Table of Contents

Software Engineering Interview Preparation

The goal of this summary is to contain all the required theoretical material needed to pass a Facebook/Google software engineering interview, but (hopefully) no more than that. It originated out of my own personal notes while preparing for such interviews. I view it as an executive summary, that should ideally take a few hours to read, and that you should read multiple times while preparing for the interview. See this post for more background and tips on preparing for an interview.

Start from SUMMARY. It's also available in GitBook format for easier reading and navigation.

Want to contribute? Great! Please read CONTRIBUTING first.

Complexity

- Big Oh – measuring run time by counting the number of steps an algorithm takes (translating to actual time is trivial).
- `f(n) = O(g(n))` means `c*g(n)` is an upper bound on `f(n)`.
- `f(n) = Ω(g(n))` means `c*g(n)` is a lower bound on `f(n)`.
- `f(n) = θ(g(n))` means `c1*g(n)` is an upper bound on `f(n)` and `c2*g(n)` is a lower bound on `f(n)`.
- With Big Oh we discard multiplication of constants: `n^2` and `1000*n^2` are treated identically.
- Growth rates: `1` < `log(n)` < `n` < `n*log(n)` < `n^2` < `n^3` < `2^n` < `n!`
  - Constant functions
  - Logarithmic: binary search, arises in any process where things are repeatedly halved or doubled
  - Linear: traversing every item in an array
  - Super-linear: quicksort, mergesort
  - Quadratic: going thru all pairs of elements, insertion sort, selection sort
  - Cubic: enumerating all triples of items
  - Exponential: enumerating all subsets of n items
  - Factorial: generating all permutations or orderings of n items
- `O(f(n)) + O(g(n))` => `O(max(f(n), g(n)))` => `n^3 + n^2 + n + 1 = O(n^3)`
- `O(f(n)) * O(g(n))` => `O(f(n) * g(n))`

Recursion Complexity Analysis

- Let `T(n)` be the recursive function.
- Bellow are examples of inner calls of `T(n)`, their complexity, and example algorithm that uses them:
  - `T(n) = T(n/2) + O(1)`, `O(log(n))`, binary search
  - `T(n) = T(n-1) + O(1)`, `O(n)`, sequential search
  - `T(n) = 2*T(n/2) + O(1)`, `O(n)`, tree traversal
  - `T(n) = 2*T(n/2) + O(n)`, `O(n*log(n))`, merge sort, quick sort
  - `T(n) = T(n-1) + O(n)`, `O(n^2)`, selection sort
- Geometric progression:

```
a(n) = a(1)*q^(n-1)
S(n) = a(1)*(q^n-1)/(q - 1)

If it converges:
S(inf) = a(1)/(1 - q)
```

Combinatorics

- All pairs: `O(n^2)`
- All triples: `O(n^3)`
- Number of all permutations: `n!`
- `n` over `k`: number of combinations for choosing `k` items from a set of `n`
  - `(n over k) = n!/(k!*(n-k)!)`
- Number of subsets from a set of `n`: `2^n`
  - Subset = any unique set of `k` elements from `n`, including the empty set).
  - For example: `set={1,2,3}`, `subsets={},{1},{2},{3},{1,2},{1,3},{2,3},{1,2,3}` (ordering in a set doesn't matter).

Handy Formulas

- `1 + 2 + ... + n = n * (n + 1) / 2`
- `x + x/2 + x/4 + ... = 2x`

Data Structures

- Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- Linked data structures are composed of distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

Arrays

- Built from fixed-size records.
- Constant access time given the index.
- Space efficient – arrays consist only of the data, so no space is wasted with links or formatting info.
- Easy to iterate over quickly, because of memory locality.
- Cannot adjust their size in the middle of a program's execution.
- *Dynamic arrays* double in size whenever insert index is out of bound (Java's `ArrayList` is dynamic, while `int[]` isn't).
- Java array max length is `Integer.MAX_VALUE = 2^31-1` (but could actually be a bit shorter because of reserved memory).

Linked Lists

- Singly-linked – each elements links to the next one.
- Doubly-linked – each elements links to the next and previous elements (Java's `LinkedList` is doubly-linked).
- Operations supported: search, insert, delete.
- Cannot overflow (as opposed to static arrays which have a finite length).
- Simpler insertion/deletion than arrays.
- Requires extra memory for pointers than arrays.
- Not efficient for random access to items.
- Worse memory locality than arrays.

Stacks and Queues

- Stacks – LIFO (push, pop)
  - Very efficient, good to use when retrieval order doesn't matter at all (like for batch

jobs).
    - LIFO usually happens in recursive algorithms.
- Queues – FIFO (enqueue, dequeue)
    - Average "waiting time" for jobs is identical for FIFO and LIFO. Maximum time varies (FIFO minimizes max waiting time).
    - Harder to implement, appropriate when order is important.
    - Used for searches in graphs.
- Stacks and Queues can be effectively implemented by dynamic arrays or linked lists. If upper bound of size is known, static arrays can also be used.

Dictionary Structures

- Enable access to data items by content (key).
- Operations: search, insert, delete, max, min, predecessor/successor (next or previous element in sorted order).
- Implementations include: **hash tables**, **(binary) search trees**.

Binary Search Trees

- Each node has a key, and *all* keys in the left subtree are smaller than the node's key, *all* those in the right are bigger.
- Operations: search, traversal, insert, delete.
- Searching is `O(log(n)) = O(h)`, where `h = height` of the tree (`log(n)` for 2 child nodes in each tree root, if tree is perfectly balanced).
- The min element is the leftmost descendant of the root, the max is the rightmost.
- Traversal in `O(n)`, by traversing the left node, processing the item and traversing the right node, recursively ("in-order traversal").
- Insertion is `O(log(n))` and requires recursion.
- Deletion is `O(log(n))` and has three cases (no children, 1 child, 2 children).
- Dictionary implemented with binary search tree has all 3 operations `O(h)`, `h = ceil(log(n))`, smallest when the tree is perfectly balanced.
- In general, a tree's height can go from `log(n)` (best) to `n` (worst), this depends on the order of creation (insertion).
- For random insertion order, on average, the tree will be `h = log(n)`.
- There are balanced binary search trees, which guarantee at each insertion/deletion that the tree is balanced, thus guaranteeing dictionary performance (insert, delete, query) of `O(log(n))`. Such implementations: **Red-Black tree**, **Splay tree** (see below).
- Tree Rotation is an operation on a binary tree that changes the structure without interfering with the order of the elements.

- A tree rotation moves one node up in the tree and one node down. It is used to change the shape of the tree, and in particular to decrease its height by moving smaller subtrees down and larger subtrees up, resulting in improved performance of many tree operations.

Use BST over hash table for:

- Range searches (closest element to some key)
- Traverse elements in sorted order
- Find predecessor/successor to element

Tree Traversal

- DFS (depth first)
  - Pre-order: root, left, right
  - In-order: left, root, right
  - Post-order: left, right, root
- BFS (breadth first)
  - Implemented using a queue, visit all nodes at current level, advance to next level (doesn't use recursion).

Balanced BST

- A self-balancing binary search tree is any node-based binary search tree that automatically keeps its height small in the face of arbitrary item insertions and deletions.
- **Red-Black Tree**
  - Java's `TreeMap` is a Read-Black tree.
  - Self-balancing is provided by painting each node with one of two colors in such a way that the resulting painted tree satisfies certain properties that don't allow it to become significantly unbalanced.
  - When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties.
  - The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.
  - The balancing of the tree is not perfect but it is good enough to allow it to guarantee searching, insertion, and deletion operations, along with the tree rearrangement and recoloring in `O(log(n))` time.
  - Tracking the color of each node requires only 1 bit of information per node because there are only two colors.
  - Properties: root is black, all null leaves are black, both children of every red node

are black, every simple path from a given node to any of its descendant leaves contains the same number of black nodes.
    - From this we get => the path from the root to the furthest leaf is no more than twice as long as the path from the root to the nearest leaf. The result is that the tree is roughly height-balanced.
    - Insertion/deletion may violate the properties of a red–black tree. Restoring the red-black properties requires a small number (`O(log(n))` or amortized `O(1)`) of color changes. Although insert and delete operations are complicated, their times remain `O(log(n))` .
- **Splay Tree**
    - A self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again (they will move nearer to the root).
    - Performs basic operations such as insertion, look-up and removal in `O(log(n))` amortized time.
    - For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.
    - All normal operations on a binary search tree are combined with one basic operation, called *splaying*.
    - Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree.
    - One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.
    - Frequently accessed nodes will move nearer to the root where they can be accessed more quickly, an advantage for nearly all practical applications and is particularly useful for implementing caches and garbage collection algorithms.
    - Simpler to implement then red-black or AVL trees.
    - The most significant disadvantage of splay trees is that the height of a splay tree can be linear (in worst case), on average `log(n)` .
    - By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.
    - The splay operation consists of rotating the tree around the accessed node and its parent, and around the parent and the grandparent (depends on the specific structure).
- **AVL Tree**
    - The heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, re-balancing is done to restore this property.
    - Lookup, insertion, and deletion all take `O(log(n))` time in both the average and

worst cases.
    - Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.
    - AVL trees are more rigidly balanced than red-black trees, leading to slower insertion and removal but faster retrieval.

Hash Tables

- An implementation of a dictionary.
- A hash function is a mathematical function that maps keys to integers.
- The value of the hash function is used as an index into an array, and the item is stored at that position in the array.
- The hash function `H` maps each string (key) to a unique (but large) integer by treating the characters of the string as “digits” in a `base-α` number system (where `α` = number of available characters, 26 for English).
- Java's `String.hashCode()` function is computed as: `s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]`.
- This produces a very large number, which is reduced by taking the remainder of `H(S) mod m` (where `m` is the size of the hash table and `H` is the hashing function).
- Two distinct keys will occasionally hash to the same value, causing a *collision*.
- One solution to collision is *chaining* – representing the hash table as a array of `m` linked lists, where each list's items are hashed to the same value.
- A second solution to collision is *open addressing* – inserting an item in its place, or if it's taken, in the next available place. Upon query, after finding the hash, you check if the key is identical. If not, you move to the next cell in the array to check there (the array is initiated with all null values, and moving on to the next cell is done until a null cell is encountered). Upon deletion of an item, all items must be reinserted to the array.
- A third solution is *double hashing* – repeatedly hashing on the first hash until the item is found, an empty cell is found, or the entire table has been scanned.
- All three solutions are `O(m)` for initializing an `m`-element hash table to set null elements prior to first insertion.
- Traversing all elements in the table is `O(n + m)` for chaining and `O(m)` for open addressing (`n` = inserted keys, `m` = max hash table size).
- Pragmatically, a hash table is often the best data structure to maintain a dictionary.
- In Java the size of a `HashMap` is always a power of 2, for the bit-mask (modulo equivalent) to be effective.
- In Java 8 the `HashMap` can store bins as trees in addition to linked lists (more than 8 objects get converted to a red-black tree).

String Hashing

- Rabin-Karp algorithms – a way to find a substr in a string. Compute the hash of the pattern p, and of every substring the length of p in the original string, then compare the hashes. This usually takes `O(n + m)`, unless we have a hash collision in which case we continue to check all collisions by the data itself.
- Hashing is a fundamental idea in randomized algorithms, yielding linear expected-time algorithms for problems otherwise `O(n*log(n))`, or `O(n^2)` in the worst case.

Priority Queues

- The heap is a priority queue.
- Provide more flexibility than simple sorting, because they allow new elements to enter a system at arbitrary intervals.
- It is much more cost-effective to insert a new job into a priority queue than to re-sort everything on each such arrival.
- Operations: insert, find-minimum/maximum, delete-minimum/maximum.
- The "priority" criteria can be "length", "size", "id", etc.

Heap

- Heaps are a simple and elegant data structure for efficiently supporting the priority queue operations insert and extract-min.
- They maintain a partial order (which is weaker than the sorted order), but stronger than random – the min item can be found quickly.
- The heap is a slick data structure that enables us to represent binary trees without using any pointers.
- We will store data as an array of keys, and use the position of the keys to implicitly satisfy the role of the pointers.
- In general, we will store the `2^l` keys of the `l`-th level of a complete binary tree from left-to-right in positions `2^(l-1)` to `2^l - 1`.
- The positions of the parent and children of the key at position `k` are readily determined. The left child of `k` sits in position `2k` and the right child in `2k + 1`, while the parent of `k` holds court in position `floor(k/2)`.
- This structure demands that there are no holes in the tree – that each level is packed as much as it can be – only the last level may be incomplete.
- This structure is less flexible than a binary search tree: we cannot move subtrees around easily by changing a single pointer. We also can't store arbitrary tree structures (only full trees) without wasting a lot of space.
- Performing a search in a heap requires linear scanning – we can't do a `log(n)` search

like in a binary search tree.
- All we know in a min-heap is that the child is larger than the parent, we don't know about the relationship between the children.
- To insert an item we add it at the end of the array, and then bubble it up if it's smaller that its parent (switch between the parent and the child), until we reach a smaller parent or the root of the tree. This insertion in `O(log(n))`.
- The minimum of the tree is the root. To extract it, we remove the root, and replace it with the last element in the array (bottom rightmost leaf). We then check if it's smaller than both its children. If not, we perform a swap with the smallest child, and bubble the swap down recursively all the way until the criteria is met. This is called *"heapify"*. This operation requires `O(log(n))`.
- Heapify gets the array, index of the node, and the length of the heap.
- To build a heap from an array iterate from the last element to the first and call heapify on each. This costs `O(n)` for a tight analysis (a less tight analysis yields `O(nlog(n))`, which is correct but not tight).

