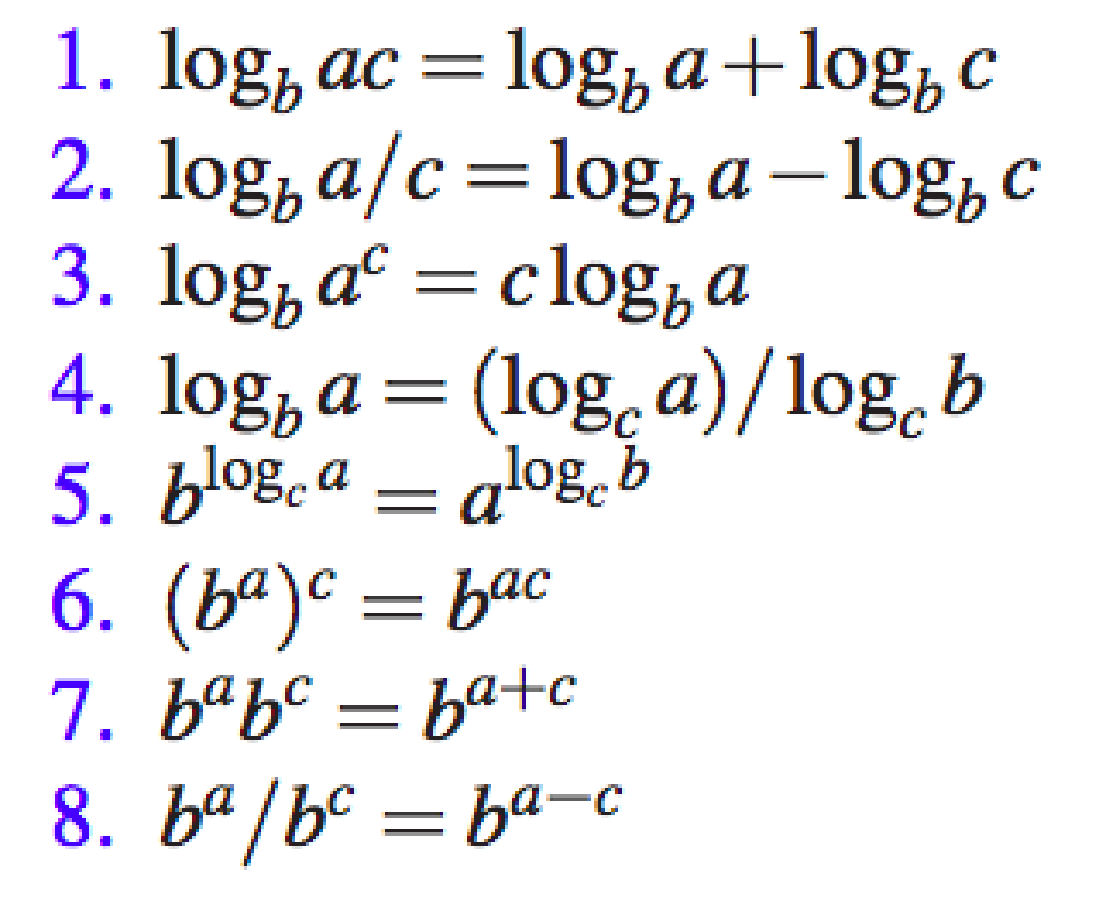
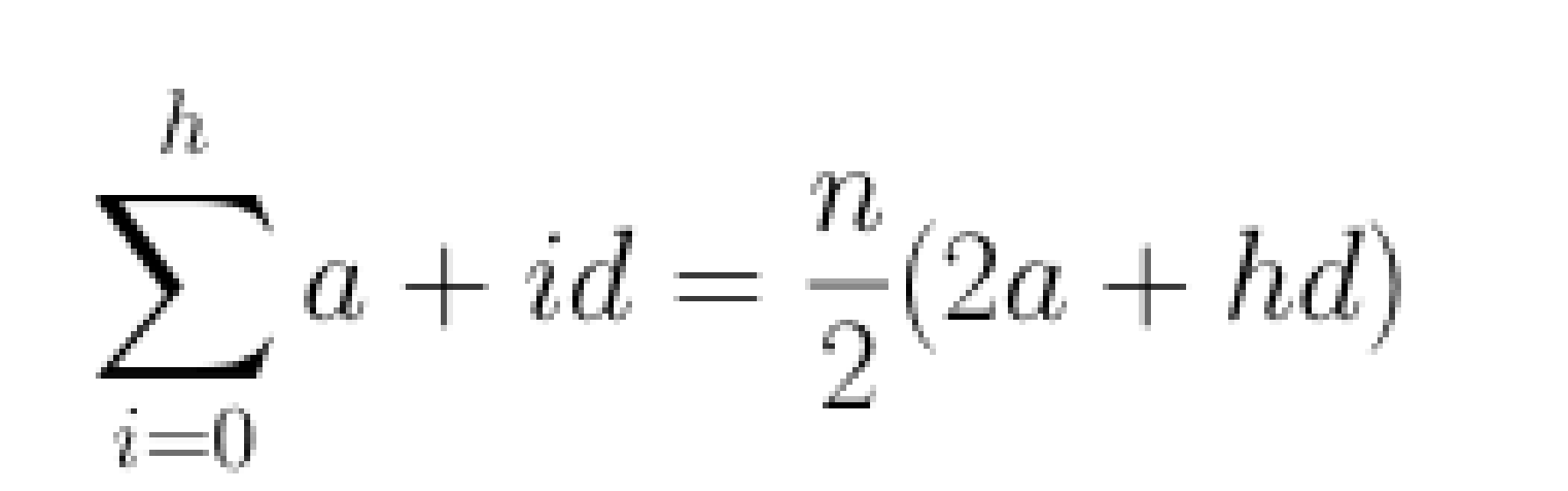
