Assignment 3

Nicholas Maisel

```
1  from Queue import Queue
2  def Bfs(graphList, target):
3      target = int(target)-1
4      Q = Queue()
5      Q.enqueue(graphList[target]) # Takes the first Vertex and adds it to Queue
6      graphList[target].mark()  # Marks target
7      while Q.isEmpty() != True:
8          v = Q.dequeue()
9          if v == None:
10             break
11         print(v.valueOfNode.vid)
12         v.valueOfNode.mark()
13         for w in v.valueOfNode.adjList:
14             if w.marked == False:
15                 w.mark()
16                 Q.enqueue(w)
```

The run time for the breadth-first traversal is quite easy to understand. If we look at line 5, we see that for each call of the Bfs function, we have to enqueue the target vertex. Line 6 marks the vertex and ensures that the target vertex is marked after it is enqueued. Then, on line 7 we ensure that there are still vertices left in the queue. If there are we continue and find each adjacent vertex and add them to the queue. One of the more important lines, especially when analyzing the run time, is line 15, each time we enqueue a vertex we mark it, ensuring it is not enqueued more than one time. Therefore we know that each node it only visited one time suggesting O(n). The enqueue and dequeue operations are O(1) and therefore in total cost O(V) where V is the number of vertices. And since the algorithm only iterates over the adjacency lists one time each, it take O(E) where E is the number of edges. Therefore, breadth first traversal takes O(E) + O(V) –> O(E + V).

```
1  def Dfs(graphList, target):
2      target = int(target)-1
3      graphList[target].mark()
4      print(graphList[target].vid)
5      for w in graphList[target].adjList:
6          if not w.marked:
7              Dfs(graphList,w.vid)
```

The run time for the depth-first traversal is very similar to the run time of the breadth-first traversal. The algorithm works by looking at the root node, and finding its adjacent nodes. Then, it takes those nodes and looks at the first

one's adjacent nodes, this repeats and can be seen on line 7, where the function calls itself recursively. Since each node is only looked at once, because just like in breadth-first traversal, the nodes are marked once viewed, we can say that the algorithm runs in O(V) where V is the number of vertices. But, we also have to travel along each edge twice when traversing the tree because if we have two nodes (A,B) both nodes have each other in their respective adjacency lists. Therefore we have O(V + 2E), the two because the graph is undirected. But we can simply drop that constant to get O(V + E).

```
1  from TreeNode import TreeNode
2  import random
3
4  comparison_count = 0
5
6  class BSTree:
7      def __init__(self):
8          self.root = None
9
10     def treeInsert(self, val):
11         if self.root is None:
12             self.root = TreeNode(val)
13         else:
14             self.insertTreeNode(self.root, val)
15
16     def insertTreeNode(self,currentNode, val):
17         if(val <= currentNode.vid):
18             if(currentNode.left):
19                 self.insertTreeNode(currentNode.left, val)
20             else:
21                 currentNode.left = TreeNode(val)
22
23         elif(val > currentNode.vid):
24             if(currentNode.right):
25                 self.insertTreeNode(currentNode.right, val)
26             else:
27                 currentNode.right = TreeNode(val)
28
29     def findInTree(self, val):
30         return(self.findTreeNode(self.root,val))
31
32     def findTreeNode(self, currentNode, val):
33         global comparison_count
34         comparison_count +=1
35         if(currentNode == None):
36             return(False)
37         elif(val == currentNode.vid):
38             return(True)
39         elif(val < currentNode.vid):
40             return(self.findTreeNode(currentNode.left, val))
41         else:
42             return(self.findTreeNode(currentNode.right, val))
```

The worst run time for the binary search tree is O(n). The only way this could

occur is in the following scenario: Suppose you had a sorted list from least to greatest and began building a binary tree from the first node forward. This would result in a tree that looks like a long list of linked items. If you were to search for the largest value in the tree, you would have to iterate over each node before finally reaching the last node. However, this is a very specific case. If the binary search tree were balanced, meaning there are the same amount of nodes on each side of the tree, than the worst case scenario would be O(log n), because for each level in the binary tree, we split the remaining results by two.

After using magicitems to create a binary search tree, and searching for 42 randomly selected items, I found that the average number of comparisons required to find each of the items out of 666 items was just 10.19047619047619. While the number is too high to be O(log n), this still makes sense because our tree is unbalanced.