Informatik-Algorithmen - Caparica Chaputa

Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Daniel Parreira Eduardo Marques Pedro Borges

1 de Fevereiro de 2014

Conteúdo

1 Miscelaneous

1.1 Notes

1 - Verify program limits regarding the data type (is `int` enough? or `long`?)
2 - Verify possible problems with precision

1.2 Limits

	Maximum	Minimum
long	9223372036854775807	-9223372036854775808
int	2147483647	-2147483648

1.3 Fast I/O

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

BufferedWriter out = new BufferedWriter(new OutputStreamWriter(System.out));
OutputStream out = new PrintStream(new BufferedOutputStream(System.out));
//or
StringBuffer out = new StringBuffer(LARGE_CONST);
out.append(...);
out.flush();
out.close();
```

2 Searching

2.1 Median - Minimizing the average of the absolute deviations

```
int[] arr = new int[](n) ;
int median_idx = (int)(arr.length/2); /* if odd, rounds down */
int median = arr[median_idx];
```

2.2 Binary search

```
int binary_search(int A[], int key, int imin, int imax)
{
  if (imax < imin)
    return KEY_NOT_FOUND;
  else
    {
      int imid = midpoint(imin, imax);

      if (A[imid] > key)  // key is in lower subset
        return binary_search(A, key, imin, imid-1);
      else if (A[imid] < key)  // key is in upper subset
        return binary_search(A, key, imid+1, imax);
      else  // key has been found
        return imid;
    }
}
```

2.3 Ternary search - Finding the maxi/minimum of a (monotone) function

```
def ternarySearch(f, left, right, absolutePrecision):
    #left and right are the current bounds; the maximum is between them
    if (right - left) < absolutePrecision:
        return (left + right)/2

    leftThird = (2*left + right)/3
    rightThird = (left + 2*right)/3

    if f(leftThird) > f(rightThird):
        return ternarySearch(f, leftThird, right, absolutePrecision)
    else:
        return ternarySearch(f, left, rightThird, absolutePrecision)
```

2.4 Backtracking using DFS

```
boolean finished = false ;

backtrack(int a[], int depth) {
   if(is_solution(a,depth)) {
      process_solution(a,depth) ;
      finished = true ;
   }
   else {
```

```
      List children = construct_children(a,depth) ;
      for(C c : children) {
         make_move(a,depth) ;
         backtrack(a,depth+1) ;
         unmake_move(a,depth) ;

         if(finished)
            return ; /* terminate early */
      }
   }
}
```

2.5 A*

```
private static State a_star(int[][] puzzle) throws InterruptedException {
   PriorityQueue<State> q = new PriorityQueue<State>() ;
   q.add(new State(puzzle)) ;

   if(q.peek().isGoal())
      return q.peek() ;

   while(!q.isEmpty()) {
      State state = q.poll() ;
      for(State possible_move : state.generateMoves()) {
         if(possible_move.isGoal())
            return possible_move ;
         else
            if(state.moves.size()+state.heuristic()>50)
               continue ;
            q.add(possible_move);
      }
   }

   return null;
}

static class State implements Comparable<State> {
   public List<State> generateMoves()  ...
   private int heuristic() ...

   @Override
   public int compareTo(State arg0) {
      int this_score = this.moves.size()+this.heuristic() ;
      int arg0_score = arg0.moves.size()+arg0.heuristic() ;
      if(this_score==arg0_score)    return 0 ;
      else return this_score<arg0_score ? -1 : 1 ;
   }
}
```

3 Number Theory

3.1 Combinations

```
/*
```
$C_k^n = \frac{n!}{k!(n-k)!}$
```
*/
```

3.2 Arranges (without repetition)

```
/*
```
$A_k^n = \frac{n!}{(n-k)!}$
```
*/
```

3.3 Greatest Common Divisor

```
static int gcd(int a, int b) {
   while(b != 0) {
      int t = b;
      b = a % b;
      a = t;
   }
   return a;
}

