Christian Vielma

Interview Study Guide – CS Fundamentals

V1

Disclaimer

I (Christian Vielma) have personally created this guide for personal use, to be used as a reference for technical interviews, work, and study. Since I believe this document might provide value to others, I have decided to make it public.

Most of the content on this document have been added from different sources, from freely available resources on the Internet, to books, with additional added content from me. I made my best effort to provide full credit to the original sources, and in no way I'm trying to take advantage of improper quotations. If you believe I have made an invalid reference to a resource, please let me know and I'll fix it.

I provide this guide as-is, with no additional guarantees. I'm also releasing it under CC-Atribution-ShareAlike License, so you are free to extend it and update it with proper attribution.

Enjoy!

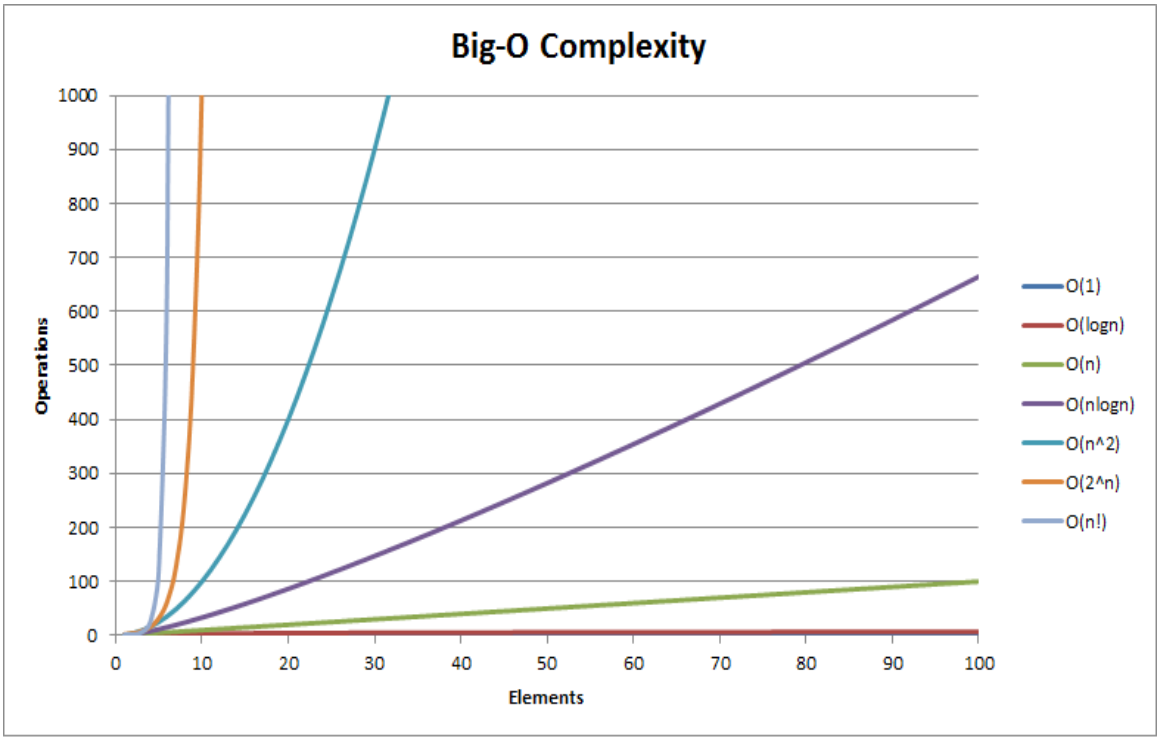
Table of Contents

Some Numbers

Algorithms Time Complexity

Complexity	How large can be n to be reasonably calculated?
n!	10
2^n	40
n^2	1,000,000
nlogn	Doesn't matter
n	Doesn't matter
logn	Doesn't matter

- http://bigocheatsheet.com/
- You can also graph different complexities by using www.wolframalpha.com



Powers of 2

Power	Number	Order	How much space if those where bytes?
2^8	256	< 1,000	
2^{16}	65,536	< 100,000	64 K
2^{31}	2,147,483,648	billions	
2^{32}	4,294,967,296	billions	4 GB
2^{48}	281,474,976,710,656	Hundred trillions	> 1TB
2^{64}	18,446,744,073,709,551,616	>Thousand trillion	

Average Computing Capabilities

Device	Capacity / Speed
Hard Drive	1 TB / 3 GB/s
Memory	16 GB / up to 6-17 GB/s
Internet Bandwidth	7-17 Mbps (US) / 3-13 Mbps (World)
Average Latency	20ms (US) – 200ms(across the world)
Average WebPage Weight (http://www.webperformancetoday.com/2014/12/02/page-bloat-update-average-top-1000-web-page-1795-kb-size/)	~2KB (top 1000)

Any network can be measured by two major characteristics: latency and bandwidth.* ***Latency* *refers to the time it takes a given bit of information to get from one point to another on the network.* ***Bandwidth*** *refers to the rate at which data moves through the network once communication is established.* ***The perfect network would have infinite bandwidth and no latency.*** *Ex: Latency affects voice services (Skype) while bandwidth affects video streaming.*

