Definitions

▼ Sorting

- O(N^2) : Bubble, Selection, Insertion
- O(nlogn) : Mergesort, quicksort
- O(n) : Counting sort, radix sort
- in place sorting: constant amount (i.e., O(**1**)) of extra space during the sorting process.
- Stable sorting: relative order of elements **with the same key value** is preserved by the algorithm after sorting is performed.
  - stable: Mergesort, counting sort, Insertion sort, Bubble sort
  - unstable: Quicksort, Heapsort and Selection Sort are unstable.
    - Extra space can make quicksort stable

▼ LL, Stack, Queue, Deque

▼ SLL Operations

- `Get(i)` **O(N)**
- `Search(v)` **O(N)**

▼ Insertion (3 cases)

- Insert at Head (i = 0) : O(1)
- In Between, i ∈ [1..N-1]: O(**N**)
- Insert at tail (end) i = N: O(1)

▼ Removal (3 cases)

- Insert at Head (i = 0) : O(1)
- In Between, i ∈ [1..N-2]: O(**N**)
- Insert at tail (end) i = N-1: O(**N**)

- Stack
  - `push` : insert at head
  - `pop` : remove from head
- Queue
  - `enqueue` : insert at end
  - `dequeue` : remove from head
- Deque
  - `addFirst` : insert at front, O(1)
  - `addLast` : insert at end, O(1)
  - `pollFirst` : remove from head, O(1)
  - `pollLast` : remove from end, O(1)
  - `peekFirst` : O(1)
  - `peekLast` : O(1)

Binary Heap/ Priority Queue

Binary Heap

A Binary (Max) Heap is a complete binary tree that maintains the Max Heap property.

Using 1-based Index Array

1. parent(i) = `i>>1`, index i divided by 2 (integer division),
2. left(i) = `i<<1`, index i multiplied by 2,
3. right(i) = `(i<<1)+1`, index i multiplied by 2 and added by 1.

Properties

Complete Binary Tree

Every level in the binary tree, except possibly the last/lowest level, is completely filled, and all vertices in the last level are as far left as possible.

Binary Max Heap property

The parent of each vertex - except the root - contains value greater than (or equal to — we now allow duplicates) the value of that vertex

- ▼ Full Binary Tree (extra)
  - Every parent node has 2 children node
- ▼ Operations
  - ▼ `Create(A)`
    - O(**N** log **N**) version (**N** calls of `Insert(v)`)
    - ▼ O(**N**) version

      Fixes Binary Max Heap property (if necessary) only from the last internal vertex back to the root.
  - `Insert(v)` **:** O(log **N**)
  - ▼ 3 versions of **ExtractMax()**:
    - Once, in O(log **N**)
    - **K** times, i.e., **PartialSort()**, in O(**K** log **N**), or
    - **N** times, i.e., **HeapSort()**, in O(**N** log **N)**
  - `UpdateKey(i, newv)` : O(log **N) if i** is known
  - `Delete(i)` : O(log **N) if i** is known
- ▼ Heapsort / partial Sort
  - HeapSort: O(**N** log N)
  - PartialSort (for k elements): O(K log N)

▼ Priority Queue (PQ)

PQ ADT is similar to normal Queue ADT, but with these two major operations:

1. `Enqueue( x )`: Put a new element (key) **x** into the PQ (in some order)
   - Able to use `Insert(x)` in O(log N) time
2. `y = Dequeue()`: Return an existing element **y** that has the highest priority (key) in the PQ and if ties, return any
   - Use `y = extractMax()` in O(log N) time

▼ UFDS

▼ Operations

- `Initialize(N,M)` : O(N)

- `FindSet(i)` **:** O(1) assumption
- `IsSameSet(i, j)` : O(1)
- `UnionSet(i, j)` O(1)

- Max height of UDFS is $log_2 n$

▼ Actual time complexity O(α(**N**))

This α(**N**) is called the inverse Ackermann function that grows extremely slowly. For practical usage of this UFDS data structure (assuming **N ≤ 1M**), we have α(**1M**) ≈ 1.

