Problem 1. DPV 5.6

Let $G = (V, E)$ be an undirected graph. Prove that if all its edge weights are distinct, then it has a unique minimum spanning tree.

Solution:

- **Proof:** Let $G = (V, E)$ be an undirected graph with unique edge weights. Let T_1 and T_2 be the two different Minimum Spanning Trees of G. Without loss of generality, let us assume that $e_1(A, B)$ in T_1 be the edge with minimum weight out of all the edges which are present in one MST but not the other. A and B must also be connected in T_2 (T_2 is a spanning tree). Let us call the path that connects A to B in T_2 as P_2. We know that there must be at least one edge in P_2 which is not present in T_1, otherwise, T_1 would have a cycle. Let us call this edge e_2. We also know that $Weight(e_1) < Weight(e_2)$ (e_1 was chosen to be the one with minimum weight out of all the edges present in one MST but not in the other, also since all the weights are unique, there can only be one such edge with the minimum weight). Now if we replace e_2 in T_2 with e_1, T_2 still remains a spanning tree while its total weight has reduced, which contradicts our initial assumption that T_2 was a Minimum Spanning Tree.

Problem 2

We are given two arrays, A and B, containing n positive integers each. You can permute each of the two arrays in any fashion such that after the permutation, we have two permuted arrays A' and B' corresponding to arrays A and B respectively. Give an efficient algorithm to compute the maximum possible value of $\prod_{i=1}^{n}(A'[i] + B'[i])$.

Solution:

- **Algorithm:** We sort the two arrays such that A is in ascending order, and B is in descending order. We can also sort A in descending order and B in ascending order. We now get the two corresponding new arrays A' and B'. For each index i, we calculate the sum of the pair of elements from A and B. We take the product of these sums to receive the maximum product of sums.

  Let a_i be the ith element of A' and b_i be the ith element of B'. Without loss of generality, we use the following ordering:

  $$a_1 \leq a_2 \leq \cdots \leq a_n$$
  $$b_1 \geq b_2 \geq \cdots \geq b_n$$

  The greedy algorithm uses the above ordering to obtain the maximum product of sums, $P_G = \prod_{i=1}^{n}(a_i + b_i)$.

- **Pseudocode:**

```
procedure calculateMaxPOS(A,B,n):
   input: two arrays A and B having n elements
   output: maximum product of sums of elements in the two arrays

   A' = sort(A, ascending)
   B' = sort(B, descending)
   product = 1
   for i=1 to n, step 1
      product = product * (A'[i] + B'[i])
   return product
```

- **Proof of Correctness:** Without loss of generality, let us sort array A in increasing order to get A' and the problem now reduces to finding the optimal permutation of array B which gives the largest product of sums. Let G_B denote greedy solution for array B' and O_B denote the optimal solution for array B'. We will now use the exchange argument to prove the correctness of our greedy solution.
  Let i be the first index where the greedy solution, G_B, differs from the optimal solution O_B i.e. $G_B[k] = O_B[k], \forall k < i$ and $G_B[i] \neq O_B[i]$. Let us assume, that in O_B, the element corresponding (equal) to $G_B[i]$ is at some index $j > i$. So in the optimal product of sums P_{old}, we have the following two terms (in addition to others $= P'$), i.e. $P_{old} = (O_B[i] + A'[i])(O_B[j] + A'[j])P'$. Note that we have $j > i$ (Both O_B and G_B are same till index i), $A'[i] \leq A'[j]$ (A' is sorted in non-decreasing order) and $O_B[i] \leq O_B[j]$ ($O_B[j] = G_B[i]$ and G_B is sorted in non-increasing order and is thus the largest number among all the numbers of O_B from index i to n). If we exchange $O_B[i]$ and $O_B[j]$, we get $P_{new} = (O_B[j] + A'[i])(O_B[i] + A'[j])P'$. We observe that $P_{new} - P_{old} = (O_B[j] - O_B[i])(A'[j] - A'[i])P' \geq 0$. Thus we observe, making an exchange like the one we made, brought the optimal solution closer to the greedy solution without decreasing the product of sums. We can make such exchanges as many times as needed and in the end we will end up with the optimal solution being the same as the greedy solution.

- **Time Complexity:** Sorting can be done in $O(n \log n)$ time. The remaining computations are linear in n, and therefore our algorithm obtains the maximum product of sums in $O(n \log n)$ time.