Bytes Sizes

Multiples of bytes V • T • E

Decimal			Binary				
Value		Metric	Value		JEDEC		IEC
1000	kB	kilobyte	1024	KB	Kilobyte	KiB	kibibyte
1000^2	MB	megabyte	1024^2	MB	Megabyte	MiB	mebibyte
1000^3	GB	gigabyte	1024^3	GB	Gigabyte	GiB	gibibyte
1000^4	TB	terabyte	1024^4	TB	Terabyte	TiB	tebibyte
1000^5	PB	petabyte	1024^5			PiB	pebibyte
1000^6	EB	**exabyte**	1024^6			EiB	exbibyte
1000^7	ZB	zettabyte	1024^7			ZiB	zebibyte
1000^8	YB	yottabyte	1024^8			YiB	yobibyte

Data Structures

Contiguous vs Linked Allocation

Name	Description	Insert	Delete	Search	Random Access
Array	A collection of elements (values or variables), each identified by at least one array index or key.	O(1) (at index)	O (1) (at index)	O(n) (by value)	O(1)
Dynamic Array	Data structure that allocates all elements contiguously in memory, and keeps a count of the current number of elements. If the space reserved for the dynamic array is exceeded, it is reallocated and (possibly) copied, an expensive operation.	O(1) (at the end) O(n) (at the beginning)	O(1) (at the end) O(n) (at the beginning)	O(n) (by value)	O(1)
Linked List	Data structure consisting of a group of nodes. Under the simplest form, each node is composed of a data and a link to the next node in the sequence; This structure allows for efficient insertion or removal of elements from any position in the sequence. **Can be double linked or circular.**	O(1) (at beginning) O(1) (at end if element is known) O(n) (at end if element is unknown)	SearchTime + O(1) (if at end and last element is unkown, then SearchTime is O(n))	O(n)	N/A

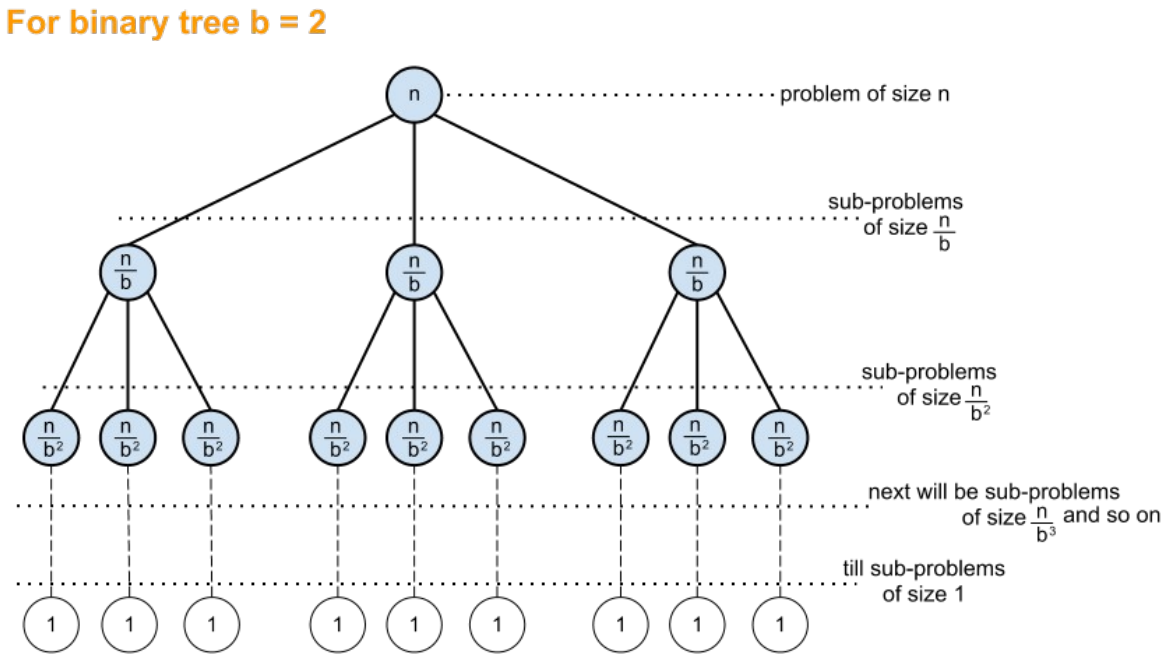
Abstract Data Types

Name	Description	Operations
Stack	Abstract data type or collection in which the relation between the push and pop operations is such that the stack is a Last-In-First-Out (LIFO) data structure	• Pop • Push • Top • isEmpty
Queue	Abstract data type or collection in which the entities in the collection are kept in order . This makes the queue a First-In-First-Out (FIFO) data structure.	• Enqueue • Dequeue • Peek
PriorityQueues	Abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it.	• insert_with_priority • pull_highest_priority
Dictionary	Abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.	• Insert • Reassign • Delete • Lookup

Log(n) Explanation

From: http://stackoverflow.com/questions/2307283/what-does-olog-n-mean-exactly

Side note: the tree below is not a binary tree.