Fibonacci Heap

- A more efficient priority queue than binary heaps in terms of time complexity.
- Harder to implement.

Graphs

- A Graph is comprised of vertices `V` (the points) and edges `E` (the lines connecting the points).
- Assume `n` is the number of vertices and `m` is the number of edges.
- Directed graphs: inner city streets (one-way), program execution flows.
- Undirected graphs: family hierarchy, large roads between cities.
- Data structures:
  - **Adjacency Matrix**: we can represent a graph using an `n * n` matrix, where element `[i,j] = 1` if `(i, j)` is an edge (an edge from point `i` to point `j`), and `0` otherwise.
  - The matrix representation allows rapid updates for edges (insertion, deletion) or to tell if an edge is connecting two vertices, but uses a lot of space for graphs with many vertices but relatively few edges.
  - **Adjacency Lists**: we can represent a sparse graph by using linked lists to store the neighbors adjacent to each vertex.
  - Lists make it harder to tell if an edge is in the graph, since it requires searching the appropriate list, but this can be avoided.
  - The parent list represents each of the vertices, and a vertex's inner list represents

    all the vertices the vertex is connected to (the edges for that vertex).
    - Each vertex appears at least twice in the structure – once as a vertex with a list of connected vertices, and at least once in the list of vertex for the vertices it's connected to (in undirected graphs).
- We'll mainly use adjacency lists as a graph's data structure.
- Traversing a graph (breadth or depth) uses 3 states for vertices: undiscovered, discovered, processed (all edges have been explored).
- Most fundamental graph operation is traversal.

Additional Trees

Trie (Prefix Tree)

- A trie is a tree structure, where each edge represents one character, and the root represents the null string.
- Each path from the root represents a string, described by the characters labeling the edges traversed.
- Any finite set of words defines a trie.
- Two words with common prefixes branch off from each other at the first distinguishing character.
- Each leaf denotes the end of a string.
- Tries are useful for testing whether a given query string `q` is in the set.
- Can be used to implement auto-completion and auto-correction efficiently.
- Supports ordered traversal and deletion is straightforward.
- We traverse the trie from the root along branches defined by successive characters of `q`. If a branch does not exist in the trie, then `q` cannot be in the set of strings. Otherwise we find `q` in `|q|` character comparisons regardless of how many strings are in the trie.
- Tries are space and time efficient structures for text storage and search.
- Very simple to build (repeatedly insert new strings) and very fast to search.
- More memory efficient if there are many common prefixes, great for language dictionaries (many words have same prefix).
- Very quick lookups if key isn't found.

Ternary Search Tree

- A type of Trie where nodes are arranged in a manner similar to a BST but with up to three children.
- More space efficient compared to Trie at the cost of speed.
- Common applications include spell-checking and auto-complete.

- Each node has a single character and pointers to three children: `equal`, `lo` and `hi`.
- Can also contain pointer to parent and flag if it's the end of a word.
- The `lo` pointer must point to a node whose character value is less than the current node (`hi` for higher).
- The `equal` points to the next character in the word.
- Average running time for operations: `O(logn)`; Worst-case: `O(n)`.
- Also relevant: Patricia Tree – A compact representation of a Trie in which any node that is an only child is merged with its parent.

Radix Tree

- A more memory-efficient (compressed) representation of a trie (prefix tree).
- Space-optimized by merging parent that have single-child nodes.

Suffix Tree

- A data structure to quickly find all places where an arbitrary query string `q` occurs in reference string `S` (check if `S` contains `q`).
- Proper use of suffix trees often speeds up string processing algorithms from `O(n^2)` to linear time.
- Suffix tree is simply a trie of the `n - 1` suffixes of an `n`-character string `S` (to construct just iterate over all suffixes and insert them into the trie).
- Suffix tree is simply a trie of all the proper suffixes of `S`. The suffix tree enables you to test whether `q` is a substring of `S`, because any substring of `S` is the prefix of some suffix. The search time is again linear in the length of `q`.

B-Tree

- *Balanced* search trees designed to work well on disks; used by many databases (or variants of B-trees).
- Typical applications of B-trees have amounts of data that can't fit into main memory at once.
- Similar to red-black trees but they are better at minimizing disk IO.
- All leaves are at the same depth in the tree.
- Nodes may have many children (1s-1000s).
- A node has `k` keys, which separate (partition) the range of values handled by `k+1` children.
- When searching for a key in the tree we make a `k+1`-way decision based on comparisons with the `k` keys stored at any node (`k` can differ between nodes).
- Nodes are usually as large as a whole disk page to minimize the number of

reads/writes.
- The root of a B-tree is usually kept in main memory.

Interval Tree

- An *interval tree* is a red-black tree that maintains a dynamic set of elements, each containing an interval.
- Two intervals `a` and `b` overlap if: `a.low <= b.high` *and* `a.high >= b.low`.
- The key of each tree node is the *low* endpoint of the node's interval (start time) => an in-order traversal lists the intervals in sorted *start time*.
- Additionally, each node stores a value `max` which is the maximum value of *any* interval's end time stored in the *subtree* rooted at that node.
- To find an overlapping interval in the tree with some external interval `i`, we begin to traverse the tree. If `i.low <= root.left.max` we recurse into the left tree; otherwise to the right one. Keep going until one of the intervals overlap or we reach `null`.

DAWG

- Directed acyclic *word* graph.
- Represents the set of all substrings of a string.
- Similar in structure to a suffix tree.

Bloom Filter

- A probabilistic data structure.
- Allows to test if an element is a member of a set.
- False positive matches **are possible**, but false negatives are not.
- A query returns either "possibly in set" or "definitely not in set".

Data Structures Examples

Stack

Backed by array:

```
public class Stack {
    private int[] arr;
    private int i = -1;

    public Stack(int capacity) {
        arr = new int[capacity];
    }

    public int size() {
        return i + 1;
    }

    public void push(int x) {
        if (i == arr.length - 1) throw new RuntimeException("Stack is full");
        arr[++i] = x;
    }

    public int pop() {
        if (i == -1) throw new RuntimeException("Stack is empty");
        return arr[i--];
    }
}
```

Queue

Backed by array:

```
public class Queue {
    private int[] arr;
    private int front = -1;
    private int rear = -1;

    public Queue(int capacity) {
        arr = new int[capacity];
    }

    public int size() {
        if (rear == -1 && front == -1) return 0;
        else if (rear >= front) return rear - front + 1;
        else return arr.length + rear - front + 1;
    }

    public void enqueue(int x) {
        if ((rear + 1) % arr.length == front) throw new RuntimeException("Queue is ful
l");

        if (size() == 0) {
            front = 0;
            rear = 0;
            arr[front] = x;
        } else {
            rear = (rear + 1) % arr.length;
            arr[rear] = x;
        }
    }

    public int dequeue() {
        if (size() == 0) throw new RuntimeException("Queue is empty");
        int res = arr[front];

        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % arr.length;
        }

        return res;
    }
}
```

Binary Tree

```
class Tree {
    public int value;
    public Tree left;
    public Tree right;

    public Tree(int value) {
        this(value, null, null)
    }

    public Tree(int value, Tree left, Tree right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
}
```

Graph

```java
public class Edge {
    public int weight;
    public int from;
    public int to;
}

public class Vertex implements Comparable<Vertex> {
    public int id;
    public Set<Edge> edges = new HashSet<>();
    public Vertex parent = null;

    public int distance = Integer.MAX_VALUE; // for dijkstra
    public boolean discovered = false;       // for bfs/bfs traversal

    public Vertex(int id) {
        this.id = id;
    }

    /*
     * For dijkstra priority queue
     */
    @Override
    public int compareTo(Vertex o) {
        if (this.distance < o.distance) return -1;
        else if (o.distance < this.distance) return 1;
        else return 0;
    }
}

public class Graph {
    public Map<Integer, Vertex> vertices = new HashMap<>();
}
```

Trie

```
public class Trie {
    public Character c;
    public Map<Character, Trie> chars = new HashMap<>();
    public boolean isLeaf = false;

    /*
     * For root element that has no char
     */
    public Trie() {
        this(null);
    }

    public Trie(Character c) {
        this.c = c;
    }
}
```

BitSet

```
public class BitSet {
    private byte[] a;

    public BitSet(int size) {
        this.a = new byte[size];
    }

    public boolean get(int i) {
        return (a[i / 8] & 1 << (i % 8)) == 1 << (i % 8);
    }

    public void set(int i, boolean value) {
        if (value)
            a[i / 8] |= 1 << (i % 8);
        else
            a[i / 8] ^= 1 << (i % 8);
    }
}
```

Algorithms

- Dynamic programming – compute solutions for smaller instances of a given problem and use these solutions to construct a solution to the problem. For example: adding a cache to recursive Fibonacci calculation.
- Greedy algorithms – compute a solution in stages, making choices that are locally optimum at each step; these choices are never undone.

Divide-and-Conquer

- Break a complex algorithm into small, often recursive parts.
- Allow better parallelization.
- To use divide-and-conquer as an algorithm design technique, we must divide the problem into two smaller subproblems, solve each of them recursively, and then meld the two partial solutions into one solution to the full problem. Whenever the merging takes less time than solving the two subproblems, we get an efficient algorithm.
- Mergesort is a classic example of divide and conquer (the merge operation is linear).

Sorting

- "Naive" sorting algorithms run in `O(n^2)` while enumerating all pairs.
- "Sophisticated" sorting algorithms run on `O(nlog(n))`.
- An important algorithm design technique is to use sorting as a basic building block, because many other problems become easy once a set of items is sorted, like: searching, finding closest pair, finding unique elements, frequency distribution, selection by order, set of numbers intersection, etc.
- *Selection Sort* – each time find the minimum item, remove it from the list, and continue to find the next minimum; takes `O(n^2)`.
- *Insertion Sort* – iterate thru `i = 2 to n`, `j = i to 1` and swap needed items; takes `O(n^2)`.
- Insertion sort is a little more efficient than selection, because the inner `j` loop uses a while, only scanning until the right place in the sorted part of the array is found for the new item. Selection sort scans all items to always find the minimum item.
- Selection and Insertion are very similar, with a difference that after `k` iterations Selection will have the `k` smallest elements in the input, and Insertion will have the arbitrary first `k` elements in the input that it processed.
- Selection Sort *writes* less to memory (Insertion writes every step because of swapping),

so it may be preferable in cases where writing to memory is significantly more expensive than reading.

Mergesort

- A recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving (merging) the two sorted lists to totally order the elements.
- The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list.
- Merging on each level is done by examining the first elements in the two merged lists. The smallest element must be the head of either of the lists. Removing it, the next element must again be the head of either of the lists, and so on. So the merge operation in each level is linear.
- This yields an efficiency of `O(nlog(n))`.
- Mergesort is **great** for sorting **linked lists** because it does not access random elements directly (like heapsort or quicksort), but **DON'T** try to sort linked lists in an interview.
- When sorting arrays with mergesort an additional 3rd array buffer is *required* for the merging operation (can be implemented in-place tho without an additional buffer, but requires complicated buffer manipulation).
- Classic *divide-and-conquer* algorithm, the key is in the merge implementation.

Quicksort

- Can be done in-place (using swaps), doesn't require an additional buffer.
- Select a random item `p` from the items we want to sort, and split the remaining `n-1` items to two groups, those that are above `p` and below `p`. Now sort each group in itself. This leaves `p` in its exact place in the sorted array.
- Partitioning the remaining `n-1` items is linear (this step is equivalent to the "merge" part in merge sort; merge => partition).
- Total time is `O(n * h)` where `h` = height of the recursion tree (number of recursions).
- If we pick the median element as the pivot in each step, `h = log(n)`; this is the best case of quicksort.
- If we pick the left- or right-most element as the pivotal element each time (biggest or smallest value), `h = n` (worst case).
- On average quicksort will produce good pivots and have `nlog(n)` efficiency (like binary search trees insertion).
- If we are most unlucky, and select the extreme values, quicksort becomes selection sort and runs `O(n^2)`.
- For the best case to work, we need to actually select the pivot randomly, or always

select a specific index and randomize the input array beforehand.
- Randomization is a powerful tool to improve algorithms with bad worst-case but good average-case complexity.
- Quicksort can be applied to real world problems – like the *Nuts and Bolts* problem (first find a match between a random nut and bolt, then split the rest into two groups: bigger and smaller. Repeat in each group).
- Experiments show that a well implemented quicksort in typically 2-3 times faster than mergesort or heapsort. The reason is the innermost loop operations are simpler. This could change on specific real world problems, because of system behavior and implementation. They have the same asymptotic behavior after all. Best way to know is to implement both and test.

Heapsort

(See also Heap in Data Structures)

- An implementation of selection sort, but with a priority queue (implemented by a balanced binary tree) as the underlying data structure, and not a linked list.
- It's an in-place sort, meaning it uses no extra memory besides the array containing the elements to be sorted.
- The first stage of the algorithm builds a max-heap from the input array (in-place) by iterating elements from last to first and calling *heapify* on each (cost: `O(n)`).
- Next, the largest element of the heap is the root; we "extract" it (swap the root with the last element) *reducing* the size of the heap (this is important), and calling *heapify* again for the new root of the smaller heap.
- Continue until we're done with the entire heap.
- Time complexity is `O(nlog(n))`.

Distribution Sort

- Split a big problem into sub (ordered) "buckets" which need to be sorted themselves, but then all results can just be concatenated. An example: a phone book. Split names into buckets for the starting letter of the last name, sort each bucket, and then combine all strings to one big sorted phone book. We can further split each pile based on the second letter of the last name, reducing the problem even further.
- Bucketing is effective if we are confident that the distribution of data will be **roughly uniform** (much like hash tables).
- Performance can be terrible if data distribution is not like we thought it is.