Problem 3

You are given $2n$ numbers. Your task is to partition them into two sets of n numbers each, so as to maximize the quantity: (sum of all numbers in the first set) minus (sum of all numbers in the second set). Devise an efficient algorithm to accomplish your task.

Solution:

- **Algorithm:**
  1. Use the linear time algorithm to find the median (described in Lecture 5 slides by Prof. Miles, Page 30 onwards).
  2. Add all the elements smaller than the median to the second set.
  3. Add all the elements greater than the median to the first set.
  4. If either of the sets do not have a size of n, fill them with the median until the size reaches n.
- **Pseudocode:**

```
procedure computePartition(S[]):
    input: S[] with 2n elements
    output: A[],B[] with n elements

    m = get_median(S)

    for i= 0->2n:
        if S[i] < m:
            B.append(S[i])
        else if S[i]> m:
            A.append(S[i])

    for i= len(B)->n:
        B.append(m)

    for i= len(A)->n:
        A.append(m)
```

- **Proof of Correctness:**
  Let A and B be the first set and the second set generated by the greedy solution respectively. Let A_{opt} and B_{opt} be the the first set and second set in the optimal answer respectively. Let's show that we can get the greedy solution A and B by exchanging elements between sets A_{opt} and B_{opt}. Let $s(X)$ be the sum of all elements in some set X.

  If the optimal answer is different than the greedy answer, i.e. $A \neq A_{opt}$ and $B \neq B_{opt}$, then there must exist a pair (x, y), such that $x \in A$, but $x \notin A_{opt}$, and $y \in B$, but $y \notin B_{opt}$. As any element must be assigned to one of the two sets, we can imply that $x \in B_{opt}$ and $y \in A_{opt}$.

  Let's show that exchanging elements x and y in the optimal answer will not make it worse. From our algorithm, we know that $x \geq m$ and $m \geq y$, thus $x \geq y$. Let A'_{opt} and B'_{opt} be the first set and the second set that we get after exchanging elements x and y in the optimal solution respectively.

  We have the following:

$$s(A_{opt}) \leq s(A_{opt}) - y + x = s(A'_{opt})$$
$$s(B_{opt}) \geq s(B_{opt}) - x + y = s(B'_{opt})$$

Thus, we have:

$$s(A_{opt}) - s(B_{opt}) \leq s(A'_{opt}) - s(B'_{opt})$$

As we can see, exchanging elements doesn't make the difference smaller. We can perform such exchanges for all such pairs (x, y) until we get sets A and B. According to the property shown above, we will get that after all exchanges:

$$s(A_{opt}) - s(B_{opt}) \leq s(A) - s(B)$$

Which implies:

$$s(A_{opt}) - s(B_{opt}) = s(A) - s(B)$$

Thus, our algorithm generates the optimal answer.

- **Time Complexity:** The median can be found in $\mathcal{O}(n)$ using the linear time median finding algorithm. The loop which iterates from $0 \rightarrow 2n$ takes order $\mathcal{O}(n)$ time. Hence,

$$complexity = \mathcal{O}(n)$$

Problem 4

Let's consider n cities $c_1, c_2, c_3, ..., c_n$. One can travel from a city to another city through a one-way bridge. Furthermore, assume that there are no cycles generated by the bridges (even in between two cities). Thus, we have a directed acyclic graph, where vertices represent cities and directed edges represent one-way bridges. Give an efficient algorithm to find the longest sequence of cities that one can travel to. Note that you are not given a city to start with; the algorithm should rather return a longest sequence of cities that is possible to travel to.

Solution:

- **Algorithm:** Let's represent the problem as finding the longest path in a graph $G = (V, E)$ where each city is represented by a node c_i in G and the one-way bridges are represented as follows:
  If there is a one-way bridge from city c_i to city c_j, there is a directed edge from node c_i to node c_j.
  The graph is a directed acyclic graph, as stated in the problem. For simplicity, we shall *additionally* create a global source node S and a global sink K. There is an edge from S to every node in G except K; there is an edge from every node in G except S to K. We ignore these nodes while printing the longest path.

  This is one approach to solving the problem. The other approach which does not involve global source/sink nodes is outlined at the end.

