Task 1

Idea: apply DFS, check whether current vertex has neighbours to explore. The first vertex of a connected component can be located if all of its neighbours have not been explored.

```
Function countSubnets(G(V, E))
    For each v_start in G do
        neighb <- v_end in E
        adjList[v_start][nb++] <- neighb
    For each v in V, do
        Explored[v] <- false
    Count <- 0
    For each v in V, do
        If explored[v] = false, then
            dfsExplore(v)
    return count

Function dfsExplore(v)
    Explored[v] <- true
    Disconnected <- true
    For each w in adjList[v], do
        If explored[w] = true, then
            Disconnected <- false
    If disconnected = true, then
        Count ++
    For each w in adjList[v], do
        If explored[w] = false, then
            dfsExplore(w)
```

Complexity analysis:

Cost for accessing all vertices once: O(V)

Cost for accessing neighbours of each vertex twice by using adjacency list:

- DFS exploring -> O(E)
- Check whether all neighbours of current vertex have been explored -> O(E)

In total: O(V+2E) -> Θ(V+E)

Task 5

Idea: check for every vertex if it is a critical point by removing it and counting the number of connected components.

```
Function findCriticalServers(G(V, E))
    originalSubnets <- countSubnets(G(V, E))
    num <- 0
    for each v in V, do
        outage <- v
        for each e in E, do
            remove e with outage
        currSubnets <- countSubnets(G(V, E))
        if currSubnets > originalSubnets, do
            criticals[num++] <- outage

Function countSubnets(G(V, E))
    For each v_start in G do
        neighb <- v_end in E
        adjList[v_start][nb++] <- neighb
    For each v in V, do
        explored[v] <- false
    count <- 0
    For each v in V, do
        If explored[v] = false, then
            dfsExplore(v)
    return count

Function dfsExplore(v)
    explored[v] <- true
    disconnected <- true
    For each w in adjList[v], do
        If explored[w] = true, then
            Disconnected <- false
    If disconnected = true, then
        count ++
    For each w in adjList[v], do
        If explored[w] = false, then
            dfsExplore(w)
```

Complexity analysis:

As applying DFS to count the number of connected components costs $\Theta(V+E)$ (same pseudocode in task1), we apply DFS counting for each vertex -> $\Theta(V * (V + E))$

In total: $\Theta(V^2 + VE)$

Task 6

Idea: apply DFS, make sure to explore the neighbour with lower ID first. Record the HRA of current vertex as the lowest ID of any of its ancestor if found, otherwise, as itself. Push the vertex to the list, while recording the push order. When exploring reaches the leaf, start popping while recording the HRA of the current vertex as the HRA of its child if the HRA of the child is smaller. After having the push order and HRA of all vertices, go through each vertex and its children to find critical vertices by following the instruction.

```
Function findCritical(G(V, E))
    For each v_start in G do
            neighb <- v_end in E
            adjList[v_start][nb++] <- neighb
            sort(adjList[v_start])
    S <- subnets(V, E)
    plist <- new list
    order <- 0
    numCritical <- 0
    criticals[ ] <- allocate memory
    porder[ ] <- allocate memory
    children[ ] <- allocate memory
    numChild <- 0
    For each v in V, do
        explored[v] <- false
        HRA[v] <- NIL
    // construct DFS trees in each connected component
    For each s in S, do
       For each v in s, do
          If explored[v] = false, then
              dfsFindCriticals (v)
    // go through each vertex again, access their children and find critical servers
    for each s in S, do
        // root of one DFS tree
        if numChild[v0] > 1, then
            criticals[numCritical ++] <- v0
        // go through the rest except for the leaf
        for v_x <- v_1 to v_n-1 in s, do
            for child in children[v_x], do
                if porder[HRA[child]] >= porder[v_x], then
                    criticals[numCritical ++] <- v_x
```

```
Function dfsFindCriticals(v)
    explored[v] <- true
    porder[v] <- order + 1
    order++
    If HRA[v] = NIL, then
        HRA[v] <- v
    // record v's HRA as the smallest server ID of any of its reachable ancestors
    For each w in adjList[v], do
        If explored[w] = true, then
            If w in plist AND w != plist->head AND w < HRA[v], then
                HRA[v] <- w
    Push(v, plist)
    For each w in adjList[v], do
        If explored[w] = false, then
            children[v][numChild++] <- w
            dfsFindCriticals(w)
        // pass HRA of the child of v, if it is smaller, to HRA of v before poping
        if plist->head != v, then
            if HRA[list->head] < HRA[v], then
                HRA[v] <- HRA[list->head]
            Pop(plist)
```

Complexity analysis:

Access all the vertices three times in the worst case:

- DFS exploring all vertices once -> O(V)
- Finding critical servers by going through children of each vertex
  - O(2V) if DFS tree looks like a linked list, almost one vertex has one child
  - O(V) if all vertices share one parent

Access all the neighbours of one vertex twice by using adjacency list:

- DFS exploring neighbours of all vertices once -> O(E)
- Passing smallest server ID of reachable ancestors by going through neighbours once -> O(E)

In total: O(3V + 2E) -> $\Theta(V + E)$

Problem 2

a) Data structure: hash table.

   Searching only takes O(1) in average case meaning it is very fast. Hash table requires a large amount of extra memory which can be provided by large FHDs.

b) Data structure: B-Tree.

   It doesn't require much memory and is still fast with searching – only takes $\Theta(\log n)$ time in average. Moreover, B-tree allows range search and easy access to disks if needed.

c) Data structure: B-Tree

   B-Tree stores data in the disk cache and is very efficient when searching for a range of values (contiguous records). And it also doesn't require much memory and remain fast ($\Theta(\log n)$ time).

d) Data structure: unsorted array

   Since the new record will be added at the end of the array and only linear search can be applied to search the record, which takes $\Theta(n)$ time. The cost of searching new items increases as the records become more.

Problem 3

a) Hash(A, i) = A[i]

   The function returns the value in the array with argument index, which is the same value as the original insertion sort pseudocode.

b) Hash(A[0…n-1], i) = n - i

   Hash(A, j+1) = n - (j + i) would always be smaller than Hash(A, j) = n - j. The complexity of insertion sort would always be $O(n^2)$.

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

   If A[j+1] is larger than A[j], -A[j+1] would be smaller than -A[j], so that the larger value would be turned into a smaller value and swapped towards the start of the array.