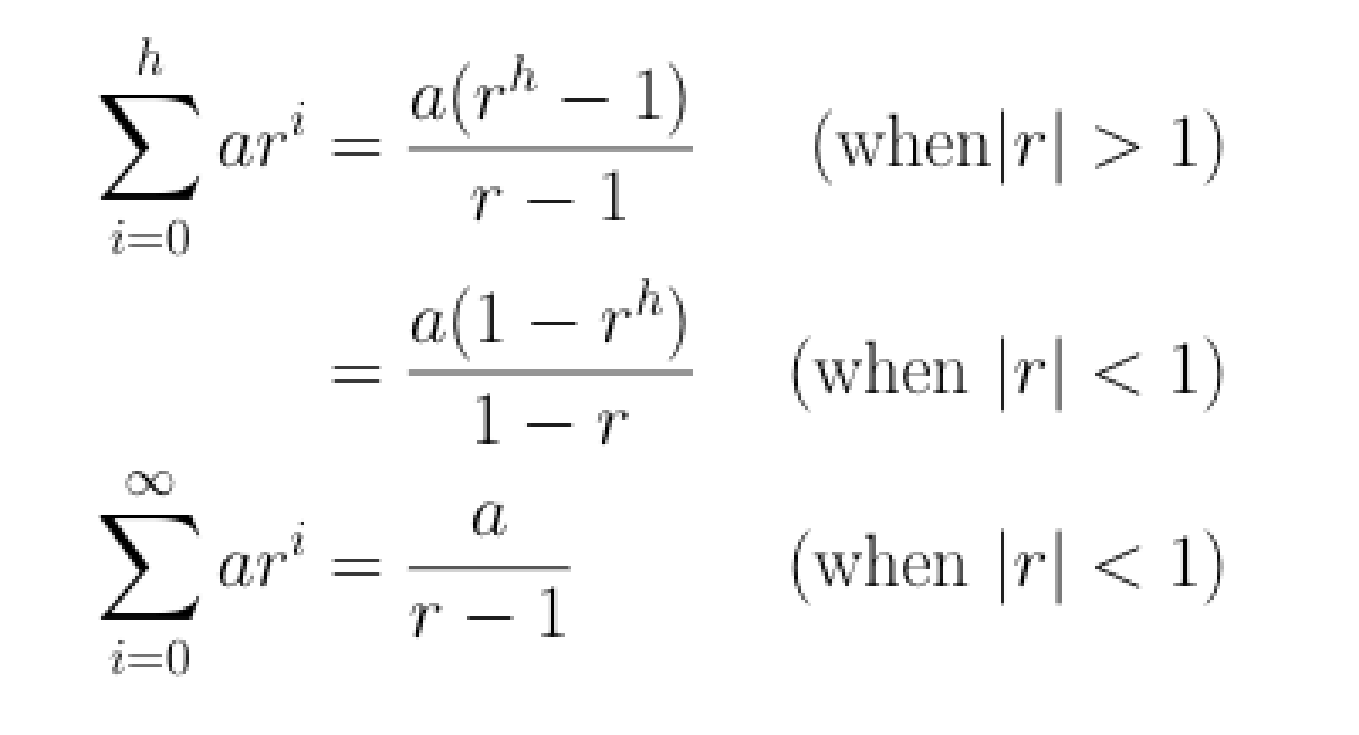
**  **

