# Complexity Analysis

Big-O, *Upper Bound*:

* For two functions f(n) and g(n), we say that f(n) is O(g(n)) iff there exists a constant c and N such that f(n) < cg(n) for all n > N

Big-Ω, *Lower Bound*:

* For two functions f(n) and g(n), we say that f(n) is Ω(g(n)) iff there exists a constant c and N such that f(n) > cg(n) for all n > N

Big-Θ, *Growth Rate*:

* For two functions f(n) and g(n), if three constants c, d, n exist such that:

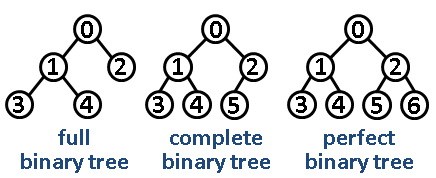
dg(n) < f(n) < cg(n) for all n > N

then f(n) is in Θ(g(n))

# Arrays vs Linked Lists

|  |  |  |
| --- | --- | --- |
|  | One Search | One Insert |
| Unsorted Array | n | 1 |
| Sorted Array | logn | n |
| Unsorted Linked List | n | 1 |
| Sorted Linked List | n | n |

# Binary Search Trees



|  |  |
| --- | --- |
|  | Description |
| Full Tree | Every node has either 0 or 2 children |
| Complete | Every level is filled (the last doesn’t matter) |
| Perfect | All inner nodes have two children and all leaves have the same depth or same level |

When the tree is perfectly balanced; insertion, deletion and search are all O(logn) since the height of the tree is at most logn. However, the worst case run time occurs when the items are inserted in sorted order, which will make the BST degenerate into a linked list resulting in O(n) time.

# Tree Traversal

|  |  |  |
| --- | --- | --- |
|  | Description | Usage |
| Pre Order | Visit root, visit left subtrees, visit right subtrees | To copy a BST |
| In Order | Visit left subtree, visit root, visit right subtree | Retrieve data in order |
| Post Order | Visit left subtree, visit right subtree, visit root | To free/delete a BST |

The in-order predecessor is the left inner node whereas the in-order successor is the right inner node. These are used to replace root or nodes having children

# Hash tables

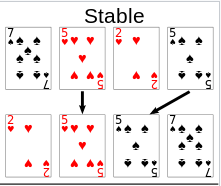
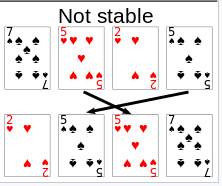
|  |  |  |
| --- | --- | --- |
|  | Description | Insertion probes |
| Chaining | Use a linked list to store values |  |
| Linear Probing | Put the item in the next available slot. Error if the table is full |  |
| Double Hashing | Choose a second hash function |  |

Has an average O(1) look up case if the hash table is managed well. The worst case is O(n) with chaining.

Hashing functions are best used in conjunction with a prime number and modulo

# Stable Sort / Distribution Counting

Distribution counting is a sorting algorithm that considers the order of the elements. The algorithm will take an input of an array, count the frequency to build a cumulative distribution array and then redistribute them into the output array.



# Selection Sort

|  |  |  |
| --- | --- | --- |
| Worst Case | Average Case | Best Case |
| n2 | n2 | n2 |

Selection sort is a naïve array sorting algorithm. It will iterate through the array looking for the smallest element and swap it with the first unsorted element. The algorithm is **in place**, but **unstable** as it will swap elements regardless of order.

# Insertion Sort

|  |  |  |
| --- | --- | --- |
| Worst Case | Average Case | Best Case |
| n2 | n2 | n (already sorted) |

Insertion sort is a naïve array sorting algorithm. It will iterate through the array and move each element to its appropriate spot in the array as it goes. The algorithm is **in place** and **stable** since it moves each element (first in first out). Insertion sort is best used when there are few elements, or we know it is close to being fully sorted.

# Quick Sort

|  |  |  |
| --- | --- | --- |
| Worst Case | Average Case | Best Case |
| n2 | nlogn | nlogn |

Quick sort is a divide and conquer sorting algorithm. It will pick a pivot point and split it into partitions – all larger elements on the right and all smaller elements on the left of the pivot. The algorithm is **in place** and **unstable** in efficient implementations since it works with the less than condition regardless of order. The worst case can be avoided by choosing a pivot closest to the middle (median) or generally between the IQR of the array.

