**Dijkstra’s**

PriorityQueue<Node> q = new PriorityQueue<>(); Node start = graph[0]; q.add(start); start.shortestPath = 0;

**while** (!q.isEmpty())

{ Node current = q.remove(); current.visited = **true**;

**for** (Edge e : current.edges)

**if** (!e.toNode.visited)

{ e.toNode.shortestPath = Math.*min*(e.toNode.shortestPath , current.shortestPath + e.length);

q.remove(e.toNode); q.add(e.toNode); } }

**UFDS**

**public static int** find(**int** x) {

**return** (*ufds*[x] == x) ? x : (*ufds*[x] = *find*(*ufds*[x])); } **public static void** merge(**int** a , **int** b) { *ufds*[*find*(a)] = *ufds*[*find*(b)]; }

**Kruskal’s**

**for** (**int** x = 0; x < nodes.size() - 1; x++)

{ Edge curr = edges.remove(0);

**if** (*find*(curr.n1) == *find*(curr.n2))

{ x--; **continue**; } **else**

{ totalCost += curr.len; *merge*(curr.n1 , curr.n2); } }

**Longest Increasing Subsequence**

**static int**[] *nums*; **static** ArrayList<ArrayList<Integer>> *maxes*;

**public static void** find(**int** index , ArrayList<Integer> seq) {

**for** (**int** i = index; i < *nums*.length; i++)

{ **if** (seq.isEmpty() || *nums*[i] > seq.get(seq.size() - 1))

{ seq.add(*nums*[i]); *find*(i + 1 , seq); seq.remove(seq.size() - 1); } }

**if** (*maxes*.isEmpty() || seq.size() >= *maxes*.get(*maxes*.size() - 1).size())

{ **if** (*maxes*.isEmpty() || seq.size() > *maxes*.get(*maxes*.size() - 1).size())

*maxes*.clear();

*maxes*.add(**new** ArrayList<>(seq)); } }

**Knapsack**

**static int**[][] *K*;

**public static int** knap(**int** n , **int**[] W , **int**[] V , **int** maxW) {

**for** (**int** i = 0; i < n + 1; i++)

**for** (**int** j = 0; j < maxW + 1; j++) { **if** (i == 0 || j == 0)

*K*[i][j] = 0; **else if** (j - W[i - 1] < 0)

*K*[i][j] = *K*[i - 1][j]; **else** *K*[i][j] = Math.*max*(V[i - 1] + *K*[i - 1][j - W[i - 1]] , *K*[i - 1][j]); }

**return** *K*[n][maxW]; }

**Wheel of Fortune**

**while** (!queue.isEmpty())

{ **int** v = queue.remove(); max = Math.*max*(max , maxLength[v]);

**for** (**int** e : adj[v])

{ maxLength[e] = Math.*max*(maxLength[e] , maxLength[v] + 1); inCount[e]--;

**if** (inCount[e] == 0)

queue.add(e); } }

**Kadane’s Algorithm**

**int**[] nums = **new int**[N];

**for** (**int** i = 0; i < nums.length; i++) nums[i] = scan.nextInt();

**for** (**int** i = 0; i < nums.length; i++)

nums[i] -= cost;

**int** current = 0; **int** max = 0;

**for** (**int** x : nums)

{ **if** (current + x > 0)

current += x; **else**

current = 0;

max = Math.*max*(current , max); }

**static final int *INF*** = Integer.***MAX\_VALUE***;

**int** n = scan.nextInt(); // Nodes **int** e = scan.nextInt(); // Edges **int** q = scan.nextInt(); // Queries

**Floyd-Warshall**

**int**[][] dist = **new int**[n][n]; // Distance from node u to node v **int**[][] next = **new int**[n][n]; // Path reconstruction

**for** (**int** i = 0; i < dist.length; i++) // Initial Distance: infinity

Arrays.*fill*(dist[i] , ***INF***);

**for** (**int** i = 0; i < n; i++)

**for** (**int** j = 0; j < n; j++)

next[i][j] = j; // next[i][endpoint] is next node in path

**for** (**int** i = 0; i < dist.length; i++)

dist[i][i] = 0; // Distance from node to itself (initialized to 0)

**for** (**int** i = 0; i < e; i++) { **int** u = scan.nextInt(); **int** v = scan.nextInt(); **int** w = scan.nextInt();

**if** (w < dist[u][v]) // Ignore non-negative edges from node to itself

dist[u][v] = w; }

**for** (**int** k = 0; k < n; k++) // Floyd–Warshall All Pairs Shortest Path

**for** (**int** i = 0; i < n; i++)

**for** (**int** j = 0; j < n; j++)

{ **if** (dist[i][k] == ***INF*** || dist[k][j] == ***INF***)

**continue**;

**if** (dist[i][j] > dist[i][k] + dist[k][j])

{ dist[i][j] = dist[i][k] + dist[k][j]; next[i][j] = next[i][k]; } }

**for** (**int** k = 0; k < n; k++)

**for** (**int** i = 0; i < n; i++)

**for** (**int** j = 0; j < n; j++)

**if** (dist[i][k] != ***INF*** && dist[k][j] != ***INF*** && dist[k][k] < 0)

dist[i][j] = -***INF***; // Detect negative cycles

**for** (**int** i = 0; i < q; i++) { **int** u = scan.nextInt(); **int** v = scan.nextInt();

**if** (dist[u][v] == ***INF***)

System.***out***.println("Impossible"); **else if** (dist[u][v] == -***INF***)