The height of the tree is answer to the following question:

How many times we divide problem of size n by b until we get down to problem of size 1? The other way of asking same question:

when $\frac{n}{b^x} = 1$ *[in binary tree b = 2]*

i.e. $n = b^x$ which is $\log_b n$ *[by definition of logarithm]*

And thats how you get O(log n) which is the amount of recursive work that needs to be done on a tree to reach a solution

Trees

Properties

- Balanced or Unbalanced: is balanced if the height of the two sub-trees differ by at most one.
- Full: is a tree in which every node other than the leaves has two children
- Complete: a tree in which every level, *except possibly the last*, is completely filled.
- Traversal order (L= Left, C=Root/Parent/Center, R=Right):
  - in-order: L C R
  - pre-order: C L R
  - post-order: L R C
  - level-order: breadth first.
- Explanatory video: https://www.youtube.com/watch?v=gm8DUJJhmY4

Name	Description	Insert		Delete		Search	
		Avg	Worst	Avg	Worst	Avg	Worst
Binary Search Tree	Is a node-based binary tree data structure which has the following properties: • The left subtree of a node contains only nodes with keys less than the node's key. • The right subtree of a node contains only nodes with keys greater than the node's key. • The left and right subtree must each also be a binary search tree. • There must be no duplicate nodes. **Is not the same a Binary Tree than a Binary Search Tree.**	O(log n) (balanced)	O(n)	O(log n) (balanced)	O(n)	O(log n) (balanced)	O(n)
Heap	Is a specialized tree-based data structure that satisfies the heap property: If A is a parent node of B then key(A) is ordered with respect to key(B) with the same ordering applying across the heap. There is no implied ordering between siblings or cousins and no implied sequence for an in-order traversal.	O(log n) (balanced)	O(n)	O(log n) (balanced)	O(n)	O(log n) (balanced)	O(n)
AVL	Is a self-balancing binary search tree In an AVL tree, the heights of the two child subtrees of any node differ by at most one. **Is based on a balance factor. Better for lookup intensiveoperations.**	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Red Black Tree	Is a type of self-balancing binary search tree. The self-balancing is provided by painting each node in such a way that the resulting painted tree satisfies	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)

	certain properties that don't allow it to become significantly unbalanced.						

The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree. (http://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/)

Tree Rotation

"Tree Rebalancing" by Mtanti at en.wikipedia. Licensed under CC BY-SA 3.0 via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Tree_Rebalancing.gif#/media/File:Tree_Rebalancing.gif

Heap Implementation

In Java (from: http://once-upon-a-java.blogspot.com/2011/09/max-heap-using-priorityqueue.html)
By default a PriorityQueue is a min-heap. For a max heap is necessary to create it with a comparator. E.g:

```
import java.util.Comparator;
public class MyComparator implements Comparator<Integer>
{
    public int compare( Integer x, Integer y ) {
        return y - x; }
};
```

A min heap on integers is simply defined as:

```
PriorityQueue heap=new PriorityQueue();
```

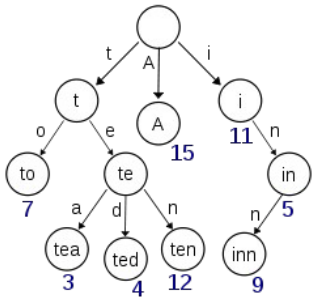
Whereas a max heap on integers, using MyComparator, is implemented as:

```
PriorityQueue heap=new PriorityQueue(n, new MyComparator());
```

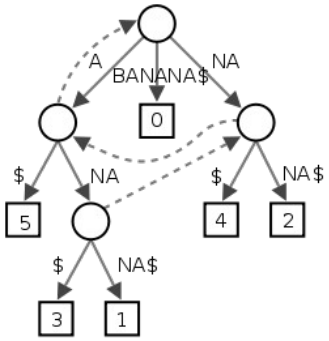
n is the initial size of the heap.

Tries and Suffix Trees

From: https://en.wikipedia.org/wiki/Trie



- Lookup= O(m) (m=String length) (sometimes better than imperfect hashtable O(n), although a good hash is O(1) and O(m) evaluating the hash).
- Application: mobile dictionaries.
- Sorting can be done traversing the trie in pre-order traversal.
- Suffix tree can be used to perform fast full text searches.



Suffix Tree (PAT Tree, position tree):

- Check if string of size m is a substring in O(m).
- Good for:
  - String search (O(n) to build (Ukkonen's, if not O(n^2)) O(m) to search)
  - Find the longest repeated substring.
  - Find the longest common substring.
  - Find the longest palindrome in a string.
  - DNA patterns, etc..

Graphs (G(V,E))

A graph is an abstract data type that is meant to implement the graph and hypergraph concepts from mathematics. Consists of a finite (and possibly mutable) set of ordered pairs, called edges or arcs, of certain entities called nodes or vertices. As in mathematics, an edge (x,y) is said to point or go from x to y. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references..

Properties

- **Directed vs undirected:** is directed if edges have a direction associated with them.
- **Weighted vs Unweighted:** Each edge or node has a weight.
- **Simple vs Non Simple:** A graph is non-simple if it contains loops or multiple edges between nodes.
- **Sparse vs Dense:** a dense graph is a graph in which the number of edges is close to the maximal number of edge
- **Cyclic vs Acyclic:** is cyclic it has loops.
- **Embedded vs Topological.**
- **Implicit vs Explicit:** is implicit if it is generated "on the fly".
- **Labeled vs Unlabeled:** each node/edge has a identifying label.

Implementations

See: http://www.geeksforgeeks.org/graph-and-its-representations/

- **Adjacency list:** Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices.
- **Incidence list:** Vertices and edges are stored as records or objects. Each vertex stores its incident edges, and each edge stores its incident vertices. This data structure allows the storage of additional data on vertices and edges.
- **Adjacency matrix:** A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices.
- **Incidence matrix:** A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

Additional Reference: Skiena. 2008. *The Algorithm Design Manual*. p 152.

