CS2010-Final-Cheatsheet

Programming Methodology (National University of Singapore)

CS2010 AY16/17 Cheat sheet

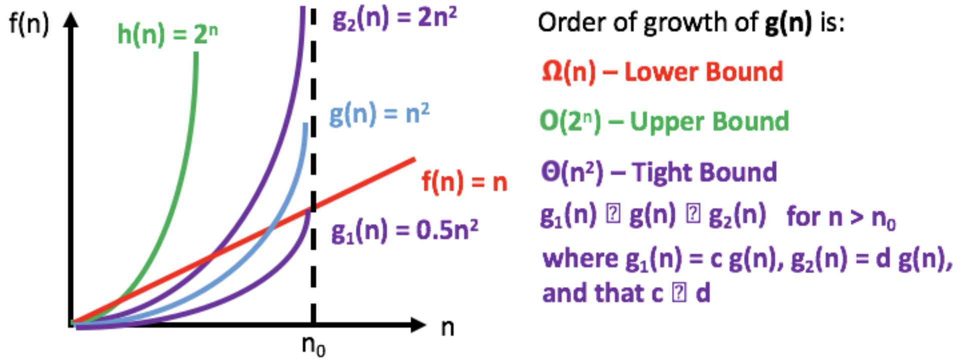
selection: find largest move to end and repeat without the last index.
Bubble sort: swap i and i+1th.
insertion: pick one insert to the right place aka like a poker hand.
merge: divide into subarray and merge back sorted. recursive. Space of N due to temporary arrays required
quick: find pivot, divide into subarray < or > pivot. recursive
stable: relative order of elements with same key value is preserved

Sorting based on **Visualgo**

Sort	Stable	Best				
Bubble	Y	N	N^2	N^2	1	Largest K items sorted at last K positions
Selection	N	N^2	N^2	N^2	1	Smallest K items sorted at first K positions
Insertion	Y	N	N^2	N^2	1	A[1..k] is sorted
Merge	Y	N log n	N log n	N log n	N	Left and right halves are sorted before merging
Quick	N	N	N log n	N^2	1	Left partition < pivot and right partition > pivot

- If two objects with equal keys appear in the same order in **sorted** output as they appear in the input array to be **sorted**.
- **Bubble sort** at every iteration ensure that the largest element will be sorted and put as the last element.
- **Selection sort** at every iteration ensure that the smallest element will be sorted and put as the first element.
- **Insertion sort** at every iteration check if the selected element is smaller or bigger than its previous element.
  - If it is smaller, replace the current index with the previous element, and recursively find the position.
  - Else, the current index is the correct position.
- **Merge sort** recursively split the array into 2 (/4 or /8 …), then it compare its value recursively and sort them accordingly.
- **Quick sort** select an element and create a left partition and right partition.
  - Smaller element will be placed at the left and Larger element will be placed at the right.
  - The selected element will be swap into the middle and the sort recursively on the left partition till the swap value and continue quick sort on a new element.

Time complexity Graph



F(N) = $N + \frac{1}{2}N + \frac{1}{3}N + \frac{1}{4}N + \ldots + 1$ → AP

➔ Summation of F(N) = $n(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots)$ = O(n lg n)

G(N) = $N + \frac{1}{2}N + \frac{1}{4}N + \frac{1}{8}N + \cdots + 1$ → GP

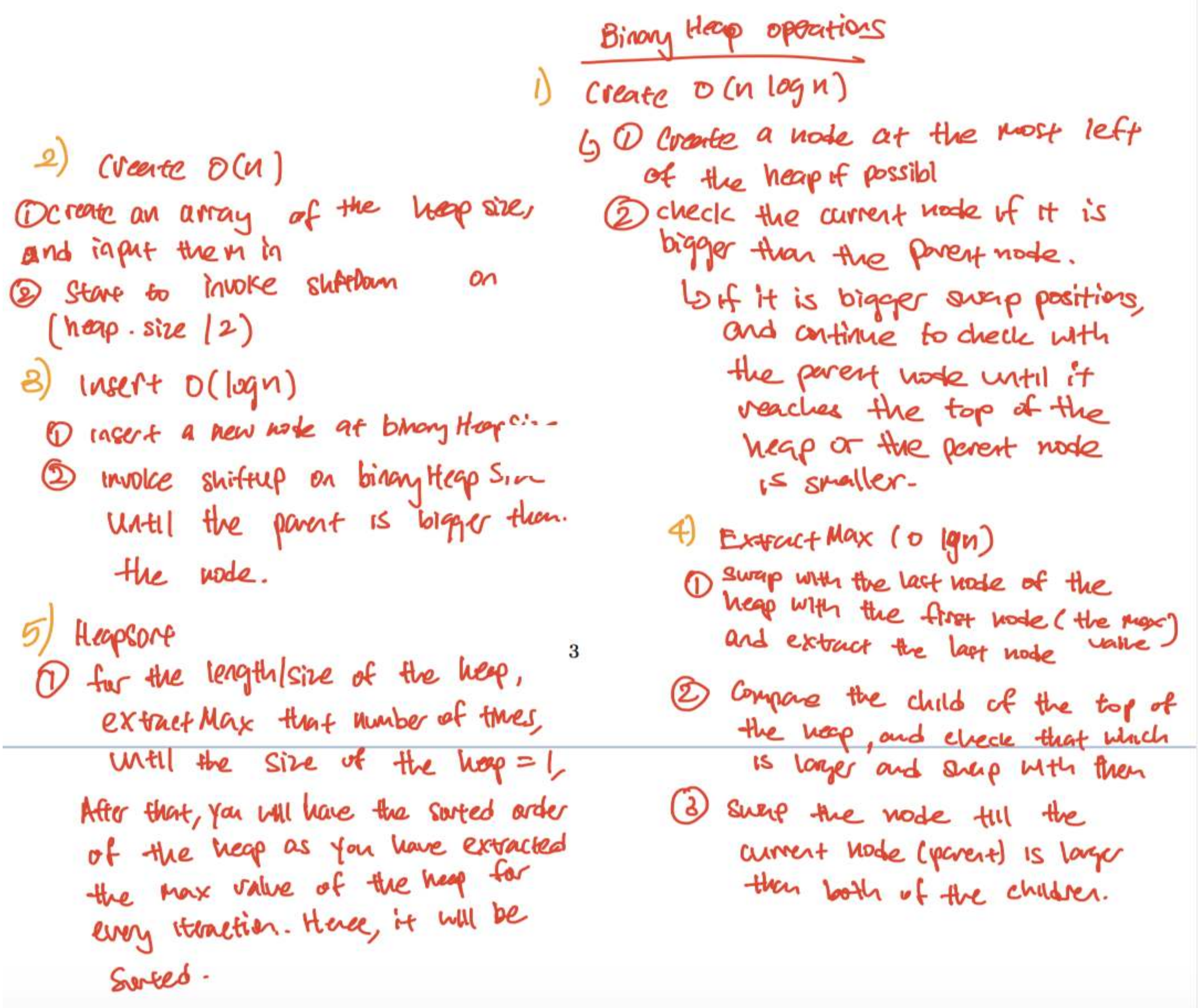
➔ Summation of G(N) = $n(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^{\log n}}) \leq 2n = O(n)$

Good Hash Functions

→ Deterministic

→ Have a simple uniformed function → Avoid Collisions

Binary Heap Operations



Binary Heap operations

1) Create O(n log n)

↳ ① Create a node at the most left of the heap if possibl

② check the current node if it is bigger than the parent node.

↳ if it is bigger swap positions, and continue to check with the parent node until it reaches the top of the heap or the perent node is smaller.

2) Create O(n)

① create an array of the heap size, and input them in

② Start to invoke shiftDown on (heap.size/2)

3) insert O(logn)

① insert a new node at binaryHeap si..

② invoke shiftup on binaryHeap Size until the parent is bigger then the node.

4) ExtractMax (O lgn)

① swap with the last node of the heap with the first node (the max value) and extract the last node

② Compare the child of the top of the heap, and check that which is larger and swap with them

③ Swap the node till the current node (parent) is larger then both of the children.

5) Heapsort

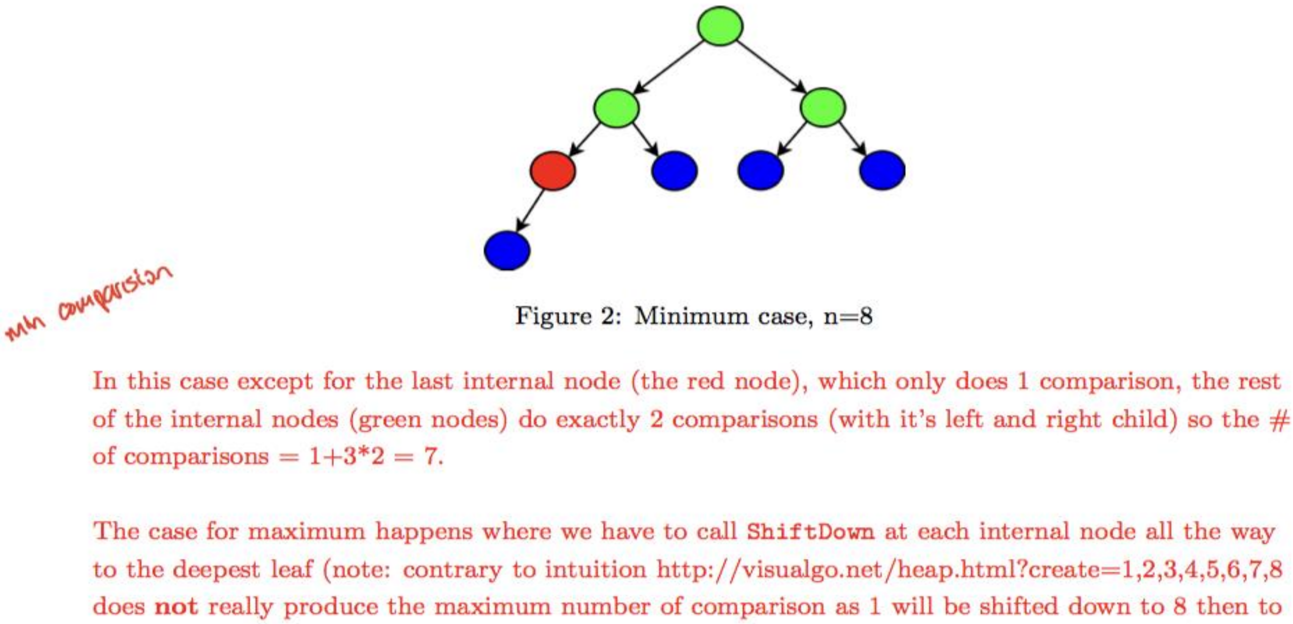
① for the length/size of the heap, extractMax that number of times, until the size of the heap = 1,

After that, you will have the sorted order of the heap as you have extracted the max value of the heap for every iteration. Hence, it will be sorted.