▼ Max Size of set

If *n* is the initial number of disjoint sets, and *k* is the number of union operations performed, then the maximum possible size of any disjoint set is given by:

Max Size = n-(k-1)

▼ Hash Tables

▼ rehashing

α = (n/m) where n is the number of keys and m is the table size

A rule of thumb is to rehash when α ≥ 0.5 if using Open Addressing and when α > small constant (close to 1.0, as per requirement) if using Separate Chaining

▼ BST/AVL

BST Property: every vertex in the left subtree of a given vertex must carry a value smaller than that of the given vertex, and every vertex in the right subtree must carry a value larger

▼ BST operations

`Search(v)` / `lower_bound(v)` / `SearchMin()` / `SearchMax()` / `Successor(v)` / `Predecessor(v)` / `Insert(v)` / `Remove(v)`

O(**h**) where **h** is the height of the BST.

▼ AVL Operations

height = log(n)

so operations are now O(logn)

- Traversal (Inorder / Preorder/ Postorder)
  - Inorder Traversal: O(N)

    Usage: get values of nodes in non-decreasing order

    ```
    if this is null
      return
    Inorder(left)
    visit this
    Inorder(right)
    ```

    visit the left subtree first, exhausts all items in the left subtree, visit the current root, before exploring the right subtree and all items in the right subtree.

  - Preorder Traversal: O(N)

    Basically, in Preorder Traversal, we visit the current root before going to left subtree and then right subtree.

    ```
    if this is null
      return
    visit this
    Preorder(left)
    Preorder(right)
    ```

    Usage : Used to create a copy of a tree. For example, if you want to create a replica of a tree, put the nodes in an array with a pre-order traversal. Then perform an *Insert* operation on a new tree for each value in the array. You will end up with a copy of your original tree.

    For the example BST shown in the background, we have: {{15}, {6, 4, 5, 7}, {23, 71, 50}}.
    PS: Do you notice the recursive pattern? root, members of left subtree of root, members of right subtree of root.

  - Postorder Traversal: O(N)

    In Postorder Traversal, we visit the left subtree and right subtree first, before visiting the current root.

```
if this is null
  return
Postorder(left)
Postorder(right)
visit this
```

usage: to delete entire tree cause it visits leaf nodes first.

- Discussion: Given a Preorder Traversal / postorder traversal of a BST can you use it to recover the original BST?
  - Preorder Traversal

    O(n) time for getting inorder traversal

    > The trick is to set a range {min .. max} for every node.

    Follow the below steps to solve the problem:

    - Initialize the range as {INT_MIN .. INT_MAX}
    - The first node will definitely be in range, so create a root node.
    - To construct the left subtree, set the range as {INT_MIN … rootData}.
    - If a value is in the range {INT_MIN .. rootData}, the values are part of the left subtree.
    - To construct the right subtree, set the range as {rootData..max .. INT_MAX}
  - Postorder Traversal
- BST

  Max possible height of BST with n elements :

  n-1

  Min possible height of BST with n elements :

  floor(log2(n))
- AVL

- How many structurally different BSTs can you form with **n distinct** elements?
    - Total number of possible BSTs with n distinct keys : $\frac{(2n)!}{((n+1)!*n!)}$
- Max Number of nodes /vertices in AVL of height h: $2^{h+1} - 1$
- Min Height of AVL tree with n nodes: **floor(log$_2$n)**
- Max Height of AVL with n nodes: can't exceed $1.44 * log_2 n$
- Max number of nodes at level l in a binary tree : 2^l
- Max number of nodes in binary tree with height h: $2^{h+1} - 1$
- Binary Tree with N nodes . Height of tree : $h \geq lg(N+1) - 1$
- Minimum number of nodes required to get X rotations for delete: N_{2x}
- Insert N elements into an AVL tree. Total num of rotations : O(N)
- After an insertion, before doing any rotations, max imbalance of node is 2.