# Merge Sort

|  |  |  |
| --- | --- | --- |
| Worst Case | Average Case | Best Case |
| nlogn | nlogn | nlogn |

Merge sort is a divide and conquer sorting algorithm. It will recursively divide the array into n sublists and repeatedly merge and sort to produce new sorted sublists until there is only one remaining. Merge sort is **not in place** but **stable** since it sorts with sublists from the left-hand side.

# Master Theorem

In general, one pass through the input will reduce the problem size by half.

Let a = number of sub problems, b = size of each sub problem and d = time taken.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

To work it out for a general case:

# Priority Queue

makePQ(); // Makes the priority queue

enqueue(PQ, item); // Adds an item to the priority queue

deletemax(PQ); // or deletemin()

emptyPQ(); // Empties all of the queue

changeWeight(PQ, item); // Changes the priority attached to an item

|  |  |  |
| --- | --- | --- |
|  | Construction | Retrieval (Max element) |
| Unsorted | n | n |
| Sorted | n2 | 1 |

# Heap

A heap is a completely balanced tree with every node satisfying the “heap condition”, the condition that **all parent keys must be greater than all child keys**. Furthermore, the heap can also be represented as an array structure.

**Parent = i, value = A[i], Children = A[i \* 2] and A[i \* 2 + 1]**

Deleting the Max:

1. Return the highest priority (root item)
2. Insert the last element of the heap into the root
3. Reduce the size of the PQ by 1
4. Fix the “heap condition” using downheap()

Downheaping:

1. Compare new root to its immediate children (left then right)
2. If a child is greater than the root, then swap with it
3. Continue swapping until the “heap condition” is satisfied

Upheaping:

1. Insert into the heap as the last element
2. Fix the “heap condition” from the bottom up

|  |  |  |  |
| --- | --- | --- | --- |
|  | Finding the Max | Deleting the Max | Inserting |
| Heap | O(1) | O(logn) | O(logn) |

# Heap Sort

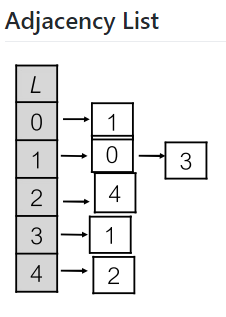
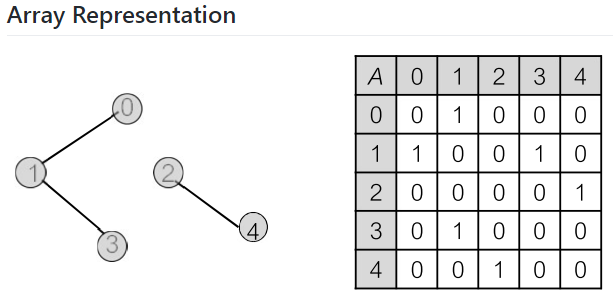
A sorting algorithm with average case O(nlogn) and a better alternative to selection sort.

1. Swap root with the last element
2. Remove the last element by decreasing PQ size by 1
3. Use downheap() to fix the heap array
4. Repeat until no elements left

# Graphs

**DAGs – Directed Acyclic Graphs**, Trees are also simply an undirected graph that is connected and acyclic with any two vertices connected by one path only.

A graph is weak if only some vertices are reachable, likewise strongly if every vertex is reachable from every other vertex. A **complete graph** will **have**



The **array representation costs O(V2)** and the **adjacency list costs O(V+E)**

# Traversals

Depth First Search – DFS:

* Looks for the **deepest** node when searching
* Can be done as: In-Order, Pre-Order and Post-Order

Breadth First Search – BFS:

* Looks for the **shallowest** node when searching

When traversing a graph, you need to have an array that keeps track of all visited nodes, so we aren’t cycling through nodes. **Overall complexity results in O(V+E)**

# Shortest Paths (Dijkstra’s)

The brute force approach results in O(n!) – not feasible.

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph. Below are the detailed steps used in **Dijkstra’s algorithm** to find the shortest path from a single source vertex to all other vertices in the given graph.