	Adjacency list	Incidence list	Adjacency matrix	Incidence matrix
Storage	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add vertex	$O(1)$	$O(1)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add edge	$O(1)$	$O(1)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Remove vertex	$O(\|E\|)$	$O(\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Remove edge	$O(\|E\|)$	$O(\|E\|)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Query: are vertices u, v adjacent? (Assuming that the storage positions for u, v are known)	$O(\|V\|)$	$O(\|E\|)$	$O(1)$	$O(\|E\|)$
Remarks	When removing edges or vertices, need to find all vertices or edges		Slow to add or remove vertices, because matrix must be resized/copied	Slow to add or remove vertices and edges, because matrix must be resized/copied

Minimum Spanning Tree

Subset of E in G(V,E) That forms a tree connecting all V with minimum edge Weight. An algorithm that calculates this is *Prim's Algorithms*. It's application include: handwriting recognition, circuit design, cluster analysis.

Clique

In an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge.

Hash Table

Name	Description	Insert		Delete		Search	
		Avg	Worst	Avg	Worst	Avg	Worst
Hash Table	In computing, a hash table (also hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)

Important to know:

- How to manage collisions:
  - Using Linked Lists (chainning)
  - Using Open Addressing: the “bucket” array where values are stored has many empty slots and collisions are stored sequentially before or after a certain occupied slot.
  - Others:
    - Coalesced Hashing.
    - Cuckoo Hashing.
    - Robin Hood Hashing.
    - 2-Choice Hashing.
    - Hopscotch Hatching

Example of Hash Functions for a String:

$$H(S) = \sum_{i=0}^{|S|-1} a^{|s|-(i+1)} * char(S_i)$$

S = String, a = number of letters in alphabet, char(S) = letter to int.

String

In computer programming, a string is traditionally a sequence of characters, either as a literal constant or as some kind of variable. The latter may allow its elements to be mutated and/or the length changed, or it may be fixed (after creation). A string is generally understood as a data type and is often implemented as an array of bytes (or words) that stores a sequence of elements, typically characters, using some character encoding. A string may also denote more general arrays or other sequence (or list) data types and structures.

Additional Data Structures

- Interval trees: http://en.wikipedia.org/wiki/Interval_tree

Algorithms

Comparison Sorts

Name	Description	Stable	Avg.	Worst	Space
Selection Sort	An in-place comparison sort. Finds the smallest element and put it at the beginning, then searches for the next smallest element and repeat until the collection is sorted..	Y	$O(n^2)$	$O(n^2)$	O(1)
Bubble Sort	Repeatedly steps through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.	Y	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	1.- Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted). 2.- Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.	Y	O(nlogn)	O(nlogn)	O(n) (auxiliary)
Quick Sort	1.- Pick an element, called a pivot, from the list. 2.- Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). 3.- After this partitioning, the pivot is in its final position. This is called the partition operation. 4.- Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.	N	O(nlogn)	$O(n^2)$	O(n) (auxiliary) O(logn) (auxiliary)
Heap Sort	In the first step, a heap is built out of the data. In the second step, a sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. Once all objects have been removed from the heap, we have a sorted array. The direction of the sorted elements can be varied by choosing a min-heap or max-heap in step one.	N	O(nlogn)	O(nlogn)	O(n)

Tip: Insertion sort average is O(n^2) but in best case can be O(n). Is good on small or almost sorted arrays.

Tip: Java sort() for objects uses Mergesort-TimSort. sort() for primitives is Quicksort.

Animations: http://www.sorting-algorithms.com/

Non Comparison Sorts

Name	Description	Stable	Avg.	Worst	Space
Bucket Sort	Partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. **Good when data is uniform.**	Y	O(n+k)	$O(n^2)$	O(n*k)
Radix Sort	Sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value	Y	O(n+k)	O(k*n)	O(n+k)

* k is the number of distinct items.

Search

Name	Description	Avg.	Worst	Space
Binary Search	In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right.	O(logn)	O(logn)	O(1)

Other Algorithms

Selection Ranking

Selection is used in many algorithms like quicksort. Useful to find the ith smallest element in a list. The best case can be O(n) but worst case $O(n^2)$.

1. Pick a random element in the array and use it as a "pivot". Partition elements around the pivot, keeping track of the number of elements on the left side of the partition.
2. If there are exactly I elements on the left, then you just return the biggest element on the left.
3. If the left side is bigger than I, repeat the algorithm on just the left part of the array.
4. If the left side is smaller than I, repeat the algorithm on the right, but look for the element with rank i-leftSize.

