Data Structures and Algorithms

Contents

1 Pre-requisite Concepts

1.1 Runtime Analysis

1.1.1 Asymptotic Analysis (Big O)

We can measure the growth rate of the time or space complexity of an algorithm using an upper bound ($\mathcal{O}(f)$), lower bound ($\Omega(f)$) or a tight bound ($\Theta(f)$) on the best, worst or average case run time. When analysing an algorithm we typically use an upper bound on the worst case.

$$\mathcal{O}(1) \leq \mathcal{O}(\log n) \leq \mathcal{O}(n) \leq \mathcal{O}(n \log n) \leq \mathcal{O}(n^2) \leq \mathcal{O}(2^n) \leq \mathcal{O}(n!)$$

$\mathcal{O}(1)$ - constant time	$\mathcal{O}(n^2)$ - quadratic time
$\mathcal{O}(\log(n))$ - logarithmic time	$\mathcal{O}(n^c)$ - polynomial time
$\mathcal{O}((\log(n))^c)$ - polylogarithmic time	$\mathcal{O}(c^n)$ - exponential time
$\mathcal{O}(n)$ - linear time	$\mathcal{O}(n!)$ - factorial time

1.1.2 Amortized Analysis

If the cost of an action has high variance, i.e. its computation is often inexpensive but is occasionally expensive, we can capture its expected behaviour using an amortized time value. If we let $T(n)$ represent the amount of work the algorithm does on an input of size n, An operation has amortized cost $T(n)$ if k operations cost $\leq k \cdot T(n)$. $T(n)$ being amortized roughly means $T(n)$ is averaged over all possible operations.

For example, a dynamic array will copy over elements to an array of double its size whenever an insert is called on an already full instance, otherwise it will simply insert the new element. For n insertions, this happens on every $2, 4, 8, ..., n$ element.

$$T(n) = \mathcal{O}(n + \frac{n}{2} + \frac{n}{4} + \cdots + 1) = \mathcal{O}(2n)$$

Therefore, an insertion takes $\mathcal{O}(n)$ time in the worst case but the amortized time for each insertion is $\mathcal{O}(1)$.

A data structure realizing an amortized complexity of $\mathcal{O}(f(n))$ is less performant than one with a worst-case complexity is $\mathcal{O}(f(n))$, since a very expensive operation might still occur, but it is better than an algorithm with an average-case complexity $\mathcal{O}(f(n))$, since the amortized bound will achieve this average on any input.

1.1.3 Logarithmic Runtime

When encountering an algorithm in which the number of elements in the problem space is halved on each step, i.e. in a divide and conquer solution like binary search, the algorithm will likely have a $\mathcal{O}(\log n)$ or $\mathcal{O}(n \log n)$ run-time. We can think of $\mathcal{O}(n \log n)$ as doing $\log n$ work n times.

Again, if we let $T(n)$ represent the amount of work the algorithm does on an input of size n,

$$\begin{aligned} T(n) &= T(n/2) + \Theta(1) \\ &= T(n/4) + \Theta(1) + \Theta(1) \\ &= \Theta(1) + \cdots + \Theta(1) \\ &= \Theta(\log n) \end{aligned}$$

1.2 Computational Complexity Theory

1.2.1 Complexity Classes

$$P \subseteq NP \subseteq EXP \subseteq R$$

1. P: The set of problems that can be solved in polynomial time.
2. NP: The set of decision problems that can be solved in non-deterministic polynomial time via a "lucky" algorithm.
3. EXP: The set of problems that can be solved in exponential time.
4. R: The set of problems that can be solved in finite time.

1.2.2 NP Problems

Nondeterminsitic Polynomial (NP) problems follow a nondeterministic model in which an algorithm makes guesses and produce a binary output of YES or NO. These are the simplest interesting class of problems and are known as decision problems. A "lucky" algorithm can make guesses which are always correct without having to attempt all options. In other words, NP is the set of decision problems with solutions that can be verified in polynomial time.

P vs. NP asks whether generating proofs of solutions is harder than checking, i.e whether every problem whose solution can be quickly verified can also be solved quickly. NP-hard problems are those at least as hard as all NP problems. NP-hard problems need not be in NP; that is, they may not have solutions verifiable in polynomial time. NP-complete problems are a set of problems to each of which any other NP-problem can be reduced in polynomial time and whose solution may still be verified in polynomial time. In fact, NP-complete = NP $\cap$ NP-hard.

1.2.3 Reductions

A reduction is an algorithm for transforming one problem into another problem for which a solution or analysis already exists (instead of solving it from scratch). A sufficient reduction from one problem to another can be used to show that the second problem is at least as difficult as the first.

NP-complete problems are all interreducible using polynomial-time reductions (same difficulty). This implies that we can use reductions to prove NP-hardness. A one-call reduction is a polynomial time algorithm that constructs an instance of X from an instance Y so that their optimal values are equal, i.e. X problem $\implies$ Y problem $\implies$ Y solution $\implies$ X solution. Multicall reductions instead solve X using free calls to Y – in this sense, every algorithm reduces the problem and model of computation.

1.3 Algorithm Design

1.3.1 In-place algorithm

An in-place algorithm is an algorithm that transforms input using no auxiliary data structure, though a small amount of extra storage space is allowed for a constant number of auxiliary variables. The input is usually overwritten by the output (mutated) as the algorithm executes. An in-place algorithm updates input sequence only through replacement or swapping of elements.

1.3.2 Time–Space– tradeoff

A time-space or time–memory trade-off is a case where an algorithm trades increased space usage with decreased time complexity. Here, space refers to the data storage consumed in performing a given task (RAM, HDD, etc), and time refers to the time consumed in performing a given task (computation time or response time).

1.3.3 Heuristics

A heuristic is a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution. This is achieved by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut.

1.3.4 Hash Functions

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. A good hash function satisfies two basic properties: it should be very fast to compute and it should minimize duplication of output values (**collisions**). For many use cases, it is useful for every hash value in the output range to be generated with roughly the same probability. Two of the most common hash algorithms are the MD5 (Message-Digest algorithm 5) and the SHA-1 (Secure Hash Algorithm).

