$O(2^{2n}) \neq O(2^n)$

Data Structures

Binary Search Tree

- Larger elements on the right subtree, smaller elements on the left subtree
- Not necessarily balanced
- Thus, all operations worst-case are O(n)
- Delete:
  1. No children: remove v
  2. 1 child: remove v, connect child(v) to parent(v)
  3. 2 child: find successor, swap(successor, v), remove v (at new position)

A **balanced** tree is a tree with height = O(logn)

AVL Tree (Height-balanced Binary Tree)

- Stores height of the subtree at every node
- **Height-balanced** if |v.left.height – v.right.height| <= 1 for every node in the tree
- A height-balanced tree with height h has **at least** $n > 2^{h/2}$
- A height-balanced tree with n nodes has height h < 2log(n)
- Right rotation: the node in question move down 1 level
- Left rotation: the node in question move up 1 level
- **Special case:**
  Left rotation on a right-heavy left-heavy node: do a right rotation then left rotation on new node
  Right rotation on a left-heavy right-heavy node: do a left rotation then right rotation on new node
- **Worst** case number of rotations after an **insertion**: 2
- Delete:
  1. No children: remove v
  2. 1 child: remove v, connect child(v) to parent(v)
  3. 2 child: swap with successor and delete
  At the end, walk up the tree and do rotation when necessary
  Thus, **deletion** may take up to **O(logn)** time
- **Note:** A standard AVL tree does **not** store weights inside nodes

Trie

- One letter in each node.
- Whether the string exists or not is whether that sequence of characters is a path in the trie
- Every operation is O(L) where L is the length of the string
- **Drawback:** Trie tends to use more **space** as Trie has more nodes and more **overhead**

Interval Tree

- AVL tree but each node is an interval
- Sort each interval by their **start time**
- Store in each node the **maxmimum endpoint** in the subtree
- Find overlap
  While (c != null && x is not in c.interval)
  If (c.left == null)
  c = c.right
  else if (x > c.left.max)
  c = c.right
  else c = c.left
  O(logn)
- Find all overlap = O(klogn) where k is the number of answer

Simple Uniform Hashing assumption:

- Every key is equally likely to map to every bucket
- Keys are mapped independently

Java Hash Function:

- Reflexive, Symmetric, Transitive, Consistent and equals(null) return false

1D Orthogonal Range Searching

- AVL Tree
- Store all points in the leaves of the tree
- Each internal node v stores the MAX of any leaf in the left sub-tree
- FindSplit(low, high) = find the highest node that is in the range [low high]
- LeftTraversal(v, low, high)
  If (low <= v.key)
  All_leaf_traversal(v.right)
  LeftTraversal(v.left, low, high)
  Else
  LeftTraversal(v.right, low, high)
- RightTraversal(v, low, high)
- If (v.key <= high)
  All_leaf_traversal(v.left)
  RightTraversal(v.right, low, high)
  Else
  RightTraversal(v.left, low, high)
- Query time: O(k + logn) where k is the number of points found
- Preprocessing (buildtree) time: O(nlogn)
- Total Space Complexity: O(n)

2D Orthoginal Range Search:

- Build the tree by the points' x-values. For each node, build another tree, y-tree, that contains all the points in that subtree sorted by their y-values
- Query-time: $O(\log^2 n + k)$
- Tree Space Complexity: O(nlogn)
- Building the tree: O(nlogn)
- Do not support insert and delete as it is expensive
- D-dimensional: Query: $O(\log^d n + k)$, buildTree: $O(n\log^{d-1} n)$, Space: $O(n\log^{d-1} n)$
- **Too complicated and some parts doesn't make sense. Use kd-tree instead**

Heap (Binary Heap or Max Heap)

