CodeFights Tutorials - https://goo.gl/vbGhW7

Arrays

Interview Essentials

Arrays are one of the most basic data structures, but they make for some very hard interview questions! It’s imperative that you know the array methods in your chosen interviewing language very well, because interviewers will definitely be judging you on this.

An array is a data structure that contains a group of elements. The most basic implementation of an array is a static array, in which the array has a fixed size that must be declared when it is initialized. In technical interviews, you’re also going to come across questions about dynamic arrays (similar to static arrays, but with space reserved for additional elements) and multidimensional arrays (arrays in which the individual elements are also arrays).

Static Array

What Is a Static Array?

A random-access data structure that is of fixed size. "Adding" or "deleting" an element from an array requires creating an appropriately sized contiguous chunk of memory, and copying the elements that you want to keep into the new array.

Crucial Terms

- **Size:** The number of elements in the array.

Strengths and Weaknesses

Strengths

- The item lookup by index for arrays is very quick - `O(1)`.
- Arrays are a simple data structure with a very low memory footprint.

Weaknesses

- Arrays have to be implemented in contiguous memory.
- Adding or deleting elements to the array is `O(n)`, where `n` is the number of elements in the array.
- Arrays do not allow for the quick rearrangement of elements.
- Searching in the array for an entry with particular attributes is `O(n)`.

Common Operations Cheat Sheet

Operation	Description	Time complexity	Mutates structure
`append(item)`	Adds `item` to the end of the list	`O(n)`	Yes
`delete(index)`	Removes `item` from list	`O(n)`	Yes

`[index]`	Retrieves element at `index`	`O(1)`	No

Dynamic Array

What Is a Dynamic Array?

A random-access, variable sized list data structure that allows elements to be added or removed. Like a regular array, but with reserved space for additional elements.

Crucial Terms

- **Logical size:** The number of elements being used by the array's contents.
- **Physical size:** The size of the underlying array, counted in either the number of elements it has room for, or the number of bytes used.

Reserved

Reserved

Reserved

A dynamic array with a physical size of 7 elements, logical size of 4 elements.

Strengths and Weaknesses

Strengths

- On average, inserting elements to the end of the dynamic array is quick.
- Item lookup by index is `O(1)`.

Weaknesses

- Dynamic arrays must be implemented in contiguous memory.
- Dynamic arrays have inconsistent run times for adding to the array, so they're not good for real-time systems.
- Dynamic arrays do not allow for the quick rearrangement of elements.

In Interviews, Use Dynamic Arrays When...

- You'll need to add or delete information from the array.
- An (occasional) long operation won't significantly affect your program's performance.

Common Operations Cheat Sheet

Operation	**Description**	**Time complexity**	**Mutates structure**
`append(item)`	Adds `item` to the end of the list	`O(1)` (amoritized)	Yes
`delete(index`	Removes `item` from list	`O(n)` in general	Yes

)			
`delete(index)`	Removes last `item` from list	`O(1)` from end of list	Yes
`[index]`	Retrieves element at `index`	`O(1)`	No

Multidimensional Arrays

What Is a Multidimensional Array?

A multidimensional array is a random-access data structure in which the data is accessed by more than one index. Some languages, such as C#, have built-in support for multidimensional arrays, while others support multiple indices by making an "array of arrays". A multidimensional array that needs `N` numbers to reference a piece of data is called a `N`-dimensional array, or `N`D array.

While multidimensional arrays of any size are possible, it's most important for you to be familiar with 2D arrays. You will often see 2D arrays used to implement:

- Matrices `M`, where `M[i][j]` represents the number in row `i` and column `j`
- Game boards `board` (such as chess or checkers boards), where `board[row][col]` represents the state of the board at location `(row, col)`
- Maps, where the map is divided into cells.

Crucial Terms

Different languages have different internal representations of multidimensional arrays. When implementing a 2D array as an "array of arrays" (i.e. `array[i]` is itself an array), there is no technical reason why `len(array[i]) == len(array[j])`. If the arrays have different lengths, we call them *jagged arrays*. When discussing multidimensional arrays, we will mean *regular* (or *rectangular*) arrays, in which each dimension has a fixed length.

- **Dimension:** the number of indicies needed to locate a single element in the array.
- **Size:** A tuple (d_1, d_2, ..., d_N) where d_i is the number of distinct indices for index `i`.
-
- **Array of arrays:** The multidimensional array `A` is represented as an array, where the entries of the array `A[i]` are arrays. Element `A[i][j]` is the `j`th element in the array `A[i]`. The arrays `A[i]` may be dynamic or static.
- **Native multidimensional array:** The multidimensional array `A` is allocated as a single block of memory. Generally, the arrays are constrained to be *regular* or *rectangular*, and you cannot alter single columns or single rows of `A`.

Strengths and Weaknesses

Strengths

- The item lookup by index for multidimensional arrays is `O(1)`.
- You can use multiple indices that make sense in the problem domain (for example, using rows and columns for a chess board), making code easier to read and maintain.
- It's easy to iterate over every element stored in a multidimensional array.

Weaknesses

- Multidimensional arrays do not allow for the quick rearrangement of elements.
- It requires a long time to change the size of a multidimensional array.

Common Operations Cheat Sheet

Operation	Description	Time complexity	Mutates structure
`append(item)`	Adds `item` to the end of the list	implem. dependent	Yes
`delete(index)`	Removes `item` from list	implem. dependent	Yes
`[index1, index2,..,indexN]`	Retrieves element at `(index1, index2,..,indexN)`	`O(1)`	No
`size()`	Returns the size of the array `(d1, d2,..,dN)`	`O(1)`	No
`dim()`	Returns the dimension of the array	`O(1)`	No

Linked Lists

Interview Essentials

A linked list is a sequential-access data structure used for storing *ordered* elements. They prioritize quick and easy data insertion and deletion over item lookup. All linked lists are collections of `node` data structures that are connected by `pointers` - that's where the "link" in "linked list" comes from.

Linked list questions are short and simple for an interviewer to ask, but they have complex solutions that really show off your problem-solving and coding skills. This means that they're very common in technical interviews, so it's important to be comfortable with using this data structure. You'll use linked lists to solve questions in which having quick insertion and deletion is important, and when it's not important to be able to have random access to elements.

Types of Linked Lists

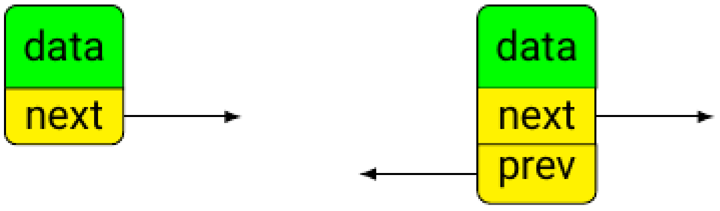
We primarily see two types of linked lists:

- **Singly linked list:** (most common)
- Each node contains the `data` we want to store, as well as a `next` pointer to the next node in the linked list. Singly linked lists support traversal in only one direction.
- *Common interview question:* Find a way to access earlier information in the list.
- **Doubly linked list:**
- Each node contains the `data` we want to store, as well as `next` and `previous` pointers to the node's neighboring elements. Doubly linked lists support traversal in both directions.
- *Common interview question:* Implement a linked list.

Doubly linked lists require more space per node and their basic operations are more expensive than for singly linked lists. However, they are often easier to manipulate than singly linked lists because they allow for fast and easy sequential access to the list in both directions.

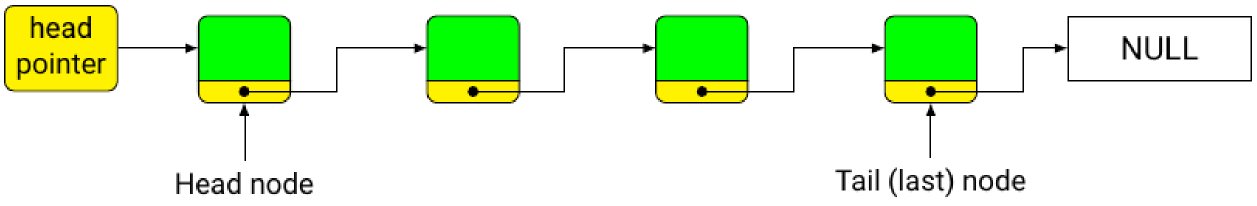
Essential Vocab

- **Pointer:** The memory location of a data structure.
- **Node:** A data structure with two fields. The `data` field contains the information we want to store, and the `next` field, which is a pointer to the next node in the linked list. A linked list can be thought of as a series of nodes. Nodes for doubly linked lists need `next` and `previous` pointers.



A *node* for a singly linked list and a doubly linked list

- **Head:** A pointer to the first node in the linked list. All nodes are accessible from the head node by visiting the chain of previous nodes.
- **Tail:** The last element in the linked list. The `next` pointer of the tail element points to `null` to indicate the end of the list.
- **Size:** The number of elements in the linked list.



A singly linked list with a *size* of 4

Strengths & Weaknesses

Linked lists are often compared to arrays, a data structure that provides random access to elements, meaning that their strengths and weaknesses are frequently compared to those of arrays as well.

Strengths

- Linked lists store ordered lists of data in nodes.
- Linked lists allow for quick (`O(1)`) addition and removal of elements (advantage over an array).
- Linked lists can be resized dynamically.
- The size of a linked list is only limited by the amount of available memory.

Weaknesses

- Linked lists can require more space in memory than arrays do.
- Linked lists are sequential-access instead of random-access, meaning that accessing the i^{th} element can be slow because you must iterate over `i` elements to get there.

In Interviews, Use a Linked List When...

- You don't know the number of elements in your list before you start.
- You have lists that need to grow (e.g. accepting input from files, the internet, other streams, etc.)
- You need to implement a more sophisticated data structure like a heap, stack, or queue.

Common Operations Cheat Sheet

Operation	**Description**	**Time complexity**	**Mutates structure**

`find(value)`	Returns a pointer to the `node` containing `value` in the data field, or NULL if it isn't in the list. This method can be used for `membership checking` as well.	*O(N)*	No
`find(index)`	Returns a pointer to the `index`th node from the head pointer, or NULL if `index` is longer than the length of the list.	*O(index)*	No
`addBeginning(value)`	Adds a new node with a data field containing `value` to the beginning of the list	*O(1)*	Yes
`addAfter(value, n)`	Adds a new node with a data field containing `value` after node `n`	*O(1)*	Yes
`remove(value)`	Removes first instance of value from list	*O(N)* if we need to find the element.	Yes
`removeNextElement(n)`	Removes node after `n`	*O(1)*	Yes
`removeThisElement(n)`	Removes node `n`	*O(1)* on doubly linked-list	Yes

Hash Tables

Interview Essentials

Hash tables are a must-know data structure for technical interviews. This data structure is used to store an *unordered*collection of `(key, value)` pairs, where the `key` has to be unique. Item lookup by key, inserting a new item, or deleting an item are all fast operations (approximately `O(1)`).

Because they give you quick and cheap insertion, deletion, and lookups, you can use hash tables to solve many different types of interview questions. Hash tables are great for things like determining whether an element belongs to a collection, solving problems with invariants such as anagrams, and for solving problems where there is a unique, non-arbitrary identifier that you can use as a key (for instance, a person's phone number or ID number).

A hash table is implemented with a function *h(x)*, called a *hash function* or simply the *hash*. The hash function takes keys as inputs and returns a *bucket index* where the information is stored. The hash table becomes less efficient as the individual buckets store more information. A hash table is best suited to *sparse* problems, roughly where the actual number of keys used is smaller than the conceivable set of keys. For example, there are 10^{10} 10-digit phone numbers, only a small fraction of which are actually in use.

Crucial Terms

- **Key:** A unique identifier for each value.
- **Value:** The information we want to access (generally)

Note: You don't have to worry about the following terms to simply *use* a hash table, but you do need to know them to *implement* one (or to answer interview questions about how they're implemented!).

- **Bucket:** Depending on the implementation, this is either where the `value` data is stored, or a pointer to the `value`data is stored.
- **Sparseness:** The property that the number of actual keys used is much smaller than the possible keys used. (e.g. there are an infinite number of strings that could be email addresses, and only a finite number in use.)
- **Hash** or **hash-function:** A function *h(key)* that takes the `key` and returns an index for the "bucket" containing the information that we want. The number of buckets is much smaller than the number of possible keys. Note this means that multiple keys are assigned to the same bucket. If there are *K* possible keys, and *B* buckets, the average number of possible keys assigned to a given bucket is *K/B*.
- **Collision:** When two keys `k1` and `k2` have the same hash (i.e. `h(k1) == h(k2)`) we say the keys *collide*, so they are assigned to the same bucket. There are different ways of storing different elements whose keys have the same hash.
- **Good hash:** A *good hash* is a hash function that provides few collisions. Unfortunately there is no universally good hash; the hash function has to be good for the distribution of keys that actually occur in your problem.

Strengths & Weaknesses

Hash tables combine random access ability with quick insertion and deletion, which makes it an extremely flexible and useful data structure. If you’re doing something other than storing keys and values, or if you need to sort elements or iterate efficiently through them, though, you should use a different data structure.

Strengths

Note: These features assume the hashtable is sparse.

- Hash tables have extremely fast lookup by key (`O(1)`).
- Insertion and deletion of data is also quick (`O(1)`).

- In a hash table, you can use the keys as the data, and use that to check if we've seen an element before. A hash table used this way is usually called a *set*.

Weaknesses

- Hash tables have no notion of order.
- Hash tables cannot match "nearby" keys, or keys that share the same prefix. So a hash table wouldn't be a good choice for checking for words that began with a certain prefix (a trie would be a better choice in that case).
- Lookup by value (instead of by key) is `O(n)`.
- A hash table loses its strengths when the amount of data in a single bucket becomes large. Lookup becomes `O(B)`, where `B` is the number of things in the bucket.

In Interviews, Use Hash Tables When...

- There is a unique identifier that you use for lookup.
- *Examples:*
    - Caller ID: Phone numbers are unique and can be used as a key to quickly retrieve a person. Note that we would **not** use a hash table that found a phone number given a person's name, since many people have the same name.
    - Car owner information: Using the license plate as a key, you can look up the owner of the car (the value).
    -
- You need to quickly determine if an element belongs to a collection. In these cases, you can represent the collection as a hash table with the elements as keys.
- *Example:*
    - A Scrabble checker: If we want to determine if `word` is allowed, we could create a hash table `valid_words`where the keys were the words. We would set `valid_words[w] = 1` for all valid words, so checking if `word in valid_words` is an `O(1)` operation.
    -
- You're dealing with problems about invariants.
- *Example:*

    - To see if a word has an anagram, we can sort the letters of the string, and use it as a `key` in a hash table. (This method yields the same `key` for any two words that are anagrams of one another). The `value` can be a dummy value (if you're interested in the "yes/no" question of whether a word has an anagram) or an array of strings with the same key (if you are looking for the actual words).

Common Operations Cheat Sheet

The `O(...)` times listed below are the answers typically expected by interviewers. This assumes that the none of the buckets has a large number of items it it. Items listed as `O(1)` are more accurately `O(B)`, where `B` is the size of the bucket assigned to by the hash function.

Operation	**Description**	**Time complexity**	**Mutates structure**
`hash[key] = value`	Adding a new `(key,value)` pair to the hash table	`O(1)`	Yes
`del hash[key]`	Removes `(key, value)` from the hash table	`O(1)`	Yes
`key in hash`	Lookup whether `key` is in hash table	`O(1)`	No

Note: Lookup by value is not supported directly in a hash table. Instead, you'd iterate through all the keys and check for the value you were looking for (an `O(n)` procedure).

Trees: Basic

Interview Essentials

A tree is a data structure composed of parent and child nodes. Each node contains a value and a list of references to its child nodes. Trees borrow a lot of language from nature (each tree has a `root` node and `leaf` nodes) and from family trees (there are `parent` and `child` nodes).

Tree traversal problems are extremely common in technical interviews, so you need to be very familiar with how to do so. You'll also run into situations in which you need to implement a tree yourself. Trees are particularly useful for storing sorted data that needs to be retrieved quickly, or for representing hierarchical data.

Crucial Terms

The tree is a collection of **nodes** that contain the data we want to store. In addition, the nodes have a collection of pointers to other nodes. The nodes referenced by the pointers inside `X` are referred to as the **children** of `X`.

- **Child:** The children of node `X` are the nodes `X` has a pointer to.
- **Parent:** The parent of a node `X` is the node that has `X` as a child.
-
- These two definitions tell us that if `X` is a parent of `Y`, then `Y` is a child of `X`.
- **Siblings:** A group of nodes that have the same parent node.
- **Descendant:** A node `Y` is a descendent of `X` if, starting at `X`, you can reach `Y` by following a series of child pointers. (Children are included as descendants of nodes.)

- **Ancestor:** A node `X` is an ancestor of `Y` if `Y` is a descendant of `X`.

In addition to terms describing the nodes in the tree, there are also some terms about the different parts of the tree.

- **Root:** The top node of a tree, and the only node that does not have a parent node. All other nodes in our tree are descendants of the root (and the root is an ancestor of all other nodes in the tree).
- **Level:** The *level* of a node `X` is the number of child pointers that need to be followed to get to `X` from the root. All the nodes at level 1 are children of the root, the nodes at level 2 are children of nodes at level 1, et cetera.
- **Leaf:** Nodes that do not have any child nodes.
- **Height of a tree:** The maximum level in the tree (i.e. the level of the node furthest from the root).
- **Depth:** Confusingly, this is a synonm for the height of a tree! Computer scientists typically draw the root node at the top of the page and work their way down, so increasing levels means going down the page.
- **Branching factor:** The maximum number of children that any node has.

There are many different types of trees that have their own terminology associated with them (such as Red-Black Trees, and AVL trees). Many of the different types of trees differ in their approach to *balancing* the tree. *Balancing* refers to inserting entries into a sorted tree in such a way that minimizes the tree's depth. While many of these trees are quite esoteric for an interview, you must know the terminology for a binary search tree (BST). A BST is a sorted tree with a branching factor of two (i.e. every node has at most two children, referred to as the `left` and `right` nodes due to the way they are drawn). A BST requires that the `left` node has a value smaller than its parent, while the `right` node has a value larger than its parent.

Tree Basics

For a collection of nodes to be a non-empty tree, they need to satisfy 4 conditions:

1. There needs to be exactly one root node (i.e. one node with no parent).

2. Each node that is not the root node has only one parent node (i.e. no node is a child of two or more nodes)
3. You need to be able to reach each node by starting at the root and following a sequence of "child" pointers.
4. A node at level `x` cannot have a child at a level less than `x`.
5. (In fact, all children of a node at level `x` have a level of `x+1`, because children of a node at level `x` are one extra step from the root.)

Strengths & Weaknesses

Strengths

- Trees are very memory-efficient and do not use more memory than they need to.
- Trees are more flexible than simple arrays because we can *balance* the tree (find the best organization for the expected use of the data).
- Trees can naturally grow to hold an arbitrary number of objects.
- By keeping the height of the tree small in a sorted tree, searches can be very fast (searches start at the root, and in a sorted tree and look at `O(tree height)` nodes when searching).
- The heirarchy of a tree is reflected in many problems we would try and model (such as representing a company heirarchy, or files on a computer, or breaking down expenses by categories followed by subcategories and line-items).
- *Decision trees* are a special type of tree used to model the effects of different choices. The nodes are the state of the system; a node `Y` is a child of `X` if it is possible to make a single decision from state `X` to transform to state `Y`. Decision trees are often used when analyizing strategy games, such as Chess.

