CMPS 102, Winter 2019 – Homework 2 Solutions, 1-2

1. (10 pts): Chapter 5, Problem 3. Determine if more than half of the bank cards examined are from the same account, using only a machine that detects if two cards are from the same account or not.

   There are two potential solutions to this problem. First is the divide and conquer solution.

   Algorithm (Divide and Conquer): (3 pts)

   Let $a_1 ... a_n$ be the account associated with each card. Two cards, c_i and c_j are equivalent if $a_i = a_j$. We are trying to determine if more than $\frac{n}{2}$ of the cards are equivalent - more than $\frac{n}{2}$ of the a_k have the same value x.

   If $n = 1$, there is just one card and it is a majority of the cards, so return it.

   Otherwise, arbitrarily divide the set of cards, S, into two sets $S_1 = \{1...\lceil \frac{n}{2} \rceil - 1\}$ and $S_2 = \{\lceil \frac{n}{2} \rceil ...n\}$ respectively. The algorithm will be recursively run on both sides in turn, assuming that if the algorithm finds that there is some value x such that more than half of the cards examined have an $a_k = x$, it will return a card c_i that has $a_i = x$.

   Call the algorithm recursively on $S1$, to obtain a card whose associated account is the majority in $S1$ (or the fact that no such card exists).n If a card is returned from S_1, test it against all the cards in S, and see if more than $\frac{n}{2}$ cards are equivalent to it. If they are, that card is associated with the majority account, and return it. If not, recursively call the algorithm on S_2, and check the returned card against all the cards in S to see if its account is associated with a majority of the cards in S, and either return it (if its account is the majority account in S) or return no majority (if neither returned card represents a majority account).

   Correctness (Divide and Conquer): (4 pts)

   If more than $\frac{n}{2}$ cards have the same a_k values, then at least one of the two sets of size roughly $\frac{n}{2}$ under examination must also have a majority of cards with the same a_k. In other words, if there exists a majority account in S, that same account must also be a majority in at least one of S_1 or S_2. This can be shown with a case analysis on if n is even or odd.

   Thus, if S has a majority account, then one of the two recursive calls must return a card that represents the majority.

   However, even if there does not exist a majority in S, there can still be a majority within either S_1 and/or S_2. The returned majority from the recursive calls must therefore be checked against S to determine whether it represents a majority in S as well as the half that it came from.

   This intuition be made more formal with a proof by induction on n. The property is $P(n)$ is that the algorithm is correct on all n-card inputs. The base case would show that $P(1)$ is true: on inputs of just 1 card, that card is a majority element and is returned by the algorithm. The inductive step would show $P(n)$ assuming $n > 1$ and $P(j)$ is true for all $1 \leq j < n$ using the argument above.

   Runtime (Divide and Conquer): (3 pts)

   The algorithm splits the starting set of size n in half each call ($O(\log n)$ levels). In the worst case, the algorithm must compare half the elements on each level against the other half ($O(n)$ uses of the machine per level). At most 2 full scans of the set are done to verify a majority ($O(n)$ additional uses of the machine). The total work is thus $O(n \log n + 2n) = O(n \log n)$.

   Alternatively, you could write the recurrence as $T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 2n$ where $T(n)$ is the number of card-comparisons used. The solution to this standard recurrence is $O(n \log n)$.

The following algorithm is also acceptable.

Algorithm: (3 pts)

In the recursive step: Pair up all cards and test them for equivalence. If both cards share an account, discard one. If both cards are different, discard both. If n is odd, there will be one card left out. Pretend it was compared against a fictitious card that had the same account and keep it. Repeat this process recursively until there is only one (or no) cards left.

If there were no cards left, return an indication that there was no majority card. If there was one card left, check this card against all the cards in S as a final post-processing step. If it is equivalent to $\frac{n}{2}$ other cards, then it represents a majority in S. If it is not, then there is no majority in S.

Correctness: (4 pts)

The first observation that can be made is that if there is a majority element, then there is a way to pair off every non majority card with a majority card and have at least one majority card left over. Each step of the algorithm discards at least half the cards left, but cannot discard more than half the majority elements (at most one element that is in the majority can be discarded from any pair). Thus, if there is a majority in in the set before any step of the algorithm, there must still be a majority after that step of the algorithm.

Another way to look at it: each mis-matched pair discards at most one majority card, and at least one non-majority card. Therefore there are more majority cards in matched-pairs than non-majority cards in matched pairs. Since 1 card is kept from each matched pair, more majority cards will be kept than non-majority cards. Again, this idea can be made into a formal proof by induction.

If there is no majority in the set, it is still possible for the algorithm to return a single card at the end of all the steps, so a final scan is done to verify whether the last card remaining represents a majority in S.