- Properties:
  1. Priority[parent] >= priority[child]
  2. Every level is full, except possibly the last (binary tree)
  3. All nodes are as far left as possible
- **Maximum height**: floor(logn)
- **Insert O(logn)**: add the element as a new leaf and bubble up, comparing it with its parents, stop when the parent is larger
- **Increase key O(logn):** change the key to new value and bubble up
- **Decrease key O(logn):** change the key to new value and bubble down (select the larger children and bubble down that path. Stop when both children are smaller than the new value)
- **Delete O(logn):** swap the node with the last value (rightmost last row), remove last then bubble down newly swapped value
- **Extract Max O(logn):** return root, delete(root)
- Compared to AVL: same cost, faster real cost (no constant factor), no rotations and better concurrency
- Store Heap in an Array:
  1. Parent(index) = floor((index – 1) / 2)
  2. Left(index) = 2 * index + 1
  3. Right(index) = 2 * index + 2
- Heapify O(n): from the largest index down to 0, bubbleDown(a[i])
- Heap -> Sorted Array O(nlogn): call Extract Max until empty
- **HeapSort O(nlogn)**: always complete in O(nlogn), unstable, faster than merge sort
- A sorted array is a Max Heap

Hashing Chaining

- Chain elements with the same hash values in a LinkedList
- Insert = O(1) //LinkedList constant insert time
- Search (Worst) = O(n + cost(hashing)) (all keys hash to the same bucket)
- Search (Expected) = O(1 + n/m) = O(1) (n = no. of element, m = no. of buckets)
- Expected maximum chain length: O(logn)

Hashing Open Address

- On collision, probe a sequence of buckets until find an empty one
- **Search**: probe until find the key is found, if a bucket is null in the process, return false
- **Delete**: cannot set to null because if the key is in the middle of a cluster then search for other keys in the cluster will fail. Instead, set the bucket to a special "deleted" value. When **insert** find this value, overwrite the deleted cell.
  When **search** find this value, continue probing.
- Problem with linear probing: **clusters**
- If the table is ¼ full, then there will be clusters of size θ(logn)
- In reality, linear probing is faster because memory access a bunch of nearby array cells at once and it costs almost 0 to access adjacent array cells
- Double Hashing: hash = f(k) + i * g(k). If g(k) is relatively prime to m, then hash hits all buckets
- For n items, in a table of size m, assuming uniform hashing, the expected cost of an operation is 1 / (1 – n / m)
- **Advantages**: saves space, rarely allocate memory and better cache performance
- **Disadvantages**: more sensitive to choice of hash functions and more sensitive to load
- **Resize table O(n)**
- Resize table:
  If (n == m), then m = 2m (every time double, at least m/2 new items were added)
  If (n < m/4), then m = m / 2 (every time shrink, at least m/4 items were deleted)
- Amortized is NOT average

Bloom Filter

- Do not store keys and values but store integer instead (0 == not exist, > 0 == exist)
- Only false positive
- Use two hash functions to determine two buckets to flip / check
- Insert: at each bucket, ++value
- Delete: at each bucket, --value
- Probability a given bit is 0: $(1 – 1/m)^{kn} == e^{-kn/m}$
- False positive: $(1 – e^{-kn/m})^k$
- Choose k = m/n ln(2)
- Error probability: 2^{-k}

Adjaceny List

- For low E
- Nodes: stored in array
- Edges: linked list per node
- Space: O(V + E)

Adjacency Matrix

- For high E
- V by V matrix of Boolean values
- Value is true if there exists an edge between the row node and column node
- Space: $O(V^2)$
- $Matrix^2$ stores the number of length 2 walks between u and v

Directed Acyclic Graph

- Topological ordering (**not unique**):
  1. Sequential total ordering or all nodes
  2. Edges only point forwa rds
- To find topological ordering, do a Post-Order DFS (i.e. Topological Sorting) O(V + E)
- Alternative Topological Sort:
  Repeat:
  S = all nodes in G that have no incoming edges
  Add nodes in S to the topo-order
  Remove all edges adjacent to nodes in S
  Remove nodes in S from the graph
- Run Relax in the topological order to get shortest paths O(V + E)

Quick Find

- Array based
- Two objects are connected if they have the same component identifier
- **Find(u, v)** O(1): check if the component identifier of u and v are the same
- **Union(u, v)** O(n): flip all component identifier of u to match that of v

