Graph Traversal Algorithms

- Many problems rely on traversing elements in a graph
  - e.g., UVa 469 – Wetlands of Florida
    - You're given a 2D grid, each cell of which can be “water” or “land”
    - Cells adjacent on the major axes or diagonals are adjacent
    - For a given water (x, y) coordinate on the grid, determine the area of the connected water
  - These problems look hard if you're not familiar with graph traversals
Graph Traversal Algorithms

- Depth-first search
  - The first and most natural way to solve this problem is by visiting every node using recursion
  - As the name implies, visit the furthest nodes from the originating node
  - Perform backtracking
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
dfs(3)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
dfs(3)
dfs(2)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
dfs(3)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
dfs(3)
dfs(4)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
dfs(3)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
dfs(3)
dfs(5)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
dfs(3)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
dfs(1)
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack
dfs(0)
Graph Traversal Algorithms

ArrayList<ArrayList<Integer>> adjList; // prefilled with adjacents

int dfs(int node) { // returns # of nodes visited from node idx
    int res = 0;

    visited[node] = true; // mark this node as visited

    for (int i = 0; i < adjList.get(node).size(); i++) {
        int neighbor = adjList.get(node).get(i);
        if (!visited[neighbor]) {
            res += dfs(neighbor); // add number of dfs nodes visited
        }
    }

    return res+1; // the +1 refers to visiting the present node
}

public static void main(String[] args) {
    System.out.println(dfs(0));
}
Graph Traversal Algorithms

- Breadth-first search
  - Visit nodes closest to the originating node before diving down into the tree
  - Implemented using Queue
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Queue
0
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Queue
1
2
Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Queue
2
3
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Queue
3
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Queue
4
5
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Queue
5
Graph Traversal Algorithms

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Queue
Graph Traversal Algorithms

```java
int bfs(ArrayList<ArrayList<Integer>> adjList) {
    Queue<Integer> q = new LinkedList<Integer>();
    boolean visited[] = new boolean[N]; // keep track of visited nodes

    q.add(0); visited[0] = true; // add to traversal queue and mark
    int count = 1; // example: keep count of nodes traversed

    while (!q.isEmpty()) {
        int node = q.poll();

        for (int i = 0; i < adjList.get(node).size(); i++) {
            int neighbor = adjList.get(node).get(i);
            if (!visited[neighbor]) { // do not visit nodes twice
                q.add(neighbor); // add to traversal queue
                visited[neighbor] = true; // mark as visited
                count++; // visited a new node! Keep count
            }
        }
    }

    return count;
}
```

Minimum Spanning Trees

- Spanning tree
  - Given: a connected, undirected graph $G = (V, E)$
    - $(V$ is the set of vertices, $E$ is the set of edges)
  - A spanning tree is a set of edges that is a tree and “covers” all vertices $V$
    - There can be several trees
  - The spanning tree with the minimum cost (sum of edge weights) is called the Minimum Spanning Tree
Minimum Spanning Trees
Minimum Spanning Trees

A spanning tree
Cost: $4 + 4 + 6 + 6 = 20$
Minimum Spanning Trees

Minimum spanning tree
Cost: 4 + 2 + 6 + 6 = 18
Minimum Spanning Trees

- Kruskal's algorithm for finding the MST
  - Repeatedly finds edges with minimum costs that do not form a cycle
  - Greedy algorithm, provably correct
Minimum Spanning Trees

- Kruskal's algorithm pseudocode

1) Sort edges by increasing weight
2) While there are unprocessed edges left
   1) Pick an edge $e$ with minimum cost
   2) If adding $e$ to the MST does not form a cycle, add $e$ to MST
Minimum Spanning Trees

- Kruskal's algorithm pseudocode
  - How to store and sort edges?
    - Using an edge list and Collections.sort
  - How to test for cycles?
    - using disjoint sets and union-find
  - Runtime?
    - Sort: $O(|E| \log |E|)$; Processing: $O(|E|)$
    - Total: $O(|E| \log |E|) = O(|E| \log |V|)$
Minimum Spanning Trees

