AMIT BORASE

Question 1

Algorithm

Let G = (V, E) be the weighted, undirected graph and T = (V,E') be the minimum spanning tree of the same. Let (u,v) in E be the edge whose weight is decreased. Below is the algorithm to find a new MST of G.

1. Build a adjacency list of the MST T of G.
2. Find a path from u to v in T, to do this we run a modified DFS (that pushes a vertex onto the queue Q and pops it when all its children are explored) starting from u as root node and stop when we discover v.
3. Queue Q obtained at the end of above DFS run will give a path from u to v. Now, traverse this path again and record the highest weighted edge (x,y) along such path.
4. Compare the weights edge (x,y) obtained in previous step with the reduced weight of edge (u,v).
   - if reduced weight of (u,v) is the least of the two, then remove the edge (x,y) from the MST T and add an edge (u,v) to it to obtain a new MST T.
   - if weight of the edge (x,y) is least of the two, then MST remains unchanged.

Proof of correctness

Claim 1.1

Any spanning tree does not contain a cycle.

Proof: We prove by contradiction, assume that a spanning tree S of undirected graph contains cycle C.

Then any edge can be removed from C and the remaining nodes will still remain connected, which is contradictory since removing an edge from spanning tree disconnects it.

Claim 1.2

Adding an edge into a spanning tree inserts a cycle C in it.

Proof: Consider spanning tree S of undirected graph and nodes u and v in it. Assume we plan to add an edge e between u and v.

Since S is a spanning tree, u and v are in spanning tree, and they have a path, say p between them.

Now if we add edge e to the spanning tree S, then there are 2 distinct undirected paths from u to v, first through p and second through edge e.

If we combine this 2 paths, we get $p \cup e$ which forms a cycle in S. Hence proved.

Claim 1.3

Cycle Property: For any cycle C in the graph, if the weight of an edge e of C is larger than the individual weights of all the other edges of C, then the edge e cannot belong to any of the MST.

Proof: Let us consider that the highest weight edge e of cycle C in graph G is a part of MST T.

Now if we remove edge e from T, it'll disconnect T into two different subtrees. But then the path C - e reconnects these two subtrees in the graph G.

=> There exists a edge e' in C - e such that e' connects vertices that are in separated subtrees of T'. Moreover the weight of such edge e' is lesser than weight of the edge e.

Now consider a MST T' formed by connecting two disconnected subtrees of T using edge e'. Such MST T' has weight less than that of a T, because the weight of e' is less than that of e. So, T' thus obtained is the actual MST and not T. Hence e can not be in any of the MST of the graph G.

Main Proof: We've a MST T of graph G and an edge e(u,v) in E - E' be the edge whose weight is reduced. Without loss of generality, this modified edge can be added to the exisitng MST T. By claim-1.2, this operation will introduce a cycle C in T. Moreover such cycle C = path p in T from u to v + edge e.

Furthermore, by using claim 1.3, the maximum weighted edge along this cycle C can not be a part of the MST. Hence, we find a maximum weighted edge along C, by running a DFS on T to find a path in T from u to v and find maximum weighted edge x along it. If the wt(x) > wt(e), then the a modified MST T' by replacing edge x by e is the new MST. Else if the wt(x) < wt(e), then the MST T remains unchanged.

Pseudo-code

Listing 1: Question 1

```
INPUT - Weighted, Undirected graph G=(V,E)
        MST T = (v, E') of G
        modified weight w' of edge (u,v)

OUTPUT - A modified MST T' of the graph G.

bool DFS(T(V, E'), x, v)
    if x == v
        return true
    S.push(x)
    vis[x] = true
    for y in adj_list[x]
        if vis[y] == false
            if DFS(T, y, v)
                return true
    S.pop()
    return false

UPDATE_MST(G, T, wt_uv, u, v)
    adj_list[] = {0}
    stack S = {}
    bool vis[] = {0}
    // Build adjacency list of T
    BUILD_ADJ_LIST_T(T, adj_list[])
    ASSERT(DFS(T, u, v))
    last = v
    max_wt = 0  // max weight edge on path from u to v
    max_left = NIL  // left_node of max_wt edge
    max_right = NIL // right_node of max_wt edge
    for !S.empty()
        S.pop() = x
        if (max_wt < wt(x, last))
            max_wt = wt(x, last)
            max_left = x
            max_right = last
        last = x
    if (max_wt > wt_uv)
        T = T - edge(max_left, max_right)
        T = T + edge(u, v)
    return T
```

Time complexity analysis

- Building an adjacency list of T takes O(V+E') time in worst case, but |E'| = |V-1| because number of edges in spanning tree is exactly one less than number of vertices. Hence, building adjacency list takes O(2V-1) = O(V) time.
- Modified DFS to find a path from u to v also takes O(V+E') time in worst case.
- Traversing a path from v to u to find max-wt edge takes O(E') time in worst case.