1.3.5 Stack vs. Heap Memory Allocation

The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block of fixed size is reserved on the top of the stack for local variables and some bookkeeping data. When that function returns, the block becomes freed for future use. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed. This makes it really simple and fast to keep track of and access the stack; freeing a block from the stack is nothing more than adjusting one pointer. Also, each byte in the stack tends to be reused very frequently which means it tends to be mapped to the processor's cache, making it very fast.

The heap is memory set aside for dynamic allocation by the OS through the language runtime. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap. The size of the heap is set on application startup, but can grow as space is needed.

This makes it much more complex to keep track of and access which parts of the heap are allocated or free at any given time. Another performance hit for the heap is that the heap, being mostly a global resource, typically has to be multi-threading safe.

2 Data Structures and ADTs

An **abstract data type (ADT)** is a theoretical model of an entity and the set of operations that can be performed on that entity

A **data structure** is a value in a program which can be used to store and operate on data, i.e. it is a programmed implementation of an ADT.

Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.

Linked data structures are composed of distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists. Recall, a pointer is a reference to a memory address which stores some data.

2.1 Lists and Arrays

2.1.1 List

A list is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. Lists are a basic example of containers, as they contain other values. Their operations include the following,

- **isEmpty(L)**: test whether or not the list is empty
- **prepend(L, item)**: prepend an entity to the list
- **append(L, item)**: append an entity to the list
- **get(L, i)**: access the element at a given index.
- **head(L)**: determine the first component of the list
- **tail(L)**: refer to the list consisting of all the components of a list except for its first (head).

A **self-organizing list** is a list that reorders its elements based on some self-organizing heuristic to improve average access time. The aim of a self-organizing list is to improve efficiency of linear search by moving more frequently accessed items towards the head of the list. A self-organizing list achieves near constant time for element access in the best case and uses a reorganizing algorithm to adapt to various query distributions at runtime.

2.1.2 Arrays

An array is a data structure implementing a list ADT, consisting of a collection of elements (values or variables), each identified by at least one array index or key.

A **bit array** (a.k.a bit map or bit mask) is a data structure which uses an array of 0's and 1's to compactly store information. An index j with a value of 1 indicates the presence of an integer corresponding to $j \in \mathbb{Z}$, operations performed on these binary arrays are analogs to set operations. We can extend the bit array to store arbitrary integers. When given a constraint on possible values that need to be stored and analyzed, we can initialize an array with a size of the range of max possible values and record frequencies directly in their corresponding index, eliminating the need to re-order an unsorted array or maintain a sorted order on every insertion.

A **dynamic array** is a data structure that allocates all elements contiguously in memory and keeps a count of the current number of elements. If the space reserved for the dynamic array is exceeded, it is reallocated and (possibly) copied, which is an expensive operation. Though its amortized insertion cost is equal to a static array, $\Theta(1)$. Python's "List" data structure is a dynamic array which uses **table doubling** to support its constant amortized operations. It is possible to implement real-time table doubling to support constant time worst-case insertions by incrementally building up a larger array on initial insertions which we can then quickly switch to when the original array is full.

Time Complexity of List operations

Operation	**Average Case**	**Amortized Worst Case**
Copy	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Append	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Pop last	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Pop intermediate	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Insert	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Get Item	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Set Item	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Delete Item	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Iteration	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Get Slice	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Del Slice	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Set Slice	$\mathcal{O}(k+n)$	$\mathcal{O}(k+n)$
Extend	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Multiply	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$
x in s	$\mathcal{O}(n)$	
min(s), max(s)	$\mathcal{O}(n)$	
Get Length	$\mathcal{O}(1)$	$\mathcal{O}(1)$

2.1.3 Linked Lists

A linked list is a data structure that represents a sequence of nodes. In a singly linked list each node maintains a pointer to the next node in the linked list. A doubly linked list gives each node pointers to both the next node and the previous node. Unlike an array, a linked list does not provide constant time access to a particular "index" within the list, i.e. to access the Kth index you will need to iterate through K elements. The benefit of a linked list is that inserting and removing items from the beginning of the list can be done in constant time. For specific applications, this can be useful. Linked structures can have poor cache performance compared with arrays. Maintaining a sorted linked list is costly and not usually worthwhile since we cannot perform binary searches.

A **doubly linked list** maintains a reference to the node previous to it, allowing for bidirectional traversal at the expense of additional book-keeping on insertions and deletions. Doubly linked lists are commonly found in least-recently used (LRU) and least frequently used (LFU) caches where they maintain an ordering of keys for the caching policy. Although double-ended queues support removing and adding items to the head and tail, removing items from middle of a queue to the head or tail is less performant than in a doubly linked list.

Python Implementation

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

2.1.4 Skip Lists

As an alternative to balanced trees examined later, a hierarchy of sorted linked lists is maintained, where a random variable is associated to each element to decide whether it gets copied into the next highest list. This implies roughly $\log n$ lists, each roughly half as large as the one above it. A search starts in the smallest list. The search key lies in an interval between two elements, which is then explored in the next larger list. Each searched interval contains an expected constant number of elements per list, for a total expected $\mathcal{O}(\log n)$ time for lookups, insertions, and removals.

This structure is well-suited for solving problems like tracking a running median which requires sort order to be maintained as new items are added and old items are deleted and which requires fast access to the n-th item to find the median or quarterlies. The primary benefits of skip lists for other purposes achievable by B+trees are ease of analysis and implementation relative to balanced trees.

2.2 Stacks and Queues

2.2.1 Stacks

A stack is an ADT container that uses last-in first-out (LIFO) ordering, i.e. the most recent item added to the stack is the first item to be removed. It supports the following operations:

- **pop()**: Remove the top item from the stack.
- **push(item)**: Add an item to the top of the stack.
- **peek()**: Return the top of the stack.
- **isEmpty()**: Return true if and only if the stack is empty.

Unlike an array, a stack does not offer constant-time access to the ith item. However, it does allow constant time adds and LIFO removals since it doesn't require shifting elements around. One case where stacks are often useful is in certain recursive algorithms where we need to push temporary data onto a stack as we recurse and then remove them as we backtrack (for example, because the recursive check failed). A stack offers an intuitive way to do this. A stack can also be used to implement a recursive algorithm iteratively which is what's otherwise done in a function's call stack. It is occasionally useful to maintain two stacks in order to have access to a secondary state, like the minimum/maximum, in constant time.