Weaknesses

- Is harder to debug.
- Manipulations can require through understanding of recursion.
- Have to make decisions how to balance the data based on expected use case, and rearranging the tree completely (e.g. sorting on a different attribute) is expensive. For example, it wouldn't be the best data structure for storing

information about people if you expected users to sort frequently by either name or age.

In Interviews, Use Trees When...

- You have data that is sorted in some way, and you want to do a lot of searches on it.
- You need to manage objects that are clustered or grouped by some attribute. File systems are a common hierarchy in which the nodes are either files or directories.
- You are trying to implement a search strategy such as backtracking.

Common Operations Cheat Sheet

The time complexity quoted below are specific to a *balanced Binary Search Tree*. When inserting and deleting node, the time taken for rebalancing the tree is also taken into account. Here `n` is the number of nodes in the tree.

Operation	**Description**	**Time complexity**	**Mutates structure**
`insert(obj)`	Adds `obj` into the BST, while maintaining a balanced tree.	`O(log n)`	Yes
`delete(obj)`	Finds and deletes `obj` from BST, while maintaining a balanced tree.	`O(log n)`	Yes
`search(obj)`	Determines if object is in tree	`O(log`	No

		`n)`	
`get(n)`	Selects the `n`th highest object	`O(log n)`	No
`rank(rand_obj)`	Returns the number of nodes in the BST that are less than or `rand_obj`. `rand_obj` does not have to appear in the tree.	`O(log n)`	No
`<iterator>.next()`	Gets the next element, starting at current element	`O(log n)`	No
`flatten()`	Returns a list containing the nodes in sorted order	`O(n)`	Implementation dependent

Heaps, Stacks, Queues

Interview Essentials

- **Heap:** A tree-based data structure in which the value of a parent node is ordered in a certain way with respect to the value of its child node(s). A heap can be either a *min heap* (the value of a parent node is less than or equal to the value of its children) or a *max heap* (the value of a parent node is greater than or equal to the value of its children).
- **Stack:** Operations are performed *LIFO* (last in, first out), which means that the last element added will be the first one removed. A stack can be implemented using an array or a linked list. If the stack runs out of memory, it’s called a *stack overflow*.

- **Queue:** Operations are performed *FIFO* (first in, first out), which means that the first element added will be the first one removed. A queue can be implemented using an array.

Along with being comfortable with implementing heaps, stacks, and queues, it’s important to be clear on the the differences between these data structures. In interviews, it’s common to get questions like, “Explain the differences between a heap and a stack.”

Heaps

What are Heaps?

The purpose of a *min-heap* is to store objects that have a partial order on them. It has a fast (`O(log n)`) method for removing the minimum object from the heap. Min-heaps are useful for calculations where you have multiple minimum computations to perform.

There is an analogous structure called a *max-heap*, which extracts the maximum value from the heap.

A heap is either a max-heap or a min-heap - it can't be both. Given an implementation of a min-heap, a max-heap can be implemented by reversing the comparison between elements. We're only going to discuss min-heap in the rest of this overview.

Heaps are sometimes referred to as *priority queues*. Technically, heaps are actually just one implementation of a priority queue.

Crucial Terms

- **key:** The values that determine the order. If you're storing numbers, the numbers can be the keys. If you're storing more complicated objects, the key is the data field that we're comparing by. Unlike in hash tables, keys in heaps do **not**have to be unique.
- **extract_min:** The method of (quickly) being able to extract the minimum element from the min-heap.

There are some implementation-specific details about heaps (such as the **heap property** and **complete binary trees**) that are useful for understanding how to build a heap from scratch, but they aren't that useful for using heaps to solve interview problems.

Strengths and Weaknesses

Strengths

- A min-heap is able to quickly extract the minimum value on the heap. Repeated extractions from a min-heap into an array will yield a sorted array.

Weaknesses

- There's no convenient way of searching for a particular `key` value in a heap. Entries are only partially ordered; clever use of the heap property can allow for some pruning of searches.

In Interviews, Use Heaps When ...

Heaps are designed to do one specific thing well, so the answer to *when you should use a heap* is repetitive: You use one when you have to do repeated minimum (or maximum) extractions. The problems where you would want to do this might look quite different from each other, however. Below we have selected some examples of common interview questions that benefit from heaps.

- **Finding the minimum distance between two nodes in a graph:**
- The standard approach to this problem is to use Dijkstra's algorithm. One of the key steps in Dijkstra's algorithm is to select the node closest to a node that you have already completed, which is a minimum calculation.
- **Getting the next event that is scheduled to occur:**
- Storing events in a heap with a timestamp as the key gives you a fast way to extract the next event (the event with the smallest timestamp will occur next).
- **Keep track of the median value while streaming:**
- This is the running median problem, where two heaps are maintained: a max-heap for values below the current median and a min-heap for values above the current median. When a new value is inserted, it is placed in the low or high pile as appropriate (and the maximum of the low values or minimum of the low values are extracted as necessary to keep the two heaps sizes' different by at most one element).
- **Find the first `k` non-repeating characters in a string in a single traversal.**

Common Operations Cheat Sheet

The operations described below are for a min-heap implemented using a full binary tree. There are obvious analogs for a max-heap.

Operation	Description	Time complexity	Mutates structure
`insert(key)`	Inserts `key` into heap. Can modify to have a `key`/`value` pair.	`O(log n)`	Yes

`extract_min()`	Returns and removes item with minimum key from the heap,	`O(log n)`	Yes
`peek_min()`	Can view minimum key, doesn't remove it from the heap	`O(1)`	No
`heapify(list)`	Creates a new heap from a list of `n` items. Functionally equivalent to starting with an empty heap and inserting elements one at a time, but has a better run time than the `O(n log n)` of this approach.	`O(n)`	N/A (creates heap)

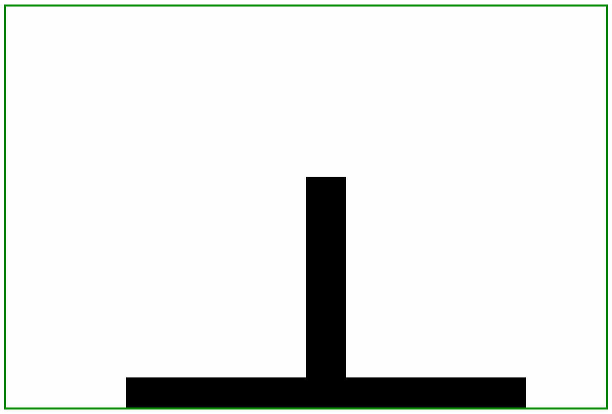
There are other ways of constructing heaps (such as `binomial heaps` and `Fibonacci heaps`) that can improve the asymptotic run time slightly, but that support the same operations.

Stacks and Queues

What Are Stacks and Queues?

Stacks and queues are grouped together because they share many of the same properties. They are both data structures designed to hold elements and return them in a specific order. The difference between the data structures is the order in which items are retrieved.

- **Stacks are First In, Last Out / Last In, First Out:**

- Consider a literal stack of books. When you put a book on the top of the stack, you have to remove it before you can access the books underneath. The book at the very bottom isn't accessible until you have removed all the other books on top of it.
- 
- If elements are added to a stack in the order [A,B,C,D,E], they will be removed in the order [E,D,C,B,A], assuming all the arrivals happened before the removals started.
- **Queues are First In, First Out:**
- In a line (or queue) at a bank, the first person to arrive is the first person to be served. When using a queue to store data, the first elements in are the first elements out.
-
- If elements are added to a queue in the order [A,B,C,D,E], they will be removed in the same order ([A,B,C,D,E]), even if the arrivals and removals are interweaved.
- (The animation suggests that the elements in queue "move forward" when the first one is removed. In efficient implementations of queues, the elements that are not leaving the queue stay in place, so dequeuing can be implemented as an `O(1)` operation.)

Because of the metaphors used, we often talk about the "top" and "bottom" of a stack, but the "front" and "back" of a queue.

Stacks and queues are both *abstract data types*, which means that we are guaranteed to be able to put elements in and remove them in the specified order. You can build a stack or queue out of whatever data structures you like, such as arrays or linked lists, if you provide a way to add elements in the correct order. For example, in Python, the standard list acts like both an array and a stack.

Crucial Terms

Stacks and queues both have an add operation (`push`/`enqueue`) and a remove operation (`pop/dequeue`).

- **push:** (stack) The generic term for adding an object to the "top" of the stack.
- **pop:** (stack) The generic term for removing the object from the "top" of the stack.
- **enqueue:** (queue) The generic term for adding an object to the "back" of the queue.
- **dequeue:** (queue) The generic term for removing an object from the "front" of the queue.

The name of the functions can differ from those shown above, depending on how you're implementing the stack or queue and the programming language you use. When using a Python list as a stack, for example, using `L.append(item)` is the `push` command, but Python does support a `L.pop()` command.

A slick implementation of an object that is both a stack and a queue is to use a doubly linked list, with a reference to the head and tail elements. This doubly linked list could implement all four functions above with `O(1)` time complexity.

Common Operations Cheat Sheet

Because stacks and queues are abstract data types, there is some debate whether time complexity is appropriate for the methods. The debate stems from questions of whether adding and removing from stacks and queues are the only thing they have to do (and time complexity is an implementation detail), or whether the time complexity is intrinsic to the definition of stacks and queues. For the table below, we have chosen the pragmatic option of including the time complexity as part of the definition, since your choice to use a stack or a queue over some other data structure is presumably because they support quick addition and removal of data in the order you want to access it.

If you are using a multipurpose data structure like Python's list (which is a stack *and* a queue), rather than an optimized queue or stack, you will probably have worse performance than indicated below.

Stack Operation	**Description**	**Time complexity**	**Mutates structure**
`push(item)`	Pushes `item` on the "top" of the stack	`O(1)`	Yes
`pop()`	Removes the most recently added item from the stack. (i.e. the item on "top")	`O(1)`	Yes
`peek()`	Accesses the most recently added item in the stack, without removing it	`O(1)`	No

`isEmpty()`	Returns `True` if there are no items on the stack, `False`otherwise	`O(1)`	No

Queue Operation	Description	Time complexity	Mutates structure
`enque(item)`	Puts `item` at the end of the queue	`O(1)`	Yes
`deque()`	Removes the item at the front of the queue	`O(1)`	Yes
`peek()`	Accesses the item at the front of the queue, without removing it	`O(1)`	No
`isEmpty()`	Returns `True` if there are no items in the queue, `False`otherwise	`O(1)`	No

Graphs

Interview Essentials

A graph is an abstract data structure composed of *nodes* (sometimes called *vertices*) and *edges* between nodes. Graphs are a useful way of demonstrating the relationship between different objects.

Some real-life examples of graphs:

- A social network can represent users as nodes, and two nodes are connected if the corresponding users are friends;
- IMDb can represent actors as nodes, and two nodes are connected if the corresponding actors have been in the same movie;
- IMDb can use actors and movies as nodes, and a "movie" node is connected to an "actor" node if the actor appeared in that movie. Movie nodes would not be connected to other movie nodes, and actor nodes are not connected to other actor nodes;
- The power grid can be modeled as a graph between "source nodes" (power stations) and "consumer nodes" (power consumers) with junction nodes representing places where power lines split or merge. The power lines between nodes would be the edges in this example.

Digraphs

In some graphs, the edges are "one way", which means that it's important to distinguish between the "source" or "beginning" of an edge, and the "target" or "end" of the edge. These are called "directed edges", and a graph containing vertices and directed edges are called a *digraph* ("directed graph").

Some examples of digraphs:

- A database can represent the table schema as nodes, and an edge from node X to Y means "table X has a foreign key to table Y".
- Twitter can represent users as nodes, and an edge from node X to Y means "user X follows user Y".
- Webpages can be modeled as nodes on a graph, and an edge from node X to Y means "webpage X has a link on it to page Y".

Graphs and Trees

If you've worked through either of the trees topics, you have already seen examples of graphs. Trees are special types of graphs - specifically, they are connected acyclic graphs with a single root node. Many problems that you have seen when using trees, such as using BFS or DFS, are also graph problems. The general graph versions tend to be a little trickier, because the graph can have cycles in it. Because there can be multiple paths between nodes, another common type of question is to find the "best" or "shortest" path between two nodes.

They are also useful ways of asking questions such as:

- "Who are the actors that have more than 7 degrees of separation from Kevin Bacon?"
- "How many single points of failure are there in the power grid?"
- "Are there any completely isolated groups? If not, how many edges would we have to remove before the graph was disconnected?"
- "What is the largest group of people on this site, where any pair of people from the group are friends?"

Crucial Terms

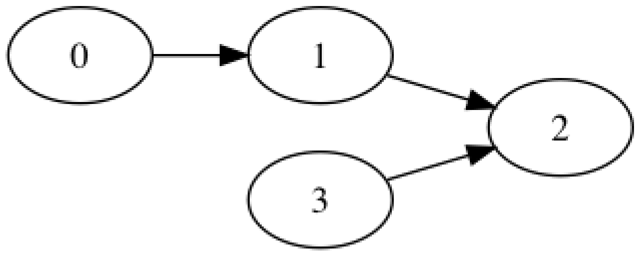
- **Node:** Also referred to as a *vertex*. The set of objects that the graph gives the relationship between.
- **Edge:** An unordered pair of nodes (v,w)(v,w). The nodes vv and ww are referred to as the *endpoints* of the edge. Graphically represented as a connection between nodes vv and ww.
- **Weighted edge:** A weighted edge is an edge in a graph that also has another piece of data associated with the edge, called the *weight*. This weight often represents a cost in terms of time or distance associated with using that edge.
- **Directed Edge:** An ordered pair of vertices, (v,w)(v,w). Graphically represented as an arrow from vv to ww. The node vv is referred to as the *source* or *origin* node, while the node ww is referred to as the *destination* or *target* node. A graph has only directed or undirected edges. An undirected edge

between vv and ww can be simulated by having a pair of directed edges (v,w)(v,w) and (w,v)(w,v).

We will use the term *graph* to mean a graph with undirected edges, and *digraph* for a graph with directed edges.

There are a large number of definitions involving graphs, but most of them are reasonably obvious when looking at the graphical representation of a graph.

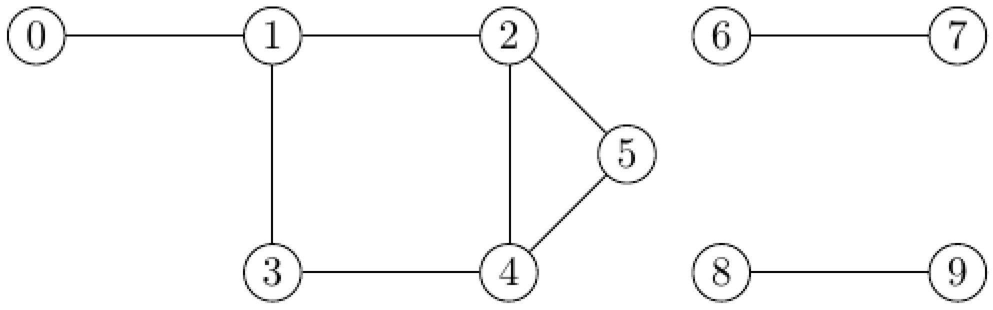
- **Path:** Intuitively, a path between nodes vv and ww is a collection of edges you can use that "start at vv" and "end at ww" without taking any breaks along the way. The technical statement is that a path between vv and ww is a sequence of nodes v_0 = v$v_0=v$, v_1v_1, v_2v_2, \ldots$\ldots$, v_n = w$v_n=w$ such that (v_i, v_{i+1})(v_i,v_{i+1}) is always an edge in the graph. This definition works for both graphs and digraphs.
- **Connected Graph:** A graph is connected if there is a path between every pair of vertices. The intuitive notion is that it is possible to get from any given node to any other node.
- For digraphs, the notion of connectedness is more subtle. In the digram below, is the graph connected?



Here it is not possible to get from node 1 to node 3, nor from node 3 to node 1. A digraph that is connected if you replace all the edges with undirected edges, such as the example above, is called *weakly connected*. A *strongly connected* digraph is a diagram where there is a directed path between any pair of vertices.

- **Induced subgraph:** An induced subgraph of G is a graph that uses a subset N of the nodes of G, and whose edges consist of all the edges that have both endpoints in N. (A simple subgraph is a subset of nodes and a subset of edges, such that each edge in the subgraph is an edge between nodes in the subgraph).

- **Connected component:** Intuitively, the connected components of a graph are the disconnected pieces that the graph is in. The graph below has 3 connected components.



The technical definition is that a connected component CC of GG is a subgraph that is connected, and for which there is no connected subgraph C^\primeC′ that contains CC as a subgraph. To find the connected components of a graph, note that each node belongs to exactly one connected component. Starting at any node, perform either depth-first or breadth-first search.

- **Dense vs sparse:** A graph with nn vertices has at most n(n-1)/2n(n−1)/2 edges. The graph is *dense* if it has a "large fraction" of this maximum number of edges, and *sparse* if it has a "small fraction" of this maximum number of edges. There isn't a technical definition of where the crossover point between dense and sparse is. When choosing graph algorithms, you typically have a choice whether to iterate over nodes or edges, and your choice can have dramatically different computation times depending on whether the graph is dense or sparse.

There are **many** more definitions used to discuss graphs, but often different sources will use slightly different definitions for the same term. Even professional mathematicians will ask clarifying questions when dealing with problems in graph theory to ensure they are using the same definitions as the interviewer is. *When preparing for graph theory, focus on learning algorithms to solve common problems, rather than trying to learn all the different terms.*

Common Representations of Graphs

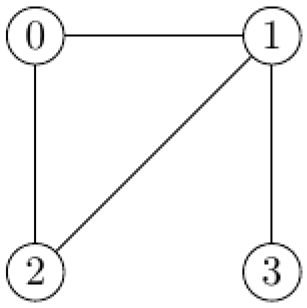
There are two common representations of graphs. To keep things simple, we'll assume that the `n` nodes are ordered by integers from `0` to `n-1`, so we can use position in an array to determine the relevant node. If the nodes have other labels, we can use HashMaps or an external array mapping integers to labels.

1. **Adjacency-list representation**
2.
3. We have a list-of-lists, `connections`, where `connections[i]` is a list of the node labels that the `i`th node is connected to. An edge `[i,j]` is represented by having `j in connections[i]`. In an undirected graph, `j in connections[i]` implies `i in connections[j]`, for a digraph `connections[i]` contains all the nodes you can move to from the `i`th node.

If the graph is weighted, we can have `connections[i]` be the tuple `(j, weight_ij)`.

1. **Adjacency-matrix representation**
2.
3. We have a n\times n$n \times n$ matrix `E`. For an unweighted graph, `E[i][j]` is `1` if there is an edge `[i,j]` in the graph, and `0`otherwise. For a weighted graph, `E[i][j]` is the weight of edge `[i,j]`, with a reserved value to indicate no edge. This reserved value is often `0` or `-1`, but this will change depending on the application. It is essential that the value used to indicate "no edge" cannot actually occur for a real edge in the problem.