External Sort

- Allows sorting more data then can fit into memory.
- For example, assume we have 900MB file, 100MB RAM.
- Read 100MB chunks of the data to memory and sort (quicksort, mergesort, etc.) then save to file, until all 900MB is sorted in chunks.
- Read the first 10MB of each sorted chunk to input buffers (=90MB) + allocate an output buffer (10MB) – sizes can be adjusted.
- Perform a 9-way merge (mergesort is 2-way by default) and store the result in the output buffer.
- Once the output buffer is full, write it to disk to the sorted destination file, empty it, and continue merging the input buffers.
- If any of the input buffers gets empty, fill it with the next 10MB from its chunk.
- Continue until all chunks are processed.
- Total runtime is `nlog(n)`.
- If we have a lot of data and limited RAM, we can run the whole process in two passes: split to chunks (500 files), combine 25 chunks at a time resulting in 20 larger chunks, run second merge pass to merge the 20 larger sorted chunks.

Linear Sorting Algorithms

- Counting sort – `O(n)`, stable, assuming input is integers between `0...k` and that `k = O(n)` – create a new array that first stores the number of appearances for each index, and then accumulate each of the cells to know how many total elements were before each index.
- Radix sort – use counting sort multiple times to sort on the least significant digit, then the next one, etc.
- Bucket (Bin) sort – `O(n)`, sorts `n` *uniformly* spread real numbers between `[0,M)`, by dividing the elements into `M/n` buckets, then sorting each bucket. Uniform distribution will yield linear time; non-uniform will be `O(nlogn)`.

Searching

Binary Search

- Fast algorithm for searching in a **sorted** array of keys.
- To search for key `q`, we compare `q` to the middle key `S[n/2]`. If `q` appears before `S[n/2]`, it must reside in the top half of `S`; if not, it must reside in the bottom half of `S`. Repeat recursively.
- Efficiency is `O(logn)`.
- Several interesting algorithms follow from simple variants of binary search:
  - Eliminating the "equals check" in the algorithm (keeping the bigger/smaller checks)

we can find the boundary of a block of identical occurrences of a search term. Repeating the search with the smaller/bigger checks swapped, we find the other boundary of the block.
    - One-sided binary search: if we don't know the size of the array, we can test repeatedly at larger intervals (`A[1]` , `A[2]` , `A[4]` , `A[8]` , `A[16]` , `...`) until we find an item larger than our search term, and then narrow in using regular binary search. This results in `2*log(p)` (where `p` is the index we're after), regardless how large the array is. This is most useful when `p` is relatively close to our start position.
- Binary search can be used to find the roots of continuous functions, assuming that we have two points where `f(x) > 0` and `f(x) < 0` (there are better algorithms which use interpolation to find the root faster, but binary search still works well).

Randomization

- Randomize array (`O(nlogn)`) – create a new array with "priorities", which are random numbers between `1 - n^3` , then sort the original array based on new priorities array as the keys.
- Randomize array in place (`O(n)`) – swap `a[i]` with `a[rand(i, n)]` .

Selection (`k` -th smallest element)

- Sort array and return `k` -th element, `O(nlogn)` .
- Quickselect – `O(n)` expected, `O(n^2)` worst – like quicksort with the partition method, but only recurses into one of the parts (where `k` is).
- Median of medians select, `O(n)` :
    - Based on quickselect.
    - Finds an approximate median in linear time – this is the *key* step – which is then used as a pivot in quickselect.
    - It uses an (asymptotically) optimal approximate median-selection algorithm to build an (asymptotically) optimal general selection algorithm.
    - Can also be used as pivot strategy in with quicksort, yielding an optimal algorithm, with worst-case complexity `O(nlogn)` .
    - In practice, this algorithm is typically outperformed by instead choosing random pivots, which has average linear time for selection and average log-linear time for sorting, and avoids the overhead of computing the pivot.
    - The algorithm divides the array to groups of size 5 (the last group can be of any size <= 5) and then calculates the median of each group by sorting and selecting the middle element.

    - It then finds the median of these medians by recursively calling itself, and selects the median of medians as the pivot for partition.

Graph Algorithms

BFS Graph Traversal

- Breadth-first search usually serves to find shortest-path distances from a given source in terms of *number of edges* (not weight).
- Start exploring the graph from the given root (can be any vertex) and slowly progress out, while processing the oldest-encountered vertices first.
- We assign a direction to each edge, from the discoverer `(u)` to the discovered `(v)` (`u` is the parent of `v`).
- This defines (results in) a tree of the vertices starting with the root (breadth-first tree).
- The tree defines the shortest-path from the root to every other node in the tree, making this technique useful in shortest-path problems.
- Once a vertex is discovered, it is placed in a queue. Since we process these vertices in first-in, first-out order, the oldest vertices are expanded first, which are exactly those closest to the root.
- During the traversal, each vertex discovered has a parent that led to it. Because vertices are discovered in order of increasing distance from the root, the unique path from the root to each node uses the smallest number of edges on any root to node path in the graph.
- This path has to be constructed backwards, from the destination node to the root.
- BFS can have any node as the root – depends on what paths we want to find.
- BFS returns the shortest-path in terms of *number of edges* (for both directed and undirected graphs), not in terms of weight (see shortest-paths below).
- Traversal is `O(n + m)` where `n` = num of vertex and `m` = total num of edges.
- Applications:
    - Find shortest-path in terms of number of edges
    - Garbage collection scanning
    - Connected components
        - "Connected graph" = there is a path between *any* two vertices.
        - "Connected component" = set of vertices such that there is a path between every pair of vertices; there must not be a connection between two components (otherwise they would be the same component).
        - Many problems can be reduced to finding or counting connected components (like solving a Rubik's cube – is the graph of legal configurations connected?).
        - A connected component can be found with BFS – every node found in the tree

is in the same connected component. Repeat the search from any undiscovered vertex to define the next component, until all vertices have been found.

- Vertex-colorings (testing a graph for bipartite-ness)
  - Assigning colors to vertices (as few colors as possible), such that no edge connects two vertices with the same color.
  - Such problems arise in scheduling applications (like register allocation in compilers).
  - A graph is "bipartite" if it can be colored without conflicts with only two colors.
  - Such graphs arise in many applications, like: had-sex-with graph for heterosexuals (men only have sex with women), that is male vertices are only connected to female vertices, and vice-versa.
  - The problem – find the separation between the vertices for the two colors.
  - Such a graph can be validated with BFS: start with the root, and keep coloring each new vertex the opposite color of it's parent. Then make sure that no non-discovered edge links two vertices of the same color. If no conflicts are found – problem solved. Otherwise, it isn't a bipartite graph.
  - BFS allows us to separate vertices into two groups after running this algorithm, which we can't do just from the structure of the graph.

DFS Graph Traversal

- Depth-first search is often a subroutine in another algorithm.
- Starts exploring the graph from the root, but immediately expanding as far as possible (in contrast to BFS: breadth vs depth). Retraction back is only done when all neighboring vertices have already been processed.
- While BFS relied on a queue (FIFO) for new discovered vertices to be processed, DFS relies on stack (LIFO) (not really uses a stack but recursion, which functions as a stack).
- DFS classifies all edges into 2 categories: tree edges and back edges. Tree edges progress you down the tree to next vertices. Back edges take you back into some earlier part on the tree, to some ancestor.
- Start with the root, find the first child, process it, and run DFS on it recursively.
- After processing all reachable vertices, if any undiscovered vertices remain, depth-first search selects one of them as a new root and repeats from that root. The algorithm is repeated until it has discovered every vertex.
- Each vertex is timestamped (by incremented int): once when it's discovered and once when a vertex's edges have been examined.
- This defines (results in) a depth-first forest, comprising of several depth-first trees.
- Traversal is `O(n + m)` where `n` = num of vertices and `m` = total num of edges.
- Applications:

- Topological sort – create a linear ordering of a DAG (directed acyclic graph), such that for edge `(u, v)` then `u` appears before `v` in the ordering; computed by DFS and adding each "visited" vertex into the *front* of a linked list.
- Finding cycles – any back edge means that we've found a cycle. If we only have tree edges then there are no cycles (to detect a back edge, when you process the edge `(x,y)` check if `parent[x] != y`).
- Finding articulation vertices (articulation or cut-node is a vertex that removing it disconnects a connected component, causing loss of connectivity). Finding articulation vertices by brute-force is `O(n(n + m))`.
- Strongly connected components – in a SCC of a directed graph, every pair of vertices `u` and `v` are reachable from each other.

Minimum Spanning Tree

- Convert a weighted graph to a tree reaching all nodes (*spanning*), while having the minimal weight (*minimum*).

Shortest-Paths

- Find the path with minimal edge *weights* between a source vertex and every other vertex in the graph.
- Similarly, BFS finds a shortest-path for an *unweighted* graph in terms of number of edges.
- Sub-paths of shortest-paths are also shortest-paths.
- Graphs with negative-weight cycles don't have a shortest-paths solution.
- Some algorithms assume no negative weights.
- Shortest-paths have no cycles.
- We will store the path itself in `vertex.parent` attributes, similar to BFS trees; the result of a shortest-paths algorithm is a *shortest-paths tree*: a rooted tree containing a shortest-path from the source to every vertex that is reachable.
- There may be more than one shortest-path between two vertices and more than one shortest-paths tree for a source vertex.
- Each vertex `v` maintains an attribute `v.distance` which is an upper bound on the weight of a shortest-path (*"shortest-path estimate"*) from source `source` to `v`; `v.distance` is initialized to `Integer.MAX_VALUE` and `source.distance` is initialized to `0`.
- The action of *relaxing* an edge `(from, to)` tests whether we can improve the shortest-path to `to` found so far by going through `from`, and if so updating `to.parent` and `to.distance` (and the priority queue).

Dijkstra's Algorithm

- Algorithm for finding the shortest-paths between nodes in a graph, which may represent, for example, road networks.
- Assumes that edge weight is nonnegative.
- Fixes a single node as the *source* node and finds shortest-paths from the source to all other nodes in the graph, producing a shortest-path tree.
- The algorithm uses a min-priority queue for vertices based on their `.distance` value (shortest-path estimate).
- It also maintains a set of vertices whose final shortest-path weights from the source have already been determined.

A*

- An extension of Dijkstra which achieves better time performance by using heuristics.

Backtracking

- General algorithm for finding all (or some) solutions to a problem, that incrementally builds candidates to the solution, and abandons ("backtracks") each partial candidate as soon as it determines that it cannot possibly be completed to a valid solution.
- The classic textbook example of the use of backtracking is the eight queens puzzle; another one is the knapsack problem.
- Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution.
- When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

Lavenshtein Distance

- A string metric for measuring the difference between two sequences.
- The Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other.

Algorithms Code Examples

All examples can use the utility function `swap`:

```
private void swap(int[] a, int i, int j) {
    int tmp = a[i];
    a[i] = a[j];
    a[j] = tmp;
}
```

Tree Traversal

In-Order

```
public void inOrder(Tree tree) {
    if (tree == null) return;

    inOrder(tree.left);
    // process tree.value
    inOrder(tree.right);
}
```

Without recursion:

```
public void inOrder(Tree tree) {
    Stack<Tree> stack = new Stack<>();
    Tree curr = tree;

    while (curr != null || !stack.isEmpty()) {
        while (curr != null) {
            stack.push(curr);
            curr = curr.left;
        }

        curr = stack.pop();
        // process curr.value
        curr = curr.right;
    }
}
```

Pre-Order

```
public void preOrder(Tree tree) {
    if (tree == null) return;

    // process tree.value
    preOrder(tree.left);
    preOrder(tree.right);
}
```

Without recursion:

```
public void preOrder2(Tree tree) {
    Stack<Tree> stack = new Stack<>();
    stack.push(tree);
    Tree curr;

    while (!stack.isEmpty()) {
        curr = stack.pop();

        // process curr.value
        if (curr.right != null) stack.push(curr.right);
        if (curr.left != null) stack.push(curr.left);
    }
}
```

Post-Order

```
public void postOrder(Tree tree) {
    if (tree == null) return;

    postOrder(tree.left);
    postOrder(tree.right);
    // process tree.value
}
```

Without recursion:

```
public void postOrder(Tree tree) {
    Stack<Tree> tmp = new Stack<>();
    Stack<Tree> all = new Stack<>();
    tmp.push(tree);
    Tree curr;

    while (!tmp.isEmpty()) {
        curr = tmp.pop();
        all.push(curr);
        if (curr.left != null) tmp.push(curr.left);
        if (curr.right != null) tmp.push(curr.right);
    }

    while (!all.isEmpty()) {
        curr = all.pop();
        // process curr.value
    }
}
```

Sorting

Insertion Sort

```
public void sort(int[] a) {
    for (int j = 1; j < a.length; j++) {
        int key = a[j];
        int i = j - 1;

        while (i >= 0 && a[i] > key) {
            a[i + 1] = a[i];
            i--;
        }

        a[i + 1] = key;
    }
}
```

Selection Sort

```
public void sort(int[] a) {
    for (int i = 0; i < a.length; i++)
        swap(a, i, min(a, i));
}

private int min(int[] a, int start) {
    int smallest = start;

    for (int i = start + 1; i < a.length; i++)
        if (a[i] < a[smallest])
            smallest = i;

    return smallest;
}
```

