Assignment 2 Written Problems

Problem 1

Task 1

```
function COMPUTECOMPONENTS(G)
  mark each node in V with 0
  count ← 0
  for each v in V do
    if v is marked 0 then
      count ← count + 1
      DFSEXPLORE(v)
  return count


function DFSEXPLORE(v)
  mark[v] ← 1
  for each edge (v, w) do
    if w is marked with 0 then
      DFSEXPLORE(w)
```

The time complexity of this algorithm depends on the time complexity of the inner function DFSEXPLORE, which in turn depends on the data structure which is used to represent the graph. If an adjacency matrix is used to represent the graph, then for each v in V we traverse a row of length V in the adjacency matrix. Therefore, the time complexity in this case depends on the number of vertices squared and would be $\Theta(|V|^2)$. If an adjacency list, is used to represent the graph, then we traverse the adjacency list in which the sum of the sizes of the adjacency lists is $|E|$ (or $|2E|$ in an undirected graph) and in which there is an adjacency list for each node in V. Therefore, the time complexity in this case depends on the number of vertices and the number of edges and would be $\Theta(|V| + |2E|) = \Theta(|V| + |E|)$. The space complexity is $\Theta(|V|)$ as we store $|V|$ marks in the mark array.

Task 5

```
function COMPUTECRITICALVERTICES(G)
    regularCount ← COMPUTECOMPONENTS(G)
    numCritical ← 0
    for each v in V do                    ▷ (remove each vertex once and count the number of components)
        mark each node in V with 0
        count ← 0
        for each v in V besides the removed vertex do
            if v is marked 0 then
                count ← count + 1
                DFSEXPLORE(v)
        if count > regularCount then      ▷ (this must be a critical vertex if there are now more connected components)
            criticalList[numCritical] ← v
            numCritical ← numCritical + 1
    return criticalList


function DFSEXPLORE(v)
    mark[v] ← 1
    for each edge (v, w) except for the removed edges do
        if w is marked with 0 then
            DFSEXPLORE(w)
```

Similarly to the algorithm in Task 1, the time complexity of this algorithm depends on the time complexity of the inner function DFSEXPLORE which depends on the data structure which is used to represent the graph. Assume an adjacency list is used to represent the graph (which we have seen entails a time complexity of $\Theta(|V| + |E|)$). We must traverse the graph $|V| + 1$ times (as we count the number of components without removing any vertices 1 time and count the number of components with one vertex removed $|V|$ times). Each traversal requires approximately $|V| + |E|$ operations. Therefore the time complexity is approximately $\Theta(|V| * (|V| + |E|)) = \Theta(|V|^2 + |V|*|E|)$. Therefore, if an adjacency matrix was used to represent the graph then we should have time complexity $\Theta(|V|*|V|^2) = \Theta(|V|^3)$.

Task 6

```
function COMPUTECRITICALVERTICES(G)
    numCritical ← 0
    mark each node in V with 0
    set the parent of each vertex to -1
    HRA of each vertex v is initialised to v
    count ← 0
    for each v in V do
        if v is marked 0 then
            DFSEXPLORE(v)
    for each parent, child (v, w) do                    ▷ (v is the parent, w is the child)
        if v is a root of a subtree with > 1 child or mark[HRA[w]] >= mark[v] then
            criticalList[numCritical] ← v
            numCritical ← numCritical + 1
    return criticalList

function DFSEXPLORE(v)
    count ← count + 1
    mark[v] ← count                                     ▷ (this count represents the push order of vertex v)
    for each edge (v, w) do
        if w is marked with 0 then
            parent[w] ← v
            DFSEXPLORE(w)
        else if mark[HRA[w]] < mark[v] and w is not the parent of v then    ▷ (we have found a back edge)
            HRA[v] ← w
            i ← parent[v]
            while mark[HRA[v]] < mark[HRA[i]] then      ▷ (all ancestors' HRAs with larger push orders can be updated)
                HRA[i] ← HRA[v]
                i ← parent[i]
```

Task 6 (cont.)

If an adjacency list, is used to represent the graph, then we traverse the adjacency list in which the sum of the sizes of the adjacency lists is |E| (or |2E| in the case of an undirected graph as we consider each edge exactly twice) and in which there is an adjacency list for each node in V (i.e. there are |V| adjacency lists which altogether contain |E| or |2E| edges). The complexity of backtracking through the parent array to update ancestor HRAs depends on the number of edges |E|. We also iterate over the parent array at the end of the algorithm to find critical vertices. Therefore, the time complexity in this case depends linearly on the number of vertices and the number of edges and would be O(|V| + |2E| + |E| + |V|) = O(|2V| + |3E|) = O(|V| + |E|).

Problem 2

a) In this scenario, I believe a hash table would be the most appropriate data structure as the main requirement is for searching to be as fast as possible and a hash table can achieve Θ(1) search time. Additionally, one of the drawbacks of hash tables is generally their higher memory requirements but the university is willing to spend extra money to make up for this.

b) In this scenario, I believe an AVL self-balancing tree would be the most appropriate data structure as AVL trees are better than red-black trees if searches are more frequent. Furthermore, self-balancing trees guarantee Θ(logn) search with less memory requirement than a hash table and tend to be better when the dictionary is fully in memory (or in an FHD in this case).

c) In this scenario, I believe a B-tree implementation would be most appropriate as again the memory requirement is less than that of a hash table. A high order B-tree could be used to take advantage of the parallel access to contiguous records and the 100x faster search in the RAM cache.

d) The university is most likely using a regular Binary Search Tree (as opposed to a hash table with clusters due to the 15% unused storage). The performance of their system is degrading when new items are being searched as the Binary Search Tree is most likely extremely unbalanced and new records are being inserted at / near the leaves which will take increasingly longer amounts of time to reach in a search.

Problem 3

a) **function** HASH(A, i)
return i

This keeps the original array unchanged as j+1 will never be < j and so no swaps will occur.

b) **function** HASH(A, i)
return -i

This causes Insertion Sort to always run in the worst-case complexity as –j - 1 will always be < -j and so the maximum number of swaps will occur.

c) **function** HASH(A, i)
return – A[i]

This causes Insertion Sort to sort the array in reverse order as the inequality in the while loop is reversed.

$-A[j+1] < -A[j] \iff A[j+1] > A[j]$

Therefore, swaps will now occur when an element is bigger than its previous element and so we will end up with a sorted array in descending / reverse order.