Problem 1

Task 1:

```
function getConnectedSubnets(V, E)
    subnetCount ← 0
    for each v in V do
        visited[v] ← 0
    for each v in V do
        if visited[v] is 0 then
            exploreNetwork(v, V, E)
            subnetCount ← subnetCount + 1
    return subnetCount

function exploreNetwork(v, V, E)
    visited[v] ← 1
    for each w adjacent to v do
        if visited[w] is 0 then
            exploreNetwork(V, E)
```

Complexity analysis: In this algorithm, we are essentially doing DFS traversal over the graph. Using adjacency lists, the time complexity becomes O(V + E).
During DFS, for each node we explore its adjacent nodes by traversing its adjacency list once, which gives rise to O(V). Since we are dealing with undirected graphs, each edge would be traversed twice – once in adjacency list of both the end nodes, adding a factor of O(2E). Thus, the overall complexity becomes **O(V) + O(2E) ~ O(V + E).**

Task 5:

This code makes use of the function **`getConnectedSubnets(V, E)`** from Task 1 which gives the number of connected subnetworks in a graph.

```
function findCriticalPoints(V, E)
    for each v in V do
        iscritical[v] ← 0
    originalSubnetCount ← getConnectedSubnets(V, E)

    for each v in V do
        // remove all edges (v,w) associated with v
        E′ = E \{(v, w)}
        newSubnetCount ← getConnectedSubnets(V\{v}, E′)
        if newSubnetCount > originalSubnetCount then
            iscritical[v] ← 1
```

Complexity analysis: This algorithm makes use of the one developed in task 1. It removes each vertex and finds the number of connected components after its removal. If it is greater than the original number of components, it is a critical vertex. So, the function from task 1 (using adjacency lists) is called 1+V times, once to find the original number of components and V times for V vertices. SO the complexity comes down to O((V+1)(V+E)) ~ O(V(V+E)) = **O(V^2 + VE)**

Task 6:

```
function CriticalNodes(V, E)
      ordercount ← 0
      for each u in V do
            visited[u] ← 0
            order[u] ← -1
            hra[u]  ← -1
            iscritical[v] ← 0
            parent[v] ← -1
      for each u in V do
            if visited[u] is 0 then
                  findCriticalVertices(u, V, E, visited, order,parent,
                                          hra, iscritical, ordercount)
      return iscritical


function findCriticalVertices(u, V, E, visited, order, parent,
                                    hra, iscritical, ordercount)
      mark u as visited
      children ← 0
      ordercount ← ordercount + 1
      order[u] ← ordercount
      hra[u] ← ordercount

      for each (u, v) ∈ E do
            if visited[v] is 0 then
                  children ← children + 1
                  parent[v] ← u
                  findCriticalVertices(u, V, E, visited, order,parent,
                                          hra, iscritical, ordercount)

                  hra[u]←min(hra[u], hra[v])

                  if parent[u] is -1 and children > 1 then
                        iscritical[u] ← 1
                  if parent[u] is not -1 and hra[v] >= order[u] then
                        iscritical[u] ← 1
            else if v is not parent[u] then
                  // back edge found
                  hra[u]←min(hra[u], order[v])
```

Complexity Analysis: The major operation happening in the above algorithm is DFS traversal and the other operations include populating the arrays. As compared to the algorithm is task 5 where we were doing DFS traversal (V+1) number of times, here we are doing it just once using adjacency list structure.
Thus, the overall complexity is the same as that of DFS, i.e., **O(V + E)**

Problem 2:

(a) **Hash Table with Double Hashing** will give the fastest search and insert ($\Theta(1)$ average with a good hash function), assuming the university doesn't need range queries. It fits their requirement as deletion is not required and there is no memory scarcity. The university will do insertions in big passes, so we can always create more buckets and reallocate the hash table if necessary. However, if range queries are a requirement, an AVL Tree would be more suitable.

(b) **AVL Tree** is the most suitable data structure in this case. It has low memory requirements and $O(\log_2 n)$ worst-case complexity for search, insertion and deletion, which is reasonably fast. Since majority operations will be searches, there is no concern about high number of rotations.

(c) B-Trees allow and assist searching a contiguous array, as multiple elements are stored at a node. Hence, using this data structure we can exploit the property of the new FHD that allows "parallel access to contiguous records using a single reading operation". Each node is read and brought to cache to search for the required value, which will be 100x faster with the new FHD.

(d) It is possible that this university is either using a **linked list** or a **Binary Search Tree** that is unbalanced and is starting to degenerate to a linked list with increasing nodes. This would mean that newly added items (towards the end) would take $O(n)$ time to reach, and this would slow down the search for a high number of student records despite having spare memory (as n is very high).

Problem 3:

Condition: `while j ≥ 0 and HASH(A, j + 1) < HASH(A, j) do swap`

(a) HASH(A, i) = i

The above function will leave the array unchanged because HASH(A, j + 1) = j+1 and HASH(A, j) = j. So, $j+1 < j$ will always be false and no swaps will be executed.

(b) HASH(A, i) = -i

The above function will lead to $\Theta(n^2)$ complexity because HASH(A, j + 1) = -(j+1) and HASH(A, j) = -j. Now, $-(j+1) < -j$ will always be true and the swap will always be executed regardless of the actual value of elements at A[j] and A[j+1].

(c) HASH(A, i) = -A[i]

The above function will sort the array in reverse. This is because the hash function essentially multiplies elements by -1, changing the condition from $A[j+1] < A[j]$ to $A[j+1] > A[j]$. Now, the swap will occur if A[j] is smaller than A[j+1], i.e., array will be sorted in descending order.