Short Circuit

Algorithms

Grokking Algorithms
by Aditya Y. Bhargava
Second Edition

Summary by Emiel Bos

1 Data structures

Data Structure	Time Complexity (Average)			Time Complexity (Worst)			Space Complexity (Worst)
	Access	Search	Insert / Delete[1]	Access	Search	Insert / Delete	
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
Hash table	N/A	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$
Binary search tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
B-tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Splay tree	N/A	$\Theta(\log n)$	$\Theta(\log n)$	N/A	$O(\log n)$	$O(\log n)$	$O(n)$

1.1 Memory

Data is stored in RAM, where each byte has an address. RAM is a sequential series of bytes, and RAM stands for Random Acces Memory, because you can access any position/address in RAM in constant time. If you want to store any data, you have to ask your OS, which will *allocate* (reserve) the requested amount of contiguous space and return the address of the first byte.

1.2 Arrays

Arrays are stored contiguously in RAM.

Static arrays have a fixed length. It never needs to be moved when it runs out of space, but it is highly inflexible. Accessing the i'th element is $O(1)$ (*random access*), inserting and/or removing the last element is $O(1)$ (which is just overwriting it), inserting and/or removing anywhere else is $O(n)$, because you need to shift every element.

Dynamic arrays double in size whenever they run out of space, with the new array allocated somewhere else. All time complexities are still the same. Even pushing n elements is still only $O(n)$, even when starting with an array of length 1, because it takes $1 + 2 + 4 + 8 + ... + \frac{n}{2} + n$ operations, which is never more than $2n$.

1.3 Linked lists

A *(singly) linked list* consists of (list) nodes, each containing a value and a pointer (address) to the next node. Linked lists do not need to be stored contiguously and never need to move. Accessing and searching is $O(n)$ due to traversal (*sequential access*). Inserting/deleting is also $O(n)$ because of traversal, except at the start or end when maintaining a `head` or `tail` pointer. The time complexities are basically the inverse of an array: finding the point of insertion/deletion is $O(n)$ for linked lists and $O(1)$ for arrays, while performing the insertion/deletion is $O(1)$ for linked lists and $O(n)$ for

arrays. In other words, arrays are better for reads, linked lists are better for writes.

A *doubly linked list* has nodes which also point to the previous node. Deleting at the end is now also $O(1)$, because we have a pointer to the penultimate node (`tail.prev`).

1.4 Stacks

A stack is a LIFO (last in, first out) data structure which supports pushing, popping and peeking (at the top) in constant time. Stacks can be implemented as dynamic arrays or doubly linked lists.

1.5 Queue

A queue is a FIFO (first in, first out) data structure which supports enqueueing to the start and dequeueing from the end in constant time. Are best implemented as linked lists.

A double-ended queue is like a generalization of a stack and a queue which supports $O(1)$ pushing and popping to either end. Python has `deque` in the `collections` module. Java has the `ArrayDeque` and `LinkedList` container classes that both implement interface `Deque`, which you can use as a stack with `push()`/`addFirst()` and `pop()`/`removeFirst()`/`remove()` for adding/removing on top/at the front, or as a queue with `addLast()`/`add()` (at the end) and `removeFirst()`/`pop()` (at the front).

1.6 Hash tables

A *hash function* is a function where you put in a string or any kind of sequence of bytes, and you get back a number. The function has to be consistent (same input maps to the same output). This output number can be used to deterministically index into an array by modulo'ing it by the array size. A *hash table*, *(hash) map*, *dictionary* (dicts in Python), or *associative array* is a hash function with an array.

Collisions happen when different keys map to the same index. Collisions are avoided with a good hash function – i.e., one that distributes values in the array evenly – and a low load factor. The *load factor* of a hash table is the percentage of occupied slots in the array. Having a load factor greater than 1 means you have more items than slots in your array. Each time the load factor is above something like 0.7, the array is doubled[2] and all existing key-value pairs are moved to their new hashing location (which is dependent on array size), and this is called "rehashing the array". But collisions may still occur, and they can be handled in different ways:

- *Chaining*: instead of one key-value pair per index, store a linked list per index.
- *Open addressing*: if you want to store a key-value pair and an index is occupied, you just put the key-value pair at the next unoccupied index. When searching, you search until the next unoccupied index (at which point you know it's not in there, or else it would be at that spot). The downside of +1'ing is that you cluster all elements, and there are better ways, like continuously squaring and modulo'ing.

Hash tables have $O(1)$ time complexity for inserting, removing and searching[3][4]. The caveat is that you can't easily traverse it in order, which would take $O(n \log n)$. They also can't contain duplicates. Use cases are fast lookup, filtering out duplicates, and caches.

1.7 Graphs

Graphs have V vertices and E edges. For directed graphs, $E \leq V^2$ (vertices can point to themselves). In undirected graphs, edges don't have a direction. In directed graphs, nodes have in-neighbours (that points toward them) and out-neighbours.

Binary trees are highly similar to linked lists, but with different terminology and no cycles. The height of a node is the longest path going down. The depth of a node is the length of the path to the root.

Graphs are most often represented as an adjacency list, which concretely is either

- a collection of node/vertex objects, each of which has a value and a list of references to other node objects.

