Jackson Bence-Collin (1169431) – DoA Assignment 2 Written Submission

**Problem 1**

**Task 1:**

function DFS(V,E)

mark each node in V with 0

count 🡨 0

for each v in V do

if v is marked 0 then

count 🡨 count + 1

DFS\_explore(V,E,v)

return count

function DFS\_explore(V,E,v,)

mark node v with 1

for each node in V do

if node is marked 0 and edge(v,node) then

DFS\_explore(V,E,node)

NB: edge(v,node) indicates that an edge exists between v and node.

The main operation of this algorithm is evaluating vertex status and iterating a depth first search, both of which take O(1) time. The algorithm in this specific pseudo will visit every vertex once, and every edge twice due to the bi-directional nature, resulting in a time complexity of (ignoring negligible constants) O(V+E).

**Task 5:**

NB: For simplicities sake assume the pseudo code from Task 1 may be accessed here.

function BF\_crit(V,E)

og\_subnetwork = DFS(V,E)

for each v in V do

V\_removed = V \*NB This must be a deep copy

remove v from V\_removed

E\_removed = E \*NB This must be a deep copy

for each edge in E do

if edge(v,~)

remove edge from E\_removed

subnetwork\_count = DFS(V\_removed, E\_removed)

if subnetwork\_count > og\_subnetwork then

mark v as a critical server

return all v marked as a critical\_server

NB edge(v,~) means that there exists an edge from, or to v.

First, we iterate through every vertex, then for every vertex we navigate every edge totalling O(VE). Then, within each vertex check, a DFS is performed, which in itself tallies a time complexity of O(V(V+E)), and summing these two produces O(V^2 + 2VE). This quadratic algorithm is also assuming a vertex and edge removal cost of 0, which is implausible.

**Task 6:**

function crit(V,E)

mark each node in V with 0

mark each node in E with 0

count 🡨 0

for each v in V do

if v is marked 0 then

children[v] 🡨 0

count 🡨 count + 1

DFS\_explore(V,E,v)

if children[v] >= 1 then

mark v as a critical\_server

return all v marked as critical\_server

function DFS\_explore(V,E,v,count)

push[v] 🡨 count

HRA[v] 🡨 count

for each edge in E do

if edge.end is marked with 0 then

children[v] 🡨 children[v] + 1

mark edge with 1

DFS\_explore(V,E,edge.end)

If HRA[edge.end] < HRA[v]

HRA[edge.end] 🡨 HRA[v]

If HRA[edge.end] >= push[v]

mark v as a critical\_server

else if edge is marked with 0 then

If HRA[v] > push[edge.end]

HRA[v} = push[edge.end]

NB: edge(v,node) indicates that an edge exists between v and node.

Similar to Task 1, this algorithm will iterate through each edge once for each node, and then again for every node that is a child element, as well as accessing each vertex once. In addition to this, there are numerous comparisons that may be reverted to a constant, so when removing these negligible factors, the time complexity of the above becomes O(V+E).

**Problem 2**

1. Given the task requirements stipulate that there is lots of memory at our disposal, insertions are not critical, deletions are irrelevant, there is no upper bound on the length of key, and search is prioritised, a preferential data structure could be a hash table. Here, searching remains at a complexity of O(1) given no clustering from efficient spreading of records resulting from a hash function that is based on the modulus of a prime number.
2. Given the task requirements stipulate searching has reasonably fast performance, most operations are searching and accessing records, and that not much memory is available, a binary search tree would be helpful here. This data structure has many implementations, of which AVL trees with search time complexity O(n log n) are best suited for this situation. This is due to their preference over alternative self-balancing trees such as red-black trees as they provide faster lookups.
3. Given the task requirements stipulate a need for parallel access to contiguous records in a FHD using a single reading operation, B-trees would be of preference here. This is due to B-trees having a cheap cost for searches in contiguous arrays, are ideal for their reduced height compared to 2-3 trees and are very efficient when searching for a range of values. Furthermore, they mitigate the risk of requiring more disk access and therefore cost, by pre-emptively splitting full nodes during insertion.
4. Given the task requirements stipulate that the search time slows with more records and that 85% of the allocated space is utilised, a hash table could have been implemented here. Using linear probing or separate chaining, a bad value of alpha – the load performance coefficient, at 0.85 in this example, is nearing the threshold value of 0.9 at which the function degrades in performance, as present in University of Mysteryport’s review. The remaining space in the allocated memory might be due to a poor hashing algorithm, where the data is not spread evenly, subsequently creating clusters hindering performance.

**Problem 3**

1. A potential Hash function that causes Insertion Sort to keep the original array unchanged is H(A,j) = j. Using this partitioning algorithm that is independent to the element value, provides the same index that was initially held and thus, the order of elements can never vary from the original.
2. A potential Hash function that causes Insertion Sort to always run in the worst case time complexity is H(A,j) = -j. Using this partitioning algorithm that is independent to the element value, provides an index that is always less than those of the previous and as such, the element will continue being shifted until it reaches the front of the sorted portion of the array, ie: the first element becomes the last and vice versa. Thus, the function, when repeated, will produce the original array on every second application, never reaching a sorted array.
3. A potential Hash function that causes Insertion Sort to sort the array in reverse is H(A,j) = -A[j]. Using this partitioning algorithm, every element, once hashed, will be sorted in decreasing order. This occurs as the largest element in the array, when given a negative magnitude, will be less than every other element and similarly, the smallest element in the array will be closest to 0 when made negative and as such, will be the final value in the array sorted by this hash function.