Quick Union

- Array based
- Two objects are connected if they are part of the same tree
- **Find(u ,v)** O(n): walk up the tree of u and v and check if they have the same parents
- **Union(u, v)** O(n): walk up the tree of u and v until the root and connect the two roots
- Trees are not balanced so max-height = O(n)

Weighted Union

- Array-based
- Store the size of the subtree rooted at each object
- **Find(u, v)** O(logn): same thing
- **Union(u, v)** O(logn): walk up tree and connect the root of the lighter tree to the root of the heavier tree
- Max Depth: O(logn)
- Optimisation:
  Path Compression: After finding the root, set the parent of each traversed node to the root (i.e. flattening out the tree)
  A = Ackermann function
  Find(u, v): Armortized α(m, n), which is basically O(1)
  Union(u, v): Armortized α(m, n), which is basically O(1)

Minimum Spanning Tree

- No cycle
- If one cut an MST, the two connected component left are also MST
- **Cycle Property**: For every cycle, the **maximum** weight edge is **not** in the MST
  For every cycle, the **minimum** weight edge **may or may not** be in the MST
- A cut of a graph is a partition of the vertices V into two disjoint subsets
- **Cut Property**: For every partition of the nodes, the minimum edge weight across the cut **is** in the MST
  For every vertex, the **minimum** outgoing edge is **always** part of the MST
  For every vertex, the **maximum** outgoing edge **may or may not** be part of the MST

Algorithms

Binary Search

- 1. While start < end:
  a[mid] <= target ? end = mid : start = mid + 1
  2. At the end, check if a[begin] == target and return accordingly
- Time: O(log(n))
- Auxillary Space: O(1)
- Invariant: start <= answerIndex <= end
- Works on any monotonically increasing functions

1D Peak Finding

- Peak definition: a[i – 1] <= a[i] && a[i] >= a[i + 1]
- While start < end:
  a[mid + 1] > a[mid] ? start = mid + 1
  : a[mid – 1] > a[mid] ? end = mid – 1
  : return mid // mid should be a peak after all the previous checks
- Time: O(log(n))
- Auxillary Space: O(1)
- Will always terminate, there will always be a peak
- Invariant: start <= answer <= end
- Steep peaks: a[i – 1] < a[i] && a[i] > a[i + 1] will requires O(n) time because the array may contains all of the same number

2D Peak Finding

- Same peak definition as before
- Find max element on border + cross
  If the element is a peak, return
  Else recurse on the quadrant containing the element bigger than max
- Border is also considered to ensure the peak in the quadrant is higher than or equal to every element on the border (ensure the peak found is also a global peak)
- Time: O(row + col)
- Auxillary Space: O(1)

Bubble Sort

- for (int i = 0; i < a.length; i++)
  for (int j = i; j < a.length; j++)
  if (a[j] > a[j + 1])
  swap(a[j], a[j + 1])
  if (no_swap)
  return;
- Time (Worst: inverse sorted + Average): $O(n^2)$

- Auxillary Space: O(1)
- Best (already sorted): O(n)
- Invariant: At the end of iteration i, the biggest i items are correctly sorted in the final i positions of the array

Selection Sort

- for (int i = 0; i < a.length; i++)
  find smallest element from i to a.length – 1
  swap(a[smallest], a[i]);
- Time: $\theta(n^2)$
- Auxillary Space: O(1)
- Invariant: At the end of iteration i, the smallest i items are correctly sorted in the first i positions of the array

Insertion Sort

```
for (int i = 0; i < a.length; i++)
        int j = i
        while (j > 0)
                if (a[j] < a[j – 1])
                        swap(a[j], a[j – 1])
                        j--
                else
                        break
```
- Time (Worst: inverse sorted + Average): $O(n^2)$
- Auxillary Space: O(1)
- Best (already sorted): O(n)
- Invariant: At the end of iteration i, the elements [0 … i] is correctly sorted

Bubble Sort slow, Insertion Sort fast: when the array is sorted but the largest element is at the front

Inplace == O(1) Auxilary Space

Stable: if a and b are two equal elements and a appears before b before being sorted, then a also appear before b after sorting is done