[2]The size of the array should actually be a prime number to further avoid collisions for math reasons, so in practice it's not quite doubling it when resizing.

[3]Whereas BSTs (aka treemaps/sets) have $O(\log n)$ for all three.

[4]Actually, $O(1)$ is the average time complexity and $O(n)$ is the worst time complexity if you have a bad hash function, but you can basically assume that $O(1)$ is simply the worst case.

- a hash table with the node value as key and list of (out-)neighbours as values (in which each element is that neighbour's key). Each node must have a unique value for this to work, which is often the case in interviews.
- A list of node pairs.

Time complexities are often expressed as something like $O(V + E)$, or sometimes like $O(n + m)$, where $n = V$ and $m = E$.

A *topological sort* is (one of multiple possible) valid ordered lists of a directional graph.

In a *weighted graph*, each edge has an associated *weight*. One way of implementing/representing a weighted graph in code is with a nested hash table: the top-level table maps each node to a nested table, which in turn maps (only) that node's out-neighbours to their weights.

Graphs are less commonly represented as a matrix, a 2D array in which each **0** is a node and each **1** is not, and then you can move up, down, left, right (and sometimes diagonally) to each neighbouring node. So the graph is laid out like a grid, with possible holes.

1.7.1 Trees

A *tree* is a connected, acyclic graph. *Rooted trees* have exactly one node that leads to all the other nodes: the *root*, which doesn't have parents. We'll only work with rooted trees. Nodes can have children and a parent. Nodes with no children are *leaf nodes*.

A *binary tree* is a special type of tree where nodes can have at most two children. The *height* of a binary tree is the longest path from the root node to any leaf node. Huffman coding (the foundation for text compression algorithms) uses binary trees.

Binary search trees In a *binary search tree* (BST), everything in a node's left/right subtree is less/greater than the node's value. A balanced BST is one where each node's left and right subtree have equal height, both `\log(n)`. Searching a balanced tree is $O(\log n)$, searching a super unbalanced tree (basically a linked list) is $O(n)$, and generally it is $O(h)$, where h is the height. The main advantage of BSTs over (sorted) arrays is that adding/removing elements/nodes can also be $O(\log n)$ (whereas it is $O(n)$ for arrays). It's somewhat of a middle ground between arrays and list.
Adding nodes as leaf nodes is easier than adding nodes as root or parent nodes, but does result in unbalances trees. This way is very similar to searching; we essentially search for the value to insert until we reach a null, and then create and add the node there.
Removing a node is more involved. First we need to search for it. If the node has 0 children, just remove it and set the parent pointer to null, if the node has 1 child, set the parent pointer to point to that child, bypassing the node. If the node has two children, we take the leftmost descendant (with the smallest value) from the right subtree, and replace the node with that. You can also take the rightmost descendant (with the largest value) from the left subtree. Finding the minimum/maximimum is easy: just keep taking left/right turns. Still O(log n). Sets and maps can be implemented with BSTs, and they're called someting like OrderedSet/OrderedMap or TreeSet/TreeMap. BST maps are called SortedDict in Python (you need package sortedcontainers), map<T,T> in C++, and TreeMap<T,T> in Java.

AVL trees *AVL trees* are a type of self-balancing BST, i.e. it will rebalance whenever the tree is out of balance (with a height difference between leaf nodes of more than 1) by rotating. A *rotation* is an operation on a binary tree that changes the structure without interfering with the order of the elements. It moves one node up in the tree and one node down. Lookup, insertion, and deletion all take $O(\log n)$ time in both average and worst case. Each node stores either its height or its *balance factor*, which tells which child subtree is taller and by how much. Negative values indicate the left tree is taller and vice versa, and a 0 indicates a balanced tree. -1 or 1 are okay, because AVL trees allow a maximum difference of 1. If you have the heights of each subtree, it is easy to compute the balance factor by subtracting the right subtree's high by that of the left subtree. New nodes are added as leaf nodes (with height or balance factor 0), and all its ancestors heights/balance factors are updated. If a node's balance factor becomes -2 or 2, it is immediately rebalanced. After a rebalance, you don't need to move further up the tree, because AVL trees require one rebalancing at most.

Splay trees A different take on balanced BSTs. If you have recently looked up an item, the next lookup for that item will be faster, because when you look up a node, it will make that node the new root. More generally, recently looked up nodes get clustered to the top and become faster to look up.

B-trees B-trees are a generalized form of binary tree, where each node can have more than one key and more than two children. More specifically, a node's number of children is one greater than it's number of keys. *Seek time* is the time

it takes for the hard disc's read/write head to move to the correct track, i.e., it's a physical mechanism that introduces a delay. B-trees optimize this seek time. They spend more time reading each node, but they seek less because they read more data into memory in one go. For this reason, they're a popular data structure for databases. Like BSTs, everything in a node's left/right subtree is less/greater than the node's value.