Mergesort

```java
public int[] sort(int[] a) {
    if (a.length <= 1) return a;
    return sort(a, 0, a.length - 1);
}

private int[] sort(int[] a, int low, int high) {
    if (low == high)
        return new int[]{a[low]};

    int mid = (low + high) / 2;
    int[] sorted1 = sort(a, low, mid);
    int[] sorted2 = sort(a, mid + 1, high);

    return merge(sorted1, sorted2);
}

private int[] merge(int[] a, int[] b) {
    int[] res = new int[a.length + b.length];
    int i = 0, j = 0;

    while (i < a.length && j < b.length) {
        if (a[i] < b[j])
            res[i + j] = a[i++];
        else
            res[i + j] = b[j++];
    }

    while (i < a.length)
        res[i + j] = a[i++];

    while (j < b.length)
        res[i + j] = b[j++];

    return res;
}
```

Quicksort

```
public void sort(int[] a) {
    sort(a, 0, a.length - 1);
}

private void sort(int[] a, int low, int high) {
    if (low >= high) return;
    int pivot = partition(a, low, high);
    sort(a, low, pivot - 1);
    sort(a, pivot + 1, high);
}

private int partition(int[] a, int low, int high) {
    int pivot = low;
    int rand = new Random().nextInt(high - low + 1) + low;
    swap(a, low, rand);

    for (int i = low + 1; i <= high; i++) {
        if (a[i] < a[pivot]) {
            swap(a, i, pivot + 1);
            swap(a, pivot, pivot + 1);
            pivot++;
        }
    }

    return pivot;
}
```

Heapsort

```
public void sort(int[] a) {
    // build a max-heap
    for (int i = a.length - 1; i >= 0; i--)
        heapify(a, i, a.length);

    // extract max element from the head to the end and shrink the size of the heap
    for (int last = a.length - 1; last >= 0; last--) {
        swap(a, 0, last);
        heapify(a, 0, last);
    }
}

// heapify for a max-heap:
private void heapify(int[] a, int root, int length) {
    int left = 2 * root + 1;
    int right = 2 * root + 2;
    int largest = root;

    if (left < length && a[largest] < a[left])
        largest = left;

    if (right < length && a[largest] < a[right])
        largest = right;

    if (largest != root) {
        swap(a, root, largest);
        heapify(a, largest, length);
    }
}
```

Counting Sort

```
public int[] sort(int[] a) {
    int max = findMax(a);
    int[] sorted = new int[a.length];
    int[] counts = new int[max + 1];

    for (int i = 0; i < a.length; i++)
        counts[a[i]]++;

    for (int i = 1; i < counts.length; i++)
        counts[i] += counts[i - 1];

    for (int i = 0; i < a.length; i++) {
        sorted[counts[a[i]] - 1] = a[i];
        counts[a[i]]--;
    }

    return sorted;
}

private int findMax(int[] a) {
    if (a.length == 0) return 0;

    int max = Integer.MIN_VALUE;
    for (int i = 0; i < a.length; i++) {
        if (a[i] > max)
            max = a[i];
    }
    return max;
}
```

Searching

Binary Search

```
public int search(int[] a, int x) {
    return search(a, x, 0, a.length - 1);
}

private int search(int[] a, int x, int low, int high) {
    if (low > high) return -1;
    int mid = (low + high) / 2;

    if (a[mid] == x) return mid;
    else if (a[mid] > x) return search(a, x, low, mid - 1);
    else return search(a, x, mid + 1, high);
}
```

Selection

Quickselect

```
public int quickselect(int[] a, int k) {
    return quickselect(a, k, 0, a.length - 1);
}

private int quickselect(int[] a, int k, int low, int high) {
    int pivot = partition(a, low, high);
    if (pivot == k) return a[pivot];
    else if (k < pivot) return quickselect(a, k, low, pivot - 1);
    else return quickselect(a, k, pivot + 1, high);
}

private int partition(int[] a, int low, int high) {
    int pivot = low;
    int rand = new Random().nextInt(high - low + 1) + low;
    swap(a, low, rand);

    for (int i = low + 1; i <= high; i++) {
        if (a[i] < a[pivot]) {
            swap(a, i, pivot + 1);
            swap(a, pivot, pivot + 1);
            pivot++;
        }
    }

    return pivot;
}
```

Graph Algorithms

Utility function:

```
private void reset(Graph graph) {
    for (Vertex v : graph.vertices.values()) {
        v.parent = null;
        v.discovered = false;
        v.distance = Integer.MAX_VALUE;
    }
}
```

BFS

```
public void bfs(Graph graph, Vertex source) {
    reset(graph);
    Queue<Vertex> q = new LinkedList<>();
    q.add(source);
    source.discovered = true;

    while (!q.isEmpty()) {
        Vertex from = q.remove();

        for (Edge e : from.edges) {
            Vertex to = graph.vertices.get(e.to);

            if (!to.discovered) {
                to.parent = from;
                to.discovered = true;
                q.add(to);
            }
        }
    }
}
```

DFS

```
public void dfs(Graph graph) {
    reset(graph);

    for (Vertex v : graph.vertices.values()) {
        if (!v.discovered)
            dfs(graph, v);
    }
}

private void dfs(Graph graph, Vertex v) {
    v.discovered = true;
    // TODO: insert application of DFS here

    for (Edge e : v.edges) {
        Vertex to = graph.vertices.get(e.to);

        if (to.discovered) {
            // cycle found (back edge)! What should we do? depends on the application.
..
        } else {
            to.parent = v;
            dfs(graph, to);
        }
    }

    // TODO: insert application of DFS here. For example: if we're doing topological s
orting
    //       then add v to head of a linked list at this point
}
```

Dijkstra

```
public void dijkstra(Graph graph, Vertex source) {
    reset(graph);
    source.distance = 0;
    PriorityQueue<Vertex> q = new PriorityQueue<>(graph.vertices.values());

    while (!q.isEmpty()) {
        Vertex from = q.remove();

        for (Edge edge : from.edges) {
            Vertex to = graph.vertices.get(edge.to);
            int newDistance = from.distance + edge.weight;

            if (newDistance < to.distance) {
                to.distance = newDistance;
                to.parent = from;
                q.remove(to);
                q.add(to);
            }
        }
    }
}
```

Bit Operators

- Leading zeros `0` (i.e., to the left) are meaningless

`~` NOT

- Invert a bit
- Examples:
  - `~0 = 1`
  - `~1 = 0`
  - `~0111 = 1000`
  - `~100 = 011`
- In Java, `~` inverts an `int` and not single bits, so:

```
int b = 0b10;
~b == 11111111111111111111111111111101
```

`&` AND

- Results in `1` in each position if the corresponding first bit *and* second bit are `1`, otherwise `0`.
- Enables to find if a certain bit in a number contains `1` or `0`. Can be considered like multiplying all bits.
- Examples:
  - `10 & 11 = 10`
  - `0011 & 0010 = 0010`

`|` OR

- Results in `1` in each position if the corresponding first bit *or* second bit are `1`, otherwise `0`.
- Enables to set a specific bit to `1`.
- Examples:
  - `10 | 11 = 11`
  - `0011 | 0010 = 0011`

`^` XOR

- Exclusive OR – results in `1` in each position if the corresponding first bit *or* second bit are `1`, but *not* both, otherwise `0`.
- Enables to compare two bits – `1` means they are different, `0` means they are the same.
- Can be used to invert selected bits in a register. Any bit can be toggled by XOR-ing it with `1`.
- XOR-ing a value against itself yields zero.
- Examples:
  - `0101 ^ 0011 = 0110`
  - `0010 ^ 1010 = 1000`
- XOR can be used for "backup":
  - Calculate `a` and `b`'s XOR: `x = a^b`
  - If needed, recover `a`: `a = b^x`
  - If needed, recover `b`: `b = a^x`

`<<` Shift Left

- Add `n` `0` bits to the right.
- A left arithmetic shift by `n` is equivalent to multiplying the number by `2^n`.
- For example: `10111 << 1 = 101110`

`>>` Shift Right

- Remove `n` bits from the right (`0` or `1`).
- A right arithmetic shift by `n` is equivalent to dividing by `2^n`.
- For example: `10010111 >> 1 = 1001011`

See also Java section on Bit Arithmetics/Operations and Data Structures code examples for BitSet.

Binary Numbers

```
0 = 0
1 = 1
10 = 2
11 = 3
100 = 4
101 = 5
110 = 6
111 = 7
1000 = 8
1001 = 9
1010 = 10
1011 = 11
1100 = 12
1101 = 13
1110 = 14
1111 = 15
10000 = 16
...
```

```
10010110
^      ^
|      |------- bit 0
|
|-------------- bit 7
```

Having a `1` in the `k`-th bit, means that the decimal number is comprised of `2^k`. For example, for the above number:

```
2^7 + 2^4 + 2^2 + 2^1 = 150
```

Numbers

- https://gist.github.com/jboner/2841832
- http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Latency

Memory

Operation	Time	Note
L1 ref	`1ns`	
L2 ref	`4ns`	
RAM ref	`100ns`	
SSD ref	`16us`	"Random read", eq. to HDD disk seek
HDD ref	`4ms`	Disk seek

Read 1MB Sequentially

From	Time
RAM	`20us`
SSD	`300us`
HDD	`2ms`

CPU

Operation	Time
Mutex lock	`17ns`
Compress 1KB	`2us`
Context switch	`5us`

Network

Operation	Time
Send 2KB over network	`1us`
Round-trip in DC	`500us`
Round-trip CA->AMS	`150ms`

Data Speeds

Operation	Speed
HDD read/write	`150 MB/s`
SSD read/write	`500 MB/s`
USB 2.0 HDD	`30 MB/s`
USB 3.0 SSD	`300 MB/s`

Time

Period	Seconds
1 hour	`3.6K`
1 day	`86K`
1 year	`30M`

Powers of 2

Power	Size	Kibi
2^3	8	
2^4	16	
2^5	32	
2^6	64	
2^7	128	
2^8	256	
2^9	512	
2^10	1024	1 `KiB`
2^16	65,536	
2^20	1,048,576	1 `MiB`
2^30	1,073,741,824	1 `GiB`
2^32	4,294,967,296	
2^40	1.09 * 10^12	1 `TiB`
2^64	1.84 * 10^19	

Operating Systems

Memory

- `Byte` is the smallest addressable unit of memory.
- The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block (also called *stack frame*) is reserved on the top of the stack for local variables and some bookkeeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO order; the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.
- The heap is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time.
- Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).
- The stack is attached to a thread, so when the thread exits the stack is reclaimed. The heap is typically allocated at application startup by the runtime, and is reclaimed when the application (technically process) exits.
- The size of the stack is set when a thread is created. The size of the heap is set on application startup, but can grow as space is needed (the allocator requests more memory from the operating system).
- The stack is faster because the access pattern makes it trivial to allocate and deallocate memory from it (a pointer/integer is simply incremented or decremented), while the heap has much more complex bookkeeping involved in an allocation or memory freeing. Also, each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast.
- *Virtual memory* technique virtualizes the main storage available to a process or task, as a contiguous address space which is unique to each running process, or virtualizes the main storage available to all processes or tasks on the system as a contiguous global address space.
- Virtual memory illustrated
- `x64` is 64 bit platform while `x86` is 32bit platform. Pointers are 64 bit vs 32 bit.
- 32 bit systems are limited to 4GB memory.

Mutexes, Semaphores, Locks

- *Lock* is a generic term for an object that works like a key and allows the access to a target object only by one thread. Only the thread which owns the key can unlock the "door".
- *Monitor* is a *cross-thread* lock.
  - In Java every object is a monitor.
- Lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes.
- Mutex and Semaphore are kernel resources that provide synchronization services.

Mutex

- Mutex is a *cross-process* lock, same as a lock but system wide.
- Mutex is a locking mechanism used to synchronize access to a resource.
- Mutex provides *mut*ual *ex*clusion, either producer or consumer can have the key (mutex) and proceed with their work.
- Java has no system-wide mutex support.

Semaphore

- Semaphore is signaling mechanism ("I am done, you can carry on" kind of signal).
- A particular lock that allows more than one thread to access the target object. It's like having more keys, so that many people can unlock the door.
- Semaphore is a **generalized mutex**; if upper bound is set to one (1) it's the same as a monitor.
- Does the same as a lock but allows `x` number of threads to enter.

Processes

- A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
- Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as *pipes* and *sockets*. IPC is used not just for communication between processes on the same system, but processes on different systems.
- Each process has it's own memory space, and addresses memory inside that space.
- Each process can contain many threads.
- Threads share the memory space of its parent process.

- Process resources:
  - Working directory
  - Environment variables
  - File handles
  - Sockets
  - Memory space

IPC (Inter-Process Communication)

- Sharing data across multiple and commonly specialized processes.
- IPC methods:
  - File – record stored on disk, or a record synthesized on demand by a file server, which can be accessed by multiple processes.
  - Signal – system message sent from one process to another, not usually used to transfer data but instead used to remotely command the partnered process.
  - Socket – data stream sent over a network interface, either to a different process on the same computer or to another computer on the network.
  - Pipe – two-way data stream between two processes interfaced through standard input and output and read in one character at a time.
  - Named pipe – pipe implemented through a file on the file system instead of standard input and output.
  - Semaphore – A simple structure that synchronizes multiple processes acting on shared resources.
  - Shared memory – Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other.

Process Signaling

- Signals are a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems.
- A signal is an asynchronous notification sent to a process.
- If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed.
- Some signals may be ignored by an application (like `SIGTERM`).
- The kernel can generate signals to notify processes of events.
- For example, `SIGPIPE` will be generated when a process writes to a pipe which has been closed by the reader.
- There are two signals which cannot be intercepted and handled: `SIGKILL` (terminate immediately, no cleanup) and `SIGSTOP`.