Weighted adjacency list by (index, weight)
0: (1, 4), (2, 4), (3, 6), (4, 6)
1: (0, 4), (2, 2)
2: (0, 4), (1, 2), (3, 8)
3: (0, 6), (2, 8), (4, 9)
4: (0, 6), (3, 9)

Pick smallest edge
No cycle
Minimum Spanning Trees

Weighted adjacency list by (index, weight)
0: (1, 4), (2, 4), (3, 6), (4, 6)
1: (0, 4), (2, 2)
2: (0, 4), (1, 2), (3, 8)
3: (0, 6), (2, 8), (4, 9)
4: (0, 6), (3, 9)
Minimum Spanning Trees

Algorithm not done! The edge list hasn't yet been exhausted

Weighted adjacency list by (index, weight)
0: (1, 4), (2, 4), (3, 6), (4, 6)
1: (0, 4), (2, 2)
2: (0, 4), (1, 2), (3, 8)
3: (0, 6), (2, 8), (4, 9)
4: (0, 6), (3, 9)

Pick smallest edge
Cycle formed, ignore

Pick smallest edge
Cycle formed, ignore
Minimum Spanning Tree

- Kruskal's code
  - An edge list, a sort, and union-find
    - Follows...
class Edge implements Comparable<Edge> {
    int A, B, w;

    public Edge(int A, int B, int w) {
        this.A = Math.min(A, B);
        this.B = Math.max(A, B);
        this.w = w;
    }

    public int compareTo(Edge e) {
        if (w != e.w) {
            return w < e.w ? -1 : 1;
        } else {
            return 0;
        }
    }
}

class UnionFind {
    int uf[];

    public UnionFind(int size) {
        uf = new int[size];
        for (int i = 0; i < size; i++) uf[i] = i;
    }

    public boolean isSameSet(int A, int B) {
        return find(A) == find(B);
    }

    public void union(int A, int B) {
        uf[find(A)] = find(B);
    }

    public int find(int A) {
        int res = uf[A];
        while (uf[res] != res) res = uf[res];
        return uf[A] = res;
    }
}
Minimum Spanning Tree

ArrayList<Edge> edgeList = parseEdgeList();
Collections.sort(edgeList);

int mstCost = 0;
UnionFind uf = new UnionFind(nVertices);

for (Edge e : edgeList) { // for each edge
    if (!uf.isSameSet(e.A, e.B)) { // if no cycle
        mstCost += e.w; // add it
        uf.union(e.A, e.B);
    }
}

System.out.println(mstCost);
Minimum Spanning Tree

- Prim's algorithm
  - Not covered, see the textbook or Wikipedia for a good overview
  - Also has \( O(|E| \log |V|) \) runtime
Single source shortest paths

- Classic problem in computer science
  - Given a node on a graph, find the shortest paths to all other nodes
Single source shortest paths

- For **undirected, unweighted** graphs:
  - Use BFS!
  - E.g., Uva 336 (A Node Too Far)
    - Given an undirected and unweighted graph $G = (V, E)$ and a vertex $v$ in $V$, find the number of nodes unreachable in $n$ hops
Single source shortest paths

From node 5, find # of nodes > 3 hops away
Single source shortest paths

Queue
5
Distances
D[5] = 0

From node 5, find # of nodes > 3 hops away
Single source shortest paths

Queue       Distances
1          D[5] = 0

From node 5, find # of nodes > 3 hops away
Single source shortest paths

Queue
0
2
9
11

Distances
D[5] = 0
D[1] = 1
D[6] = 1
D[10] = 1
D[0] = D[1] + 1 = 2

From node 5, find # of nodes > 3 hops away
From node 5, find # of nodes > 3 hops away

Queue
3
4
8
12

Distances
D[5] = 0
D[1] = 1
D[6] = 1
D[10] = 1
D[0] = 2
D[2] = 2
D[9] = 2
D[11] = 2
D[4] = D[0] + 1 = 3
Single source shortest paths

From node 5, find # of nodes > 3 hops away

Queue
7

Distances
D[5] = 0
D[1] = 1
D[6] = 1
D[10] = 1
D[0] = 2
D[2] = 2
D[9] = 2
D[11] = 2
D[3] = 3
D[4] = 3
D[8] = 3
D[12] = 3
Single source shortest paths