Trie A *trie*, or prefix tree, is a rooted tree data structure used for retrieval of a key in a dataset of strings. It's used for autocomplete, spellchecking, IP routing, etc. Each node in the tree has a link for each possible character (which by default are null) and a boolean indicating if the current node ends a valid word that has been added (else its just a prefix).[5] If m is the word/key length, insertion is $O(m)$ time and space, searching for the exact input word and searching any word with the given prefix is $O(m)$ time and $O(1)$ space. For insertion and search, hash tables are equally as good, but they don't support prefix search.

1.7.2 Forest

A *forest* is a disjoint union (basically a collection) of trees.

Union-find A *union-find*, also known as a disjoint-set or merge-find, stores a collection of disjoint (non-overlapping) sets – a partitioning of a set into disjoint subsets. They're often implemented as a forest of trees data structure that can be used on undirected graphs with disjoint connected components for counting the number of connected components and for cycle detection. The trees aren't meant to accurately represent the graph. Given the number of nodes and a list of unique, undirected edges (so `[1,2]` and `[2,1]` won't both appear), we assume each node is disjoint initially. We keep track of a parent pointer for each node (stored in a hash map, mapping node value to parent node), and the rank (height) of each tree (also stored in a hash map, with the root IDs as keys), which we want as small as possible for best performance. Iterating through the list of edges, for each edge (i.e. pair of nodes), we find the roots of the trees of both nodes by following the nodes' parent pointers. If the roots are the same, the nodes are already in the same tree, and since edges in the list are unique, there is a cycle. Else, we parent the root of the smallest tree (`ranks[root]`) to the root of the largest tree. If both trees have the same rank, choose one of two parentings and increase the relevant rank. While finding the roots, we can do path compression, i.e. we keep track of all nodes in the path, and then parent those all to the root after we found it.

1.8 Priority queues

Priority queues are like queues, except the order of popping is done on the minimum/maximum value. Lookup of this value is $O(1)$ (compared to $O(\log n)$ for BSTs), because that element is always the root. Popping it and replacing it with the next value is $O(\log n)$ time. They're often implemented with heaps, but implementations with other data structures are possible. *Heaps* are binary trees with two properties:

- The *structure property* dictates that it is a complete binary tree, meaning only the lowest level may have holes, and those holes should be on the right (so nodes are added in breadth-first order).
- The *order property* states that the value of a node should be smaller/larger than or equal to all values in both its subtrees.

Duplicate values are allowed. Under the hood, heaps use arrays. The 0'th index is skipped, and all other nodes are in the array in breadth-first order. For every node at index i, its left child is at index $2i$, its right child at index $2i + 1$, and its parent at $\lfloor \frac{i}{2} \rfloor$.

Adding a node is done by putting it at the next proper spot, and then keep swapping it with its (current) parent while it is larger/smaller than that parent (a good term for it is "percolating" upwards). Popping the highest priority node is done by returning and removing the root node, replacing it with the last node, and then percolating that down, i.e. continually replacing it with the smallest/largest of its two children.

You can heapify an existing array in $O(n)$ time by first moving the first element to last to create space for the 0'th dummy element and then looping from the back of the array to the front, percolating each element down.[6] Because $\frac{n}{2}$ nodes don't move at all, $\frac{n}{2}$ only move one down, $\frac{n}{4}$ move at most two down, etc., you get a time series that basically approximates n. Searching a priority queue for an element is $O(n)$, but that's not their use anyways.

[5] You can also visualize it as the nodes indicating characters, with the root node representing the empty string "", but I find it more intuitive to have the links contain the characters.

[6] You can actually start the algorithm at index $\lfloor \frac{i}{2} \rfloor$, because that's the last node in the array with at least one child.

2 Algorithms

2.1 Big O notation

Big O notation tells you how fast an algorithm is and how the running time or memory allocation size increases as the input size increases, assuming worst case scenarios. It is expressed as the number of operations. Constants are left out, because those capture hardware/performance/implementation differences that are irrelevant and are therefore abstracted away. Common complexities are, from fastest to slowest:

- $O(1)$, or *constant time*, e.g. hash table lookup.
- $O(\log n)$, or *logarithmic time*, e.g. binary search.
- $O(n)$, or *linear time*, e.g. linear search.
- $O(n \log n)$, or *linearithmic time*, e.g. quicksort.[7]
- $O(n^2)$, or *quadratic time*, e.g. selection sort.[8]
- $O(n^3)$, or *cubic time*.
- $O(n^k)$, or *polynomial time*, where k is a constant. A generalization of quadratic and cubic time.
- $O(k^n)$, or *exponential time*.
- $O(n!)$, or *factorial time*, e.g. traveling salesman.

2.2 Recursion

Recursion is where a function calls itself. There's no performance benefit; it is used when it makes the solution clearer. Every recursive function has a *base case* (or else it would loop infinitely) and the *recursive case*. Each of the recursive function calls takes up some memory on the *call stack*. If your recursion may go too deep and the call stack runs out of space, you can rewrite your code to use a loop instead, or use *tail recursion*.

One-branch recursion (for example: calculating a factorial) is at least $O(n)$ time and space complexity (because the function call takes up space). Two-branch recursion (for example: calculating Fibonacci sequence) is also at least $O(n)$ time and space complexity.

Divide-and-conquer algorithms are two-branch recursive algorithms that divides or splits up a problem in smaller problems, each is solved recursively, until a base case is hit.