// algoritmo de euclides extendido
static long[] extended_gcd(int a, int b) {
   int x = 0, lastY = x;
   int lastX = 1, y = lastX;
   while(b != 0) {
      int quotient = a / b;
      int tmp = a % b;
      a = b;
      b = tmp;

      tmp = lastX - quotient*x;
      lastX = x;
      x = tmp;

      tmp = lastY - quotient*y;
      lastY = y;
      y = tmp;
   }
   return new int[]{lastX, lastY};
}
```

3.4 Least common multiple

$lcm(a, b) = ab/gcd(a, b)$

3.5 Equação Diofantina Linear

1) Existe uma solução para $ax + by = c$, se $d = mdc(a, b)$ divide c
2) Se $d|c$, então determinando o u e v de $ua + vb = d$, consegue-se encontrar uma solução se atribuirmos

$$x_0 = uc/d \text{ and } y_0 = vc/d$$

Sendo todas as outras soluções dadas por

$$x = x_0 + (b/d)t,\ y = y_0 - (a/d) * t \text{ para } t \in \mathbb{Z}$$

3.6 Linear congruency

```
static lonb mod(long x, long m) {
   return x % m + (x < 0) ? m : 0;
}

// Acha a menor solucao nao negativa de a * x = b. Retorna -1 se for impossivel
static long solve_mod(long a, long b, long m) {
   if (m < 0)
      return solve_mod(a, b, -m);
   else if(a < 0 || a >= m || b < 0 || b >= m)
      return solve_mod(mod(a, m), mod(b, m), m);
   else {
      long[] t = extended_gcd(a, m);
      long d = t[0] * a + t[1] * m;
      if(b % d != 0)
         return -1;
      else
         return mod(t[0] * (b / d), m);
   }
}
```

3.7 Sieve of Erastóstenes

```
static boolean[] primes(int upTo) {
   boolean[] sieve = new boolean[upTo];
   Arrays.fill(sieve, true);

   sieve[0] = sieve[1] = false;
   for(int i = 2; i * i < upTo; i++)
      if(sieve[i])
         for(int j = i * i; j < upTo; j += i)
            sieve[j] = false;

   return sieve;
}
```

3.8 Fibonacci Numbers

```
static long fib(int n) {
   long[] fibs = new long[n+1];
```

```
  fibs[0] = 1;
  fibs[1] = 1;
  for(long i = 2; i <= n; i++)
     fibs[i] = fibs[i-1]+fibs[i-2];
  return fibs[n];
}
```

3.9 Propriedades de Matrizes

$$A^1 + ... + A^{n-1} + A^n = (A^1 + ... + A^{n/2}) + A^{\frac{n}{2}}.(A^1 + ... + A^{\frac{n}{2}})$$

3.10 Prime Factorization

4 Dynamic Programming

4.1 Unbounded Knapsack

$$m[0] = 0$$

$$m[w] = max_{w_i \leq w}(v_i + m[w - w_i])$$

4.2 Knapsack 0-1

```
static int knapsack_0_1(int capacity, int n, int[] weights, int[] values) {
   int[][] m = new int[n + 1][capacity + 1];
   for (int i = 0; i < capacity; i++)
      m[0][i] = 0;
   for (int i = 0; i < n; i++)
      m[i][0] = 0;
   for (int i = 0; i < n; i++)
      for (int j = 0; j <= capacity; j++) {
         if (weights[i] > j)
            m[i + 1][j] = m[i][j];
         else
            m[i + 1][j] += Math.max(m[i][j], m[i][j - weights[i]]
                  + values[i]);
      }
   return m[n][capacity];
}

static boolean[] knapTrace(int[][] knapsack, int[] weights, int n) {
   boolean[] solution = new boolean[n];
   int i = knapsack.length - 1;
   int j = knapsack[0].length - 1;
   for (int p = 0; p < n; p++)
      solution[p] = false;
   while (i > 0 && knapsack[i][j] != 0) {
      int val = knapsack[i][j];
      while (val == knapsack[i - 1][j])
         i--;
      solution[i - 1] = true;
      j -= weights[i - 1];
      i--;
   }
   return solution;
}
```

5 Graphs

5.1 Minimum Spanning Tree

5.1.1 Kruskal (when more vertices than edges)