Hence total running time of above algorithm is O(V+E'). But for a spanning tree, |E| = |V| - 1 Hence total time taken is: O(V + V - 1) = O(V)

Question 2

Assume G(V,E) is the undirected graph. Let T(V,E') be the MST of G, then E' is a subset of E.

Let H=(V^H, E^H) be the subgraph of G, then $V^H and E^H$ are subsets of V and E respectively.

Now without loss of generality, for every edge e in E' we can find a cut [S, V-S], such that e connects a vertex from S to a vertex from V-S and is the least weighted among all the edges crosing such cut.

Now Use the same cut to divide subgraph H into two parts, then obtained cut is [S', $V^H - S'$] where S' = $S \cap V^H$ and $V^H - S'$ = $V^H - S \cap V^H$.

I.e. the cut [S', $V^H - S'$] will be such that, the nodes of either sides of the cut will be subsets of the nodes in the original cut [S, V-S] of G.

Also the edges crossing across cut [S', $V^H - S'$] will be subsets of the edges crossing across the cut [S,V-S] of G.

There are 2 possibilities, either edge e is in E^H or not. If it not in E^H then nothing to prove.

If the edge e in E' is present in E^H as well then e will also be the least weighted edges crossing across the cut [S', $V^H - S'$] of H. (because if an element e is least in the superset then e will be the least in the subset as well, if e is in a subset).

Then according to cut-property of the graph H, edge e will be in every MST of H.

And hence it proves that if the edge e is in both T and H, then it has to be in every spanning tree of graph H.

Hence $T \cap H$ is always contained in MST of H.

Question 3

Question 3A

We use the greedy approach to solve this problem.

Greedy Strategy: We select the highest possible currency value and pay using it as much as possible.

Algorithm

Let V be the total amount of sum to be paid using currency system. Since c,k > 1 and v^1 = 1, therefore $v^(k) > v^(k-1) > ... > v^2 > v^1$

1. Start with $v^(k-1)$.
2. Try to pay as much as possible of sum V, using currency selected in 1. And update the V
3. Get the next highest valued currency v^i and repeat 2
4. Repeat till V becomes 0

Proof of correctness

Consider an optimal solution O = $o_1, o_2, .., o+k$. Let G = $g_1, g_2, .., g+k$ be the greedy solution, where $o_i and g_i$ are coins of value v_i chosen by optimal and greedy algorithm repsectively.

Base Case: $g_k * v_k >= o_k * v_k$ (Base Case)

Proof: Since v_k is the highest valued coin and the value to pay i.e. V is same initially for both greedy and optimal solution. Then by the greedy strategy, the number of coins of type v_k chosen by greedy will always be greater than or equal to that chosen by the optimal algorithm.

Hence $g_k >= o_k$ Multiplying both sides by v_k we get, $g_k * v_k >= o_k * v_k$. hence Proved.

Induction Step: $g_k * v_k + .. + g_i * v_i >= o_k * v_k + ... + o_i * v_i$ Proof: This can be proved with the same argument that greedy always tries to pay the highest most amount possible at each individual step.

Otherwise, if the amount paid by optimal solution at stage i is greater than that of greedy then, for some j in i to k, optimal has to have paid using v_j instead of v_{j+1}, while the greedy paid using v_{j+1}.

Then the amount paid by optimal may be great but the number of coins paid in the process will always be greater than that paid by greedy because, $v_{j+1} = c.v_j$ where c > 1. Hence, it won't be an optimal solution. So to stay optimal, optimal solutions has to pay using v_{j+1} and not using $v - j$.

Hence greedy will be the optimal solution till.

Pseudo-code

Listing 2: Question 3

```
INPUT: currency system $v^1=1, v^2=c^1,...., v^(k-1)=c^(k-2), v^(k)=c^(k-1)$ , ↩
    where c and k > 1.
OUTPUT: Total coin count.
COIN_SYSTEM(V)
    coin_count = 0
    for i=k; i>=1; i--
        count += Div(V, $v^i$)  // division operation to get quotient
        V = mod(V, $v^i$)       // modulo operation to get reminder
        if V == 0
```

```
            break
    ASSERT(V==0)
```

Time complexity analysis

- There are k diferent coins, hence for loop executes k times.
- Lines 233 through 236 take constant time for execution.

Hence the total run time complexity remains k*O(1) = **O(k)**, where k is number of different valued coins available in coin system.

Question 3B

We prove this by providing a counter example. Let the coin system be 1, 3, 7, 16. This coin system satisfies the given constraint because it is increasing in value by the factor of at least 2.

i.e. 3/1=3, 7/3=2.33, 16/7=2.28 are all greater than 2. Now We use above greedy algorithm to pay for the sum of 21. The result will be 1 coin of value 16, 0 coins of value 7, 1 coin of value 3 and 2 coins of value 1. In total 4 coins will be required.