2.3 Binary search

Binary search of an element in a sorted array is a divide-and-conquer algorithm with $O(\log n)$ time complexity and $O(1)$ space complexity.

```
def binary_search(nums, target):
   if len(nums) == 0:
     return -1

   lo, hi = 0, len(nums) - 1
   while lo <= hi:
     mid = (lo + hi) // 2
     if nums[mid] > target:
       hi = mid - 1
     elif nums[mid] < target:
       lo = mid + 1
     else:
       return mid

   return -1
```

[7] Exponentiation is another example of a calculation that costs $O(\log n)$, though some languages may be implemented in a way that exponentiation on integers costs $O(1)$.

[8] If you come up with an algorithm that is $O(n^2)$, always consider if sorting the input array ($O(n \log n)$) allows solving the problem in $O(n \log n)$ or better.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(\log n)$
Mergesort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log n)$	$\Theta(n(\log n)^2)$	$O(n(\log n)^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log n)$	$O(n \log n)$	$O(n)$

2.4 Sorting algorithms

Stable sorting algorithms preserve the original order if there's a tie with the comparison operator, while unstable sorting algorithms don't guarantee this.

2.4.1 Selection sort

The simplest sorting algorithm is *selection sort*, which goes through the list searching for the minimum element again and again. $O(n^2)$.

2.4.2 Insertion sort

Insertion sort is stable and $O(n^2)$. Outer loop loops with `i` through the array from front + 1 to back. Everything to the left of `i` is sorted. Inner loop brings the element at `i` from `i` backwards to its place by swapping as long as it's out of order.

2.4.3 Mergesort

Mergesort is divide-and-conquer (two-branch recursion); it splits the array into two newly allocated arrays, recursively sorts each half and then merges those sorted subarrays into the original array. Time complexity is $O(n \log n)$: there are $\log n$ levels/splits that each require n operations. It's stable if you take care to prioritize the first half when merging.

```
def mergeSort(array):
    if len(array) < 2:
        return

    left = array[:len(array)//2]
    right = array[len(array)//2:]
    mergeSort(left)
    mergeSort(right)

    i = j = 0

    # Add until we reach either end of either left or right
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            array[i+j] = left[i]
            i += 1
        else:
            array[i+j] = right[j]
            j += 1

    # Add remaining elements. Note that only one of these loops will execute
    while i < len(left):
        array[i+j] = left[i]
        i += 1

    while j < len(right):
        array[i+j] = right[j]
```

```
        j += 1
```

Mergesort can theoretically be done in-place (avoiding extra array allocation), but implementing the merge step is messy and complex, and it's usually slower than the standard version because of all the extra element shifting and careful indexing.

2.4.4 Quicksort

Quicksort, developed by Tony Hoare, is similar to mergesort. It is also divide-and conquer, but no additional arrays are needed; everything happens in-place. Also, quicksort first partitions the array and then make two recursive calls, whereas mergesort first makes recursive calls for the two halves and then merges them. At each level, the rightmost element is selected as pivot, and the rest of the array is looped front-to-back, and if an element is smaller than the pivot, it is swapped with the element at a separate "swapper" pointer that also starts at the beginning and is incremented with every swap. Lastly, the pivot itself is swapped with the element at the swapper pointer, and it then separates the array in two halves, each of which is unsorted but all elements on the left side are smaller than all elements on the right side, and each side is recursed (without the pivot).

```
def quicksort(nums):
    if len(nums) < 2:
        return nums
    else:
        pivot = nums[0] # This isn't using Lomuto partition scheme, which uses the last element
        less = [i for i in nums[1:] if i <= pivot]
        greater = [i for i in nums[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

Worst time complexity is $O(n^2)$ with very unlucky pivots, but in practice its $O(n \log n)$. Even though quicksort is $\Theta(n \log n)$ (on average) and merge sort is $O(n \log n)$ (always), in practice, quicksort's average case is faster than merge-sort's average case and quicksort hits its average case way more often than its worst case. The performance of quicksort heavily depends on the choice of pivot, but always taking a random will guarantee the $\Theta(n \log n)$ average runtime.
The above method of partitioning the array into halves smaller and larger than the pivot is called Lomuto partition scheme. There are many partition schemes, but this one is compact and easy to understand, though less efficient than Hoare's original scheme, e.g. when all elements are equal. With this scheme, time complexity degrades to $O(n^2)$ when the array is already in order, due to the partition being the worst possible one.

You can't sort an array in $O(n)$ time on a single CPU core, but there is a parallel version of quicksort that will sort an array in $O(n)$ time.

Quickselect, also developed by Tony Hoare, is a highly similar *selection algorithm*, which are algorithms for finding the kth smallest (or largest, by tweaking the below code slightly) element in an unordered list. The array is partitioned in two halves just like in quicksort, but because we know in which partition the desired element lies, only one side needs to be recursed. This has time complexity $\omega(n)$ on average (which is almost always the case in practice), and $O(n^2)$ worst-case complexity, but this only happens on edge cases.