```
List<Edge> mstKruskal(List<Edge> edges, int numVertices) {
   int mstSize = numVertices - 1;
   DisjointSetForest ds = new DisjointSetForest(numVertices);
   PriorityQueue<Edge> queue = new PriorityQueue<Edge>();
   queue.addAll(edges);
   List<Edge> mst = new ArrayList<Edge>(mstSize);
   while(mst.size() < mstSize) {
      Edge temp = queue.poll();
      int leader1 = ds.find(temp.source);
      int leader2 = ds.find(temp.dest);
      if(leader1 != leader2) {
         mst.add(temp);
         ds.union(leader1, leader2);
      }
   }
   return mst;
}

class Edge implements Comparable<Edge>{
   int origin;
   int destin;
   int weight;

   public Edge(int origin, int destin, int weight) {
      super();
      this.origin = origin;
      this.destin = destin;
      this.weight = weight;
   }

   @Override
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (getClass() != obj.getClass())
         return false;
      Edge other = (Edge) obj;
      if (destin != other.destin)
         return false;
      return true;
   }

   @Override
   public int compareTo(Edge o) {
```

```
        return weight - o.weight;
    }
}
```

5.1.2 Prim (when more edges than vertices)

```
public static List<Edge> prim(List<List<Edge>> graph, int numVertices) {
   boolean[] visited = new boolean[numVertices];
   int[] cost = new int[numVertices];

   PriorityQueue<Edge> connected = new PriorityQueue<Edge>();
   List<Edge> mst = new ArrayList<Edge>();

   for(int i = 0; i < numVertices; i++) {
      visited[i] = false;
      cost[i] = Integer.MAX_VALUE;
   }

   int origin = 0;
   cost[origin] = 0;
   connected.add(new Edge(origin, origin, cost[origin]));
   while(!connected.isEmpty()) {
      Edge e = connected.poll();
      visited[e.destin] = true;
      if(origin != e.destin)
         mst.add(e);
      for(Edge out : graph.get(e.destin)) {
         int destin = out.destin;
         if(!visited[destin] && out.weight < cost[destin]) {
            boolean vertexIsInQueue = cost[destin] < Integer.MAX_VALUE;
            cost[destin] = out.weight;
            if(vertexIsInQueue)
               connected.remove(new Edge(e.origin, destin, cost[destin]));
            connected.add(out);
         }
      }
   }
   return mst;
}
```

5.2 Disjoint-set (Union Find)

```
class DisjointSetForest {
    int[] partition;
    int[] compression;

    DisjointSetForest(int domain) {
        partition = new int[domain];
        compression = new int[domain];
        Arrays.fill(partition, -1);
    }
```

```
    //ONLY union if rep1!=rep2 !!
    void union(int rep1, int rep2) {
        if (partition[rep1] <= partition[rep2]){
            partition[rep1] += partition[rep2];
            partition[rep2] = rep1;
        } else {
            partition[rep2] += partition[rep1];
            partition[rep1] = rep2;
        }
    }

    int find(int element) {
        int node = element;
        int count = 0 ;
        while (partition[node] >= 0) {
            compression[count++] = node ;
            node = partition[node];
        }
        for(int i = 0 ; i<count ; i++)
            partition[compression[i]] = node ;
        return node;
    }
}
```

5.3 Depth First Search (Breadth with Queue instead of Stack)

```
static void dfs(int origin, List<List<Integer>> graph) {
   Stack<Integer> stack = new Stack<Integer>();
   stack.add(origin);
   boolean[] visited = new boolean[graph.size()];
   while (!stack.isEmpty()) {
      Integer v = stack.pop();
      visited[v] = true;
      for (Integer out : graph.get(v))
         if (!visited[out])
            stack.push(out);
   }
}
```

5.4 Shortest Paths from A

5.4.1 Dijkstra (no negative-edges)

```
private static int[] dijkstra(List<Edge>[] adjencency_list, int source) {
   int n = adjencency_list.length;
   int[] dists = new int[n] ;
   int[] previous = new int[n] ;
   int[] via = new int[n] ;
   boolean[] visited = new boolean[n] ;