Shuffling (Fisher-Yates)

From Wikipedia: https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

1. Write down the numbers from 1 to N.
2. Pick a random number k between one and the number of unstruck numbers remaining (inclusive).
3. Counting from the low end, strike out the kth number not yet struck out, and write it down elsewhere.
4. Repeat from step 2 until all the numbers have been struck out.
5. The sequence of numbers written down in step 3 is now a random permutation of the original numbers.

```
To shuffle an array a of n elements (indices 0..n−1):
  for i from n − 1 downto 1 do
       j ← random integer with 0 ≤ j ≤ i
       exchange a[j] and a[i]
```

Generate random integer from min to max(inclusive):

```
min + (int) (Math.random() *(max−min +1));
```

Implementations

Merge Sort

https://github.com/gaylemcd/ctci/blob/master/java/Chapter%2011/Introduction/MergeSort.java

QuickSort

https://github.com/gaylemcd/ctci/blob/master/java/Chapter%2011/Introduction/Quicksort.java

HeapSort

From Wikipedia: https://en.wikipedia.org/wiki/Heapsort

```
function heapSort(a, count) is
     input:  an unordered array a of length count

     (first place a in max-heap order)
     heapify(a, count)
     end := count-1 //in languages with zero-based arrays the children are 2*i+1 and 2*i+2
     while end > 0 do
         (swap the root(maximum value) of the heap with the last element of the heap)
         swap(a[end], a[0])
         (decrease the size of the heap by one so that the previous max value will
         stay in its proper placement)
         end := end - 1
         (put the heap back in max-heap order)
         siftDown(a, 0, end)

 function heapify(a, count) is
     (start is assigned the index in a of the last parent node)
     start := (count - 2 ) / 2

     while start ≥ 0 do
         (sift down the node at index start to the proper place such that all nodes below
          the start index are in heap order)
         siftDown(a, start, count-1)
         start := start - 1
     (after sifting down the root all nodes/elements are in heap order)

 function siftDown(a, start, end) is
     input:  end represents the limit of how far down the heap
                   to sift.
     root := start

     while root * 2 + 1 ≤ end do          (While the root has at least one child)
         child := root * 2 + 1        (root*2 + 1 points to the left child)
         swap := root        (keeps track of child to swap with)
         (check if root is smaller than left child)
         if a[swap] < a[child]
             swap := child
         (check if right child exists, and if it's bigger than what we're currently swapping with)
         if child+1 ≤ end and a[swap] < a[child+1]
             swap := child + 1
         (check if we need to swap at all)
         if swap != root
             swap(a[root], a[swap])
             root := swap          (repeat to continue sifting down the child now)
         else
             return
```

Graphs Traversal

Breadth First Search

The algorithm uses a **queue data structure** to store intermediate results as it traverses the graph, as follows:

1. Enqueue the root node
2. Dequeue a node and examine it
    - If the element sought is found in this node, quit the search and return a result.
    - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.

Useful to find shortest path.

Pseudocode:

```
procedure BFS(G,v):
      create a queue Q
      enqueue v onto Q
      mark v
      while Q is not empty:
          t ← Q.dequeue()
          if t is what we are looking for:
              return t
          for all edges e in G.adjacentEdges(t) do
             u ← G.adjacentVertex(t,e)
             if u is not marked:
                  mark u
                  enqueue u onto Q
     return none
```

Depth Search First

Is more space-efficient.

Useful to maze generation or solution.

Pseudocode:

```
 procedure DFS(G,v):
      label v as explored
      for all edges e in G.adjacentEdges(v) do
          if edge e is unexplored then
              w ← G.adjacentVertex(v,e)
              if vertex w is unexplored then
                  label e as a discovery edge
                  recursively call DFS(G,w)
              else
                 label e as a back edge
```

Other Graphs Algorithms

Dijkstra

Solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. It travels the graph from a specific node and sets the distance from the initial node to each other of the nodes, saving which of the paths are the shortest. O(|E| + |V| log |V|). Is the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. All the edges must be non-negative. (Wikipedia).

The trick is to use a heap (priority queue) to always get the unvisited vertex with minimum estimated distance and connect from there.

In a graph without weighted edges, performing BFS will suffice to find shortest path.

Bellman-Ford

Is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.[1] It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. O(|V|*|E|) (Wikipedia)

The algorithm is basically: for each vertex, iterate through all the edges and compare if the path is the shortest. At the end, run the check on the edges one more time, if it decreases, then there's a negative cycle.

Floyd-Warshall

Algorithm used to determine shortest path among all pair of vertices. Doesn't work for negative cycles. Check: http://stackoverflow.com/questions/11704643/am-i-right-about-the-differences-between-floyd-warshall-dijkstras-and-bellman

Basically:

```
for k from 1 to |V|
    for i from 1 to |V|
       for j from 1 to |V|
          if dist[i][j] > dist[i][k] + dist[k][j]
            dist[i][j] ← dist[i][k] + dist[k][j]
         end if
```

Minimum Spanning Tree

Prim's Algorithm

Complexity: O(|E| log |V|) or O($|V|^2$) if using adjacency list.

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

Kruskal's Algorithm

Complexity: O(E log V)

- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty and *F* is not yet spanning
  - remove an edge with minimum weight from *S*
  - if that edge connects two different trees, then add it to the forest *F*, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

Other

- Graph Coloring (http://en.wikipedia.org/wiki/Graph_coloring): useful for scheduling conflicting events.
- Clique (http://en.wikipedia.org/wiki/Clique_%28graph_theory%29): subgraph in a graph where all 2 vertices are connected by an edge. Determining if a graph has a clique is NP.
- Bipartite (http://en.wikipedia.org/wiki/Bipartite_graph): is a graph whose vertices can be divided into two disjoint sets and (that is, and are each independent sets) such that every edge connects a vertex in to one in . Vertex set and are often denoted as partite sets. Determining it can be done using DFS coloring the parent, then neighbors, then neighbor's neighbors with alternating colors. If at any given point we find a colored node with the same color of the current node we are checking, then it is not bipartite.