From node 5, find # of nodes > 3 hops away

Queue

Distances
D[5] = 0
D[1] = 1
D[6] = 1
D[10] = 1
D[0] = 2
D[2] = 2
D[9] = 2
D[11] = 2
D[3] = 3
D[4] = 3
D[8] = 3
D[12] = 3
D[7] = 4

Answer: 1
Single source shortest paths

- For undirected, unweighted graphs:
  - Will the same method as above work?
    - Yes
Single source shortest paths

- For directed, weighted graphs:
  - *Without negative weights*
  - Use Dijkstra's! $O((|V| + |E|) \log |V|)$
  - This can be done using a priority queue
    - Works kind of like a greedy, modified BFS
  - Shown by example...
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
{ (0, 2) }

Start from node 2

Distance table:

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[i]</td>
<td>INF</td>
<td>INF</td>
<td>0</td>
<td>INF</td>
<td>INF</td>
</tr>
</tbody>
</table>
Single source shortest paths

Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
{(0, 2)}
{(2, 1), (6, 0), (7, 3)}

Add all unvisited nodes from node 2 to the priority queue. The PQ sorts the distances so the “next closest” node floats to the top. Right now the closest node is 1, followed by 0, then 3.
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
{(0, 2)}
{(2, 1), (6, 0), (7, 3)}
{(5, 3), (6, 0), (7, 3), (8, 4)}

Poll from the PQ to get node 1.
Add all neighboring nodes to node 1 that haven't been polled yet.
BUT be sure to add all nodes that may already be in the queue with longer distances – there may be a shorter way to reach them
Single source shortest paths

Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
{(0, 2)}
{(2, 1), (6, 0), (7, 3)}
{(5, 3), (6, 0), (7, 3), (8, 4)}
{(6, 0), (7, 3), (8, 4)}

Poll from the PQ to get node 3.
Since we know there is a faster way to get node 4, don't bother adding node 4 to PQ
Single source shortest paths

Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
{(0, 2)}
{(2, 1), (6, 0), (7, 3)}
{(5, 3), (6, 0), (7, 3), (8, 4)}
{(6, 0), (7, 3), (8, 4)}
{(7, 3), (7, 4), (8, 4)}

Poll from the PQ to get node 0.
Since we know there is a faster way to get node 4, don't bother adding node 4 to PQ
Single source shortest paths

Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
{(0, 2)}
{(2, 1), (6, 0), (7, 3)}
{(5, 3), (6, 0), (7, 3), (8, 4)}
{(6, 0), (7, 3), (8, 4)}
{(7, 3), (7, 4), (8, 4)}
{(7, 4), (8, 4)}

Now the (7, 3) state is ignored because it's been determined that 7 is a longer path than another existing path to node 3
Single source shortest paths

Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)

\{ (0, 2) \}
\{ (2, 1), (6, 0), (7, 3) \}
\{ (5, 3), (6, 0), (7, 3), (8, 4) \}
\{ (6, 0), (7, 3), (8, 4) \}
\{ (7, 3), (7, 4), (8, 4) \}
\{ (7, 4), (8, 4) \}
\{ (8, 4) \}

Nowhere to go, so nothing is added to the PQ

Distance table

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[i]</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)

\{(0, 2)\}
\{(2, 1), (6, 0), (7, 3)\}
\{(5, 3), (6, 0), (7, 3), (8, 4)\}
\{(6, 0), (7, 3), (8, 4)\}
\{(7, 3), (7, 4), (8, 4)\}
\{(7, 4), (8, 4)\}
\{\}

State (8, 4) is ignored because 8 > 7

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[i]</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
Dijkstra's in code

```java
public static void main(String[] args) {
    LinkedHashMap<Integer, LinkedHashMap<Integer, Integer>> adj;
    // State is a pair (dist, index)
    PriorityQueue<State> pq = new PriorityQueue<State>();
    int dist[] = new int[V]; Arrays.fill(dist, 1 << 20); // INF
    pq.add(new State(2, 0)); // Initial state

    while (!pq.isEmpty()) {
        State s = pq.poll();
        if (s.dist == dist[s.index]) { // true if has not been updated
            LinkedHashMap<Integer, Integer> nbors = adj.get(s.index);
            for (Map.Entry<Integer, Integer> e : nbors.entrySet()) {
                int nbor = e.getKey();
                int nborDist = e.getValue();
                if (nborDist + dist[s.index] < dist[nbor]) {
                    // have found a closer path
                    dist[nbor] = nborDist + dist[s.index];
                    pq.add(new State(nbor, nborDist + dist[s.index]));
                }
            }
        }
    }
}
```
Single source shortest paths

