Romerico David Jr
COSC 483.001 Final Exam
Professor Nguyen

# Question 1

**Problem:** Given an array $A[]$ of $n$ integers, find the length of the longest non-decreasing subsequence such that for every two adjacent elements in the subsequence, say $x \leq y$, we have $y \geq 2x$.

### (1) Thought Process and Approach

The problem is a variation of the longest increasing subsequence problem with the constraint that adjacent elements in the subsequence must satisfy $next \geq 2 \times previous$. A brute force solution would be to try all possible subsequences but this would be exponential time. A greedy approach would not be straightforward because of the constraint, so a dynamic programming approach would be an optimal choice. The idea as follows:

1. Initialize $dp[i]$ which represents the length of the longest valid subsequence ending at index $i$.
2. Each element starts with a subsequence length of 1. For each element $A[i]$, iterate through all previous elements $A[j]$ and check if $A[i]$ can extend the subsequence ending at $A[j]$, that is, if $A[j] <= A[i]$ and $A[i] >= 2 \times A[j]$.
3. If the conditions are met, update $dp[i] = max(dp[i], dp[j] + 1)$ to reflect the longest subsequence length ending at $A[i]$.

### (2) Pseudo-Code

```
function longest_special_subsequence(A):
    n = length(A)
    dp = array of length n, initialized to 1

    for i from 0 to n-1:
        for j from 0 to i-1:
            if A[j] <= A[i] and A[i] >= 2 * A[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

### (3) Proof of Correctness/Optimality

- **Correctness/Optimality:** $dp[i]$ represents the length of the longest valid subsequence ending at index $i$. By iterating over all $j < i$, we ensure that we

consider all possible predecessors that can form a valid pair with $A[i]$. We only extend subsequences if they maintain the non-decreasing order and the double constraint. Therefore, each state is correctly computed based on optimal solutions of smaller subproblems, satisfying optimal substructure and correctness.

**(4) Time and Space Complexity**

- **Time Complexity:** The nested loops each run up to $n$ times, so the complexity is $O(n^2)$.
- **Space Complexity:** We use an array $dp$ of length $n$, so $O(n)$ space.

# Question 3

## (1) Thought Process and Approach:

The problem pretty much asks us to partition the set of required trucks per zone into two subsets (morning shift and afternoon shift) such that the maximum sum of any one subset is minimized. We have a list of integers (truck requirements per zone) that we want to split them into two groups, $M$ and $A$, to minimize $max(sum(M), sum(A))$.

We can brute force the problem by checking all partitions possible, but this would be exponential in time. Instead, we can do dynamic programming similar to a Knapsack type solution. The approach is as follows:

- Compute the total sum $S$ of all truck requirements.
- We want to find a subset whose sum is as close as possible to $\frac{S}{2}$. If we can find a subset that sums to $\frac{S}{2}$ (when $S$ is even), then the other subset will also have $\frac{S}{2}$, minimizing the maximum sum.
- If $S$ is odd or an exact half partition isn't possible, we find the closest achievable sum to $\frac{S}{2}$.
- The minimal maximum sum will then be $max(subset\_sum, S - subset\_sum)$, which is minimized when $subset\_sum$ is close to $\frac{S}{2}$.

## (2) Pseudo-code:

```
function minimize_trucks(zones):
    N = length(zones)
    S = sum(zones)

    # Initialize DP array
    dp = two-dimensional boolean array of size (N+1) x (S+1), initialized to False
    dp[0][0] = True

    # Fill the DP table
    for i from 1 to N:
        for j from 0 to S:
            dp[i][j] = dp[i-1][j]
            if j - zones[i-1] >= 0 and dp[i-1][j - zones[i-1]]:
                dp[i][j] = True

    # Find the closest sum to S/2
    closest_sum = 0
    for j from 0 to S:
        if dp[N][j] and abs(j - S/2) < abs(closest_sum - S/2):
```

```
            closest_sum = j

    return max(closest_sum, S - closest_sum)
```

**(3) Proof of Correctness or Optimality:**

- **Correctness**: $dp[i][j]$ ensures that all possible subsets of the given zones up to index $i$ are explored, with each state representing whether a sum $j$ can be formed. Once the table is filled, $dp[N]$ encapsulates all achievable subset sums using all zones. Scanning this row to find the sum closest to $\frac{S}{2}$ guarantees a partition with the minimal possible maximum subset sum. Minimizing $max(sum(M), sum(A))$ is equivalent to making the subset sums as even as possible. Since $S$ is fixed, achieving one subset close to $\frac{S}{2}$ ensures the other subset is also close to $\frac{S}{2}$, yielding the smallest possible maximum sum without violating the constraints.

**(4) Time and Space Complexity Analysis:**

- **Time Complexity**: The DP approach loops through $n$ items and $s$ possible sums resulting in $O(ns)$ time complexity.
- **Space Complexity**: The DP table is of size $(n + 1)(s + 1)$ resulting in $O(ns)$ space complexity.

## Question 4

**Problem:** Solve the following linear program (LP) using the Simplex Method.

Maximize $Z = 3x + 5y + 2z$

Subject to the constraints:

$$x + 2y + z \leq 10$$
$$3x + 2y + 4z \leq 24$$
$$2x + 5y + 3z \leq 30$$
$$x, y, z \geq 0$$

### (1) Thought Process and Approach

We can perform the Simplex Method (standard algorithm for solving LP problems). Here's the step-by-step approach:

1. Convert the inequalities into equalities by adding slack variables.
2. Set up the initial simplex table.
3. Perform pivot operations to find a sequence of improved feasible solutions.
4. Continue until an optimal solution is found (no more positive coefficients in the objective row for a maximization problem).

### (2) Pseudo-Code to Solve the Problem using Simplex Method

