CS180 Solution 4

1. Given n files and an array $f[1..n]$, where $f[i]$ stores the access frequency count of file i. You are asked to build a binary tree and put n files as n *leaves* of the tree. The cost of accessing each file i is the depth of the leaf where file i is placed at. The goal is to minimize the total cost of all accesses $\sum_{i=1}^{n} f[i] \cdot \text{depth}(i)$. In class, we have seen an $O(n \log n)$ time algorithm builds the optimal binary tree using the heap. A student suggested in class that if the array f is *sorted* in increasing order, the optimal binary tree can be constructed in $O(n)$ time. Give an $O(n)$ algorithm assume f is sorted and prove your algorithm works.

   Solution:

   The problem asks for an implementation of Huffman coding. The greedy rule of the Huffman coding algortihm is: pick 2 files with the smallest frequencies as two leaves , merge their frequencies into a big file as the root. Recursively solves the remaining files with the big file. If f is sorted, we can use two queues to help us to extract the two smallest frequency files during the construction.

```
HUFFMAN(f[1..n])
  Q_f ← QUEUE(f)          //construct a queue from f
  Q_internal ← QUEUE()    //construct an empty queue
  while there are two or more files in Q_f and Q_internal
      take two files a and b with the smallest frequency from Q_f and Q_internal
      merge them into an internal file ab with f[ab] = f[a] + f[b]
      construct the tree with a, b as the child and ab as the parent.
      enqueue ab in Q_internal
  return the last node in Q_internal as the rooted optimial binary tree
```

   The time complexity of this algorithm is apparently $O(n)$.

2. Given a list of n items such that the order of any two items can be determined by one comparison. Design an $O(n)$ algorithm to construct a heap contains all n items.

   Solution:

```
HEAPIFY(A[1..n]) // Max Heap
  for i ← floor(n/2) downto 1
      SIFTDOWN(A,i)
```

```
SIFTDOWN(A[1..n], i)
  l = LEFT(i) // index of the left child of A[i]
  r = RIGHT(i)
  largest = i
  if A[i] > A[largest]
      largest = l
  if A[r] > A[largest]
      largest = r
  if largest ≠ i
      A[i], A[largest] = A[largest], A[i] // swap
      SIFTDOWN(A, largest)
```

Proof for O(n) time complexity:
Consider the height of the bottom level as 0. A heap of size n has at most $\lceil n/2^{h+1} \rceil$ nodes at height h.
For a node at height h, SIFTDOWN takes O(h) time.
The total cost in heap building:

$$T(n) = \sum_{h=0}^{\log n} \lceil \frac{n}{2^{h+1}} \rceil \times O(h)$$
$$= O(n \sum_{h=0}^{\log n} \frac{h}{2^h})$$

Recall that $\sum_{m=0}^{\infty} m x^m = \frac{x}{(1-x)^2}$ when $x < 1$. We have

$$\sum_{h=0}^{\log n} \frac{h}{2^h} < \sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

Therefore, $T(n) = O(n)$

3. Given a directed weighted graph $G = (V, E)$ with every edge e has distinct weight $w(e) > 1$. Assume G contains a node s that can reach all other nodes.

   (a) Prove that there exists a directed tree T in G with root s as a subgraph in G, with the property that the path from s to any other node v in T is the shortest path from s to v in G.

   Solution:

   The shortest paths from node s to other nodes forms a tree. To see this, we can apply the exchange arguement. Suppose there is a cycle $s \rightsquigarrow u \rightsquigarrow s$ in the underlying undirected graph of all shoretest paths, there must exists shortestpaths $s \rightsquigarrow u \rightsquigarrow v$ and $s \rightsquigarrow u \rightsquigarrow w$ such that s can reach u by two different paths and one of them have smaller length. We can make $s \rightsquigarrow u \rightsquigarrow v$ or $s \rightsquigarrow u \rightsquigarrow w$ even shorter by substituing the shorter $s \rightsquigarrow u$ path, and that contradicts the fact both $s \rightsquigarrow u \rightsquigarrow v$ and $s \rightsquigarrow u \rightsquigarrow w$ are shortest paths

(b) Suppose we square the weights of the edges in G. In other words, we have the same graph G except every edge e has a squared weight $w_1(e) = w(e)^2$. The shortest path tree now might change compared to what you have before. Prove that if we change the tree by squaring the weights again and again, eventually from some point the shortest path tree will stay the same. Thus, these shortest paths now can be defined with no reference to the actual weights, i.e., given two paths instead of evaluating whether path p_1 is shorter than path p_2 can be determined just by comparing weights, without having the operation of addition of weights. Describe this characterization of the way of comparing the length of paths.