```
def kthSmallest(nums, k):
    def partition(left, right, pivot):
        nums[pivot], nums[right] = nums[right], nums[pivot]

        store_index = left
        for i in range(left, right):
            if count[nums[i]] < nums[pivot]:
                unique[store_index], unique[i] = unique[i], unique[store_index]
                store_index += 1

        nums[right], nums[store_index] = nums[store_index], nums[right]

        return store_index

    def quickselect(left, right, k) -> None:
        if left == right: # Base case; the list contains only one element
            return

        pivot = random.randint(left, right) # Select a random pivot index
        pivot = partition(left, right, pivot) # Find the pivot index in a sorted list

        if k == pivot: # If the pivot is in its final sorted position
```

```
            return
        elif k < pivot:
            quickselect(left, pivot - 1, k)
        else:
            quickselect(pivot + 1, right, k)

    n = len(nums)
    quickselect(0, n - 1, k)
    return nums[k]
```

2.4.5 Bucket sort

Bucket sort is $O(n)$ but is unstable and can only be used when all values of elements are from some small set. In one pass of the array, all possible values are counted in a separate array where the indices are the values, and in a second pass, the original array is overwritten using these counts. Only used in certain situations.

In a more general formulation, the buckets span ranges of values, and the buckets are individually sorted with some other sorting algorithm before being concatenated.

2.5 Graphs

2.5.1 Breadth-first search (BFS)

Either for determining whether a path exists between two nodes, or for finding the path along the least number of edges. BFS radiates out from the starting point and is $O(V + E)$.

```
def bfs(searched_item):
    queue = deque()
    queue += graph[searched_item]
    searched = set()
    while queue:
        current_item = queue.popleft()
            if not current_item in searched: # Avoid infinite loops (not needed for trees)
                if current_item == searched_item:
                    return True
                else:
                    queue += graph[current_item]
                    searched.add(current_item)
    return False
```

Breadth-first search cannot be done recursively, needs to be done iteratively with a queue. For a strict per-level BFS:

```
while(queue):
  for i in range(len(queue)):
    <process and add new stuff to queue>
```

2.5.2 Depth-first search (DFS)

Ill-suited for finding a shortest path. For topological sort.

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    if start in visited:
        return
    visited.add(start)
    for neighbor in graph.get(start, []):
        dfs(graph, neighbor, visited)
    return visited
```

2.5.3 Dijkstra's Algorithm

For finding the shortest path from `start` to `end` in a weighted graph. Only works on graphs with no cycles and where all the edges are nonnegative.

1. Initialize: create two hash tables: one that maps each node to the sum of its shortest paths from `start` (initialized as ∞)

and one that maps each node to its parent/preceding node on the shortest path. Set the path costs of `start`'s out-neighbours (and `start` as their parent). Create a set for processed nodes. 2. Loop until all nodes are processed: 2a. Find the node with currently the shortest path from `start`. The key idea of the algorithm is that there is no other shorter path to this node.[9] 2b. For all this node's out-neighbors of this node, if the route via this node is shorter, update that out-neighbours shortest path cost and set it's parent to this node. Add the node to the set of processed nodes. 4. Calculate the final path by following the parent nodes back to `start`.

Assuming the graph is given as a nested graph:

```
costs = {}
parents = {}
# ...initialize costs and parents here
processed = set()

node = find_lowest_cost_node(costs)
while node is not None:
   cost = costs[node]
   neighbors = graph[node] # A nested hash table
   for n in neighbors.keys():
      if costs[n] > cost + neighbors[n]:
         costs[n] = cost + neighbors[n]
         parents[n] = node
   processed.add(node)
   node = find_lowest_cost_node(costs)

# THERE IS A BETTER VERSION OF THIS THAT DOESN'T LOOP THROUGH ALL NODES EACH TIME, NAMELY WITH
    PRIORITY QUEUES. WILL UPDATE LATER
def find_lowest_cost_node(costs):
   lowest_cost = math.inf
   lowest_cost_node = None
   for node in costs: Goes through each node
      cost = costs[node]
      if cost < lowest_cost and node not in processed:
         lowest_cost = cost . . . sets it as the new
         lowest-cost nodelowest_cost_node = node
   return lowest_cost_node
```

2.5.4 Trees

DFS on BST Depth-first search on BST:

```
def dfs(root):
   if not root:
      return
   dfs(root.left)
   print(root.val) # Inorder traversal
   dfs(root.right)
```

In preorder traversal, the operation is done before recursing into the left subtree. In postorder traversal, the operation is done after recursing into the right subtree.
Recursive is almost always easiest, but can also be implemented iteratively with a stack. The different forms of iterative traversal (in order of intuitiveness):

- Preorder: at each current node, do the operation, push the right child onto the stack, set the left child as the current node, and repeat. If the current node is null, pop the current node from the stack. Repeat until the stack is empty.
- Inorder: at each current node, push that current node to the stack, set the current node to the left child, and repeat. If the current node is null, pop the current node from the stack, do the operation, and set the right node as current node. Repeat until the stack is empty.
- Postorder: at each current node, push its right child on the stack (mark as unvisited), then its left child (mark as unvisited), then itself (mark as visited), then set its left child, then pop the current node from the stack and repeat. In order to avoid pushing a right child twice, we keep track of a separate stack of booleans called visisted with the same structure and ordering as the node stack. If we pop a node from the stack that is visited, we only do the operation and don't push children, because those have been processed already.