```
function simplex_method():
    # Given LP:
    # Max Z = 3x + 5y + 2z
    # Constraints:
    #   x + 2y + z      + s1 = 10
    #   3x + 2y + 4z    + s2 = 24
    #   2x + 5y + 3z    + s3 = 30
    # x, y, z, s1, s2, s3 >= 0

    # Step 1: Form the initial table
    # Basic variables: s1, s2, s3
    # Rows represent constraints, columns represent variables x,y,z,s1,s2,s3 and RHS.

    # Initial Table (variables order: x, y, z, s1, s2, s3 | RHS):
    # Row 1 (s1):  1    2    1    1    0    0   | 10
    # Row 2 (s2):  3    2    4    0    1    0   | 24
    # Row 3 (s3):  2    5    3    0    0    1   | 30
    # Z-row:       -3   -5   -2    0    0    0   |  0   # We put negatives because we move Z
to LHS: Z - 3x -5y -2z = 0
```

```
# Step 2: While there is a negative coefficient in the objective row:
while (there exists a negative coefficient in the Z-row):
    entering_col = choose_most_negative_coefficient_in_Z_row()
    pivot_row = choose_pivot_row(entering_col) # using minimum ratio test
    perform_pivot_operation(pivot_row, entering_col)

# Step 3: Once no negative coefficients remain in the Z-row, the solution is optimal.
# Read off solution from table:
# The variables corresponding to basic columns give the final solution.
# Z value is given in the RHS of Z-row.