Example: Suppose when we started there are just two paths, each of two edges, from s to some v, One path has two edges of weights 5 and 5 and the other 7 and 2. At the beginning, the 7,1 path is the shortest because its length is 9 and the other is 10, but after one squaring we will get 25, 25, and 49, 4, respectively. Now the shortest path is the previous 5,5! Why this happened? The most weighty edge between the two paths is 7, and squaring enough will get it to dominate. If the two paths were tied with respect to the most weighty, then the second most weighty will break the tie, etc.

Solution:

We define a order between two paths A and B. We sort the weights of edges in the *decreasing* order and $A = \{w_1, w_2, \ldots, w_i\}, B = \{w'_1, w'_2, \ldots, w'_j\}$. We say A is bottleneck-smaller than B, if the first different weight edge in A has smaller weight than the edge in B, more precisely if for the smallest index k in A at which $w'_k \neq w_k$, we have $w_k < w'_k$. If there is not such k that $w'_k \neq w_k$, the path with smaller number of edges is bottleneck-smaller. We define the minimum bottleneck path as the path that is bottleneck-smaller than any other path. After sufficiently squaring the weights of the edges, it is easy to see the shorestest path after sufficiently squaring the weights in the graph corresponds to the minimum bottleneck path in the original G because the larger term among the weights of all edges will dominate other smaller terms if you square them enought times, which corresponds to the asymptotic growth of exponiential function.

(c) Given a directed weighted graph $G' = (V, E)$ such that every edge is bi-directional, i.e., an edge (u, v) in G implies another edge (v, u) and they have same weights. Fix an arbitrary node v in G, and like what we have done in part(c), we can find a stable tree T'_k for G'_k. Prove that T'_k is the minimum spanning tree of G' if ignoring the directions on G'.

Solution:

Since the minimum bottle neck path corresponds to stable shorestest path tree after sufficient squaring the weights. We can modify the Dijkstra algorithm to find the stable tree in the original graph G.

Consider the following variation of Dijkstra algorithm. Instead of computing the distance, we calculate the minimum bottleneck path from the source s to any other node. Each node v maintain a variable $P(v)$ that stores the minimum bottleneck path from the s and ends at v. Moreover, the operation on the priority queue is base on the order of $P(v)$.

```
DIJKSTRAMST(s)
  put s in the priority queue
  T ← {}
  while the priority queue is not empty
      extract node u from the priority queue with the minimum bottleneck path
      remove u from priority queue
      T ← T ∪ {u}
      for all edge u − v that v is not in T
          if v is not in priority queue
              P(v) ← {P(v), u − v}
              put v in the priority queue
          else if P' = {P(u), u − v} is bottleneck-smaller than P(v)
              P(v) ← {P(v), u − v}
```

After the algorithm, every node will be added to the tree, and $P(v)$ defines a path from s to v for all nodes, and similarly to prim algorithm, there is no cycle. So the set defines a spanning tree of the graph. Moreover, the spanning tree defined by DIJKSTRAMST is the mst. For any path from s to arbitrary node v, if $s \to v$ has minimum bottleneck, then the subpath from node i to v where i is a node on path $s \to v$ has minimum bottleneck. Let B_{ij} be the bottleneck value of the path from i to j. By the algorithm, every path from s to arbitrary node v has minimum bottleneck. So we need to show that for every pair of node (i, j) has minimum bottleneck. Let i, j be 2 different nodes, then there exists a node k, such that k is on path $s \to i$ and $s \to j$ in the tree, also $k \to i$ and $k \to j$ has no common node besides k. Then the path from i to j in the tree is $i \to k \to j$. Suppose this path is not the minimum bottleneck, then there exists a path from i to j, such that $B_{ij} < B_{ikj}$. So $B_{ij} < max\{B_{ik}, B_{kj}\}$, w.l.o.t., we assume $B_{ik} < B_{kj}$. So every path on the tree has minimum bottleneck.

Next , we show that given two nodes u and v, the minimum bottleneck path is the the path P that connects u and v in the MST.

Theorem 1. *The minimum bottleneck path P from node u to v is the path defined by the MST tree*

Proof. Proof by contradiction. Suppose there is another path P' such that the first different weight edge e' has the smaller weight than e in P ($w_{e'} < w_e$). We remove the edge e_b in the MST and disconnect the MST into two components, one component A contains the node u and the other B contains the node v. Since the path P' connect u with v, we can find a edge e_b in P' that connects A and B. Moreover, the weight of this edge $w_b \leq w_{e'} < w_e$. So, we can remove the edge e in the MST and add the edge e_b to it to get a new spanning tree such that it has the smaller weight than the original MST. Therefore, we get a contradiction and P is the minimum bottleneck path. □

Assume the graph has distinct weights, the mimium bottle neck path is uniqlely determined.Therefore MST and the spanning tree returned by DIJKSTRAMST is also uniqlely determined, and so they are the same tree.

(d) Use the way we compared directed paths before to extend it to compare between two spanning tree which one is "better." Argue that the MST defined as the tree that has the minimum sum of weights of its edges, is the same tree that is smaller than all other trees if we just determine which is smaller among 2 trees by comparison rather than addition.

