COMP20007 DOA Assignment 2 Report

Name: Lang (Ron) Chen

Student ID : 1181506

Question 1

Task 1.

```
Function Task1_1(V, E)
//Returns the number of connected components (subnetworks) in this graph
//Input: an int array V of increasing neighbouring integers starting at 0 denoting each vertex while E is an array of 2 component int arrays denoting edges (the 0th and 1st element of the edge array denote the start vertex and end vertex).
//Output: the number of connected vertices (subnetworks) as an int.

L ← sorted adjacency list of G(V, E)
    /This is an array of int arrays where the int array in the 0th position of the original array hold the sorted neighbouring nodes of the 0th vertex and so forth (assume sorted upon input)
S ← createStack()
n ← length(V)
marked ← int array of size n with all cells initiated to 0

S.Push(V[0])
marker ← 0
subnetworks ← 1

while marker != n do
    c ← S.Pop()
    if c = Null do
        subnetworks ← subnetworks + 1
        for i ← 0 to n-1 do
            if marked[i] = 0 do
                c ← i
                break
    marker ← marker + 1
    marked[c] ← marker
    for j ← length(L[c])-1 to 0 do
        if marked[ L[c][j] ] = 0 and L[c][j] not currently in stack do
            S.Push(L[c][j])
return subnetworks
```

The time complexity is Θ (|E|+|V|) as this algorithm is a fairly standard implementation of depth first search on a graph represented by an (sorted) adjacency list. In analysis, the first basic operation of this algorithm is testing whether marked[L[c][j]] equals to 0 (whether each neighbour of the cth vertex has already been marked and if not add it to the queue). Even though it is a for loop in the while loop, the number of times the comparison happens will always equal to the number of edges in the graph – each c's adjacency list is run through, and the number of items in the adjacency list equals to the number of edges. Therefore, this part gives complexity Θ(|E|). The Θ(|V|) comes from the second basic operation which is the need to visit every node (S.Pop()). Thus, summing these two operations we get the total complexity of Θ(|V|+|E|).

Task 5.

```
**Function** Task1_5(V, E)
//Returns the number of critical components in this graph via brute force
//Input: an int array V of increasing neighbouring integers starting at 0 denoting each vertex while E is an array of 2 component int arrays denoting edges (the 0th and 1st element of the edge array denote the start vertex and end vertex).
//Output: an array storing the indices of the critical components in the graph

n ← length(V)
subnetworks ← Task1_1(V, E)
out ← int array size n with all cells initiated to 0
nout ← 0

for i ← 0 to n-1 do
    V' ← V\{i}
    E' ← E\{all edges with i as start or end}
    newsubnetworks ← Task1_1(V' E')
    if subnetworks < newsubnetworks do
        out[nout] ← i
        nout ← nout + 1
return the 0th to (nout-1)th items of array out
```

The basic operation of function Task1_5 is running Task1_1. This is required |V| times at Θ((|V|-1)+(|E|-a)) where a is a number bounded above by |V| (the number of edges deleted as a result of the ith vertex being deleted – max case it was connected to every other node so deleting it will remove |V|-1 edges), so on average asymptotically the time complexity is O(|V|+|E|) for one Task1_1 run. Another run of Task1_1 is required at Θ(|V|+|E|). This results in the cost of (|V|+1) * O(|V|+|E|) which asymptotically equals to O(|V|(|V|+|E|)). (Note the Θ was changed to O because the n runs were big O instead of big theta so when added together the strongest claim of big theta could not be made on the final complexity).

Task 6.

```
Function Task1_6(V, E)
//Returns the list of critical points in the graph G(V, E)
//Input: an int array V of increasing neighbouring integers starting at 0 denoting each vertex while E is an array of 2 component int arrays denoting edges (the 0th and 1st element of the edge array denote the start vertex and end vertex).
//Output: an array storing the indices of the critical components in the graph

L ← adjacency list of G(V, E)
    /This is an array of int arrays where the int array in the 0th position of the original array hold the sorted neighbouring nodes of the 0th vertex and so forth (assume sorted upon input)
S ← createStack()
PushOrder, ncrit ← 0
PO, HRA ← int array of size n with all cells initiated to -1
out, nchild, parent, marked ← int array of size n with all cells initiated to 0
n ← length(V)

S.Push(V[0])
marker ← 0
PO[0] ← PushOrder
PushOrder ← PushOrder + 1

while marker != n do
    c ← S.Pop()
    if c = Null do
        ncrit ← FindCrit(nchild, parent, marked, HRA, ncrit, out, PO, n)
        for i ← 0 to n-1 do
            if marked[i] = 0 do
                c ← i
                PO[c] ← PushOrder
                PushOrder ← PushOrder + 1
                break

    marker ← marker + 1
    marked[c] ← marker
    HRA[c] = c
    for j ← length(L[c])-1 to 0 do
        if marked[ L[c][j] ] = 0 and L[c][j] = 1 do
            if PushOrder[ L[c][j] ] != -1 do
                S.Push( L[c][j] )
                parent[ L[c][j] ] ← c
                PO [ L[c][j] ] ← PushOrder
                PushOrder ← PushOrder + 1
                parent[ L[c][j] ] ← c
            else
                parent[ L[c][j] ] ← c
        if marked[ L[c][j] ] != 0 and L[c][j] = 1 do
            if L[c][j] != prev and marked[ HRA[c] ] > marked[ L[c][j] ] do
                HRA[c] = marked[ L[c][j] ]
    prev ← c
ncrit ← FindCrit(nchild, parent, marked, HRA, ncrit, out, PO, n)

return the 0th to (ncrit-1)th items of array out
```

```
Function FindCrit(nchild, parent, marked, HRA, ncrit, out, PO, n)
//Returns use the given information to add any critical points of this subnetwork into the out array and
update the number of critical points
//Input: arrays nchild, parent, marked, HRA, out, PO (assuming changes here are reflected globally because
of pointers), int ncrit and buddy variable n for all the arrays.
//Output: the updated ncrit value

for node in this subsequence (in reverse order of visit) do
    for all children of node do
        if HRA[node] > HRA[child] do
            HRA[node] = HRA[child]

update nchild array by counting appearances of a node's index (nodes from this subnetwork only) in parent
array.

for node in this subsequence do
    if node is first of subnetwork and nchild[node]>1 do
        out[ncrit] ← node
        ncrit ← ncrit + 1
    else if nchild[node] != 0 do
        for each child of node do
            if PO[ HRA[child] ] >= PO[node] do
                out[ncrit] ← node
                ncrit ← ncrit + 1
return ncrit
```

