Problem 1:

//Wrapper isn't necessary, but it makes things a bit simpler

```
int FillKnapsack(Vector<objectT> objects, int targetWeight) {
   return RecFillKnapsack(objects,targetWeight,0);
}
int RecFillKnapsack(Vector<objectT> objects, int weight, int score) {
   if (weight < 0)
      return 0;
   int bestScore = score;
   for (int i = 0; i < objects.size(); i++) {
      Vector<object> left = objects;
      int currScore = score + left[i].score;
      int currWeight = weight -= left[i].weight;
      left.removeAt(i);
      if (bestScore < currScore)
         bestScore = currScore;
      currScore = RecFillKnapsack(left,currWeight,currScore);
      if (bestScore < currScore)
         bestScore = currScore;
   }
   return bestScore;
}
```

20 points:

- 4 points: base case when you reach "weight" limit
- 3 points: iterates over all remaining objects
- 3 points: updates score and weight for each objects
- 3 points: removes object from Vector
- 3 points: makes recursive call
- 4 points: updates "bestScore" if the score returns by recursive call is higher

2a)

```
   private:
   DBTree *children[3];
   Vector<int> values;

   DBTree::DBTree() {
   for (int i = 0; i < 3; i++)
       children[i] = NULL;
   }

   DBTree::~DBTree() {
       for (int i = 0; i < 3; i++) {
           delete children[i];
       }
   }
```

5 points:

- 1 point: represents children

- 1 point: represents values
- 1 point: sets children to null in Constructor
- 2 points: deletes children in Destructor

2b)

```
bool DBTree::IsLeaf() {
    for (int I = 0; I < 3; i++)
        if (children[i])
            return false;
    return true;
}

Int DBTree::GetIndexForValue(int value) {
    If (values[0] > value)
        return 0;
    else if (values[1] > value)
        return 1;
    return 2;
}

void DBTree::insert(int value)  {
    if (IsLeaf()) {
        if (values.size() < 2)
            values.add(value)
        else  {
            int index = GetIndexForValue(value);
            children[index] = new DBTree;
            children[index].values.add(value);
        }
    } else {
        Int index = GetIndexForValue(value);
        If (!children[index])
            children[index] = new DBTree;
        children[index]->insert(value);
        }
}
```

5 points

- 2 points: add value if at a leaf and leaf has fewer than 2 values
    - 1 points: checks if it's a leaf
    - 1 points: checks if it has fewer than 2 values
- 2 points: determines proper index for value (eg: GetIndexForValue())
- 1 points: creates a new DBTree if the corresponding child pointer is NULL

2c)

```
int DBTree::GetChildIndex() {
   for (int i = 0; i < 3; i++) {
      if (children[i])
         return i;
   }
}
int DBTree::GetChildToFollow(int value) {
   if (value < values[0] && children[0])
      return 0;
   if (value < values[1] && children[1])
      return 1;
   if (children[2])
      return 2;
   return -1;
}
void DBTree::remove(int value) {
   for (int i = 0; i < values.size(); i++) {
      if (values[i] == value) {
         values.removeAt(i)
         index = i;
         break;
      }
   }
   if (index != -1) {
      if (IsLeaf())
         values.removeAt(index)
      else {
         int childIndex = GetChildIndex();
         int promotedValue;
         if (childIndex > index)
            promotedValue = ExtractSmallest();
         else
            promotedValue = ExtractLargest();
         values[index] = promotedValue;
      }
   } else {
      int childIndex = GetChildIndexToFollow();
      if (childIndex == -1)
         Error("Value not found");
      children[childIndex]->remove(value);
   }
}
```

7 points

- 1 point: Removes value when it's found
- 4 points: Value found at interior node
    - 2 points: properly promotes a value
    - 2 point: shifts over existing value if necessary

- 2 points: Value not found at current node
    - 2 point: finds the proper index to keep looking and checks if child is NULL

2d)

```
int DBTree::ExtractSmallest() {
   int smallest = values[0];
   if (children[0]) {
      smallest = children[0]->ExtractSmallest();
      if (children[0]->values.size() == 0)
         delete children[0];
      return smallest
   }
   if (IsLeaf()) {
      values.removeAt(0);
      return smallest;
   }
   int childIndex = GetChildIndex();
   int promotedValue;
   if (childIndex > 0)
      promotedValue = ExtractSmallest();
   else
      promotedValue = ExtractLargest();
   if (childIndex == 2)
      values[0] = values[1];
      values[1] = promotedValue;
   else
      values[0] = promotedValue;
   return smallest
}
```

8 points

- 2 points: Checks left child
    - 1 point: Checks if null
    - 1 point: Recursively calls ExtractSmallest() on left child
- 1 point: smaller value not found via recursive call, and node is a leaf
    - 1 point: If node is a leaf, removes value from values
- 5 point: smaller value not found via recursive call, and node is an interior node
    - 1 point: finds a child to promote value from
    - 2 point: extracts a value from child pointer
    - 1 point: swaps values if necessary
    - 1 point: adds promoted value to values

