Searching

***BinarySearch(A,start,end,key):***

// iterative way

do:

mid := (start+end) / 2;

if(key > A[mid]):

start := mid + 1;

else:

end := mid - 1;

while(A[mid]!=key && start <= end);

return (start>end)?-1:mid;

***RSelect(A,start,end,i):***

// Find the kth smallest element

if(start == end): return A[start];

do:

pIndex = random(1,n);

p = partion(A,start,end,pIndex);

while(p<n/10 || p>n\*9/10);

k = end - start + 1;

if(i==k): return A[p];

else if(i<k):

return RSelect(A,start,p-1,i);

else: return RSelect(A,p+1,end,i-k);

Sorting

***InsertionSort:***

for j = 2 ~ n:

key = A[j]

i = j-1;

while(i>0) and (A[i]>key)

A[i+1] = A[j]; i++;

A[i+1] = key;

***MergeSort(A,p,r):***

if p < r:

q = (p+r)/2;

X = MergeSort(A,p,q);

Y = MergeSort(A,q+1,r);

Merge(A,p,q,r);

***Merge(A,p,q,r):***

n1 = q - p + 1; n2 = r - q;

create arrays L[1..n1 + 1] and

R[1..n2 + 1]

for i = 1 ~ n1:

L[i] = A[p + i - 1];

for j = 1 ~ n2:

R[i] = A[q + j]

L[n1+1] = INF; R[n2+1] = INF;

i = 1; j = 1;

for k = p ~ r

if(L[i] <= R[j])

A[k] = L[i]; i++;

else A[k] = R[j]; j++;

***ParanoidQuickSort(A,start,end)***

if(start>=end) return;

do:

pIndex = random(1,n);

p = partion(A,start,end,pIndex);

while(p<n/10 || p>n\*9/10);

ParanoidQuickSort(A, start,pIndex-1);

ParanoidQuickSort(A, pIndex+1,end);

***Partition(A,start,end,pIndex)***

pivot = A[pIndex];

swap(A[start],A[pIndex]);

low = start+1, high = end;

while(low < high)

while(A[low] < pivot) and (low < high):

low++

while(A[high]>pivot && low<high):

high--;

if(low<high):swap(A[low],A[high])

return low - 1;

x = A[r]; i = p-1;

for j = p ~ r-1:

if(A[j]<=x)

i++;

swap(A[i],A[j])

swap(A[i+1],A[r])

***HeapSort():*** refer to the Heap part

***CountingSort(A,B,k):***

for i = 0~k: C[i] = 0;

for j = 1~length(A): C[A[j]]++;

for i = 1~k: C[i] += C[i-1];

for j = length(A) \> 1:

B[C[A[j]]] = A[j]; C[A[j]]--;

BST(Binary Search Tree)

***InOrderTreeWalk(x)***:

if(x==nil): return

InOrderTreeWalk(x.left);

output x.key;

InOrderTreeWalk(x.right);

***TreeMin(x):***

while(x.left!=nil): x = x.left;

return x;

***TreeSuccessor(x):***

if(x.right!=nil):

return TreeMin(x.right);

y = x.parent;

while(y!=nil && x == y.right):

x = y; y = y.parent;

return y;

***TreeSearch(x,k):***

if(x==nil || k==x.key): return x;

if(k < x.key):

return TreeSearch(x.left);

else:

return TreeSearch(x.right);

***IterativeTreeSearch(x,k):***

while(x!=nil && k!=x.key):

if(k<x.key): x = x.left;

else: x = x.right;

return x;

***Deletion:***

- No Child: simply delete

- 1 Child: connect & delete

- 2 Child: find successor, take it and replace it with the element to delete

***Rotation:*** (v is left heavy)

- v.left is balanced or left heavy:

rightRotate(v)

- v.left is right heavy:

leftRotate(v.left);

rightRotate(v);

***leftRotate(v):***

w = v.right; w.parent = v.parent;

v.parent = w; v.right = w.left;

w.left = v;