Parent(i)	Find the parent node of node I	Floor(i/2) , expect for I = 1 (root)
Left(i)	Find the left child	**2*I**, no left child when : left(i) > heapsize
Right(i)	Find the right child	**2 *I + 1**, no right child when :right(i)>heapsize
Insert	Add item at end of heap	O(log n)
Find min/max	Find min/max in (min/max) heap	O(1) → Extract the top
Delete min/max	Remove item(highest priority)	O(log n)
Heapify	Convert array to heap	O(n) Need to loop through the whole array
Heap-sort	Convert heap to sorted array	O(n log n)
Update Key	Update priority of an item	O(log n)

- Given a Binary Max Heap, calling ShiftDown(i) ∀i > heapsize/2 will never change anything in the Binary Max Heap. (True)
- The third largest element in a Binary Max Heap that contains > 3 distinct integers is always one of the children of the root? (False)
- The second smallest element in a Binary Max Heap that contains > 3 distinct integers is always at one of the leaves. (False)

Binary Heap Min Comparisons



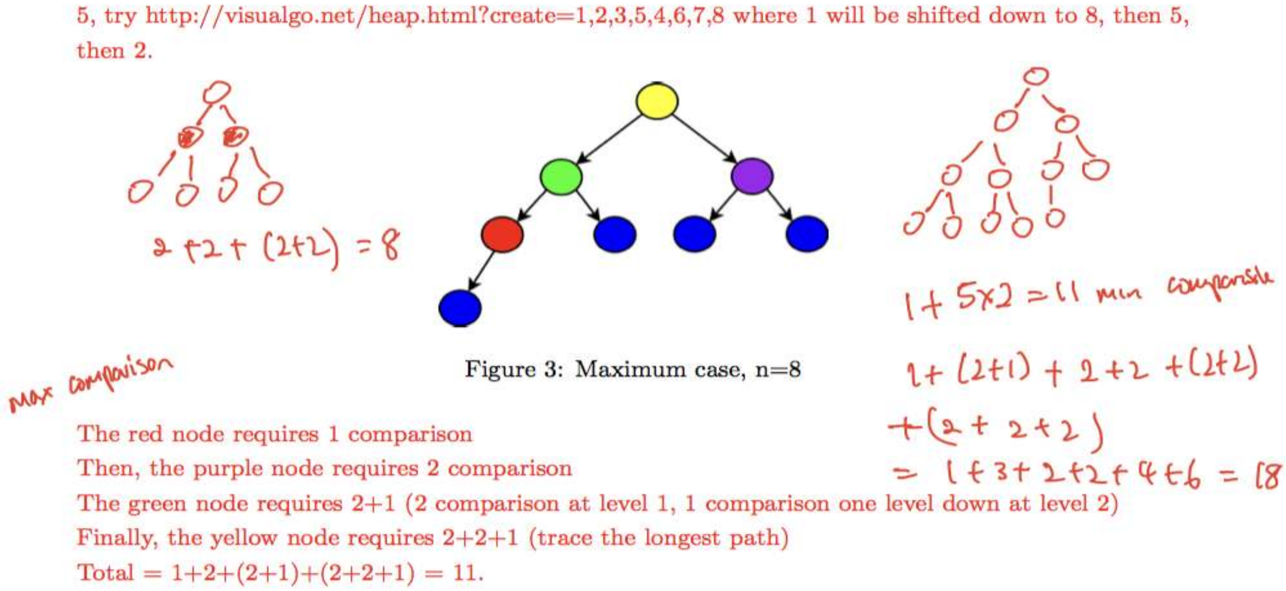
min comparison

Figure 2: Minimum case, n=8

In this case except for the last internal node (the red node), which only does 1 comparison, the rest of the internal nodes (green nodes) do exactly 2 comparisons (with it's left and right child) so the # of comparisons = 1+3*2 = 7.

The case for maximum happens where we have to call `ShiftDown` at each internal node all the way to the deepest leaf (note: contrary to intuition http://visualgo.net/heap.html?create=1,2,3,4,5,6,7,8 does **not** really produce the maximum number of comparison as 1 will be shifted down to 8 then to

Binary Heap Max Comparison



5, try http://visualgo.net/heap.html?create=1,2,3,5,4,6,7,8 where 1 will be shifted down to 8, then 5, then 2.

2 +2 + (2+2) = 8

1+ 5x2 = 11 min comparisn

1+ (2+1) + 2+2 +(2+2) +(2+ 2+2) = 1+3+2+2+4+6 = 18

Figure 3: Maximum case, n=8

max comparison

The red node requires 1 comparison
Then, the purple node requires 2 comparison
The green node requires 2+1 (2 comparison at level 1, 1 comparison one level down at level 2)
Finally, the yellow node requires 2+2+1 (trace the longest path)
Total = 1+2+(2+1)+(2+2+1) = 11.

Binary Heap Max Swap

1 (layer 4)	1
2 (layer 3)	2 3
4 (layer 2)	4 5 6 7
8 (layer 1)	8 9 10 11 12 13 14 15
7 (layer 0)	16 17 18 19 20 21 22

Swap only occurs on bubble down

layer 1 swaps = 1 + 1 + 1 + 1 = 4 (with leaf)
layer 2 swaps = 2 + 2 + 1 + 1 = 6
layer 3 swaps = 3 + 2 = 5
layer 4 swaps = 4
Total: 19

Binary Heap Min Swap → Always 0
(Best case, not require to do any swap. Insert at the right position.)

BST & AVL Tree

Pre Order → Node Left Right
Post Order → Left Right Node
In Order → Left Node Right

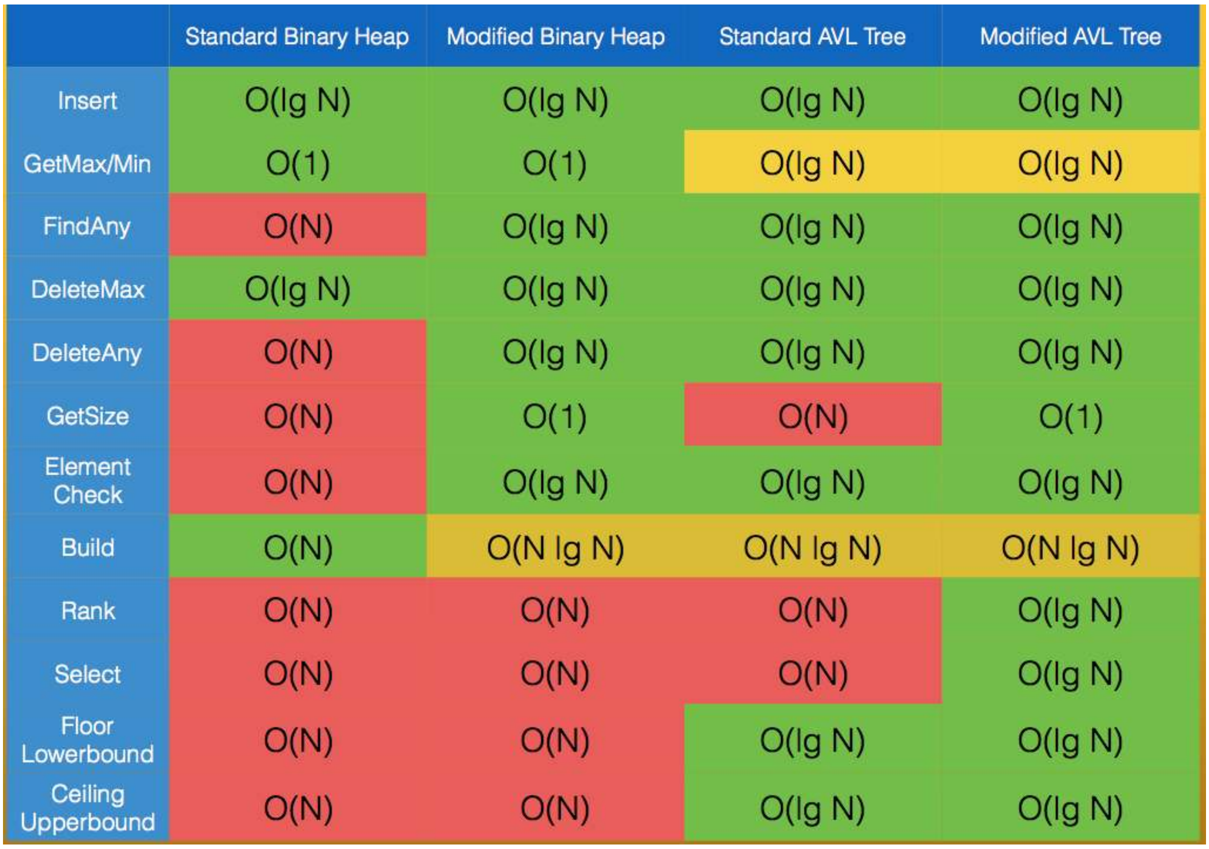
Min Height of BST → Refer to Appendix
Max Height of BST → $Log2\ N$

Different BST with N Distinct Elements → Refer to Appendix

- The insert operation in BST is **always not commutative** in the sense that inserting two distinct elements x and then y into an existing BST (not necessarily balanced) always produce structurally different BST as inserting y and then x. (False)
- The delete operation in BST is **always commutative** in the sense that deleting x and then y from an existing BST (not necessarily balanced) always produces structurally the same BST as deleting y and then x. Note that x ≠ y and both x and y exist in the BST. (False)
- The **smallest** element in any non-empty BST always has **no predecessor**.
- The **largest** element in any non-empty BST always has **no right child**.

***Careful of BST Structure, Visualgo will set tricky questions on validity of BST. ***
Draw the graph if there are sufficient time.

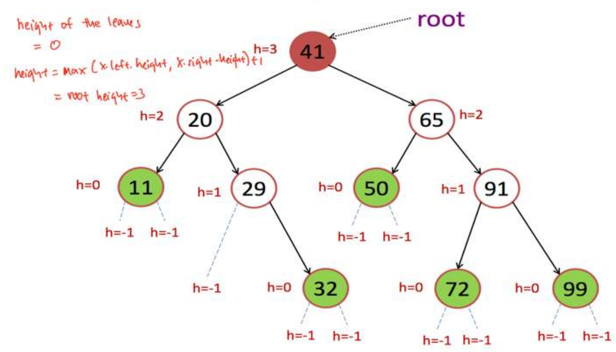
Insert	Add item into BST		**O (h)**	Order matters (balance!)
Search	Find item in BST		**O (h)**	-
Delete	Remove item in BST		**O (h)**	Find successor. / predecessor.
Min	Find min. item in BST		**O (h)**	”Leftmost Child”
Max	Find max. item in BST		**O (h)**	“Rightmost Child”
Successor	Find ”next” element		**O (h)**	-
Predecessor	Find “last” element		**O (h)**	-
Traversal	**In-Order**	Left -> Root -> Right	**O (n)**	Convert BST to sorted list
	Pre-Order	Root -> Left -> Right		Used in tree duplication
	Post-Order	Left -> Right -> Root		Postfix (To-Read: RPN)



	Standard Binary Heap	Modified Binary Heap	Standard AVL Tree	Modified AVL Tree
Insert	O(lg N)	O(lg N)	O(lg N)	O(lg N)
GetMax/Min	O(1)	O(1)	O(lg N)	O(lg N)
FindAny	O(N)	O(lg N)	O(lg N)	O(lg N)
DeleteMax	O(lg N)	O(lg N)	O(lg N)	O(lg N)
DeleteAny	O(N)	O(lg N)	O(lg N)	O(lg N)
GetSize	O(N)	O(1)	O(N)	O(1)
Element Check	O(N)	O(lg N)	O(lg N)	O(lg N)
Build	O(N)	O(N lg N)	O(N lg N)	O(N lg N)
Rank	O(N)	O(N)	O(N)	O(lg N)
Select	O(N)	O(N)	O(N)	O(lg N)
Floor Lowerbound	O(N)	O(N)	O(lg N)	O(lg N)
Ceiling Upperbound	O(N)	O(N)	O(lg N)	O(lg N)

Height of BST

Binary Search Trees: Height (h)