1. **Create a set that keeps track of vertices** included in the shortest path. We **initialise this set with infinity and 0 for the current source node**
2. While the set does not include all vertices, we want to:
   * **Pick a vertex u** which is not in the set and has a minimum distance value
   * **Add u** to the set
   * For all **adjacent vertices v**, if the sum of distance values of u (from the source) and the weight of the edge u-v is less than the distance value of v, update the distance value of v

* void run(int\*\* G, int Vsize, int s, int\* pred, int\* dist)
* {
* pq\_node\_t\* pq; /\*\*\*\*\*\*\*\*\*\* Assuming PQ is a minheap \*/
* int u, v;
* pq = makePQ(G); /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* big-O(V) \*/
* while( !emptyPQ(pq) ) /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* big-O(V) \*/
* {
* u = deletemin(pq); /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* big-O(log V) \*/
* for(/\*each v conneted to u \*/)
* if(dist[u] + edgeweight(u,v) < dist[v])
* update(v, pred, dist, pq); /\*\*\*\*\* big-O(log V) \*/
* /\*\*\*\*\*\*\*\*\*\*\*\* big-O(E) for for loop \*/
* }
* }

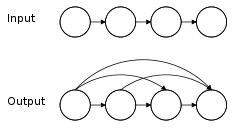
**Dijkstra will have worst case complexity of O(V2logV)** if the graph is dense since E=V2 for dense graphs. **On average, Dijkstra has O((V+E)logV).**

Dijkstra also assumes there are no negative edges – use the **Bellman-Ford algorithm** if there are negative edges **with complexity O(VE).** Note that Bellman-Ford still cannot work with negative cycles.

Using Dijkstra’s for every vertex will yield a **worst-case O(V3logV)** and an **average case O(V(V+E)logV)** – this is called **Johnson’s algorithm** which utilises **a Fibonacci heap** to cut costs.

# Transitive Closure

Transitive closure is the definition of finding out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in a given graph.



In another words, if the above figure represents A->B->C->D, then by transitive closure we can say that A->D is possible through B and C

# Floyd-Warshall Algorithm

The **Warshall** Algorithm – an algorithm that aims to achieve transitive closure in O(v3)

/\* intermediate nodes \*/

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

/\* source node \*/

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

/\* to/destination node \*/

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

if( A[s][i] && A[i][t])

A[s][t] = TRUE; /\* TRUE == 1 \*/

The **Floyd-Warshall** algorithm combines **Warshall** with **Dijkstra’s** in order to solve the All Pairs Shortest Path problem. The end goal is to find the shortest distance for every vertex to any other vertex.

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm

void floydWarshall (int graph[][V])

{

/\* dist[][] will be the output matrix that will finally have the shortest

distances between every pair of vertices \*/

int dist[V][V], i, j, k;

/\* Initialize the solution matrix same as input graph matrix. Or

we can say the initial values of shortest distances are based

on shortest paths considering no intermediate vertex. \*/

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

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

dist[i][j] = graph[i][j];

/\* Add all vertices one by one to the set of intermediate vertices.

---> Before start of an iteration, we have shortest distances between all

pairs of vertices such that the shortest distances consider only the

vertices in set {0, 1, 2, .. k-1} as intermediate vertices.

----> After the end of an iteration, vertex no. k is added to the set of

intermediate vertices and the set becomes {0, 1, 2, .. k} \*/

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

{

// Pick all vertices as source one by one

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

{

// Pick all vertices as destination for the

// above picked source

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

{

// If vertex k is on the shortest path from

// i to j, then update the value of dist[i][j]

// THIS IS DIKSTRA’S CONDITIONAL IF

if (dist[i][k] + dist[k][j] < dist[i][j])

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

// Print the shortest distance matrix

printSolution(dist);

}

# Greedy Algorithms

Greedy algorithms are used in optimization problems. They work on the assumption that the global optimal solution can be found by choosing the local optimal solution for each step.

Optimal:

* An algorithm that never includes an incorrect solution
* May miss a few right answers and not be “complete”

Complete:

* An algorithm that always returns the correct solution
* May include a few wrong answers and not be “optimal”

An optimal algorithm will never return a false result, whereas a complete algorithm will address all possible inputs and return a correct answer

# Spanning Trees / Minimum Spanning Trees

A **spanning tree** of a graph is just a subgraph that contains all the vertices and is also a tree. A graph may have *several* spanning trees.

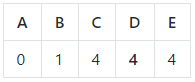
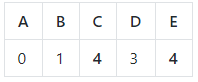
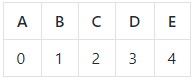
The **minimum spanning tree** is the spanning tree which has the lowest length possible.