- Pressing `Ctrl+C` in the terminal sends `SIGINT` (interrupt) to the process; by default, this causes the process to terminate.
- `$ kill -9` will send a `SIGKILL`.
- Notable signals:
  - `SIGINT` (2) – Terminal interrupt
  - `SIGQUIT` (3) – Terminal quit signal
  - `SIGKILL` (9) – Kill (cannot be caught or ignored)
  - `SIGTERM` (15) – Termination
  - `SIGSTOP` (na) – Stop executing (cannot be caught or ignored)
- In the JVM we can register for `SIGTERM` / `SIGINT` by:
  `Runtime.getRuntime().addShutdownHook(new Thread(...));`

Threads

- Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.
- Threads exist within a process – every process has at least one. Threads share the process' resources, including memory and open files. This makes for efficient, but potentially problematic, communication.
- Thread creation is a costly operation, and resources need to be constructed and obtained for a new thread.
- Creating and tearing down threads isn't free: there'll be some CPU overhead each time we do so.
- There may be some moderate limit on the number of threads that can be created, determined by the resources that a thread needs to have allocated (if a process has 2GB of address space, and each thread has 512KB of stack, that means a maximum of a few thousands threads per process).
- Solution: use a threadpool.

Thread Scheduler

- Responsible for sharing out the available CPUs in some way among the competing threads.
- On multiprocessor systems, there is generally some kind of scheduler per processor, which then need to be coordinated in some way.
- Most systems use priority-based round-robin scheduling to some extent:
  - A thread of higher priority (which is a function of base and local priorities) will preempt a thread of lower priority.

    - Otherwise, threads of equal priority will essentially take turns at getting an allocated slice or quantum of CPU.
- A thread can be in states:
    - `New` – created and waiting to create needed resources
    - `Runnable` – waiting for CPU allotment
    - `Waiting` – cannot continue, waiting for a resource like lock/IO/memory to be paged/sleep to finish/etc.
    - `Terminated` – finished by waiting to clear resources
- Each thread has a quantum, which is effectively how long it is allowed to keep hold of the CPU.
- Thread quanta are generally defined in terms of some number of clock ticks.
- A clock tick is typically 1ms under Linux.
- A quantum is usually a small number of clock ticks, between 10-200 clock ticks (i.e. 10-200 ms) under Linux.
- At key moments, the thread scheduler considers whether to switch the thread that is currently running on a CPU. These key moments are usually: periodically, if a thread ceases to be runnable, when some other attribute of the thread changes.
- At these decision points, the scheduler's job is essentially to decide, of all the runnable threads, which are the most appropriate to actually be running on the available CPUs.
- Priorities differ between OS, and can be partially set by the user.
- Great article on CPU Clocks and Clock Interrupts, and Their Effects on Schedulers.

Context Switching

- This is the procedure that takes place when the system switches between threads running on the available CPUs.
- Every time the thread running on a CPU actually changes – often referred to as a context switch – there'll be some negative impact due to, e.g., the interruption of the instruction pipeline or the fact that the processor cache may no longer be relevant.
- Switching between threads of different processes will carry a higher cost, since the address-to-memory mappings must be changed, and the contents of the cache almost certainly will be irrelevant to the next process.
- Context switches appear to typically have a cost somewhere between 1 and 10 microseconds.
- A modest number of fast-case switches — e.g. a thousand per second per CPU will generally be much less than 1% of CPU usage for the context switch per se.
- A few slower-case switches in a second, but where each switched-in thread can do, say, a milliseconds or so of worth of real work (and ideally several milliseconds) once switched in, where the more memory addresses the thread accesses (or the more cache lines it hits), the more milliseconds we want it to run uninterrupted for.

- So the worst case is generally where we have several "juggling" threads which each time they are switched in only do a tiny amount of work (but do some work, thus hitting memory and contending with one another for resources) before context switching.
- The cause of excessive context switching comes from contention on shared resources, particularly synchronized locks:
  - Rarely, a single object very frequently synchronized on could become a bottleneck.
  - More frequently, a complex application has several different objects that are each synchronized on with moderate frequency, but overall, threads find it difficult to make progress because they keep hitting different contended locks at regular intervals.
- Solutions include holding on to locks for less time and (as part of this), reducing the "housekeeping" involved in managing a lock.

Deadlocks, Livelocks, Starvation

- **Deadlock** describes a situation where two or more threads are blocked forever, waiting for each other.
- Deadlock is the phenomenon when, typically, two threads each hold an exclusive lock that the other thread needs in order to continue.
- In principle, there could actually be more threads and locks involved.
- Deadlock examples:
  - Locking on two different bank account with two threads, in an opposite order.
  - Two friends bowing to each other, and each one only gets up when the other one gets up.
- How to avoid – make locking order fixed (e.g: always lock on the smaller bank account first, sort by identity hash code, etc.).
- **Livelock** – A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked – they are simply too busy responding to each other to resume work.
- This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphonse moves to his right, while Gaston moves to his left. They're still blocking each other, so...
- Two thread can communicate with `wait();` and `notifyAll();` (see Java ITC)
- Livelock examples:
  - Two people passing in the corridor
  - Two threads trying to simultaneously avoid a deadlock

    - Husband and wife eating dinner with 1 spoon, and keep passing the spoon to the other spouse
- How to avoid live-locking – try to add some randomness.
- **Starvation** describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Race Conditions

- A race condition occurs when 2 or more threads are able to access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any point, you don't know the order at which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are 'racing' to access/change the data.
- Often problems occur when one thread does a "check-then-act" (e.g. "check" if the value is X, and then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act".
- In order to prevent race conditions occurring, typically you would put a lock around the shared data to ensure that only one thread can access the data at a time.

System Architecture

- Elements of Scale: Composing and Scaling Data Platforms
- Please stop calling databases CP or AP
- What we talk about when we talk about distributed systems
- A Comprehensive Guide to Building a Scalable Web App on Amazon Web Services
- Notes on Google's Site Reliability Engineering Book or the book itself

HDD

- Stores data on a series of constantly-spinning magnetic disks, called platters.
- Actuator arm with read/write heads positions the heads over the correct area of the drive to read/write (nanometers above platters).
- The drive may need to read from multiple locations in order to launch a program or load a file, which means it may have to wait for the platters to spin into the proper position multiple times before it can complete the command.
- If a drive is asleep or in a low-power state, it can take several seconds more for the disk to spin up to full power.
- Latency measured in milliseconds.
- SAS – Serial Attached SCSI – faster than HDD but still with a spinning platter.

SSD

- Don't rely on moving parts or spinning disks.
- Data is saved to a pool of NAND flash.
- NAND itself is made up of what are called floating gate transistors. Unlike the transistor designs used in DRAM, which must be refreshed multiple times per second, NAND flash is designed to retain its charge state even when not powered up.
- NAND flash is organized in a grid. The entire grid layout is referred to as a *block*, while the individual rows that make up the grid are called a *page*.
- Common *page* sizes are 2K, 4K, 8K, or 16K, with 128 to 256 pages per block.
- *Block* size therefore typically varies between 256KB and 4MB.
- Data read/written at the *page* level (individual rows within the grid).
- Data erased at the *block* level (requires a high amount of voltage).
- Latency measured in microseconds.
- SSD controllers have caches and a DDR3 memory pool to help with managing the NAND.

RAID

- Redundant Array of Independent Disks
- *Performance* increase is a function of *striping*: data is spread across multiple disks to allow reads and writes to use all the disks' IO queues simultaneously.
- *Redundancy* is gained by creating special stripes that contain parity information, which can be used to recreate any lost data in the event of a hardware failure.
- Hardware RAID – a dedicated hardware controller with a processor dedicated to RAID calculations and processing.
- RAID 0 – *Stripping* – splits blocks of data into as many pieces as disks are in the array, and writes each piece to a separate disk.
    - Single disk failure means all data lost.
    - Increased in throughput: throughput of one disk * number of disks.
    - Total space is sum of all disks.
    - Requires min. 2 drives.
- RAID 1 – *Mirroring* – duplicates data identically on all disks.
    - Data preserved until loss of last drive.
    - Reads faster or the same, writes slower.
    - Total space = size of smallest disk.
    - Requires min. 2 drives.
- RAID 10 – combination of RAID 1 and 0 (in that order). Create arrays of RAID 1, and then apply RAID 0 on top of them. Requires min. 4 disks, additional ones need to be added in pairs. A single disk can be lost in each RAID 1 pair. Guaranteed high speed and availability. Total space = 50% of disk capacity.
- RAID 5 – uses a simple XOR operation to calculate parity. Upon single drive failure, the information can be reconstructed from the remaining drives using the XOR operation on the known data. Any single drive of the drives we have can fail (total number of drives >= 2). In case of drive failure the rebuilding process is IO intensive.
- RAID 6 – like RAID 5, but two disks can fail and no data loss.

CAP Theorem

- Shared-data systems can have at most two of the three following properties: Consistency, Availability, and tolerance to network Partitions.
- Consistency
    - There must exist a total order on all operations such that each operation looks as if it were completed at a single instant.
    - A system is consistent if an update is applied to all relevant nodes at the same logical time.

    - Standard database replication is not strongly consistent because special logic must be introduced to handle replication lag.
    - Consistency which is both instantaneous and global is impossible.
    - Mitigate by trying to push the time resolutions at which the consistency breaks down to a point where we no longer notice it.
- Availability
    - Every request received by a *non-failing node* in the system must result in a response (that contains the results of the requested work).
    - Even when severe network failures occur, every request must terminate.
- Partition Tolerance
    - When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost.
    - Any distributed system can experience a network partition.
    - For a distributed system to *not* require partition-tolerance it would have to run on a network which is *guaranteed* to never drop messages (or even deliver them late) – **but such a network doesn't exist!**
- If one node is partitioned, then that node is *either* inconsistent or not available.
- If a system chooses to provide Consistency over Availability in the presence of partitions, it will preserve the guarantees of its atomic reads and writes by refusing to respond to some requests (like refuse writes on all nodes).
- If a system chooses to provide Availability over Consistency in the presence of partitions, it will respond to all requests, potentially returning stale reads and accepting conflicting writes (which can later be resolved with Vector Clocks, last-write wins, etc.).
- Most real-world systems require substantially less in the way of consistency guarantees than they do in the way of availability guarantees.
- Failures of consistency are usually tolerated or even expected, but just about every failure of availability means lost money.
- The choice of availability over consistency is a business choice, not a technical one.

MapReduce

- Programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.
- Map procedure performs filtering and sorting.
- Reduce procedure performs a summary operation.
- Data transferred between map and reduce servers is compressed. Because servers aren't CPU bound it makes sense to spend on data compression and decompression in order to save on bandwidth and I/O.
- Advantages:

- Nice way to partition tasks across lots of machines.
- Handle machine failure.
- Works across different application types. Almost every application has map reduce type operations. You can precompute useful data, find word counts, sort TBs of data, etc.
- Computation can automatically move closer to the IO source.
- Programs can be very small. As little as 20 to 50 lines of code.

For example, create search inverted index:

```
// Mapper

/*
 * Processes a key/value pair to generate a set of intermediate key/value pairs
 *
 * @param key id of document
 * @param value text of page
 * @returns pair of word and document id
 */
public Tuple<String, String> map(String key, String value) {
  for (String word : tokenize(value))
    return new Tuple<>(word, key);
}
```

```
// Reducer

/*
 * Merges all intermediate values associated with the same intermediate key
 *
 * @param key word
 * @param value list of document ids
 * @returns word to document ids pair
 */
public Tuple<String, List<String>> reduce(String key, List<String> value) {
  return new Tuple<>(key, value)
}
```

Distributed Hash Table

- A decentralized distributed system that provides a lookup service similar to a hash table.
- Any participating node can efficiently retrieve the value associated with a given key.
- Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption.

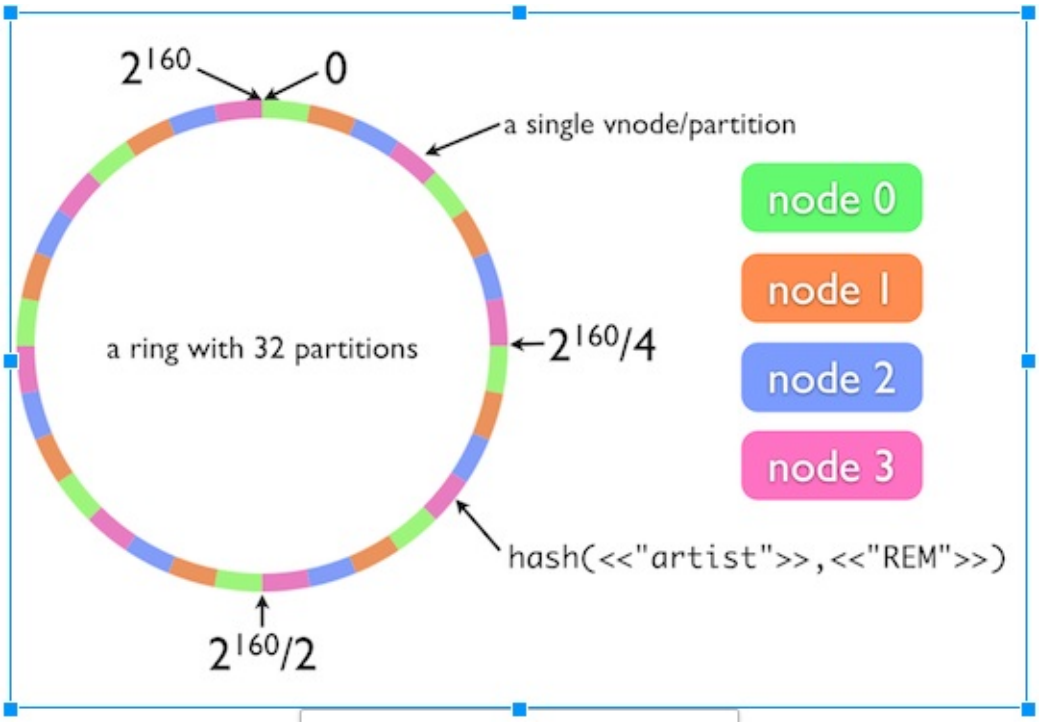
- Example implementations: Google's BigTable, Cassandra, Hazelcast, Aerospike, Riak.