# Week 1: Introduction

**Need to Know**

• ADTs & CDTs • Reading/writing Java code and pseudocode • Basic memory concepts • Arrays

**Example Question**

Arrays store a set of values by index.

a) Explain how an array is allocated in memory in the Java virtual machine. You may draw a diagram of the memory allocation to assist your explanation.

b) Explain why this provides efficient access to elements in an array cell.

**Abstract Data Type (ADT)**

* + Formal model of a data structure that specifies
    - type of data stored
    - operations supported
    - types of parameters of the operations
  + Specifies what each operation does - not how it does it

**Concrete Data Type (CDT)**

* + Specific implementation of an ADT
  + CDTs are implemented as classes in Java

**Arrays**

* Data structure consisting of a group of elements having a single name that are accessed by indexing
  + computer science definition of an array
* Occupies a contiguous area of storage
  + most programming languages
* Each element has the same data type
  + statically typed programming languages

# Week 1 & 2: Algorithm Analysis

**Need to Know**

• Formal definitions of big-O, big-Omega, and big-Theta • Counting operations

• Calculating time complexity (formally and informally) • Memory complexity

**Example Question**

a) Describe asymptotic analysis, explaining what it tells you about an algorithm or data structure.

Sol: asymptotic computational complexity is the usage of asymptotic analysis for the estimation of computational complexity of algorithms and computational problems, commonly associated with the usage of the big O notation.

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn’t depend on machine specific constants, and doesn’t require algorithms to be implemented and time taken by programs to be compared.

To perform an asymptotic analysis of the worst-case running time of an algorithm

* find the worst-case number of primitive operations executed as a function of the input size – f(n)
  + since constant factors and lower-order terms do not affect the growth rate for large n they are usually disregarded when counting primitive operations
* express this function with big-O notation

b) What does big-O complexity tell us about an algorithm or data structure?

# Week 2: Recursion

**Need to Know**

• Types of recursion • Solving recurrences • The divide and conquer algorithm paradigm

**Example Question**

Express the following recurrence in big-O notation

T(n) = O(1) if n = 1; O(1) + T(n/3) otherwise

T(n) = O(1) + T(n/3) = O(1) + O(1) + T(n/9) = O(1) + O(1) + O(1) + T(n/27) in O(log3n)

**Linear Recursion**

* Perform a single recursive call
* Define each possible recursive call so that it makes progress towards a base case

**Tail Recursion**

* Tail recursion occurs when a linearly recursive method makes its recursive call as its last step
* Easily converted into iterative forms

**Binary Recursion**

* Binary recursion occurs whenever there are two calls for each non-base case

**Multiple Recursion**

* Multiple recursion makes potentially many recursive calls
  + not just one or two
* Multiple recursion is a way of enumerating all possible combinations of a set of elements

**Divide and Conquer**

Binary search uses the Divide and Conquer paradigm

* Divide - Break the problem into smaller subproblems of the same type
* Recur - Recursively solve these subproblems
* Conquer - Combine the solutions for the subproblems into a solution for the problem itself

# Week 2 & 3 sorting algorithm

**Need to Know**

• Insertion sort, selection sort, quicksort, mergesort, bucket-sort, radix-sort • Runtime and memory usage of these algorithms • How to choose/apply these algorithms • Quickselect (covered in week 10)

**Example Question**

Give a list of 4 numbers that exhibits the best-case runtime when sorted using insertion sort