Solution:

We compare two spanning trees by comparing all the edges in lexicographic order, i.e., sort the weights of edges in the *decreasing* order and $A = \{w_1, w_2, \ldots, w_i\}, B = \{w'_1, w'_2, \ldots, w'_j\}$ and compare them lexicographically. We show that MST is a lexicographically minimum spanning tree. Proof by contradiction. Let A be the lexicographically minimum spanning tree $(w_1, \ldots, w_{n-1})$, B be the minimum spanning tree $(w'_1, \ldots, w'_{n-1})$. Let w_i be the smallest i such that $w_i \neq w'_i$. Since A is the lexicographically minimum spanning tree, we have $w_i < w'_i$. We can add the edge w_i to B. This creates a cycle in tree B. This cycle contains an edge from the set $\{w'_i, w'_{i+1}, \ldots, w'_{n-1}\}$. Otherwise all the edges of the cycle are from the set $\{w'_1, w'_2, \ldots, w'_{i-1}\} = \{w_1, w_2, \ldots, w_{i-1}\}$, and this means that there is a cycle consisting of edge from the set $\{w_1, w_2, \ldots, w_{i-1}, w_i\}$ in tree A which is a contradiction. Therefore adding w_i to B creates a cycle in B that contains an edge from $\{w'_i, w'_{i+1}, \ldots, w'_{n-1}\}$. The weight of all edges in $\{w'_i, w'_{i+1}, \ldots, w'_{n-1}\}$ is greater than w_i, and deleting one of them from the cycle gives us a new tree with weight less than the minimum spanning tree, a contradiction.

4. Given a set $\Omega = \{e_1, e_2, ..., e_n\}$ and a collection of sets $S_1, S_2, .., S_n$. Each element e_i has a corresponding set $S_i = \{e_i\}$. You are now given a sequence of commands of type **Union**(e_x, e_y) interspersed with commands **Query**(e_x, e_y). The union operation takes the two sets one containing e_x, and the other containing e_y (they might be the same set, but not at the beginning) and union it to a single set containing the elements of both. As we go creating sets we query command asks whether the two elements in its argument NOW (at the time you reached it in the sequence) belong to the same set or a different set.

   (a) Given a list of $O(n)$ operations of Union and Query. Design an algorithm that "processes" the sequence of commands in $O(n \log n)$ time. To process the sequence means to answer all the queries in the sequence. (Hint: keep a set as a rooted tree with depth that is $O(\log n)$. The Crux is how to union two trees and keep this property.)

   Solution:
   Design a data structure such that we can finish Union or Query in $O(\log n)$ time.
   Consider a set as a rooted tree in which each element is represented as a node and each node points to its parent. If two nodes have the same root, they are in the same set.
   Here we use an array to record the parent for each node.
   To maintain an $O(\log n)$ depth, when we union two trees, we want the root of the smaller tree to point to the larger tree. Therefore, we record the subtree size for each node.

```
class UNIONQUERY

  initialize(n)
      // Every set has only one element at the beginning. Each node points to itself.
      parent ← [1, 2, ..., n]
      size ← [1, 1, ..., 1]

  root(i) // find the root for e_i
      if parent[i] == i
          return i
      return root(parent[i])

  Union(x, y) // union the set of e_x and the set of e_y
      i ← root(x)
      j ← root(y)
      if i ≠ j
          if size[i] ≤ size[j]
              parent[i] ← j // let the smaller tree root point to the larger tree root
              size[j] += size[i]
          else
              parent[j] ← i
              size[i] += size[j]

  Query(x, y) // query if e_x and e_y are in the same set
      return root(x) == root(y)
```

(b) Given an undirected weighted graph $G = (V, E)$. Show an implementation of Kruskal's Minimum Spanning Tree(MST) algorithm by using the Union and Query commands. Your implementation should have $O(|E| \log |V|)$ time complexity. (Hint: In Kruskal's algorithm, it builds the MST by processing every edge in G and asks whether two endpoints of an edge belong to two disjoint trees. If so, the algorithm adds the edge making two disjoint trees into a single tree.)

Solution:

```
KRUSKAL(G = (V, E))
  MST ← [ ]
  E ← sort(E)
  UQ ← UNIONQUERY(|V|)
  for e = (u, v) in E
      if UQ.Query(u, v) == FALSE
          UQ.Union(u, v)
          add e to MST
  return MST
```

Time complexity analysis:
Denote $|E|$ as m and $|V|$ as n.
Sorting E takes $O(m \log m)$. As $m \leq n^2$, we can rewrite it as $O(m \log n^2) = O(m \times 2 \log n) =$

$O(m \log n)$.
There can be at most $2n$ Query operations $\Rightarrow O(n \log n)$.
There can be at most $m-1$ Union operations $\Rightarrow O(m \log n)$.
It is $O(m \log n)$ in total.