While task1_1 and task 1_5 share the same framework of a DFS but task 1_5 is significantly more involved, due to all other operations being less than O(|V|+|E|), they are absorbed by O(|V|+|E|) in asymptotic complexity and hence the final time complexity of this graph is still O(|V|+|E|). The only extra loops other than task1_1 are the three in the helper function FindCrit. All other additional operations should be constant time even if considering them to be additional basic operations so does not change the asymptotic time complexity.

For FindCrit, each run first does a loop in terms of |V'| with a loop for the children inside. As the number of children ultimately adds up to |V'|-1, the two loops have a joint complexity of O(|V'|). The second loop (updating the nchild array) takes time O(|V'|), while the third loop is the same as loop 1. So overall FindCrit is of time O(3|V'|) which asymtopically O(|V'|). Although the times FindCrit can be called up to |V| times theoretically, which means the overall addition to O(Task1_1) should be O($|V|^2$) and thus increasing the original complexity, but upon close analysis the total of all V' must sum up to V, so the total addition to the original O(|V|+|E|) by the use of FindCrit is O(|V|), totalling to O(2|V|+|E|). Overall O(2|V|+|E|) is asymptotically O(|V|+|E|), so Jan Arts' algorithm is indeed in time O(|V|+|E|).

Question 2

(/FHD are still hard drives so for question a, b and c the basic operation is still pointer access into memory rather than comparisons (instructions given by head tutor))

a. Use Hashing - linear probing with rehashing functions for clashes; a good hashing function and rehashing function will guarantee O(1) searching, while the downside of extra memory (double) required when the existing array runs out is not a concern the university is willing to pay more to buy memory. Other downsides of this data structure of occasionally slow insertion because of the need to rehash and a complex/impossible deleting process is also not a concern because the university is not as concerned with insertion and never deletes.

b. Use AVL trees – this guarantees searching of O(logn) – still reasonably fast – while extra space is only requested when a new item comes in (compared with the above which doubles every time so much of the space are often left unused). Insertion and Deletion are all O(logn) but again are not of concern.

c. Sorted array – this guarantees searching at O(logn) using binary search and minimal use of space if malloc exactly the required amount of cells for each big pass of insertion (less memory than linked lists because don't need to store pointers in each node). Sorting can be done after the 'big pass' is finished using the in-place O(nlogn) heapsort, guaranteeing relative speed for this special insert circumstance (even though there are no strict requirements for insertion) and users can pull masses of contingent data into the cache for 100x speed search which is both good for queries or repeated searches near the original request.

d. They are likely using Hashing with linear probing (with 'jumps' size i) which has not only bad hashes but also i is not coprime with the initial hash array size n – the latter explaining how only 85% of the allocated space are used as some spaces are jumped through and never reached (and likely doubling of space is automatically requested if found that the item cannot be put in current array even though there are still empty spaces). Their degrading output for new items is because of the bad hash function creating clusters and lots of clashes, meaning that newly inserted items are put far away from their initial hash value/index and finding them require stepping through the hash array more than O(1) time complexity (worst case O(n) time complexity).

Question 3

(/As have already discussed with head tutor, for this question assuming array values are integers only and that negative values are allowed for hash function output)

a. My solution is to return the value of the integer j inputted as a parameter from the hash function, as this means

   HASH(A, j) = j < j+1 = HASH(A, j+1)   for all integers j

   so the 'while' line of the insertion sort never runs and hence no swaps are done and the order of the original array is unchanged.

   Function HASH(A, j):
   return j

b. My solution is to return (-1) * the integer j inputted as a parameter from the hash function, as this means

   HASH(A, j+1) = -(j+1) < -(j) = HASH(A, j)   for all integers j

   This holds as by axioms of reals:

   a < b <=> -a > -b; and here j < j+1 so -(j) > -(j+1) for all integers j

   Hence this makes the while loop run j to 0 for each run of the i loop from 1 to n-1, doing all possible comparisons and hence triggering the worst case ($O(n^2)$) behaviour of insertion sort. Note even though the average case for insertion sort is also $O(n^2)$, this hash guarantees the absolute worst behaviour as it does the maximum number of comparisons, and comparisons are the basic operation for insertion sort.

   Function HASH(A, j)
   *return –1 * j*

c. My solution is to return "(-1) multiplied by the value at A[j]" from the hash function, as this means

   If A[j] < A[j+1] then HASH(A, j) = -(A[j]) > -(A[j+1]) = Hash(A, j+1) which triggers a swap, resulting in a decreasing order for this adjacent pair

   If A[j] > A[j+1] then HASH(A, j) = -(A[j]) < -(A[j+1]) = Hash(A, j+1) which doesn't trigger a swap, preserving the original decreasing order.

   Thus, in the end all pairs are in decreasing order (by induction), or, intuitively speaking the hash gives the opposite result all the time to the insertion sort mechanism so the array would be sorted in reverse.

   Function HASH(A, j)
   *return -1 * A[j]*