Sol: 1,2,3,4

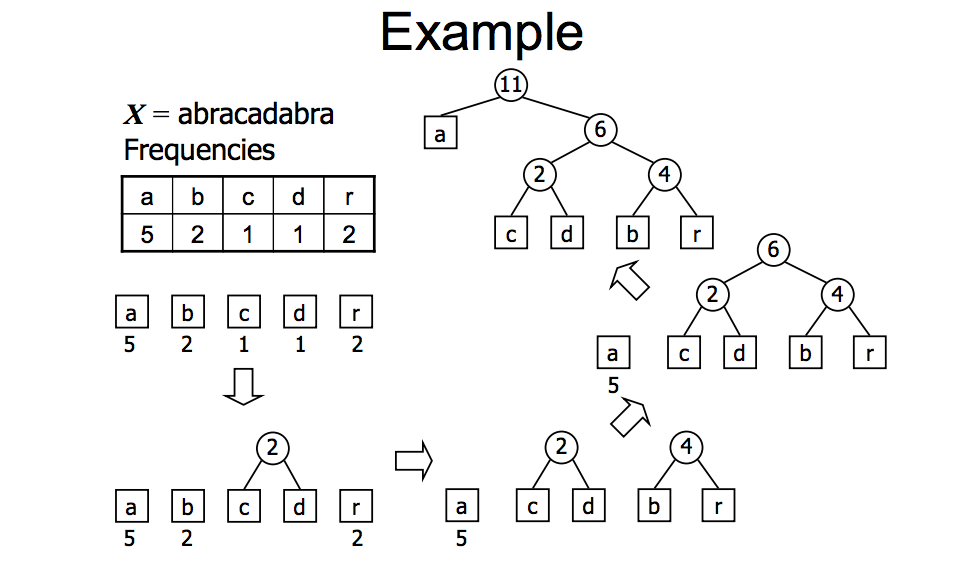
* Inserting sort process
  + Select the first unsorted element
  + Swap other elements to the right to create the connect position and shift the unsorted element
  + Advance the marker to the right one element
* Selection Sort Process
  + Scan each element of the array – find the largest (emax)
  + Swap emax with the last element of the array
  + Repeat this process on the first n - 1 elements
* Divide-and-Conquer used in Merge-Sort
  + Divide: partition A into two halves of about n÷2 elements each
  + Recur: recursively sort each half of A
  + Conquer: merge the two halves of A into a sorted sequence
* Randomised Quick-sort (based on divide-and-conquer paradigm)
  + Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  + Recur: sort L and G
  + Conquer: join L, E and G
* In-Place Quick-Sort
  + Perform partition using two indices to split S into L and E & G (a similar method can split E & G into E and G).
  + Repeat until j and k cross: (j start from left most & k start from right most)
    - Scan j to the right until finding an element > x.
    - Scan k to the left until finding an element < x.
    - Swap elements at indices j and k
* Bucket-Sort
  + Process
    - Let S be a sequence of n (key, element) entries with keys in the range [0, N - 1]
    - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket B[k]
  + Key-type property
    - The keys are used as indices into an array and cannot be arbitrary objects
    - No external comparator
  + Stable Sort property
    - The relative order of any two items with the same key is preserved after the execution of the algorithm
  + Extensions
    - Integer keys in the range [a, b]
      * Put entry (k, o) into bucket B[k - a]
    - String keys from a set D of possible strings, where D has constant size (e.g. names of the 50 U.S. states)
      * Sort D and compute the rank r(k) of each string k of D in the sorted sequence
      * Put entry (k, o) into bucket B[r(k)]
* Lexicographic Order
  + A d-tuple is a sequence of d keys, where key ki is said to be the i-th dimension of the tuple
  + Compared by first dimension, then by second dimension
* Lexicographic-Sort
  + Let Ci be the comparator that compares two tuples by their i-th dimension
  + Let stableSort(S, C) be a stable sorting algorithm that uses comparator C
  + Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm stableSort, one per dimension
* Radix-Sort
  + Specialisation of lexicographic-sort that uses bucket-sort as the stable sorting algorithm

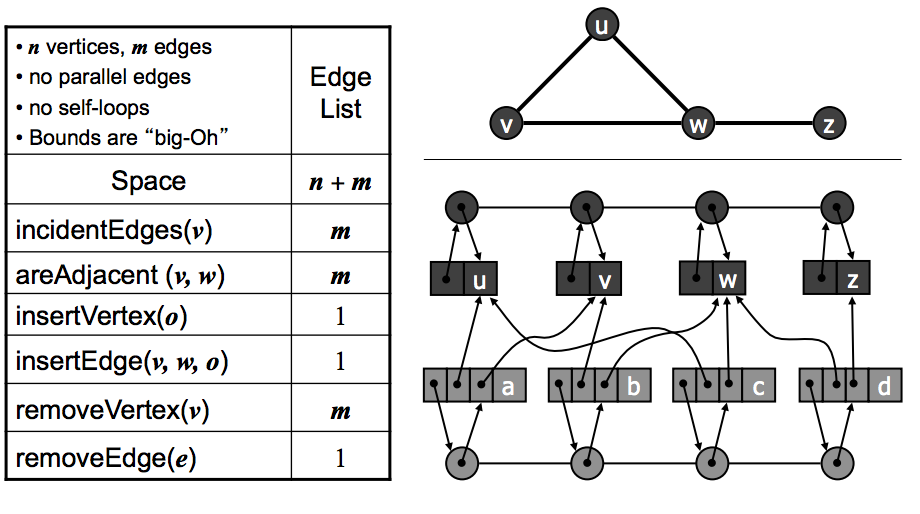
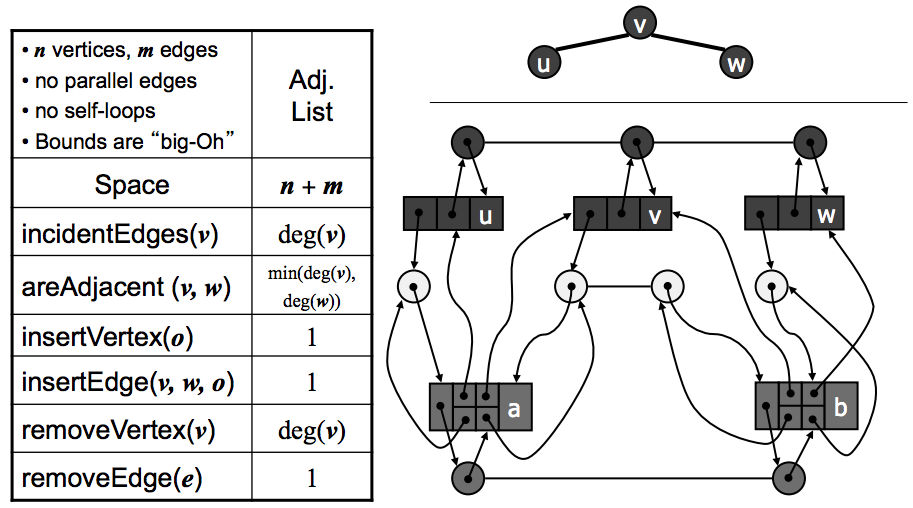
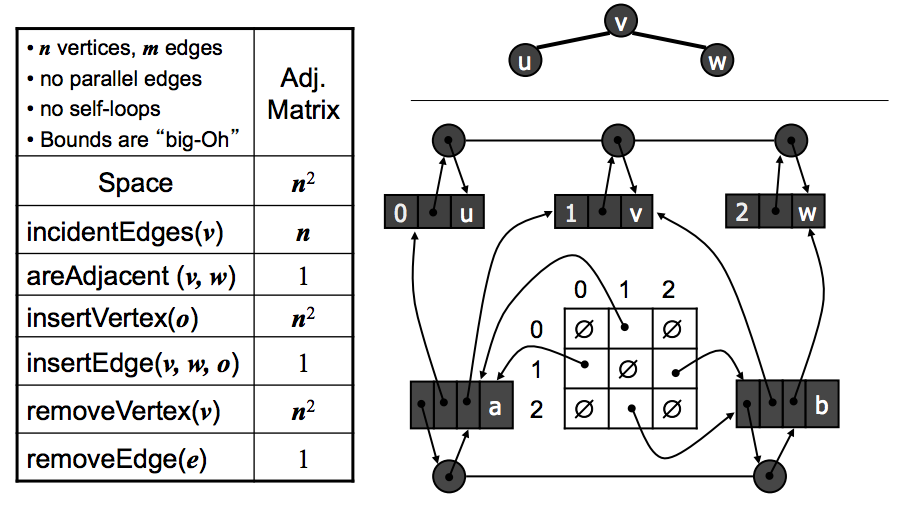
Summary of Sorting Algorithms

