SEQUENCING:

```
class Job:
    def _init_(self, id, time_machine_A, time_machine_B):
        self.id = id
        self.time_machine_A = time_machine_A
        self.time_machine_B = time_machine_B

def schedule_jobs(jobs):
    jobs.sort(key=lambda job: min(job.time_machine_A, job.time_machine_B) / 2)

    machineA = []
    machineB = []

    for job in jobs:
        min_time = min(job.time_machine_A, job.time_machine_B) / 2
        if job.time_machine_A <= job.time_machine_B:
            machineA.append(job.id)
        else:
            machineB.append(job.id)

    print("Machine A job sequence:", machineA)
    print("Machine B job sequence:", machineB[::-1])  # Reverse the order for Machine B

def main():
    m = int(input("Enter the number of jobs: "))
    jobs = []

    for i in range(m):
        id = i + 1
        time_machine_A = int(input(f"Enter processing time for job {id} on Machine A: "))
        time_machine_B = int(input(f"Enter processing time for job {id} on Machine B: "))
        jobs.append(Job(id, time_machine_A, time_machine_B))

    schedule_jobs(jobs)
main()
```

SIMPLEX METHOD:

```
import numpy as np

def print_tableau(tableau):
    for row in tableau:
        print("\t".join(f"{val:.2f}" for val in row))
    print()

def simplex_method(tableau):
    m, n = tableau.shape
    while tableau[-1, :-1].min() < 0:
        entering_col = np.argmin(tableau[-1, :-1])
```

```python
        ratios = tableau[:-1, -1] / tableau[:-1, entering_col]
        positive_ratios = np.where(ratios > 0)[0]
        leaving_row = positive_ratios[np.argmin(ratios[positive_ratios])]

        pivot_element = tableau[leaving_row, entering_col]
        tableau[leaving_row, :] /= pivot_element

        for i in range(m):
            if i != leaving_row:
                factor = -tableau[i, entering_col]
                tableau[i, :] += factor * tableau[leaving_row, :]

        print("Pivoting at:", entering_col, leaving_row)
        print_tableau(tableau)

# Get user input for problem
num_constraints = int(input("Enter the number of constraints: "))
num_variables = int(input("Enter the number of variables: "))

# Input matrix A for constraints
A = []
print("Enter the coefficients matrix A ({} x {}):".format(num_constraints,
num_variables))
for _ in range(num_constraints):
    row = list(map(float, input().split()))
    A.append(row)

# Input coefficients for the maximizing function
c = list(map(float, input("Enter the coefficients for maximizing function ({} values):
".format(num_variables)).split()))

# Input constants b for constraints
b = list(map(float, input("Enter the constants b for constraints ({} values):
".format(num_constraints)).split()))

# Create the initial tableau
tableau = np.zeros((num_constraints + 1, num_variables + num_constraints + 1))
tableau[:-1, :num_variables] = A
tableau[:-1, num_variables:num_variables+num_constraints] =
np.eye(num_constraints)
tableau[:-1, -1] = b
tableau[-1, :num_variables] = -c  # Negating the NumPy array c

print("\nInitial Tableau:")
print_tableau(tableau)

# Solve using simplex method
simplex_method(tableau)
```

```
# Output the final constraints and the objective function
print("\nOptimal Constraints:")
constraints = tableau[:-1, -1]
for i in range(num_constraints):
    print("Constraint {}: {:.2f}".format(i + 1, constraints[i]))

print("Optimal Objective Function:", -tableau[-1, -1])
```