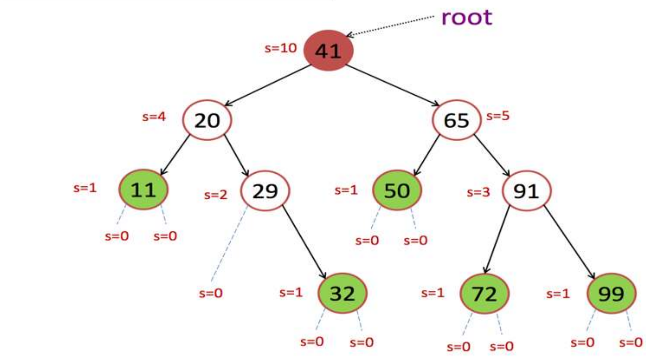


A vertex x is said to be height-balanced if:

|x.left.height - x.right.height| ≤ 1

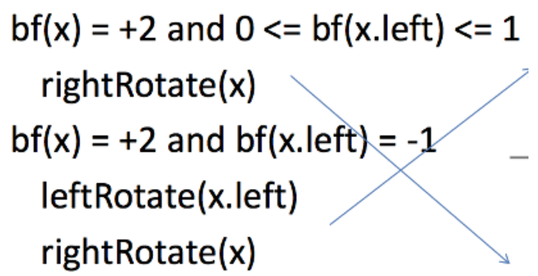
Size of BST

Binary Search Trees: Size (s)



Balance Factor (x):
x.left.height – x.right.height

Once we have a vertex of balance factor of +2 or -2, have to rebalance it



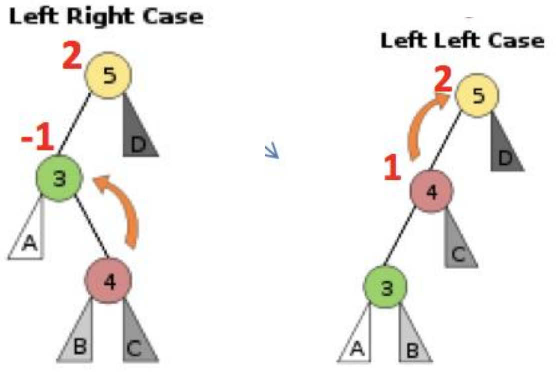
bf(x) = +2 and 0 <= bf(x.left) <= 1

rightRotate(x)

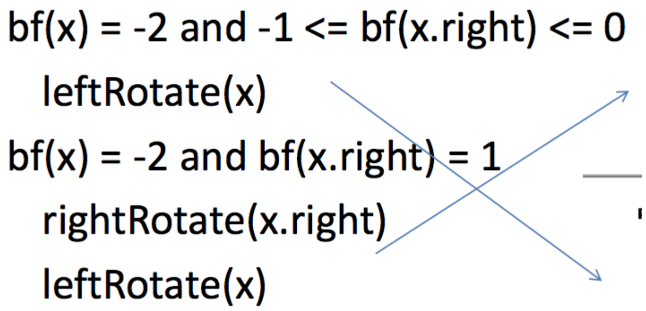
bf(x) = +2 and bf(x.left) = -1

leftRotate(x.left)

rightRotate(x)



LeftRotate is the most Left Pic
RightRotate is the most Right Pic



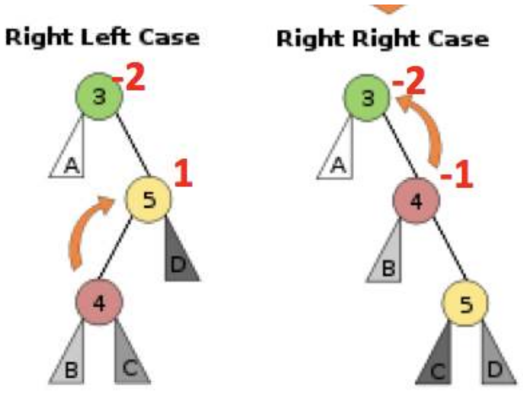
bf(x) = -2 and -1 <= bf(x.right) <= 0

leftRotate(x)

bf(x) = -2 and bf(x.right) = 1

rightRotate(x.right)

leftRotate(x)



RightRotate is the most left Pic
LeftRotate is the most right Pic

Graphs Terminologies :

Sparse = not so many edges
Dense = many edges (No guideline for how many)

Complete Graph
- Simple graph with N vertices and $NC2$ edges → $\frac{N(N-1)}{2}$

In / out degree of a vertex
- Number of in/out edges from a vertex

(Simple) Path
- Sequence of vertices connected by a sequence of edges
- Simple = no repeated vertex

Path Length/Cost
- In unweighted graph, usually number of edges in the path
- In weighted graph, usually sum of edge weight in the path

(Simple) Cycle
- Path that starts and ends with the same vertex
- With no repeated vertices except start/end vertex

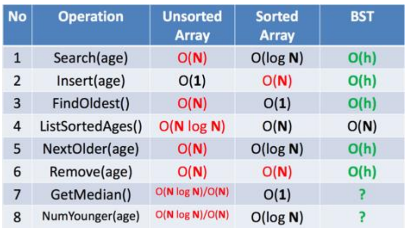
Component
- A group of vertices in undirected graph that can visit each other via some path

Connected graph
- Graph with only 1 component

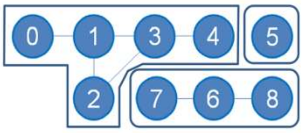
Reachable/Unreachable Vertex

Acyclic
- Has no cycle

Subgraph
- Subset of vertices (and their connecting edges) of the original graph



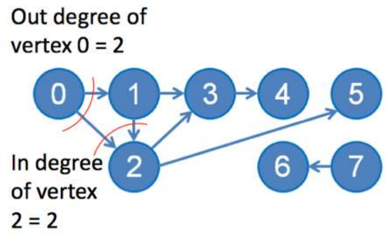
No	Operation	Unsorted Array	Sorted Array	BST
1	Search(age)	O(N)	O(log N)	O(h)
2	Insert(age)	O(1)	O(N)	O(h)
3	FindOldest()	O(N)	O(1)	O(h)
4	ListSortedAges()	O(N log N)	O(N)	O(N)
5	NextOlder(age)	O(N)	O(log N)	O(h)
6	Remove(age)	O(N)	O(N)	O(h)
7	GetMedian()	O(N log N)/O(N)	O(1)	?
8	NumYounger(age)	O(N log N)/O(N)	O(log N)	?



- There are 3 components in this graph
- Disconnected graph
(since it has > 1 component)
- Vertices 1-2-3-4 are reachable from vertex 0
- Vertices 5, 6-7-8 are unreachable from vertex 0
- {7-6-8} is a sub graph of this graph

Directed Acyclic Graph (DAG)

- Directed graph that has no cycle

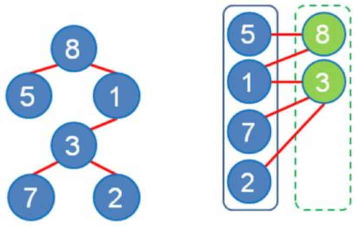


Tree (Left)

- Connected graph, E = V – 1
- One unique path between any pair of vertices

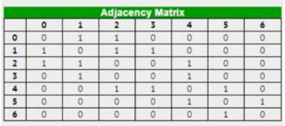
Bipartite Graph (Right)

- If we can partition the vertices into two sets so that there is no edge between members of the same set



Adjacency Matrix

- A 2D array Adjacency Matrix contains value 1 if there exists a edge (It is not sorted)
- Space Complexity : O(V²) → V is the number of vertices in Graph

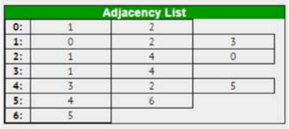


Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

- Format : array Adjacency List of V lists, one for each vertex
- Space Complexity : O(V + 2E) ➔ O(V + E)
- Take Note E = number of edges in Graph, worst case : E = O(V²)

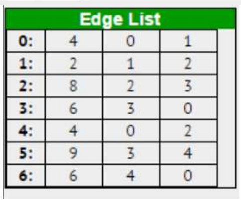


Adjacency List

0:	1	2	
1:	0	2	3
2:	1	4	0
3:	1	4	
4:	3	2	5
5:	4	6	
6:	5		

Edge List (Can be sorted to display based on the number of edges)

- Contains an (integer) triple {w(u,v), u , v}
- Space Complexity : O(E)
- Take Note , E = O(V²)



Edge List

0:	4	0	1
1:	2	1	2
2:	8	2	3
3:	6	3	0
4:	4	0	2
5:	9	3	4
6:	6	4	0

	Adjacency Matrix	Adjacency List	Edge List
Looping through neighbours	O(V)	O(deg(v))	O(E)
Check edge existence	O(1)	O(V)	O(E)
Count the number of edges	O(V^2)	O(V)	O(1)

Edges need to be sorted -> Edge list
Existence of edge is frequently asked -> Adjacency matrix
Neighbors frequently enumerated -> Adjacency matrix and adjacency list (Both take linear time), however if memory allocated < (No. of vertices)², only can use **adjacency list**.

UFDS

Given **n** disjoint sets initially in a UFDS, is it possible to call unionSet(i, j) and/or findSet(i) operations to get a single tree with actual height h that represents a certain set? Both path-compression and union-by-rank heuristics are used.

Calculate 2^h. If it is greater than n then not possible.

Graph Traversal (Breath First Search & Depth First Search)

Adjacency list is more compact to Adjacency Matrix
When the graph is very dense, adjacency list will be almost same size as adjacency matrix

Enumerating neighbours of a vertex u

- **O(V) for Adj Matrix,**

  scan Adj Matrix [v][i] ∀I ∈ [0 , V-1]
- **O(K) for Adj List**, scan Adj List[V]
  - K is the number of neighbours of vertex V (Output sensitive algorithm)

Very important difference between Adj Matrix vs Adj List

Counting Edge of a vertex u

- O(1) for Edge list (Take note that bidirectional edges may be listed once (or twice) in edge list, depending on the need.
- $O(V^2)$ for Adj Matrix
  (Have to count all the non - zero entries)
- O(V + E) for Adj List
  (Sum of length of all vertex lists)

Checking the existence of edge (u, v)

- O(1) for Adj Matrix (check if Adj Matrix [u][v] is non - zero)
- O(K) for Adj List (check if Adj List[u] contains v)

Breadth First Search (BFS)

- Start from source **s**
- If a vertex **v** is reachable from **s**, then all neighbours of v will also be reachable from **s.**
- BFS visits vertices of G In **breath-first** manner

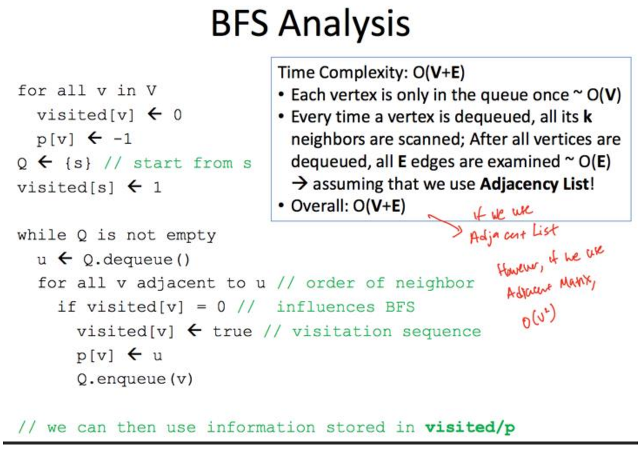
When viewed from source vertex s,

- Use queue **Q,** initially to contain only **s**
- 1D array/Vector **visited** of size V
  - visited[v] = 0 initially and visited[v] = 1 when v is visited
- 1D array/Vector **p** of size V,
  - **p[v]** denotes the predecessor of **v**