|  |  |  |
| --- | --- | --- |
| Algorithm | Time | Notes |
| selection-sort | O(n^2) | - slow - in-place (good for small inputs) |
| insertion-sort | O(n^2) | ^ |
| heap-sort | O(nlogn) | - fast - in-place  - for large data sets (1K - 1M) |
| merge-sort | O(nlogn) | - sequential data access  - fast (good for huge inputs) |
| quick-sort | O(nlogn) expected | - in-place, randomised  - fastest (good for large inputs) |
| Radix sort | O(d(n + N)) | - stable - fastest (with constraints on keys) |

**Week 3 & 4: Linear Data Structures**

* Array based rep of binary trees
  + Rank (root) = 0
  + rank(left) = 2 \* rank(parent) + 1
  + rank(right) = 2 \* rank(parent) + 2
* List based priority queue
  + Unsorted list
    - Insert is O(1)
    - RemoveMin is O(N)
    - Min is O(N)
  + Sorted List
    - Insert is O(N)
    - RemoveMin and Min are O(1)
* Heaps
  + Binary tree storing keys at its nodes
  + Relational property, parents have higher priorities than children.
  + Structural property, heaps must be complete, except last layer which is left filled
  + The last node is the right most deepest external node.
  + Heap order key geq parent.
  + A heap storing N keys has height log N
  + Operations on the heap are proportional to the height
  + Insertion into a heap
    - Find the insertion node (new last node)
    - Store the new k there.
    - Restore the heap order (upheap)
    - Upheaping:
      * Swap k up the heap
      * Stop when k is the root or both of its children are greater than it and its parent is less than it.
    - Removal from the heap
      * Replace the root with the most recent insert
      * Restore the heap order (downheap)
    - Downheap:
      * Swap key down from the root, following the lower children
      * Terminates when a node is reached where both children are greater
* The Map ADT
  + Models a searchable collection of key-value entries
  + Entries can be inserted, deleted and located
  + Keys MUST BE UNIQUE
  + Keys are a unique identifier assigned by an application or user to an associated value
  + A map is appropriate when each key can be viewed as a kind of unique index address for its value
* List based map
  + Unsorted doubly-linked list
  + Put takes O(n) because of need to check for the presence of the key
  + Get and remove take O(n) because if the key is not present the whole list must be inspected
  + Effective only for small maps
* Hashmap
  + Bucket array
  + Hash function - maps an arbitrary object to an integer
    - Hash code and compression function
  + Good hash function
    - Avoids collisions
    - Spreads keys evenly in the array
    - Easy to compute
  + Usually not all the keys are used so we can use a smaller array with a compression function to complete the mapping
  + Collision handling
    - Separate chaining: linked list in each bucket
      * Worst case of O(n) if all the entries end up in the same bucket
    - Open addressing
      * Put an item in a different cell if there is a collision
    - Linear probing
      * Put element in the next available cell
      * Each table cell inspected is called a probe
      * Colliding items clump together, causing long probe chains
      * To handle insertions and deletions we need the sentinel AVAILABLE which replaces deleted items
    - Double hashing
  + Load factor
    - Fraction of table that is full
    - Hash tables are fast provided that the load factor is low.
* Bloom filters
  + Start with a bit array
  + Hash object with k different hash functions
  + Set all the bits given by the hash functions to 1
  + Query
    - Hash object and check all the bits
    - If all are set: maybe
    - If one or more are unset: no.
* Dictionary ADT
  + A searchable **key-value pair collection**
  + find(k) an entry with the key k
  + findAll(k) iterator with keys
  + remove(entry)
  + Unsorted list implementation
    - Insert O(1)
    - find and remove O(N)
    - Effective for small dictionaries or when most operations are insertions (e.g. logs)
    - Suits unordered dictionary
* Hash table implementation
  + Use separate chaining to handle collisions, each operation can be delegated to a list-based dictionary at each hash table cell.
  + Also suits unordered dictionaries.
* Ordered dictionaries
  + Keys come from a total order
  + first()
  + last()
  + successors(k) - iterator of entries with keys greater than or equal to k
  + predecessors (k)
* Search table
  + A dictionary implemented by means of a sorted array
  + Store the items of the dictionary in an array-based sequence sorted by key
  + Use an external comparator for keys
  + Use binary search for find
  + Have to move keys around for insert and remove so both are O(n)
  + Effective only for dictionaries where searches are the most common operation
* Skip lists
  + A skip list is a set of (key, value) pairs in a series of lists such that
    - Each list contains the special keys plus/minus infinity
    - The lowest list contains all the entries in non-decreasing order
    - each list is a subsequence of the previous one
    - The top list contains only the two special keys
