**Selection Sort**

* Always O(n2) because selection sort always iterates through the entirety of the unsorted sub-array before choosing a value to add to the sorted subarray

**Insertion Sort**

* Easy to code **–** Fast on small inputs – fast on nearly sorted inputs – O(n2) worst case – O(n2) average (equally-likely inputs) case – O(n2) reverse-sorted case

**Merge Sort**

* Divide and conquer: 1) split array in half 2) recursively sort subarrays 3) linear-time merge-step – O(nlgn) worst case – doesn’t sort in place.
* Runs in O(nlgn) time regardless of the sorting of the input list

**Heap Sort**

* Uses heap data structure (complete binary tree in which parent key > children’s keys) –

O(nlgn) worst case – sorts in place – fair amount of memory shuffling

**Quick Sort**

* Divide and conquer: 1) Partition array into two subarrays, recursively sort 2) all of first subarray < all of second subarray 3) no merge step needed – O(nlgn) average case – fast in practice – O(n2) worst case – Naïve implementation: worst case on sorted input – address this with randomized quicksort
* All comparison sorts (ie. The only operation used to gain ordering info is comparing two elements) must do at least n comparisons)
* Decision trees: Abstraction of any comparison sort, each node is a pair of elements compared, each edge is the result of a comparison

**Heaps**

* Heaps are complete binary trees (full except for the bottom row, which is full from left to right)
* Parent (i) = return [(i-1/2)], left(i) = return 2i + 1, right(i) = return 2i + 2
* Heapify(): maintain the heap property – takes O(lg n) time
* BuildHeap(): walk backwards through the array from node n/2-1 to 0, calling heapify on each node, order guarantees that the children of node I are heaps when I is processed –

O(n)

**Binary Search Trees**

* In-order walk: left first, then node, then right subtree – preorder: print root, then left, then right – post-order: print left, then right, then root
* TreeSearch and TreeInsert run in O(h) time
* Successor: if x has a right subtree, successor is minimum node in right subtree, if x has no right subtree, successor is first ancestor of x whose left child is also ancestor of x
* Deletion: If x has no children, remove x, if x has one child splice out x, if x has two children replace with successor

**Hash Tables**

* Two ways to deal with collisions in hash tables: chaining and open addressing
* Open addressing: To insert, if a slot if full, try another slot until an open slot in found (probing) – for search, if reach element with correct key, return, if reach null, element not in table – good for fixed sets (addition but no deletion)
* Linear Probing: if there’s a collision skip by a constant number of entries
* Quadratic probing: the more collisions I get, the more space I skip
* Double hashing: h(k,i) = (h1(k) + i\*h2(k)) mod m
* Chaining: Assume simple uniform hashing: each key in the table is equally likely to be hashed to any slot – given n keys and m slots in the table, the load factor α = n/m = average num of keys per slot – the average cost of an unsuccessful search will be O(1+load factor) – average for successful serach = O(1+load factor/2) = O(1+ load factor) – so the cost of searching = O(1+ load factor) – if the number of keys (n) is proportional to the number of slots in the table (m), what is α? It’s O(1), ie we can make the expected cost of searching constant if we make α a constant.

**DFS/BFS**

* DFS generates a search tree showing paths from one vertex to all other connected vertices
* Overall running time of BFS = O(|V| + |E|)
* BFS makes sure to visit vertices in increasing order of their distances from the starting point
* BFS finds the shortest paths in any graph whose edges have unit length
* Dijkstra’s is similar to BFS, except that it uses a priority queue to choose vertices in a way that takes the lengths into account
* Dijkstra’s runtime = O((|V| + |E|) log |V|) if the priority queue is implemented using a binary heap.
* Don’t use Dijkstra’s to compute shortest path when you have an unweighted graph or a graph with edges that have negative weights

**Greedy Algorithms – Kruskal’s and Prim’s**

* Kruskal’s: Start with an empty graph and repeatedly add the next lightest edge that doesn’t produce a cycle –Sort edges: O(E lg E) – O(V) MakeSet()s –O(E) FindSet()s –O(V) Union()s –Best disjoint-set union algorithm makes above three operations take O(E x α(E,V)), α almost constant – Overall thus O(E lg E), almost linear without sorting – Note that this is also O (E lg V), since E = O(V2).
* Prim’s: Build it: O(VlgV) – while loop (V times) do ExtractMin O(lg V) – for loop: (E times altogether) do find v in Q and increase key[v] is lg(V) – Total Time: O(ElgV) + O(VlgV) = O(ElgV)

**Graphs**

* Prefer to use an adjacency list rather than a matrix when there are few vertices or the graph is sparse
* For a graph with adjacency lists, the size of the graph is O(E)
* For a graph with an adjacency matrix, the size of the graph is O(V2).
* For a topological sort, a graph must be connected, directed, and acyclic

**Count Sort** – Running time = (n + k)

**Radix Sort** – Best = O(nk) – Average = O(nk), worst = O(nk), worst space complexity = O(n+k)