- For **undirected, weighted** graphs:
  - Will the same method as above work?
    - Yes
Single source shortest paths

- For directed, weighted graphs with negative weights:
  - Dijkstra's algorithm does not work and will get stuck in an infinite loop if there is a cycle, always finding a better path
  - Instead, use Bellman-Ford algorithm:
    - Repeat the “relaxing” part of Dijkstra's |V|-1 times, regardless of how close the nodes are
Single source shortest paths

ArrayList<ArrayList<Edge>> adjList = parseAdjList();

int dist[] = new int[N]; // Distance from node 0 to each
Arrays.fill(dist, 1 << 20); // INF
dist[0] = 0;

for (int i = 0; i < N-1; i++) {
    for (int u = 0; u < N; u++) {
        for (int j = 0; j < adjList.get(u).size(); j++) {
            Edge e = adjList.get(u).get(j);
            // min((distance to j), (distance to u) + (distance from u to j)
            dist[e.B] = Math.min(dist[e.B], dist[u] + e.w);
        }
    }
}
Single source shortest paths

- Why Bellman-Ford works:
  - Performing relaxing on the graph $|V|-1$ times guarantees shortest paths are found
    - Proof omitted
  - If relaxing can happen after $|V|-1$ loops, then a negative cycle exists
  - And it runs in $O(|V| \times |E|)$ time with an adjacency list, much greater than Dijkstra's
All pairs shortest paths

- What happens if you want to find the shortest distance between all pairs of nodes?
  - On a weighted, connected graph, use Floyd Warshall algorithm
  - Implement in ~4 lines of code
  - $O(V^3)$ instead of $N$ Dijkstra's algorithm, which would be $O(V^3 \log V)$
Floyd Warshall in code

// inside int main()
// precondition: m[i][j] contains the weight of edge (i, j)
// or INF if there is no such edge
// (m is an adjacency matrix)

for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            m[i][j] = min(m[i][j], m[i][k] + m[k][j]);

// common error: remember that loop order is k->i->j
Graph algorithms

- Now you've seen the bread and butter of graph algorithms
  - There are many more problems associated with graphs
    - Find the width of a graph, find strongly connected components
  - There are special kinds of graphs and smarter algorithms for them
    - Trees, directed acyclic graphs (DAGs), bipartite graphs, eulerian graphs
Map problems tricks

- Demo in Eclipse
Readings from this class

• Readings:
  • Sections 4.1-4.5
    • Mostly what we went over in class, plus more
  • Look at Table 4.4 in the book to have a brief rundown of what we covered today
Forming Quiz Teams

• Given $2N$ points on a grid, make $N$ pairs so that the sum of the distances of the paired points is minimized

  • $1 \leq N \leq 8$, so 16 points
Forming Quiz Teams

• Recurrence:
  - \( \text{dp[ used grid points ]} = \text{the minimum sum of distances between all remaining grid points} \)
  - The answer is \( \text{dp[ all grid points ]} \)
  - The recursive step is to find the minimum sum by trying matching each pair of remaining grid points
  - There are a lot of overlapping states, so store the subresults
Forming Quiz Teams

• Example:

(2, 2)
(4, 6)
(5, 2)
(5, 4)
(6, 8)
(7, 5)
(8, 3)
(8, 6)
Forming Quiz Teams

• Example:

Recursion depth 0

What is the minimum sum for all the points?