  + implementation - quad node
  + Search for key x
    - Start at the first position of the top list
    - At the current position p we compare x with y ← key(next(p))
      * x == y: we return element(next(p))
      * x > y: we scan forward p ← next(p)
      * x < y: we drop down p ← below(p)
      * If we try to drop down past the bottom list, we return null
  + Randomised algorithms
    - Uses random events to control its execution
    - Running time depends on the outcome of the coin tosses
    - We analyse the expected running time of a randomized algorithm under the following assumptions
      * Coins are unbiased
      * Coin tosses are independent
    - Worst-case running time is often large but has a low probability
    - Use a randomized algo to insert items into a skip list
  + Insertion
    - Toss a coin until we get tails; I is the number of times we get heads
    - If I geq h we add new lists
    - We search for x in the skip list and find positions of the items with largest key less than x in each list.
    - For j ← 0 to I, insert item into list Sj after position Pj
  + Space Usage
    - Depends on the outcome of the random events
    - Expected usage is O(N)
* Binary search trees
  + Search trees can be used to implement the dictionary ADT
  + Binary search trees
    - Stores keys at intervals
    - Key of left <= key of parent <= key of right
    - An inorder traversal gets the keys in order
    - Insertion
      * Search for the key
      * Assume its not in the tree and insert at the leaf
  + AVL trees (self balancing binary tree)
    - Height balance property
      * For every instance node, V of T, the heights of the children of V differ by at most 1
    - After an insertion, at worst, the height balance property will be violated by at most one
    - 3 internal nodes can have 5 arrangements as a binary tree but only one is height balanced
* Splay trees
  + Frequency based access
  + A binary tree in which a node is moved to the root after an access
  + Moving to the root is called splaying
  + Deepest internal node accessed is splayed
  + Not always balanced
  + Amortised running time for searches, insertions and deletions (log n)
  + Splays are always performed, even after a failed search
  + remove - splay the parent of the internal node that was actually removed
* (2,4) trees
  + Multiway search tree with
    - Node size every internal node has at most 4 children
    - Depth property - all external nodes have the same depth
  + Insertion
    - At the parent of the leaf node reached by searching for K
    - This preserves the depth property
    - This may cause overflow - split the node
  + Deletion - may cause underflow
    - fusion - when the adjacent sibling of the empty node are 2-nodes
      * move the child of the empty node and a key from the parent
      * Underflow may propagate
    - Transfer - when the sibling of the deleted node is a 3 or 4-node
      * Move a child of sibling to empty
      * Move an item of parent to empty
      * Move an item from sibling to parent
      * No underflow occurs now
* Red-black trees
  + Binary tree representation of a (2,4) tree
  + Red nodes are owned by black nodes
  + Root property - must be black
  + External property - Every leaf is black
  + Internal property - the children of a red node are black
  + Depth property - all leaves have the same black depth
  + Height - at most twice the height of the equivalent search tree so O(logn)
  + search - same as a binary tree
  + AVL trees
    - Tighter restrictions on height
    - Better search performance, slower insert-delete
    - Insertion - Search for the right spot and insert a new red node
      * May result in a double red
    - Fix a double red
      * If the sibling of the red parent is black, fix by restructuring
        + Only one restructure is ever required
      * If that sibling is red, fix by recolouring
        + The parent and the sibling become black and the grandparent becomes red
        + The double red may propagate up
        + Up to O(logn) required
    - Deletion:
      * Run deletion for a binary tree
      * Let v be the internal node removed, w the external node removed and r the sibling of w
        + If either v or r was red, we color r black and finish
        + Else (v and r were both black) we colour r double black, which represents a violation of the internal depth property and requires a reorg
* Set ADT
  + No duplicate elements in a set
  + No explicit notion of keys or order
  + Operations
    - A.union(B)
    - A.intersect(B)
    - A.subtract(B)
  + Generic Methods
    - integer size()
    - boolean isEmpty()
  + Template method
    - Set union
      * aIsLess(a, S)
        + S.insertLast(a)
      * bIsLess(b, S)
        + S.insertLast(b)
      * bothAreEqual(a, bS)
        + S.insertLast(a)
    - Set intersection
      * aIsLess(a, S)
        + { do nothing }
      * bIsLess(b, S)
        + { do nothing }
      * bothAreEqual(a, bS)
        + S.insertLast(a)
  + Generalised merge of two sorted lists A and B
    - Uses auxiliary methods
    - Runs in O(nA + nB)