   PriorityQueue<Edge> q = new PriorityQueue<Main.Edge>(n) ;
   for(int i=0 ; i<n ; i++) {
```

```
      dists[i] = Integer.MAX_VALUE ;
      via[i] = -1 ;
   }

   //initial edge
   dists[source] = 0 ;
   via[source] = source ;
   q.add(new Edge(source,source,dists[source])) ;

   while(!q.isEmpty()) {
      Edge u = q.poll() ;
      visited[u.dst] = true ;

      for(Edge e : adjencency_list[u.dst]) {
         int new_dist = dists[u.dst] + e.weight ;
         //if dist through 'u' plus 'e' edge weight better than previous best,
         //   update shortest path
         if(new_dist < dists[e.dst]) {
            q.remove(new Edge(e.src,e.dst,dists[e.dst])) ;
            dists[e.dst] = new_dist ;
            via[e.dst] = u.dst ;
            q.add(new Edge(e.src,e.dst,dists[e.dst]));
            previous[e.dst] = u.dst ;
         }
      }
   }

   return dists;
}

static class Edge implements Comparable<Edge> {
   final int src, dst, weight;

   Edge(int src_, int dst_, int weight_) {
      this.src = src_ ;
      this.dst = dst_ ;
      this.weight = weight_ ;
   }

   @Override
   public int compareTo(Edge o) {
      if(this.weight==o.weight)
         return 0 ;
      else
         return this.weight<o.weight ? -1 : 1 ;
   }
}
```

5.4.2 Bellman Ford (detects negative cycles)

```
static Pair<int[], int[]> bellmanFord(Edge[] edges, int origin,
      int n) throws NegativeWeightCycleException{
```

```
    int[] via = new int[n];
    int[] length = new int[n];
    for(int i = 0; i < n; i++) {
        via[i] = -1;
        length[i] = Integer.MAX_VALUE;
    }
    length[origin] = 0;
    via[origin] = origin;
    boolean changes = false;
    for (int i = 1; i < n; i++) {
        changes = updateLengths(edges, length);
        if (!changes)
            break;
    }
    // Negative-weight cycles detection.
    if (changes && updateLengths(edges, length))
        throw new NegativeWeightCycleException();
    else
        return new Pair<int[], int[]>(length, via);
}

static boolean updateLengths(Edge[] edges, int[] length, int[] via) {
    boolean changes = false;
    for (Edge e : edges) {
        int startPoint = e.origin;
        int endPoint = e.destin;
        if (length[startPoint] < Integer.MAX_VALUE) {
            int newLength = length[startPoint] + e.weigth;
            if (newLength < length[endPoint]) {
                length[endPoint] = newLength;
                via[endPoint] = endPoints[startPoint];
                changes = true;
            }
        }
    }
    return changes;
}
```

5.5 All-Pairs Shortest Paths - Floyd-Warshall (no negative-cost cycles)

```
void go(List<Edge> edges, int nvertices) {
    //(MAXINT/2)-1 to avoid overflow when we sum two non-existing edges
    int MAXINT = (Integer.MAX_VALUE/2)-1 ;

    //initialize matrix
    int[][] adjency_matrix = new int[nvertices+1][nvertices+1] ;
    for(int i=0 ; i<adjency_matrix.length ; i++)
        Arrays.fill(adjency_matrix[i], MAXINT) ;

    //fill matrix
    for(Edge e : edges)
        adjency_matrix[e.src][e.dst] = e.weight ;
```

```
    floydWarshall(adjency_matrix, nvertices) ;

    //...do something with the resulting matrix
}


void floydWarshall(int[][] adjency_matrix, int nvertices) {
    //k -> middle path between i and j
    for(int k=1 ; k<=nvertices ; k++)
        for(int i=1 ; i<=nvertices ; i++)
            for(int j=1 ; j<=nvertices ; j++) {
                int dist_through_k = adjency_matrix[i][k]+adjency_matrix[k][j] ;
                //if path through k is lower than previous direct path, update
                if(dist_through_k < adjency_matrix[i][j])
                    adjency_matrix[i][j] = dist_through_k ;
            }
}
```

5.6 Max Flow - Edmonds-Karp