return optimal_solution, optimal_value
```

## (3) Proof of Correctness/Optimality

- **Correctness:** The Simplex Method guarantees correctness and optimality by iteratively moving from one feasible corner point to another which improves the objective value at each step. If an LP problem has an optimal solution, it will be found at a vertex of the feasible region. The pivot steps are designed to ensure feasibility and to increase/maintain the objective function value until no further improvement is possible which occurs when there are no negative coefficients in the objective row. This signifies that the optimal solution has been reached.

## (4) Time and Space Complexity Analysis

- **Time Complexity:** In worst case, the time complexity can be exponential, but it performs very efficiently in practice. For an LP with $m$ constraints and $n$ variables, each pivot step is polynomial in $m$ and $n$. On average, it often runs in expected polynomial time $O(mn)$, but worst-case complexity can be $O(2^{min(m,n)})$ in pathological examples.
- **Space Complexity:** Storing the table requires $O(mn)$ space.

## Actual Solution:

Maximize $P = 3x + 5y + 2z$

subject to:

$x + 2y + z \leq 10$

$3x + 2y + 4z \leq 24$

$2x + 5y + 3z \leq 30$

with $x, y, z \geq 0$

1) Convert Inequalities to Eqns

Slack variables $s_1, s_2, s_3 \geq 0$

$x + 2y + z + s_1 = 10$

$3x + 2y + 4z + s_2 = 24$

$2x + 5y + 3z + s_3 = 30$

Obj Function: $z - 3x - 5y - 2z = 0$

| Basis | x | y | z | $s_1$ | $s_2$ | $s_3$ | RHS |
|---|---|---|---|---|---|---|---|
| z | -3 | -5 | -2 | 0 | 0 | 0 | 0 |
| $s_1$ | 1 | 2 | 1 | 1 | 0 | 0 | 10 |
| $s_2$ | 3 | 2 | 4 | 0 | 1 | 0 | 24 |
| $s_3$ | 2 | 5 | 3 | 0 | 0 | 1 | 30 |

$s_1 = s_1/2$

| Basis | x | y | z | $s_1$ | $s_2$ | $s_3$ | RHS |
|---|---|---|---|---|---|---|---|
| z | -3 | -5 | -2 | 0 | 0 | 0 | 0 |
| y | 1/2 | 1 | 1/2 | 1/2 | 0 | 0 | 5 |
| $s_2$ | 3 | 2 | 4 | 0 | 1 | 0 | 24 |
| $s_3$ | 2 | 5 | 3 | 0 | 0 | 1 | 30 |

$s_2 = s_2 - 2s_1$

$s_3 = s_3 - 5s_1$

$z = z + 5s_1$

| Basis | x | y | z | $s_1$ | $s_2$ | $s_3$ | RHS |
|---|---|---|---|---|---|---|---|
| z | -1/2 | 0 | 1/2 | 5/2 | 0 | 0 | 25 |
| y | 1/2 | 1 | 1/2 | 1/2 | 0 | 0 | 5 |
| $s_2$ | 2 | 0 | 3 | -1 | 1 | 0 | 14 |
| $s_3$ | -1/2 | 0 | 1/2 | -5/2 | 0 | 1 | 5 |

$s_2 = s_2/2$

| Basis | x | y | z | $s_1$ | $s_2$ | $s_3$ | RHS |
|---|---|---|---|---|---|---|---|
| z | -1/2 | 0 | 1/2 | 5/2 | 0 | 0 | 25 |
| y | 1/2 | 1 | 1/2 | 1/2 | 0 | 0 | 5 |
| x | 1 | 0 | 3/2 | -1/2 | 1/2 | 0 | 7 |
| $s_3$ | -1/2 | 0 | 1/2 | -5/2 | 0 | 1 | 5 |

$s_1 = s_1 - \frac{1}{2}s_2$

$s_3 = s_3 + \frac{1}{2}s_2$

$z = z + \frac{1}{2}s_2$

| Basis | x | y | z | $s_1$ | $s_2$ | $s_3$ | RHS |
|---|---|---|---|---|---|---|---|
| z | 0 | 0 | 1.25 | 2.25 | 0.25 | 0 | 28.5 |
| y | 0 | 1 | -0.25 | 0.75 | -0.25 | 0 | 1.5 |
| x | 1 | 0 | 1.5 | -0.5 | 0.5 | 0 | 7 |
| $s_3$ | 0 | 0 | 1.25 | -2.75 | 0.25 | 1 | 8.5 |

=> All coefficients in z are non-negative so the current solution is optimal.

Solution:

$x = 7, y = 1.5, z = 0, P = 28.5$

## Question 5

**Problem:** In a graph $G$, an edge $(u, v)$ is good if every other vertex in $G$ is adjacent to both $u$ and $v$. Starting from a complete graph $K_n$ (a clique of size $n$), what is the minimum number of edges that must be removed so that no "good" edges remain? Also, is determining this minimum number an NP problem?

**(1) Thought Process and Approach**:
The problem begins with a complete graph $K_n$, where every edge is initially "good." To ensure an edge $(u, v)$ is no longer good, at least one other vertex $w$ must not be adjacent to both $u$ and $v$. Thus, strategically removing edges creates gaps in the adjacency structure, invalidating the "good" property for every edge. The goal is to minimize the number of edges removed to achieve this.

For small graphs, it may not be trivial to find a closed-form solution for the minimum number of edges to remove. For larger graphs, the problem becomes significantly more challenging. We aim to "hit" (invalidate) every edge by removing a minimal set of edges. Hitting set problems are generally NP-hard. The decision version, determining whether it is possible to remove $k$ edges to eliminate all good edges, is in NP because verifying a solution (checking if no edges remain good) can be done in polynomial time. This suggests the problem is likely NP-hard but it at least belongs to NP.

**(2) Pseudo-Code to Solve the Decision Version**

(We formulate the decision problem. Given $K_n$ and a number $k$, decide if removing at most $k$ edges can ensure no good edges remain.)

```
function is_feasible_removal(K_n, k):
    # This is a brute force or backtracking pseudo-code for small n.
    # For large n, this is not efficient and only conceptual.

    # edges = all edges of K_n
    # Try subsets of edges of size ≤ k and check condition.

    for subset_of_edges_to_remove in all_subsets_of_size_at_most(edges, k):
        G' = remove_these_edges(K_n, subset_of_edges_to_remove)
        if no_good_edges(G'):
            return True
    return False

function no_good_edges(G):
    for each edge (u,v) in G:
        # Check if there's a vertex w not adjacent to both
        if for all w != u,v: w adjacent to u and w adjacent to v:
```

```
            return False
    return True
```

## (3) Proof of Correctness/Optimality

- **Correctness of Verification:** We examine each remaining edge after removing a set of edges and verifying the adjacency condition; if no edge satisfies the "good" test, the solution is valid. However, finding a minimum solution is challenging, as the exhaustive approach in the pseudocode is only feasible for small cases. For large values of $n$, we would need to rely on complexity classification. Despite this, the verification step is straightforward and ensures correctness.

## (4) Time and Space Complexity Analysis

- **Time Complexity (of the verification step):**
  Checking if an edge is good involves checking its adjacency with all other vertices. For an $n$-vertex graph, there are $O(n^2)$ edges and each check can be $O(n)$, leading to $O(n\text{^}3)$ time to verify. The decision problem (via brute force) is combinatorial and can be $O(\binom{n^2}{k} \cdot n^3)$.
- **Space Complexity:** Storing the graph requires $O(n^2)$ space. Checking adjacency is $O(1)$ if using adjacency matrices, so total space is $O(n^2)$.

**Is the Problem in NP?** Yes. Given a proposed solution (which edges to remove), we can verify in polynomial time that no good edges remain. Thus, the decision version of the problem (given $k$, does such a removal set exist?) is in NP.