So which one should you use? It depends on your use case. If `m` denotes the number of edges and `n` denotes the number of vertices, then we know for connected graphs (n - 1)\leq m \leq n(n-1)/2$(n-1)\leq m \leq n(n-1)/2$. For example, on the graph below:



has the representations

- `adj_list = [[1,2],[0,2,3],[0,1],[1]]`

-
  ```
  adj_matrix = [[0,1,1,0],
                [1,0,1,1],
                [1,1,0,0],
                [0,1,0,0]]
  ```

The properties of the representations are summarized in the table below.

Representation	Lookup time	Memory usage
Adjacency List	`O(m)` (worst), `O(m/n)` average	`O(m)` (assuming connected)
Adjacency Matrix	`O(1)`	$O(n^2)$

The adjacency list is never worse in memory than the adjacency matrix, and if the graph is sparse then it can be competitive in lookup time. If we drop the assumption that the graph is connected, then the memory usage of the adjacency list is `O(n+m)`, as we need at least one array index per node in the graph.

For dense graphs, which structure is better for your problem depends on whether you are memory constrained, in which case you should choose an adjacency list, or if you will be time constrained with lookups, in which case you should choose an adjacency matrix. You can always choose to store edges in another datastructure, such as storing the edges as keys to a hash table or in a set. However, most problems will give you the graph encoded using either an adjacency list or an adjacency matrix.

Classic Graph Algorithms to Expect in Interviews

Finding the Shortest Path

There are two variants for shortest path problems.

In an unweighted graph, we want to find a path between two vertices using the smallest number of edges possible. The go-to algorithm is *breadth-first search*. For example, if you wanted to find the number of degrees of separation between "Kevin Bacon" and "Gong Li", using a graph that had actors as nodes and connected edges if they were in the same movie would be a shortest path problem that would use BFS. (At the time of writing, "Kevin Bacon" and "Gong Li" had two degrees of separation).

If the graph is weighted, the length of the path is defined as the sum of the weights on the edges, not the number of edges. If none of the weights are negative, the standard approach is to use *Dijkstra's algorithm*. Here we couple BFS with a min-heap to look for the shortest path.

An pseudo-code implementation of Dijkstra's algorithm is

```
#
# source = index of the source node
# edges  = representation of edges
# n      = number of nodes in graph
#
def shortest_path(source, edges, n){
  # make an array of distances, which are the distance
  # from the source node.
  for index from 0 to n-1 {
     dist[index] = inf
  }
  # the source node is 0 distance from itself
  dist[source] = 0

  # make a min-heap of unvisited nodes
  for index from 0 to n-1 {
    toVisit.add( (dist[index], index) )
  }

  while toVisit is not empty{
    # find the closest node to source so far. Because
    # no edge lengths are negative, there are no alternate
    # routes involving nodes further from source that give
    # a shorter path to this node

    # distance is stored in dist[] array, so discard.
    # keep index
    _, current = toVisit.extractMin();

    for each destination, edge_dist in edges[current]{
      # does a detour through the current lead to a
      # shorter distance?
      dist_to_node = min(dist[destination], dist[current] + edge_dist)
      if dist_to_node is not dist[destination]:
        # need to update the heap / priority queue.
        # Finds index and updates first element to
        # dist_to_node, and reimplements to the heap invariant
        # takes O(log n) time.
        toVisit.updateElements( (dist_to_node, index))
    }
  }

  return dist
}
```

Notice that Dijkstra returns the distances between the source node and all the other nodes in the graph. If you are only interested in the distance between two particular nodes, the loop can be interrupted once you reach the target node. If the graph isn't connected, the loop can be optimized by returning as soon as `dist[current] = infinity`, as at this point the algorithm has visited all the nodes that are connected to `source`.

If you need to find the distances between any pair of nodes, or to find the shortest path between two nodes if there are negative weights on edges, you should use a different algorithm such as Bellman-Ford.

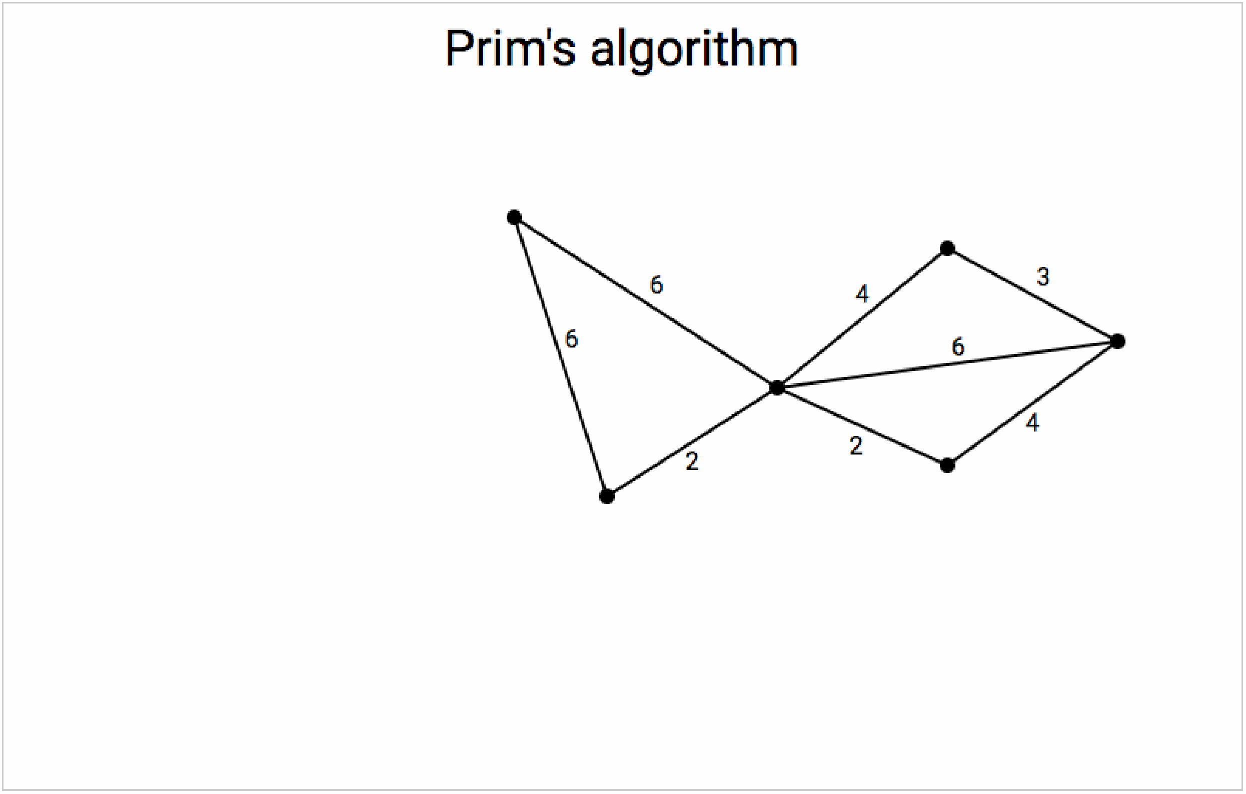
Minimum Spanning Tree

A "spanning tree" of a connected component with `n` nodes is a subgraph that has no cycles and leaves the `n` nodes connected. Every spanning tree has `n-1` edges; for a typical graph there are multiple spanning trees.

In a weighted graph, the weight of the spanning tree is the cost of all edges used in the tree. The tree with the lowest weight is called the "minimum spanning tree". A typical example of where a minimum spanning tree is useful is where the edges represent potential connections we could make between objects, and the weights are the cost of building those connections. A minimum spanning tree then gives a way of connecting all the objects for the smallest costs. While the cost of a minimum spanning tree is unique, the tree itself is generally not unique.

There are two well known algorithms for finding a minimum spanning tree: building node-by-node (Prim's algorithm), or edge-by-edge (Kruskal's algorithm).

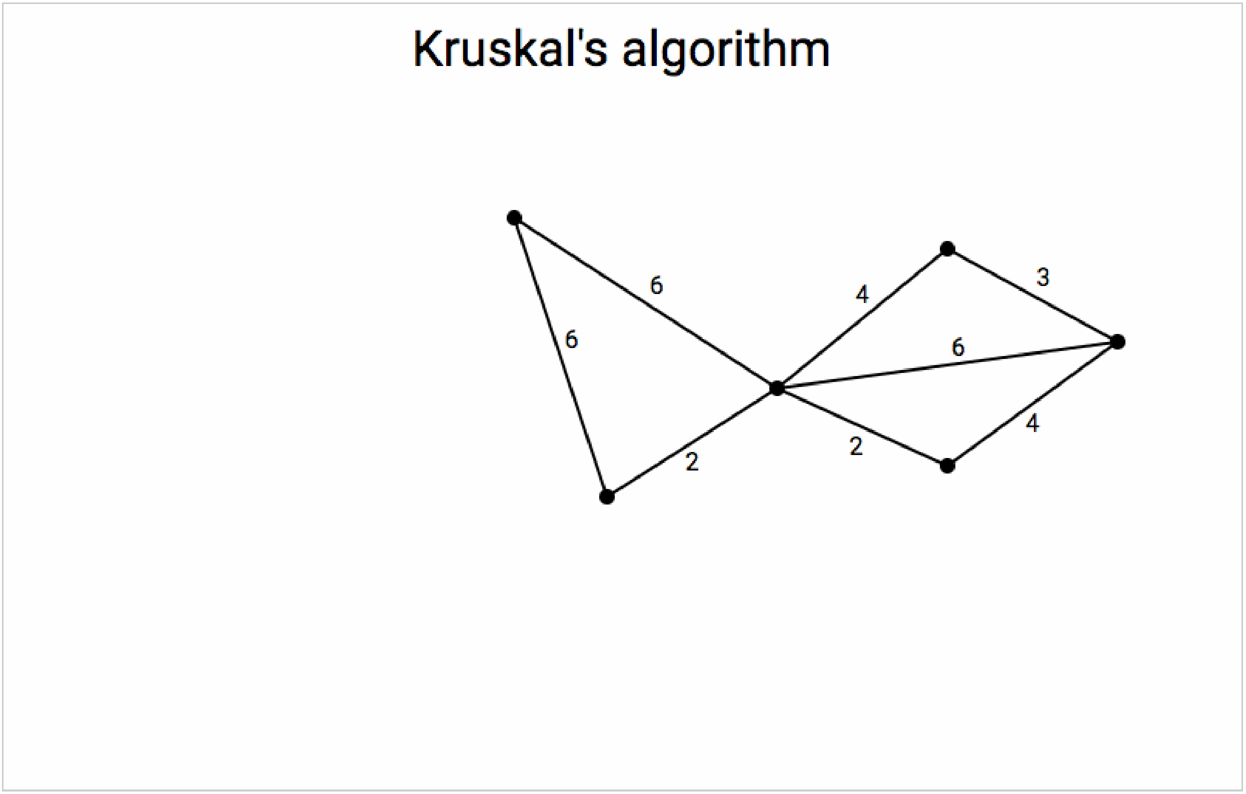
- **Prim's algorithm**
  - *Initialize:* Choose a vertex from the connected component at random. This vertex (and no edges) is your current solution.
  - While there are still nodes not in your current solution:
    - Find the vertex v^\prime$v'$ that is closest to a vertex in your current solution. Break ties arbitrarily.
    - Add v^\prime$v'$ and the shortest edge from a node in your current solution to v^\prime$v'$ to your current solution.



Most of the work in Prim's algorithm is finding the closest vertex to any node in the current solution. That is, we are looking for the lowest weight edge with one endpoint in the current solution, and the other endpoint not in the solution. An efficient approach is to maintain a min-heap of edges with exactly one endpoint in the solution. This slick use of a heap comes at the cost of having to do extra bookkeeping. Whenever a node v^\prime$v'$ is absorbed into the solution, all edges between v^\prime$v'$ and other nodes in the solution have to be removed from the heap, and all edges between v^\prime$v'$ and edges not in the solution have to be added.

1. **Kruskal's algorithm**

- *Initialize:* Start with a sorted list of edges EE in the original graph. Start with an empty set of desired edges SS.
- While |E| is not equal to `n-1`:
    - Remove the shortest edge ee from the set EE;
    - If adding ee wouldn't create a cycle, add the edge ee to SS.



Most of the work in Kruskal's algorithm comes from sorting the original edges, which is an `O(m log m)` operation. Since m \leq n^2$m \leq n^2$, \log m \leq 2 \log n$\log m \leq 2 \log n$, this is also an `O(m log n)` operation.

The other place we might be concerned about the amount of work is when we try to determine if adding an edge ee would create a cycle. Using a *union-find* data structure (also called a *disjoint set*), we can make this a very fast operation. The idea is that we are going to label vertices according to which connected component of the *current solution* they belong to. (They all belong to the same connected component of the complete graph!) In more detail:

- Start with an array of `n` integers in an array `compt`, all initialized to zero. Initialize `nextComponent = 1`.
- When adding an edge with endpoints `i` and `j`, check `compt[i]` and `compt[j]`. There are four possibilities:
    - If `compt[i] == compt[j]`, and they are not zero, then `i` and `j` belong to the same connected component of the solution. Adding this edge would make a cycle, so skip this edge.
    - If exactly one of `compt[i]` and `compt[j]` is zero, you are growing the non-zero component. Set both `compt[i]`and `compt[j]` to the non-zero value, and use the edge.
    - If both `compt[i]` and `compt[j]` are both zero, then we are creating a new component that is disconnected from the edges we collected in SS. Set `compt[i]` and `compt[j]` to `nextComponent`, and increase `nextComponent` by 1. Use the edge.
    - If `compt[i] ≠ compt[j]`, then we are merging two different components. Scan `compt`, changing all instances of `compt[i]` to `compt[j]`. Use the edge.

Coloring a Graph

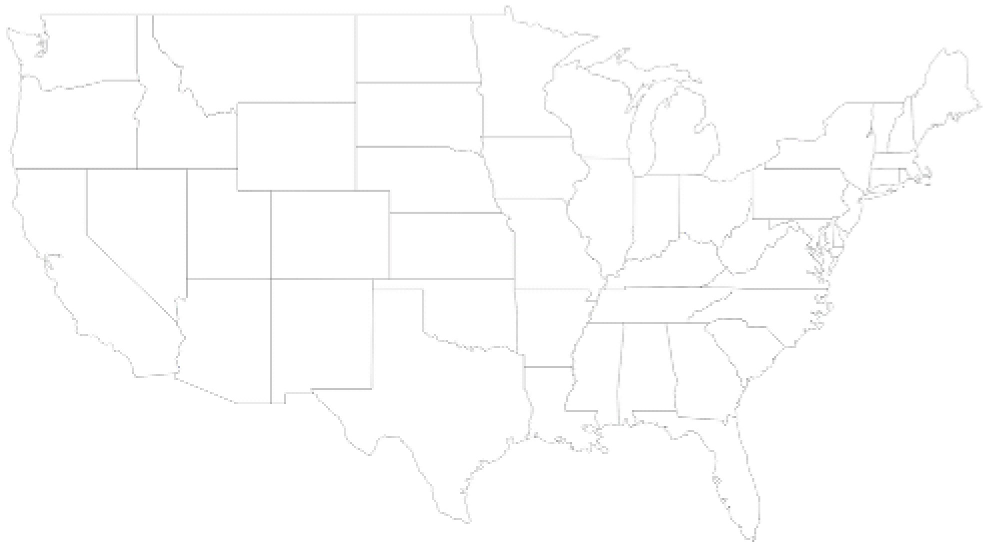
A coloring of a graph is a way of assigning a color to each node such that no two nodes that are connected by an edge have the same color.

A graph is kk-colorable if it can be colored with at most kk distinct colors. The *chromatic number* of a graph is the smallest number of colors needed to make a coloring of the graph.

Coloring problems are surprisingly difficult. To determine the chromatic number of an arbitrary graph is an NP-complete problem (in other words, it's really hard). Perhaps more surprising, answering the "yes/no" question of whether a graph is 3-colorable is also NP-complete! The simplest approach for coloring, at least in the types of graphs that are likely to come up as an interview task, is to use a backtracking algorithm for assigning the nodes different colors. If you're trying to determine the chromatic number of a graph, try determining if it's two colored using a trick, then try using backtracking with 3 colors, then 4, then 5, etc. until you have found a coloring that works.

The trick for determining if a graph is 2-colorable is by looking at its simple cycles (the paths that only repeat the first and last vertex). If all simple cycles have an even number of vertices, the graph is 2-colorable, otherwise it isn't. Technically a "graph" can be 1-colorable, but only if it has no edges.

The coloring of a graph is really just a labeling of its vertices. The term "coloring" originated from mapmakers trying to color maps. Different areas were represented as nodes, and an edge between two nodes meant they shared a border and shouldn't be the same color. As an example, the animation below shows the process of turning the map of the 48 contiguous United States into a graph, and how a coloring of the graph can be used to provide a coloring of the map.



The 4-color theorem states that any *planar graph*, that is, a graph that can be written on a piece of paper without any edges crossing, is 4-colorable. Any map we choose will produce a planar map, provided we ignore "borders" that consist of a single point (such as the 4-corners in the United States).

A lot of constraint problems can be expressed as graph coloring problems:

- Determining the number of meeting times needed for busy people can be cast as a coloring problem. Specifically, if a group of people have to break into groups for meetings, two meetings that don't have any attendees in common can be held at the same time. If there are common attendees, the meetings shouldn't overlap. To cast this as graph problem, let the nodes be the different meetings that have to occur, and two nodes are connected by an edge if some people have to attend both meetings. If provided with a coloring of this graph, every meeting assigned to the same color can be held simultaneously (because no two nodes assigned the same color can share an edge). The number of different colors is the number of non-overlapping meeting times needed.
- You need to seat a large family at a wedding. Sadly, there are some people that shouldn't sit together. To turn this into a coloring problem, make the nodes people, and draw an edge between them if they **shouldn't** be at the same table. Given a coloring of this graph, you can safely assign all people that have been tagged with the same color to the same table. The number of colors used is the number of tables you will need.

The tricky part about setting up problems as coloring problems is that we need to connect "things" in the problem that are not allowed to be together in the solution.

Matching

A matching on an undirected graph is a way of selecting edges so that every vertex is incident to at most one edge. One of the most common matching problems is also the simplest: looking for matches when the nodes have two different types, and only nodes of different types can be joined. These are called *bipartite graphs*, and can be technically defined as graphs that are 2-colorable.

A famous example is making matchings between medical students who are looking for residencies, and the positions available at hospitals. Each resident can be assigned to only one available position, and each position can be filled by at most one resident. If we create nodes that are the positions and the residents, and an edge between a "resident" node and a "position" node means "this resident is assigned to that position", a valid placement of residents into positions is a matching on this graph. This is similar to the problem of job applicants applying to positions.

Generally, not all matches are created equal. On a weighted graph, you might be trying to select the edges in the graph that make a matching with the highest weight. In the resident/hospital application problem, residents have preferences about which hospitals they want to work at, and hospitals will have preferences about the residents they want to work there. In this case, you might have some measure of how to get everyone as close to their first choice as possible. A *stable match* is one where neither party is tempted to "defect" for a better offer once they've accepted the position.

In Interviews, Use Graphs When ...