Runtime: (3 pts)

The final post-processing step requires $n - 1$ comparisons and is done only once. The recursive step takes $T(n)$ comparisons where $T(1) = 0$ and $T(n) \leq T(\lceil n/2 \rceil) + n/2$ comparisons, and the bound for this recurrence is $O(n)$.

Adding the two, the algorithm takes $O(n)$ time.

2. (6 pts): Chapter 4, Problem 2. Two true or false statements (3 pts each).

(a) True. If a graph is created with edge costs c_e^2, Kruskal's algorithm will sort and process those edges in the same order as it would the edges in the original graph, since if $a > b$ then $a^2 > b^2 \; \forall$ $a, b \geq 0$. Thus, the same subset of edges will be placed in the resulting MST. (Note that the answer would be reversed if the graph had negative cost edges.)

(b) False. Let $G(V, E)$ be a graph with $V = \{s, t, v\}$, $E = \{(s, v), (v, t), (s, t)\}$, and edge costs $|(s, v)| = 2$, $|(v, t)| = 2$, and $|(s, t)| = 3$. The shortest s to t path is (s, t), with cost 3. However, after squaring the edge costs, the shortest path becomes $(s, v) - > (v, t)$ with cost 8, as opposed to the original path (s, t) which now has cost 9.

CMPS 102 — Winter 2019 — Homework 2 — Problems 3 & 4

Problem 3 - Event sub-sequence identification Chapter 4 - Greedy Algorithms - Problem 4: Give an algorithm that takes two sequences, S' of length m and S of length n, each possibly containing an event more than once, and decides in time $O(m+n)$ whether S' is a sub-sequence of S.

Solution:

- Clearly describe your algorithm (3 pts)

  Definition 1. *A sequence of events S' is a* sub-sequence *of S if there is a way to delete certain events from S such that the remaining sequence of events is equal to the sequence S'. That is, if the sequence of events S' appears in the sequence of events S (i.e., in the same order but not necessarily consecutively), we say that S' is a "sub-sequence" of S.*

  The following algorithm takes as input two sequences of events, $S' = [e'_1, ..., e'_m]$ of length $m \geq 1$ and $S = [e_1, ..., e_n]$ of length $n \geq 1$. if S' is a sub-sequence of S, it returns **true**. Otherwise, **false**. It steps through the elements of S checking to see if they match the next element of S', advancing in S' when they do. It keeps track of the position $k[j]$ where S matches $S'[j]$.

```
1:  function SUB-SEQUENCE( S', m, S, n )
2:      initialize k[1..m] to an array of 0's
3:      j ← 1                                   ▷ index of event in S' being matched
4:      for i from 1 to n do
5:          if S'[j] == S[i] then
6:              k[j] ← i                        ▷ so S[k[j]] = S'[j]
7:              if j == m then
8:                  return "true"
9:              end if
10:             j ← j + 1                       ▷ advance index of event in S' being matched
11:         end if
12:     end for
13:     return "false"
14: end function
```

  Not that the $k[]$-array is used primarily to help the proof of correctness (although it could be used to return the positions of the match if one is found).

- Prove that it is correct (4 pts).

  Some observations:

  1. Since i is increasing, whenever some $k[j]$ is set to i, it will be greater than $k[j-1]$, and all $k[\ell]$ values (for $1 \leq \ell < j$ will have been set to non-zero values.
  2. $S'[j] = S[k[j]]$ for all values j where $k[j]$ is set to non-zero.
  3. j is initialized to 1 and only increases (by 1) when a match for $S'[j]$ is found in S.

  Claim 2. *If the algorithm returns true, then there is a subsequence of S that is equal to S'.*

Proof. The algorithm returns true only after $k[m]$ is set to a non-zero value. Therefore, all $k[\ell]$ for $1 \leq \ell \leq m$ have been set to non-zero values, and each $S[k[\ell]] = S'[\ell]$. Since the $k[\ell]$ values are increasing, the $S[k[\ell]]$ values form a subsequence of S that is equal to S'. ■

Claim 3. *If there is a subsequence of S that is equal to S', then the algorithm returns true.*

The proof is a version of greedy-stays-ahead.

Proof. Let $S[h_1], S[h_2], \ldots, S[h_m]$ be any subsequence of S that is equal to S'. Thus each $S[h_\ell] = S'[\ell]$.
We claim that $h_\ell \geq k[\ell]$ for $1 \leq \ell \leq m$, and prove this claim by induction. Let $P(\ell)$ be the statement that $0 < k[\ell] \leq h_\ell$.