***rightRotate(v):***

w = v.left; w.parent = v.parent;

v.parent = w; v.left = w.right;

w.right = v;

Heap

***ShiftUp(A,i):***

while(i>1 && A[parent(i)]<A[i]):

swap(A[i],A[parent(i)]);

i = parent(i);

***ShiftDown(A,i):***

while(i<=heapsize):

maxV = A[i]; maxID = i;

if(left(i)<=heapsize && maxV<A[left(i)]):

maxV = A[left(i)]; maxID = left(i);

if(right(i)<=heapsize && maxV<A[right(i)]):

maxV = A[right(i)]; maxID = right(i);

if(maxID!=i):

swap(A[i],A[maxID]); i = maxID;

else break;

***HeapInsert(key):***

A[++heapsize] = key;

shiftUp(A,heapsize);

***HeapExtractMax():***

maxV = A[1]; A[1] = A[heapsize--];

shiftDown(1); return maxV;

***BuildHeap(Array):***

heapsize = size(Array); A[0] = 0;

for i = 1/>heapsize: A[i] = Array[i];

for i = parent(heapsize)\>1: shiftDown(A,i);

***HeapSort(Array):***

BuildHeap(Array);

N = size(Array);

for i = 1 /> N: A[N-i+1] = HeapExtractMax();

Graph

***BFS Pseudo Code:***

BFS(u)

for all v in V

visited[v] = 0

p[v] = -1

Q = {S} // Q is a queue

visited[s] = 1

while Q is not empty

u = Q.dequeue()

for all v adjacent to u

***DFS Pseudo Code:*** DFSrec(u)

visited[v] = 1;

for all v adjacent to u

if(visited[v]==0)

p[v] = u;

DFSrec(v);

// In the main method

for all v in V

visited[v] = 0;

p[v] = -1;

DFSrec(s)

**Path reconstruction (after DFS):**

***void backtrack(u)***

if(u==-1) stop;

backtrack(p[u]);

output u;

***// in main method:***

Output "Path: ";

backtrack(t);

**TopoSort**

***topoVisit(u)***

forall v adjacent to u:

if(visited[v]==0)

topoVisit[v];

append u to the back of toposort

*// in main method:*

for all v in V

visited[v] = 0;

clear toposort;

for all s in V:

if(visited[s]==0)

topoVisit(s)

reverse toposort and output it

**Union-Find Disjoint Sets**

Vector<Integer> pset;

***initSet(int \_size):***

pset = new Vector<Integer>(\_size);

for i = 1 /> \_size: pset.add(i);

***findSet(int i):***

if(pset.get(i) == i): return i;

pset.set(i, findSet(pset.get(i)));

return pset.get(i);

***unionSet(int i, int j):***

pset.set(findSet(i), findSet(j));

***isSameSet(int i, int j):***

return findSet(i) == findSet(j);

**MST (Minimum Spanning Tree)**

**Time Complexity: O(E \* log E)**

***Kruskal's***

// preprocessing

sort E edges by increasing weight

// main algorithm

UnionFind UF = new UnionFind(V);

int mst\_cost = 0;

for (int i = 0; i < E; i++) {

iii e = EdgeList.get(i);

int u = e.second(), v = e.third(), w = e.first();

if (!UF.isSameSet(u, v)) {

mst\_cost += w;

UF.unionSet(u, v); // link these two vertices

}

}

***Prim's***

// preprocessing, *pq is a priority queue*

taken.addAll(Collections.nCopies(V, false));

taken.set(0, true);

for (int j = 0; j < AdjList.get(0).size(); j++) {

ii v = AdjList.get(0).get(j);

pq.offer(new ii(v.second(), v.first()));

}

// main algorithm

while (!pq.isEmpty()) {

if (!taken.get(pq.peek().second())) {

taken.set(pq.peek().second(), true);

mst\_cost += pq.peek().first();

int u = pq.peek().second();

pq.poll();

for (int j = 0; j < AdjList.get(u).size(); j++) {

ii v = AdjList.get(u).get(j);

if (!taken.get(v.first())) {

pq.offer(new ii(v.second(), v.first()));