In some cases, the question will be explicitly about graphs, such as asking how many connected components are in a graph given its adjacency matrix. If given a map of cities, and the roads between them along with their distances, and you are asked to find the shortest path between city A and city B, you would probably be thinking about Dijkstra's algorithm.

In other problems, the connection is more subtle. It isn't immediately obvious that the question of how many dinner tables you need to seat people, if certain people cannot be seated at the same table, is a graph coloring problem in disguise. Strictly speaking, it's an NP-complete graph theory problem in disguise! A direct method of approaching this problem, with backtracking, might be equivalent to the way you solve the graph theory coloring problem. So what does knowledge of graph theory get you in this instance? Since you know already that the problem is hard, you can relax a little when you start writing your backtracking algorithm when you realize that it isn't going to be polynomial time (because you know solving this problem is as hard as solving an NP-complete problem). Or maybe you know that if you want a reasonable running time, you are going to need to memoize some intermediate results.

The big value in graph theory comes in finding the relationship between problems, as well as knowing the best approaches for those classes of problem.

Depth-First Search & Breadth-First Search

Interview Essentials

Depth-first search (DFS) and breadth-first search (BFS) are common algorithms for searching trees or graphs.

In **DFS**, you start at the root of the tree (or a selected node if it’s a graph) and search as far as possible along each branch before backtracking. DFS is great for searching trees that are deeper than they are wide. In technical interviews, DFS is good for solving puzzles that have only one solution (like mazes).

In **BFS**, on the other hand, you start at the root of the tree (or a selected node in a graph) and search each subsequent level of nodes before moving on to the next level. BFS is great for searching trees that are wider than they are deep. In an interview, BFS is a great option for problems about file system traversal, printing hierarchies in order, and finding the shortest path between two nodes in a graph.

DFS and BFS are both methods of searching a tree or graph for nodes with particular properties. Both methods allow the reconstruction of a path from the starting node to the solution node(s).

Because they both do the same thing, it can be difficult to decide whether DFS or BFS is the better approach. For problems where we are looking for solutions that are close to the starting node, BFS is a better choice since it searches the closest nodes first. For many problems, the approach depends on the structure of the tree or graph.

Because trees and graphs can contain a lot of nodes, the memory requirements may guide whether you use DFS or BFS. For trees, BFS requires you to store all the nodes at a given level, while DFS requires you to store an entire path from root to leaf. Because trees are often balanced, i.e. structured in a way to minimize depth, DFS will generally require less memory. However, some decision trees and some generated trees can have branches that are infinitely long, in which case BFS would be a better choice.

A full binary search tree with depth `d` will require `O(log d)` memory for DFS but `O(2^d)` memory for BFS.

Important Concepts

- **Branching factor (of a tree):**
- The maximum number of children that a node in a tree has.
- **Depth or height (of a tree):**
- The maximum path length between the root and a leaf in the tree.
- **Visited:**
- A *visited* node is any node that the search algorithm has already looked at. This is especially important for graphs, as keeping track of visited nodes ensures that we don't produce infinite loops.

Related Concepts

- **Trees and Graphs:**
- Both DFS and BFS search are methods for searching through a *graph* (and often this graph is a *tree*) to find a particular node. It's important that you are familiar with trees before you continue on with this article.
- **Acyclic graphs:**
- Any graph where there is at most one path between any two nodes (i.e. there are no "loops"/"cycles" in the graph). All trees are acyclic by definition.
- **Stacks:**
- DFS can be implemented using a stack to store the nodes that have already been visited.
- **Queues:**
- BFS can be implemented using a queue to store the nodes in the next level that haven't been visited.
- **Backtracking:**
- Backtracking can be thought of as a version of DFS that is used when a graph/tree has additional structure that allows you to know that descendants of a node cannot satisfy your search criteria, and will not visit them.

Generalized Algorithms

Our goal is to find all nodes that satisfy some condition. We will refer to these nodes as *solutions*. Here we keep track of all the visited nodes, which can use a substantial amount of memory. The array `visited` is there to prevent the code running in an infinite loop, so if we know we have a tree (or more generally any *acyclic graph*) we don't need `visited`.

Recursive DFS

```
function DFS_recur(node current , set solutions, set visited):
    add current onto visited

    if current is a solution:
        add current to solutions

    for each edge from current:
        new_state <- state obtained from current by following edge
        if new_state not in visited:
          DFS_recur( new_state, solutions, visited )
```

If the nodes can be altered, it is often useful to store the "visited" status in the nodes instead of in a separate data-structure. Then looking up whether `new_state` has been visited requires looking at the appropriate field of `new_state`, rather than searching through the `visited` data structure.

For a tree, or other acyclic graph, the recursive algorithm for DFS is simpler and requires less memory:

```
// Only use this version for acyclic graphs!
function DFS_recur(node current , set solutions ):
    if current is a solution:
        add current to solutions

    for each edge from current:
        new_state <- state obtained from current by following edge
        DFS_recur( new_state, solutions )
```

In some languages, there is a limit to how many nested recursive calls you are allowed to make. Exceeding this limit raises a *stack overflow* error. (Python is notorious for defaulting to a low number of nested recursive calls.) Behind the scenes, calling a function recursively pushes all your variables to the *call stack*. We can get around this limit by implementing our own stack in an iterative version.

BFS and Iterative DFS

BFS is usually implemented using a queue (First In, First Out) to store nodes, and looks quite different from our recursive DFS. By writing an iterative version of DFS using an explicit stack (First In, Last Out) to store nodes, we can see that they're quite similar.

```
function BFS_iter(node start_node):
    create empty set visited
    create empty queue Q
    create empty set solutions

    add start_node to visited
    Q.enque(start_node)

    while Q is not empty:
      current = Q.deque()
      if current is a solution:
        add current to solutions

      for each edge from current:
        new_state <- state obtained from current by following edge
        if new_state not in visited:
          add new_state to visited
          Q.enque(new_state)
    return solutions
```

Compare this to an iterative version of DFS:

```
function DFS_iter(node start_node):
  create empty set visited
  create empty stack Stk
  create empty set solutions

  add start_node to visited
  Stk.push(start_node)

  while Stk is not empty:
    current = Stk.pop()
    if current is a solution:
      add current to solutions

    for each edge from current:
      new_state <- state obtained from current by following edge
      if new_state not in visited:
        add new_state to visited
        Stk.push(new_state)
    return solutions
```

We see that other than the names of the methods for adding and removing elements (`push` and `pop` vs `enque` and `deque`), the algorithms are almost exactly the same.

If we have a tree or other acyclic graph, the code can be simplified more by dropping all the lines that refer to the `visited`set.

Pathfinding

For some problems, such as finding a path through the maze, finding the node (the exit) isn't that interesting. Instead, we're looking for the path to the desired node.

An easy modification that allows a reconstruction of a path is to make the visited elements contain two pieces of information: the node that was visited, and where it was visited from. Once you have the desired node, you can follow the from nodes back to the start to recover the path.

For An Interview, Expect To See Problems Like...

- **Finding a path out of a maze**;
- **Finding the shortest path between two nodes**;
- **Finding all friends and friends of friends in a social network**;
- **Finding all nodes reachable from a given node in a graph** (i.e. identifying the *connected component* containing a given node).

Backtracking

Interview Essentials

Backtracking is a technique used to build up to a solution to a problem incrementally. These "partial solutions" can be phrased in terms of a sequence of decisions. Once it is determined that a "partial solution" is not viable (i.e. no subsequent decision can convert it into a solution) then the backtracking algorithm retraces its step to the last viable "partial solution" and tries again.

Visualizing the decisions as a tree, backtracking has many of the same properties of depth-first search. The difference is that depth-first search will guarantee that it visits every node until it finds a solution, whereas backtracking doesn't visit branches that are not viable.

Because backtracking breaks problems into smaller subproblems, it is often combined with dynamic programming, or a divide-and-conquer approach.

Common interview tasks that use backtracking are crossword puzzles, Sudoku solvers, splitting strings, and cryptarithmetic puzzles.

Important Concepts

- **State:** A "partial solution" to the problem.
- **Rejection criterion:** A function that rejects a partial solution as "unrecoverable". i. e. no sequence of subsequent decisions can turn this state into a solution. Without rejection criteria, backtracking is the same as depth-first search.
- **Viable:** A state that may still lead to a solution. This reflects what is known "at this stage". All states that fail the rejection criteria are not viable. If we have a state that is initially viable, and find that all paths to leaves eventually terminate at a state that fail the rejection criteria, the state will become non-viable.
- **Heuristic:** Backtracking will (eventually) look at all the different viable nodes by recursively applying all the possible decisions. A heuristic is a quick way of ordering which decisions are likely to be the best ones at each stage, so that they get evaluated first. The goal is to find a solution early (if one exists).
- **Pruning:** The concept of determining that there are no nodes / states along a particular branch, so it is not worth visiting those nodes.

Related Concepts

Here are some concepts that will help you implement a backtracking algorithm.

- **Depth-first search:**
- A backtracking algorithm is conceptually very similar to depth-first search (DFS). A DFS will "roll back" the current state to a node up the tree once it reaches a leaf node, and is guaranteed to find a solution (or search every node). Backtracking can be thought of as DFS with early pruning.
- **Recursion:**
- Backtracking is often implemented using recursion, so it helps to be familiar with how to write recursive functions.
- **Stacks:**
- If you need a lot of nested recursive calls, trying to use simple recursion might result in a *stack overflow* error. You can mimic recursion by using a stack data structure.

Generalized Algorithm

In the algorithms below, it is assumed that each decision is irreversible, so there is only one path to each state. If modeling a game like chess, you would have to be more careful, as it is possible to arrive at the same state multiple different ways. If it is possible to reach each state multiple ways, then we would also need to keep track of which states we had already visited.

A recursive implementation of a backtracking algorithm takes the general form

```
function doBacktrack( current ):
    if current is a solution:
        return current
    for each decision d from current:
        new_state <- state obtained from current by making decision d
        if new_state is viable:
            sol <- doBacktrack(new_state)
            if sol is not None:
                return sol
    # indicate there is no solution
    return None
```

In some languages, there is a limit to how many nested recursive calls you are allowed to make. Exceeding this limit raises a *stack overflow* error. (Python is notorious for defaulting to a low number of nested recursive calls.) Behind the scenes, calling a function recursively pushes all your variables to the *call stack*. We can get around this limit by implementing our own stack.

```
function doBacktrackIterative(start):
    stack <- initialize a stack
    stack.push(start)

    while (stack not empty):
        current = stack.pop()

        if current is a solution:
            return current

        for each decision d from current:
            new_state <- state obtained from current by making decision d
            if new_state is viable:
                stack.push(new_state)

    return None
```

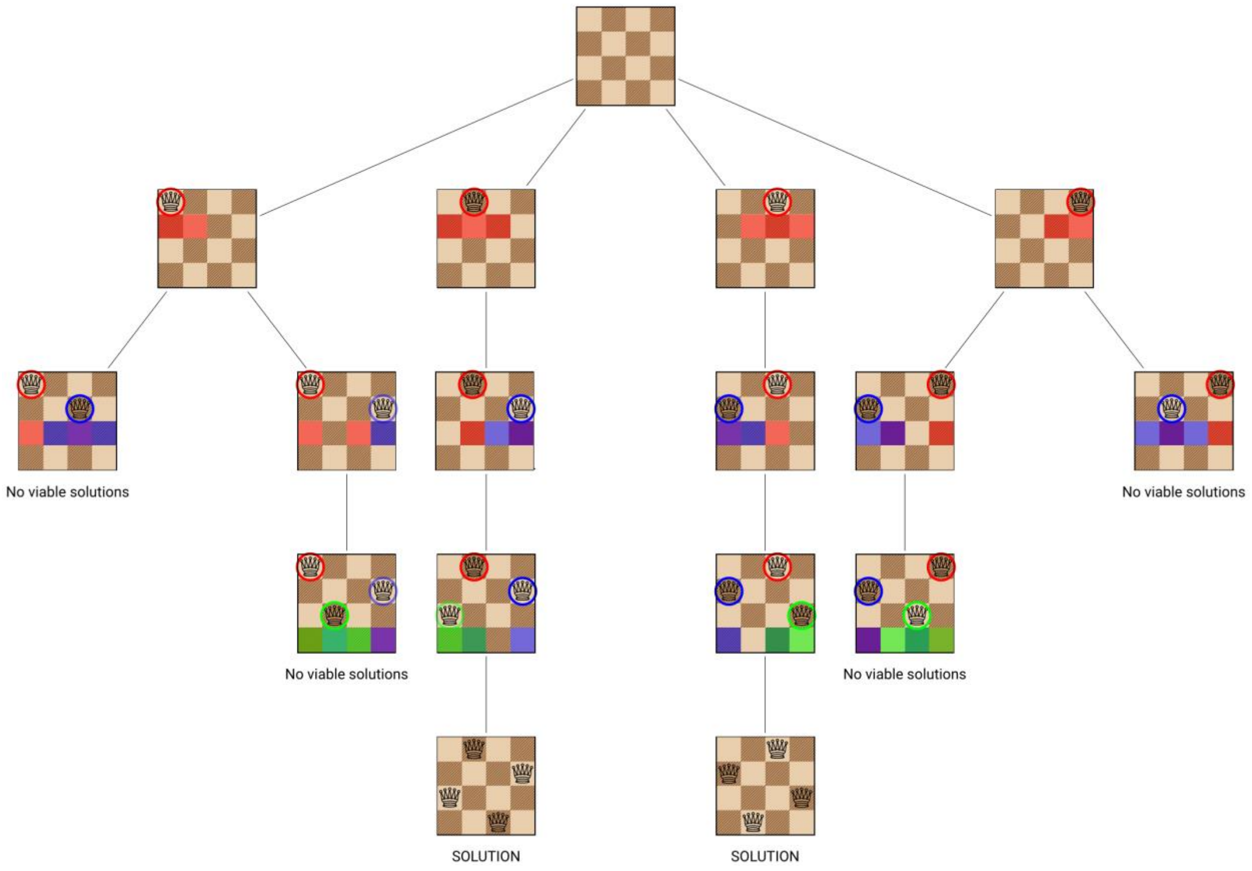
For An Interview, Expect To See Problems Like...

1. **Placing n queens on an n x n chess board so no queen can be taken**

2.
3. We know that we need one queen in every row, and one queen in every column.

- *Initial state:* An empty board
- *The decision:* The decision at stage *i* is which column to put the queen on row *i* into.
- *Rejection criterion:* Reject all configurations that do not allow any queen on any of the lower rows. (Adding more queens to the board will not prevent this row from being blocked.)

Here are the states that the backtracking algorithm explores for a 4x4 board. Note that naively there are 4! = 24 *complete configurations* with 4 queens on the board, if we place the queens row-by-row, and avoid reusing a column that is already used. With backtracking, many configurations are eliminated early. This algorithm shows *all* solutions, instead of stopping at the first one found.



1. **Finding a subset of a set of positive integers *S* that sum to a particular number**

2.
3. We have a set of numbers *S = {10, 7, 5, 18, 12, 20, 8}*, and a target value *T = 23*. The goal is to find a subset of *s* of *S*whose elements sum to the target value, or report that such a sum doesn't exist.

- *Initial state:* The subset *s* is empty
- *The decision:* Placing an element from *S - s* into *s*.
- *Rejection criterion:* If the sum of elements in *s* so far is greater than 18 (and is not a solution), then this solution is infeasible. This is because 5 is the smallest element in *S*.

1. **Finding the best possible move in a game**
2.
3. A generic approach for finding the best strategy (sequence of moves) in a game is:

- *Initial state:* The position of the pieces at the beginning of the game.
- *The decision:* Making a valid move.
- *Rejection criterion:* Varies a lot by game.

Sorting

Interview Essentials

Sorting algorithms are used to put the elements of an array in a certain order, generally using a comparison function to determine order (a generalization of *less than*). Some comparison functions will allow for ties between different objects. A **stable** sorting algorithm will guarantee that if two elements "tie", then whichever element occured first in the original array will also occur first in the sorted array. An **unstable** sorting algorithm will only guarantee that the final elements are in order, and makes no guarantee about the order of "tied" elements.

Being asked to implement a sorting algorithm from scratch used to be a frequently asked interview question, but this is becoming less popular. Just in case, though, it's worth knowing how to implement two of the more useful sorting algorithms:

- **Quick sort**

- **Merge sort**

For other sorting algorithms, you might be asked about their time complexity, their space complexity, and whether or not the sort is stable. In decreasing order of probability of coming up in an interview, the other common sorts are:

- **Insertion sort**
- **Heap sort**
- **Bubble sort** (it's usually sufficient to explain why bubble sort is terrible)
- **Radix sort** (the only non-comparison based sort on this list)

Quick Sort

Important Concepts

- Quick sort is a comparison sort, which means that it can only be implemented in arrays in which there's a notion of "less than" and "greater than".
- If it's implemented correctly, quick sort can be up to 3x as fast as merge sort and heap sort.

Time Complexity

- On average, quick sort can sort `n` items in `O(n log n)` comparisons. (In the worst case scenario, it can take $O(n^2)$, but this is fairly rare. However, if you need to guarantee an `O(n log n)` solution in an interview, you should mention other sorting options if you choose to use quick sort.)

Space Complexity

- Quick sort works in place, so it has a space complexity of `O(1)`.

Generalized Algorithm

1. Pick a pivot element from the array.
2. Reorder the array so that all the elements with values less than the pivot now come before the pivot, while all elements with values greater than the pivot now come after it. Equal values can go either before or after. After this partitioning operation, the pivot element will be in its final position.
3. Recursively apply these steps to the sub-array of elements with smaller values.
4. Recursively apply these steps to the sub-array of elements with greater values.
5. The base case of the recursive algorithm is arrays with a size `0` or `1`, which don't need to be sorted.

Merge Sort

Important Concepts

- Merge sort is a comparison sort, which means that it can only be implemented in arrays in which there's a notion of "less than" or "greater than".

Time Complexity

- The average and worst-case time complexity of merge sort is `O(n log n)` because the input is halved on each loop.

Space Complexity

- Merge sort doesn't sort in place. We make a new array to copy into when merging, and at the last step we merge two arrays of size n/2. The amount of additional space we need to copy in to is `O(n)`.

Generalized Algorithm

Merge sort is a recursive algorithm. Suppose the function is called `merge(L)`, which returns the sorted array.

1. Given an array, if its length is bigger than 1, split it into two pieces `left` and `right`. (The pieces should be as close to evenly sized as possible). If the array has length 1 or zero, then it is automatically sorted, so just return that trivial array.
2. Call store `mL = merge(left)` and `mR = merge(right)`. This will give you a sorted copy of the left and right half of the array.
3. Start with a reference to the beginning of `mL` and `mR`, called `ref_L` and `ref_R` respectively. Build up a new array `ans` that merges the contents of `mL` and `mR` by taking the smaller of `ref_L` and `ref_R`, and updating the reference taken to the next element. This merge guarantees that `ans` is in order from smallest to greatest.
4. Return the merged array `ans`.

Overview of the Rarer Sorting Algorithms

Insertion Sort

An in-place sorting algorithm.

1. Start with the index `current` at 0. (Insertion sort ensures that all elements up to and including the element at index `current` are in sorted order, which is trivial for the first element).
2. All elements before `current` are sorted, elements after `current` are unsorted. Get the first element after `current`.
3. Find where the next element fits in the sorted portion of the array.
4. Now the sorted portion of the array is one index larger. Move the `current` index one place further toward the end.