Data Partitioning (Sharding)

- Split data between multiple nodes.
- Data should be well distributed throughout the set of nodes.
- When a node is added or removed from set of nodes the expected fraction of objects that must be moved to a new node is the minimum needed to maintain a balanced load across the nodes.
- System should be aware which node is responsible for a particular data.
- Simple approach to partitioning of `hashCode(key) % n` breaks when we want to add/remove nodes.

Consistent Hashing

- Special kind of hashing such that when a hash table is resized and consistent hashing is used, only `K/n` keys need to be remapped on average, where `K` is number of keys and `n` is number of buckets.
- In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped.
- Originated as a way of distributing requests among a changing population of Web servers.
- Consistent hashing maps objects to the same cache machine, as long as possible.
- When a cache machine is added, it takes its share of objects from all the other cache machines and when it is removed, its objects are shared between the remaining machines.
- Consistent hashing is based on mapping each object to a point on the edge of a circle (or equivalently, mapping each object to a real angle). The system maps each available machine (or other storage bucket) to many pseudo-randomly distributed points on the edge of the same circle.
- Apache Cassandra, for example, uses consistent hashing for data partitioning in the cluster.
- An alternative is *Rendezvous Hashing*.



(source)

Distributed System Consensus

- A fundamental problem in distributed computing is to achieve overall system reliability in the presence of a number of faulty processes.
- Often requires processes to agree on some data value that is needed during computation, e.g., whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, and atomic broadcasts.
- Example of algorithms:
  - Paxos
  - Raft

Resiliency Concepts (Advanced)

- Controlled Delay
  - Limit the size of a server's request queue without impacting reliability during normal operations.
  - Sets short timeouts, preventing long queues from building up.
  - If the queue has not been empty for the last `N` ms, then the amount of time spent in the queue is limited to `M` milliseconds. If the service has been able to empty the queue within the last `N` milliseconds, then the time spent in the queue is limited to `N` milliseconds.

    - Prevents a standing queue (because the `lastEmptyTime` will be in the distant past, causing an `M`-ms queuing timeout) while allowing short bursts of queuing for reliability purposes.
    - Values of `5ms` for `M` and `100ms` for `N` tend to work well across a wide set of use cases.
- Adaptive LIFO
    - During normal operating conditions, requests are processed in FIFO order, but when a queue is starting to form, the server switches to LIFO mode.
    - Places new requests at the front of the queue, maximizing the chance that they will meet the deadline set by Controlled Delay.
- Backup requests
    - Initiate a second ("backup") request to a different server `N` ms after the first request was made, if it still hasn't returned.
    - Dramatic improvement for high percentiles.
    - Example flow:
        1. Send request to first replica.
        2. Wait 2 ms, and send to second replica.
        3. Cancel request on other replica when starting to read second request's response.

System Design Question Concepts

- Amount of data (disk)
- RAID configuration
- Amount of RAM – does everything need to be in RAM? (a lot of times – yes)
- Requests/sec
- Request time
- Data transfer rates
- Network requests in single data center
- Geographically separated locations
- Operations should never be more than `nlogn`, preferably `n`
- Sharding/Partitioning (by user/date/content-type/alphabetically) – how do we rebalance?
- Frontend/backend
- Backups? Backup a single user on how many servers?
- Many requests for the same data/lots of updates – all in cache
- Distribute so that the network is negligible
- Parallelizing network requests
- Fan-out (scatter and gather)
- Caching

- Load-balancing
- High availability
- MapReduce
- CDN
- Queues
- Timeouts
- Fail fast
- Circuit breakers
- Throttling
- Consistent hashing for sharding/partitioning
- Compression
- Possible to work harder on writes in order to make reads easier

Platform Needs

- Monitoring (app, resp. time, throughput, OS level, errors)
- Alerting
- Metrics
- Distributed tracing + logging
- Datastores
- Caching
- Queues
- Service discovery
- Configuration
- Debugging, profiling
- ALM and deployment
- Auto scaling
- Load balancing, traffic splitting/control
- KPIs and actions based on them
- Deployment (blue-green), rollback, canary/testbed

Membership Protocols (Advanced)

Summary mainly based on this slide deck and the SWIM paper.

- Determining who are the live peers out of a dynamic group of members.
- Communicating over a network, assuming: packet loss, partitions.
- Useful for discovery in distributed systems.
- Protocols measured by:

    - Completeness (do all non-faulty members eventually detect a faulty node)
    - Speed of failure detection (time from failure until any node detects the failure)
    - Accuracy (rate of false positives)
    - Network load

Heartbeating

- Every interval `T`, notify peers of liveness.
- If no update received from peer `P` after `T * limit`, mark as dead.
- Provides membership + failure detection.
- Broadcast a liveness message on every interval.
- Can be implemented with IP broadcast/multicast (generally disabled) or gossip.
- Network load is `O(n^2)` messages on each interval – becomes a bottleneck for large clusters.

SWIM

- *S*calable *W*eakly-Consistent *I*nfection-Style Process Group *M*embership Protocol.
- Failure detection done by randomly pinging a single peer every interval and marking failed nodes.
- To improve accuracy, before marking a node as failed try indirect ping.
- Indirect ping is asking a set of `K` random peers to ping the failing candidate. This improves accuracy by attempting other network routes and reduce chance of packet loss.
- Network load is `O(n)`, making is feasible for large clusters.
- State updates on membership (joining/leaving the cluster) is achieved by piggybacking on the ping messages for gossip (infection/epidemic).
- Each ping/ack message sent between random nodes adds information about new and left/failed nodes.
- Because pings are sent randomly, nodes will learn about new/failing nodes at different times => eventually (weakly) consistent.
- Ping + piggybacking state updates = membership protocol.

System Architecture Examples

Spelling Suggestion Service

- Read a big file with lots of words.
- Normalize the file: remove anything that's not a-z and make it lower-case.
- Generate a count parameter for the frequency of each word in the English language, giving it a sort of score.
- When the spelling suggestion word comes in, process it to find the following variants of the word (suitable corrections):
  - Transposing letters
  - Adding letters
  - Removing letters
  - Replace existing letter
- Total variations for a word of length n is: ~54n+25 (for English alphabet with 26 letters).
- Decide if 1 edit is enough, or if you want to try 2/3/4 edits, etc. (1/2 is prob. enough).
- To add another edit just run the function that generates edits on the results of the first run.
- Now keep only variations that are known as words, from the words list we generated at the beginning.
- Factor in plural form, present/past tenses, etc.
- Sort by score, descending.

Search Engine

- Separate the system to different parts:
  - Web crawler – retrieve new documents, update content of existing ones
  - Document store – contain document content and all metadata (URL, date, etc.)
  - Indexer – create the index
  - Querying – serving search requests
  - Ranking – sorting the results

Indexing

- Define the corpus (document collection). How will we get it? Designing a web crawler is a whole topic on its own (but might still be in the scope of such a question).
- Likely want to ignore casing, grammatical tenses, "stop words" (most common words in

a language, e.g., the, is, at, which, on, etc.).
- Build an *Inverted Index* from parsing the documents. Given a query the index can return the list of documents relevant for it.
- Each vocabulary term is a key in the index whose value is its documents list.
- Inverted index will also contain additional information such as: overall number of occurrences (frequency), occurrence position in each document.
- Parse each document:
  - Lower case all input.
  - Extract tokens; Token = alphanumeric `[a-z0-9]` characters terminated by a non-alphanumeric character.
  - Filter out stop words.
  - Use stemming (Porter Stemmer for English, for example) to remove common endings from words (-s, -ing, -er, -ed, etc.).
- Inverted index will be a (distributed) hash table in memory: `Map<String, List<DocumentOccurrences>>` (see below).
- Index can be persisted to disk in a simple format, or saved to datastore.
- Rough rule of thumb: for a document collection of size X, the index size will be 0.66X.
- Distribution of terms in a collection generally follows Zipf's Law, which states that the frequency of a word is inversely proportional to its rank; i.e., most frequent word will appear 2x more than second most popular and 5x most then fifth most popular word.

```
class DocumentOccurrences {
    public String documentId;
    public int[] positions;
}
```

Querying

- Query types:
  - Single word queries
  - Free text queries – documents containing all keywords, regardless of order
  - Phrase queries – documents containing all keywords in exact query order
- Read the inverted index from disk or datastore into the same data structure.
- Document word transformations (grammar, stop words, stemming) will also be done on incoming querie as well (but maybe not for exact phrase queries).
- For single word queries: get the list of `documentId` s that contain that word from the index and return that list.
- For free text queries:
  - Tokenize the query.
  - Get the list of documents for each token.
  - Take the *union* of all results.

- For exact phrase queries:
  - Just like free text queries but take the *intersection* of all `documentId`s from all results (can use Java's `List.retainAll` or `Set.retainAll`).
  - Next, for each document, get positions for each search term, and place the positions in separate lists as the order or the search terms.
  - Subtract `i-1` from the elements (positions) of the `i`-th list, starting from the second list (not changing the first one).
  - Take the intersection of the lists (or better: sets). If it's not empty then that document is a match for the exact phrase. Perform the same for the rest of the documents.

Ranking

Ranking algorithms options:

- tf-idf (term frequency–inverse document frequency) – Reflect how important a word is to a document in a collection. Used as a weighing factor by assigning each term in a document a weight based on its term frequency (tf) and inverse document frequency (idf). Terms with higher weight scores are considered more important. tf-idf value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.
- Okapi BM25 – state-of-the-art variant of tf-idf.
- PageRank – PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.
- Apply a ranking algorithm for the search query against each of the returned documents, and then sort by rank.

Scaling

- A single word could appear in too many documents, and so maintaining a single key-value pair for that word is not fesible.
- Most likely need to shard documents based on URL (domain), and then query all shards (fan-out) for each keyword in the search query.

Networking

OSI Model

- *Open Systems Interconnection Model*

Layers

Layer	Data unit	Examples
1. Physical	bit	Ethernet, USB, Wi-Fi, Bluetooth, DSL
2. Data link	Frame	L2TP, PPP
3. Network	Packet	**IPv4**, **IPv6**
4. Transport	Segment (Datagram)	**TCP**, **UDP**
5. Session	Data	HTTP, FTP, SMTP, DNS, IMAP, SSH, TLS
6. Presentation	Data	ASCII, JPEG
7. Application	Data	Chrome, Mail.app

Data Units

- Frame
  - Structure: Frame header (e.g. Ethernet) + (Network header + Transport header + data) + Frame footer
  - Source and destination MAC addresses, length, checksum
- Packet
  - Structure: Network header (e.g. IP) + (Transport header + data)
  - Version (IPv4/IPv6), source and destination IP addresses, length flags, TTL, protocol, checksum
- Segment
  - Structure: Transport header (e.g. TCP) + data
  - Source and destination ports, sequence number, ack number, data offsets, flags, checksum

IP

- *Internet Protocol*

- No concept of connection.
- Packets are passed from one computer to the next, until reaching the destination.
- No delivery guarantee, no receiving ack.
- Sometimes multiple copies of the same packet are passes, taking different paths, and thus arriving at different times.
- Designed to be able to route around connectivity problems.

TCP

- *Transmission Control Protocol*
- Built on-top of IP.
- Connection-based.
- Once a connection has been made between two parties, sending data between the two is much like writing to a file on one side and reading from a file on the other.
- Reliable and ordered, i.e., arrival and ordering are guaranteed.
- Takes care of splitting your data info packets and sending those across the network, so you can write bytes as a stream of data.
- Makes sure it doesn't send data too fast for the Internet connection to handle (flow control).
- Hides all complexities of packets and unreliability.
- Sends an ack for every packet received.
- Queues up data until there's enough to send as a packet.
- TCP tends to induce packet loss for UDP packets whenever they share a bottleneck node (same LAN/WAN).

UDP

- *User Datagram Protocol*
- Built on-top of IP, very thin layer over it.
- Unreliable protocol, usually around 1-5% packet loss.
- No guarantee of ordering.
- Minimizes transmission delay.
- Send a packet to destination IP address and port; the packet will get passed from computer to computer and arrive at destination or get lost.
- Receiver simply listens on specific port and gets notified when a packet arrives, with the sender address:port, and packet size.
- One guarantee over IP – a packet will either arrive as a whole (all of it) at destination or not at all (no partial delivery).
- You need to manually break your data up into packets and send them.

- You need to make sure you don't send data too fast for your Internet connection to handle.
- Good for when you want data to get as quickly as possible from client to server without having to wait for lost data to be resent, usually real-time data.
- Examples: real-time gaming, metrics reporting, video/audio streaming.

Strings

Char Sets

- Character sets translate characters to numbers.