* Partition (union-find data structure or merge-find set)
  + Collection of disjoint sets
  + A partition of a set X is a division of X into non-overlapping and non-empty “subsets” that cover all of X
  + Operations:
    - find(p): Determine which set a particular element is in
      * Return the set containing the element in position p
    - union(A, B): Combine or merge two sets into a single set
      * Return the set A U B, destroying the old A and B
    - makeSet(x): Makes a set containing only a given element (a singleton)
      * Return the position storing x in this set
* Quick-Select
  + Randomised selection algorithm based on the prune-and-search design pattern
  + Process:
    - Prune: pick a random element x (called pivot) and partition S into
      * L elements less than x
      * E elements equal x
      * G elements greater than x
    - Search: depending on k, either answer is in E or we need to recurse in either L or G
    - Expected running time:
      * Good call: the sizes of L and G are each less than 3s/4
      * Bad call: one of L and G has size greater than 3s/4
      * O(n)
* Pattern-Matching Definitions
  + Let P be a string of size m
    - A substring P[i .. j] of P is the subsequence of P consisting of the characters with ranks between i and j
    - A prefix of P is a substring of the type P[0 .. i]
    - A suffix of P is a substring of the type P[i .. m - 1]
* Pattern Matching Algorithms
  + Brute Force
  + Idea 1 - preprocess the pattern, P
    - The Boyer-Moore algorithm
    - The Knuth-Morris-Pratt algorithm
  + Idea 2 - preprocess the text, T
* Brute Force (PMA)
  + Compares P with the T for each possible shift of P relative to T, until either
    - a match is found
    - all placements of the pattern have been tried
  + O(nm)
* Idea 1
  + Pre-process P and only compare characters when absolutely needed
  + Characteristics necessary
    - Fixed finite alphabet size
    - Patterns can be long
* Boyer-Moore Heuristics
  + Looking-glass heuristic
    - Compare P with a subsequence of T moving backwards
  + Character-jump heuristic
    - When a mismatch occurs at T[i] = c
      * If P contains c, shift P to align the last occurrence of c in P with T[i]
      * Else, shift P to align P[0] with T[i + 1]
* Last-Occurrence Function
  + Boyer-Moore’s algorithm pre processes P and the alphabet Z to build last-occurrence function L mapping Z to integers, where L(c) is defined as
    - the largest index i such that P[i] = c or
    - -1 if no such index exists
  + O(m + s), m is the size of P, s is the size of Z
* Boyer-Moore’s Algorithm
  + O(nm + s)
    - n = size of T
    - m = size of P
    - s = size of alphabet
* KMP Algorithm
  + Process
    - Compares the pattern to the text left-to-right
    - Shifts the largest prefix of P that is a suffix of P[1 .. j]
  + Failure Function F(j)
    - Defined as the size of the largest prefix of P that is also a suffix of P[1 .. j]
    - O(m)
  + Modifies brute-force algorithm so that if a mismatch occurs at P[j] != T[i], set j <- F(j - 1)
  + O(m + n)
* Trie
  + Compact data structure
  + Represents a set of strings
  + Ordered Tree
  + Each node except the root is labelled with a character
  + The children of a node are alphabetically ordered
  + The paths from the external nodes to the root yield the strings of S
  + Worst-case O(n) space
  + Insertions and deletions in time O(dm)
    - n: total size of the strings in S
    - m: size of the string parameter of the operation
    - d: size of the alphabet
  + Each leaf stores references to the word in the text
* Compressed trie
  + Internal nodes of degree at least two
  + Obtained by compress chains of ‘redundant” nodes
  + Stores at the nodes ranges of indices instead of substrings
  + O(s) space
    - s is the number of strings in the array
  + e.g. (i, j, k) -> s[i][j .. k]
* Suffix trie
  + Compressed trie of all the suffixes of string X
  + Uses O(n) (worst-case)
  + Arbitrary pattern matching queries in X in O(dm) time, where m is the size of the pattern
  + Can be constructed in O(n) time
* The greedy method: An algorithm design pattern
  + configurations: different choices, collections or values to find
  + objective function: a score or cost assigned to configurations, which we want to either maximise or minimise
  + search: a solution is found by a sequence of choices of local improvements from a starting configuration
    - 1. Compute cost for configuration
    - 2. Identify decision that makes best cost improvement, make change (go to 1 until no further improvements)
    - when the “greedy-choice property’ holds, the solution is globally optimal
* Huffman codes (optimal variable-length prefix codes)
  + Computer frequency f(c) for each character c
  + Encode high-frequency characters with short code words
  + No codeword is a prefix of another code
  + Use an optimal encoding tree to determine the code words
    - Each external node stores a character
    - The codeword of a character is given by the path from the root to the external node (0 - left child, 1 - right child)
* Huffman algorithm