System.***out***.println("-Infinity"); **else**

System.***out***.println(dist[u][v]); }

**public static int** LIS(**int**[] nums) {

ArrayList<Integer> lisEnds = **new** ArrayList<>();

**if** (nums.length != 0)

lisEnds.add(nums[0]);

**for** (**int** i = 1; i < nums.length; i++)

**O(n log n) LIS Size**

{ **int** index = Collections.*binarySearch*(lisEnds , nums[i]);

**if** (index < 0) // lisEnds does not contain nums[i]

{ index = -index - 1; // index = insertion point

**if** (index == lisEnds.size())

lisEnds.add(nums[i]); **else**

lisEnds.set(index , nums[i]); } }

**return** lisEnds.size(); }

**public** BIT(**int**[] nums) {

BIT = **new int**[nums.length + 1];

**for** (**int** i = 0; i < nums.length; i++)

update(i , nums[i]); }

**Binary-Indexed Tree (Fenwick)**

**public int** rangeSum(**int** i , **int** j) {

**return** getSum(j) - getSum(i - 1); }

**public void** update(**int** i , **int** add) {

i += 1;

**while**(i < BIT.length)

{ BIT[i] += add; i += Integer.lowestOneBit(i); } }

**public int** getSum(**int** i) {

**int** sum = 0; i += 1;

**while**(i != 0) { sum += BIT[i]; i -= Integer.lowestOneBit(i); }

**return** sum; }

**int public static void** main(String[] args) { **int**[] nums = **new int**[10000000]; **int**[] tree = **new int**[nums.length \* 4 + 1]; Arrays.*fill*(nums , 2);

*buildST*(nums , tree , 0 , nums.length - 1 , 0); *update*(tree , 0 , nums.length - 1 , 0 , 567 , 5); *query*(tree , 3 , 1000 , 0 , nums.length - 1 , 0); // prints 2001 (998 \* 2 + 5) } }

buildST(**int**[] nums , **int**[] tree , **int** start , **int** end , **int** node) {

**if** (start == end)

**return** tree[node] = nums[start];

**int** mid = (start + end) / 2;

**int** left = *buildST*(nums , tree , start , mid , 2\*node + 1); **int** right = *buildST*(nums , tree , mid + 1 , end , 2\*node + 2);

**return** tree[node] = left + right; } **int** query(**int**[] tree , **int** q1 , **int** q2 , **int** start , **int** end , **int** node) {

**if** (start >= q1 && end <= q2)

**return** tree[node]; **else if** (end < q1 || start > q2)

**return** 0; // Identity of operation **else**

{ **int** mid = (start + end) / 2; **int** left = *query*(tree , q1 , q2 , start , mid , 2\*node + 1); **int** right = *query*(tree , q1 , q2 , mid + 1 , end , 2\*node + 2);

**return** left + right; } } **void** update(**int**[] tree , **int** start , **int** end , **int** node , **int** index , **int** add) {

**while** (start != end)

{ tree[node] += add; **int** mid = (start + end) / 2;

**if** (index >= start && index <= mid)

{ node = 2\*node + 1; end = mid; } **else**

{ node = 2\*node + 2; start = mid + 1; } }

tree[node] += add; } **Segment Tree**

Output Formatting

%c character

%d decimal (integer) number (base 10)

%e exponential floating-point number

At least five wide printf("'%5d'", 10); ' 10' %f floating-point number

%i integer (base 10)

left-justified printf("'%-5d'", 10); '10 '

%o octal number (base 8)

zero-filled printf("'%05d'", 10); '00010'

%s a string of characters

with a plus sign printf("'%+5d'", 10); ' +10'

%u unsigned decimal (integer) number

%x number in hexadecimal (base 16)

Five-wide, plus sign, left-justified printf("'%-+5d'", 10); '+10 '

%% print a percent sign

\% print a percent sign

**Specifier Description Example** f Display the floating point number using decimal representation 3.1415 e Display the floating point number using scientific notation with e 1.86e6 (same as 1,860,000) E Like e, but with a capital E in the output 1.86E6 g Use shorter of the two representations: f or e 3.1 or 1.86e6 G Like g, except uses the shorter of f or E 3.1 or 1.86E6

100.200 // %.3f, putting 3 decimal places always 100 // %.3g, putting 3 significant figures 3.142 // %.3f, putting 3 decimal places again 3.14 // %.3g, putting 3 significant figures

**import** java.text.\*; **import** java.math.\*;

**static void** customFormat(String pattern , **double** value) {

DecimalFormat myFormatter = **new** DecimalFormat(pattern); String output = myFormatter.format(value); System.***out***.println(output); } *customFormat*("###,###.###", 123456.789); // 123,456.789 *customFormat*("###.##", 123456.789); // 123456.79 *customFormat*("000000.000", 123.78); // 000123.780 *customFormat*("$###,###.###", 12345.67); // $12,345.67

BigDecimal ans = a.divide(b , digits , BigDecimal.***ROUND\_HALF\_UP***);