```
GaleShapely(womenPreferences{}, menPreferences{})
    While there is a man m who is free and hasn't proposed to every woman
        Choose such a man m
        Let w be the highest-ranked woman in m's preference list to whom m has not yet proposed
        If w is free then
            (m, w) become engaged
        Else w is currently engaged to m'
            If w prefers m to m' then
                (m, w) become engaged
                m' becomes free
            Else
                m remains free

            Endif
        Endif
    Endwhile
    Return the set S of engaged pairs
```

Design Strategy: Greedy

T = O(n^2)

S = due to preference lists O(n)

```
DFS(Graph[], visited[], vertex):
   Mark the current vertex as visited (and print)
   For each neighbor of the current vertex
      If the neighbor has not been visited
         Recursively call DFS for the neighbor
```

Design Strategy: Backtracking

T = O(V+E)

V = O(V+E)

```
BFS(Graph[], visited[], start)
  Mark the starting vertex as visited (and print)
  Append the starting vertex to the queue.
  While the queue is not empty
    Pop a vertex from the queue
    For each neighbor of the current vertex
      If the neighbor has not been visited
        Mark it as visited (and print)
        Append it to the queue
```

Design Strategy: Level-order traversal

T = O(V+E)

S = O(V+E) due to adjacency list

```
Merge(left[], right[])
   Initialize an empty array 'merged'
   Initialize pointers i and j to 0
   While i < length(left) and j < length(right)
      If left[i] < right[j]
         Append left[i] to merged
         Increment i by 1
      Else
         Append right[j] to merged
         Increment j by 1
   Append remaining elements from left[i:] to merged
   Append remaining elements from right[j:] to merged
   Return merged

MergeSort(arr[])
   if length(arr) <= 1
      return arr

   mid = length(arr) // 2
   left_half = MergeSort(arr[0:mid])
   right_half = MergeSort(arr[mid:])

   return Merge(left_half, right_half)
```

Design Strategy: Divide and Conquer

T = O(n log n)

S = O(n) due auxiliary array of size (n) to merge the sorted halves

```
Partition(array[], low, high):
   Set pivot to array[low]
   Set i to low + 1
   For j from low + 1 to high + 1:
      If array[j] <= pivot:
         Swap array[i] with array[j]
         Increment i by 1
   Swap array[i - 1] with array[low]
   Return i - 1

QuickSort(array[], low, high):
   If low < high:
      Set pi to Partition(array[], low, high)
      quickSort(array[], low, pi-1)
      quickSort(array[], pi+1, high)
```

Design Strategy: Divide and Conquer

T = O(n log n) for best and avg case and O(n^2) for worst case

S = due to recursive stack O(log n) for best and avg case and O(n) for worst case

```
merge_and_count(left_arr[], right_arr[]):
    Initialize an empty array 'merged' and two pointers i and j to 0.
    Initialize an inversion count to 0.
    While i < length(left_arr) and j < length(right_arr)
        If left_arr[i] is less than or equal to right_arr[j]
            append left_arr[i] to merged array and increment i by 1
        Else
            append right_arr[j] to merged array, increment j by 1
            add the number of remaining elements in left_arr to inversion count
    Append any remaining elements from left_arr and right_arr to merged array.
    Return merged array and inversion count
```

```
mergeSort_and_count(array):
    If the array has one or zero elements
        Return the array and 0 inversions
    Divide the array into two halves
    Recursively call mergeSort_and_count on each half.
    Merge the two halves using merge_and_count
    Add the inversion counts from the left half, right half, and merge steps
    Return the sorted array and the total number of inversions
```

Design Strategy: Divide and Conquer

T = O(n log n) due to the merge sort process

S = O(n) for the auxiliary arrays used during merging

Dijkstra’s(Graph[], src)

Let S be the set of explored nodes

For each $u \in S$, we store a distance, distance(u)

Initially, S = {src} and distance(src) = 0

While $S \neq V$ do

Select a node $v \notin S$ with at least one edge from S for which

distance'(v) = min_{e=(u,v) : $u \in S$} {distance(u) + weight(e)} is as small as possible

Add v to S

Define distance(v) = distance'(v)

EndWhile

Design Strategy: Greedy

T = O(V^2) without using min-heap

S = O(E+V) due to adjacency list

```
Prims(Graph[], visited[], source, Vertices):
   Initialize visited[source] = True
   Initialize edges = 0 and cost = 0
   While edges < Vertices-1:
      Set minimum = INF, x = 0, y = 0
      For each vertex i from 0 to Vertices-1:
         If visited[i] is True:
            For each vertex j from 0 to Vertices-1:
               If visited[j] is False and Graph[i][j] is  not 0:
                  If Graph[i][j] < minimum:
                     minimum = Graph[i][j], x = i, y = j
      Print edge (x, y) and weight Graph[x][y]
      Set visited[y] as True
      Add Graph[x][y] to cost
      Increment edges by 1
   Print cost
```

Design Strategy: Greedy

T = O(V^2) without using min-heap

S = O(V^2) due to adjacency matrix

Kruskals(Graph, Vertices)