Repeat steps 2 - 4 until the current index is at the end. The resulting array is completely sorted (all elements occur before `current`).

Insertion sort does well in cases where the array is already partially sorted, as it can "move past" elements that are already in order.

Heap Sort

An in-place sorting algorithm.

1. Pass through the array one element at a time, converting the subarray before the current index into a min-heap.
2. Once the entire array has been converted into a min-heap, use the heap's `extract-min` method, and place the retrieved element at the end of a new array. Since each element is the minimum of the heap at the time of extraction, this process ends with a sorted array.

Selection Sort

An in-place sorting algorithm.

1. Start with the index `current` at 0. (Selection sort maintains the invariant that the first `(current-1)` smallest elements appear in sorted order at the beginning of the array. This invariant is trivial when `current` is zero.)
2. Scan the subarray of the elements from `current` to the end of the list. Select the smallest element in the subarray.
3. Swap the element at `current` with the smallest element found in step 2.
4. The first `current` smallest elements are at the beginning of the array. Move the index `current` forward one place.

Repeat steps 2 - 4 until the `current` index is at the end of the array. The resulting array is sorted.

A lot of the time in selection sort is spent finding the minimum element in the unsorted portion of the array. Heap sort (discussed above) is essentially a selection sort in which the heap is used to speed up the repeated selection of the minimum with `extract-min`.

Bubble Sort

An in-place sorting algorithm. Bubble sort is a very poor sorting algorithm; its use in interviews is limited to talking about why it's a poor choice. Here is how bubble sort would operate on an array `a`:

1. Start by looking at the first two elements, `a[0]` and `a[1]`. If they are out of order, swap them.
2. Then compare the two elements, `a[1]` and `a[2]`. If they are out of order, swap them.
3. Iterate this procedure, putting items `a[i]` and `a[i+1]` in the correct order, until you reach the end. This process of going through the array with up to `n-1` swaps is called a *pass*.
4. Repeat this process until the array is fully sorted, which can take at most `n` passes.

Head-to-Head Table of Comparison-Based Sorting Algorithms

If only one entry is shown for complexity, it means that average and worst-case complexity scale the same way.

Sorting algorithm	**Time Complexity (Avg / Worst)**	**Additional Space Complexity (Avg / Worst)**	**Deterministic?**	**Stable?**
Quick Sort	`O(n log n)`/`O(n²)`	`O(log n)`	No	No
Merge Sort	`O(n log n)`	`O(n)`	Yes	Yes

Insertion Sort	$O(n^2)$ (*)	In-place (`O(1)`)	Yes	Yes
Heap Sort	`O(n log n)`	`O(1)`	Yes	No
Bubble Sort	$O(n^2)$	In-place (`O(1)`)	Yes	Yes
Selection Sort	$O(n^2)$	In-place (`O(1)`)	Yes	Implem. dependent

(*) Insertion sort is fast if the input is already partially sorted. If no item is more than `k` positions from its final position, the runtime is `O(nk)`.

Example of different sorts

We show the sequence of swaps the different swapping algorithms make to put the array `[6,5,3,1,8,7,2,4]` in order.

For insertion and bubble sort, we only included steps where the list changed, to reduce the number of steps shown. Merge sort was excluded because it isn't a simple swap, and quick sort was excluded because it isn't deterministic.

Swap #	Selection Sort	Insertion Sort	Bubble Sort
0	[6, 5, 3, 1, 8, 7, 2, 4]	[6, 5, 3, 1, 8, 7, 2, 4]	[6, 5, 3, 1, 8, 7, 2, 4]
1	[1, 5, 3, 6, 8, 7, 2, 4]	[5, 6, 3, 1, 8, 7, 2, 4]	[5, 6, 3, 1, 8, 7, 2, 4]
2	[1, 2, 3, 6, 8, 7, 5, 4]	[5, 3, 6, 1, 8, 7, 2, 4]	[5, 3, 6, 1, 8, 7, 2, 4]
3	[1, 2, 3, 6, 8, 7, 5, 4]	[3, 5, 6, 1, 8, 7, 2, 4]	[5, 3, 1, 6, 8, 7, 2, 4]
4	[1, 2, 3, 4, 8, 7, 5, 6]	[3, 5, 1, 6, 8, 7, 2, 4]	[5, 3, 1, 6, 7, 8, 2, 4]
5	[1, 2, 3, 4, 5, 7, 8, 6]	[3, 1, 5, 6, 8, 7, 2, 4]	[5, 3, 1, 6, 7, 2, 8, 4]
6	[1, 2, 3, 4, 5, 6, 8, 7]	[1, 3, 5, 6, 8, 7, 2, 4]	[5, 3, 1, 6, 7, 2, 4, 8]
7	[1, 2, 3, 4, 5, 6, 7, 8]	[1, 3, 5, 6, 7, 8, 2, 4]	[3, 5, 1, 6, 7, 2, 4, 8]
8		[1, 3, 5, 6, 7, 2, 8, 4]	[3, 1, 5, 6, 7, 2, 4, 8]
9		[1, 3, 5, 6, 2, 7, 8, 4]	[3, 1, 5, 6, 2, 7, 4, 8]
10		[1, 3, 5, 2, 6, 7, 8, 4]	[3, 1, 5, 6, 2, 4, 7, 8]

11		[1, 3, 2, 5, 6, 7, 8, 4]	[1, 3, 5, 6, 2, 4, 7, 8]
12		[1, 2, 3, 5, 6, 7, 8, 4]	[1, 3, 5, 2, 6, 4, 7, 8]
13		[1, 2, 3, 5, 6, 7, 4, 8]	[1, 3, 5, 2, 4, 6, 7, 8]
14		[1, 2, 3, 5, 6, 4, 7, 8]	[1, 3, 2, 5, 4, 6, 7, 8]
15		[1, 2, 3, 5, 4, 6, 7, 8]	[1, 3, 2, 4, 5, 6, 7, 8]
16		[1, 2, 3, 4, 5, 6, 7, 8]	[1, 2, 3, 4, 5, 6, 7, 8]

Dynamic Programming: Basic

Interview Essentials

Dynamic programming is a powerful technique that helps you solve complex problems by breaking them down into simpler recurring subproblems. This is generally done by taking a recursive algorithm and finding the repeated calls, the results of which you can then store for future recursive calls. Since you've compute the answer for an input and saved it for later, you don’t have to recompute it the next time around!

In technical interviews, dynamic programming is especially useful when you’re solving problems about optimization, search, and counting. If, as you're analyzing a problem, you see that it can be broken down into smaller overlapping subproblems, it's a good signal that you can employ dynamic programming to solve it.

Important Concepts

There are two common dynamic programming strategies that you'll need to familiarize yourself with, bottom-up and top-down.

- **Bottom-up strategy (iteration):** In this strategy, you start solving from the smallest subproblem up towards the original problem. If you're doing this, you'll solve all of the subproblems, thereby solving the larger problem. The cache in bottom-up solutions is usually an array (either one or multi-dimensional).
- **Top-down strategy (recursion):** Start solving the identified subproblems. If a subproblem hasn't been solved yet, solve it and cache the answer; if it has been solved, return the cached answer. This is called *memoization*. The cache in top-down solutions is usually a hash table, binary search tree, or other dynamic data structure.

Generalized Algorithm

1. Find a recursive or iterative solution to the problem first (although usually these will have bad run times).
2. This helps you identify the subproblems, which are "smaller" instances of the original problem.
3. Determine whether the subproblems are overlapping. If they're overlapping, a recursive algorithm will be solving the same subproblem over and over. (If they're not overlapping and the recursive algorithm generates new subproblems each time it runs, DP isn't a good solution; try a divide-and-conquer solution like merge sort or quick sort instead.)
4. Store/cache the results for every subproblem.
5. Once the subproblems have solutions, the original problem is easy to solve!

In Interviews, Expect to See Problems Like...

- Knapsack problems
- Scheduling problems
- Compute the nth Fibonacci number

Common Techniques: Basic

Interview Essentials

There are some common problem-solving techniques that you should keep in your back pocket during technical interviews. They'll come in handy for a lot of different questions, and interviewers really want to see that you know them! Three of the most common problem-solving techniques you'll need to employ during interviews are:

- **Binary search:** an efficient search algorithm for sorted arrays that divides the input in half on each pass;
- **Two pointers:** a technique that uses two pointers to optimize naive solutions that involve using multiple loops;
- **Prefix sums:** a technique for quickly finding the sum of any contiguous slice of an array;

Binary Search

Important Concepts

- Binary search only works on *sorted* arrays.

Time complexity

- Since you're essentially halving your data input on each pass, the time complexity of this algorithm is `O(log n)`.
- Because it is very slow to perform insertions or deletions, binary search isn't useful if the sorted array changes often.

Space Complexity

- The algorithm takes only `O(1)` additional space. We have to allocate references to the lower bound, the upper bound, and the current element, regardless of how large the array is.

Generalized Algorithm

1. Choose an element in the middle. Is that the one you’re looking for? Great! Otherwise, determine whether the element you chose is smaller or greater than the one you’re looking for.
2. If the element you’re looking for is smaller than the one you chose, you know that you don’t need to worry about any elements to the right of your initial choice. Alternately, if the element you’re looking for is larger than the one you chose, you know that you don’t need to worry about any elements to the left of your initial choice. Either way, you’ve just halved the amount of elements that you need to look at.
3. Do the same thing again and again, until you’ve found the element you’re looking for!

Pseudocode

```
def binary_search(l, value):
    # Set a min and a max (the 0th element and the n-1 element)
    min = 0
    max = len(l)-1
    # Check to see if guess is what you’re looking for. If so, stop! You’re done!
    while min <= max:
        # Set a guess as the median of min and max. If the answer isn’t an integer,
round down.
        guess = (min + max)//2
        # If your guess was smaller than the target number, you’ll reset your max to
guess - 1.
        if l[guess] > value:
            max = guess - 1
        # If your guess was larger than the target number, you’ll reset your min to
guess + 1.
```

```
        elif l[guess] < value:
            min = guess + 1
        # Compute a new guess, as in step 2, and start again!
        else:
            return guess
    return -1
```

In Interviews, Expect to See Problems Like...

- Implementing binary search from scratch.
- Knowledge questions comparing binary search in an array to a binary search tree.
- Using a binary search to count the number of occurrences of a number in a sorted array.
- Using a binary search to count the number of occurrences larger than, or smaller than, a given number.
- Using the binary search strategy to locate inputs of a function without an explicit array, e.g. if `f(x)` is monotonically increasing that takes integer arguments, binary search can be used to solve invert `f(x)`, or show there is no solution (if `f(x)` takes floats as an argument, binary search can generally only approximate a monotone function to a given *tolorance*).
- Doing multiple searches for an item in a large array.

Two Pointers

Two Pointers techniques come up frequently when manipulating singly-linked lists. Having pointers to two different nodes can be used to overcome the restriction that you can only traverse a singly-linked list in one direction.

Important Concepts

- The two pointer technique can be used on any data structure that can be iterated through.

There are two main strategies for this technique:

- Have a fast-runner pointer and a slow-runner pointer, where the fast-runner is ahead of the slow-runner.
- Have one pointer start at the beginning of the data structure and one pointer start at the end, moving towards each other until they meet.

Time Complexity

- `O(n)`

In Interviews, Expect to See Problems Like...

- Remove duplicates from a sorted array
- Reverse the characters in a string.
- Reversing a linked list in place.
- Detecting a cycle in a linked list.

Prefix Sums

Prefix sums are a method of pre-computation that will allow us to calculate the sum of any contiguous block of an array in `O(1)` time! The time to compute the prefix sum initial is `O(n)`.

Generalized Algorithm

To create the prefix sum array

In general, prefix sum of an array `x` is an array `y` that can be calculated as:

$y_i = y_{i-1} + x_{i-1}$ for $i > 0$, and $y_0 = 0$

The array `y` will have one more element than `x`.

To use the prefix sum array

To get the sum of the elements $x_a + x_{a+1} + ... + x_b$ we would calculate $y_{b+1} - y_a$.

Note: Some sources may not use the leading 0 at the beginning of the prefix sum array. The convention used here wastes a little bit of space, but simplifies some of the bounds checking when using the array.

Time Complexity

- `O(n)` to create the prefix sum array.
- `O(1)` to use the prefix sum array.

Space Complexity

Given a prefix sum array, it is possible to recover the original array by looking at the difference between neighboring elements. This means we can choose to overwrite the original array with the prefix array, or we can choose to make a separate array.

- `O(n)` additional memory, if creating the prefix sum as a new array.
- `O(1)` additional memory, if creating the prefix array in place.

In Interviews, Expect to See Problems Like...

- Line of sight problems (calculating whether one point is visible from another, given an array of heights);
- Find a cumulative distribution (for example, find what percentage of income the top 1% of earners make from an array of incomes);
- Calculate the moving average on an array.

Strings

Interview Essentials

The string data type is used to represent text, and string values must be enclosed in either single or double quotation marks. Even though strings themselves are a basic data type, interviewers come up with questions that are anything but simple! Make sure that you are very familiar with string methods for your chosen interviewing language. In most languages, strings are implemented as character arrays, which means that string questions and array questions are interchangeable in a lot of cases.

Bits

Interview Essentials

Binary digits, or *bits*, are variables that can hold two values: `0` or `1`. Bits are grouped together in *bytes*, which is the smallest amount memory that can be allocated. Most interviewers will allow you to assume there are 8 bits in a byte, so each byte can have one of $2^8=256$ possible values.

Some interviewers will ask you to solve problems that are explicitly about manipulating bits. For these questions, it's helpful to know how integers are represented in binary, so that you can recognize that `7` is `0111`$_2$. It is also helpful to become familiar with the *two's-complement* representation for negative integers.

The bitwise operations (`AND`, `OR`, `XOR`, `NOT`) can be incredibly useful even when the task is not explicitly about bits. Bitwise operations are incredibly fast, and you can compress *N* yes/no decisions into *N* bits (rather than *N* bytes if we stored the numbers `0` and `1` for each decision).

Bitwise operations are particularly useful when doing image manipulation, or implementing the hash functions of hash tables. An advanced use of bits is often used in dynamic programming when looking at which of `N` objects we are going to select, as we can use an `N` bit integer to represent which objects were selected (if the `i`th bit is 1, the `i`th object is selected).

Table of contents

Our information about bits is broken into three parts:

- **Representation of integers in different bases**
- **Bitwise operations**
- **Bitmasks**

Many applications of bits don't actually require you to think about them as integers. Individual bits can be used to represent a decision. For example, the subset-sum problem asks if there is a subset of a given set that add to a given number. As an example, is there a subset of `{ 2, 3, 4, 6, 21, 14}` that sums to 12? One way of arranging the elements in an array (sets have no order!) and labeling the subsets by using the `i`th bit to determine if the `i`th element was in the subset or not. For example, the subset `{2, 4, 6}` (which does sum to 12) can be represented as the bit-string `101100` because

Element index	**2**	**3**	**4**	**6**	**21**	**14**
Subset bitmask	1	0	1	1	0	0

In order to use bits this way, and effectively manipulate the bits, it is important to know bitmasks, which in turn depend on bitwise operations. It is **not** as important to know that most programs think of 101100_2 as the number `44`.

We have included how integers are represented as bits because:

- When debugging, it is convenient to be able to print out your bitmask and interpret it.
- There are problems where we are using bits to determine the properties of numbers (e.g. determining if a number `n`is a power of 2 or not can be done very quickly using bitmasks).
- Interviewers may ask general knowledge questions about how bits are arranged.

What you **really** need to know about integers is

- `-1` will give you all bits set to `1`.
- `0` will give you all bits set to `0`.

Integer representations in different bases

Getting familiar with bases mathematically.

The most familiar representation of integers is in base 10 (i.e. using 10 digits, 0 through 9), with an optional sign in front. The sign will be treated a little bit differently when coding `N` bit integers. Because this is the "normal" way of representing a number, we often explicitly say that the number is written in base 10. When we see the number `101`, we generally assume it is one-hundred and one. If we want to be explicit, we can write the base as a subscript, such as 101_{10}.

One-hundred and one can be written as

$$101_{10} = 1 \cdot 100 + 0 \cdot 10 + 1 \cdot 1 = 1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0$$

We can summarize this somewhat systematically in a table:

power	**2**	**1**	**0**	
10^{power}	100	10	1	
Digits	1	0	1	
Contribution	+100	0	+1	
Result				101_{10}

If we wanted to represent one-hundred and one in powers of 8, for example, we would look at the powers of 8 that are less than 101, namely 1, 8, and 64. Each digit can take 8 values (0 through 7), so we know a number written in base 8 will never have an 8 or 9 in it.

$101_{10} = 1 \cdot 64 + 4 \cdot 8 + 5 \cdot 1 = \mathbf{1} \cdot 8^2 + \mathbf{4} \cdot 8^1 + \mathbf{5} \cdot 8^0 = 145_8$

In table form:

power	**2**	**1**	**0**	
8^{power}	64	8	1	
Digits	1	4	5	
Contribution	+64	+32	+5	
Result				101_{10}

N-bit unsigned integers

When writing an unsigned number in binary, or base 2, we are expanding it in powers of 2. The i^{th} bit, counted from the *right*, tells us about how many copies of 2^i we would need when expanding our number in powers of two. As an example, to "decode" an 8-bit number such as 01100111_2

power	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	

2^{power}	128	64	32	16	8	4	2	1	
Binary digits	0	1	1	0	0	1	1	1	
Contribution	0	+64	+32	0	0	+4	+2	+1	
Result									103_{10}

Because `64 + 32 + 4 + 2 + 1 = 103`, the number $01100101_2 = 103_{10}$.

If the number of bits is fixed (as it for the `short` and `long` datatypes in C/C++/Java) at `N` then we can only represent 2^Ndifferent values. The smallest number that can be represented is `0` (where all bits are zero), and the highest number that can be represented is $2^N - 1$ (where all the bits are set to one).

N-bit signed integers and 2's-complement

An "obvious" way of representing negative numbers would be to have the leftmost bit be `0` for a positive number and `1`for a negative number, much like the ± signs used in math. This encoding isn't widely used because it gives the number `0`two encodings. Demonstrating with 4-bit integers, 0000_2 and 1000_2 would both represent zero!

The standard encoding method is called the *two's complement*, which works in the following way:

- The leftmost bit represents -2^{N-1}
- The `N-1` other bits are interpreted as a `N-1` bit number.

For example, here are all 3-bit signed integers:

Bits	Expansion	#		Bits	Expansion	#
000	$0\cdot(-2^2) + 0 \cdot 2^1 + 0 \cdot 2^0$	0_{10}		100	$1\cdot(-2^2) + 0 \cdot 2^1 + 0 \cdot 2^0$	-4_{10}
001	$0\cdot(-2^2) + 0 \cdot 2^1 + 1 \cdot 2^0$	1_{10}		101	$1\cdot(-2^2) + 0 \cdot 2^1 + 1 \cdot 2^0$	-3_{10}
010	$0\cdot(-2^2) + 1 \cdot 2^1 + 0 \cdot 2^0$	2_{10}		110	$1\cdot(-2^2) + 1 \cdot 2^1 + 0 \cdot 2^0$	-2_{10}
011	$0\cdot(-2^2) + 1 \cdot 2^1 + 1 \cdot 2^0$	3_{10}		111	$1\cdot(-2^2) + 1 \cdot 2^1 + 1 \cdot 2^0$	-1_{10}

With 3 bits, we can represent the $2^3 = 8$ values from `-4` to `3`. An *N* bit signed integer can represent the values from -2^{N-1} to $+2^{N-1} - 1$.