Trade-Off	
Adjacency Matrix	Adjacency List
Pros	
Existence of edge i-j can be found in O(1) Good for dense graph	O(K) to enumerate k neighbours of a vertex Good for sparse graph/ Dijkstra's/ DFS/ BFS O(V+E) space
Cons	
O(V) to enumerate neighbours of a vertex $O(V^2)$ space	O(K) to check the existence of edge i-j A small overhead in maintain the list (sparse graph)

Sometime the number stored in separate variable so that we are not required to re-compute every iteration e.g. O(1). For e.g. if the graph do not change after it is been created.

```
Backtrack(u) {
    If (u == -1)
        Stop
    Backtrack(p[u])
    Output u
```



BFS Analysis

```
for all v in V
  visited[v] ← 0
  p[v] ← -1
Q ← {s} // start from s
visited[s] ← 1

while Q is not empty
  u ← Q.dequeue()
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences BFS
      visited[v] ← true // visitation sequence
      p[v] ← u
      Q.enqueue(v)

// we can then use information stored in visited/p
```

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

- Each vertex is only in the queue once ~ O(**V**)
- Every time a vertex is dequeued, all its **k** neighbors are scanned; After all vertices are dequeued, all **E** edges are examined ~ O(**E**)
  → assuming that we use **Adjacency List**!
- Overall: O(**V+E**)

if we use Adjacent List

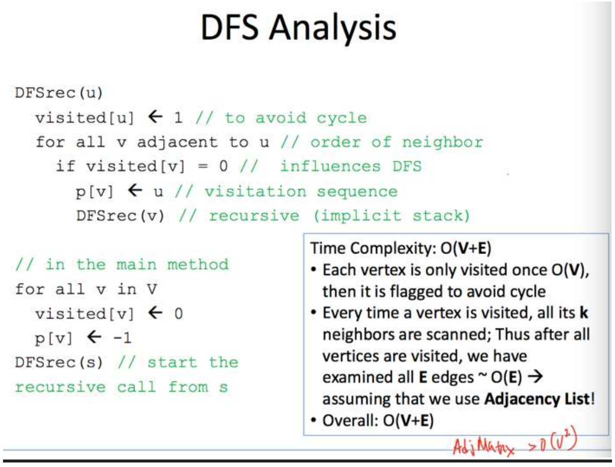
However, if we use Adjacent Matrix, $O(V^2)$

Depth First Search (DFS)

- Start from source **s**
- If a vertex **v** is reachable from **s**, then all neighbours of v will also be reachable from **s.**
- DFS visits vertices of G In **depth-first** manner

When viewed from source vertex s,

- Use stack **S**
- 1D array/Vector **visited** of size V
    - visited[v] = 0 initially and visited[v] = 1 when v is visited
- 1D array/Vector **p** of size V,
  p[v] denotes the predecessor of **v**



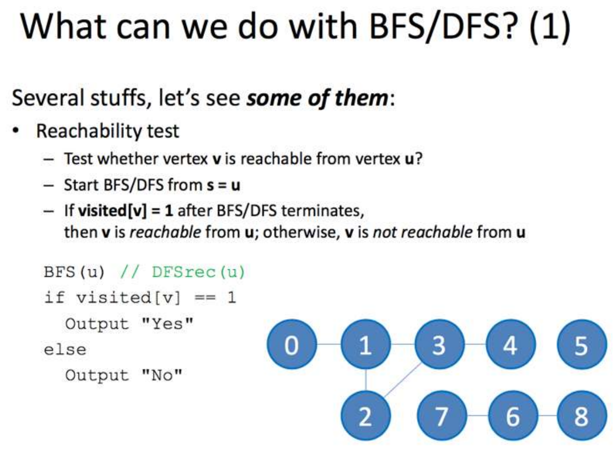
DFS Analysis

```
DFSrec(u)
  visited[u] ← 1 // to avoid cycle
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences DFS
      p[v] ← u // visitation sequence
      DFSrec(v) // recursive (implicit stack)

// in the main method
for all v in V
  visited[v] ← 0
  p[v] ← -1
DFSrec(s) // start the
recursive call from s
```

Time Complexity: O(**V+E**)
- Each vertex is only visited once O(**V**), then it is flagged to avoid cycle
- Every time a vertex is visited, all its **k** neighbors are scanned; Thus after all vertices are visited, we have examined all **E** edges ~ O(**E**) → assuming that we use **Adjacency List**!
- Overall: O(**V+E**)

AdjMatrix → O(V^2)

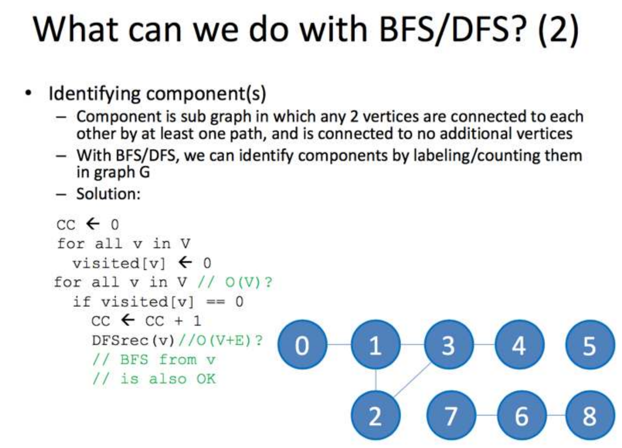


What can we do with BFS/DFS? (1)

Several stuffs, let's see ***some of them***:

- Reachability test
  - Test whether vertex **v** is reachable from vertex **u**?
  - Start BFS/DFS from **s = u**
  - If **visited[v] = 1** after BFS/DFS terminates, then **v** is *reachable* from **u**; otherwise, **v** is *not reachable* from **u**

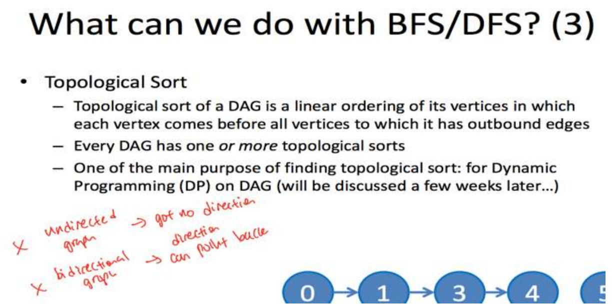
```
BFS(u) // DFSrec(u)
if visited[v] == 1
  Output "Yes"
else
  Output "No"
```



What can we do with BFS/DFS? (2)

- Identifying component(s)
  - Component is sub graph in which any 2 vertices are connected to each other by at least one path, and is connected to no additional vertices
  - With BFS/DFS, we can identify components by labeling/counting them in graph G
  - Solution:

```
CC ← 0
for all v in V
  visited[v] ← 0
for all v in V // O(V)?
  if visited[v] == 0
    CC ← CC + 1
    DFSrec(v)//O(V+E)?
    // BFS from v
    // is also OK
```

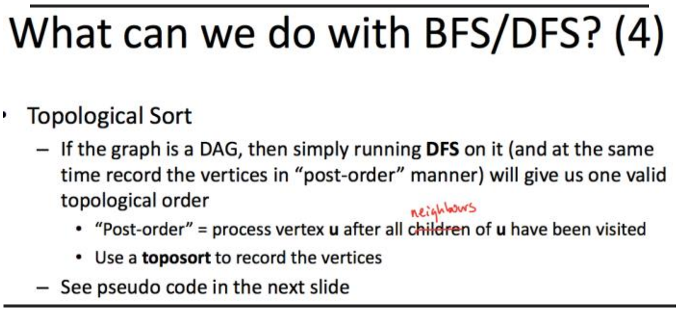


What can we do with BFS/DFS? (3)

- Topological Sort
  - Topological sort of a DAG is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound edges
  - Every DAG has one *or more* topological sorts
  - One of the main purpose of finding topological sort: for Dynamic Programming (DP) on DAG (will be discussed a few weeks later…)

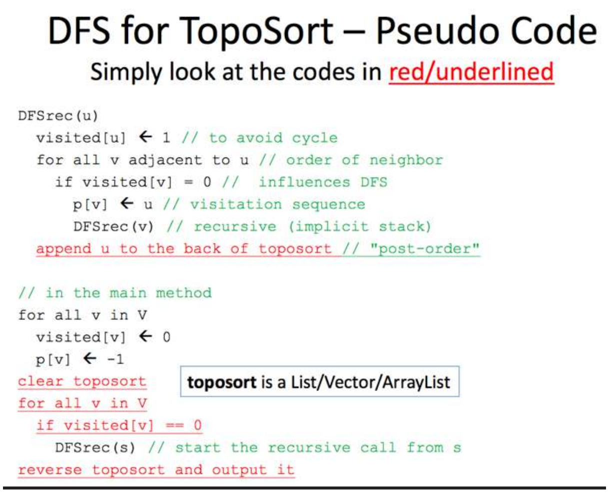
✗ undirected graph → got no direction

✗ bidirectional graph → direction can point back



What can we do with BFS/DFS? (4)

- Topological Sort
  - If the graph is a DAG, then simply running **DFS** on it (and at the same time record the vertices in "post-order" manner) will give us one valid topological order
    - "Post-order" = process vertex **u** after all ~~children~~ neighbours of **u** have been visited
    - Use a **toposort** to record the vertices
  - See pseudo code in the next slide



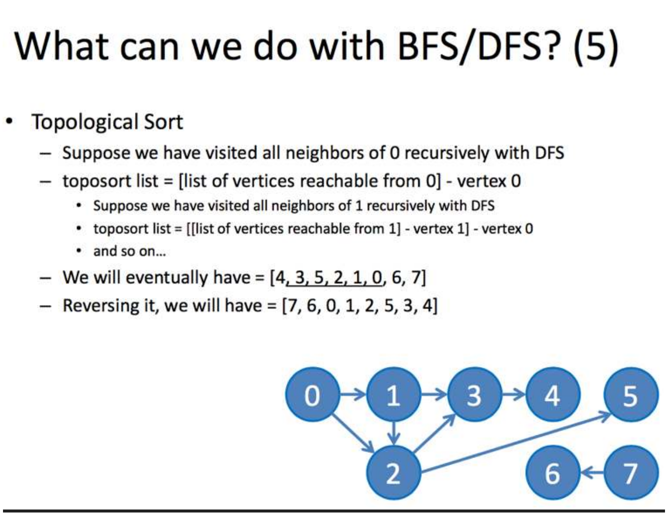
DFS for TopoSort – Pseudo Code

Simply look at the codes in red/underlined

```
DFSrec(u)
  visited[u] ← 1 // to avoid cycle
  for all v adjacent to u // order of neighbor
    if visited[v] = 0 //  influences DFS
      p[v] ← u // visitation sequence
      DFSrec(v) // recursive (implicit stack)
  append u to the back of toposort // "post-order"

// in the main method
for all v in V
  visited[v] ← 0
  p[v] ← -1
clear toposort
for all v in V
  if visited[v] == 0
    DFSrec(s) // start the recursive call from s
reverse toposort and output it
```

toposort is a List/Vector/ArrayList



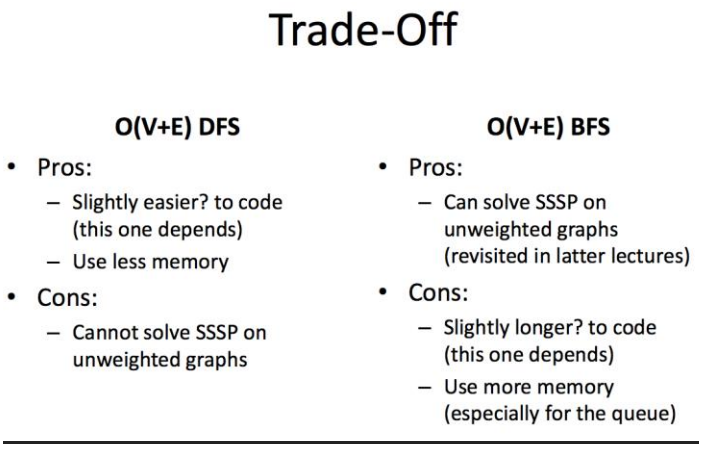
What can we do with BFS/DFS? (5)