All the previous sorting algorithms are inplace and stable

MergeSort

```
If (n == 1)
        Return
X = MergeSort(A[0 … n / 2], n/2)
Y = MergeSort(A[n/2 + 1 … n – 1], n/2)
Return Merge(X, Y, n/2);
```
- Time: O(nlog(n))
- Auxillary Space: O(n) //Depth-First algorithm so only a tree of recursion is only expanding along 1 branch at a time
  Lecture Slides says O(nlogn)
- Inplace: **No**
- Stable: **Yes**

QuickSort

```
If (n == 1)
        Return
p = partition(a, n)
x = QuickSort(a[1 … p – 1], p);
y = QuickSort(a[p + 1 … n – 1], n – p – 1);
```

```
partition(A[1..n], n, pIndex)        // Assume no duplicates, n>1
    pivot = A[pIndex];               // pIndex is the index of pivot
    swap(A[1], A[pIndex]);           // store pivot in A[1]
    low = 2;                         // start after pivot in A[1]
    high = n+1;                      // Define: A[n+1] = ∞
    while (low < high)
        while (A[low] <= pivot) and (low < high) do low++;
        while (A[high] > pivot) and (low < high) do high– – ;
        if (low < high) then swap(A[low], A[high]);
    swap(A[1], A[low–1]);
    return low–1;
```

- Time: O(nlog(n))
- Space: O(log(n)) // from recursion stack not from extra arrays
- Inplace: **Yes**
- Stable: **No**
- For duplicates, either pack duplicates or 3-way-partitioning. See slides for implementation. 3-way maintains 4 regions (<n, =n, processing and >n)
- Selection of pivots are randomized.
- **Variant:**
- Paranoid QuickSort: do partition until split into two of at least 1:10 and 9:10
  2/10 chances of selecting a bad pivot => the loop runs at most 2 times in expectation
- **Optimisation:**
  Halt recursion early and do Insertion Sort on small arrays

Order Statistic (finding k^{th} smallest element in an unsorted array)

- Choose a random element as pivot and partition
  If (target < element)
  Recurse left
  Else if (target > element)
  Recurse right
  Else
  Return pivot
- Time: O(n)
- Auxillary Space: O(1)

Tree Traversal

- Can be pre-order, in-order, post order or level-order (BFS). Time: O(n). Space: O(logn)

Graph Searching

- Breadth-First Search: maintains a **queue** of vertices to visit next
- Depth-First Search: maintains a **stack** of vertices to visit next
- Time: O(V + E)
- Space (BFS) = max degree in a graph
- Space (DFS) = max depth in a graph

Bellman-Ford Shortest Path

```
For (i = 0; i < V.length; i++)
        For (Edge e : graph)
                Relax(e)
```
- Relax(e) = if (est[v] > est[u] + e.weight) est[v] = est[u] + e.weight
- Time: O(VE)
- Space: O(V) //est array
- **Invariant**: at the i-th iteration, the estimates of vertices i-th hops or less from the starting node is correct
- Can return **early** when relaxing all edges does not make any changes
- Does not work when there is negative weight cycle
- **Allow** negative weights, just **not** negative weight cycles
- To identify negative weight cycle, run the outer loop V + 1 times. If at the V + 1 iteration, there are still changes, then the vertices that has their estimate changes are on a negative weight cycle
- **Note**: can be used to find longest path if there is no positive weight cycle and the path length is monotonically decreasing

Dijkstra's Shortest Path

- At each step, relax the vertice with the **shortest pat**h (i.e. lowest priority). DecreaseKey if other connected vertices existed or insert if not existed.
- Do not allow negative weights **at all**
- Time: O((V+E)logV) = O(ElogV)
- Space: O(V) //priority queue with max V vertices inside
- **Invariant**: every vertice that is popped from the priority queue and relaxed has the correct estimate
- **On tree**: relax with BFS or DFS order and time: O(V)

