COMP20007 – Assignment 2

Problem 1

Task 1 ***(Code adapted and modified from Lecture Slides 7, pg. 7)***

```
function CONNECTEDCOMPONENTS (⟨V,E⟩)
    mark each node in V with 0
    count ← 0
    num_components ← 0
    for each v in V do
        if v is marked 0 then
            num_components ← num_components + 1
            DFSEXPLORE(v)
    return num_components
function DFSEXPLORE (v)
    count ← count + 1
    mark v with count
    for each edge (v,w) do
        if w is marked with 0 then
            DFSEXPLORE(w)
```

In function CONNECTEDCOMPONENTS, the cost of marking each node with 0 and looping through V to call depth-first search are both $\Theta(|V|)$, for looping through each vertex in the list. In DFSEXPLORE, called exactly once per node, we loop through its neighbours to further explore. Using adjacency list (data structure chosen as it's more efficient than adjacency matrix) to find neighbours, the sum of each node's neighbour is around the number of edges, taking $\Theta(|E|)$ for traversal in adjacency list, yielding a $\Theta(|V| + |E|)$ overall complexity.

Task 5 ***(Reuses functions from Task 1)***

```
function CRITICALNODES (⟨V,E⟩)
    init(array)                                   (array to store critical nodes)
    subnet_before ← CONNECTEDCOMPONENTS (⟨V,E⟩)
    for each v in V do
        V' ← V \ {v}, E' ← E
        for each edge (v,w) do
            E' ← E' \ {(v, w)}
```

```
        subnet_after ← CONNECTEDCOMPONENTS (⟨V',E'⟩)
        if subnet_after > subnet_before then
            add(array, v)                          (adds v to array)
    return array
```

Computing original subnetworks takes $\Theta(|V| + |E|)$ time. Assuming constant-time removal of each node and edge, it takes in total $\Theta(|V| + |E|)$ time for removals as each node and edge is removed exactly once (adjacency list to store edges), each call to CONNECTEDCOMPONENTS for each node removed takes approximately $\Theta(|V| + |E|)$ time. We have $|V|$ nodes, so removing each node and computing the number of subnetworks takes $\Theta(|V| \cdot (|V| + |E|))$. Hence the algorithm takes approximately $\Theta(|V|^2 + |V||E|)$ in total.

Task 6

```
function CRTITICALNODES (⟨V,E⟩)
    mark each node in V with 0
    count ← 0, init(HRA)          (array to store push order of HRA of each node)
    for each v in V do
        if v is marked 0 then
            DFSEXPLORE(v)
    return list of all critical nodes
function DFSEXPLORE (v)
    count ← count + 1
    mark v and HRA of v with count
    for each edge (v,w) do
        if w is marked with 0 then
            DFSEXPLORE(w)
            HRA[v] ← min(HRA[v], HRA[w])
            if v is a root node and v has more than 1 child then
                mark v as critical
            else if v is not a root node and HRA[w] ≥ count[v] then
                mark v as critical
        else if w is not parent of v then
            HRA[v] ← min(HRA[v], HRA[w])
```

This algorithm uses depth-first search and uses adjacency list to store edges. Every v is traversed in CRTITICALNODES to check and (potentially) perform DFS, taking $\Theta(|V|)$ time.

For DFSEXPLORE, it's performed exactly once per node, and traverses its neighbours in adjacency list once, hence the total search is Θ(|E|) for |V| nodes. We use arrays to keep track of each node's parents and number of children, as well as the push order of their HRA's, taking constant time in their respective operations. In conclusion, the complexity is indeed Θ(|V| + |E|).

Problem 2

a. Array(s) of pointers to records with ID as index. This is because random access allows fast searching in array and records, deletion is not a concern, and insertion is done through acquiring larger FHDs in passes for new array spaces if necessary.

b. Self-balancing Binary Search Tree with ID as keys, and records as nodes. Balanced tree allows Θ(log n) searching and access performance which is reasonably fast, and need for costly insertion and subsequent balancing is infrequent (during admission), while saving costs of acquiring new FHDs.

c. B-trees (e.g., order 6) with ID as keys, and multiple records per node. Again, it allows fast (reduced height) logarithmic search and access, with rare costly insertion and consequent splitting operation, and FHD allows reading of contiguous records in a node, while storing these results in cache allows even faster search.

d. Hash table with separate chaining and bad hash function, where a lot of records are hashed to the same cell, leaving some cells empty. Collisions are handled badly, with new records added to the end of cell's long linked list, it takes increasingly longer to search them.

Problem 3

a. HASH(A, x) = 0

This hashes any value of x to 0, the same value. Since HASH(A, j+1) has to be less than HASH(A, j) for the swap to take place, it will never happen if they both return the same value, leaving the original array unchanged.

b. HASH(A, x) = -x

This makes x negative. As a result, HASH(A, j+1) = -(j+1) is always less than HASH(A,j) = -j causes the swap to happen every time for every pair of (i,j) as long as j is non-negative, $(\sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1) = \frac{n(n-1)}{2}$ swaps always giving Θ(n^2) complexity, and probably an unsorted array.

c. HASH(A, x) = -A[x]

This makes the element of A[x] negative. As a result, swap will happen when -A[j+1] < -A[j], in other words, when A[j+1] > A[j], swapping larger values to the front and smaller to the back for each (i,j), sorting the array in reversed order.