[9] This is also why negative weights break the algorithm; the algorithm relies on the fact that, once you process a node, there shouldn't be a shorter path to it, but with negative weights, a shorter path can still turn up. The Bellman–Ford algorithm can deal with negative weights.

A $O(n \log n)$ sorting algorithm for arrays is to first build a BST and then depth-first adding the values to the array like above.

The least common graph representation is the adjacency matrix, a 2D array in which each cell represents an edge from node i to node j, and the value indicates whether the edge exists. Takes $O(V^2)$ space. Uncommon on interviews.

2.6 Backtracking

Backtracking algorithms (or rather, metaheuristics) find solutions to some computational problems, notably constraint satisfaction problems, by incrementally building candidates to the solutions, and abandoning a candidate ("backtracking") as soon as they determine that the candidate cannot possibly be (part of) a valid solution. Backtracking can be applied only for problems which can have "partial candidate solutions" and an efficient test of whether it is invalid. If that's the case, backtracking is often much faster than brute-force enumeration of all complete candidates. Conceptually, the partial candidates can be represented as nodes of a search tree. Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further. A backtracking algorithm traverses this search tree recursively, from the root down, in depth-first order.

2.7 Dynamic programming

Dynamic programming (DP) breaks a problem down into subproblems and solves each of those while storing the results. Can replace brute-force recursion when there is a lot of repeated work being done. It is based on the idea of *memoization*: calculating the result of a subproblem and storing it (in a hash table or array) for lookup later. Two variations:

- Top-down DP still uses recursion; just with caching added; it starts with the main problem and works down.
- Bottom-up DP is iterative and starts with base cases and then works towards the main problem. Often called "true DP", and likely has better space complexity than top-down. In 1D dynamic programming, if the answer to subproblem i only depends on only a few answers to subproblems $i-1, i-2, ...$, you can save space by only memorizing those last couple of answers, rotating them in a constant-size array.

Giveaways that a problem should be solved with dynamic programming are: asking for the minimum or maximum of something, having to make decisions that may depend on previously made decisions (very typical of a problem involving subsequences), when you're trying to optimize something given a constraint.

Characteristics of problems that can be solved with dynamic programming:

- Optimal substructures: the optimal result relies on the optimal result of one or more substructures.
- Overlapping sub-problems
- It can be broken down into discrete subproblems that don't depend on each other

Dynamic programming only works when each subproblem is discrete; when it doesn't depend on other subproblems.

2.8 Sliding window algorithms

Sliding window algorithm use two pointers demarcating an interval (of fixed or variable size) and slide those across an array.
Two pointers algorithms use two pointers and generally only focus on the two elements pointed at. Sliding window algorithms are sort of a subset of two pointers algorithms, though the window in between matters.

2.9 Greedy algorithms

A *greedy algorithm* makes the locally optimal ("greedy") choice at each step, reducing each given problem into one smaller problem, i.e. it never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. While greedy is optimal for some problems (e.g. classroom scheduling problem), for many problems, a greedy strategy does not produce an optimal solution but can heuristically yield a locally optimal solution faster (e.g. set covering problem). Greedy algorithms are a subset of *approximation algorithms*.

2.10 Decision problems

An *optimization problem* ask the minimum or maximum of something. A *decision problem* has a yes-or-no answer. An optimization problem ("What is the shortest path?") can be repurposed as a decision problem ("Is there a path of length

5?"). All problems discussed in this subsection are decision problems. The class of (decision) problems of which the solution can be found in polynomial time (and, naturally, verified in polynomial time) is called P. For some problems, there is no known way to find an answer in polynomial time, but of which answers can be verified in polynomial time; these are in class NP (standing for "nondeterministic polynomial time"). P is a subset of NP. The *P vs. NP problem* is a famous, unsolved computer science problem that asks whether every problem that is quick to verify is also quick to solve, i.e., whether P = NP.

A *reduction* happens when you reformulate a problem you can't solve into one you do know how to solve. Problem A is reducible to problem B, if an algorithm for solving problem B efficiently could also be used as a subroutine to solve problem A efficiently. If so, solving A cannot be more complex than solving B. A problem is NP-hard if any problem in NP can be reduced, in polynomial time (because else the reduction may be the bottleneck), to that problem. As a consequence, a polynomial time solution for any one NP-hard problem gives us a polynomial time solution for every problem in NP. A problem is NP-complete if it is both NP and NP-hard.

2.10.1 Satisfiability problem

The *(boolean) satisfiability problem*, often abbreviated as SAT or B-SAT, is an NP-complete problem that asks whether there exists an assignment of a given propositional logic formula's boolean variables (i.e. consistently replacing them with `true` or `false`) that makes the formula evaluate to `true`. There's no efficient algorithm; you have to fill the entire truth table (in which each row is an assignment) in $O(2^n)$ time.

SAT is the first problem that was proven to be NP-complete (the Cook–Levin theorem), meaning that all problems in the complexity class NP are at most as difficult to solve as SAT.

2.10.2 Traveling salesman problem

The *traveling salesman problem*, often abbreviated as TSP, is an NP-hard problem.

