FIND in $O(1)$ time, and UNION in $O(\log n)$ amortized time (amortized over the $n - 1$ UNION operations). Thus the running time of the rest of the algorithm is $O(n)$ (for the n MAKESET operations at the beginning), plus $O(m)$ (for the $2m$ FIND operations), plus $O(n \log n)$ (for the $n - 1$ UNION operations). Thus the total running time is $O(m + n \log n)$.

6. Suppose we have a length-m intermixed sequence of `Push`, `Pull`, and `Decimate` operations. Each `Push` operation takes $O(1)$ time and so we charge \$1 to the element that was pushed by the `Push` operation. Each `Pull` operation takes $O(1)$ time and so we charge \$1 to the element that was pulled by the `Pull` operation. Each `Decimate` operation takes $O(n)$ time when it is applied on a dequeue with n elements. Also, the `Decimate` operation pulls out $n/10$ elements from this dequeue and so we can charge \$10 to each of the $n/10$ elements that were pulled out. In this manner, the $O(n)$ cost of `Decimate` has been equally distributed among the $n/10$ elements that were pulled out. This charging scheme ensures that the total running time of the m operations in the given sequence has been charged to elements that spent at least some time in the dequeue.

   Now I argue that at the end of the m operations in the given sequence each element that has spent any time in the dequeue is charged at most \$11. Any element that is currently is the queue is charged \$1 by the `Push` that pushed it in. Any element that was in the queue in the past, but

is not currenty in the queue, is charged either \$2 or \$11, depending on whether it left the queue because of a `Pull` operation or because of a `Decimate` operation.

Let K be the total number of elements that spent any time ever in the dequeue. Thus, we have shown that the total running time of all m operations is at most $11 \cdot K$. Therefore the amortized running time over the m operations is at most $11 \cdot K/m$. Now note that for every element that has ever spent any time in the dequeue, there is a `Push` operation that pushed that element in. Hence, $m \geq K$, and therefore $11 \cdot K/m \leq 11 = O(1)$.