* Dynamic Programming Technique
  + Simple subproblems: the subproblems can be defined in terms of a few indices
  + Subproblem optimality: the global optimum value can be defined in terms of optimal subproblems
  + Subproblem overlap: the subproblems are not independent, but instead they overlap
* Longest Common Subsequence (LCS) problem
  + longest subsequence common to two strings
  + Brute Force Solution
    - enumate all subsequences of X
    - test which are also subsequences of Y
    - pick longest
    - 2^n subsequences (X is of length n)
  + Dynamic Programming Approach
    - Define L[i, j] to be length of LCS of X[0..i] and Y[0..j]
    - L[-1, k] = 0 and L[k, -1] = 0, indicate no match
    - L[i, j]
      * If xi = yi, then L[i, j] = L[i - 1, j - 1] + 1 (add as match)
      * If xi != yi, then L[i, j] = max{L[i - 1, j], L[i, j - 1]} (no match)
* **Graph Terminology attached for anyone that wants it**
* Properties
  + Notation
    - n = number of vertices
    - m = number of edges
    - deg(v) = degree of vertex v
  + Sumv (deg(v)) = 2m
  + In undirected graph, no self-loops, no multiple edges
    - m <= n (n - 1)/2
* Subgraph
  + Vertices of S are subset of vertices of G
  + ^ of edges
* Representing a Graph
  + Edge list
  + Adjacency list
  + Adjacency matrix
* Sparse graph is a graph G = (v, e), |e| = O(|v|)
* Dense graph is a graph G = (v, e), |e| = Θ(|v|^2)
* Edge List Structure:
  + 
* Adjacency List Structure
  + 
* Adjacency Matrix Structure
  + 
* Depth-First Search
  + Euler tour is to Binary Tree
  + Properties
    - DFS(G, v) visits all vertices and edges in connected component of v
    - Discovery edges labeled by DFS(G, v) form a spanning tree of the connected component of v
  + Runs in O(n + m) provided the graph is represented by the adjacency list structure
* Breadth-First Search
  + Notation
    - Gs connected component of s
  + Properties
    - BFS(G, s) vistis all vertices and edges of Gs
    - The discovery edges labeled by BFS(G, s) form a spanning tree Ts of Gs
    - For each vertex in Li
      * Path of Ts from s to v has i edges
      * Every path from s to v in Gs has at least i edges
  + Runs in O(n + m) provided the graph is represented by the adjacency list structure
* Digraph
  + Graph whose edges are all directed
  + If G is simple, m <= n \* (n - 1)
* Transitive Closure
  + Given a digraph G, transitive closure of G is digraph G\*
    - G\* has same vertices as G
    - if G has a directed path from u to v (u != v), G\* has directed edge from u to v
* Shortest Path Properties
  + A subpath of a shortest path is itself a shortest path
  + There is a tree of shortest paths from a start vertex to all other verticies
* Dijkstra’s Algorithm
  + Assumptions
    - Graph is connected
    - Edges are undirected
    - Edge weights are nonnegative
  + Store each vertex a label d(v) representing distance of v form s in subgraph consisting of cloud and its adjacent verticies
  + Each step
    - Add to the cloud the vertex u outside the cloud with smallest distance label, d(u)
    - Update labels of vertices adjacent to u
  + Consider edge e = (u, z)
    - u is vertex most recently added to the cloud
    - z is not in the cloud
  + Relaxation of edge e updates distance d(z)
    - d(z) <- min{d(z), d(u) + weight(e)}
  + Runs in O((n + m)log n)