ASCII

- *American Standard Code for Information Interchange*
- Encodes 128 characters in 7 bits.
- Encoded are numbers 0 to 9, lowercase letters a to z, uppercase letters A to Z, basic punctuation symbols, control codes and space.
- *ANSI* standard – different "code pages" for characters 128-255 (the 1 extra bit) which differ between countries and languages.

Unicode

- Character set for most of the world's writing systems.
- List of characters with unique numbers (code points).
- There are more than 120,000 characters covering 129 "scripts" (a collection of letters), there's no limit on number of letters.
- Letters map to code points.
- Every letter in every alphabet is assigned a number, for example the letter `A` = `41` (`U+0041`); the number is *hexadecimal*.
- For example, the list of numbers represent the string "hello": `104` `101` `108` `108` `111` .
- There are more than 65,526 (2^16) chars, so not every Unicode letter can be represented by two Bytes.
- Unicode character in Java: `\u00fc`
  - `String s = "\u00fc";`

Encoding

- An encoding is a way to translate between Strings and Bytes.
- Encoding is how these numbers are translated into binary numbers to be stored on disk or in memory (Encoding translates numbers into binary).
- It doesn't make sense to have a string without knowing what encoding it uses!

UTF-8

- UTF-8 is a transmission format for Unicode, i.e., *encoding*.
- Capable of encoding all 1,112,064 possible characters (code points) in Unicode.
- Variable-length, code points are encoded with 8-bit code units.
- Every code point from 0-127 is stored in a *single Byte*.
- Code points 128 and above are stored using 2, 3, or 4 Bytes.
- English text looks exactly the same in UTF-8 as it did in ASCII.
- ASCII text is valid UTF-8-encoded Unicode.
- `byte[]` however has an encoding.
- To convert a string object to UTF-8, invoke the `getBytes(Charset charset)` on the string with UTF-8.
- 84.6% of all Web pages use UTF-8.
- Java `String` uses UTF-16 encoding internally.
- For example, UTF-8 encoding will store "hello" like this (binary): `01101000` `01100101` `01101100` `01101100` `01101111`

UTF-16

- Capable of encoding all 1,112,064 possible characters in Unicode.
- Variable-length, code points are encoded with one or two 16-bit code units.
- The `String` class in Java uses UTF-16 encoding internally and can't be modified.

Punycode

- A way to represent Unicode with the limited character subset of ASCII supported by DNS.
- For example: "bücher" => "bcher-kva"

Communicating Encoding

- Email:
  - `Content-Type: text/plain; charset="UTF-8"` header in the beginning of the message.
- Web page:
  - `<meta http-equiv="Content-Type" content="text/html; charset=utf-8">` meta tag, has to be the very first thing in the `<head>`.
  - As soon as the web browser sees this tag it's going to stop parsing the page and start over after reinterpreting the whole page using the encoding specified.
  - Can also use the `Content-Type` header like in email, but the `<meta>` tag is preferable.

Java

Ideally, you should have one mainstream programming language down very well, to use in interviews. Java is mine.

- Google Java Style Guide

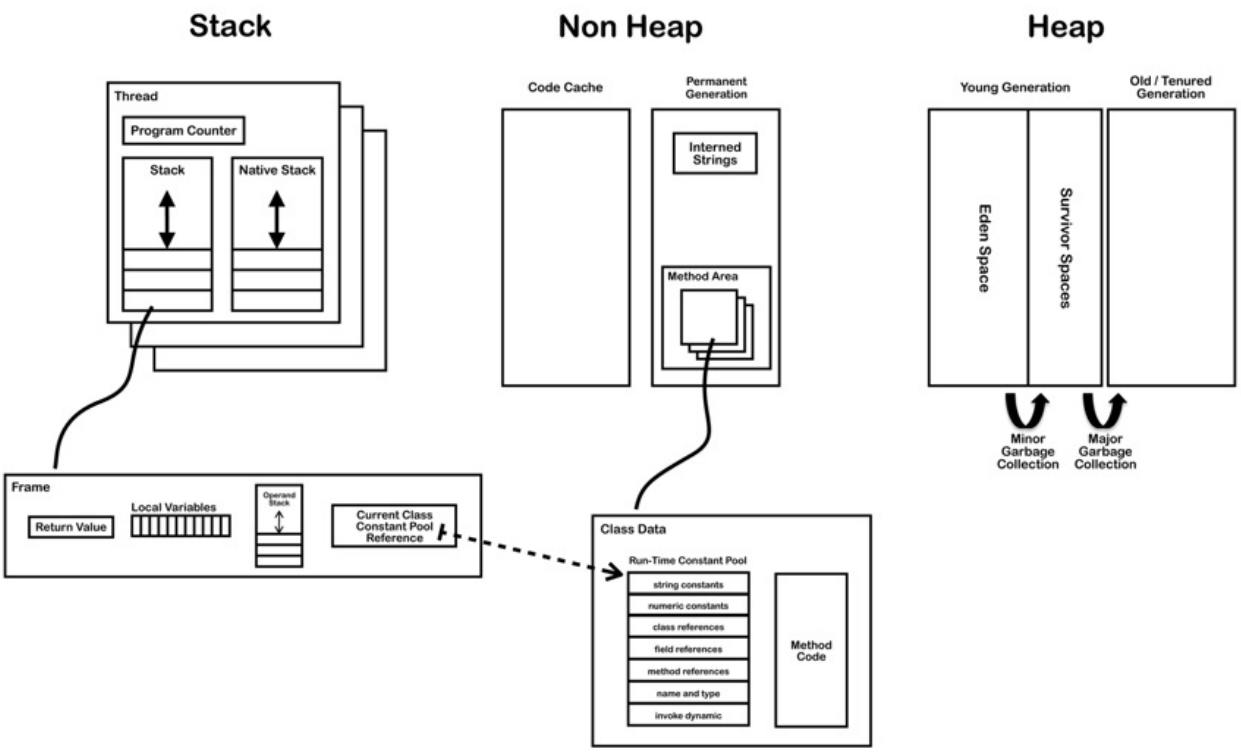
Threads

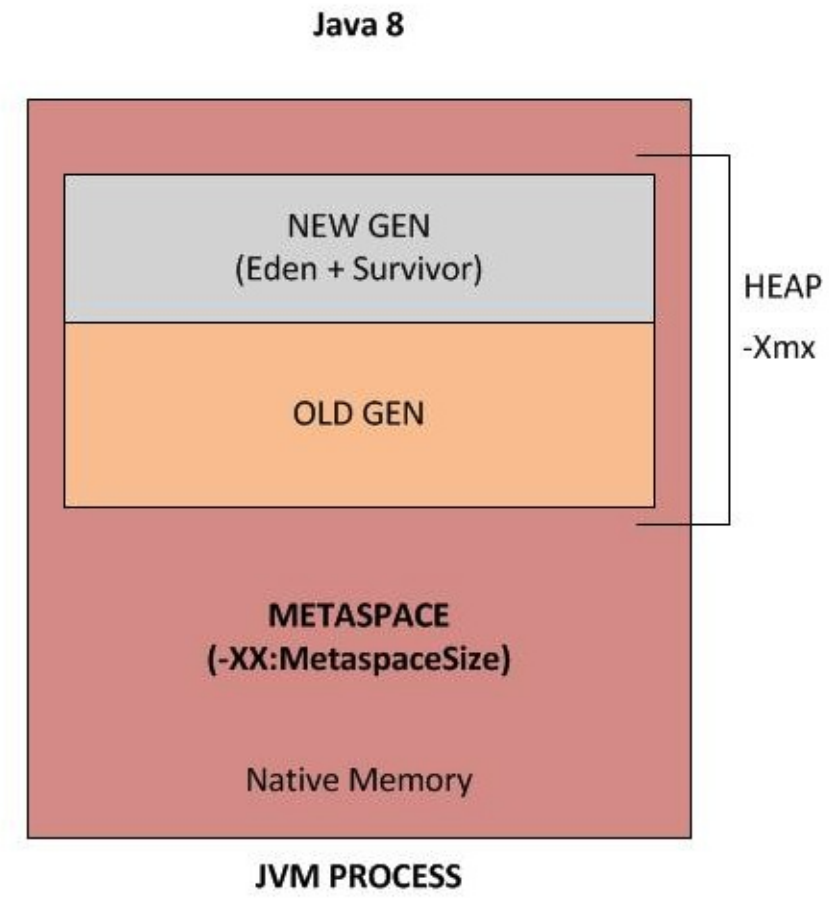
- An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.
- A thread sends an interrupt by invoking `interrupt()` on the `Thread` object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.
- The `join` method allows one thread to wait for the completion of another. If it is a Thread object that is currently executing, then `thread.join();` causes the current thread to pause execution until its thread terminates.
- When a JVM starts up, there is usually a single non-daemon thread (which typically calls the method named `main` of some designated class). The JVM continues to execute threads until either of the following occurs:
  - The exit method of class `Runtime` has been called and the security manager has permitted the exit operation to take place.
  - All threads that are not daemon threads (`thread.setDaemon(true);` wasn't called) have died, either by returning from the call to the `run` method or by throwing an exception that propagates beyond the `run` method.

Memory

Type	bits	Bytes
boolean	1 bit	1 Byte
byte	8 bit	1 Byte
short	16 bit	2 Byte
char	16 bit	2 Byte
int	32 bit	4 Byte
float	32 bit	4 Byte
long	64 bit	8 Byte
double	64 bit	8 Byte

- Memory reference (pointer): 4 Bytes.
- Object header: a few Bytes (~8 Bytes) for housekeeping info (GC, class, id, etc.).
- Array header: 12 Bytes (4 Bytes for array length).
- `BitSet` – a bit vector with growing size, each element is a boolean with 1-bit size (not 8-bit as regular booleans).
- Thread stacks are allocated *outside* the heap. The heap only stores objects.
- Thread stacks contain local variables (primitives, heap references) and calls to methods. Sometimes the JVM optimizes and allocates objects that never leave the stack scope on the stack (function-local objects).
- Arrays are like Objects, and are generally stored on the heap.
- Default stack size on 64-bit Linux is 256 KB for a *single thread*.
- Frame size for a simple method call is 32 Bytes.
- JVM Internals (read at least until *Class File Structure*.)





- In JDK8 8 PermGen is replaced with Metaspace, which is very similar. The main difference is that Metaspace can expand at runtime (less `java.lang.OutOfMemoryError: PermGen`).
- If you don't specify `MaxMetaspaceSize` the Metaspace will dynamically resize depending

on the application demand at runtime.
- Metaspace allocations are out of native memory.
- Metaspace garbage collection of dead classes and classloaders is triggered once the class metadata usage reaches the `MaxMetaspaceSize` option.
- Since JDK 6 interned strings are no longer allocated in the PermGen, but are instead allocated in the regular heap.

Garbage Collection

- Objects are created on the heap.
- Garbage collection is a mechanism provided by the JVM to reclaim heap space from objects eligible for GC.
- GC is done by the GC thread.
- Before removing an object from memory, GC thread invokes the `finalize()` method of the object, allowing for cleanup operations (but *don't* use finalizers – Effective Java 2nd Ed. Item 7).
- As a developer you can't force GC, it will only be triggered if the JVM thinks it needs to based on the Java heap size.
- You can *request* GC (by `Runtime.gc()` or `System.gc()`) but it's not *guaranteed* to happen.
- If the JVM heap is full, and a new object can't be created, an `OutOfMemoryError` exception is thrown.
- An object becomes eligible for GC if it's not reachable from any live thread of any static reference, meaning if all its references are null.
- Cyclic dependencies aren't references, and they will be GCed if no other objects have a reference to any of them.
- GC scenarios: all references to an object are set to null, object declared in a block and now out of scope, object is a member and the parent is eligible for GC, object in a `WeakHashMap`.
- Java memory leaks:
  - Can be created by some hacks involving `ThreadLocal`, see link
  - Static final fields
  - `myString.intern()` – places string in memory pool that can't be removed
  - Unclosed open streams (file, network, etc.) or connections
  - Native hooks that aren't accessible to GC
  - Misuse of JVM options (parameters)
  - `HashSet` / `Map` which uses incorrect `hashCode`, so elements are always added
- Available garbage collectors and how to select one.

Common JVM Options

- `-Xms1g` – Initial heap size
- `-Xmx2g` – Hax heap size
- `-XX:MaxMetaspaceSize=200m` – Add a limit to Metaspace (not limited by default)
- `-Xmn500m` – Initial *and* max young gen size
- `-XX:SurvivorRatio=4` – Ratio of survivor size relatively to eden size (ratio = young/survivor - 2)
- `-Xss256k` – Thread frame stack size
- More options recommendations and details

Bit Arithmetics/Operations

- Can use `java.util.BitSet` to work with single bits.
- Positive integers are stored as simple binary numbers:

```
int a = 0b101;
assertEquals(5, a); // true

int zero  = 0b0;
int one   = 0b1;
int two   = 0b10;
int three = 0b11;
int five  = 0b101;
```

- Negative integers are stored as the two's complement of their absolute value. The two's complement of a positive number is, when using this notation, a negative number.
- To find the negative of a number (`x`) in two's complement:
  - Invert all the bits (`~x`)
  - Add `1` bit (`x + 1`)

```
int x = 4;
int minusX = ~x + 1;
assertEquals(-4, minusX); // true

int minusOne   = 0b11111111111111111111111111111111;
int minusTwo   = 0b11111111111111111111111111111110;
int minusThree = 0b11111111111111111111111111111101;
int minusFour  = 0b11111111111111111111111111111100;
int minusFive  = 0b11111111111111111111111111111011;
```

- `>>`

    - Signed shift right.
    - Uses the sign bit (left most bit) to fill the trailing positions after the shift.
- `>>>`
    - Unsigned (logical) shift right.
    - Shift right while filling zeros `0`, irrespective of the sign of the number.