Mathematics and Probability

Prime Numbers

Proving primality: http://en.wikipedia.org/wiki/Primality_test

1) Check number divisibility by 2 or 3
2) If not, check divisibility by 6k +/- 1 <= sqrt(n) (rounded up)
3) Complexity might be O(sqrt(n)) where n is the number, but it will really depend on the number of bits. (http://programmers.stackexchange.com/questions/197374/what-is-the-time-complexity-of-the-algorithm-to-check-if-a-number-is-prime)

Each number can be decomposed to:

$$2^x * 3^y * 5^z \ldots.$$

Having 2 numbers *x* and *y*, each one expressed as:

$$x = 2^{j0} * 3^{j1} * 5^{j2} \ldots.$$

$$y = 2^{k0} * 3^{k1} * 5^{k2} \ldots.$$

The following properties can be derived:

- **Divisibility (x mod y ==0):** if $j_i <= k_i$
- **GCD(x,y)** = $2^{\min(j0, k0)} * 3^{\min(j1,k1)} * \ldots.$
- **LCM(x,y)** = $2^{\max(j0, k0)} * 3^{\max(j1,k1)} * \ldots.$
- GCD*LCM = x.y

Sieve of Eratosthenes (determine all primes lower than n)

Create a list of consecutive integers from 2 to *n*: (2, 3, 4, ..., *n*).

1. Initially, let *p* equal 2, the first prime number.
2. Starting from *p*, count up in increments of *p* and mark each of these numbers greater than *p* itself in the list. These will be multiples of *p*: 2*p*, 3*p*, 4*p*, etc.; note that some of them may have already been marked.
3. Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this number (which is the next prime), and repeat from step 3.

TIP: important step three for efficiency, get nextPrime!

Probability

- P(A and B) = P(B given A) * P(A)
- If A and B independents => P(A and B) = P(A)*P(B)
- P(A or B) = P(A) + P(B) – P(A and B)
- If A and B are mutually exclusive => P(A or B) = P(A) + P(B)
- **Bayes Theorem (P(B|A) = Prob of B given A):**

$$P(A|B) = \frac{P(A)\,P(B|A)}{P(B)},$$

Other

- **Median:** get number of elements and if even select the average between the two elements in the middle. If odd select the element in the middle.
- Pythagoras: (hypotenuse) $c^2 = a^2 + b^2$
- Circumference = 2*pi*radius.

Counting

Formula	What it counts
$(n-1)!$	Circular permutations of n elements.
n^k	Number of k-words in a n-alphabet.
$n!$	Permutations of n elements.
2^n	Number of subsets from a n-set.
$\binom{n}{k} = \frac{n!}{(n-k!)k!} = \frac{n^{\underline{k}}}{k!}$	(Binomial coefficient) Number of k-subsets from a n-set
$n-m+1$	Number of substrings of length m from a string of length n.
$\binom{n-1}{k-1}$	Number of k-subsets of a n-set that contain n.

Binomial Coefficient Properties

1. $\binom{n}{0} = \binom{n}{n} = 1$
2. $\binom{n}{k} = \binom{n}{n-k}$
3. $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

Other Formulas

Formula	What it means
$n^{\underline{k}} = n*(n-1)*(n-2)*...(n-(k-1))$	(Falling factorial power)
$n^{\overline{k}} = n*(n+1)*(n+2)*...(n+(k-1))$	(Rising factorial power)
$\sum_{k=0}^{n-1} a*r^k = a(\frac{1-r^n}{1-r})$	Geometric Series (Constant radio between successive terms)
$n+(n-1)+(n-2)+....(n-(n-1)) = \frac{n(n+1)}{2}$	(Simplification)
$m = \frac{y2-y1}{x2-x1}$	Slope given 2 points.
$y = mx+b$	Y-intercept given the slope.

Matrices

http://en.wikipedia.org/wiki/Matrix_%28mathematics%29

- **Matrix multiplication ($O(n^3)$):**
  - Multiplying X*Y and Y*Z => M[i,j] = M[i,j] + A[i,k] * A[k,j]
- **Identity:** all elements in diagonal are 1 and rest are 0.
- **Determinants (only square matrices) ($O(n^3)$ - $O(n!)$):** http://en.wikipedia.org/wiki/Determinant#n.C2.A0.C3.97.C2.A0n_matrices
  - $$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a\begin{vmatrix} e & f \\ h & i \end{vmatrix} - b\begin{vmatrix} d & f \\ g & i \end{vmatrix} + c\begin{vmatrix} d & e \\ g & h \end{vmatrix}$$
  - $$= a(ei - fh) - b(di - fg) + c(dh - eg)$$
  - $$= aei + bfg + cdh - ceg - bdi - afh.$$
  - M is singular (doesn't have inverse) iff |M| = 0.
  - Point lies to the left or right of a line or plane: determinant sign.
  - det(A*B) = det(A)*det(B)

- **Permanents ($O(n^2 2^n)$ – NP Hard):**
  - Similar to determinant but doesn't take signs into consideration (just sums).
  - Perm(A*B) != Perm(A) * Perm(B)

Regular Expressions

- In Java use Pattern.compile(expression), and then pattern.matcher(yourstring).matches(). If using special characters (+, ., etc,) you have to escape it using double backslash. Doesn't apply for \d, \w, etc.