2.2.2 Queues

A queue is an ADT container that implements FIFO (first-in first-out) ordering, i.e. items are removed in the same order that they are added. It supports the following operations:

- **push(item)**: Add an item to the end of the queue.
- **popLeft()**: Remove and return the first item in the queue.
- **peek()**: Return the top of the queue.
- **isEmpty()**: Return true if the queue is empty.

One place where queues are often used is in breadth-first search or in implementing a cache. In breadth-first search we may use a queue to store a list of the nodes that we need to process. Each time we process a node, we add its adjacent nodes to the back of the queue. This allows us to process nodes in the order in which they are viewed.

A queue can be implemented with a linked list and moreover, they are essentially the same thing as long as items are added and removed from opposite sides.

The deque module (short for double-ended queue), provides a data structure which pops from or pushes to either side of the queue with the same $\mathcal{O}(1)$ performance.

Python Implementation

```
from collections import deque

q = deque()
for item in data:
    q.append(item)

while len(q):
    next_item = q.popleft() # array.pop(0)
    print(next_item)
```

Time Complexity of collections.deque operations

Operation	Average Case	Amortized Worst Case
Copy	$\mathcal{O}(n)$	$\mathcal{O}(n)$
append	$\mathcal{O}(1)$	$O(1)$
appendleft	$\mathcal{O}(1)$	$\mathcal{O}(1)$
pop	$\mathcal{O}(1)$	$\mathcal{O}(1)$
popleft	$\mathcal{O}(1)$	$\mathcal{O}(1)$
extend	$\mathcal{O}(k)$	$\mathcal{O}(k)$
extendleft	$\mathcal{O}(k)$	$\mathcal{O}(k)$
rotate	$\mathcal{O}(k)$	$\mathcal{O}(k)$
remove	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2.2.3 Priority Queues

A priority queue is an ADT container that retrieves items not by the insertion time (as in a stack or queue), nor by a key match (as in a dictionary), but instead retrieves items with the highest priority value. Priority queues provide more flexibility than simple sorting because they allow new elements to enter a system at arbitrary intervals. It is much more cost-effective to insert a new job into a priority queue than to re-sort everything on each arrival. Max-priority queues order values in descending order with higher priority first and are typically implemented

with a max-heap. Min-priority queues sort values in ascending order with lower priority values first and are typically implemented with a min-heap.

The basic priority queue supports three primary operations:

- **insert(Q, x)**: Given an item x with key k, insert it into the priority queue Q.
- **findMinimum(Q)** or **findMaximum(Q)**: Return a pointer to the item whose key value is smaller (larger) than any other key in the priority queue Q.
- **deleteMinimum(Q)** or **deleteMaximum(Q)**: Remove the item from the priority queue Q whose key is minimum (maximum).

There are several choices in which underlying data structures can be used for a basic priority queue implementation:

1. Binary heaps are the right answer when the upper bound on the number of items in your priority queue is known, since you must specify array size at creation time. Though this constraint can be mitigated by using dynamic arrays
2. Binary search trees make effective priority queues, since the smallest element is always the leftmost leaf, while the largest element is always the rightmost leaf. The min (max) is found by simply tracing down left (right) pointers until the next pointer is nil. Binary tree heaps prove most appropriate when you need other dictionary operations, or if you have an unbounded key range and do not know the maximum priority queue size in advance.
3. Sorted arrays are very efficient in both identifying the smallest element and deleting it by decrementing the top index. However, maintaining the total order makes inserting new elements slow. Sorted arrays are only suitable when there will be few insertions into the priority queue.
4. Bounded height priority queue work as their name suggests.
5. Fibonacci and pairing heaps can be used to implement complicated priority queues that are designed to speed up decrease-key operations, where the priority of an item already in the priority queue is reduced. This arises, for example, in shortest path computations when we discover a shorter route to a vertex v than previously established.

Note: See binary heap section for more details about heapq. The heapq module creates min-heaps by default, max priority queues can be created by inverting priority values before insertion. The Queue.PriorityQueue module is a partial wrapper around the heapq module which also implements min-priority queues by default.

Python Implementation

```
# The queue module for a min-priority queue
from queue import PriorityQueue

pq = PriorityQueue()
for item in data:
    pq.put((item.priority, item))

while not pq.empty():
    next_item = pq.get()
    print(next_item)

# The heapq module for a min-priority queue
import heapq
```

```
pq = []
for entry_count, item in enumerate(data):
    heapq.heappush(pq, (item.priority, entry_count, item))

while pq:
    next_item = heapq.heappop(pq)
    print(next_item)
```

2.2.4 Indexed Priority Queues

An Indexed Priority Queue gives us the ability to change the priority of an element without having to go through all the elements. It can be thought of as a combination of a hash table, used for quick lookups of values, and a priority queue, to maintain a heap ordering.

2.2.5 Monotonic Stacks and Queues

This structure maintains an ordering so that its elements are either strictly increasing or strictly decreasing. It differs from a heap in that instead of re-ordering elements as they're processed, it will discard previous numbers that do not follow the monotonic condition before appending a new number. It is often used to reduce a quadratic time algorithm to a linear one by maintaining a kind of short-term memory of the most recently processed larger or smaller value.

For example, a monotonic increasing stack can be used to find the index distance of the next smaller number given an unordered list of numbers. Instead of searching for the next smallest number for every number with a nested for loop, resulting in an $\mathcal{O}(n^2)$ runtime, we only need to check the top of the stack of indicies of previously visited numbers, giving us a $\mathcal{O}(n)$ runtime. In an monotonic increasing stack, we append values and pop from the right.

A geometric variant of the question asks about the area that exists in the spaces bounded above from a given elevation map or histogram. We can use increasing stacks but need to keep track of lower bounds and may need to record a cumulative or max value across the entire sequence.