ASSIGNMENT

```
import numpy as np

def subtract_min_row(cost_matrix):
    min_row_values = [min(row) for row in cost_matrix]
    for i, row in enumerate(cost_matrix):
        cost_matrix[i] = [value - min_row_values[i] for value in row]
    return cost_matrix

def subtract_min_col(cost_matrix):
    min_col_values = [min(col) for col in zip(*cost_matrix)]
    for i, row in enumerate(cost_matrix):
        cost_matrix[i] = [value - min_col_values[j] for j, value in enumerate(row)]
    return cost_matrix

def assign_zeros(cost_matrix):
    rows, cols = len(cost_matrix), len(cost_matrix[0])
    assignments = []

    while len(assignments) < min(rows, cols):
        # Analyze rows
        for i, row in enumerate(cost_matrix):
            zeros_indices = [j for j, value in enumerate(row) if value == 0]

            if len(zeros_indices) == 1 and zeros_indices[0] not in [a[1] for a in
assignments]:
                j = zeros_indices[0]
                assignments.append((i, j))

                # Cross out other zeros in the column
                for k in range(rows):
                    cost_matrix[k][j] = float('inf')

        # Examine columns
        for j in range(cols):
            zeros_indices = [i for i in range(rows) if cost_matrix[i][j] == 0]
```

```
        if len(zeros_indices) == 1 and zeros_indices[0] not in [a[0] for a in
assignments]:
            i = zeros_indices[0]

            if (i, j) not in assignments:
                assignments.append((i, j))

                # Cross out other zeros in the row
                cost_matrix[i] = [float('inf') for _ in range(cols)]

    return assignments

def is_optimal_assignment(assignments):
    rows = set([a[0] for a in assignments])
    cols = set([a[1] for a in assignments])
    return len(rows) == len(assignments) and len(cols) == len(assignments)

def draw_lines(cost_matrix, marked_rows, marked_cols):
    rows, cols = len(cost_matrix), len(cost_matrix[0])
    lines = [[False] * cols for _ in range(rows)]

    # Highlight unassigned rows
    unassigned_rows = [i for i in range(rows) if i not in marked_rows]
    for i in unassigned_rows:
        lines[i] = [True] * cols

    # Label columns with zeros in marked rows
    for i in marked_rows:
        for j, value in enumerate(cost_matrix[i]):
            if value == 0:
                marked_cols.add(j)

    # Highlight rows with assignments in marked columns
    for j in marked_cols:
        for i in range(rows):
            if cost_matrix[i][j] == 0:
                marked_rows.add(i)

    # Draw lines through unmarked rows and marked columns
    for i in range(rows):
        if i not in marked_rows:
            lines[i] = [True] * cols
        else:
            for j in range(cols):
                if j not in marked_cols:
                    lines[i][j] = True

    return lines
```

```
def subtract_min_uncovered(cost_matrix, lines):
    min_uncovered = min(cost_matrix[i][j] for i in range(len(cost_matrix)) for j in
range(len(cost_matrix[0])) if not lines[i][j])
    for i in range(len(cost_matrix)):
        for j in range(len(cost_matrix[0])):
            if not lines[i][j]:
                cost_matrix[i][j] -= min_uncovered
            elif lines[i][j]:
                cost_matrix[i][j] += min_uncovered
    return cost_matrix

def solve_assignment(cost_matrix):
    rows, cols = len(cost_matrix), len(cost_matrix[0])
    cost_matrix = subtract_min_row(cost_matrix)
    cost_matrix = subtract_min_col(cost_matrix)
    assignments = assign_zeros(cost_matrix)

    while not is_optimal_assignment(assignments):
        marked_rows = set([a[0] for a in assignments])
        marked_cols = set([a[1] for a in assignments])
        lines = draw_lines(cost_matrix, marked_rows, marked_cols)
        cost_matrix = subtract_min_uncovered(cost_matrix, lines)
        assignments = assign_zeros(cost_matrix)

    return assignments

# Get the cost matrix from user input
rows = int(input("Enter the number of rows: "))
cols = int(input("Enter the number of columns: "))

cost_matrix = []
print("Enter the elements of the cost matrix:")
for _ in range(rows):
    row = list(map(int, input().split()))
    cost_matrix.append(row)

# Solve the assignment problem
assignments = solve_assignment(cost_matrix)

# Print the results
print("Final Assignment:")
print("-----------------")
for i, j in assignments:
    print(f"Task {i+1} -> Worker {j+1}")
print("-----------------")
```

NORTHWEST

```
import numpy as np

def northwest_corner_method(supply, demand, costs):
    num_suppliers = len(supply)
    num_consumers = len(demand)

    allocated = np.zeros((num_suppliers, num_consumers))

    supplier_idx = 0
    consumer_idx = 0

    while supplier_idx < num_suppliers and consumer_idx < num_consumers:
        allocation = min(supply[supplier_idx], demand[consumer_idx])
        allocated[supplier_idx][consumer_idx] = allocation

        supply[supplier_idx] -= allocation
        demand[consumer_idx] -= allocation

        if supply[supplier_idx] == 0:
            supplier_idx += 1
        else:
            consumer_idx += 1

    total_cost = np.sum(allocated * costs)

    return allocated, total_cost

# Example input (replace with your actual data)
supply = [50, 40, 60]
demand = [20, 95, 35]
costs = np.array([[5, 8, 4],
                  [6, 6, 3],
                  [3, 9, 6]])

allocated, total_cost = northwest_corner_method(supply, demand, costs)
print("Allocated:")
print(allocated)
print("Total Cost:", total_cost)
```

SIMPLEX METHOD:

```
import numpy as np

def simplex_method(tableau):
    m, n = tableau.shape

    while tableau[-1, :-1].min() < 0:
        entering_col = np.argmin(tableau[-1, :-1])
```

```
        leaving_row = np.argmin(tableau[:-1, -1] / tableau[:-1, entering_col])

        pivot_element = tableau[leaving_row, entering_col]
        tableau[leaving_row, :] /= pivot_element

        for i in range(m):
            if i != leaving_row:
                factor = -tableau[i, entering_col]
                tableau[i, :] += factor * tableau[leaving_row, :]

def main():
    num_constraints = int(input("Enter the number of constraints: "))
    num_variables = int(input("Enter the number of variables: "))

    A = []
    b = []
    c = []

    for _ in range(num_constraints):
        row = list(map(float, input("Enter the coefficients for constraint {} ({} values):
".format(_, num_variables)).split()))
        A.append(row)
        b.append(float(input("Enter the constant for constraint {}: ".format(_))))

    c = np.array(list(map(float, input("Enter the coefficients for the maximizing function
({} values): ".format(num_variables)).split())))

    tableau = np.zeros((num_constraints + 1, num_variables + num_constraints + 1))
    tableau[:-1, :num_variables] = A
    tableau[:-1, num_variables:num_variables+num_constraints] =
np.eye(num_constraints)
    tableau[:-1, -1] = b
    tableau[-1, :num_variables] = -c

    print("\nInitial Tableau:")
    print(tableau)

    simplex_method(tableau)

    print("\nOptimal Solution:")
    print("Constraints:", tableau[:-1, -1])
    print("Objective Function:", -tableau[-1, -1])

if _name_ == "_main_":
    main()
```