2.10.3 Set cover problem

Q: Given a set of elements $\{1, 2, \ldots, n\}$ (the *universe*), and a collection S of m subsets of universe whose union equals the universe, identify a smallest sub-collection of S whose union equals the universe.

A: The the set cover problem is an NP-hard problem. A greedy algorithm for polynomial time approximation chooses, at each step, the set that contains the largest number of uncovered elements

```
while uncovered_elements:
   best_subset = None
   elements_covered = set()
   for subset, elements_in_subset in stations.items():
      covered = uncovered_elements & elements_in_subset # Intersection
      if len(covered) > len(elements_covered):
         best_subset = subset
         elements_covered = covered
   uncovered_elements -= elements_covered # Set subtraction
   chosen_subsets.add(best_subset)
```

3 Problems

3.1 Knapsack

Q: Given a set of items, each with a weight and a value, determine a subset of items that maximizes the total value while satisfying the weight constraint.

A: Evaluating all subsets is $O(2^n)$. The greedy strategy of picking the most valuable thing at every step is suboptimal. 2D dynamic programming gives an optimal solution by solving the problem for smaller knapsacks and subsets of items. You row-wise fill a table of subproblems with as columns the max weights from 1 up until the actual max weight[10], and as rows subsets of items, with each row adding another item (the order doesn't matter, but it needs the same order as the order of the items in the weight vector). Every cell is calculated as follows:

[10] These can likely be integers, but if you have decimal weights, you need to have a finer column granularity, i.e. a column step size that accounts for this.

$$M_{i,j} = \max(M_{i-1,j}, M_{i,j-\mathbf{w}_i})$$

subject to $j - \mathbf{w}_i \geq 0$, where $M_{i,j}$ is the cell at row i and column j in matrix M, and $\mathbf{w}_i$ is the weight of the i^{th} least heavy item. In other words, we either the take the previous max value (i.e. without the item under consideration), or we take the value of the item plus the max value of the remaining space (as long as that's at least 0).

3.2 Longest Common Substring

Q: Determine the longest common *substring* (i.e. consecutive characters) between two given strings.

A: Done with 2D dynamic programming, with the letters of the two words along the axes of the grid. Initialize the grid with all-zeroes, then loop (row-wise or column-wise, doesn't matter) and only change the value of a cell if:

```
if word_a[i] == word_b[j]:
   cell[i][j] = cell[i-1][j-1] + 1
```

For most dynamic programming problems, the answer is in the last cell, but the solution here is the largest number in the grid, which isn't necessarily in the last cell.

3.3 Longest Common Subsequence

Q: Determine the the longest common *subsequence* (doesn't need to be consecutive) between two given strings.

A: Done similarly to longest common substring, but you set each cell as:

```
if word_a[i] == word_b[j]:
   cell[i][j] = cell[i-1][j-1] + 1 // Same as longest common substring
else:
   cell[i][j] = max(cell[i-1][j], cell[i][j-1])
```

The answer is in the last cell.

3.4 Linked List Cycle Detection

Q: Determine whether there is a cycle in a linked lists in $O(n)$ time and $O(1)$ space.

A: A two pointers algorithm. Each iteration, the fast pointer moves by two and the slow moves by one; when they meet, there's a cycle. This is known as Floyd's fast and slow pointers algorithm or Floyd's tortoise and hare algorithm.[11]

3.5 Contains Duplicate

Q: Given an integer array, determine if any value appears at least twice in the array or if every element is distinct.

A: Loop once through the array, adding each element to a hash set and checking whether it's already in there. Time: $O(n)$ Space: $O(n)$

3.6 Contains Duplicate II

Q: Given an integer array and integer k, determine if any value appears at least twice in the array and those elements are at most k indices apart, i.e. $||i - j|| \leq k$.

A: Same as the answer to Contains Duplicate, but instead we use a sliding window and delete the $i - k$'th number in the array from the hash set. Time: $O(n)$ Space: $O(\min(n, k))$

3.7 Contains Duplicate III

Q: Same as the answer to Contains Duplicate II, but instead of equal values, we're looking for two numbers that differ at most by t, i.e. $||\mathbf{x}_i - \mathbf{x}_j|| \leq t$.

A: Same as the answer to Contains Duplicate, but instead of a hash set, we use buckets with width $w = t + 1$. The bucket

[11] The algorithm is named after Robert W. Floyd, who was credited with its invention by Donald Knuth. However, the algorithm does not appear in Floyd's published work, and this may be a misattribution.

of any value v can be determined with $\lfloor \frac{v}{w} \rfloor$. We return `true` if, for any value, there already exists a value in the same bucket, or whether there exists a value in the two adjacent buckets with values close enough together. Because there can be at most one value per bucket (or else we can return `true`), we can use a hash map for the buckets that maps bucket ID to its singular value. Of course, we remove values outside the window. Time: $O(n)$ Space: $O(\min(n, k))$

```
def containsNearbyAlmostDuplicate(nums, k, t):
   if t < 0:
      return False
   buckets = {}
   w = t + 1 # Increment by 1 to handle the range correctly
   for i in range(len(nums)):
      bucketIndex = self.getID(nums[i], w)
      if bucketIndex in buckets:
         return True # There's already a value in the same bucket
      if bucketIndex - 1 in buckets and abs(nums[i] - buckets[bucketIndex - 1]) < w:
         return True # There's a value in a neigbouring bucket that's close enough
      if bucketIndex + 1 in buckets and abs(nums[i] - buckets[bucketIndex + 1]) < w:
         return True
      buckets[bucketIndex] = nums[i]
      if i >= k:
         del buckets[self.getID(nums[i - k], w)]
   return False
```

There's also a $O(n \log n)$ answer that uses a BST as a data structure.