In some application we might also need to remove elements from the front and back, thus a double-ended queue (deque from collections module) should be used. In an increasing queue, we find the first element smaller than the current, either on the left (from pushing in) or on the right (from popping out). In a decreasing queue we find the first element larger than current, either in the left (from pushing in) or in the right (from popping out). A monotonic queue can also be useful for implementing a variant of the sliding window. See sliding window section for an example.

Python Implementation

```
def monotonic_stack(A):
    smaller_to_right = [-1] * len(A)
    stack = []
    for i, v in enumerate(A):
        while stack and A[stack[-1]] > v: # use < for larger_to_right
            cur = stack.pop()
            smaller_to_right[cur] = i - cur
        stack.append(i)
    return smaller_to_right

def trap_water(height):
```

```
    stack, water = [], 0
    for i, v in enumerate(height):
        while stack and v >= stack[-1][0]:
            right_border, _ = stack.pop()
            # we need a left and right border to contain water
            if stack:
                left_border, j = stack[-1]
                # compute the cumulative water
                water += min(left_border-right_border, v-right_border)*(i-j-1)
        stack.append((e, i))
    return water

import collections
def monotonic_deque(A):
    dq = collections.deque()
    smaller_to_left, smaller_to_right = [-1] * len(A), [-1] * len(A)
    for i, v in enumerate(A):
        while dq and A[dq[-1]] >= v:
            smaller_to_right[dq.pop()] = v
        if dq:
            smaller_to_left[i] = A[dq[-1]]
        dq.append(i)
    return smaller_to_left, smaller_to_right
```

2.3 Hash Tables

A hash table is a data structure that maps keys to values for highly efficient lookup. There are a number of ways of implementing this, but a simple and common implementation is known as **separate chaining**. In this implementation, we use an array of linked lists and a hash code function. To insert a key (which might be a string or essentially any other data type) and value, we do the following:

1. First, compute the key's hash code, which will usually be an *int* or *long*. Note that two different keys could have the same hash code, as there may be an infinite number of keys and a finite number of hash codes.
2. Then, map the hash code to an index in the array. This could be done with something like $hash(key) \mod array\_length$.
3. At this index, there is a linked list of keys and values. Store the key and value in this index. We must use a linked list because of collisions: you could have two different keys with the same hash code, or two different hash codes that map to the same index.

To retrieve the value pair by its key, you repeat this process. Compute the hash code from the key, and then compute the index from the hash code. Then, search through the linked list for the value with this key. If the number of collisions is very high, the worst case runtime is $\mathcal{O}(n)$, where n is the number of keys. However, we generally assume a good implementation that keeps collisions to a minimum, in which case the lookup time is $\mathcal{O}(1)$. Alternatively, we can implement the hash table with a balanced binary search tree. This gives us an $\mathcal{O}(\log n)$ lookup time. The advantage of this is potentially using less space, since we no longer allocate a large array. We can also iterate through the keys in order, which can be useful sometimes.

The other strategy used to resolve collisions is to not only require each array element to contain only one key, but to also allow keys to be mapped to alternate indices when their original spot is already occupied. This is known as **open addressing**. In this type of hashing, we have a

parameterized hash function h that takes two arguments, a key and a positive integer. Searching or **probing** for an item requires examining not just one spot, but many spots until either we find the key, or reach a None value. After we delete an item, we replace it with a special value Deleted, rather than simply None. This way, the Search algorithm will not halt when it reaches an index that belonged to a deleted key.

The simplest implementation of open addressing is **linear probing**: start at a given hash value and then keep adding some fixed offset to the index until an empty spot is found. The main problem with linear probing is that the hash values in the middle of a cluster will follow the exact same search pattern as a hash value at the beginning of the cluster. As such, more and more keys are absorbed into this long search pattern as clusters grow. We can solve this problem using **quadratic probing**, which causes the offset between consecutive indices in the probe sequence to increase as the probe sequence is visited. **Double hashing** resolves the problem of the clustering that occurs when many items have the same initial hash value and they still follow the exact same probe sequence. It does this by using a hash function for both the initial value and its offset.

A useful hash function for strings is,

$$H(S, j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \cdot char(s_{i+j}) \mod m$$

where α is the size of the alphabet and $char(x)$ is the ASCII character code. This hash function has the useful property allowing hashes of successive m-character windows of a string to be computed in constant time instead of $\mathcal{O}(m)$.

$$H(S, j+1) = (H(S, j) - \alpha^{m-1} char(s_j))\alpha + s_{j+m}$$

2.3.1 Dictionaries and Hash Tables

The dictionary data type (a.k.a. hash table or hash map) permits access to data items based on its content. You may insert an item into a dictionary to retrieve it in constant time later on. To resolve hash collisions, Python dicts use open addressing with random probing where the next slot is picked in a pseudo random order. The entry is then added to the first empty slot.

The primary operations a hash table supports are:

- **search(D, k)** – Given a search key k, return a pointer to the element in dictionary D whose key value is k, if one exists.
- **insert(D, x)** – Given a data item x, add it to the set in the dictionary D.
- **delete(D, x)** – Given a pointer to a given data item x in the dictionary D, remove it from D.
- **max(D)** or **min(D)** – Retrieve the item with the largest (or smallest) key from D. This enables the dictionary to serve as a priority queue.
- **predecessor(D, k)** or **successor(D, k)** – Retrieve the item from D whose key is immediately before (or after) k in sorted order. These enable us to iterate through the elements of the data structure.

Time Complexity of Dictionary Operations

Operation	Average Case	Amortized Worst Case
k in d	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Copy	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Get Item	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Set Item	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Delete Item	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Iteration	$\mathcal{O}(n)$	$\mathcal{O}(n)$

2.3.2 Sets

In mathematical terms, a set is an unordered collection of unique objects drawn from a fixed universal set. A hash set implements the set ADT using a hash table. The core operations that sets support are:

1. Test whether $u_i \in S_j$.
2. Compute the union or intersection of S_i and S_j.
3. Insert or delete members of S.

Sets are commonly implemented with the following data structures:

1. Containers or dictionaries – A subset can also be represented using a linked list, array, or dictionary containing exactly the elements in the subset.
2. Bit vectors – An n-bit vector or array can represent any subset S on a universal set U containing n items. Bit i will be 1 if $i \in S$, and 0 if not.
3. Bloom filters – We can emulate a bit vector in the absence of a fixed universal set by hashing each subset element to an integer from 0 to n and setting the corresponding bit.