- Topological Sort
  - Suppose we have visited all neighbors of 0 recursively with DFS
  - toposort list = [list of vertices reachable from 0] - vertex 0
    - Suppose we have visited all neighbors of 1 recursively with DFS
    - toposort list = [[list of vertices reachable from 1] - vertex 1] - vertex 0
    - and so on…
  - We will eventually have = [4, 3, 5, 2, 1, 0, 6, 7]
  - Reversing it, we will have = [7, 6, 0, 1, 2, 5, 3, 4]

Topological Sort

- Toplogocial sort of a DAG is a linear ordering of its vertices in which each vertex comes before all vertices to which it has outbound edges
- Every DAG has **one or more** topological sorts
- One of the main purpose of finding topologic sort : for Dynamic Programming (DP) on DAG
- Topological sort **cannot sort Undirected** (No direction) & **Bidirectional** (Direction that point back) graph
- If Graph is a DAG, simply running **DFS** give us one valid topological order.
    - "Post-order" = process vertex u after all the neighbours of u been visited

  Use a toposort to record the vertices



Trade-Off

O(V+E) DFS

- Pros:
    - Slightly easier? to code (this one depends)
    - Use less memory
- Cons:
    - Cannot solve SSSP on unweighted graphs

O(V+E) BFS

- Pros:
    - Can solve SSSP on unweighted graphs (revisited in latter lectures)
- Cons:
    - Slightly longer? to code (this one depends)
    - Use more memory (especially for the queue)

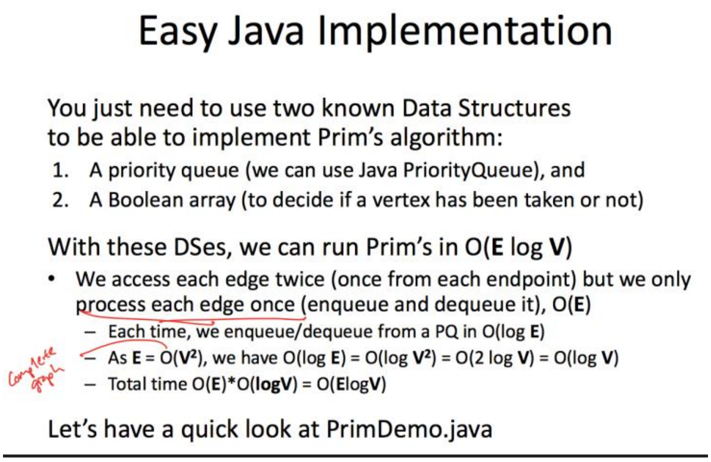
Minimum Spanning Tree

Tree T

- **T** is a connected graph that has **V vertices** and **V-1 edges**
- **Important** : One unique path between any two pair of vertices

Spanning Tree ST

- **ST** is a tree that spans (covers) every vertex



Easy Java Implementation

You just need to use two known Data Structures to be able to implement Prim's algorithm:

1. A priority queue (we can use Java PriorityQueue), and
2. A Boolean array (to decide if a vertex has been taken or not)

With these DSes, we can run Prim's in O(**E** log **V**)

- We access each edge twice (once from each endpoint) but we only process each edge once (enqueue and dequeue it), O(**E**)
    - Each time, we enqueue/dequeue from a PQ in O(log **E**)
    - As **E** = $O(V^2)$, we have O(log **E**) = O(log V^2) = O(2 log **V**) = O(log **V**)
    - Total time O(**E**)*O(**logV**) = O(**ElogV**)

Complete graph

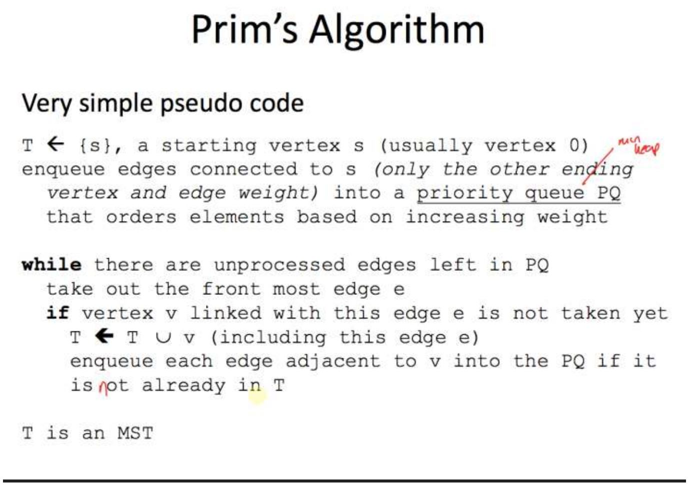
Let's have a quick look at PrimDemo.java

The (standard) MST Problem

- **Input :** A connected undirected weighted graph
- **Select some edges of G** such that the graphs form a spanning tree, with minimum total weight
- **Output:** Minimum Spanning Tree (MST) of G

Several efficient algorithms

- **Prim**
    - **Uses priority queue**
- **Krukal**
    - **Uses Union-Find Data Structure**
- **Boruvka**



Prim's Algorithm

Very simple pseudo code

```
T ← {s}, a starting vertex s (usually vertex 0)      min heap
enqueue edges connected to s (only the other ending
  vertex and edge weight) into a priority queue PQ
  that orders elements based on increasing weight

while there are unprocessed edges left in PQ
  take out the front most edge e
  if vertex v linked with this edge e is not taken yet
    T ← T ∪ v (including this edge e)
    enqueue each edge adjacent to v into the PQ if it
    is not already in T

T is an MST
```

Why Prim's Works? (2)

with visual explanation

Proof by contradiction:

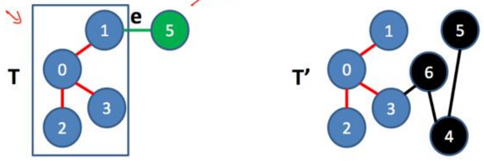
Assume that edge **e** is the first edge at iteration k chosen by Prim's which is not in any valid MST.

Let **T** be the tree generated by Prim's before adding **e**.

Now **T** must be a subtree of some valid MST **T'**

Everything here is correct

this is wrong

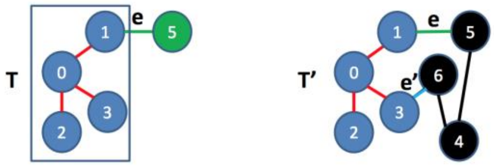


Why Prim's Works? (3)

with visual explanation

Adding edge **e** to **T'** will now create a cycle.

Since e has 1 endpoint in **T** (the valid endpoint) and one endpoint outside **T**, trace around this cycle in **T'** until we get to some edge **e'** that goes back to **T**



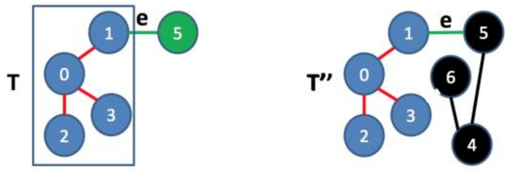
Why Prim's Works? (4)

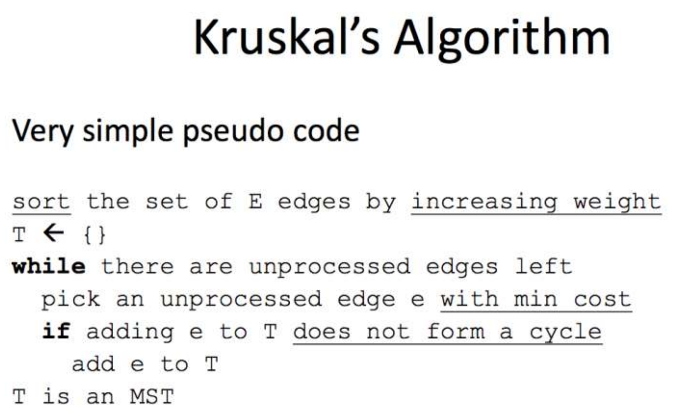
with visual explanation

By Prim's algorithm **e** and **e'** must be candidate edges at iteration k, but **e** was chosen meaning w(**e**) ≤ w(**e'**)

Now replacing **e'** with **e** in **T'** must give us tree **T''** covering all vertices of the graph s.t w(**T''**) ≤ w(**T'**)

Contradiction that **e** is first edge chosen wrongly

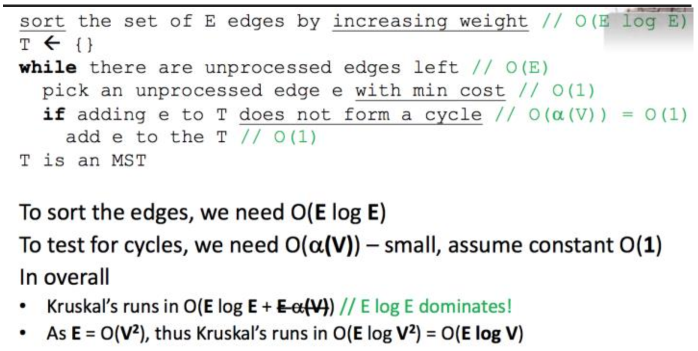




Kruskal's Algorithm

Very simple pseudo code

```
sort the set of E edges by increasing weight
T ← {}
while there are unprocessed edges left
  pick an unprocessed edge e with min cost
  if adding e to T does not form a cycle
    add e to T
T is an MST
```



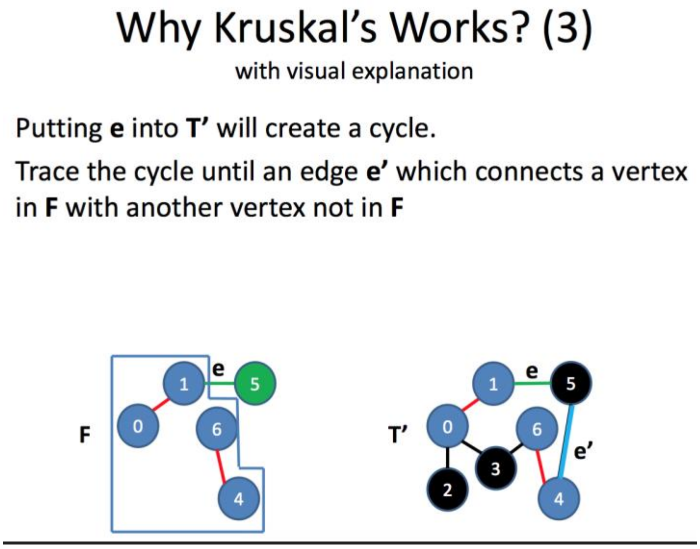
```
sort the set of E edges by increasing weight // O(E log E)
T ← {}
while there are unprocessed edges left // O(E)
  pick an unprocessed edge e with min cost // O(1)
  if adding e to T does not form a cycle // O(α(V)) = O(1)
    add e to the T // O(1)
T is an MST
```

To sort the edges, we need O(**E** log **E**)

To test for cycles, we need O(α(**V**)) – small, assume constant O(**1**)

In overall

- Kruskal's runs in O(**E** log **E** + ~~**E** α(**V**)~~) // E log E dominates!
- As **E** = O(**V**^{2}), thus Kruskal's runs in O(**E** log **V**^{2}) = O(**E log V**)



Why Kruskal's Works? (3)

with visual explanation

Putting **e** into **T'** will create a cycle.