```
Pair<Integer, int[][]> edmondsKarp(List<List<Edge>> graph, int source,
        int sink) {
    List<List<Edge>> network = buildNetwork(graph);
    int numVert = network.size();
    int[][] flow = new int[numVert][numVert];
    for (List<Edge> list : graph)
        for (Edge e : list)
            flow[e.origin][e.destin] = 0;

    int[] via = new int[numVert];
    int flowValue = 0;
    int increment;
    while ((increment = findPath(network, flow, source, sink, via)) != 0) {
        flowValue += increment;

        int vertex = sink;
        while (vertex != source) {
            int origin = via[vertex];
            flow[origin][vertex] += increment;
            flow[vertex][origin] -= increment;
            vertex = origin;
        }
    }
    return new Pair<Integer, int[][]>(flowValue, flow);
}

List<List<Edge>> buildNetwork(List<List<Edge>> graph) {
    List<List<Edge>> network = graph.clone();

    for (int i = 0; i < graph.size(); i++)
        for (Edge e : outEdges.get(i))
```

```
            network.get(e.destin).add(new Edge(e.destin, e.source, 0));

    return network;
}

int findPath(List<List<Edge>> network, int[][] flow, int source, int sink,
      int[] via) {
   int numVert = network.size();
   Queue<Integer> waiting = new LinkedList<Integer>();
   boolean[] found = new boolean[numVert];
   for (int i = 0; i < network.size(); i++)
      found[i] = false;
   int[] pathIncr = new int[numVert];
   waiting.offer(source);
   found[source] = true;
   via[source] = source;
   pathIncr[source] = Integer.MAX_VALUE;

   do {
      int origin = waiting.poll();
      for (Edge e : network.get(origin)) {
         int destin = e.destin;
         int residue = e.weight - flow[origin][destin];
         if (!found[destin] && residue > 0) {
            via[destin] = origin;
            pathIncr[destin] = Math.min(pathIncr[origin], residue);
            if (destin == sink)
               return pathIncr[destin];
            else {
               waiting.enqueue(destin);
               found[destin] = true;
            }
         }
      }
   } while (!waiting.isEmpty());
   return 0;
}
```

5.7 Topological Sort

```
public static void topologicalSort(List<List<Integer>> graph,
         List<List<Integer>> graphIncident) {
      Queue<Integer> ready = new ArrayBlockingQueue<Integer>(graph.size());
      int[] inCounter = new int[graph.size()];
      for (int i = 0; i < graph.size(); i++) {
         inCounter[i] = graphIncident.get(i).size();
         if (inCounter[i] == 0) {
            ready.add(i);
         }
      }
      while (!ready.isEmpty()) {
         Integer vertex = ready.poll();
```

```java
      //TREAT(vertex);
      for (Integer w : graph.get(vertex)) {
         inCounter[w]--;
         if (inCounter[w] == 0)
            ready.add(w);
      }
    }
  }
```

5.8 Strongly Connected Components - Tarjan

```java
class Node {
   int id, index = -1, lowlink;

   public Node(int id) { this.id = id; }

   public String toString() {
      return "[" + id + "]";
   }
}

class Tarjan {

   private int index = 0;
   private Deque<Node> stack = new ArrayDeque<Node>();
   private ArrayList<ArrayList<Node>> SCC = new ArrayList<ArrayList<Node>>();
   public static boolean[] inStack;

   public ArrayList<ArrayList<Node>> tarjan(Node v, ArrayList<ArrayList<Node>> list){
      v.index = index;
      v.lowlink = index;
      index++;
      stack.push(v);
      inStack[v.id] = true;
      for(Node n : list.get(v.id)){
         if(n.index == -1){
            tarjan(n, list);
            v.lowlink = Math.min(v.lowlink, n.lowlink);
         } else if(inStack[n.id]){
            v.lowlink = Math.min(v.lowlink, n.index);
         }
      }
      if(v.lowlink == v.index){
         Node n;
         ArrayList<Node> component = new ArrayList<Node>();
         do{
            n = stack.pop();
            inStack[n.id] = false;
            component.add(n);
         }while(n != v);
         SCC.add(component);
      }
```