For the base case we show $P(1)$.
We defined $S[h_1]$ such that $S[h_1] = S'[1]$. Therefore if $j = 1$ when line 5 is executed for the $i = h_1$ loop, then $k[1]$ will be set to h_1. If $j > 1$ when line 5 is executed for the $i = h_1$ loop then j was set to 2 at some previous iteration i' where $S[i] = S'[1]$, and $k[1] = i' < h_1$. Since $i' \geq 1$, $k[1] > 0$.

For the inductive step, Assume that $1 \leq \ell < m$ and $P(\ell)$ is true to show that $P(\ell + 1)$ is true.
From $P(\ell)$, $h_\ell \geq k[\ell]$ and $k[\ell] \geq 0$. Therefore, at iteration $k[\ell]$, the algorithm sets $j = \ell + 1$ and starts matching against $S'[\ell + 1]$. If a match of $S'[\ell + 1]$ is found before iteration $h_{\ell+1}$, then $k[\ell + 1]$ will be less than $h_{\ell+1}$. If not, then at iteration $h_{\ell+1}$ the algorithm with find a match for $S'[\ell + 1]$ (since $S[h_{\ell+1}] = S'[\ell + 1]$).

This completes the induction.

Since $0 < k[m] \leq h_m \leq n$, we have that that $k[j]$ for $j = m$ is set during some iteration of the loop, and immediately afterwords (on line 8) the algorithm returns "true". ■

The above can also be shown by contradiction along the the following outline: Assume to the contrary that some h_ℓ is less than $k[\ell]$, and consider the lowest value of ℓ where this occurs.

- Analyze its running time (2 pts).

  The initialization of the k array takes $O(m)$ time. The "for i" loop runs for at most n iterations, and takes constant time per iteration. The initialization of j and returning false each take constant time. Therefore the total time for the algorithm is $O(m + n)$.

CMPS 102 — Winter 2019 — Homework 2 — Problems 3 & 4

- **Problem 4**: Clock skew elimination.

  Problem 24 - Chapter 4 - Greedy algorithms:

  Consider a complete balanced binary tree with n leaves, where n is a power of two. Each edge e of the tree has an associated length l_e, which is a positive number. Recall that in complete binary trees, each node is either a leaf or an internal node with 2 children.

  The *distance* from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

  The root generates a clock signal which is propagated along the edges to the leaves.

  We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

  Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem. We want the leaves to be completely synchronized, and all to receive the signal at the same time. To make this happen, we will have to increase the lengths of certain edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have zero skew.

  Our goal is to achieve zero skew in a way that keeps **the sum of all the edge lengths** as small as possible.

  Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew and the total edge length is as small as possible.

  Solution:

  Let V denote the set of nodes in T and E the set of edges $e(u, v)$ in T, such that u is the parent of v.

  Each edge $e(u, v)$ in the tree has an associated length $l_{e(u,v)}$, which is a positive number. Let l denote the set of those lengths.

  - Clearly describe your algorithm (3 pts)

    The following algorithm takes as input the root of a complete balanced binary tree and it increases the weights of certain edges so as to remove any skew while keeping the total edge length as small as possible.

```
1:  function AdjustSkew(v)                    ▷ returns the length of v-to-leaf paths
2:      L ← 0                                 ▷ L will be path lengths from v to leaves
3:      if v is a leaf then
4:          return L
5:      else                                  ▷ v has two children
6:          let v_lc and v_rc be the left and right children of v
7:          L_lc ← AdjustSkew(v_lc)
8:          L_rc ← AdjustSkew(v_rc)
9:          leftLength ← L_lc + l_e(v, v_lc)
10:         rightLength ← L(v_lc) + l_e(v, v_lc)
11:         L ← max{leftLength, rightLength}
12:         if leftLength > rightLength then
```

13: $l_e(v, v_{rc}) \leftarrow l_e(v, v_{rc}) +$ leftLength $-$ rightLength
14: **else if** leftLength $<$ rightLength **then**
15: $l_e(v, v_{lc}) \leftarrow l_e(v, v_{lc}) +$ rightLength $-$ leftLength
16: **else**
17: do not change
18: **end if**
19: **end if**
20: **end function**

– Prove that that it correctly equalizes the delays (2 pts):

Proof. First observe that the algorithm only increases the edge lengths and does not decrease them.

We will actually prove that AdjustSkew called on the root will equalizes the delays in any full binary tree, not just complete binary trees. The proof will be by induction on the number of internal nodes.

Let $P(n)$ be the statement: In every full binary tree with n internal nodes, root r, and lengths on the edges, AdjustSkew(r) increases the edge lengths so that all root-leaf paths have length L, the value returned.

Base case: Show $P(0)$:

The 0-internal node binary tree consists of a single leaf, and only the trivial root-to-leaf path of no edges. In this case, AdjustSkew returns 0, the length of all root-to-leaf paths.