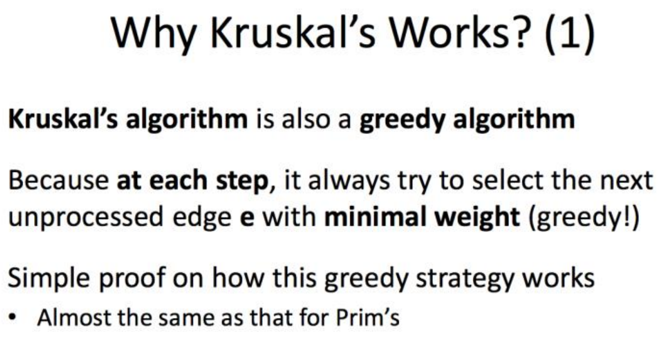
Trace the cycle until an edge **e'** which connects a vertex in **F** with another vertex not in **F**

To sort the edges:

- We use EdgeList to store graph information
- Then use "any" sorting algorithm (Collection.sort)

To test for cycles:

- We use **UFDS**



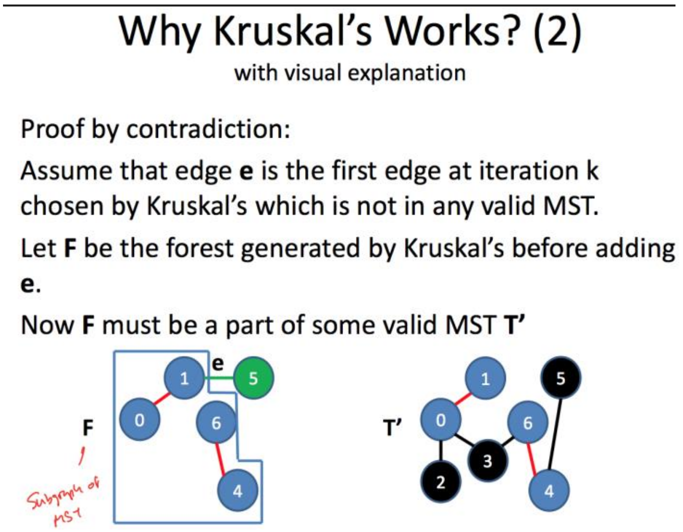
Why Kruskal's Works? (1)

Kruskal's algorithm is also a **greedy algorithm**

Because **at each step**, it always try to select the next unprocessed edge **e** with **minimal weight** (greedy!)

Simple proof on how this greedy strategy works

- Almost the same as that for Prim's



Why Kruskal's Works? (2)

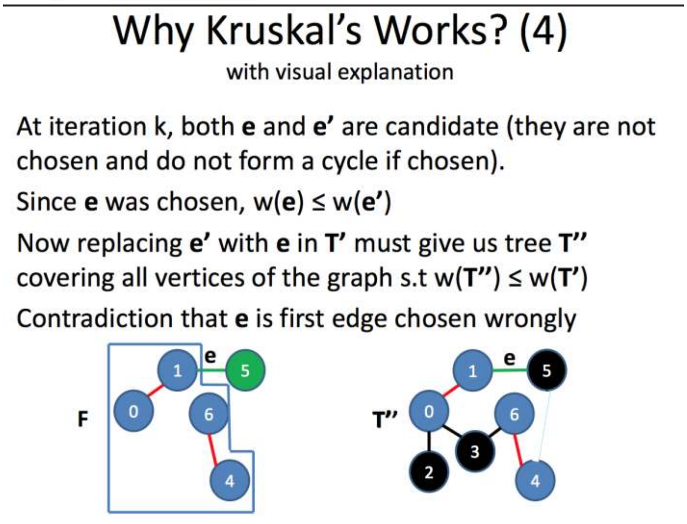
with visual explanation

Proof by contradiction:

Assume that edge **e** is the first edge at iteration k chosen by Kruskal's which is not in any valid MST.

Let **F** be the forest generated by Kruskal's before adding **e**.

Now **F** must be a part of some valid MST **T'**



Why Kruskal's Works? (4)

with visual explanation

At iteration k, both **e** and **e'** are candidate (they are not chosen and do not form a cycle if chosen).

Since **e** was chosen, w(**e**) ≤ w(**e'**)

Now replacing **e'** with **e** in **T'** must give us tree **T''** covering all vertices of the graph s.t w(**T''**) ≤ w(**T'**)

Contradiction that **e** is first edge chosen wrongly

If the following undirect graph has only one unique minimum spanning tree?

Draw the Minimum Spanning Tree to check if there is a vertex there **has two similar edge weight** and it is also the **smallest edge weight** of the vertext

Select the edge that does not belong to any **minimum/maximum** spanning tree

Use Krustal to select based on edge weight and find the Minimum/Maximum Spanning Tree and select all the unused edges

This is a risky but fast method
Or left last N-1 Edges. For e.g. 8 vertices, select the edges that do not belong to any **Maxmimum** Spanning Tree. Use kruskal to select the **MIN** edge until 7 (V-1) edges is left. **Becareful of bridges, you should not select them.**

Draw a simple connected weighted undirected graph with 10 edges and 8 vertices so that the **optimized krukal alogirthm** examine all 10 edges before stopping.

Optimized krukal algorithm stopped when all vertices had been found.

1) Take out a vertex and form a cycle with the rest of the vertex.
2) Connect the last vertex with the last biggest weight edge.

Draw a simple connected weight graph with 3 vertices and 4 directed edges such that Modified Dijkstra algorithm will run indefinitely.

Draw a negative cycle

Minimum maintenace cost

Select the required edge shown by the question then, Use Krustal by selecting the min edge to connect the rest of the vertices

Second best minimum spanning tree
Use Krustal until the last or second last vertex choose the 2nd smallest edge weight

Select the edges (in any order) that form 2 **connected components** and **total weight of the componenets is minimum.**

Depending on the question required how many connected componets, use kruskal to and find the required number of components.

Click the edge that has the **MAXimum/Minimum** edge weight along **MiNiMAX/Maximin** from **vertex 2 (source) to vertex 4 (destination)**

If **MAXimum** edge weight, find the **minimum** spanning tree and trace from **source to destination.** And **select the maxmium edge**

If **Minimum** edge weight, find the **maxmimum** spanning tree and trace from **source to destination.** And **select the minimum edge**

One-Pass BellmanFord Algorithm

Click the topological order

Single Source Shortest Paths (SSSP)

BFS Algorithm
Bellman Ford's algorithm

Vertex set **V** (e.g. Stree Intersections , houses)
Edge set **E** (e.g. streets, road, avenues)

- Directed (one way road)
- Weighted (distance, time toll)

(Simple) Path = a path with no repeated vertex!

Normal Cycle

Dij Algo **Terminate** with **Correct output**
Bellman Ford **Terminate** with **Correct output**

Negative Cycle

Bellman Ford **Terminate** with **Incorrect output**
Dij Algo **Terminate** with **Incorrect output**
Mod Dij Algo will **Not Terminate**

Negative Weight

Bellman Ford **Terminate** with **Correct output**
Dij Algo **Terminate** with **Incorrect output**
Mod Dij Algo **Terminate** with **Correct output**

Optimized Bellmond ford

If 8 edges → 0 – 7 – 6 – 5 – 4 – 3 – 2 – 1

Modified Dij Algo with limited weight

K vertices = K edge → 1 triangle

11, 10 → one Triangle

11, 12 → Two Triangle

Positive edge decreasing , negative edge increasing

Modified Dij Algo (Infinitely)

Negative cycle ***Remember distinct weight***

Subset from source vertex 0 , greater than weight 44

Exclude source vertex "0"

Summary

Complete Graph
Number of vertex = N
Number of edges = N (N-1) /2

Bipartite Graph
Maximum edges = N^2 / 4

Directed Acyclic Graph
Maximum edges = (N-1) + (N-2) + … + 1 + 0 = (N-1) * (N) / 2

Minimum Spanning Tree / Tree
Number of edges = N - 1

Handshake Theorem
Every undirected graph has an even number of vertices of odd degree.

UFDS

Heuristic → Helps to make the resulting combined tree shorter

	UFDS	UFDS with one Heuristic	UFDS with two Heuristic	Modified UFDS with two Heuristics
FindSet	O(N)	O(lg N)	O(1)	O(1)
isSameSet	O(N)	O(lg N)	O(1)	O(1)
UnionSet	O(N)	O(lg N)	O(1)	O(1)
GetSize	O(N)	O(N)	O(N)	O(1)

- Check if two items belong to the same set
- Find which set an item belongs to
- Each set is modeled as a tree
- If same rank, UnionSet(x,y) → x will go under y. Else, follow the ranking, shorter will go under the taller tree.

Applications of Depth-First Search (DFS) Algorithm :
1). To test if vertex v is reachable from vertex u,
2). Find/Label/Count components of an undirected graph,
3). Find topological sort of a Directed Acyclic Graph,
4). Check if an undirected graph is a Bipartite Graph.
5) Flood Fill,
6) Check if a graph is cyclic or acyclic,
7) Find Articulation Points and Bridges,
8) Find Strongly Connected Component in a Directed Graph.

Applications of Breath-First Search (BFS) Algorithm :

1) For traversing the graph,

2). For checking if two vertices a and b are reachable: BFS(a), check if dist[b] is no longer INF,

3). For checking if the graph is connected,

4). For solving the SSSP problem on an unweighted graph,

5). For checking if the graph is bipartite.

6) For checking if the graph is a tree, for solving the SSSP problem on a tree,

Applications of Kruskal's Algorithm :

1). To find min (or max) ST weight (or the tree) of a connected weighted undirected graph,

2). To find the minimax (or maximin) path,

3). To find the Minimum Spanning Forest of k trees by stopping after we have k components

4) Second Best Spanning Tree

Applications of Prism's Algorithm :

1). To find min (or max) ST weight (or the tree) of a connected weighted undirected graph

2)

3)

Applications of Bellman Ford's Algorithm :

1). Find the shortest path

2) Detect Negative cycles

3)

Applications of Floyd Warshall Algorithm :

1). Print the actual shortest path using predecessor matrix

2) Solving transitive closure problem (determine if vertex I is connected to vertex j directly (via edge) or indirectly (via path))

3) Solving Minimax/Maximin

4) Detecting +ve/-ve cycle

Applications of Modified Dijkstra Algorithm :