Sorted order turns the problem of finding the union or intersection of two subsets into a linear-time operation, just sweep from left to right and see what you are missing. It makes element searching possible in sublinear time. Though, printing the elements of a set in a canonical order reminds us that order doesn't matter.

If each subset contains exactly two elements, they can be thought of as edges in a graph whose vertices represent the universal set. A system of subsets with no restrictions on the cardinality of its members is called a hypergraph.

Time Complexity of Set Operations

Operation	Average Case	Amortized Worst Case
x in s	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Union	$\mathcal{O}(len(s) + len(t))$	
Intersection	$\mathcal{O}(min(len(s), len(t))$	$\mathcal{O}(len(s) * len(t))$
Multiple intersection	$\mathcal{O}(n-1) * \mathcal{O}(max(len(s_i)))$	
Difference	$\mathcal{O}(len(s))$	
Difference Update	$\mathcal{O}(len(t))$	
Symmetric Difference	$\mathcal{O}(len(s))$	$\mathcal{O}(len(s) * len(t))$
Symmetric Difference Update	$\mathcal{O}(len(t))$	$\mathcal{O}(len(t) * len(s))$

2.4 Trees

A tree is an ADT composed of nodes such that there is a root node with zero or more child nodes where each child node can be recursively defined as a root node of a sub-tree with zero or more children. Since there are no edges between sibling nodes, a tree cannot contain cycles. Furthermore, nodes can be given a particular order, can have any data type as values, and they may or may not have links back to their parent nodes.

2.4.1 Binary Trees

A binary tree is a tree in which each node has up to two children.

A **binary search tree** is a binary tree in which every node n follows a specific ordering property: all left descendants $\leq n <$ all right descendants. An inorder traversal (Left-Node-Right) of a binary search tree will always result in a monotonically increasingly ordered sequence.

A **complete** binary tree is a binary tree in which every level of the tree is filled, except for perhaps the last level and all of the nodes in the bottom level are as far to the left as possible. A complete tree with n nodes has $\lceil \log n \rceil$ height. There is no ambiguity about where the "empty" spots in a complete tree are so we do not need to use up space to store references between nodes, as we do in a standard binary tree implementation. This means that we can store its nodes inside an zero-indexed array. For a node corresponding to index i, its left child is stored at index $2i + 1$, and its right child is stored at index $2i + 2$. Going backwards, we can also deduce that the parent of the node at index i (when $i > 0$) is stored at index $\lfloor (i-1)/2 \rfloor$.

A **full** binary tree is a binary tree in which every node has either zero or two children. A **perfect** binary tree is one that is both full and complete. A full binary tree with n leaves has $n - 1$ internal nodes. Then the number of unique rooted full binary tree with $n + 1$ leaf nodes, equivalently n internal node, can be counted using the nth Catalan number, C_n. View section on Catalan numbers for more details.

Python Implementation

```
# n-ary tree using default dictionary
from collections import defaultdict
tree = lambda: defaultdict(tree)

# Object-oriented binary tree
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
```

```
        self.left = left
        self.right = right
```

2.4.2 Binary Heaps

The **heap property** states that the key stored in each node is either greater than or equal to or less than or equal to the keys in the node's children, according to some total order. A **min-heap** is a complete binary tree (filled other than the rightmost elements on the last level) where each node is smaller than its children. The root, therefore, is the minimum element in the tree. The converse ordering holds for a **max-heap**. We have two main operations on a heap:

- **insert(x)**: When we insert into a min-heap, we always start by inserting the element at the bottom. We insert at the rightmost spot so as to maintain the complete tree property. Then, we maintain the heap property by swapping the new element with its parent until we find an appropriate spot for the element. We essentially bubble up the minimum element. This takes $\mathcal{O}(\log n)$ time, where n is the number of nodes in the heap.
- **findMin()** or **findMax()**: Finding the minimum element of a min-heap is inexpensive since it will always be at the top. The challenging part is how to remove it while maintaining the heap property. First, we remove the minimum element and swap it with the last element in the heap (the bottommost, rightmost element). Then, we bubble down this element, swapping it with one of its children until the heap property is restored. This algorithm will also take $\mathcal{O}(\log n)$ time.

A heap will be better at findMin/findMax ($\mathcal{O}(1)$), while a BST is performant at all finds ($\mathcal{O}(\log n)$). A heap is especially good at basic ordering and keeping track of max and mins.

Note, heapq creates a min-heap by default. To create a max-heap, you will need to invert values before storing and after retrieving them. Alternatively, you can define a class to wrap the module and override and invert the comparison method. The heapreplace(a, x) method returns the smallest value originally in the heap regardless of the value of x. Alternatively, heappushpop(a, x) pushes x onto the heap before popping the smallest value.

Python Implementation

```
# Using heapq module
import heapq

items = [1, 6, 4, 2, 5, 7, 3, 9, 8, 10]

## Min-heap
min_heap = []
for item in items:
    heapq.heappush(min_heap, item)

in_place = items.copy()
heapq.heapify(in_place)
heapq.nsmallest(3, in_place)
while in_place:
    min_item = heapq.heappop(in_place)
    print(min_item)

## Max-heap
```

```python
max_heap = []
for item in items:
    heapq.heappush(max_heap, -item)

while max_heap:
    max_item = -heapq.heappop(max_heap)
    print(max_item)

in_place = items.copy()
heapq._heapify_max(in_place)
heapq.nlargest(3, in_place)
while in_place:
    max_item = heapq._heappop_max(in_place)
    print(max_item)

# Max heap full implementation
class MaxHeap:
    def __init__(self, items=[]):
        self.heap = [0]
        for i in items:
            self.heap.append(i)
            self._floatUp(len(self.heap) - 1)

    def push(self, data):
        self.heap.append(data)
        self._floatUp(len(self.heap) - 1)

    def peek(self):
        if self.heap[1]:
            return self.heap[1]
        else:
            return False

    def pop(self):
        if len(self.heap) > 2:
            self._swap(1, len(self.heap) - 1)
            maxVal = self.heap.pop()
            self._bubbleDown(1)
        elif len(self.heap) == 2:
            maxVal = self.heap.pop()
        else:
            maxVal = False
        return maxVal

    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def _floatUp(self, index):
        parent = index // 2
        if index <= 1:
            return
        elif self.heap[index] > self.heap[parent]:
            self._swap(index, parent)
            self._floatUp(parent)

    def _bubbleDown(self, index):
        left = index * 2
        right = index * 2 + 1
        largest = index
        if len(self.heap) > left and self.heap[largest] < self.heap[left]:
            largest = left
        if len(self.heap) > right and self.heap[largest] < self.heap[right]:
```

```
            largest = right
        if largest != index:
            self._swap(index, largest)
            self._bubbleDown(largest)
```