(FALSE): If every node of a binary tree has 0 or 2 children, then height is O(lgN)

(FALSE): In an AVL, median is either root node, or one of its two children

(FALSE): N inserts in an AVL can be O(N). (correct ans : O(NlgN) due to rebalancing)

(FALSE): At any nodes in an AVL, left and right subtree differ by $\leq$1 height. Therefore, any two leaf nodes differ by $\leq$1 depth.

(FALSE): Let X be any node in AVL. `leftRotate(X)` followed by `RightRotate(X)` does not change the tree.

(FALSE): AVL tree with height h has at least 2^h nodes. ($h_3 \ min = 7 \not\geq 8$)

▼ Trees + Graphs

▼ Terminologies

▼ General

Planar:

- It can be drawn in such a way that no edges cross each other.
- Fact 1: E ≤ 3V for simple planar graphs.

- Fact 2: planar graphs remains planar after edge contractions/deletions.

Diameter of a graph: *maximum shortest distance between any two nodes in a graph G(V,E)*

▼ Undirected Graph

undirected edge:
e: (**u**, **v**) is said to be incident with its two end-point vertices: and

adjacent:

Two vertices are called adjacent (or neighbor) if they are incident with a common edge For example, edge (0, 2) is incident to vertices 0+2 and vertices 0+2 are adjacent

Two edges are called adjacent if they are incident with a common vertex. For example, edge (0, 2) and (2, 4) are adjacent.

degree:

The degree of a vertex **v** in an undirected graph is the number of edges incident with vertex . A vertex of degree 0 is called an isolated vertex. For example, vertex 0/2/6 has degree 2/3/1, respectively.

subgraph:

A subgraph **G'** of a graph **G** is a (smaller) graph that contains subset of vertices and edges of **G**. For example, a triangle {0, 1, 2} is a subgraph of the currently displayed graph.

path:

A path (of length **n**) in an (undirected) graph **G** is a sequence of vertices **{v0, v1, ..., vn-1, vn}** such that there is an edge between **vi** and **vi+1** ∀**i** ∈ [0..**n**-1] along the path

If there is no repeated vertex along the path, we call such path as a simple path

For example, {0, 1, 2, 4, 5} is one simple path in the currently displayed graph

Connected:
An undirected graph **G** is called connected if there is a path between

every pair of distinct vertices of **G**. For example, the currently displayed graph is not a connected graph.

Connected Component:

An undirected graph **C** is called a connected component of the undirected graph **G** if:

1. **C** is a subgraph of **G**;
2. **C** is connected;
3. no connected subgraph of **G** has **C** as a subgraph and contains vertices or edges that are not in (i.e., is the maximal subgraph that satisfies the other two criteria).

For example, the currently displayed graph have {0, 1, 2, 3, 4} and {5, 6} as its two connected components.

Trivial cycle (not a cycle):

In an undirected graph, each of its undirected edge causes a *trivial* cycle (of length 2) although we usually will not classify it as a cycle.

▼ Directed Graphs

Directed edge **e,**(**u** → **v**):
we say that **v** is adjacent to **u** but not necessarily in the other direction.
For example, 1 is adjacent to 0 but 0 is not adjacent to 1 in the currently displayed directed graph.

Degree:

we have to further differentiate the degree of a vertex **v** into in-degree and out-degree. The in-degree/out-degree is the number of edges coming-into/going-out-from **v**, respectively.
For example, vertex 1 has in-degree/out-degree of 2/1, respectively.

Strongly Connected Component (SCC):
we extend the concept of Connected Component (CC)
into *Strongly* Connected Component (SCC).

In the currently displayed directed graph, we have {0}, {1, 2, 3}, and {4, 5, 6, 7} as its three SCCs.

