Algorithms

Topological Sort (DFS-based)

```
TopologicalSort(Graph):
  Initialize visited[v] = False for all v
  Initialize stack = empty

  For each node v in Graph:
    If not visited[v]:
      DFS(v)

  Return reversed stack

DFS(v):
  Mark visited[v] = True
  For each neighbor u of v:
    If not visited[u]:
      DFS(u)
  Push v to stack
```

Dijkstras Algorithm (Min Heap)

```
Dijkstra(Graph, source):
  distance[v] =  for all v
  distance[source] = 0
  PriorityQueue pq = [(0, source)]

  While pq not empty:
    (dist, u) = pq.pop()
    For each neighbor v of u with weight w:
      If distance[u] + w < distance[v]:
        distance[v] = distance[u] + w
        pq.push((distance[v], v))

  Return distance[]
```

Prims Algorithm (Minimum Spanning Tree)

```
Prim(Graph):
  mstSet[v] = False for all v
  key[v] =  for all v
  key[0] = 0
  PriorityQueue pq = [(0, 0)]  // (key, vertex)

  While pq not empty:
    (key_u, u) = pq.pop()
    If mstSet[u] is True: continue
```

```
    mstSet[u] = True

    For each (v, weight) in neighbors of u:
      If not mstSet[v] and weight < key[v]:
        key[v] = weight
        pq.push((key[v], v))

  Return key[]
```

Quick Sort

```
QuickSort(arr, low, high):
  If low < high:
    pi = Partition(arr, low, high)
    QuickSort(arr, low, pi - 1)
    QuickSort(arr, pi + 1, high)

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

Gale-Shapley Stable Marriage

```
GaleShapley(men, women):
  All men and women are free
  While there exists a free man m who hasn't proposed to all:
    w = highest-ranked woman on ms list not yet proposed to
    If w is free:
      Pair m and w
    Else if w prefers m over her current partner m':
      Break m' and w
      Pair m and w
    Else:
      w rejects m
  Return pairings
```

Inversion Count (Using Merge Sort)

```
CountInversions(arr):
  Return MergeSort(arr, 0, n - 1)
```

Algorithms

```
MergeSort(arr, left, right):
  If left >= right: return 0
  mid = (left + right) / 2
  count = MergeSort(arr, left, mid)
  count += MergeSort(arr, mid + 1, right)
  count += Merge(arr, left, mid, right)
  Return count

Merge(arr, left, mid, right):
  Merge two halves and count cross-inversions
  Return inversion count
```

Kruskals Algorithm

```
Kruskal(Graph):
  Sort edges by weight
  Initialize DSU for all vertices
  MST = []

  For each edge (u, v, weight) in sorted edges:
    If Find(u) != Find(v):
      Union(u, v)
      Add edge to MST

  Return MST
```

Merge Sort

```
MergeSort(arr, left, right):
  If left >= right: return
  mid = (left + right) / 2
  MergeSort(arr, left, mid)
  MergeSort(arr, mid + 1, right)
  Merge(arr, left, mid, right)

Merge(arr, left, mid, right):
  Merge the two sorted halves into one
```