Induction step: Assume $n > 0$, and $P(j)$ for all $0 \leq j < n$ to prove $P(n)$.

Let r be the root of an arbitrary full binary tree with n internal nodes and consider the behavior of AdjustSkew(v) when $v = r$. Each of r's children, v_{lc} and v_{rc}, are the root of a full binary tree with less than n internal nodes. By the induction hypotheses, after line 8, L_{lc} and L_{rc} are the lengths of all paths from v_{lc} to leaves in its subtree, and v_{rc} to leaves in its subtree respectively. leftLength and rightLength are the lengths of the paths from v to all leaves in the left and all leaves in the right subtrees respectively, and L is set to the maximum of these. Finally, if those lengths are not the same, the edge from v to the subtree with the shorted path lengths is increased so they are equal. Thus after the call all v-to-leaf paths have the same length L, the value returned by the call. ■

– Prove that it uses the least possible additional sum of edge lengths (5 pts)

In fact, the algorithm gives the only optimal solution. We will prove this by contradiction, assuming that there is a different optimal solution and showing that the different solution can be improved, and thus is not optimal. But first we will establish some facts about the algorithm and optimal solutions.

Note: that the algorithm increases at most one outgoing edge from each subtree root. Therefore there is a root-to-leaf path where none of the edges have been increased. Since the final root-to-leaf lengths are all the same, and none are decreased, the algorithm adjusts edge lengths so that all root-to-leaf paths become the same length as the (original) longest root-to-leaf path.

Lemma 1. *Let v be an internal node in the tree, if a solution adds positive length to both edges coming out of v then it is not optimal.*

Proof. (By contradiction) Assume to the contrary that in some optimal solution S, some vertex v has positive length added to both of its outgoing edges. If v is the root, subtract from both edges until one of them has 0 added to it. The root-leaf distance drops the same amount for every leaf. If v is not the root, zero the smaller amount added to the outgoing edges, subtract the same amount from the other edge, and add that amount to the edge between v and v's parent. Both cases reduces the total amount added to the edges while keeping the root-to-leaf paths all the same length, contradicting the optimality of S. ■

Lemma 2. *Let v be an internal node. No optimal solution increases the length of all v-to-leaf paths for the leaves in v's subtree.*

Although this may seem obvious, the proof is a little tricky as we must find a way move the length increases up to node v while preserving the root-to-leaf lengths.

Proof. Assume to the contrary that some optimal solution S increases the length of all the paths from v to leaves in its subtree. Let T be the subtree rooted at v. For each leaf ℓ in T, let e_ℓ be the first edge on the v-to-ℓ path that has its length increased by S, and E be the set of these first edges. Each leaf $\ell \in T$ has one, and only one, edge on its v-to-ℓ path in the set E. Let $\delta > 0$ be the minimum amount of increase over the edges in E. If v is not the root, decrease the lengths of all the edges in E by δ while adding δ to node v's incoming edge. This keeps all root-to-leaf path lengths the same and keeps the edge-length changes non-negative. Since no edge is on all v-to-leaf paths $|E| \geq 2$, so the change reduces the total added edge length by at least δ and creates a now solution that is better than S, contradicting S's optimality. If v is the root, decrease the lengths of all edges in E, decreasing the lengths of all root-to-leaf paths. This creates a new solution that is better than S contradicting S's optimality. ■

We are now ready for the main proof.

Claim 3. *The algorithm produces the unique optimal solution to the clock-skew problem.*

Proof. Let A be the solution produced by the algorithm and let S be some different optimal solution. Let L be the length of the longest (un-modified) root-to-leaf path, both the algorithm and S (by Lemma 2) make every root-to-leaf path have length L. Let v be a node closest to the root where A and S add different lengths to the edges coming out of v, and c be the length of the of the modified root-to-v path (this is the same in both S and A since edges above v have the same modified costs). Both S and A make all v-to-leaf paths have length $L - c$. From the proof of A, at least one original v-to-leaf path has length $L - c$, so both A and S add zero skew to the same outgoing edge of v. Let v' be the other child of v, so A and S add different amounts of skew to the (v, v') edge.

If A adds more skew to the (v, v') edge, then S would have to add skew to all v'-to-leaf paths in order to make up the difference, and S would not be optimal by Lemma 2.

Consider the case where A adds less skew to the (v, v') edge, and let δ be the skew A adds to this edge. By the above note, some leaf ℓ in teh subtree rooted at v' has a v'-to-leaf path with (original) length exactly $L - c - \delta$, and if S adds $\delta' > \delta$ skew to the (v, v') edge, then the root-to–that-leaf path will have length $L - c - \delta + c + \delta' > L$, contradicting that S makes all root-to-leaf paths length L. ■