Prim's MST Algorithm

- PriorityQueue to store distance from the processed set to the rest
  At the start, startNode in queue
  Repeat until queue is empty:
  1. Dequeue
  2. Add all nodes connected to the dequeued node to the PriorityQueue with priority = weight of edge from dequeue to it. If the node alr existed, decreaseKey
  3. Also put parents of these nodes to the dequeued node
- Time: O((V + E)logV) = O(ElogV)
- Space: O(V) // all nodes are in the Priority Queue
- Variant: All edges have weights from 1 to 10
  1. Use an array of size 10 and linkedlist to implement PriorityQueue
  2. Insert: put node in the correct list
  3. Remove: lookup node and remove from linkedlist
  4. ExtractMin: remove from the minimum bucket
  5. DecreaseKey: look up node and move to the correct linklist

  Time: O(E)

Kruskal MST Algorithm

- Sort all edges
  From smallest to heaviest weight edges:
  1. If the two nodes of edge is already connected, continue
  2. Else, connect the two nodes
- Time: O(ElogE) = O(ElogV^2) = O(ElogV)
- Space: O(E)

If all of the edges have the same weight, use DFS or BFS. Any spanning tree found is a MST.

Maximum Spanning Tree

- Negate all edges and run either Kruskal or Prim

Dynamic Programming

- **Optimal substructure**: the answer for the bigger problem can be derived from the answers to smallest problems
- **Overlapping subproblem**: the answer for the smaller problems is used multiple times by different larger problems (difference between DP and Devide-and-Conquer)

Longest Increasing Subsequence

- Process the array into a DAG (edge from u to v if u is smaller than v)
- From the rightmost node to the leftmost one:
  ans[u] = max(ans[u + 1 … end] + 1)
  ans[u + 1 … end] only contains nodes that has incoming edge from u
- Time: $O(n^2)$
- Space: O(n)

Prize Collecting

- Sub-problem: P[v, k] = maximum prize the one can collect starting at v and taking exactly k steps
- P[v, k] = MAX { P[w_1, k – 1] + w(v, w_1), P[w_2, k – 1] + w(v, w_2), … } where w are nodes connected to v
- Time: $O(kV^2)$
- Space: O(kV)

Minimum Vertex Cover

- Set of node C where every edge is adjacent to at least one node in C
- S[v, 0] = size of vertex cover in subtree rooted at node v, if v is not covered
- S[v, 1] = size of vertex cover in the subtree rooted at node v is v is covered
- S[v, 0] = S[w_1, 1] + S[w_2, 1] + … (w are neighbours of v)
- S[v, 1] = min(S[w_1, 0], S[w_1, 1]) + min(S[w_2, 0], S[w_2, 1]) + …
- Time: $O(V^2)$
- Space: O(2V) = O(V)

Floyd-Warshall All Pair Shortest Paths

- Optimal Sub-structure: If P is the shortest path (u -> v -> w), then P contains the shortest path from (u -> v) and from (v -> w)
- Sub-problem: Let S[v,w,P] be the shortest path from v to w that only uses intermediate nodes in the set P
- Limit P to n + 1 sets. Empty, contains 1, contains 2, …, contains all
- S[v, w, P_8] = min(S[v, w, P_7] , S[v, 8, P_7] + S[8, w, P_7]);
- Time: $O(V^3)$
- Space: $O(V^2)$ // stores only the first hops for each destination, not the entire path
- Good for **dense** graph, but bad for **sparse** graph (Dijkstra V times O(V2logV) is better)

nCr = n! / (r! * (n-r)!)

nPr = n! / (n – r)!

The number of different wats to choosee k out of n unique items:

- Without repetition and without order-significance: nCr
- Without repetition and with order-significance: nPr
- With repetition and with order-significance: n^k
- With repetition and without oder-significance: (k + n – 1)Ck

Every possible path = $\mathbf{2^m}$, where m is the number of edges

Both open **address** and **chaining** has O(n) **worst case** operation

Adjacency List is **more** space efficient than Adjacency Matrix in all scenarios

ab-tree (least **a** and most **b** children, all leaves have same depth) Search Time: O(logn)

1 + 2 + 4 + … + n = O(n)

log(n) because only returns the no. of nodes and do not iterate them out

Shallowest node out of balance for AVL balancing

Symbol table can be implemented with AVL tree

Height = 0 at leaves

Height balanced tree are not (3, 4) weight-balanced when extended far enough