Primitives

- `int` casting drops any decimal, essentially rounding down.

Strings

- Backed by char array, can get it by calling `toCharArray()`.
- Strings are always immutable, and the class is `final`.
- Strings concatenation with the `+` operator translates to `StringBuilder` operations by the compiler, but creates a new instance of `StringBuilder` for every concatenation. Use `StringBuilder` directly for *repeated* concatenation in multiple statements with intermediate strings (single statements are fine, e.g., for `String s = a + b + c` use `StringBuilder`).

Interning

- String literals, i.e. `String s = "hello"`, are interned by the compiler.
- Interned strings are privately maintained in a pool, which is initially empty, by the class `String`.
- The `public String intern()` function on `String` places that instance in the pool and returns the canonical representation for that string object.
- When the `intern` method is invoked, if the pool already contains a string equal to this `String` object as determined by the `equals(Object)` method, then the string from the pool is returned.
- Otherwise, this `String` object is added to the pool and a reference to this `String` object is returned.
- All literal strings and string-valued constant expressions are interned.
- `==` tests for reference equality (whether they are the same object).
- `.equals()` tests for value equality (whether they are logically *equal*).
- `new String("test") == "test"` is `false`
- `new String("test") == new String("test")` is `false`
- `"test" == "test"` is `true`
- You almost *always* want to use `.equals()`. In the rare situation where you *know* you're

dealing with interned strings, you can use `==`.

Concurrency

ITC (Inter-Thread Communication)

- Java includes an inter-process communication mechanism via the `wait()`, `notify()`, and `notifyAll()` methods.
- These methods are implemented as final methods in `Object`, so all classes have them.
- All three methods can be called only from within a `synchronized` method.
- `wait()` tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`.
- Additional forms of `wait()` exist that allow you to specify a period of time to wait.
- `notify()` wakes up the first thread that called `wait()` on the same object.
- `notifyAll()` wakes up all the threads that called `wait()` on the same object. The highest priority thread will run first.
- You should probably not use `wait` / `notify` directly and use the `Lock` class and others from `java.util.concurrent.locks` instead.

Access Levels

- Having *no modifier* is also called package-private.
- When a class or interface is accessible to clients for each modifier:

Modifier	Class	Package	Subclass	World
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
`private`	Yes	No	No	No

Autoboxing-Unboxing

- *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes (e.g., converting an `int` to an `Integer`).
- If the conversion is the other way around (object to primitive) it's *unboxing*.
- The Java compiler applies autoboxing/unboxing when a primitive/object is:

    - Passed as a parameter to a method that expects an object/primitive of the corresponding wrapper/primitive type.
    - Assigned to a variable of the corresponding wrapper/primitive type.
- The compiler uses `valueOf` to autobox and `intValue` to unbox (for `int`, for example).
- Beware of unnecessary Object creation with autoboxing/unboxing.
- Object types don't support operators (`+` for example), so using it on `Integer` will trigger unboxing and then autoboxing.
- For arithmetics always prefer primitives.
- Equality checks with `==` don't unbox and so reference are compares, not values. So to compare two `Integer`s use `.equals()` or preferrable use primitive types.
- You can get `NullPointerException`s when unboxing.

Java Code Examples

Arrays

```
int[] a = new int[10];
int[] b = new int[]{34};
```

Threads

```
public static void main(String[] args) {
    Thread thread = new Thread(new MyRunnable());
    thread.start();
}

private static class MyRunnable implements Runnable {
    @Override
    public void run() {
        // ...
    }
}
```

Threadpool (`ExecutorService`)

```
import java.util.concurrent.*;

ExecutorService pool = Executors.newFixedThreadPool(4);
// or:
ExecutorService pool = new ThreadPoolExecutor(4, 4, 0L, TimeUnit.MILLISECONDS, new Lin
kedBlockingQueue<Runnable>());

pool.submit(new Runnable() {
    @Override
    public void run() {
      // ...
    }
});

// can also submit a Callable that returns a value (instead of Runnable's void):
Future<String> result = pool.submit(new Callable<String>() {
    @Override
    public String call() throws Exception {
        return null;
    }
});

result.get(); // blocking until the future completes
```

Number Parsing

```
// java.lang.NumberFormatException thrown for bad formatting:
int i = Integer.parseInt("42");
float f = Float.parseFloat("1.2");
long l = Long.parseLong("1000");
double d = Double.parseDouble("3.14");

// or with Objects instead of primitives (parsed the same):
Integer i = Integer.valueOf("42");
Float f = Float.valueOf("1.2");
Long l = Long.valueOf("1000");
Double d = Double.valueOf("3.14");
```

Random

```
Random rnd = new java.util.Random();
int i = rnd.nextInt(5); // exclusive of upper bound
```

Inheritance

```
interface Fruit {
  String name();
}

interface Serializable {
    String serialize();
}

interface MySerializable extends Serializable {}

class Orange implements Fruit, MySerializable {
    @Override
    public String name() {
        return null;
    }

    @Override
    public String serialize() {
        return null;
    }
}

// class and interface inheritance
class BaseFruit {
    // ...
}

class Apple extends BaseFruit implements Serializable {
    @Override
    public String serialize() {
        return null;
    }
}
```

Try/Catch/Finally

```
try {
    // ...
} catch (IllegalArgumentException | NullPointerException e) {
    // ...
} finally {
    // ...
}
```

Enums

```
public enum Color {
    Yellow,
    Green
}

public enum Size {
    Small(1),
    Medium(2),
    Large(2);

    private int size;

    private Size(int size) {
        this.size = size;
    }

    public int getSize() {
        return size;
    }
}
```

Regex

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

Pattern pattern = Pattern.compile("...");
Matcher matcher = pattern.matcher(input);

if (matcher.find()) {
    String firstGroup = matcher.group(0);
    // ...
} else {
    // no match found
}
```

Common String Operations

```
String s = "abcdefg";
String s = String.valueOf(1);
String s = new String("abc") // or new String(chars) for char[]

char c = s.charAt(index);
int res = s.compareTo("abd"); // == -1 (compare lexicographically; also: compareToIgno
reCase)
boolean b1 = s.contains("abc"); // true
boolean b2 = s.endsWith("efg"); // true
boolean b3 = s.equalsIgnoreCase("AbCDeFg"); // true
byte[] bytes = s.getBytes(); // {97, 98, 99} (array of 8-bit numbers [-128,127])
int i1 = s.indexOf('d'); // or with string "d"; can also use lastIndexOf()
int i2 = s.indexOf('f', startIndex);
boolean empty = s.isEmpty();
int len = s.length();
boolean b4 = s.matches("[a-z]*");
String s2 = s.replace('a', 'A');
String s3 = s.replace("ab", "AB");
String s4 = s.replaceAll("[a-z]", "x");
String[] arr = s.split("[0-9]"); // split around digits and remove them from output
boolean b5 = s.startsWith("abc"); // true
String sub = s.substring(startIncluding, endExcluding);
char[] chars = s.toCharArray();
String lower = s.toLowerCase(); // also: toUpperCase
String trimmed = s.trim(); // remove any leading/trailing whitespace
```

Generics

```
// generic classes:
class FruitWriter<S, T extends Fruit & Serializable> {
    public FruitWriter(S s) {
        // ...
    }

    public String write(T t) {
        return t.name();
    }
}

// generic methods:
public <T, E> T foo(T t, E e) {
    return t;
}

// bounded wildcard:
interface HasWord {}
class Baz<T> {}
public void foo(Baz<? extends HasWord> bazWithWord) {
    // method foo only accepts types that extends HasWord
}
```

Iterator

```
interface Iterator<T> {
    boolean hasNext();
    T next();
    void remove();
}
```

Iterable

```
class MyList<T> implements Iterable<T> {

    public Iterator<T> iterator() {
        Iterator<T> it = new Iterator<T>() {
            public boolean hasNext() {
                // TODO: implement
            }

            public T next() {
                // TODO: implement
            }

            public void remove() {
                // TODO: implement
            }
        };

        return it;
    }
}
```

Comparator

```
import java.util.Comparator;
import java.util.Objects;

class IntegerComparator implements Comparator<Integer> {

    @Override
    public int compare(Integer o1, Integer o2) {
        if (o1 < o2) return -1;
        else if (Objects.equals(o1, o2)) return 0;
        else return 1;
    }
}
```

Comparable

```
class MyClass implements Comparable<MyClass> {

    @Override
    public int compareTo(MyClass o) {
        if (this < o) return -1;
        else if (this.equals(o)) return 0;
        else return 1;
    }
}
```

Collections

- The following overview is *greatly simplified* and leaves out many details/interfaces/methods

```
// Implementing this interface allows an object to be the target of the "foreach" stat
ement
interface Iterable<E> {
    Iterator<E> iterator();
}

// The root interface in the collection hierarchy
interface Collection<E> extends Iterable<E> {
    // added in Collection only:
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    boolean containsAll(Collection<?> c);
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean addAll(Collection<? extends E> c);
    boolean remove(Object o);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
}

interface List<E> extends Collection<E> {
    // added in List only:
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    List<E> subList(int fromIndex, int toIndex);
}
```

```
class Stack<E> implements List<E> {
    E push(E item);
    E pop();
    E peek();
}

interface Queue<E> extends Collection<E> {
    boolean add(E e);
    boolean offer(E e);
    E remove();
    E poll();
    E element();
    E peek();
}

interface Set<E> extends Collection<E>

class ArrayList<E> implements List<E>
class LinkedList<E> implements List<E>, Queue<E>
class HashSet<E> implements Set<E>
class TreeSet<E> implements Set<E>
```

```
interface Map<K, V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
}

class HashMap<K,V> implements Map<K, V>
class TreeMap<K,V> implements Map<K, V>
```

Instanceof

- A well-designed object-oriented program should almost *never* use `instanceof`

```
public void doSomething(Object obj) {
    if (obj instanceof Foo) {
        Foo foo = (Foo) obj;
        // do something with foo
    } else if (obj instanceof Bar) {
        Bar bar = (Bar) obj;
        // do something with bar
    } else {
        // do something with obj
    }
}
```

Varargs

- A shortcut to creating an array manually
- Inside method varargs parameter is seen as an array
- Varargs parameter must be the last one, and only one in a method signature

```
public void foo(String... strings) {
    // strings is treated like an array here
}

// invocation:
foo();
foo("a");
foo("a", "b");
foo(new String[]{"a", "b"});
```

Read/Write File

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

Charset charset = Charset.forName("UTF-8")
Path path = Paths.get("/path/to/file");
```

Read

```
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        // process line
    }
} catch (IOException x) {
    // handle
}

// alternatively:
List<String> lines = Files.readAllLines(path, charset) // other overloads available
```

Write

```
String s = ...;

try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}

// alternatively:
Files.write(path, lines, charset);
```

Synchronized

```
class MyClass {

    public synchronized void foo() {
        // synchronizes on the *instance* of the class
    }

    public synchronized void bar() {
        // also synchronizes on the *instance* of the class, sharing the same monitor
with foo()
    }

    public void baz() {
        synchronized(this) {
            // same as bar/baz
        }
    }

    private final Object myLock = new Object();

    public void unrelated() {
        synchronized(myLock) {
            // ...
        }
    }
}
```

Locks, Semaphores

Lock

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

Lock lock = new ReentrantLock();
lock.lock(); // acquire the lock, block until acquired
boolean locked = lock.tryLock(); // acquire the lock if it's free
boolean locked = lock.tryLock(50, TimeUnit.MILLISECONDS); // acquire the lock if it's
free within the given waiting time
lock.unlock(); // release lock, can only be done by thread that acquired lock
```

Other types of locks include `ReadWriteLock` and `StampedLock`.

Semaphore

```
import java.util.concurrent.Semaphore;

Semaphore semaphore = new Semaphore(3); // initialize with 3 permits
semaphore.acquire(); // acquires a permit from this semaphore, blocking until one is a
vailable
boolean acquired = semaphore.tryAcquire(); // aquire a permit only if one is available
semaphore.release(); // release a permit, *no* requirement that releasing thead acquir
ed a permit
```

OOP

- Encapsulation – an object contains (encapsulates) both (1) data and (2) the relevant processing instructions, as we have seen. Once an object has been created, it can be reused in other programs.
- Inheritance – once you have created an object, you can use it as the foundation for similar objects that have the same behavior and characteristics.
- Polymorphism – generics, the presence of "many shapes." In object-oriented programming, polymorphism means that a message (generalized request) produces different results based on the object that it is sent to.

Design Patterns Examples

Creation

- Factory
- Builder
- Singleton

Composition (Structural)

- Adapter
- Facade
- Decorator
- Proxy

Behavioral

- Chain of responsibility
- Command
- Iterator
- Visitor

P, NP, NP-complete

- A problem is in class `P` if its solution may be *found* in polynomial time.
- A problem is in class `NP` if its solution may be *verified* in polynomial time.
- A problem in `P` is in `NP` by definition, but the converse may not be the case.
- `NP`-complete is a family of `NP` problems for which you know that if one of them has a polynomial solution then everyone of them has.
- Examples of `NP`-complete problems: traveling salesman, knapsack, graph coloring.
- Once you've reduced a problem to `NP`-complete, you know to give up on an efficient fast algorithm and to start looking at approximations.
- For `NP`-complete problems, no polynomial-time algorithms are known for solving them (although they can be verified in polynomial time).
- The most notable characteristic of `NP`-complete problems is that no fast solution to them is known.
- `NP`-hard: non-deterministic polynomial time.