Bitwise operations

The binary bit function is any function that takes two bits, and returns a single bit. It is common to refer to a bit being `1` as `True`, and a bit being `0` as `False`. If `x` and `y` are individual bits, then here are the most common binary bit functions:

- **AND(x,y):** Returns the bit `1` if `x` **and** `y` are `1`, otherwise returns the bit `0`.

- **OR(x,y):** Returns the bit `1` if either `x` **or** `y` (or both) are `1`, otherwise returns the bit `0`.
- **XOR(x,y):** Returns the bit `1` if only one `x` or `y` is `1`, otherwise returns the bit `0`. Called *eXclusive OR* to rule out both being `1` as an option.

Table of bitwise operations

x	**y**		**OR(x,y)**	**AND(x,y)**	**XOR(x,y)**
0	0		0	0	0
0	1		1	0	1
1	0		1	0	1
1	1		1	1	0

Most variables we deal with have multiple bits. For convenience, we label the `i`th bit of a variable `X` as X_i. Then we have the following **bitwise operations** between two `N` bit variables `X` and `Y`:

- `X & Y` (bitwise AND):
- The `i`th bit of `X&Y` is equal to `AND(`X_i`, `Y_i`)`.
- `X | Y` (bitwise OR):
- The `i`th bit of `X|Y` is equal to `OR(`X_i`, `Y_i`)`.
- `X ^ Y` (bitwise XOR):
- The `i`th bit of `X^Y` is equal to `XOR(`X_i`, `Y_i`)`.

Given a bit function, we can always define the corresponding bitwise operation that applies the bit function bit-by-bit. As an example, let's apply each of these bitwise operations to the 8 bit unsigned ints `01101110 → 110`$_{10}$ and `11000110 → 198`$_{10}$:

- Example of bitwise AND: `01101110 & 11000110 = 01000110` or `110 & 198 = 70`

```
  01101110
& 11000110
==========
  01000110
```

* Example of bitwise OR: `01101110 | 11000110 = 11101110` or `110 | 198 = 238`

```
  01101110
| 11000110
==========
  11101110
```

* Example of bitwise XOR: `01101110 ^ 11000110 = 10101000` or `110 ^ 198 = 168`

```
  01101110
^ 11000110
==========
  10101000
```

In addition to the binary bitwise operations `&`, `|`, and `^`, there is the bitwise NOT operator (usually referred to as `~` in programming languages). As the name suggests, every bit in `~x` is different from the corresponding bit in `x`.

Bit shifting

Our last operations aren't really bitwise operations (they don't perform operations bit-by-bit), but they are important enough to be included anyway. They are:

- **left shift:** `x << n`
- Takes the variable `x` and shifts it to the left `n` bits, filling new spaces on the right with zeros. For example $45_{10} = 00101101_2$, so
- `45 << 2` is $101101\mathbf{00}_2 = 180_{10}$
- The added zeroes have been shown in bold. For programming languages with fixed bit width data types like `short`and `long`, bits that get shifted too far to the right are dropped. Other programming languages like Python return an object with a greater number of bits.
- Left shifting an integer by `n` is the same as multiplying by `2<sup>n</sup>`.
- **(logical) right shift:** `x >>> n`
- Takes the variable `x` and shifts it to the right `n` bits, filling the new bits on the left with zeros. For example, $45_{10} = 00101101_2$, so
- `45 >>> 2` is $\mathbf{00}001011_2 = 11_{10}$
- Seeing the same result for a signed negative number is useful. We have $-45_{10} = 11010011_2$ so
- `-45 >>> 2` is $\mathbf{00}110100_2 = 52$
- When making a logical right-shift of a negative number, the answer will depend on the number of bits that are used to store the integer. Some languages that have variable-sized integers (notably Python) don't have a logical right-shift.
- **(arithmetic) right shift:** `x >> n`
- Takes the variable `x` and shifts it to the right `n` bits, filling new bits on the left with whatever is contained in the leftmost bit. For positive numbers, the leftmost bit is zero, so the arithmetic and logical right shifts are the same:
- `45 >> 2` is $\mathbf{00}001011_2 = 11_{10}$
- For negative numbers, there is a difference. Using the 8 bit signed integer `-45` again, we have
- `-45 >> 2` is $\mathbf{11}110100_2 = -12$

Bitmasks

There are many different interpretations possible for what each bit represents. In the introduction, we used the `i`th bit represents whether the `i`th element was in the subset for the subset-sum problem.

For bitmasks we'll make an analogy to a switch, where if bit `i` is 1 we say the `i`th switch is on, and when bit `i` is 0 we say the `i`th switch is off. The `0`th bit will be at the right end, and we will count back toward the front. This has the nice property that the `i`th bit is the coefficient of `2`^{i}.

Getting the `i`th bit / Checking the state of the `i`th switch

We can find the value of the `i`th bit of `n` by using `(n & (1 << i)) != 0`. Can you see why it works?

We use the following properties of the `AND` function:

- `AND(0,x) = 0`, i.e. AND-ing with zero ignores that bit
- `AND(1,x) = x`, i.e. AND-ing with one returns the value of the bit

The number `1 << i` is the number with `0` everywhere, except for a one in the `i`th position. Therefore `n & (1 << i)`ensures all bits except the `i`th bit is zero. Looking at an example for finding the 4th or 3rd bit:

11010001 n & 00010000 (1<<4) ========== 00010000 (n & (1 << 4))

11010001 n & 00001000 (1<<3) ========== 00000000 (n & (1 << 3))

If the \`i\`th bit is zero, then the output is zero. Checking if the output is non-zero will tell us the value of the \`i\`th bit.

Setting the `i`th bit / Turning the `i`th switch on

We can get a number that is `n` with the `i`th position changed to `1` with the code `n | ( 1 << i )`. Can you see why it works?

We are using the following properties to the `OR` function:

- `OR(0,x) = x`, i.e. OR-ing with zero returns the value of the bit.
- `OR(1,x) = 1`, i.e. OR-ing with one always gives one.

Looking at two examples:

```
11010001 n | 00010000 (1<<4) ==========   11010001 (n | ( 1 << 4))
```

```
11010001 n | 00001000 (1<<3) ==========   11011001 (n | ( 1 << 3))
```

Since every bit except the `i`th bit of `n` is OR-ed with zero, they are all copied into the new variable.

Clearing the `i`th bit of `n`/ Turning the `i`th switch off.

We can force a bit to zero using `AND(0,x) = 0`. We can preserve the other bits by using `AND(1,x) = x`. So if we can make a number `mask` that has all of its bits as `1` except the `i`th bit, we want to return `n & mask`.

To generate `mask`, note that we know how to set *only* the `i`th bit with `(1 << i)`. We want the bitwise NOT of this: `mask = ~(1 << i)`.

So our final result is `n & (~ (1 << i))`.

Clearing the `i` least significant bits of `n`/ Turning the first `i` switches off.

This is interesting as it shows how to set multiple values at once. We are interested in switching off the `i` rightmost (i.e. lowest power of 2) bits to zero. Forcing bits to zero suggests using AND. Ideally we would make a `mask = 1111..100000`where the rightmost `i` bits were zero, and all the others were one. Then our answer would be `n & mask`.

How do we create `mask`? If we start with a variable that is all `1`s, then we can left shift it `i` times to get the `i` rightmost bits set to `0`. The number that has all of its bits set to one for a *signed int* is `-1`. So `mask = (-1 << i)`.

Our final result is `n & (-1 << i)`.

Toggle the `i`th bit of `n` / toggling the `i`th switch

XOR is great here. Like OR, `XOR(0,x) = x` so we can use a mask with zero bits to preserve the bits we want to leave alone. We also have `XOR(1,x) = NOT(x)`! In fact, we can use this toggle property to eliminate the `~` operator altogether, because `~X` is the same as `X ^ (-1)`.

To toggle the `i`th bit of `n`, we would return `n ^ (1 << i)`.

Updating the `i`th bit of `n` to `x`.

Let's say that we know `x` is 1 or 0. How would we use it to set the `i`th bit to `x`, given that we used AND to clear the bit (set it to `0`) but OR to set the bit (set it to `1`)?

Here is one approach for what to return `(n & (~(1 << i))) | (x << i)`. What this does is force the `i`th bit to zero, then ORs it with `0` (if `x` is zero) or ORs it with a number with only the `i`th bit set, forcing that bit to 1 if `x` is 1.

Final comments on bits in Interviews

This is one of the longer overviews! There is a lot of background detail, and bit problems are generally *very* compact. The hard part isn't the length of the code that you're writing, but becoming comfortable thinking in bits, knowing some standard tricks, and getting a feel for when to use OR (forcing a bit to `1`), AND (forcing a bit to `0`), or XOR (forcing a bit to toggle).

Common Techniques: Advanced

Interview Essentials

There are some common approaches to problems that, when you use them, can dramatically improve your programs. These techniques are more advanced, because it takes practice to be able to identify problems where these approaches work well. Two tricky approaches that apply to a broad range of problems are:

- **Greedy Algorithms:** an approach to optimizing a "score" through a sequence of choices, which only uses the choice that makes the largest immediate improvement;
- **Divide and Conquer:** a technique that breaks problems into smaller (and easier) subproblems before merging them back to a solution of the original problem.

This section also looks at problems whose solutions utilize **disjoint sets**, which allow us to group a collection of objects into non-overlapping sets. This fast and powerful data structure enables us to quickly label objects as belonging togethe

RegEx

Interview Essentials

A *regex* is a string that encodes a pattern to be searched for (and perhaps replaced). Interviewers rarely test your ability to write a complicated Regex for regex sake, but they can be used in other interview tasks such as:

- Finding patterns of text, such as searching for the variable names in our code, or for formatted phone numbers in a long document;
- Writing database queries in text fields;
- Allowing you to validate input data quickly;
- Allowing to do bulk replacements, such as changing date formats throughout a document.

Proficiency with regex will enable you to use tools like `awk` and `sed` to quickly massage files into the desired format, use tools like `grep` to search for information more quickly, and enable you to perform bulk renaming of files. They also make the "search" feature of most text editors far more useful.

Writing a good regex is tough - you have to match everything you want, and nothing that you don't. If you don't feel confident with regex tricks, only try using very simple regexes during technical interviews to avoid getting flustered.

Regex is generally used interchangeably with the term *regular expression*, but there is a technical definition of regular expression that means the strings it searches for has to be identifiable with a finite state machine (i.e. it works on so-called *regular* languages). Most regexes in practical use are more general than the strict definition of a regular expression.

A *regex engine* interprets your regex string, and processes the other string to see if it is a match. Each programming language has a standard regex engine, and there is a slight variation amongst supported features and syntax, at least when using ASCII character sets.

In what follows, the regex strings will not be enclosed by double quotation marks. Even though the regexes are strings, there are different (non-portable) ways of expressing the strings to minimize the number of character escapes that need to be performed. For example, in Python it is standard to use "raw strings" to make a regex, so we would search for a digit using `r'\d'`. In this tutorial, we'll denote the same regex as `\d`.

We can broadly break the pieces of a regex up into:

- **Character classes:** What to specify, i.e. what characters we want to match (or exclude);
- **Quantifiers:** How many times we want a match to occur;
- **Anchors:** Where in the string we want the match to occur;
- **Modifiers:** Do we care about line-breaks? Do we care about case?

The syntax for modifiers varies the most, so we won't discuss them here. A regex doesn't need to specify all of the different attributes. For example, if we want to find the string `"CodeFights"` anywhere in our string, the regex `CodeFights` would work well. If we wanted to match only strings that started with CodeFights, we would use `^CodeFights`, where `^` plays the role of an anchor. By default, if no anchor is specified, most regex engines will look anywhere in the string for a match.

Character classes

Some general rules:

- Character classes are a group of characters where you can match or exclude any one of them.
- The simplest character class is just the character by itself, such as `h`.
- If you want to match multiple characters, you can include the characters you want in square brackets, such as `[012345]`. If you want to exclude certain characters, put a `^` at the beginning, such as `[^012345]`.
- **Important:** Inside a character class, `^` acts as a negation (i.e. exclude the following characters). Outside a character class, `^` acts as an anchor to the beginning of the string.
- Some character classes are used often, so there are special characters for them, such as `\d`, which is the same as `[0123456789]`. Negated versions of these special character classes are denoted with capital letters, such as `\D`.

Character class	**What it matches**
`a`	The literal letter `a` (as an example)
`[^a]`	Everything except `a`. The square brackets here are important. `^a` would match strings that *begin*with `a`!
`[a-z]`	Any lowercase letter `a` - `z`
`[^a-e]`	Everything except the letters `a-e`
`[a-zA-M]`	Any lowercase letter, or uppercase letters up to and including `M`

`\d`	Matches any digit. Shorthand for `[0-9]`
`\D`	Matches any non-digit. Shorthand for `[^\d]`
`\w`	Matches any "word" character. This is the same as `[a-zA-Z\d_]`
`\W`	Matches any character except "word" characters.
`\s`	Matches any whitespace character (space, tab, newline, carriage return, and vertical tab)
`\S`	Matches any non-whitespace character.
`.`	Matches any character (except newlines, in most regex engines)
`\1`	Matches the first "captured group"

Quantifiers

Often we are interested in matching more than one character. For example, if we are trying to match car license plates of the format "3 upper case letters, followed by 4 digits", we could use the ugly regex `[A-Z][A-Z][A-Z]\d\d\d\d` (but please see the footnote after the table!). A quantifier helps us with repeated regexes, so we could write this as `[A-Z]{3}\d{4}`instead. Quantifiers will also allow ranges, so if we knew that the plates we were looking for had 2 or 3 upper case letters, and between 1 and 4 digits, we could write the succinct `[A-Z]{2,3}\d{1,4}`.

Quantifier	**What it does**	**Example**
`X{n}`	Matches `n` copies of `X`	`\d{4}` matches 4 consecutive digits
`X{a,b}`	Matches between `a`copies of `X` and `b`copies of `X`	`\d{2,4}` matches any string with 2 to 4 consecutive digits
`X{a,}`	Matches a string that has `a` or more copies of `X`	`[a-zA-Z]{5,}` matches "words" (strings of letters, not including digits) that are at least 5 characters long.
`X*`	Matches 0 or more copies of `X`. Same as `X{0,}`.	`.*` matches anything (except newlines) as many times as needed. `Chocolate.*cookies` matches `Chocolate cookies`, `Chocolate chip cookies`, and `Chocolate-Pecan cookies`. Note in this case, `Chocolate.*?cookies` is probably a better regex, which is explained in "Greedy

		vs non-greedy matches".
`X+`	Matches 1 or more copies of `X`. Same as `X{1,}`.	`a+` matches at least one `a`, so `aaa` would match, `bac` would match, but `bbc` would not.
`X?`	Matches 0 or 1 copies of `X`. Same as `X{0,1}`. Can be thought of as making `X` optional	`colou?r` matches `color` and `colour`

Warnings on upper limits

The upper limit on characters can lead to unexpected results. For our license plate example, would you expect the regex `[A-Z]{3}\d{2,4}` to match the string "JKF224456"? Your first thought might be that it doesn't match because there are more than 4 digits after the three capital letters. It does match, which we can show by underling the match: "**JFK2244**56". The same regex also matches "ABCD**EFG1234**576". The problem with our match is that we haven't restricted what can appear before and after the four digits. (If you are wondering why the match is to "EFG1234" rather than just matching "EFG12", it is because regexes are *greedy* by default).

A solution you might try is to make sure whatever appears before the three capital letters **isn't** a capital letters, and whatever follows after the digits **isn't** a digit. The regex is `[^A-Z][A-Z]{3}\d{2,4}\D`. Let's check it on three strings:

- `[^A-Z][A-Z]{3}\d{2,4}\D` **matches** "The license plate ABC555 is for sale" (what we want).
- `[^A-Z][A-Z]{3}\d{2,4}\D` **doesn't match** "Did you know ABC12345666 isn't a valid plate number?" (what we want).
- `[^A-Z][A-Z]{3}\d{2,4}\D` **doesn't match** "ABC555" (**not** what we want).
- The problem in this example is that there is nothing before "ABC555", and our regex insists that there is something, and that something isn't a capital letter.

To solve this problem more generally we will use anchors (see the next section). The moral for this section is that upper limits on character classes can be misleading. Generally you can only trust them to match the exact number when you have specified what happens on the left **and** right of the part of the regex with the quantifier.

Anchors

Anchors are generally thought of as "locations" within the string.

Regex	**Meaning**	**Example**
`^`	Matches at the beginning of the line	`^abc` matches occurences of "abc" at the beginning of the string, or immediately after a newline.
`\A`	Matches at the beginning of the string	`\Aabc` matches "abc" if it is the

	(same as `^` if there are no newlines).	beginning of the string
`$`	Matches at the end of a line	`xyz$` matches occurences of "xyz" at the end of the string, or immediately before a newline.
`\Z`	Matches at the end of the string (same as `$` is there are no newlines).	`xyz\Z` matches "xyz" at the end of the string.
`\b`	Word boundary. Defined by only one edge side of this position being a word character (as defined by \w)	`\b[a-zA-Z]{7}\b` matches "iPhone7" and "Numbers are great", but not "iPhone12".
`\B`	Not a word boundary.	`\bman\B` matches "woman" and "batman", but doesn't match "mankind" or "man".

For our license plate problem, if we wanted to match strings that were *just* the license plate, we could use the regex `^[A-Z]{3}\d{2,4}`. If we wanted to look for strings that contained valid license plates, we could use word boundaries with the regex `\b[A-Z]{3}\d{2,4}\b`.

Greedy vs non-greedy matches

By default, a regex will try and make the largest match it possibly can. This is referred to as a *greedy* match. Consider the following string:

"I like chocolate chip cookies, chocolate pecan cookies, as well as raisin cookies"

What does the regex `chocolate.*cookies` match? By default, we have a greedy match, so what is the longest string that matches this pattern? The longest substring that starts with "chocolate" and ends with "cookies" is

"chocolate chip cookies, and chocolate pecan cookies, as well as raisin cookies"

The intent of this regex was probably to match all types of chocolate cookies. We can modify the regex using the `?`modifier after `*`. The regex `chocolate.*?cookies` makes smallest possible matches, so we end up matching the underlined portions of the string:

"I like chocolate chip cookies, chocolate pecan cookies, as well as raisin cookies"

Look-arounds

Usually when a regex makes a match (or an exclusion), later parts of the regex only reference later parts of the string. The purpose of a look-ahead regex is to determine if the rest of the string matches the regex, *and then returns to the original position*. A look-behind regex checks if positions prior to the current position match, without changing the current position.

There are 4 types of look-arounds:

- **Positive look-ahead:** `X(?=regex)` matches strings with the character `X` immediately followed by `regex`.
- For example, `\d+(?=\s*USD)` matches the 120 in "120 USD", and matches "$450USD". This can be used to capture just the amounts.