Time Complexity of heapq operations

Operation	Average Case	Amortized Worst Case
heapify	$\mathcal{O}(n)$	$\mathcal{O}(n)$
heappush	$\mathcal{O}(1)$	$O(\log n)$
heappop	$\mathcal{O}(\log n)$	$O(\log n)$
peek	$\mathcal{O}(1)$	$\mathcal{O}(1)$
heappushpop	TBD	TBD
heapreplace	TBD	TBD
nlargest	TBD	TBD
nsmallest	TBD	TBD

2.4.3 Tries (Prefix Trees)

A trie is a variant of an n-ary tree in which alphanumeric characters are stored at each node. Each path down the tree may represent a word. The * nodes (sometimes called "null nodes") are often used to indicate complete words. The actual implementation of these * nodes might be a special type of child (i.e. a TerminatingTrieNode class which inherits from TrieNode) or we can use a boolean flag. A node in a trie could have anywhere from 1 through size of alphabet + 1 children (or, 0 through size of alphabet if a boolean flag is used instead of a * node).

The complexity of creating a trie is $\mathcal{O}(|W|L)$, where W is the number of words, and L is an average length of the word: you need to perform L lookups on average for each of the W words in the set. Same goes for looking up words later: you perform L steps for each of the W words. Very commonly, a trie is used to store the entire English language for quick prefix lookups. Many problems involving lists of valid words leverage a trie as an optimization. Note that it can often be useful to preprocess the words before inserting them into a trie, possibly by parsing or reversing.

To get autocomplete suggestions, it helps to store the cumulative word along with the end indicator. We can then search the trie up to the prefix end and perform a recursive DFS to find all word end points while appending the results to an array of suggestions. To handle misspellings, a function considers four types of edits: a deletion (remove one letter), a transposition (swap two adjacent letters), a replacement (change one letter to another) or an insertion (add a letter). The edit function returns a list of words within a max N edits from the specified word that exist in a dictionary of known words. We can then find suggestions with DFS as before. For better semantics, we can use bayesian probabilities to determine the intent of the user[1].

Note, most methods in the trie follow a similar traversal pattern, create a pointer to the root of the tree, iterate over characters of a given word, check if the character is not in the current pointers children and then move the pointer forward to the corresponding child.

Python Implementation

[1] http://norvig.com/spell-correct.html

```python
# Trie node as lambda function
import collections
TrieNode = lambda: collections.defaultdict(TrieNode)

# Trie node as class with end of word attribute
class TrieNode:
    def __init__(self):
        self.word = False
        self.children = {}

# Trie dict with end of word indicated by special character $
class Trie:
    def __init__(self):
        self.root = {}

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node:
                node[char] = {}
            node = node[char]
        node['$'] = word

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node:
                return False
            node = node[char]
        return '$' in node

    def startsWith(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node:
                return False
            node = node[char]
        return True

    def suggestions(self, prefix):
        def dfs(node):
            results = []
            if not node: return results
            if '$' in node:
                results.append(node['$'])
            for child in node:
                results.extend(dfs(node[child]))
            return results

        node = self.root
        for char in prefix:
            if char not in node:
                return []
            node = node[char]
        return dfs(node)
```

2.4.4 Suffix Trees/Arrays

A special kind of trie, called a suffix tree, can be used to index all suffixes in a text in order to carry out fast full text searches. The construction of such a tree for the string S takes linear time and space relative to the length of S. A suffix tree is basically like a search trie: there is a root node, edges going out of it leading to new nodes, and further edges going out of those, and so forth. Unlike in a search trie, the edge labels are not single characters. Instead, each edge is labeled using a pair of integers: [from, to], which are pointers into the text. In this sense, each edge carries a string label of arbitrary length, but takes only $\mathcal{O}(1)$ space (two pointers).

Some example use cases are as follows:

- Find all occurrences of q as a substring of S: In collapsed suffix trees, it takes $\mathcal{O}(|q| + k)$ time to find the k occurrences of q in S.
- Locating a substring if a certain number of mistakes or edits are allowed
- Locating matches for a regular expression pattern
- Finding Longest common substring to a set of strings in linear-time
- Find the longest palindrome in S

Storing a string's suffix tree typically requires significantly more space than storing the string itself. Observe that most of the nodes in a trie-based suffix tree occur on simple paths between branch nodes in the tree. Each of these simple paths corresponds to a substring of the original string. By storing the original string in an array and collapsing each such path into a single edge, we have all the information of the full suffix tree in only $\mathcal{O}(n)$ space. The label for each edge is described by the starting and ending array indices representing the substring.

The suffix tree for the string S of length n is defined as a tree such that:

1. The tree has exactly n leaves numbered from 1 to n.
2. Except for the root, every internal node has at least two children.
3. Each edge is labelled with a non-empty substring of S.
4. No two edges starting out of a node can have string-labels beginning with the same character.
5. The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix $S[i \cdots n]$, for i from 1 to n.

Suffix arrays do most of what suffix trees do, while using roughly four times less memory. They are also easier to implement. A suffix array is, in principle, just an array that contains all the n suffixes of S in sorted order. Thus a binary search of this array for string q suffices to locate the prefix of a suffix that matches q, permitting an efficient substring search in $O(\log n)$ string comparisons. With the addition of an index specifying the common prefix length of all bounding suffixes, only $\log n + |q|$ character comparisons need be performed on any query, since we can identify the next character that must be tested in the binary search.