From Java's API: http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

. any character.
$ the end of a line.
\d single digit
\w word
\s whitespace
\t tab
\r carriage return
\n newline
 a | b | is OR, a and b are expressions.
() capture group.
{m, n} number of repetitions of previous expressions. At least m, no more than n.
[^abc] any character except a, b or c.
^ start of line.

Bit Manipulation

Operators (Java)

Operator	Meaning
^	XOR
~	NOT
&	AND
|	OR
<< >>	Shifts (signed) (pattern << positions)
>>>	Shift (unsigned)

Plus: +, -, *, /

Tips:

- Shifting to the right is like dividing by 2. Shifting to the left is like multiplying by 2 (except in C).
- When using BinarySearch and the key is not found in the array, it will return **(-(insertion point) -1) = insertion**. To know at which position in the array should we insert it, we can use **~insertion**.
- Sum:
  - XOR sums without carrying (sum = a^b).
  - AND + Shift carries without sum (carry = (a & b) << 1).

  Code:

```
int add(int a, int b)
{    if(b == 0) return a;
     sum = a ^ b;
     carry = (a & b) << 1;
     return add(sum,carry); }
```

- `c = c & (c - 1);` clears the least significant bit.
- For many problems, they can be reduced to creating a mask and operating based on this mask.

Big-endian (Java) => MSB lowest address. Little-endian => LSB lowest address.

Hexadecimal: Hexadecimal notation, prefix with `0x`. Letters A to F can be either lower case or upper case. An integer literal is of type "long" if it is suffixed with `L` or lower case `l`; otherwise it is of type "int". **Example:** 0x10 = 16, 0xA =10.

Example of shifting (with Integer.MIN_VALUE and Integer.MAX_VALUE):

```
Normal
10000000000000000000000000000000
01111111111111111111111111111111
>>1
11000000000000000000000000000000
00111111111111111111111111111111
<<1
00000000000000000000000000000000
11111111111111111111111111111110
>>>1
01000000000000000000000000000000
00111111111111111111111111111111
>>31
11111111111111111111111111111111
00000000000000000000000000000000
>>>31
00000000000000000000000000000001
00000000000000000000000000000000
```

Threads and Locks

(check SCJP study guide or move this from here to that guide)

Implementing Threads in Java

Can be done implementing the **Runnable** interface (**run** method) and passing it to a **Thread** instance constructor or extending from **Thread** directly.

Synchronized Blocks

Only a class or an object can be **synchronized** (using **synchronized(Object or Class)**). The **synchronized** can be use for methods or blocks inside methods.

Tip: To synchronize a HashSet can be used Collections.SynchronizedSet(new HashSet());

Locks

Besides the synchronized way to support concurrency, there are also ways using locks, which provides a few more features. To use a lock you must create a lock object (ex: **lock = new ReentrantLock()**), use **tryLock() or lock()** (lock try to get the lock and disables the thread until it gets the lock) to get the lock and **unlock()** method to unlock it.

Deadlocks

A deadlock occurs when 4 conditions are met:

1. **Mutual exclusion:** At least two resources must be non-shareable
2. **Hold and Wait:** A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. **No Preemption:** The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.
4. **Circular wait:** A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.

Avoiding any of these conditions will suffice to prevent a deadlock.

Busy Wait

Is when a thread makes an active waiting while other thread is working. The best way to avoid it is to use wait() (the thread will be suspended until object is notified).

Tip: Wait() has to be called inside a Synchronized block.

Miscellaneous

Turing Machine

From: http://en.wikipedia.org/wiki/Non-deterministic_Turing_machine

In essence, a Turing machine is imagined to be a simple computer that reads and writes symbols one at a time on an endless tape by strictly following a set of rules. It determines what action it should perform next according to its internal "state" and what symbol it currently sees. An example of one of a Turing Machine's rules might thus be: "If you are in state 2 and you see an 'A', change it to 'B' and move left."

In a **deterministic Turing machine**, the set of rules prescribes at most one action to be performed for any given situation. A **non-deterministic Turing machine** (**NTM**), by contrast, may have a set of rules that prescribes more than one action for a given situation. For example, a non-deterministic Turing machine may have both "If you are in state 2 and you see an 'A', change it to a 'B' and move left" and "If you are in state 2 and you see an 'A', change it to a 'C' and move right" in its rule set.

NP Problems

From: http://www.geeksforgeeks.org/np-completeness-set-1/

What are NP, P, NP-complete and NP-Hard problems?
P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).
Informally, NP is set of decision problems which can be solved by a polynomial time via a “Lucky Algorithm”, a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn’t need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.

How to prove that a given problem is NP complete?
From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