- **Negative look-ahead:** `X(?!regex)` matches strings with the character `X` that are **not** immediately followed by `regex`. To find all currencies that are not in USD, we could use `\$\d+(?!\s*USD)`.
- To match examples of `q` not followed by `u`, we could use `q(?!u)`. This is subtly different from `q[^u]`, because `q[^u]` asserts that `q` is followed by something (and that something isn't a `u`), so it would fail on a `q` at the end of a string.
- **Positive look-behind:** `(?<=regex)X` which matches strings with `regex` immediately followed by the character `X`.
- If we wanted to match all the different three letter currency codes, we could use `(?=\d+)\s*[A-Z]{3}`.
- **Negative look-behind:** `(?<!regex)X` which matches strings where the character `X` is not immediately preceded by `regex`.
- If you're trying to find words written in `UpperCamelCase`, the regex `[a-z]([A-Z][a-z]+)` won't match "Upper" because it is at the beginning of the string. To find something that isn't preceded by a capital letter, we could use the regex `(?<![A-Z])[A-Z][a-z]+` instead.

Examples

Regex	What it does
`\b[A-Z]\w*\b`	Matches capitalized words, where a word consists of letters, digits and underscores.
`\b(\w+)\s+\1\b`	Matches two repeated words, such as "In the the cupboard" (useful for finding typos)
`((?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#\$\!\^].{6,20}))`	Matches strings that are between 6 to 20 characters, and contain at least one digit, one lowercase letter, one uppercase letter, and one of the special characters "@#$!^". Useful as a password validator.

`/\b\d{2}(\d{2})\b/$1/g`	Replaces 4 digit numbers with the last two digits (e.g. 2001 -> 01). Doesn't replace numbers joined to words (e.g. ComicCon2012 would remain unchanged). PERL style replacement.
`(["'])[^\1]`*?CodeFights[^\1]*`?\1`	Matches quotes about CodeFights (i.e. CodeFights contained by either matching single or double quotation marks).
`[\(]?\d{3}[\)]?[\s-]*\d{3}[\s-]?\d{4}`	Matches 10 digit phone numbers, with optional brackets around the area code and optional spaces and dashes between the groups.

Number Theory

Interview Essentials

Number theory is the theory of integers. Knowing key results from number theory can help you write efficient algorithms. Number theory is has many somewhat esoteric facts that are simple once you see them, but are difficult to discover "on the spot". Fortunately, interviewers only expect you to have experience with a small subset of number theory.

You should practice calculating series (particularly the Fibonacci series), using modular arithmetic, and finding the greatest common divisor between numbers. You should also know a technique for generating prime numbers, such as the Sieve of Eratosthenes.

Your interviewer is probably happy to provide you with definitions of unfamiliar concepts in an interview, but being familiar with the topics above will help you understand your task much more rapidly than you would if you were going in with just the definitions.

Unless you're interviewing at a company that implements encryption, you're unlikely to see pure number theory problems in technical interviews. Instead, you can use results from number theory to help you solve your main problem.

Non-recursive series

Some of the standard series results allow you to figure out the sum of `n` objects in `O(1)` instead of `O(n)`. Some of the more commonly occurring series are:

- The sum of the first `n` natural numbers:

\sum_{i=1}^n i = \frac{n(n+1)}{2} $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- The sums of squares:

\sum_{i=1}^n i^2 = \frac{(2n + 1)(n+1)n}{6} $\sum_{i=1}^{n} i^2 = \frac{(2n+1)(n+1)n}{6}$

- The sums of cubes:

\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4} $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

- The geometric series:

\sum_{i=1}^n r^{i-1} = \frac{1-r^n}{1-r} $\sum_{i=1}^{n} r^{i-1} = \frac{1-r^n}{1-r}$

Primes

You should be able to:

- Find all the primes under `N`;
- Find the first `N` primes;
- Find the prime factorization of a number `N`.

The simplest algorithm for finding primes is the Sieve of Eratosthenes. It is often *introduced* as a way of finding all the elements that are less than or equal to `N` in the following way:

1. Write down all the numbers from `2` ... `N`.
2. Remove the smallest number in the array and store it as a prime, and then "cross out"/remove all higher multiples of that number up to `N`.
3. If there are numbers that have not been removed (either because they are primes, or multiples of primes), go back to step 2.

There are better ways of implementing it than by directly implementing the algorithm above. The main drawbacks of a direct implementation are that you have to allocate space for the integers `2` through `N`, and that you have to know where you want to stop.

An iterative implementation of the Sieve is:

1. Start with `primes = [2,3]`, and `candidate = 5`
2. Check if `candidate` is divisible by anything in the list `primes`:

- If it is, `candidate` is not a prime.
- If it isn't divisible by any smaller prime, it is itself prime. Add `candidate` to the list of primes.

1. If we still need more primes, then increase `candidate` by 2 (i.e. to the next *odd* number) and go back to step 2. Typical stopping conditions are:

- Stop when you have found the first `N` primes (e.g. `len(primes) == N`).
- Stop when you have found all primes less than `N`.

The advantages over the naïve implementation are that you don't need to keep all the integers in your Sieve in memory, and it becomes easier to implement a custom stopping condition. For example, you could see how you could use this method of generating primes to generate primes until you found a prime with the consecutive digits `314` (the first three digits in π) appeared.

(In case you're curious, the first prime with `314` to appear in it — in base 10 — is `13147`.)

Finding the greatest common divisor (Euclid's algorithm)

A number `d` is a divisor of `n` if `n / d` is an integer. Given two numbers `m` and `n`, the *greatest common divisor*`gcd(m,n)` is the largest number that is a divisor of both `m` and `n`. The GCD doesn't exist if `m` and `n` are both zero.

Note: Many number theory references won't assume a notion of division, because it is possible for `n / d` to be a non-integer even when `n` and `d` are both integers. So don't be surprised if you see the equivalent definition that "`d` is a divisor of `n` if there is some other integer `q` such that `d*q == n`".

To find the GCD of `m` and `n`, it helps to know that for any pair of positive integers `m` and `n`, we can write:

```
 m = q n + r,   q an integer,    0 ≤ r < n
```

Here `q` is called the *quotient* (the "number of times `n` divides `m`") and `r` is the *remainder*. In many computer languages that do integer division, we have:

q = m / n (integer division) = floor(m/n) (math division)

r = m % n (modulus operation)

Recast in this language, a number `d` divides `n` if and only if `n % d == 0`.

The key insight to finding the GCD of `m` and `n` is that if a number `d` divides `m` and `n`, then it must also divide the remainder because

```
 m % d = q (n % d) + (r % d) ==> 0 = q(0) + (r%d) ==> (r%d) == 0
```

Our goal is to find the largest such `d`. A couple of observations help:

1. If `r` is zero, then `n` divides `m`. Therefore `gcd(n,m) = n`, since the greatest divisor of any positive integer `n` is the integer itself.
2. If `r` isn't zero, then we can look for the greatest common divisor between `n` and `r`. Since this greatest divisor can be factored out, it is also a divisor of `m`.

Taken together, these observations imply:

```
gcd(m,n) = gcd(n, r),   where r = m % n
```

The first case is true because `gcd(n,0) = n` whenever `n` is a positive integer. We have to use this as a base case because we are writing an algorithm that works on *positive* integers.

This gives a recursive method for finding the GCD of two integers:

```
def gcd(m,n):
  n, m = abs(n), abs(m)
  if n + m == 0:
     return float('NaN')

  # base case: if m or n are zero, the other one
  # is the gcd
  if n * m == 0:
    return n + m
  return gcd(n, m % n)
```

This algorithm is known as *Euclid's algorithm*, and it can be shown to make at most `5 log₁₀(min(m,n)) + 1` recursive calls.

An important special case is when `gcd(a,b) = 1`, in which case the numbers `a` and `b` are called `coprime`. They have no factors in common except `1`, and *every* integer has `1` as a factor.

Modular arithmetic

Modular arithmetic looks at addition and multiplication of integers that "wrap around" a certain value called the modulus. This is sometimes called "clock arithmetic" because it is similar to the way that clocks work. For example, if it is now 20:00 (i.e. 8 PM) and we wait 6 hours, then it is 02:00 (i.e. 2 AM) even though `20 + 6 = 26`. Twenty-four hour time "wraps around" at 24 hours.

Technically, modular arithmetic deals with equivalence classes of integers. We call two numbers "congruent modulo `m`" if they have the same remainder after division by `m`. Using our example above, 26 and 2 are congruent modulo 24, because they both have remainder 2 after division by 24.

From a pragmatic (and interview-centric) point of view, we can think of integer arithmetic as "remainder arithmetic". That is, we are considering the addition and multiplication of remainders. The remainders $r_i \in \{0,1,2,3,...,m-1\}$.

If you use a language that has fixed size `unsigned ints`, then technically all integer arithmetic being performed with this native data type is modular arithmetic. (You can always build a `BigInt` class, or use a library if you need arbitrarily sized integers.) The `signed ints` also wrap around and would be examples of modular arithmetic, but the negative numbers mean we couldn't consider this the "arithmetic of remainders". One of the advantages offered by modular arithmetic is that you have an upper limit on the size of the integers you have to process, so particular operations are fast.

For addition, subtraction, and multiplication mod `m`, we can use the "normal" operation, and then take the modulus at the end. We can also just deal with the remainder of the inputs, and get the same value:

Operation mod `m`	**Normal operation, then mod `m`**	**Mod inputs**
Add `a` and `b` `mod m`	`(a + b) % m`	`( (a % m) + (b % m) ) % m`
Subtract `a` and `b` `mod m`	`(a - b) % m`	`( (a % m) - (b % m) ) % m`
Multiply `a` and `b` `mod m`	`(a * b) % m`	`( (a % m) * (b % m) ) % m`

If `a` and `b` are much larger than `m`, then the second column in this chart can be much more efficient since you only perform the arithmetic operation on numbers smaller than `m`. If you already know that `a` and `b` are numbers between `0` and `m-1` (inclusive), the middle row is more efficient.

There are a few counter-intuitive properties of modular arithmetic. Here are two of the more common ones. For the following, I will assume that `0 ≤ a,b < m` (i.e. `a` and `b` have already been converted into "remainders").

- The **zero-property** doesn't hold.
- In "regular arithmetic", we know that if `a*b = 0`, then one of `a` or `b` is zero. In modular arithmetic, `(3*2) % 6 == 0`, even through neither `2` or `3` are `0`.
    i. What we can say is that if `(a*b) % m == 0`, then `a*b` is a multiple of `m`, so `a` or `b` must share factors with `m`. If all the factors are in `a`, then `a` must be zero (we're dealing with the remainders here). Similar comments apply to `b`.
    ii. If `a` and `m` are *coprime* (i.e. `gcd(a,m) = 1` so we know `a` has no non-trivial common factors in common with `m`) then if `(a*b) % m == 0` we know that `b = 0`. In other words, `b` is a multiple of `m`, which means `b = 0` if `b` has been converted to a remainder.
- Division is weird. Consider the following three problems:
    i. `(3*a) % 6 = 0` which has a solution `a = 2` as well as a solution `a = 0`. The technique used above (do the regular operation and then take the modulus) misses the solution `a = 0`. Treating this like a regular equation, `3*a = 0`, and rearranging to get `a = (0/3) % 6 = 0` misses the solution `a = 2`.
    ii. `(3*a) % 6 = 1` has no solutions. For integer `a`, `3*a` is always a multiple of 3. Any multiple of 3 is either divisible by 6, or has a remainder of 3 after dividing 6; neither case will leave a remainder of 1.
    iii. `(3*a) % 6 = 3` has multiple solutions (namely `a = 1`, `a = 3`, `a = 5`).
- The idea that there can be either no solution or multiple solutions suggests that there isn't a sensible notation of division for dividing by 3 (at least in modulo 6).

To replace division, we look for multiplicative inverses. A number a^{-1} is the multiplicative inverse of `a` (modulo `m`) if $(a^{-1} * a) \% m = 1$. We can replace the idea of *dividing by a* with the notion of *multiplying by* a^{-1}, if it exists. The multiplicative inverse of `a` modulo `m` exists if and only if `gcd(a,m) = 1`. If there **is** a multiplicative inverse of `a`, you can be guaranteed that `(a * x) % m = b` has a unique solution for `x ∈ {0, 1, ..., m-1}`.

In particular, if working with modulo p for any prime p, we are guaranteed that every non-zero number $1, 2, \ldots, p-1$ has a multiplicative inverse modulo p.

Recursive sequence (with and without modular arithmetic)

A recursive sequence has two types of terms:

- The base cases, which are generally given explicitly.
- The recursive step, which tells us how to generate the i^{th} term of the sequence in terms of the previous terms, provided this is not one of the bases cases.

One of the most well-known recursive sequences are the *Fibonacci numbers* F_i, where:

$$F_i = F_{i-1} + F_{i-2} \quad (i > 1), \quad F_0 = 0, \quad F_1 = 1$$

The first two terms are the base case. From the base case, we can use the recursive step to find the other numbers in the sequence.

A related sequence, called the *Lucas numbers* L_i are defined by the following relation:

$$L_i = L_{i-1} + L_{i-2} \quad (i > 1), \quad L_0 = 2, \quad L_1 = 1$$

In other words, the recursive step is exactly the same, and only the base cases have changed.

While it's generally easy to write a recursive function directly from the definition, they often have poor runtimes. A recursively defined sequence where the a_nth term depends on a constant number j of previous terms has a runtime of $O(j^n)$. (For the Fibonacci and Lucas sequences, $j = 2$.) Using memoization, or an iterative approach, often works better. It can generate the term a_n in $O(n)$, assuming that the operations such as multiplication and addition take constant time (which isn't technically true and can become a major issue as the numbers become large).

Recursively defined sequences mod m

If a recursively defined sequence is defined in terms of the j previous terms, then when we have a series of $j$consecutive terms repeat, the sequence has to repeat since the recursive step would always put out the same numbers.

For a specific case, such as the Fibonacci numbers, if we have distinct i and j such that:

1. $F_i = F_j$
2. $F_{i+1} = F_{j+1}$
3. the recursive condition would guarantee that $F_{i+2} = F_{j+2}$, which in turn guarantees that $F_{i+3} = F_{j+3}$, and .... i.e., $F_{i+k} = F_{j+k}$ for $k \geq 0$.

This doesn't ever happen for the Fibonacci sequence, since the sequence is strictly increasing. However, if we look at the Fibonacci sequence modulo m, then for a consecutive pair of terms there are only m^2 possible pairs (m possibilities for the first number, and m for the second number). After at most m^2 terms, the Fibonacci sequence mod m **has** to repeat. If a series a_i that outputs integers depends on the previous j values, the series $a_i \% m$ will have to repeat after at most m^j terms.

Linear Diophantine, or integer, equations

Often we are interested in whether we can satisfy an equation with multiple variables using only integer quantities. An example of this sort of question is the "McNuggets question": If we are able to buy McNuggets in 4 packs, 6 packs, and 9 packs, am I able to buy exactly 41 McNuggets?

Assuming we only allow purchases of whole packs (and these are the only types available) we are asking if it is possible to solve

```
4 X + 6 Y + 9 Z = 41
```

where `X` represents the numbers of 4 packs, `Y` represents the numbers of 6 packs, and `Z` represents the numbers 9 packs. The answer is "yes", as `X = 2`, `Y = 1` and `Z = 3` works (as do other combinations).

Compare this to the seemingly similar

```
3 X + 6 Y + 9 Z = 41
```

which doesn't have any solutions, as the left-hand side is a multiple of three for any integer choices `X`,`Y`,`Z`, while the right-hand side isn't.

If we're just concerned with whether a solution $(x_1,x_2,\ldots x_N)$ exists for a problem of the form

$$A_1 x_1 + A_2 x_2 + \ldots + A_N x_n = C$$

for a known A_i, then there is a simple test. There are solutions if and only if

$$\text{gcd}(A_1, A_2, A_3, \ldots A_N) \text{ divides } C$$

Showing that this condition is necessary is easy. In the `3X + 6Y + 9Z` case, the left-hand side is a multiple of the GCD of all the coefficients `Aᵢ`, so for there to be a solution the right-hand side must be as well. It's a little trickier to show that this is also a sufficient condition.

The **extended Euclidean algorithm** solves this problem for two variables. Given two integers `a` and `b`, the extened Euclidean algorithm outputs `x`, `y`, and `gcd(a,b)` where `x` and `y` satisfy

```
A x + B y = gcd(a,b)
```

In contrast, the standard Euclidean algorithm only output `gcd(a,b)`. To see that this solves any equation of the form `AX + BY = C`, we know there is only a solution if `gcd(A,B)` divides `C`. If this isn't the case, we simply report no solutions. Multiplying the equation solved by the extended Euclidean algorithm by `C/gcd(A,B)` shows that `X = C x /gcd(A,B)` and `Y = C y/gcd(A,B)` are solutions to `A X + B Y = C`.

The recursive algorithm for the extended Euclidean algorithm can be made by first identifying a base case. Since a*a* and b*b*appear symmetrically in ax + by = gcd(a,b)*ax*+*by*=*gcd*(*a*,*b*), suppose b*b* is the smaller of the two, and that a >0*a*>0, b \geq 0*b*≥0. We have the base case:

- b = 0*b*=0, so gcd(a,b) = a*gcd*(*a*,*b*)=*a* and x = 1*x*=1. y*y* is multiplied by zero and is therefore arbitrary.

If we don't have the base case, we need to coerce the second argument to zero.

- If b\neq 0*b*≠0, we can break `a` into `a = qb + r` with 0 \leq r < |b|0≤*r*<|*b*|.
- From our discussion of `gcd(a,b) = gcd(b,r)`.

We can substitute `b` for `a` in our original equation:

$$a x + b y = gcd(a,b) \Rightarrow (qb + r) x + by = gcd(a,b) ax+by=gcd(a,b)\Rightarrow(qb+r)x+by=gcd(a,b)$$

Rearranging slightly:

b\underbrace{(q x + y)}_X + r \underbrace{x}_Y = gcd(a,b) = gcd(b,r)

That is, our equation is $bX + rY = gcd(b,r) bX+rY=gcd(b,r)$, which we can solve with a call to the extended Euclidean algorithm. We have replaced $a a$ with $b b$ (and we picked $b b$ to be the smaller coefficient), and we replaced $b b$ with $r r$ (where the remainder is guaranteed to be less that $b b$), so the second argument is getting smaller. There can be at most $|r| |r|$ more calls until the second argument reaches 0, getting to the base case. Once we have $(X,Y) (X,Y)$, we can reconstruct the original $(x,y) = (Y, X - qY) (x,y)=(Y,X-qY)$.