```
        return SCC;
    }
}
```

5.9 Articulation

```
public class Articulation {
    Set<Integer> art = new HashSet<Integer>();
    int n;
    Node[] graph;
    int num;
    int[] dfn;
    int[] low;
    public Articulation(int vertex) {
        n = vertex;
        graph = new Node[n];
        dfn = new int[n];
        low = new int[n];
    }
    public void init() {
        for (int i = 0; i < n; i++) {
            dfn[i] = low[i] = -1;
        }
        num = 0;
    }
    public void articulation() {
        init();
        articulation(0, -1);
    }

    private void articulation(int check, int parent) {
        int childCount = 0;
        dfn[check] = low[check] = num++;
        for (Node adj = graph[check]; adj != null; adj = adj.link) {
            int w = adj.vertex;
            if (dfn[w] < 0) {
  childCount++;
                articulation(w, check);
                low[check] = (low[check] < low[w]) ? low[check] : low[w];
                if (parent >=0 && low[w] >= dfn[check]) {
                    art.add(check);
                }
            } else if (w != parent) {
      low[check] = (low[check] < dfn[w]) ? low[check] : dfn[w];
            }
        }
        if (parent < 0 && childCount > 1) {
  art.add(check);
        }
    }

    public void add(int x, int y) {
```

```
        Node tt = new Node();
        tt.vertex = y;
        tt.link = graph[x];
        graph[x] = tt;
        tt = new Node();
        tt.vertex = x;
        tt.link = graph[y];
        graph[y] = tt;
    }
}
```

6 Sorting algorithms

6.1 MergeSort

```
private static <T extends Comparable<T>> List<T> mergesort(List<T> list) {
   if(list.size()<=1)
      return list ; //base case: sorted

   int middle = list.size()/2 ;
   List<T> left = new ArrayList<T>() ;
   List<T> right = new ArrayList<T>() ;
   for(int i=0 ; i<middle ; i++)
      left.add(list.get(i)) ;
   for(int i=middle ; i<list.size() ; i++)
      right.add(list.get(i)) ;

   //recursive mergesort
   left = mergesort(left) ;
   right = mergesort(right) ;

   //merge sorted sublists
   return merge(left, right) ;
}

private static <T extends Comparable<T>> List<T> merge(List<T> left, List<T> right) {
   List<T> merged = new ArrayList<T>(left.size()+right.size()) ;
   int left_idx=0 ;
   int right_idx=0 ;
   while(left_idx<left.size() && right_idx<right.size()) {
      T left_value = left.get(left_idx) ;
      T right_value = right.get(right_idx) ;
      //<= to be stable (we always pick left most element)
      if(left_value.compareTo(right_value)<=0) {
         merged.add(left_value) ;
         left_idx++;
      }
      else {
         merged.add(right_value) ;
         right_idx++ ;
      }
   }
   //if one list is not yet exausted
   while(left_idx<left.size())   {
      merged.add(left.get(left_idx++)) ;
   }
   while(right_idx<right.size())   {
      merged.add(right.get(right_idx++)) ;

   }
   return merged;
}
```

6.2 QuickSort

```
private static <T extends Comparable<T>> List<T> quicksort(List<T> list) {
   if(list.size()<=1)
      return list ; //base case: sorted
   //could optimize with insertion sort for lists with less 16 elements
   Collections.shuffle(list) ; //to "mitigate" n^2 worst case
   int pivot_idx = list.size()/2 ; //could be optimized for median
   T pivot = list.get(pivot_idx) ;

   List<T> less = new ArrayList<T>(list.size()) ;
   List<T> greater = new ArrayList<T>() ;
   for(int i=0 ; i<list.size() ; i++)
      if(i!=pivot_idx) {
         T e = list.get(i) ;
         if(e.compareTo(pivot)<=0)
            less.add(e) ;
         else
            greater.add(e) ;
      }