Cycle:

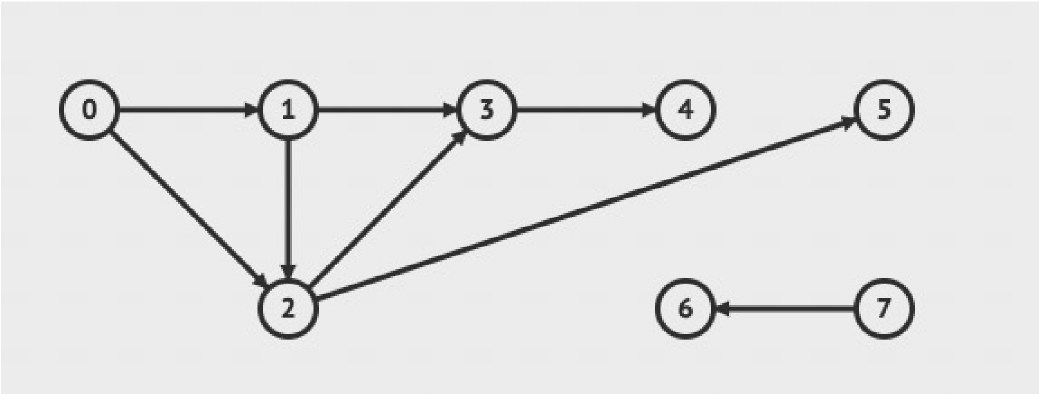
is a path that starts and ends with the same vertex.

Acyclic graph:

is a graph that contains no cycle.

Directed Acyclic Graph (DAG) :

A directed graph that is also acyclic has a special name: Directed Acyclic Graph (DAG). As shown below.



▼ Trees

Tree is a connected graph with **V** vertices and **E = V-1** edges, acyclic, and has **one unique path** between any pair of vertices. Usually a Tree is defined on undirected graph.

As a Tree only have **V-1** edges, it is usually considered a **sparse** graph.

▼ Complete graph

Complete graph is a graph with **V** vertices and **E = V*(V-1)/2** edges (or **E** = O(V^2)),

▼ Bipartite

Bipartite graph is an **undirected** graph with **V** vertices that can be partitioned into two disjoint set of vertices of size **m** and **n** where **V = m+n**.

There is no edge between members of the same set. Bipartite graph is also free from odd-length cycle.

Trees are bipartite

- DAG

DAG is a directed graph that has no cycle, which is very relevant for Dynamic Programming (DP) techniques.

Each DAG has at least one Topological Sort/Order which can be found with a simple tweak to DFS/BFS Graph Traversal algorithm.

Graphs Structures

- Adjacency Matrix

Used when we want to know the existence of edge (u,v)

Space complexity : O(V^2)

Time complexity in finding neighbours of u: O(V)

- Adjacency List

When we need to enumerate through neighbours of u frequently

Space complexity: O(**V**+**E**)

Time complexity in finding neighbours of u: O(K)

- Edge List

When we want edges in sorted order, easily sortable

Space complexity: O(**E**)

Graph Traversals

- DFS

same as pre-order traversal

O(V+E) if we can visit all K vertices in O(k) time, which is only achievable by AL

- BFS

O(V+E) if we can visit all K vertices in O(k) time, which is only achievable by AL

unweighted graph, BFS spanning tree of the graph equals to its SSSP spanning tree if start from same source vertex