3.8 Top K Frequent Elements

Q: Given an integer array and an integer k, return the k most frequent elements. You may return the answer in any order.

A: Use quickselect. Use the code in 2.4.4, but tweak it to return the k largest elements.

3.9 Group Anagrams

Q: Given an array of lowercase strings (without numbers), group the anagrams together. You can return the answer in any order.

A: For each string, count its characters and create a character count string from those counts, e.g. `abbccc` will be represented as `#1#2#3#0#0#0...#0`. Use these character count string representations as keys in a hash map that maps them to lists of the strings with the same character counts. Time: $O(nk)$, space: $O(nk)$, where n is the length of the string array and k is the maximum length of a string in the array.

A slower solution is to sort every string and use those as keys. Time: $O(nk \log k)$, space: $O(nk)$

3.10 Product of Array Except Self

Q: Given an integer array `nums`, return an array with the value at each index `i` equal to the product of all the elements of `nums` except `nums[i]`, without using the division operation.

A: Create an array `left` and make a pass over the input array from left to right, cumulatively multiplying the input array (so `left[i] = nums[i] * left[i-1]`), and do the same from right to left in array `right`. Then, each element in the output array is `left[i-1] * right[i+1]`. Time: $O(n)$, space: $O(n)$.

A $O(1)$ space complexity optimization is to use the output array as the `left` array, and to only maintain an integer product for the right part:

```
def productExceptSelf(nums):
   result = [0] * len(nums)
   result[0] = nums[0]
   for i in range(1, len(nums)):
      result[i] = nums[i] * result[i-1]
   prod = 1
   for i in reversed(range(1, len(nums))):
      result[i] = result[i-1] * prod;
      prod *= nums[i];
   result[0] = prod;
   return result
```

3.11 Longest Consecutive Sequence

Q: Given an unsorted array of integers, return the length of the longest consecutive elements sequence in $O(n)$ time.

A: Put all numbers in a hash set (for $O(1)$ lookup), loop through all the numbers (either through the array or the set, the order doesn't matter), and for each number `num`, check if `num-1` is in the set. If it is, this number is not the start of a consecutive sequence and we skip. If it is, count the length of the consecutive sequence that starts with this number by incrementing it and checking whether this increment is in the set. Return the maximum of such a length.

```
def longestConsecutive(nums):
   longest_streak = 0
   num_set = set(nums)

   for num in num_set:
      if num - 1 not in num_set:
         current_num = num
         current_streak = 1

         while current_num + 1 in num_set:
            current_num += 1
            current_streak += 1

         longest_streak = max(longest_streak, current_streak)

   return longest_streak
```

This is $O(n)$ time and space, because each element is processed a constant number of times; the inner while loop only runs for each consecutive sequence.

3.12 BFS on Matrix Graphs

Q: A BFS algorithm for finding the length of the shortest path from the top left to the bottom right in a given matrix graph.

A:

```
def bfs(grid):
   ROWS, COLS = len(grid), len(grid[0])
   visit = set()
   queue = deque()
   queue.append((0, 0))
   visit.add((0, 0))

   length = 0
   while queue:
      for i in range(len(queue)):
         r, c = queue.popleft()
         if r == ROWS - 1 and c == COLS - 1:
            return length

         neighbors = [[0, 1], [0, -1], [1, 0], [-1, 0]]
         for dr, dc in neighbors:
            if (min(r + dr, c + dc) < 0 or
               r + dr == ROWS or c + dc == COLS or
               (r + dr, c + dc) in visit or grid[r + dr][c + dc] == 1):
               continue
            queue.append((r + dr, c + dc))
            visit.add((r + dr, c + dc))
      length += 1
```

$O(r * c)$

3.13 DFS on Matrix Graphs

Q: A recursive backtracking DFS algorithm for counting the number of unique, nonlooping paths from the top left to the bottom right in a given matrix graph.

A:

```
def dfs(grid, r, c, visit):
```

```
    ROWS, COLS = len(grid), len(grid[0])
    if (min(r, c) < 0 or
        r == ROWS or c == COLS or
        (r, c) in visit or grid[r][c] == 1):
        return 0
    if r == ROWS - 1 and c == COLS - 1:
        return 1

    visit.add((r, c))

    count = 0
    count += dfs(grid, r + 1, c, visit)
    count += dfs(grid, r - 1, c, visit)
    count += dfs(grid, r, c + 1, visit)
    count += dfs(grid, r, c - 1, visit)

    visit.remove((r, c))
    return count

print(dfs(grid, 0, 0, set()))
```

$O(4^{r*c})$