# Union-Find Data Structure

Allows a data structure to perform two useful operations: **find** and **union**.

Find: Determines which subset a particular element is in. This can be used for determining if two elements are in the same subset

Union: Join two subsets into a single subset



1. Create an array which maps each node to its corresponding spanning tree
2. Apply Union(C,E) where C can either take E’s value or vice versa
3. Apply Union(C,D) which results in D taking C’s root – 4
4. We continue to do this until the whole array is a single number corresponding to a single root

|  |  |  |
| --- | --- | --- |
|  | Union | Find |
| Array | O(V) | O(1) |
| Tree | O(1) | O(V) |

# Kruskal’s Algorithm

**Kruskal’s** algorithm creates a minimum spanning tree by adding the next lowest weight edge which does **NOT** form a cycle. Kruskal’s algorithm requires the **Union-Find** algorithm as shown above in order to work efficiently.

1. Sort the edges of the graph in increasing order by weight
2. Create an empty subgraph
3. For each edge e in the sorted array
   * If the endpoints of e are disconnected in the subgraph, then add e to the subgraph
4. Return the subgraph

If we take a look at complexity, sorting **dominates** the majority of time complexity – an average case of O(ElogE). We can **improve** this by partial sorting.

# Prim’s Algorithm

**Prim’s** algorithm is best utilised when the graph is dense, and best applied using a matrix representation. The algorithm relies on picking the next best edge (lowest weight) that joins two sets of vertices: vertices already in the tree S and vertices NOT in the tree (V-S).

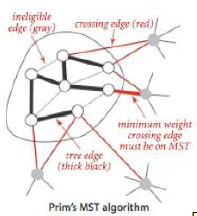
Some definitions:

A **cut** (V,V-S) of G is a partition of V

**Cross**: an edge E=(u,v) with one endpoint in S and the other in (V-S)

**Light Edge**: the minimum weight edge crossing the cut

**Respect**: a cut that **respects** a set A if no edges in A cross the **cut**

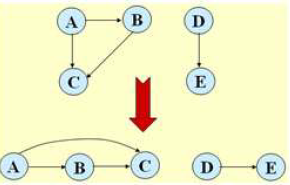


|  |  |  |
| --- | --- | --- |
|  | Prim’s | Kruskal’s |
| General Case | (E+V)logV | ElogV |
| Dense Graph V≤E | ElogV | ElogE |
| Sparse Graph | VlogV | VlogV |

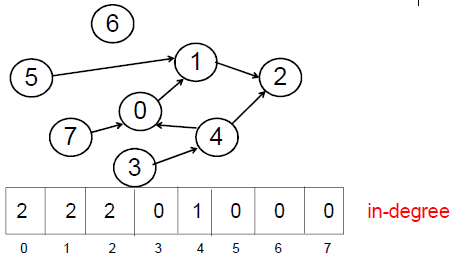
**Prim’s** will be **faster** for **dense graphs** since V≤E, but **Kruskal’s** will be faster for sparse graph since it utilises the **Union-Find** algorithm data structure.

# Topological Sort

Topological sort is a sorting algorithm for DAGs. It is a **partial ordering** that must go in a horizontal direction. It is possible for Toposort to show that a Hamiltonian path exists (but not the actual path)



The **In-Degree Array** is an array that stores the number of incoming edges for every node.



To sort:

1. Identify any **source** (has in-degree of 0)
2. Output the node and remove it from the DAG
3. Update the in-degree array.

For example, if we removed the node 3 from the array since it is a source, we would update the array to have 0 incoming nodes for 4 and as a result, 4 becomes a source node.

**Toposort** must have **at least one source** and **one sink** to work. This can be logically derived from DAGs which must be acyclic. Toposort has complexity O(V+E)

# NP-Completeness

**P** is the class of problems solvable in polynomial time by a deterministic Turing machine. **NP** is the class of problems solvable in polynomial time by a non-deterministic Turing machine. From this, P will always be a subset of NP. We can classify “hardness” of problems using a concept of “reducibility”.

A is **polynomial-time reducible** to B iff there is some function f(w) such that for all w, w is an element of A and f(w) is an element of B.

B is **NP-Hard** iff A ϵ NP is reducible to B in polynomial time. B is **NP-Complete** iff B ϵ NP and B is **NP-Hard**.