(2, 2)
(4, 6)
(5, 2)
(5, 4)
(6, 8)
(7, 5)
(8, 3)
(8, 6)
Forming Quiz Teams

• Example:

Recursion depth 1
What is the minimum sum for the 2nd, 3rd, 4th, 5th, 6th, and 7th points?
Forming Quiz Teams

• Example:

Recursion depth 2

What is the minimum sum for the 4th, 5th, 6th, and 7th points?
Forming Quiz Teams

• Example:

Recursion depth 3

What is the minimum sum for the 6th and 7th points?
Forming Quiz Teams

• Example:

Recursion depth 4

What is the minimum sum no points?

Answer: Base case, 0
Forming Quiz Teams

- Example:

Recursion depth 3

What is the minimum sum for the 6th and 7th points?

Answer: $0 + \text{dist}(P[6], P[7]) = 3$
Forming Quiz Teams

- Example:

Recursion depth 2

What is the minimum sum for the 4th, 5th, 6th, and 7th points?

So far: $3 + \text{dist}(P[4], P[5]) = 3 + 3.16 = 6.16$
Forming Quiz Teams

• Example:

Recursion depth 3

What is the minimum sum for the 5th and 7th points?
Forming Quiz Teams

- Example:

Recursion depth 3

What is the minimum sum for the 5th and 7th points?

Answer: $0 + \text{dist}(P[5], P[7]) = 1.41$
Forming Quiz Teams

• Example:

Recursion depth 2

What is the minimum sum for the 4th, 5th, 6th, and 7th points?

From before: 6.16

Just computed: $1.41 + \text{dist}(P[4], P[6]) = 1.41 + 5.39 = 6.80$

Still: 6.16
Forming Quiz Teams

• Example:

Recursion depth 3

What is the minimum sum for the 5th and 6th points?
Forming Quiz Teams

- Example:

Recursion depth 3

What is the minimum sum for the 5th and 6th points?

Answer: 2.24
Forming Quiz Teams

• Example:

Recursion depth 2

What is the minimum sum for the 4th, 5th, 6th, and 7th points?
From before: 6.16
Just computed: $2.24 + \text{dist}(P[4], P[7]) = 5.07$
Now: 5.07
Forming Quiz Teams

- Example:

Recursion depth 2

What is the minimum sum for the 4th, 5th, 6th, and 7th points? *We have tried everything, so we can definitively answer 5.07.*
Forming Quiz Teams

• Example:

Recursion depth 1

What is the minimum sum for the 2nd, 3rd, 4th, 5th, 6th, and 7th points?

So far: 5.07 + dist(P[2], P[3]) = 7.07
Forming Quiz Teams

• Fast-forward...
Forming Quiz Teams

• Example:

Recursion depth 1

What is the minimum sum for the 1st, 3rd, 4th, 5th, 6th, and 7th points?
Forming Quiz Teams

• Example:

Recursion depth 2

What is the minimum sum for the 4th, 5th, 6th, and 7th points?
Forming Quiz Teams

• Example:

What is the minimum sum for the 4th, 5th, 6th, and 7th points?

Answer: 5.07 (from the memoization table)
Forming Quiz Teams

- How to implement:
  - Use bitmasks!
  - Use a memoization table with $2^{16}$ elements
  - Each entry in the table is considered a bitmask representing the set of all grid points chosen
- Another similar DP solution is the $O(2^n \times n)$ solution to the Traveling Salesman Problem
Forming Quiz Teams

int N;
int x[] = new int[16], y[] = new int[16]; // grid coordinates
double dp[] = new double[1 << 16]; // 2^16 entries

public double solve(int mask) {
    if (dp[mask] >= 0) return dp[mask]; // memoization step
    double res = INFINITY;

    for (int i = 0; i < 2*N; i++) {
        for (int j = i+1; j < 2*N; j++) { // filters out permutations
            if (((1 << i) | (1 << j)) & mask) == 0) { // unused set elmnts
                double dist = sqrt(pow(x[i] – x[j], 2) + pow(y[i] – y[j], 2));
                res = min(res, solve(mask | (1 << i) | (1 << j));
            }
        }
    }
    return dp[mask] = res; // store the solution in memo table
}

public void main() { // left out the parsing details
dp[(1 << (N*2)) – 1] = 0.0; // base case: all points used = 0 min dist
System.out.printf(solve(0)); // 0 = empty bit mask = all points rem}