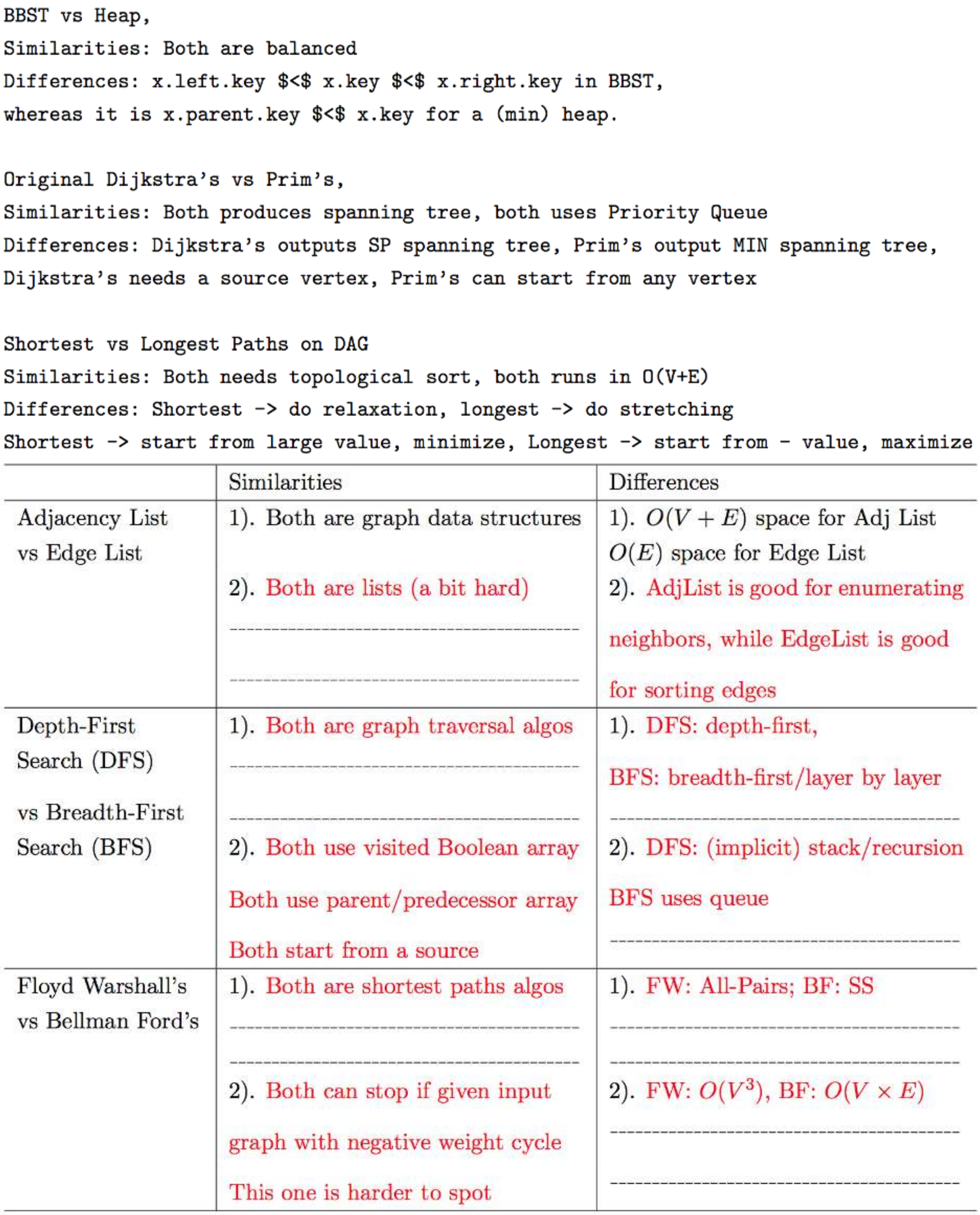
1). Able to find SSSP in negative weight graph

2) Unable to use on negative cycle

Dijkstra Algorithm vs Modified Dijkstra Algorithm

1). Unable to use on negative cycles **AND** cycle

2) O((V+E) log V) on no negative weight and cycle graph



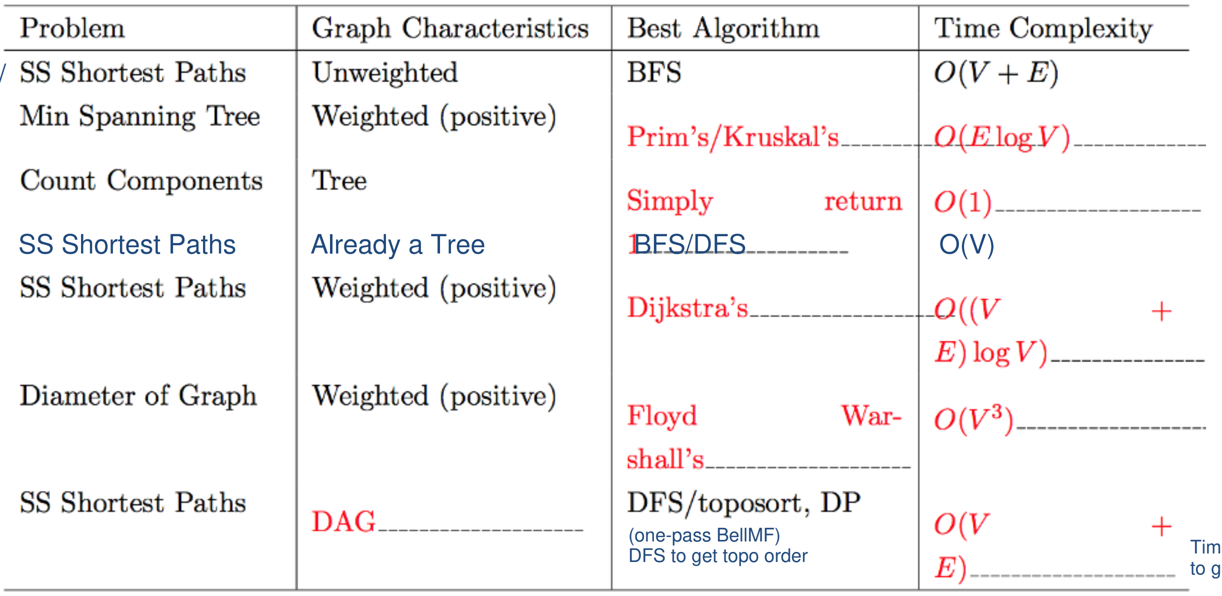
BBST vs Heap,
Similarities: Both are balanced
Differences: x.left.key $<$ x.key $<$ x.right.key in BBST,
whereas it is x.parent.key $<$ x.key for a (min) heap.

Original Dijkstra's vs Prim's,
Similarities: Both produces spanning tree, both uses Priority Queue
Differences: Dijkstra's outputs SP spanning tree, Prim's output MIN spanning tree,
Dijkstra's needs a source vertex, Prim's can start from any vertex

Shortest vs Longest Paths on DAG
Similarities: Both needs topological sort, both runs in O(V+E)
Differences: Shortest -> do relaxation, longest -> do stretching
Shortest -> start from large value, minimize, Longest -> start from - value, maximize

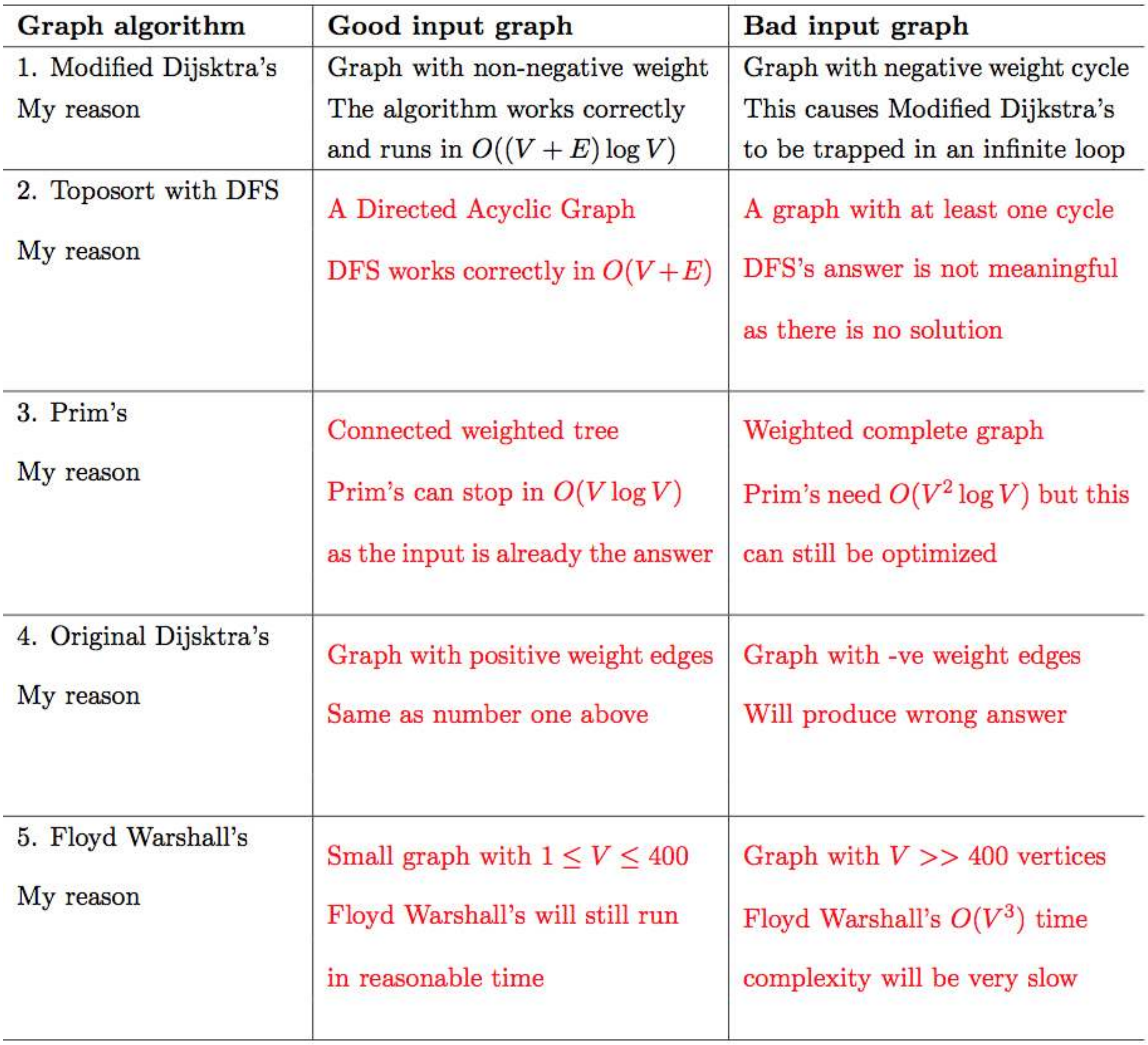
	Similarities	Differences
Adjacency List vs Edge List	1). Both are graph data structures 2). Both are lists (a bit hard)	1). $O(V+E)$ space for Adj List $O(E)$ space for Edge List 2). AdjList is good for enumerating neighbors, while EdgeList is good for sorting edges
Depth-First Search (DFS) vs Breadth-First Search (BFS)	1). Both are graph traversal algos 2). Both use visited Boolean array Both use parent/predecessor array Both start from a source	1). DFS: depth-first, BFS: breadth-first/layer by layer 2). DFS: (implicit) stack/recursion BFS uses queue
Floyd Warshall's vs Bellman Ford's	1). Both are shortest paths algos 2). Both can stop if given input graph with negative weight cycle This one is harder to spot	1). FW: All-Pairs; BF: SS 2). FW: $O(V^3)$, BF: $O(V \times E)$

SS Shortest Paths General weighted graph BellmanFord O(VE) <<< *slow, use disktra better



Problem	Graph Characteristics	Best Algorithm	Time Complexity
MST/ SS Shortest Paths	Unweighted	BFS	$O(V+E)$
Min Spanning Tree	Weighted (positive)	Prim's/Kruskal's	$O(E \log V)$
Count Components	Tree	Simply return 1	$O(1)$
SS Shortest Paths	Already a Tree	BFS/DFS	O(V)
SS Shortest Paths	Weighted (positive)	Dijkstra's	$O((V+E) \log V)$
Diameter of Graph	Weighted (positive)	Floyd Warshall's	$O(V^3)$
SS Shortest Paths	DAG	DFS/toposort, DP (one-pass BellMF) DFS to get topo order	$O(V+E)$ Time from DFS to get topological order

SS Shortest Paths | no Neg weight cycle(may have -ve weight) | Modified Dikstra | O((V+E) log V)



