## Contents



# 1 ./python/graphs/kuhn.py

```python
def dfs(adj, visited, right, left, u):
    visited[u] = True
    for v in adj[u]:
        # If the adj node unpaired then pair it to this.
        # Otherwise try and pair the adj one's pair to somewhere else.
        if (left[v] == -1 or
            (not visited[left[v]] and
             dfs(adj, visited, right, left, left[v]))):
            left[v] = u
            right[u] = v
            return True
    return False
def kuhn(N, M, adj):
    right_pair = [-1] * N
    left_pair = [-1] * M
    visited = [False] * N
    for i in xrange(N):
        if right_pair[i] != -1:
            continue  # Already paired
        visited = [False] * N
        # If false then this node can't be paired.
        dfs(adj, visited, right_pair, left_pair, i)
    return left_pair
```

# 2 ./python/graphs/kruskal.py

```python
# Kruskal's algorithm for minimum spanning tree.
# Runs in O(E*log(E)) time (equiv. O(E*log(V))), where E is the number of
# edges and V is the number of vertices.
# Includes Union-Find.
parent = {}
rank = {}
def make_set(vertex):
    parent[vertex] = vertex
    rank[vertex] = 0
def find(vertex):
    if parent[vertex] != vertex:
        parent[vertex] = find(parent[vertex])
    return parent[vertex]
def union(vertex_a, vertex_b):
    root_a = find(vertex_a)
    root_b = find(vertex_b)
    if root_a != root_b:
        if rank[root_a] > rank[root_b]:
            parent[root_b] = root_a
        else:
            parent[root_a] = root_b
    if rank[root_a] == rank[root_b]:
        rank[root_b] += 1
# Takes an object with the following structure:
# {
#   'edges': [0, 1, 2, 3, ..],
#   'vertices': [(w0, u0, v0), (w1, u1, v1), ...],
# }
# Where w0 = weight of edge from u0 to v0.
def kruskal(graph):
    mst = set()
    edges = list(graph['edges'])
    edges.sort()
    for vertex in graph['vertices']:
        make_set(vertex)
    for edge in edges:
        weight, vertex_a, vertex_b = edge
        if find(vertex_a) != find(vertex_b):
            union(vertex_a, vertex_b)
            mst.add(edge)
    return sorted(mst)
```

# 3 ./python/graphs/prim.py

```python
import heapq
def complex_dist(a, b):
    import cmath
    return cmath.polar(a - b)[0]
# Takes a list of points of the form complex(x, y).
# Can be modified to take edge weights if the line marked #metric is changed.
# Runs in O(V*V) time, where V is the number of vertices.
def prims(N, points):
    cost = 0
    pq = [(0, 0)]
    in_tree = [False] * N
    tree_dist = [1000000] * N
    tree_size = 0
    while tree_size < N and pq:
        d, u = heapq.heappop(pq)
        if in_tree[u]:
            continue
        in_tree[u] = True
        cost += d
        tree_size += 1
        for v in range(N):
            if u == v or in_tree[v]:
                continue
            dist = complex_dist(points[u], points[v]) #metric
            if dist > tree_dist[v]:
```

```python
                continue
            tree_dist[v] = dist
            heapq.heappush(pq, (dist, v))
    return cost
```

## 4 ./python/maths/gcd.py

```python
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b%a, a)
# Euler's totient
def phi(n):
    result = 1
    for i in range(2, n):
        if gcd(i, n) == 1:
            result += 1
    return result
```

## 5 ./python/geometry/lines.py

```python
# Return true if line segments (x1, y1)->(x2, y2) and (x3, y3)->(x4, y4)
# intersect.
def intersect(x1, y1, x2, y2, x3, y3, x4, y4):
    denom = ((x1 - x2) * (y3 - y4)) - ((y1 - y2) * (x3 - x4))
    if denom == 0:
        return False
    x = (((y1 - y3) * (x4 - x3)) - ((x1 - x3) * (y4 - y3))) / denom
    y = (((y1 - y3) * (x2 - x1)) - ((x1 - x3) * (y2 - y1))) / denom
    return (x >= 0 and x <= 1) and (y >= 0 and y <= 1)
```

## 6 ./python/geometry/convexhull.py

```python
def convex_hull(points):
    # Dedupe & sort lexicographically.
    points = sorted(set(points))
    # Boring case. No points, or a single point, maybe repeated.
    if len(points) <= 1:
        return points
    # 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross
    # product.
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)
    upper = []
    for p in points[::-1]:
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)
    # Concat lower and upper hulls.
    # Last point of each is omitted as it is repeated at the beginning of the
    # other list.
    return lower[:-1] + upper[:-1]
```

## 7 ./cpp/dp/knapsack.cc

```cpp
#include <vector>
using namespace std;
struct Object {
  Object(int i_, int v, int w) : i(i_), value(v), weight(w) {}
  int i, value, weight;
};
// Runs in O(N * C).
vector<int> Knapsack(int cap, const vector<Object> &objs) {
  vector<vector<int>> values(objs.size() + 1, vector<int>(cap + 1, 0));
  vector<vector<bool>> taken(objs.size(), vector<bool>(cap + 1, false));
  for (auto &item : objs) {
    for (int c = 0; c <= cap; ++c) {
      if (c < item.weight ||
          values[item.i][c - item.weight] + item.value < values[item.i][c]) {
        // Don't take if can't hold or would reduce value.
        values[item.i + 1][c] = values[item.i][c];
      } else {
        values[item.i + 1][c] = values[item.i][c - item.weight] + item.value;
        taken[item.i][c] = true;
      }
    }
  }
  vector<int> taken_items;
  int c = cap;
  for (int i = objs.size() - 1; i >= 0; --i) {
    if (taken[i][c]) {
      taken_items.push_back(i);
      c -= objs[i].weight;
    }
  }
  return taken_items
}
```

## 8 ./cpp/graphs/prim.cc

```cpp
#include <math.h>
#include <functional>
#include <queue>
#include <utility>
#include <vector>
using namespace std;
struct Point {
  Point(double x_, double y_) : x(x_), y(y_), in_tree(false) {}
  double x;
  double y;
  bool in_tree;
};
double Dist(const Point &a, const Point &b) {
  return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}
double PrimsMST(int N, const vector<Point> &pts) {
  typedef pair<double, Point *> CostPoint;
  priority_queue<CostPoint, vector<CostPoint>, greater<CostPoint>> pq;
  pq.emplace(0, &pts[0]);
  double cost = 0;
  int total_nodes = 0;
```

```
while (total_nodes < N) {
  if (pq.top().second->in_tree) {
    pq.pop();
    continue;
  }
  cost += pq.top().first;
  ++total_nodes;
  Point *p = pq.top().second;
  pq.pop();
    p->in_tree = true;
    for (int i = 0; i < N; ++i) {
      if (!pts[i].in_tree)
        pq.emplace(Dist(*p, pts[i]), &pts[i]);
    }
  }
  return cost;
}
```