▼ Applications of DFS/BFS

▼ Reachability test

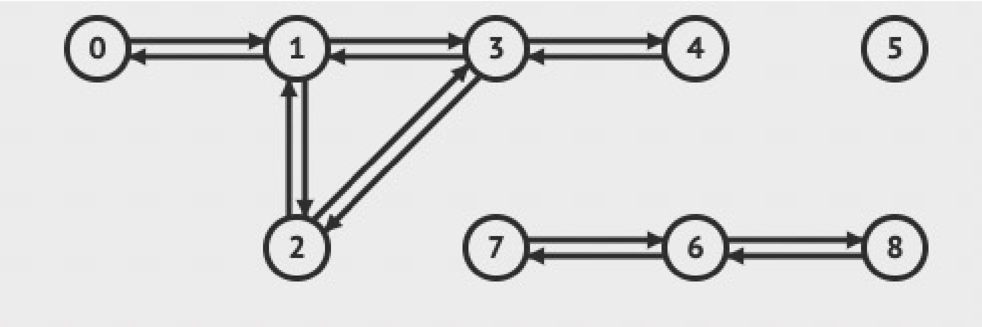
If you are asked to test whether a vertex **s** and a (different) vertex **t** in a graph are reachable, i.e., connected directly (via a direct edge) or indirectly (via a simple, non cyclic, path), you can call the O(**V**+**E**) `DFS(s)` (or `BFS(s)`) and check if `status[t] = visited`

▼ Actually printing the traversal path

```
method backtrack(u)
  if (u == -1) stop
  backtrack(p[u]);
  output vertex u
```

▼ Identifying, Counting, Labelling Connected Components (CCs) of undirected graphs

▼ Identifying CCs



We can enumerate **all** vertices that are reachable from a vertex **s** in an **undirected graph** (as the example graph shown above) by simply calling O(**V**+**E**) `DFS(s)` (or `BFS(s)`) and enumerate all vertex **v** that has `status[v] = visited`

Example: **s = 0**, run `DFS(0)` and notice that `status[{0,1,2,3,4}] = visited` so they are all reachable vertices from vertex 0, i.e., they form one **Connected Component (CC)**

▼ Counting CCs

We can use the following pseudo-code to count the number of CCs:

```
CC = 0
for all u in V, set status[u] = unvisited
```

```
for all u in V
  if (status[u] == unvisited)
    ++CC // we can use CC counter number as the CC label
    DFS(u) // or BFS(u), that will flag its members as visited
output CC // the answer is 3 for the example graph above, i.e.
// CC 0 = {0,1,2,3,4}, CC 1 = {5}, CC 2 = {6,7,8}
```

You can modify the DFS(u)/BFS(u) code a bit if you want to use it to label each CC with the identifier of that CC.

Time Complexity : O(V+E)

▼ Detecting if a graph is cyclic

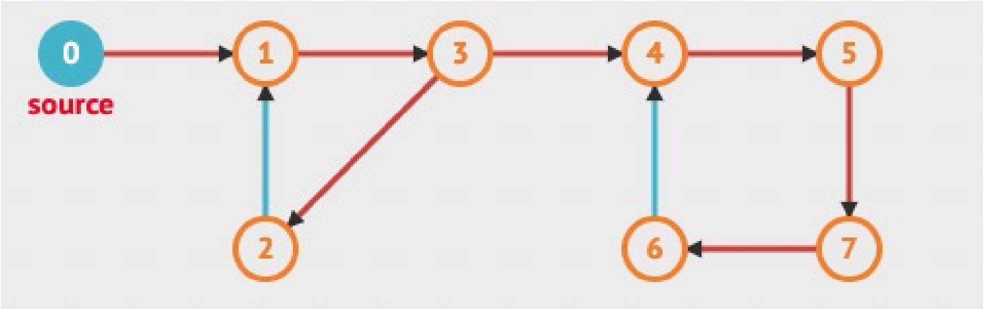
In this visualisation, we use blue colour to highlight **back** edge(s) of the DFS spanning tree. The presence of at least one back edge shows that the traversed graph (component) is **cyclic** while its absence shows that at least the component connected to the source vertex of the traversed graph is **acyclic**

Back edge can be detected by modifying array `status[u]` to record **three** different states:

1. **unvisited**: same as earlier, DFS has not reach vertex **u** before,
2. **explored**: DFS has visited vertex **u**, but at least one neighbour of vertex **u** has not been visited yet (DFS will go depth-first to that neighbour first),
3. **visited**: now stronger definition: all neighbours of vertex **u** have also been visited and DFS is about to backtrack from vertex `u` to vertex `p[u]`

If DFS is now at vertex **x** and explore edge **x** → **y** and encounter `status[y] = explored` `,` we can declare **x** → **y** is a **back edge** (a cycle is found as we were previously at vertex **y** (hence **status[y] = explored**), go deep to neighbour of **y** and so on, but we are now at vertex **x** that is reachable from **y** but vertex **x** leads back to vertex **y**).