Graph algorithm	Good input graph	Bad input graph
1. Modified Dijsktra's My reason	Graph with non-negative weight The algorithm works correctly and runs in $O((V+E) \log V)$	Graph with negative weight cycle This causes Modified Dijkstra's to be trapped in an infinite loop
2. Toposort with DFS My reason	A Directed Acyclic Graph DFS works correctly in $O(V+E)$	A graph with at least one cycle DFS's answer is not meaningful as there is no solution
3. Prim's My reason	Connected weighted tree Prim's can stop in $O(V \log V)$ as the input is already the answer	Weighted complete graph Prim's need $O(V^2 \log V)$ but this can still be optimized
4. Original Dijsktra's My reason	Graph with positive weight edges Same as number one above	Graph with -ve weight edges Will produce wrong answer
5. Floyd Warshall's My reason	Small graph with $1 \leq V \leq 400$ Floyd Warshall's will still run in reasonable time	Graph with $V >> 400$ vertices Floyd Warshall's $O(V^3)$ time complexity will be very slow

Different BST with N Distinct Elements (2N Choose N) / (N + 1)		
N	**Value**	**Value**
0	1	1
1	1	1
2	2	2
3	5	5
4	14	14
5	42	42
6	132	132
7	429	429
8	1,430	1,430
9	4,862	4,862
10	16,796	16,796
11	58,786	58,786
12	208,012	208,012
13	742,900	742,900
14	2,674,440	2,674,440
15	9,694,845	9,694,845
16	35,357,670	35,357,670
17	129,644,790	129,644,790
18	477,638,700	477,638,700
19	1,767,263,190	1,767,263,190
20	-2,025,814,172	6,564,120,420
21	-1,303,536,756	24,466,267,020
22	1,288,250,424	91,482,563,640
23	-537,770,030	343,059,613,650
24	1,413,958,524	1,289,904,147,324
25	43,422,380	4,861,946,401,452
26	2,072,914,456	18,367,353,072,152
27	-1,969,606,236	69,533,550,916,004
28	-1,694,929,704	263,747,951,750,360
29	-1,286,772,120	1,002,242,216,651,360
30	-1,018,710,512	3,814,986,502,092,300
31	-125,737,443	14,544,636,039,226,900
32	300,814,726	55,534,064,877,048,100
33	-365,696,858	212,336,130,412,243,000
34	-1,768,236,596	812,944,042,149,730,000
35	-1,767,445,106	3,116,285,494,907,300,000
36	645,964,916	11,959,798,385,860,400,000
37	1,351,610,652	45,950,804,324,621,700,000
38	573,153,112	176,733,862,787,006,000,000
39	1,347,646,022	680,425,371,729,975,000,000
40	1,945,953,300	2,622,127,042,276,490,000,000
41	-470,547,964	10,113,918,591,637,900,000,000
42	480,774,088	39,044,429,911,904,400,000,000
43	-1,461,302,116	150,853,479,205,085,000,000,000
44	-1,928,063,192	583,300,119,592,996,000,000,000
45	-1,485,159,592	2,257,117,854,077,240,000,000,000
46	-999,164,816	8,740,328,711,533,170,000,000,000
47	-650,538,190	33,868,773,757,191,000,000,000,000
48	720,643,548	131,327,898,242,169,000,000,000,000
49	906,311,356	509,552,245,179,617,000,000,000,000
50	992,169,208	1,978,261,657,756,160,000,000,000,000
51	1,211,138,972	7,684,785,670,514,310,000,000,000,000
52	1,465,961,064	29,869,166,945,772,600,000,000,000,000
53	-25,663,368	116,157,871,455,782,000,000,000,000,000
54	-1,115,027,920	451,959,718,027,953,000,000,000,000,000
55	-199,068,796	1,759,414,616,608,810,000,000,000,000,000
56	580,984,888	6,852,456,927,844,870,000,000,000,000,000
57	-698,208,744	26,700,952,856,774,800,000,000,000,000,000
58	1,063,564,208	104,088,460,289,122,000,000,000,000,000,000
59	-1,006,060,344	405,944,995,127,577,000,000,000,000,000,000
60	-545,638,224	1,583,850,964,596,120,000,000,000,000,000,000
61	-1,159,917,872	6,182,127,958,584,850,000,000,000,000,000,000
62	-1,052,324,832	24,139,737,743,045,600,000,000,000,000,000,000
63	1,929,153,885	94,295,850,558,772,000,000,000,000,000,000,000
64	1,922,044,102	368,479,169,875,816,000,000,000,000,000,000,000
65	-1,076,489,466	1,440,418,573,150,920,000,000,000,000,000,000,000
66	2,072,635,148	5,632,681,584,560,310,000,000,000,000,000,000,000
67	-987,563,842	22,033,725,021,956,500,000,000,000,000,000,000,000
68	-1,250,052,332	86,218,923,998,960,300,000,000,000,000,000,000,000
69	-107,241,284	337,485,502,510,216,000,000,000,000,000,000,000,000
70	668,962,456	1,321,422,108,420,280,000,000,000,000,000,000,000,000
71	-243,208,578	5,175,569,924,646,100,000,000,000,000,000,000,000,000

72	-1,776,536,924	20,276,890,389,709,400,000,000,000,000,000,000,000,000
73	1,395,670,036	79,463,489,365,077,400,000,000,000,000,000,000,000,000
74	-1,916,317,208	311,496,878,311,103,000,000,000,000,000,000,000,000,000
75	-1,523,631,508	1,221,395,654,430,370,000,000,000,000,000,000,000,000,000
76	1,442,778,376	4,790,408,930,363,300,000,000,000,000,000,000,000,000,000
77	1,365,163,256	18,793,142,726,809,900,000,000,000,000,000,000,000,000,000
78	1,170,735,792	73,745,243,611,532,500,000,000,000,000,000,000,000,000,000
79	-666,196,954	289,450,081,175,265,000,000,000,000,000,000,000,000,000,000
80	406,944,500	1,136,359,577,947,330,000,000,000,000,000,000,000,000,000,000
81	-1,439,902,124	4,462,290,049,988,320,000,000,000,000,000,000,000,000,000,000
82	191,845,928	17,526,585,015,616,700,000,000,000,000,000,000,000,000,000,000
83	-780,236,460	68,854,441,132,780,200,000,000,000,000,000,000,000,000,000,000
84	1,987,032,376	270,557,451,039,395,000,000,000,000,000,000,000,000,000,000,000
85	-1,379,733,016	1,063,353,702,922,270,000,000,000,000,000,000,000,000,000,000,000
86	-1,425,015,408	4,180,080,073,556,520,000,000,000,000,000,000,000,000,000,000,000
87	351,484,988	16,435,314,834,665,400,000,000,000,000,000,000,000,000,000,000,000
88	-1,464,981,176	64,633,260,585,762,900,000,000,000,000,000,000,000,000,000,000,000
89	-321,967,384	254,224,158,304,000,000,000,000,000,000,000,000,000,000,000,000,000
90	432,467,024	1,000,134,600,800,350,000,000,000,000,000,000,000,000,000,000,000,000
91	-1,472,877,320	3,935,312,233,584,000,000,000,000,000,000,000,000,000,000,000,000,000
92	-670,233,648	15,487,357,822,491,800,000,000,000,000,000,000,000,000,000,000,000,000
93	925,755,312	60,960,876,535,340,300,000,000,000,000,000,000,000,000,000,000,000,000
94	-1,690,248,992	239,993,345,518,077,000,000,000,000,000,000,000,000,000,000,000,000,000
95	-749,775,374	944,973,797,977,428,000,000,000,000,000,000,000,000,000,000,000,000,000
96	-30,374,756	3,721,443,204,405,950,000,000,000,000,000,000,000,000,000,000,000,000,000
97	-1,434,425,252	14,657,929,356,129,500,000,000,000,000,000,000,000,000,000,000,000,000,000
98	-1,095,497,800	57,743,358,069,601,400,000,000,000,000,000,000,000,000,000,000,000,000,000
99	2,126,189,612	227,508,830,794,229,000,000,000,000,000,000,000,000,000,000,000,000,000,000

Binary Heap Max Number of Comparison	
N	Value
0	0
1	0
2	1
3	2
4	4
5	6
6	7
7	8
8	11
9	
10	
11	
12	
13	
14	
15	
16	26
17	30
18	31
19	32
20	34
21	**36**
22	37
23	38
24	41
25	44
26	45
27	46
28	48
29	50
30	51
31	52
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	
49	
50	

Maximum number of Swap	
N	Value
0	
1	0
2	
3	1
4	3
5	
6	4
7	4
8	
9	
10	
11	
12	
13	
14	
15	
16	15
17	15
18	16
19	16
20	18
21	18
22	19
23	19
24	22
25	22
26	23
27	23
28	25
29	25
30	26
31	26
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	
49	
50	

Binary Heap Min Number of Comparison	
N	Value
0	0
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8
10	9
11	10
12	11
13	12
14	13
15	14
16	15
17	16
18	17
19	18
20	19
21	20
22	21
23	22
24	23
25	24
26	25
27	26
28	27
29	28
30	29
31	30
32	31
33	32
34	33
35	34
36	35
37	36
38	37
39	38
40	39
41	40
42	41
43	42
44	43
45	44
46	45
47	46
48	47
49	48
50	49

Min Height of AVL Tree F(N) = F(N - 1) + F (N - 2) + 1	
Height	Vertices
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54
8	88
9	143
10	232
11	376
12	609
13	986
14	1596
15	2583
16	4180
17	6764
18	10945
19	17710
20	28656
21	46367
22	75024
23	121392
24	196417
25	317810
26	514228
27	832039
28	1346268
29	2178308
30	3524577
31	5702886
32	9227464
33	14930351
34	24157816
35	39088168
36	63245985
37	102334154
38	165580140
39	267914295
40	433494436
41	701408732
42	1134903169
43	1836311902
44	
45	
46	
47	
48	
49	
50	

UFDS (If 2^Height Smaller or equal to N == Yes)	
Height	2^Height
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1,024
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288
20	1,048,576
21	2,097,152
22	4,194,304
23	8,388,608
24	16,777,216
25	33,554,432
26	67,108,864
27	134,217,728
28	268,435,456
29	536,870,912
30	1,073,741,824
31	2,147,483,648
32	4,294,967,296
33	8,589,934,592
34	17,179,869,184
35	34,359,738,368
36	68,719,476,736
37	137,438,953,472
38	274,877,906,944
39	549,755,813,888
40	1,099,511,627,776
41	2,199,023,255,552
42	4,398,046,511,104
43	8,796,093,022,208
44	17,592,186,044,416
45	35,184,372,088,832
46	70,368,744,177,664
47	140,737,488,355,328
48	281,474,976,710,656
49	562,949,953,421,312
50	1,125,899,906,842,620
51	2,251,799,813,685,250
52	4,503,599,627,370,500
53	9,007,199,254,740,990
54	18,014,398,509,482,000
55	36,028,797,018,964,000
56	72,057,594,037,927,900
57	144,115,188,075,856,000
58	288,230,376,151,712,000
59	576,460,752,303,423,000

Different spanning trees are there in a complete graph	
Vertex	Value
0	
1	
2	
3	3
4	16
5	125
6	1296
7	16807
8	262144
9	4782969
10	100000000
11	2357947691
12	61917364224
13	1.79216E+12
14	5.66939E+13
15	1.9462E+15
16	7.20576E+16
17	2.86242E+18
18	1.2144E+20
19	5.48039E+21
20	2.62144E+23
21	1.32485E+25
22	7.05429E+26
23	3.94716E+28
24	2.31551E+30
25	1.42109E+32
26	9.10669E+33
27	6.08267E+35
28	4.22775E+37
29	3.05313E+39
30	

Bellman Ford Working

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7	8	9