Watch out for

Coding

- **Validate input.**
- **Mention force brute approaches.**
- Static methods being called in non-static contexts.
- Casting of return types for API classes when instantiating.
- Parameters in recursive functions.
- Check error conditions.
- **Never check doubles or floats with == but instead that the difference is less than an epsilon.**
- Always design structures. Almost always more is better.
- Watch Lists! Sometimes instead of a list could be better think of a **Stack.**
- **When Storing, think in BST(if data needs to be sorted) or Heap(if extracting min or max needs to be fast) too.**
- **Consider "Runner"** for searching loops in LinkedLists.
- **Consider "Tail Recursion"** (is like iterating).
- **Consider top-down and bottom-up** when dealing with recursion problems. Also consider DP.
- In graphical problems with matrix traversal x,y a point in the Cartesian grid is A[y][x] instead of A[x][y].
- When stuck, consider advanced or not common features of the language.
- Be careful with parameters passed in recursive functions.
- Longest Increasing subsequence O(nLogn) using Patience Sort. Longest common subsequence O(n^2) with DP.
- String operations: watch for uppercase, lowercase, and **encoding!** (among other things).
- Watch repeated data as in list of names, ages, etc.
- It's better to use int than Integer because of the overhead creating the object.
- Test even and odd sized arrays as inputs.
- Watch variable names.
- Check functions signatures in calling methods.
- Don't forget return (types and keyword).
- In Java Lists use size(), not length(). Length() is for strings.
- Java always passes parameters by value.
- Watch out for the type of the variable (interface vs specific class).
- When you need something (a list, for example) of a specific size, consider padding.
- Be generous with the classes, create enums, or well specified classes for better code.
- Careful with Floats vs Doubles or assuming ints.
- Try to simplify base case in recursions after implementation.
- .toArray method returns Object[].
- Maps have methods containsKey and containsValue.

Object-Oriented

- Ask 6 Ws: who, what, where, when, how, why. **Ask a lot!**
- When designing try to think in all the possible fields not mentioned.
- Consider Maintainability and style.
- Overload != Override.

Scalability

- Consider if the data will fit in memory.
- How to store data? (Files, SQL, XML, don't assume database). Who will consume it?

Search and Sorting

- Is important to be a stable sort?
- When searching 2 elements don't assume they're different.
- Consider empty string in string searches.
- Think order the other way around if one of the ways is expensive.
- If there is not need for a complete ordering, consider Bucket Sort.

Trees

- When checking complexity, it will be O(logN) only if tree is balanced.
- When checking BST, keep track of the values.
- The best trick to implement a tree is to have a class Tree with a TreeNode root. TreeNode will have the value and children. All the operations are performed at Tree, so it's easier to handle pointers.
- AVL: use function to get height. Red Black: depends on coloring properties, use pointer to parent.
- When thinking about algorithms, think maybe converting the tree to an array (pre-order, in-order).

Probability

- Check complementary set.

Testing

1. Type of test (white/black box)
2. Who will use it and why?
3. What are the use cases?
4. What are the bounds of use?
5. What are the stress conditions / failure conditions?
6. What are the test cases? How would you perform the testing? (automated, manual etc)

Additional

- Ask about the problem as much as possible before giving answers.
- C does static binding.
- **Watch for overflows in bit operations.**
- **Use the idea of flip bits to make decisions with bit operations.**
- If it seems easy: be sure of the answer (test) and really careful coding design.
- SQL be careful what to return.
- In Java Str.charAt(i) – '0' converts str to int
  (check: Morgan, Giguere, Kindler. 2012. Programming Interviews Exposed: Secrets to Landing your Next Job).
- When sorting, check first what data and if any data structure could help. **Stable sort is unnecessary if the data is unique.**
- Adding data to objects increases space complexity, check if it affects the Big Oh.
- Abstract class can have member attributes, methods definitions, interfaces no.

Sorts

- Quicksort:
  - First partition, then quicksort.
  - Return pivot after partition (in place).
  - Consider that is not divided in two halves.
- Mergesort:
  - Watch indexes of start and end when moving.
- Heapsort:
  - SiftDown is repairing that tree, not all the tree.

Testing

- When testing consider:
    - 6 W.
    - Use cases.
    - Bounds of use, stress/failure.
    - How to test.
    - Black/White Testing.
    - Manual vs Automate.
    - Metrics to use.
    - Include empty, border case, different length in case of comparisons.
    - Check parameters input.
    - Final only means the value cannot be re-assigned, but the state of the object can be modified.

Additional Tips

Data Structures

- Think heap for storage retrieval

Find duplicates

Some approaches:

- Two loops (O(n^2)).
- Counter.
- Equation (sum and product of the array).
- XOR (based on rule that A^A = 0)
    - XOR all the elements together.
    - Get a different bit set in X and Y (non-repeating numbers) (setbit = xor & ~(xor-1)).
    - Loop through the array dividing the array with number with the bit set and not set, using xor in both set. The number in one set will be X and the other will be Y.
- Sorting.
- Hashtable or HashSet.

Longest Palindrome

Some approaches:

- $O(n^3)$: check strings from length 1 to n, starting on each position. Can be improved by starting with the larger words first. If found, we can stop.
- $O(n^2)$: Using DP. Initialize boolean matrix ([x][y] palindrome words starting in x with length y) for all one char and 2 chars palindromes. Then check words with length >= 3, by comparing if the internal string is palindrome and the two characters surrounding it are equal.
- O(n): Using Manacher algorithm. Preprocess the string by adding separator characters and line start and end, keeps an array p with value p[i] equal to the length of the palindrome centered in character I. Increase p[i] by comparing the characters around center I.

Longest Increasing Subsequence

Some approaches:

- $O(n^2)$: Brute force: for each element n calculate the longest subsequence that ends in that character by recursively calling it on n-1 and summing up the max.
- $O(n^2)$: DP: doing the same as above but storing the temporary results.
- O(nlog(n)): Using patience sorting.

Find Minimum Distance among elements in two lists

Can merge both lists using a tag and then search through the list

(Gayle. 2011. Cracking the Coding Interview. 5Th Edition. Problem 18.5).