But we have better solution for this, which takes only 3 coins of value 7.

Above counter example proves that the greedy algorithm listed above does not work if for all 1 i < k, $v_(i+1)/v_i$ c for some integer c>1

Question 4

This problem can be solved using greedy approach.

Greedy strategy: Work on the node with max degree requirement and create an edge from it to other nodes with decreasing degree demands.

Algorithm

Let $d_1, d_2, ..., d_n$ be the non increasing ordering of the given degree sequence. Let G= (V,E) be the graph (if it exists) and $v_1, v_2, ..., v-n$ be the corresponding vertices of the graph, such that degree-of(v_i) = d_i, for all i in 1 to n.

We start with E = NIL.

We maintain a list active-demand (which has to be empty upon successful termination) to keep track of vertices that still have residual degree demand. To start with, active-demand is initialized with $d_1, d_2, ..., d-n$. An element is removed from such list as soon as it becomes zero, i.e. its corresponding vertex's demand gets fulfilled.

1. Sort the given degree sequence to obtain a non-decreasing degree sequence $d_1, d_2, ..., d_n$.
2. Initialize active-demand with non-zero d_i in $d_1, d_2, ..., d_n$.
3. Start from head of the active-demand and mark the element as e.
    (a) get next element e' in active-demand list. Now an edge is possible between vertices represented by e and e'. If unable to get next element e', then graph is not possible, return FALSE.

(b) Now an edge is possible between vertices represented by e and e', hence decrement the e and e' both by 1.

(c) if e' becomes zero, i.e. the degree requirements of vertex corresponding to e' are satisfied, then remove element e' from the active-demand list

(d) if e is not zero, then

- if there are vertices remaining to be iterated for e, then repeat again from 3.a
- if there are no vertices remaining to be iterated for vertex a , then the graph with given degree sequence is not possible. return FALSE.

(e) if e is zero, then remove e from the active-demand list

(f) sort the active-demand list in decreasing order residual demand

(g) if active-demand is not empty, repeat 3

(h) if active-demand list empty, then graph is possible, return TRUE.

Proof of correctness

CLAIM 1: Let S = $d_1, d_2, ..., d_n$ be the degree sequence of n integers, then there exists a a graph for S only if the S' = $d_2 - 1, d_3 - 1, ..., d_k - 1, d_{k+1} - 1, d_{k+2}, .., d_n$ of n-1 integers is also a degree sequence with possible graph, where k = d_1 Proof: Let us consider S' has a graph G' with a given degree sequence. Now consider k vertices with degree's $d_2 - 1, d_3 - 1, ..., d_k - 1, d_{k+1} - 1$ in the graph G'.

Now add a new vertex v_1 to G' such that v_1 is shares an edge with all these k vertices. The newly obtained graph will have a degree sequence = $d_1, d_2, ..., d_n$ with d_1 = k. This is nothing but degree sequence S for the graph G. Hence G' exists then G will always be.

Now, we prove that if decreasing order sequence S = (d1, d2, . . . , dn) with d1 d2 ...dn1 dn has a graph G, for which S is the degree sequence.

Pseudo-code

Listing 3: Question 3

```
INPUT: A degree sequence D = $(d_1',d_2',.....,d_n')$
OUTPUT: Returns TRUE if a graph G for given degree sequence exists, else ↩
   returns false.

bool GRAPH_DEGREE_SEQUENCE(D)
    $(d_1,d_2,...,d_n)$ = sort(D= $(d_1',d_2',.....,d_n')$)   // in non-↩
       increasing order
    for all i from 1 to n
        act_demand.add_end($d_i$)
    while act_demand != EMPTY
    do
        d = act_demand.head()
        d_next = d
        for d_next = act_demand.next(d_next)
            if d_next == NIL
                return FALSE
            act_demand.decrement(d_next)
            act_demand.decrement(d)
            if act_demand.value(d_next) == 0
                act_demand.remove(d_next)
            if act_demand.value(d)  == 0
                break
            else if act_demand.value(d)  != 0
                if act_demand.next(d) == NIL
                    return FALSE
                else
                    continue
        if act_demand.value(d) == 0
            act_demand.remove(d)
        act_demand.sort_by_value(decreasing_order)
    if act_demand.empty() == true
       return TRUE
```

Time complexity analysis

- To start with, we sort the given degree sequence in non-increasing fashion, this takes O(nlog(n)) time.
- The while loop executes n times in the worst case
- The inner for loop executes n-1 times in the worst case
- sort within a while loop takes another O(nlog(n)) time.

Hence complete run-time taken will be

= O(nlog(n)) + n [(n-1) + nlog(n)]

= O(nlog(n)) + O($n^2) + O(n^2 log(n))$

= O(nlog(n)) + O($n^2 log(n))$

= O($n^2 log(n))$