   //recursive calls and concatenation of the results
   return concatenate(quicksort(less), pivot, quicksort(greater)) ;
}

private static <T extends Comparable<T>> List<T> concatenate(List<T> left, T middle, List<T> right) {
   List<T> result = left ;
   result.add(middle) ;
   result.addAll(right) ;
   return result ;
}
```

7 Computational Geometry

7.1 Line equations

$$Ax + By = \alpha$$

- if $Ax + By = 0$, intersected
- if $Ax + By > 0$, above
- if $Ax + By < 0$, under

7.2 Closest Pairs in 2D - Quad Tree

```
public class QuadTree {
   final static int QT_NODE_CAPACITY = 4 ;
   List<Point> points ;

   final int x_center ;
   final int y_center ;

   final int x_right ;
   final int x_left ;
   final int y_bottom ;
   final int y_top ;

   QuadTree parent ;

   QuadTree TopLeft ;
   QuadTree TopRight ;
   QuadTree BottomLeft ;
   QuadTree BottomRight ;

   QuadTree(int x_left, int x_right, int y_top, int y_bottom) {
      this.x_center = (x_right-x_left)/2 + x_left ;
      this.y_center = (y_top-y_bottom)/2 + y_bottom ;
      this.x_right = x_right ;
      this.x_left = x_left ;
      this.y_bottom = y_bottom ;
      this.y_top = y_top ;
      this.points = new ArrayList<Point>(4) ;
   }

   QuadTree(int x_left, int x_right, int y_top, int y_bottom, Collection<Point> initialPoints) {
      this(x_left, x_right, y_top, y_bottom) ;
      points.addAll(initialPoints) ;
   }

   public boolean inRange(Point p) {
      return p.x >= x_left && p.x <= x_right
            && p.y <= y_top && p.y >= y_bottom ;
   }
```

```java
public boolean insert(Point p) {
   if(!inRange(p)) {
      return false ;
   }

   if(points!=null) {
      if(points.size()<=QT_NODE_CAPACITY) {
         points.add(p) ;
         p.container = this ;
      }
      else { //capacity exceded, divide and insert in the subtrees
         subdivide();
         insert(p) ;
      }
   }
   else {
      //insert in the correct subtree
      if(!TopLeft.insert(p))
         if(!TopRight.insert(p))
            if(!BottomLeft.insert(p))
               if(!BottomRight.insert(p))
                  throw new RuntimeException("UPS");
   }

   return true ;
}

// create four children which fully divide this quad into four quads of equal area
public void subdivide() {
   TopLeft    = new QuadTree(x_left,      x_center, y_top,      y_center);
   TopRight   = new QuadTree(x_center,    x_right,  y_top,      y_center);
   BottomLeft = new QuadTree( x_left,      x_center, y_center,   y_bottom);
   BottomRight = new QuadTree(x_center,    x_right,  y_center,   y_bottom);

   List<Point> old_points = points ;
   points = null ;

   //add to subtrees
   for(Point p : old_points) {
      insert(p) ;
   }
}

public Collection<Point> containedPoints() {
   if(points!=null)
      return points ;
   else {
      Collection<Point> list = new ArrayList<Point>() ;
      list.addAll(TopLeft.containedPoints()) ;
      list.addAll(TopRight.containedPoints()) ;
      list.addAll(BottomLeft.containedPoints()) ;
```

```java
            list.addAll(BottomRight.containedPoints()) ;
            return list ;
        }
    }
}

public class Point {
    final double x ;
    final double y ;
    QuadTree container ;
    Point(double x_, double y_) {
        this.x = x_ ;
        this.y = y_ ;
    }

    public double distanceTo_SQRT(Point p) {
        return Math.sqrt((this.x-p.x)*(this.x-p.x) + (this.y-p.y)*(this.y-p.y));
    }

    public double distanceTo(Point p) {
        return (this.x-p.x)*(this.x-p.x) + (this.y-p.y)*(this.y-p.y);
    }


    public Collection<Point> adjacentPoints() {
        Collection<Point> result ;
        QuadTree current = this.container ;

        //check two levels above current tree
        if(current.parent==null) {
            result = current.containedPoints() ;
        }
        else if(current.parent.parent==null) {
            result = current.parent.containedPoints() ;
        }
        else {
            result = current.parent.parent.containedPoints() ;
        }

        result.remove(this) ; //remove itself from result
        return result ;
    }
}
```