  The algorithm to compute the longest sequence of cities that one can travel to, first computes the topological order of the nodes in the graph.

  We define the following notations for each node v in G:

  - Let $dist(v)$ be the length of the longest path that terminates at node v.
  - Let $parent(v)$ be the node immediately preceding v in the longest path that terminates at v.

  Thus, $dist(S) = 0$ and $parent(S) = null$. We can define $dist(v)$ for every other node as $dist(v) = max_u\ dist(u) + 1$ where u in an immediate predecessor of v in G and $parent(v) = u$.

  We immediately observe that this is a modification of the relaxation condition in shortest path algorithms along with a stricter topological order of traversal.

  Therefore, we can recursively compute the distances and the longest path that terminates at every node, when we parse the nodes in topological order. We can then backtrack from the sink K to traverse the longest sequence of cities one can travel to.

- **Pseudocode:**

```
procedure LongestPath():
 Construct graph G = (V, E)
 for each u in V do
   dist[u] = -Inf
   parent[u] = null
 end for

 dist[s] = 0
 V_list = Topological_sort(V)

 for each v in V_list do
   for each u which is a predecessor of v do
      if dist[v] < dist[u] + 1 then
          dist[v] = dist[u] + 1
          parent[v] = u
      end if
```

```
   end for
end for

current = parent[K]
while current <> S do
  print current
  current = parent[current]
end while
```

- **Proof of Correctness:**
  We first note that the recursive formulation computes $dist(v)$ based on computed values of $dist(u)$ where u is a predecessor of v in G. We do not at any further point in time recompute $dist(u)$. Since we traverse the nodes in topological order, this property is guaranteed to hold.

  We shall now prove that $dist(v)$ and $parent(v)$ is computed correctly for every node v by induction on the index of v in the topological order.

  Base case: For the source node S, $dist(S) = 0$ and $parent(S) = null$ are correctly computed since there is no other source node for this graph.

  Inductive hypothesis: For every other node v, there should exist at least one predecessor node u, where we assume that $dist(u)$ and $parent(u)$ are correctly computed. Then, $dist(v)$ and $parent(v)$ are correctly computed.

  Inductive step: Length of the path at v is $dist(u) + 1$, where $dist(u)$ is the length of the longest path that terminates at u. We consider all the immediate predecessors of v and correctly compute the maximum of $dist(u) + 1$ to correctly set the distances and the parent variables.

  Therefore, by induction hypothesis, $dist(K)$ represents the longest path in the graph, i.e the longest possible sequence of cities one can travel to excluding the artificial source and sink nodes.

- **Time Complexity:** Topological sort has $O(|V| + |E|)$ time complexity and the rest of the algorithm has the time complexity $O(|V| + |E|)$, the overall time complexity is $O(|V| + |E|)$

Alternative solution:
Let's find the topological order of the graph first. Then we compute $f(v)$, which is the length of the longest path that starts at node v, for every node v in the graph. We compute values $f(v)$ recursively following the topological order and we compute them for each node **only once**:

- if $f(v)$ has been computed, then return the value of $f(v)$
- If v is a leaf, then assign $f(v) = 0$.
- If v is not a leaf, then first find values $f(u)$ for all nodes u, such that there is an edge (v, u). Then, we assign $f(v) = \max(f(u)) + 1$.

Then, we find a node s, such that $f(s)$ is maximized. We output the path recursively starting at node s. After visiting node s, we output it and then find **one** node u, if such exists, such that there is an edge (s, u) and $d(u) = d(s) - 1$. Then, we recursively go to node u. The correctness of the algorithm can be proven inductively in a similar way as to the first solution.

We can find the topological order in a linear time. Finding the values $f(v)$ also works in a linear time as we process each node only once. The same can be said about finding the path itself. Thus, the overall complexity is $\mathcal{O}(|V| + |E|)$.