Sort all the edges in non-decreasing order of their weight

Initialize edges and cost to 0

While edges < Vertices - 1

Pick the smallest edge not yet picked

Check if it forms a cycle with the spanning tree formed so far

If the cycle is not formed

Include this edge in the minimum spanning tree

Increment the cost by its edge weight

Design Strategy: Greedy

T = O(E logE) or O(E logV)

S = O(V + E)

Sort the events based on their finish times

Create an array 'p' where $p[i]$ gives previous compatible job

Initialize an array 'M' of size n with 0s, where n is the number of events.M[i] will store the maximum profit achievable considering the first i events

Initialize an empty array 'selected' to store selected intervals

```
Compute_Opt(j)
   if j < 0
      Return 0
   if M[j]
      Return M[j]
   M[j] = max(Compute_Opt(j-1), weight(j) + Compute_Opt(p[j]))
   Return M[j]
```

```
FindSolution(k)
   if k < 0
      Return
   if k == 0 or M[k] != M[k - 1]
      append lst[k] to 'selected' array
      findSolution(p[k])
   else
      findSolution(k - 1)
```

Design Strategy: Dynamic Programming

T = O(n^2)

S = O(n)

```
SubsetSum(weights[], maxWeight, n)
   M = array of size (n+1) x (maxWeight+1) initialized to False, where n is the number of items
   For i = 0 to n
      M[i][0] = True
   For i = 1 to n
      For w = 0 to maxWeight
         If w < weights[i]
            M[i][w] = M[i-1][w]
         Else
            M[i][w] = M[i-1][w] or M[i-1][w - weights[i-1]]
   Return M

FindSubsets(weights[], maxWeight, n, M):
   If M[n][maxWeight] is False:
      Return empty array since No subset with given sum
   Initialize an empty array 'selected_items'
   Initialize w = maxWeight
   For i = n to 0
      If M[i][w] is True and M[i-1][w] is False
         append weights[i-1] to selected_items
         Decrement w by weights[i-1]
   Return selected_items
```

Design Strategy: Dynamic Programming

T = O(n×W)

S = O(n×W)

```
Knapsack(items[], maxWeight, n)
   dp = array of size (n+1) x (maxWeight+1) initialized to 0, where n is the number of items
   For each item i from 1 to n
      For each weight w from 0 to maxWeight
         If the weight of the item i is less than or equal to w
            dp[i][w] is the maximum of not taking the item i (dp[i-1][w]) or taking the item i (dp[i-1][w - weight of item i] + value of item i)
         Else
            dp[i][w] is the same as dp[i-1][w]
   Initialize w to maxWeight
   For each item i from n to 1
      If dp[i][w] is not equal to dp[i-1][w], it means the item i was included in the optimal solution
         Add the item i to the list of selected items and update w to w - weight of item i
   Return maximum value, which is dp[n][maxWeight] and list of selected items
```

Design Strategy: Dynamic Programming

$T = O(n \times W)$

$S = O(n \times W)$

```
BellmanFord(Graph, source, Vertices):
   dist = array of size Vertices initialized to infinity
   dist[source] = 0

   // Relaxation
   for i = 1 to Vertices-1
      for each edge (u, v) with weight w in Graph
         if dist[v] > dist[u] + w
            dist[v] = dist[u] + w

   // Negative Cycle Detection
   for each edge (u, v) with weight w in Graph
      if dist[v] > dist[u] + w then Negative cycle exists
         Return

   Return dist
```

Design Strategy: Dynamic Programming

$T = O(V \times E)$

$S = O(V)$

Initialize sets col, posDiag, and negDiag to track columns and diagonals under attack

Create an empty 'board' of size n×n and an empty list 'res' to store results

Backtrack(r, n)

If all queens are placed (r == n), append the current board configuration to 'res'

For each column c in the current row r,

Check if placing a queen there is safe, a position is safe if c is not in col, r+c is not in posDiag, and r-c is not in negDiag

If safe, place the queen (update sets)

Recursively call backtrack for the next row (r+1)

After the recursive call, remove the queen (update sets)

Design Strategy: Backtracking

T = O(n!) since its a brute force approach

S = O(n×n) due to board

```
TSP(graph[], visited[], current_pos, Vertices, count, cost, path[])
   If all nodes have been visited (count == Vertices) and there is an edge from the current node to the starting node (graph[current_pos][path[0]])
      Return the total cost of the tour and the path (including the return to the starting node)

   ans = infinity
   best_path = empty array

   For i = 0 to Vertices-1

      If visited[i] is False
         Mark visited[i] = True

         new_cost, new_path = TSP(graph[], visited[], i, Vertices, count + 1, cost + graph[current_pos][i], path[] + [i])

         If new_cost < ans
            ans = new_cost
            best_path = new_path

         Mark visited[i] = False
   Return the minimum cost (ans) and the best path (best_path) found.
```

Design Strategy: Backtracking

T = O(n!)

S = O(n×n)