In a suffix array, a suffix is represented completely by its unique starting position (from 1 to n) and read off as needed using a single reference copy of the input string. Some care must be taken to construct suffix arrays efficiently, however, since there are $O(n^2)$ characters in the strings being sorted. One solution is to first build a suffix tree, then perform an in-order traversal of

it to read the strings off in sorted order. However, more recent breakthroughs have lead to space/time efficient algorithms for constructing suffix arrays directly.

Python Implementation

```
from itertools import zip_longest, islice

def to_int_keys(l):
    seen = set()
    ls = []
    for e in l:
        if not e in seen:
            ls.append(e)
            seen.add(e)
    ls.sort()
    index = {v: i for i, v in enumerate(ls)}
    return [index[v] for v in l]

def suffix_array(s):
    """
    suffix array of s, TC: O(n * log(n)^2)
    """
    n = len(s)
    k = 1
    line = to_int_keys(s)
    while max(line) < n - 1:
        line = to_int_keys(
            [a * (n + 1) + b + 1
             for (a, b) in
             zip_longest(line, islice(line, k, None),
                         fillvalue=-1)])
        k <<= 1
    return line
```

2.4.5 Merkle Trees

A **hash tree** or Merkle tree is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures. Hash trees are a generalization of hash lists and hash chains.

Python Implementation

```
from hashlib import sha256

def hash(x):
    S = sha256()
    S.update(x)
    return S.hexdigest()

def merkle(node):
    if not node:
        return '#'
    m_left = merkle(node.left)
    m_right = merkle(node.right)
    node.merkle = hash(m_left + str(node.val) + m_right)
    return node.merkle

```

```python
# Two trees are identical if the hash of their roots are equal (except for collisions)
def isSubtree(s, t):
    merkle(s)
    merkle(t)

    def dfs(node):
        if not node:
            return False
        return (node.merkle == t.merkle or
                dfs(node.left) or dfs(node.right))
    return dfs(s)
```

2.4.6 Kd-Trees

Kd-trees and related spatial data structures hierarchically partition k-dimensional space into a small number of cells, each containing a few representatives from an input set of points. This provides a fast way to access any object by position. We traverse down the hierarchy until we find the smallest cell containing it, and then scan through the objects in this cell to identify the right one. Building the tree can be done in $O(N \log N)$, where the bottleneck is a requirement of presorting the points and finding the medians (but we only need to do this once). Search, Insert, Delete all have runtime of $\mathcal{O}(\log N)$, similar to how a normal binary tree works (with a tree balancing mechanism).

Typical algorithms construct kd-trees by partitioning point sets. Ideally, this plane equally partitions the subset of points into left/right (or up/down) subsets. Partitioning stops after $\log n$ levels, with each point in its own leaf cell. Each box-shaped region is defined by $2k$ planes, where k is the number of dimensions. Useful applications are as follows:

- Point location – To identify which cell a query point q lies in, we start at the root and test which side of the partition plane contains q.
- Nearest neighbor search – To find the point in S closest to a query point q, we perform point location to find the cell c containing q
- Range search – Which points lie within a query box or region? Starting from the root, check whether the query region intersects (or contains) the cell defining the current node. If it does, check the children; if not, none of the leaf cells below this node can possibly be of interest.
- Partial key search – Suppose we want to find a point p in S, but we do not have full information about p. Say we are looking for someone of age 35 and height 5'8" but of unknown weight in a 3D-tree with dimensions of age, weight, and height. Starting from the root, we can identify the correct descendant for all but the weight dimension

Kd-trees are most useful for a small to moderate number of dimensions, say from 2 up to maybe 20 dimensions, otherwise they suffer from what's known as the curse of dimensionality. Algorithms that quickly produce a point provably close to the query point are a recent development in higher-dimensional nearest neighbor search. A sparse weighted graph structure is built from the data set, and the nearest neighbor is found by starting at a random point and walking greedily in the graph towards the query point.

Python Implementation

```python
# Using heap and distance without square root
def KNN_to_origin(points, K):
    def distance_to_origin(p):
        return p[0]**2 + p[1]**2
    q = []
    for point in points:
        heapq.heappush(q, (distance_to_origin(point), point))
    return [
        heapq.heappop(q)[1]
        for _ in range(K)
    ]

# Using a KDTree
from scipy import spatial
def KNN_to_origin(points, K):
    tree = spatial.KDTree(points)
  # x is the origin, k is the number of closest neighbors, p=2 refers to
    choosing L2 norm (euclidean distance)
    distance, idx = tree.query(x=[0,0], k=K, p=2)
    return [points[i] for i in idx] if K > 1 else [points[idx]]
```

2.4.7 Segment Trees

A Segment Tree can be used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point. In principle, it's a static structure, meaning it cannot be modified once it's built. For n intervals, a segment tree uses $\mathcal{O}(n \log n)$ storage, can be built in $\mathcal{O}(n \log n)$ time, and support searching for all the intervals that contain a query point in $\mathcal{O}(\log n + k)$, k being the number of retrieved intervals or segments.

1. Segment tree T is a binary tree.
2. Leaves in T correspond to the intervals described by the endpoints in set of intervals I. The ordering of intervals is maintained in the tree, i.e. the leftmost leaf corresponds to the leftmost interval.
3. The internal nodes of T correspond to intervals that are the union of elementary intervals.
4. Each node or leaf in T stores the interval of itself and a set of intervals in a data structure.

Similar trees that operate on intervals are described below:

- **Segment trees** stores intervals and are optimized for "which of these intervals contains a given point" queries.

  $\mathcal{O}(n \log n)$ preprocessing time, $\mathcal{O}(k + \log n)$ query time, $\mathcal{O}(n \log n)$ space. Interval can be added/deleted in $\mathcal{O}(\log n)$ time.

- **Interval trees** store intervals as well, but are optimized for "which of these intervals overlap with a given interval" queries. They can also be used for point queries - similar to segment tree.

  $\mathcal{O}(n \log n)$ preprocessing time, $\mathcal{O}(k + \log n)$ query time, $\mathcal{O}(n)$ space. Interval can be added/deleted in $\mathcal{O}(\log n)$ time

- **Range trees** store points and are optimized for "which points fall within a given interval" queries.