▼ Example



The edges in the graph that are not tree edge(s) nor back edge(s) are coloured grey. They are called **forward or cross edge(s)** and currently have limited use (not elaborated).

Now try `DFS(0)` on the example graph above with this new understanding, especially about the 3 possible status of a vertex (unvisited/normal black circle, explored/blue circle, visited/orange circle) and back edge. Edge 2 → 1 will be discovered as a back edge as it is part of cycle 1 → 3 → 2 → 1 (as vertex 2 is `explored' to vertex 1 which is currently `explored') (similarly with Edge 6 → 4 as part of cycle 4 → 5 → 7 → 6 → 4).

Note that if edges 2 → 1 and 6 → 4 are reversed to 1 → 2 and 4 → 6, then the graph is correctly classified as acyclic as edge 3 → 2 and 4 → 6 go from `explored' to `fully visited'. If we only use binary states: `unvisited' vs `visited', we cannot distinguish these two cases

- ▼ Topological Sort (only on DAGs)

  Topological sort of a DAG is a linear ordering of the DAG's vertices in which each vertex comes before all vertices to which it has outbound edges.

  DFS: add one more line, basically post-order traversal

  BFS: Khan’s Algorithm: add in vertices with no incoming edges

▼ Augmentations/modifications

- ▼ 0-1 BFS

  Use a Deque,

  0 weight edges to front of deque (same level)

  1 weight edges to back of deque (next level)

- ▼ Dial’s Algorithm

basically limited range dijsktra, each bucket having their own queue

then start from smallest bucket to largest

▼ SSSP

▼ Definitions:

Bellman-Ford Algorithm O(V*E)

Unweighted graph / positive constant weight : BFS O(**V**+**E**)

Graphs without negative weight: Dijsktra O((**V**+**E**) log **V**)

Graphs without negative weight cycle: Modified Dijsktra O((**V**+**E**) log **V**)

Tree : dfs/bfs O(**V**+**E**)

DAG : DP O(**V**+**E**)

Negative-weight cycle graphs : UNSOLVABLE

Termination

Graph property	Optimised Bellman-Ford		Original Dijkstra		Modified Dijkstra	
	Terminate	Result	Terminate	Result	Terminate	Result
Negative **cycle**	YES	WA	YES	WA	NO	NA
Negative **edge**	YES	AC	YES	WA/AC*	YES	AC†
Dijkstra Killer	YES	AC	YES	WA	YES	AC‡
BF Killer	YES	AC	YES	AC	YES	AC

Special graphs:

- Dijkstra Killer: No negative **cycle**
- BF Killer: No negative **edge**

Footnotes:

*: Depends on the graph, could be either!

†: Might take more than $O((V+E)\ log\ V)$

‡: Takes exponential time (very long)!

SSSP Strategies

Graph property	Best strategy	Time complexity
Tree	BFS/DFS	$O(V+E)$
DAG	Relax vertices in topologically sorted order	$O(V+E)$
Unweighted	BFS	$O(V+E)$
No negative weighted edge	Original Dijkstra	$O((V+E)\ log\ V)$
No negative weighted cycle	Modified Dijkstra	$\approx O((V+E)\ log\ V)$
Negative weighted cycle	None! Distance is ill defined. Can be detected using Bellman-Ford	N.A

▼ MST

Select a **subset** of edges of **G** such that the graph is still connected but with minimum total weight.

Prim's : (E log V)

Kruskal's : (E log V)