Here is a recursive implementation of the extended Euclidean algorithm:

```
def extended_gcd(a,b):
  " Returns (gcd(a,b), x, y) as solution to ax + by = gcd(a,b)"
  # error: a and b are both zero, gcd not defined
  if abs(a) + abs(b) == 0:
    return (float('NaN'),float('NaN'),float('Nan'))
  # base case
  if b == 0:
    return (a,1,0)

  # recurse, sending second argument to zero first
  # because of the way base case is arranged
  gcd_ab, x, y = extended_gcd(b, a % b)
  return (gcd_ab, y, x - (a/b)*y)
```

Note that there are a few details from our original motivation that didn't make it into the code. For example, we never explicitly check that `b` is smaller than `a`. Once the first recursive call `extended_gcd(b, a%b)` has been made, the second argument is smaller than the first. Once a solution (x,y) has been found, other solutions can be found via

$$x_i = x + bi, \quad\quad\quad y_i = y - ai$$

With multiple variables, a "merge and conquer" approach can be taken. For example

$$A X + B Y + C Z = N \Rightarrow AX + gcd(B,C)(b Y + c Z) = N, \quad\quad\quad b = \frac{B}{gcd(B,C)}, \quad\quad\quad c = \frac{C}{gcd(B,C)}$$

can be solved using the extended Euclidean algorithm for X and $Q = bY + cZ$. Once complete, the extended Euclidean algorithm can be used again to solve $bY + cZ = Q$.

Counting

Interview Essentials

Many interview questions are dependent on you being able to calculate the number of something, or to calculate the number of arrangements that satisfy particular properties. Often these numbers are huge, and it isn't feasible to generate all the solutions by brute force.

Interviewers are looking to see if you know how look at a problem and determine if the order of items in an arrangement is important, and if it isn't. Once you have determined what to count, they will be looking for your ability to take a naive solution and optimize it using more advanced methods.

The Multiplication Principle

If there are n_1 ways to select item 1, and n_2 ways to select item 2, then there are n_1n_2 ways to select pairs of items (if the order matters).

To convince yourself of this, select a particular item for item 1. Then there are n_2 choices for item 2. A different choice of item 1 will still have n_2 choices for item 2. Since there are n_1 different choices for item 1, and each choice contributes n_2 new pairs, there are n_1n_2 distinct pairs.

As an example, suppose we had to choose a letter from $\{A,B,C,D\}$ (item 1) and a number from $\{1,2,3\}$. Since there are 3 possible numbers for each letter picked, there are twelve possible pairs:

Letter	Number
A	1
A	2
A	3
B	1
B	2
B	3
C	1
C	2

C	3
D	1
D	2
D	3

Important: The multiplication principle only holds if the *number of choices* for each item doesn't depend on the item chosen. Here are two examples:

1. Pick a letter from \{A,B,C\}$\{A,B,C\}$ and a number from \{1,2,3\}$\{1,2,3\}$ *and* the number cannot be the 1-based index of the letter in the alphabet (i.e. A1, B2, C3 are excluded).
2. The multiplication rule *does* hold, because regardless of which of the 3 letters we pick, there are 2 numbers we can use. There are 3 \times 2 = 6$3 \times 2 = 6$ possible combinations.
3. Pick a letter from \{A,B,C,D\}$\{A,B,C,D\}$ and a number from {1,2,3} *and* the number cannot be the 1-based index of the letter in the alphabet (i.e. A1, B2, C3 are excluded).
4. The multiplication rule *doesn't* hold, because the if we pick 'A', 'B' or 'C' there are 2 numbers we can choose, but if we pick 'D' then there are 3 numbers we can pick from.

The Addition Principle

If we are counting items that are in non-overlapping classes, the total number of items is just the sum of items in each class. A simple example: If the only places to eat in town are 3 fancy restaurants and 2 fast food joints, then there are 3+2 = 53+2=5 total places to eat (assuming that no fast food joint qualifies as "fancy").

If there are overlaps, then we need to subtract all the items that are double-counted. For example, if we have 8 pasta dishes, and 4 spaghetti dishes, then we only have 8 dishes, not 12, since every spaghetti dish is also a pasta dish. Our addition method of counting for two classes is:

\text{(Things in class 1 or 2)} = \text{(Things in class 1)} + \text{(Things in class 2)} - \text{(Things in both class 1 and 2)}(Things in class 1 or 2)=(Things in class 1)+(Things in class 2)−(Things in both class 1 and 2)

If the classes don't overlap, we get simple addition. In the case of more than two classes with overlaps, the more complicated *inclusion-exclusion* principle tells us how to systematically count.

The addition principle can be used alongside the multiplication principle. Taking our previous example of picking a letter from \{A,B,C,D\}{*A*,*B*,*C*,*D*} and a number from {1,2,3} with the restriction that we cannot pair a letter with the 1-based indexed letter of the alphabet, we could approach the problem in the following way:

- Pick a letter from \{A,B,C\}{*A*,*B*,*C*} and a valid number from \{1,2,3\}{1,2,3}. There are 3 choices for the letter, and regardless of which of these letters we pick, 2 choices for the number. Therefore there are $3\times 2 = 6$ choices.

- Then pick the letter from $\{ D \}$, and any number from $\{1,2,3\}$. There is one choice for the letter, and three for the number, so the multiplication principle tells us we have $1 \times 3 = 3$ choices.
- Pairs that start with one of $\{A,B,C\}$ are distinct from pairs that start with D. Therefore we can use the addition principle to count $6 + 3 = 9$ pairs that fit the above constraints.

The Subtraction Principle

If we have N items, and a constraint excludes n of them, then we have $N-n$ items from our original set that obey the constraint. For example, suppose we wanted to count the number of days of the week that don't contain the letter 'u' in their English names. We could either count the number of days that don't have 'u' directly to get 3 (Monday, Wednesday, Friday), or we could count the number that do have a 'u' (Tuesday, Thursday, Saturday, Sunday) and subtract that number from 7: $7 - 4 = 3$.

In the "days of the week" example, both approaches are approximately the same amount of work. If we wanted to count the number of "non-February" days in a year, then we could start with 365 days in a year, and subtract the 28 days in February to get $365-28 = 337$. For leap years we have $366 - 29 = 337$ non-February days as well.

The subtraction principle is most useful when combined with the addition or multiplication principles. Going back to the example of picking a letter from $\{A,B,C,D\}$ and a number from $\{1,2,3\}$ such that we don't pair a letter with its 1-based index in the alphabet, we can count the following way:

- First, ignore the constraint. The multiplication principle tells us there are $4 \times 3 = 12$ possible pairings;
- There are three pairings that are not allowed: A1, B2, C3. The subtraction principle tells us that the number of allowed pairings is $12 - 3 = 9$.

Permutations

A permutation of n objects is a way of ordering those objects.

Permutations of Distinct Objects

If they are all distinct, then there are $n!$ distinct permutations. This follows directly from the multiplication principle. Imagine selecting a particular permutation in order (without replacement):

- The first item can be any of the n items;
- The second item can be any of the remaining $n-1$ items;
- The third item can be any of the remaining $n-2$ items;
- ...
- Until you reach the last item, for which there is one choice left.

The multiplication principle tells us that there are n(n-1)(n-2) .... (2)(1) = n!$n(n-1)(n-2)....(2)(1)=n!$

possible permutations.

Often we are interested in the question how many permutations there are if we select fewer than nn items. For example, there is a variant of the gambling game Lotto where we have to pick the numbers drawn in the order they were drawn. If there are 60 possible numbers, and 6 are drawn, we are interested in how many permutations there are of 6 objects drawn from 60. From the multiplication principle:

- There are 60 choices for the first ball;
- There are 59 choices for the second ball;
- ...
- There are 55 choices for the sixth ball.
- Therefore the number of permutations is

\prod_{i=55}^{60} i = 60 \times 59 \times 58 \times 57 \times 56 \times 55 = 36,045,979,200$\prod_{i=55}^{60} i=60\times59\times58\times57\times56\times55=36,045,979,200$

There is a simpler way of writing this by multiplying and dividing by 54!:

\frac{60 \times 59 \times 58 \times 57 \times 56 \times 55 \times 54!}{ 54!} = \frac{60!}{54!}$\frac{60\times59\times58\times57\times56\times55\times54!}{54!}=\frac{60!}{54!}$

There is an alternative way of thinking of this formula. Imagine the permutations of all 60 objects (of which there are 60!). You only care about the first 6 objects drawn, so each way of shuffling the remaining 54 objects is unimportant to you. So each permutation of

6 objects you care about appears on the list of 60! total permutations 54! times. By dividing 60! by the 54! repeats, you get the number of permutations of the first 6 objects.

In general, the formula for counting the number of permutations for selecting r objects from N is:

$${}^NP_r = \frac{N!}{(N - r)!} = \prod_{i=N-r}^N i$$

Important: The factorial notation $N!/(N-r)!$ is tidier notation, and easier to use when trying to simplify expressions. If you are implementing a permutation function on a computer, the "product" method is preferred because there are fewer multiplications, and you are not as likely to cause an overflow error during your calculation. For example, if calculating the "trivial" problem of the number of permutations of 2 objects drawn from 60, calculating $60 \times 59 = 3540$ is not going to overflow, but calculating the mathematically equivalent $60! / 58!$ requires being able to handle integers up to 8×10^{81}.

Permutations of Objects with Duplicates

In interview problems, you may see problems where you are have identical objects. There may be problems such as:

- How many distinct anagrams of "ANAGRAMS" are there?
- If you're given a bag with 3 red balls and 4 blue balls, how many different permutations can you make when drawing all 7 balls from the bag?

The general approach to these questions is to over-count, and then determine by what factor you have over-counted. To find the number of distinct anagrams of "ANAGRAMS", note that "ANAGRAMS" is an 8 letter word. If all the letters were distinct, then there would be 8! = 40320 anagrams. Because there are 3 'A's in the word, any permutation among only the "A"s leaves the string the same. Every valid anagram has 3! ways of rearranging the "A"s, so

$$\text{Num distinct anagrams of "ANAGRAMS"} = \frac{8!}{3!} = 6720$$
Num distinct anagrams of "ANAGRAMS"=3!8!=6720

When selecting the 7 balls from the bag, if they were each distinct then there would be 7! = 5040 different permutations. Because we can shuffle the red balls among themselves 3! ways, and the blue balls among themselves 4! ways, the multiplication principle tells us there are $3! \times 4!$ ways of rearranging the balls without changing the order in which the colors appeared. Therefore:

$$\text{Num distinct permutations of 3 red balls and 4 blue balls} = \frac{(3+4)!}{3!4!} = \frac{7!}{3!4!} = 35$$
Num distinct permutations of 3 red balls and 4 blue balls=3!4!(3+4)!=3!4!7!=35

The general formula is if you have N items, and each distinct item i appears n_i times, then:

$$\text{Num of permutations of all items} = \frac{N}{n_1! n_2! \cdots n_j!}$$
Num of permutations of all items=n1!n2!⋯nj!N

where there are j distinct items. Note that we have $N = \sum_{i=1}^j n_i$. In the context our our examples:

- **Finding the anagrams of "ANAGRAMS"** There are 8 letters in "anagrams", of which $j = 6$ of them are distinct. We have $n_A = 3$, $n_N = 1$, $n_G = 1$, $n_R = 1$, $n_M = 1$, and $n_S = 1$. The number of distinct anagrams is:

$$\frac{8!}{3!1!1!1!1!1!} = 6720$$

- **Finding the number of arrangements of the 7 balls** There are 7 balls, and two distinct types. There are $n_{\text{red}} = 3$ balls and n_{blue} balls. The number of arrangements is:

$$\frac{7!}{3!4!} = 35$$

A more challenging permutation problem might be to ask for the number of different possible permutations of 5 balls from the collection of 4 blue balls and 3 red ones. This is tricky because we don't know out of the 5 selected balls how many are red and how many are blue. One way of approaching this problem is to list how many red balls and how many blue balls we could have to make 5 total, and then use our formula for permutations to determine how many permutations there are of each of these arrangements:

Red Balls	**Blue balls**	**Number of distinct permutations**

1	4	5!/(1! \times 4!) = 55!/(1!×4!)=5
2	3	5!/(2! \times 3!) = 105!/(2!×3!)=10
3	2	5!/(3! \times 2!) = 105!/(3!×2!)=10

Since each of these selections is non-overlapping (no arrangement with exactly 3 red balls is the same as an arrangement with exactly 2 red balls) we can apply the addition rule to get 25 different possible permutations of 5 balls.

Writing a Function for Permutations

Sometimes we actually need to generate the permutations. Even if we are only looking to count the number of permutations that satisfy some condition, we may need to look at the actual permutation to determine if it satisfies our constraints. Be warned that the number of permutations of nn distinct objects is n!$n!$, which grows extremely rapidly, so this is not normally a scalable solution.

It can be practical if you can write your constraints in such a way that you can determine if there are any feasible permutations by looking at a prefix of a permutation. That way you can stop iterating down a particular branch early.

For languages that have generators, we can generate all permutations in a memory friendly way. Using Python as the prototype language:

```
def permutation(my_list, start_index = 0):
    if start_index == len(my_list):
        yield my_list
    else:
        for index in range(start_index, len(my_list)):
            # if my_list[:start_index] isn't viable we can break
            # and not compute the remaining permutations
            my_list[start_index], my_list[index] = my_list[index],
my_list[start_index]
            for i in permutation(my_list, start_index + 1):
                yield i[:]
            my_list[start_index], my_list[index] = my_list[index],
my_list[start_index]
```

The nice feature of this approach is that the prefix is the list up to and including `start_index`, which allows us to prune branches and "exit early" if we can determine that it is impossible to construct permutations that satisfy our constraints.

(If you are using Python, rather than just a generator-friendly language, and are allowed to use whatever external libraries you like, you should definitely use `itertools`.)

If you are using a language without generators, then it is probably easier to use an accumulator pattern, where the permutations are stored in results.

```
def permutation(my_list, results, start_index = 0):
    if start_index == len(my_list):
        return my_list[:]
    else:
        for index in range(start_index, len(my_list)):
            # again, we can test the prefixes here to return from the
            # function if we can determine the permutations with the current
            # prefixes are not viable with our constraint.
            my_list[start_index], my_list[index] = my_list[index],
my_list[start_index]
            new_perm = permutation(my_list, results, start_index + 1)
            if new_perm:
                results.append(new_perm)
            my_list[start_index], my_list[index] = my_list[index],
my_list[start_index]
```

Combinations

Combinations are arrangements of items where what matters is what is selected, not the order in which they are selected. For example, when selecting optional extras for your car, it doesn't matter which order you select them in. If you are picking up fruit for a fruit salad, it doesn't matter which order you selected the fruit.

Sometimes emotional aspects of decisions can cloud whether or not order matters when counting arrangements. For example, when counting the number of possible soccer teams we can make, we would be interested in which group of 7 people are all playing together, not the order they were picked. However, when actually making the soccer team, we are used to the order being important (the better players usually get selected first). When counting the number of *possible*teams, order doesn't matter. When actually making the teams on the field, order *does* matter!

With permutations, we talked about the number of permutations of nn distinct objects (of which there are n!) and the number of permutations where we select rr of nn objects (of which there are {}^nP_r = n!/(n-r)!${}^nP_r=n!/(n-r)!$). For combinations, there is only one combination of nn objects (simply take all of them). The more interesting question is how many combinations are there where we take rr objects from nn. There are two notations for this: {}^nC_r${}^nC_r$ and {n\choose r}$\binom{n}{r}$. With our knowledge of permutations, we can find the formula for the number of combinations easily:

\text{Number of ways of picking r out of n distinct objects} = {n \choose r} = \frac{n!}{(n-r)! r!}$\text{Number of ways of picking r out of n distinct objects}=\binom{n}{r}=\frac{n!}{(n-r)!r!}$

We see that it is the number of permutations we could make (n! / (n-r)!$n!/(n-r)!$) divided by the number of different ways we could have arranged those rr objects. This makes sense because each of those r!$r!$ different permutations of the rr objects is still the same combination, so every combination has been over-counted r!$r!$ times.

Probability, Combinations vs Permutations

When there are identical objects, it becomes a lot trickier to give a general prescription for counting objects. For example, when selecting the 5 balls from 3 red balls and 4 blue balls, there were 3 combinations:

- 1 red ball, 4 blue balls (5 permutations)
- 2 red balls, 3 blue balls (10 permutations)
- 3 red balls, 2 blue balls (10 permutations)

Each of these different combinations had a different number of permutations.

When asked probability questions, it is usually assumed that each *permutation* is equally likely, but that we are interested in the probability of a particular combination. If we took our bag of 3 red balls and 4 blue balls, and asked for the probability that a "random draw" of 5 balls would have 1 red ball, and 4 blue balls the expected answer would be 1/5, because 5 of the 25 possible permutations make this combination.

A more extreme example is looking at the number of heads you get when flipping 100 coins. There are 101 different combinations (0 heads, all the way up to 100 heads), but 2^{100}2100 different permutations. Assuming the coin is fair, the chance of a particular total number of heads coming up is equal to the fraction of permutations that have that many heads.

What to Expect for an Interview

You should be comfortable writing your own permutation and combination functions. Even outside the realm of counting, it is one of the more challenging recursive problems you can expect to see on an interview. Most places you interview will want you to have some idea about the difference between the ideas of combinations and permutations, but only companies whose work is highly mathematical or algorithm-driven will expect you to have memorized the formula for counting the number of permutations and combinations. It's in the Extra Credit plan of Interview Practice for a reason!

Here are the techniques related to generating combinations and permutations that you should know:

- Writing a function that can generate the permutations of a list;

- Writing a function that can generate the combinations of a list;
- Writing a function that can generate the power set of a set (that is, the list of all possible 2^n2^n subsets of a set with n$n$items).

Geometry

Interview Essentials

In technical interviews, geometry problems act as both general knowledge questions (do you remember what a scalene triangle is?) and as a source for challenging coding questions. From the interviewer's perspective, geometry questions are useful because they don't require much explaining or many data structures beyond "points" and "lines".

An interviewer is interested in how effectively you use your knowledge about the problem to eliminate solutions. When looking for the intersection of multiple lines, how do you deal with the approximate nature of floating point arithmetic - or do you start with a more robust approach? When looking at a collection of N points to see if four of them can make a rectangle, do you iterate through all the different ways of connecting each set of four, or do you eliminate possibilities using the distances between the points first? Do you check the corner cases (no pun intended)?

Crucial Terms

Interviewers are usually happy to explain geometric terms or concepts that you’re not familiar with, but in general you should be very familiar with the following terms.

- Plane: A flat, 2D surface
- Point: A location on a plane

- Points on a plane are represented by two numbers, the x and y axes, which are often presented as an array
- Equilateral triangle: All sides are equal length and all angles are equal
- Scalene triangle: All sides and all angles are unequal
- Right triangle: One internal angle is equal to 90 degrees
- Vertex: The intersection point of two sides of a plane figure
- Right angle: 90 degrees
- Acute angle: Between 0 and 90 degrees
- Obtuse angle: Between 90 and 180 degrees

Important Concepts

As with any interview question type, you should always be thinking about optimizing the time and space complexity of your solution. For instance, if you're trying to determine whether you can connect 4 points out of a collection of N points to create a parallelogram, a naive solution might be to iterate through all possible ways of connecting the points, for a time complexity of $O(N^2)$, where N^2 is the number of possible combinations. Then sort the edge lengths, looking for two pairs of edges of the same length, and where the 4 edge lengths are between only four points. A better solution would be to eliminate some combinations right away using the distances between the points.

For An Interview, Expect To See Problems Like...

- Given a set of points, determine whether they contain a specific shape;
- Connect n points and determine whether they create a specific shape;
- Find the area of a region inscribed by circles and lines;

- Given a set of rules (for example, a viewing angle equal to n degrees), determine whether one point is “visible” from another point;
- Divide a shape into a certain number of pieces.