PROBLEM 1

Problem 1 - Task 1

```
G = (V, E)
V = number of vertices, vertices numbered from 0..V-1
E = Array of edges in form of (start, end)

function getComponents(G)
    boolean checkedArray[0..V-1] <-- FALSE
    components <-- 0

    for i = 0 to numV - 1 do

        if checkedArray[i] == FALSE then
            components += 1
            getConnections(G, i, checkedArray)

    return components

function getConnections(G, i, checkedArray)
    checkedArray[i] <-- TRUE
    numE from E    // number of edges

    for i=0 to numE - 1 do
        if i in edgeStart or edgeEnd then
            NBR from edge  *// Other side of edge, that isnt i*
            if checkedArray[NBR] == FALSE then
                getConnections(G, NBR, checkedArray)
```

Our worst case scenario is that we have 1 component, and that each vertice is connected to exactly one other vertice, a straight line of connected vertices. As we would then have to call getConnection on every vertice and check the entirety of the edge list for every iteration, giving us O(V * numE) time complexity

Problem 1 - Task 5

```
G = (V, E)
V = number of vertices, vertices numbered from 0..V-1
E = Array of edges in form of (start, end)

function getCritical(G)
    criticalServers[]

    components = getComponents(G, NULL)

    for i=0 to numV - 1 do
        curComponents = getComponents(G, i)

        if curComponents > components then
            append i to criticalServers

    return criticalServers


function getComponents(G)
    boolean checkedArray[0..V-1] <-- FALSE
    components <-- 0

    for i = 0 to numV - 1 do

        if checkedArray[i] == FALSE then
            components += 1
            getConnections(G, i, checkedArray)

    return components

function getConnections(G, i, checkedArray)
    checkedArray[i] <-- TRUE
    numE from E    // number of edges

    for i=0 to numE - 1 do
        if i in edgeStart or edgeEnd then
            NBR from edge  // Other side of edge, that isnt i
            if checkedArray[NBR] == FALSE then
                getConnections(G, NBR, checkedArray)
```

This has a similar time complexity to Task 1, as it is just looping through and calling the algorithm created in Task1 numVertices + 1 times, once for the initial check of how many components, and one for checking the number of components when each node is disconnected. So the worst case time complexity for this algorithm becomes O((V + 1) * (V * numE))

Problem 1 - Task 6

```
Needs Global access to these
G = (V,E), V = number of vertices, E = Adjacency Array
criticalNodes[0..V-1] <-- FALSE
pushOrder[0..V-1] <-- NULL
HRA[0..V-1] <-- INFINITY
parent[0..V-1] <-- NULL
visited[0..V-1] <-- FALSE
pushCount = 0


// This algorithm returns an Array criticalNodes which contains true or
// false values of if a node is critical or not

function getCritical()
    for i=0 to V-1 do
        if visited[i] == FALSE
            DFS(i)

    return criticalNodes


// DFS search from current Node
function DFS(curNode)
    pushCount++
    visited[curNode] == TRUE
    pushOrder[curNode] = HRA[curNode] = pushCount
    children = 0

    for i=0 to V-1 do
        if E[curNode][i] == 1
            if visited[i] == FALSE
                children = children + 1
                parent[i] = curNode
                DFS(i)

                HRA[curNode] = min(HRA[curNode], HRA[i])

                if parent[curNode] == NULL and children > 1 then
                    criticalNodes[curNode] = TRUE

                if parent[curNode] != NULL and HRA[i] >= pushOrder[curNode] then
                    criticalNodes[curNode] == TRUE

            else if parent[curNode] != i then
                HRA[curNode] = min(HRA[curNode], pushOrder[i]
```

This algorithm runs in O(V + E) time as the algorithm only ever considers each edge and node once, and never has to double check a Node or edge due to using the back edge to check and store for ancestors in the graph. Therefore we get a worst and average time of O(V + E).

PROBLEM 2

Problem 2 - A

```
Hash Table - Seperate Chaining
The university wants a system that is fast for searching and to use Student ID's as
keys. Using a large enough Hash Table with a sufficient Hash function, would give us
average case O(1), although worst case can be O(n) given a poor hash function and a
low amount of storage. Though in this case they can choose to upgrade storage and
modify the Hash function in order to get less collisions. If student numbers were
contiguous and non-finite then an Identity Hash would be a better option giving us
average and worse case of O(1).
```

Problem 2 - B

```
Sorted Array
The university wants a system that is fast for searching and has a lower memory
cost, though they dont care too much about insertion and deletion speed. Due to this
a sorted Array is the better option as it has O(log(n)) search time and is more
memory efficient when compared to a Hash Table that is too large for the dataset, as
we dont end up with any empty index's. Although if we can assume that the student
ID's are contigious from 0 then an Identity Hash will be the better option.
```

Problem 2 - C

```
B-Tree
Due to the nature of allowing faster operation with a ram Cache, if we use a B-Tree,
the tree will have less depth then a binary tree, and we can use the RAM Cache to
store parallel record to the current search object to speed up search of these
parallel records.
```

Problem 2 - D

```
Hash Table - Linear Probing
Based on the case of newly inputed records having a slower access time, despite the
fact their storage is at 85%, I would assume they use a Hash Table with linear
probing and a less then ideal Hash function. With this case newly inputed records
have a higher chance of collision with records already in the Hash Table and hence
being inserted further down the array, and thus when accessing the newer records
they have to first get the Hash of the ID and then search down the array, hence
making newer records take longer to search for.
```

Is only just over half a page in office, just looks big here

Problem 3

Problem 3 - A

```
HASH(A, j) = 1

If the hash always returns 1, HASH(A, j+1) is never less than HASH(A, j), and hence
no items ever get swapped and the original array remains unchanged
```

Problem 3 - B

```
HASH(A, j) = -j

If the hash returns the negative of the index, it ensures that a swap always occurs,
hence running on worst case time complexity
```

Problem 3 - C

```
HASH(A, j) = -A[j]

If HASH(A, j) = A[j] the psuedocode will run as a normal insertion sort, though by
making the HASH return the -A[j] we can swap which value is greater, and hence
performing the sort in reverse.
```