Problem 5 (Extra Credit)

Jersey Ike is aiming to enter the Guinness Book of World Records with the fastest and most massive delivery of sandwiches to customers in a single location. The company's strategy is to set up a VERY long service counter, with K Jersey Ike employees standing behind it. Before the timer starts, N customers take their places side by side at the counter, and place their sandwich orders. For each sandwich, it is known how much time is required to make the sandwich (all of the Jersey Ike employees are identically fast on all sandwich types). Jersey Ike must assign its employees to contiguous sections of the counter, so as to minimize the time required to make all of the customers' sandwiches. (In other words, a single employee cannot make the sandwiches of, say, customers 13, 22 and 57. But, a single employee could be assigned to make the sandwiches of customers 13, 14 and 15.) Also, the sandwich-making must respect a core value of the Jersey Ike company, namely, that any given sandwich must be made by exactly one employee. Devise an efficient algorithm that determines how Jersey Ike should optimally assign sections of the counter (i.e., contiguous intervals of customers) to its K employees.

Solution:

- **Algorithm:** The algorithm has two parts:
  - **To find whether a *given* time of completion can be met.**
    This can be done by assigning the jobs greedily to an employee until total time assigned to them, exceeds the given time of completion. Once the condition is achieved, the subsequent jobs are to be assigned to a new employee using the same criterion as before. If we are able to assign all the jobs in such a way we return true, otherwise false.
  - **Find the minimum time in which jobs can be completed.**
    We can perform binary search to determine the lowest possible time in which the jobs can be completed. The upper bound for the binary search is set to the sum of all the required times to make the sandwiches and the lower bound is set to the max of all required times.
- **Pseudocode:**
  $t1$, $t2, \cdots, tN$ are the time required to complete a sandwich for the respective customers.

```
procedure MinPossibleTime (Input t = [t1,t2,...,tN], K, Output time):
    start = max({t1,t2,...,tN})
    end = sum({t1,t2,...,tN})
    answer = end
    while (start <= end):
        mid = (start + end) / 2
        if (isPossible([t1,t2,...,tN], K, mid):
            answer = mid
            end = mid
        else:
            start = mid + 1
    return answer

procedure isPossible([t1,t2,...,tN], K, max):
    jobAssigned = 0
    for (t = 1...K):
        timeWorked = 0
        while (timeWorked + t [jobAssigned + 1] < max):
            timeWorked += t [jobAssigned + 1]
            jobAssigned += 1
     if (jobAssigned >= N): return True
    return False
```

- **Proof of Correctness:**

  Proof for isPossible. Let's prove that the greedy algorithm applied to this problem provides us an optimal assignment of N customers to K employees by arguing that *greedy* stays ahead of any other solution at every step.

  Let $G = g_1, ..., g_K$ be the sequence of the time taken by the employees that is produced by the greedy algorithm.

  Let $O = o_1, ..., o_K$ be the sequence of the time taken by the employees that is produced by an optimal solution.

  Assume that the first j customers have been covered by first i employees i.e. the greedy sequence (G) and the optimal sequence (O) look identical till iteration i.

  Now, next employee $(i + 1)$ can be assigned customers at counters starting $(j + 1)$.

  By the greedy measure that we have defined, we assign the customers to an employee until the total time for the employee exceeds the given time of completion (T). Since we are allowed to assign the contiguous customers to every employee, let us assign customers from counter $(j + 1)$ to m to employee $i + 1$, i.e. $g_{i+1} \leq T$.

  We claim that this is an optimal assignment for employee $(i+1)$ since the algorithm stops assigning the customers to the employee only when the total time for employee exceeds T as per the greedy measure. This means that customer at counter $(m + 1)$ cannot be assigned to employee $(i + 1)$ and customers from $(j+1)$ to m were the best assignment for employee $(i+1)$. Thus, we conclude $g_{i+1} \geq o_{i+1}$ and $g_{i+1} \leq T$. In other words, greedy stays ahead of any other optimal solution after iteration $i + 1$. This holds true for all i from 1 to K.

  Proof for application of binary search
  If we can achieve an assignment that completes in time T, any other time greater that T need not be considered.

  If there does not exist an assignment that finishes in time T, there does not exist an assignment for a time less than T. If an assignment exists for time $T-1$ the same assignment would hold good for time T. Hence, if assignment an does not exist for T, we do not need to check if assignments are possible for any given time less than T.

  We can now apply binary search ignoring the values which do not affect our solution as proved above. All the time values which are possible solutions are correctly searched.

- **Time Complexity:** Each call of isPossible takes $\mathcal{O}(N)$. The whole algorithm calls isPossible $\mathcal{O}(\log(S))$ times, where $S = sum(t1, t2, ..., tN)$. Then the total time is

$$\mathcal{O}(N \cdot \log(S))$$