Name: Jaime Arana-Rochel

Homework #3

Q1: Exercise 19 Chapter 4

Algorithm

We simply run either Kruskal's or Prim's algorithm with the edge costs equaling the negative bandwidth value b_e associated with each edge.

Correctness

We first make the assumption that all of the bandwidth values b_e are distinct. If they are not, then we apply the strategy in the book of perturbing the edge costs by extremely small amounts so they become distinct, which does not affect the solution MST. Also, having negative costs obviously does not affect the execution of either Kruskal's or Prim's since we are just comparing edge costs, so they both do return an MST T.

Assume that running either Kruskal's or Prim's algorithm with negative bandwidth costs for each edge does not return a MST T in which the bottleneck rate of each $u - v$ path in T is equal to the best achievable bottleneck rate for the pair u, v in G. That means that there is a pair of vertices u, v in which the path P in T does not have the highest bottleneck rate; some other $u-v$ path P' in G has a higher bottleneck rate. Consider the edge e in this path P with minimum bandwidth which is therefore not in P'. We notice that there is a $u-v$ path which does not use this edge e but instead uses the rest of the edges in P and the edges in P'. That means that in G, there is a cycle C containing e. Because we are using negative bandwidth values, that means that in the MST T produced by either Kruskal or Prim, e has the maximum edge cost in the cycle C. By the Cycle Property, this edge would then have not been included in the MST and so we arrive to a contradiction in our original assumption.
Therefore, the MST T returned by either Kruskal's or Prim's algorithm, with the edge costs being the negative bandwidth values, does indeed satisfy the required conditions.

Running Time Analysis

There are no alterations to either Kruskal's or Prim's algorithms; only a change in the input which does not affect the solution found. Therefore the running time is the same as in the different implementations of those algorithms.

Q2: Exercise 1 Chapter 5

Algorithm

```
median(n, A, B, indexA, indexB):
   k = ceiling(1/2*n) //value of half the database size

   // find the median of the two individual databases
   medianA = Query(A, indexA + k)
   medianB = Query(B, indexB + k)

   // base case
   if n=1 then return min(medianA, medianB)

   // recursive step
   if medianA < medianB
      // check the last half of A, and the first half of B
      return median(k, A, B, indexA+floor(1/2*n), indexB)
   else
      // check the first half of A, and the last half of B
      return median(k, A, B, indexA, indexB+floor(1/2*n))
```

The algorithm is then run with initial inputs

$$median(n, A, B, 0, 0)$$

where A and B are ints specifying their respective database.

Correctness

Proof by induction on n:

Base Case: n = 1
When n=1, the algorithm return the minimum of the only value in the databases A and B. This equals the median value of the set of combined values from A and B.

Inductive Step:
Let the induction hypothesis I.H be the following: Assume that for all i, $1 \leq i \leq n-1$, the algorithm correctly returns the median value from the set of combined values of databases A and B, which each have size i.

Now suppose that the algorithm is run with input size $i = n$ for databases A and B. The first thing the algorithm does is query A and B to figure out their respective median values ($medianA$ and $medianB$). It does this by calling the function $Query$ with $k = \lceil n/2 \rceil$. The algorithm then checks for two possibilities; either $medianA$ is less than $medianB$ or the opposite is true. The argument will be the same in either case with the only change being

the database names.

Case [$medianA < medianB$]:
If this is true, then we see that $medianB$ is greater than the first k smallest values of A, and obviously greater than the first $k-1$ smallest values of B. It follows then that $medianB$ is greater than at least $2k-1$ database values. Therefore $medianB$ is greater than or equal to the median of the combined database values, so we can ignore values in B that are greater than $medianB$ in the recursive call (we focus on the first k smallest values in B). This is shown in the algorithm by leaving the index to B ($indexB$) the same as it was in the beginning of the execution, which initially is 0.
Likewise, we see that $medianA$ is less than the last k to n smallest values in B, and less than the $k+1$ to n smallest values in A. It follows that $medianA$ is less than at least $2n-2k-1$ database values. Therefore $medianA$ is less than or equal to the median of the combines database values. That means we can ignore values smaller than $medianA$ in the recursive call (we focus on the greater half of database A). This is shown in the algorithm by making $indexA$ equal to $indexA + \lfloor n/2 \rfloor$, which points the index into A starting on the $k+1$ smallest value.
In updating the indices into the databases, effectively ignoring some values which we know cannot be the median, we are narrowing down the interval of values which can be the median by a factor of k defined above. It follows from this that the median of the combined set of values of A and B is contained in the median of this new set of values; the median is found in the recursive call on these updated databases. When the algorithm makes the recursive call, the value of n gets updated to k, so by the Induction Hypothesis, we know that the algorithm correctly finds the median in this new call.
This completes the inductive step, and by induction that means that the algorithm correctly calculates the median.

Running Time Analysis

It is assumed that the time to query a database is constant, $Query = O(1)$. In each recursive call we halve the size of our input n and make to queries. Therefore we get the following recurrence relation for the algorithm

$$T(n) = T(\lceil n/2 \rceil) + 2$$

Summing these two query operations over $log\ n$ levels of recursion, we get a running time of $O(2 * \lceil log\, n \rceil) = O(log\, n)$.

Q3: Exercise 3 Chapter 5

Algorithm

```
equivalence(S):
   n = |S|
```

```
If n = 1
   return card c in S
If n = 2
   c1 = first card in S
   c2 = second card in S
   // test if the two cards are equivalent
   if equiv_tester(c1,c2) = true
      return card c1
   else return "FALSE"

// divide the set of cards S evenly in 2
Let S1 = the first floor(n/2) cards in S
Let S2 = the last ceiling(n/2) cards in S

If equivalence(S1) returns a card c_1
   call equiv_tester() with c_1 against every other card in S

If c_1 is not the card for which more than n/2 cards are equivalent to it

   If equivalence(S2) returns a card c_2
      call equiv_Tester() with c_2 against every other card in S

   If c_2 is not the card for which more than n/2 cards are equivalent to it
      return "FALSE"
      else return c_2

else return c_1
```

Correctness

If among the collection of n cards there is a set of more than $n/2$ of them that are equivalent to one another, then at least one of the two halves of the set S will have more than half of their cards ($n/4$) equivalent to this majority set. Therefore, assuming this majority set exists, one of the recursive calls to a halve of S will return a card which is equivalent to the majority set. That returned card is then tested against every other card in the set, which ensures that the algorithm does indeed determine if there is a majority set by returning that card. If there is no majority set, then the algorithm returns "FALSE" due to no card being returned by the algorithm described.

Running Time Analysis

The algorithm makes two recursive calls with half the input size each. In the worst-case running time, the algorithm will have to run the equivalence tester for two cards against every other card; meaning that there might be at most $2n$ equivalence tests performed. So

we get the following recurrence relation

$$T(n) = 2T(n/2) + 2n$$

We know from the book that this type of recurrence relation has solution $O(n\,log\,n)$.