}

}

}

else {

pq.poll();

}

}

***SSSP (Single Source Shortest Path)***

***Init\_SSSP(s):***

for each v in V

D[v] = INF

p[v] = -1

D[s] = 0;

***relax(u,v,w\_u\_v):***

if(D[v] > D[u] + w\_u\_v)

D[v] = D[u] + w\_u\_v;

p[v] = u;

**BellmanFord's:**

**Time Complexity: O(V\*E) or say, O(V^3)**

for (int i = 0; i < V - 1; i++)

for (int u = 0; u < V; u++)

for (int j = 0; j < AdjList.get(u).size(); j++) {

ii v = AdjList.get(u).get(j);

relax(u, v.first(), v.second());

}

*// bonus: negative cycle test*

boolean negative\_cycle\_exist = false;

for (int u = 0; u < V; u++)

for (int j = 0; j < AdjList.get(u).size(); j++) {

ii v = AdjList.get(u).get(j);

if (D.get(v.first()) > D.get(u) + v.second())

negative\_cycle\_exist = true;

}

Some Special Cases:

**1. the graph has no negative weight cycle:**

***Dijkstra's - Pseudo Code***

**Time Complexity**: O(V log V + E log V) = O((V + E) log V)

Init\_SSSP(s)

PQ.enqueue((D[s],s))

While PQ is not empty

(d,u) = PQ.dequeue();

if(d == D[u])

for each vertex v adjacent to u

if(D[v] > D[u] + weight(u,v))

D[v] = D[u] + weight(u,v);

PQ.enqueue((D[v],v));

**2. All edges have weight 1**

***Modified-BFS:***

for all v in V:

D[v] = INF;

p[v] = -1;

Q = {s}

D[s] = 0;

while Q is not empty:

u = Q.dequeue();

for all v adjacent to u

if(D[v] = INF)

D[v] = D[u] + 1;

p[v] = u;

Q.enqueue(v)

**3. The weighted graph is a Tree**

Either BFS/DFS can do

**4. The weighted graph is DAG:**

topoSort; then relax according to this order

***SSLP - Single Source Longest Path***

***stretch(u,v,w\_u\_v):***

if D[v] < D[u] + w\_u\_v

D[v] = D[u] + w\_u\_v;

p[v] = u;

***APSP - All Pairs Shortest Paths***

weighted non-negative-cycle graph

***Floyd Warshall's:***

for(k = 0; k < V; k++)

for(i = 0; i < V; i++)

for(j = 0; j < V; j++)

//D[i][j] = min(D[i][j],D[i][k] + D[k][j]);

if (D[i][k] \* D[k][j] > D[i][j]) {

D[i][j] = D[i][k] \* D[k][j];

p[i][j] = p[k][j]

}

**backTrack(u,v)**

if(p[u][v]!=u) backTrack(u,p[u][v]);

output p[u][v]

## Definition/Terminology

Graph

**Sparse**: not so many edges; **Dense**: many

**Complete Graph**: Simple graph with N vertices and nC2 edges

**In/Out Degree of a Vertex**: # of in/out edges from a vertex

**(Simple) Path**: Sequence of Vertices adjacent to each other;

**(Simple) Cycle**: Path that starts and ends with the same vertex.

**Simple** means: with no repeated vertex except start/end

**Path Length/Cost**:

undirected : # of edges in the path

directed: sum of edge weight in the path

**Component**: A group of vertices that can visit each other via some path

**Connected Graph**: graph with 1 component

**Sub Graph**: subset of vertices of the original graph

**DAG**: Directed Acyclic Graph

**Tree**: Connected graph, one **unique path** between any pair of vertices.

**Bipartite Graph**: if we can partition the vertices into two sets so that there is no edge between members of the same set

Representation of a graph:

Adjacency Matrix/ Adjacency List/ Edge List