3)

```
Set<treeT *> BuildAllBSTs (Vector<char> & sortedValues) {
   Set<treeT*> bsts;
   BuildAll(sortedValues,bsts,NULL);
   return bsts;
}

void BuildAll(Vector<char> values, Set<treeT*> &bsts, treeT* treeSoFar) {
   if (values.size() == 0) {
      foreach (treeT *root in bsts) {
         if (treesEqual(root,treeSoFar))
            return;
      }
      bsts.add(treeSoFar);
   }
   for (int i = 0; i < values.size(); i++) {
      Vector<char> newValues = values;
      char ch = values[i];
      newValues.removeAt(i);
      treeT *root = CloneTree(treeSoFar);
      Add(root,ch);
      BuildAll(newValues,bsts,root);
   }
      //Note: you would want to delete treeSoFar here, but we didn't require
      //   it for this problem
}

treeT *CloneTree(treeT *root) {
   if (root == NULL)
      return NULL;
   treeT *newRoot = new tree;
   newRoot->value = root->value;
   newRoot->left = CloneTree(root->left);
   newRoot->right = CloneTree(root->right);
   return newRoot;
}

bool TreesEqual(treeT *root1, treeT *root2) {
   if (root1 == NULL && root2 == NULL)
      return true;
   if (root1 == NULL || root2 == NULL)
      return false;
   if (root1->value != root2->value)
      return false;
   return TreesEqual(root1->left, root2->left) &&
         TreesEqual(root1->right,root2->right);
}
void Add(treeT *&root, char value) {
   if (root == NULL) {
      root = new treeT;
```

```
      root->value = value;
      root->left = root->right = NULL;
      return;
   }
   if (value < root->value)
      Add(root->left,value);
   else
      Add(root->right,value);
}
```

25 points:

- 4 points: Base Case – if no more values, adds BST to Set<treeT*>
- 3 points: iterates over all remaining values
- 7 points: Properly clones the BST currently being constructed
- 5 points: adds value to cloned BST
- 4 points: recursively builds up BST
- 1 point: performs recursion properly
- 1 point: initializes node children pointers to NULL

Note: The solution above actually creates many copies of the same tree, which necessitates the TreesEqual() check. We didn't mark down for not doing this because Aubrey forgot to do this when he wrote the solution for this problem =).

4)

```
void PrintAllBridges(Set<nodeT *> & allNodes, Set<arcT *> & allArcs) {
   foreach(arcT *arc in allArcs) {
      nodeT *start = arc->a;
      nodeT *end = arc->b;
      if (!PathExists(start,end,arc) {
         cout << arc->name << endl;
      }
   }
}

bool PathExists(nodeT *start, nodeT *end, arcT *arc) {
   Queue<nodeT*> toExpand;
   Set<nodeT*> visited;
   toExpand.enqueue(start);
   while (!toExpand.isEmpty()) {
      nodeT *next = toExpand.dequeue();
      if (visited.contains(next))
         continue;
      visited.add(next):
      foreach (arcT *nextArc in next->arcs) {
         if (nextArc == arc)
            continue;
         if (nextArc->a == end ||
            nextArc->b == end)
```

```
            return true;
         if (nextArc->a == next)
            toExpand.enqueue(nextArc->b);
         else
            toExpand.enqueue(nextArc->a);
      }
   }
   return false;
}
```

20 points:

- 2 points: iterates over all arcs
- 18 points: Pathfinding function.
    - 2 points: Has a Queue of nodes to expand
    - 2 points: Has a Set of visited nodes
    - 2 points: keeps expanding nodes while the Queue is not empty
    - 2 points: checks if dequeued node is in visited set
    - 2 points: adds node to visisted set
    - 2 points: iterates over node's arc set
    - 4 points: skips arc if it's the arc we're considering removing
    - 2 points: adds endpoint of arc to Queue

5a) This is a bad hash function because for any given string, any permutation of the string will have the same hash value.

3 points:

- 3 points: Give them the points if they have any sort of answer that has to do with permutations of the same string having the same hash value

5b) There were 2 points we were thinking of were:

- The Lexicon can perform containsPrefix() in O(d) time (where d is the length of the prefix you are looking for). For a Set<string>, performing this function would run in O(N)
- A Lexicon of the English language can take up much less memory than a Set<string>, due to the techniques we covered in class.

6 points

- 3 points: If they mention that the Lexicon has containsPrefix()
- 3 points: If they mention that the Lexicon would be smaller than the Set<string>

5c) Because we aren't concerned with finding the shortest path, we should use Depth First Search. Because the graph is densely connected, if we used Breadth First Search our Queue of nodes to expand and out Set of visited nodes would grow to be very large, but with Depth first search we don't need to maintain these data structures.

6 points

- 2 points: They say they would use DFS

- 4 points: Their justification for using DFS. It doesn't have to be exactly what is written in the criteria, but it should at least mention the fact that significantly less memory would be required for DFS, and that we don't need to use BFS because we aren't looking for a shortest path.
    - 1 points: Mention they don't need to perform BFS since we aren't concerned with finding the shortest path
    - 3 points: Mention that less memory is required