$\mathcal{O}(n \log n)$ preprocessing time, $\mathcal{O}(k + \log n)$ query time, $\mathcal{O}(n)$ space. New points can be added/deleted in $\mathcal{O}(\log n)$ time

- **Binary indexed trees** stores items-count per index, and are optimized for "how many items are there between index m and n" queries.

  $\mathcal{O}(n \log n)$ preprocessing time, $\mathcal{O}(\log n)$ query time, $\mathcal{O}(n)$ space. The items-count per index can be increased in $\mathcal{O}(\log n)$ time

Python Implementation

```
#Segment tree node
class Node(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.total = 0
        self.left = None
        self.right = None

class SegmentTree(object):
    def __init__(self, nums):
        self.root = createTree(nums, 0, len(nums)-1)
    def createTree(nums, l, r):
        if l > r:
            return None
        #leaf node
        if l == r:
            n = Node(l, r)
            n.total = nums[l]
            return n
        mid = (l + r) // 2
        root = Node(l, r)
        #recursively build the Segment tree
        root.left = createTree(nums, l, mid)
        root.right = createTree(nums, mid+1, r)
        #Total stores the sum of all leaves under root
        #i.e. those elements lying between (start, end)
        root.total = root.left.total + root.right.total
        return root
```

2.5 Self-balancing Trees

Balanced search trees use local **rotation operations** to restructure search trees, moving more distant nodes closer to the root while maintaining the in-order search structure of the tree. The **balance factor** of a node in a binary tree is the height of its right subtree minus the height of its left subtree.

Among balanced search trees, AVL and 2/3 trees are now considered out-dated while red-black trees seem to be more popular. A particularly interesting self-organizing data structure is the splay tree, which uses rotations to move any accessed key to the root. Frequently used or recently accessed nodes thus sit near the top of the tree, allowing faster searches.

2.5.1 AVL Trees

An AVL tree is a self-balancing binary search tree. A node satisfies the **AVL invariant** if its balance factor is between -1 and 1. A binary tree is AVL-balanced if all of its nodes satisfy the AVL invariant, so we can say that an AVL tree is a binary search tree that is AVL-balanced.

To maintain the AVL condition, perform an insertion/deletion using the typical BST algorithm, then if any nodes have the balance factor invariant violated, restore the invariant. We can simply do so after the recursive Insert, Delete, ExtractMax, or ExtractMin call. So we go down the tree to search for the correct spot to insert the node, and then go back up the tree to restore the AVL invariant. In fact, these restrictions make it straightforward to define a small set of simple, constant-time procedures to restructure the tree to restore the balance factor in these cases. Recall, these procedures are called rotations.

The worst-case running time of AVL tree insertion and deletion is $\mathcal{O}(h)$, where h is the height of the tree, the same as for the naive insertion and deletion algorithms. An AVL tree with n nodes has height at most $1.44 \log n$. AVL tree insertion, deletion, and search have worst-case running time $\Theta(\log n)$, where n is the number of nodes in the tree

2.5.2 Red–black Trees

A red–black tree is a kind of self-balancing binary search tree. Each node of the binary tree has an extra bit which is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently. The balancing of the tree is not perfect, but it is good enough to allow it to guarantee searching in $O(\log n)$ time.

Properties:

1. Each node is either red or black.
2. The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes.

2.5.3 2-3 Trees and B-Trees

A B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time ($\mathcal{O}(\log n)$). The B-tree generalizes the binary search tree, allowing for nodes with more than two children and multiple keys. It is commonly used in databases and file systems.

The idea behind a B-tree is to collapse several levels of a binary search tree into a single large node containing multiple keys, so that we can make the equivalent of several search steps before another disk access is needed. This is because we utilize whole blocks of disk memory per level of B-tree when the CPU performs low-bandwidth reads from disk memory. This is in contrast to the CPU performing high-bandwidth reads of words from cache memory.

The **branching factor B** indicates the number of keys and children a node may have.

$$B \leq \text{number of children} < 2B$$
$$B - 1 \leq \text{number of keys} < 2B - 1$$

A **2-3 tree** is a B-tree with branching factor of 2. This means it has at most 2 keys and at most 3 children.

B-trees are constructed in a bottom-up way: values are inserted into a node based on binary search. If the node reaches its capacity based on the degree of the B-tree, then it is split in half with left or right bias and an appropriate root (median value) and children are selected and appointed to existing or new nodes.

For implementing multi-level indexing in a database, every node will have a key to be indexed by a pointer to its child nodes in their memory blocks as well as a pointer to a record on the database (value). In a **B+ tree**, only leaf nodes contain a record pointer with leaf nodes also containing a copy of corresponding parent keys.

2.6 Graphs

A graph is simply a collection of nodes, some of which may have edges between them. With this definition, we see that a tree is a connected graph that does not have cycles. Graphs can be either **directed** or **undirected**. A graph might consist of multiple isolated subgraphs. If there is a path between every pair of vertices, it is called a **connected** graph. A graph can also have cycles (or not), an **acyclic** graph is one without cycles. Note that a tree is undirected and acyclic, which differs from a directed and acyclic graph in which sibling nodes can be joined by edges in a diamond-like shape. There are two common ways to represent a graph: adjacency lists and adjacency matrices.

In an **adjacency list** representation, every vertex stores a list of adjacent vertices. In an undirected graph, an edge like (a, b) would be stored twice: once in a's adjacent vertices and once in b's adjacent vertices. An adjacency list is faster and uses less space for sparse graphs and conversely, it will be slower and contain redundancies for dense graphs.

An **adjacency matrix** is an NxN boolean matrix (where N is the number of nodes), where a true value at $M_{i,j}$ indicates an edge from node i to node j. (You can also use an integer matrix with 0s and 1s.) In an undirected graph, an adjacency matrix will be symmetric. In a directed graph, it will not (necessarily) be. An adjacency matrix will be faster for dense graphs and simpler for graphs with weighted edges, but it will use more space, always having $\mathcal{O}(V^2)$ space complexity.

Some questions to ask when deciding on a representation include:

1. How big will your graph be? – Adjacency matrices make sense only for small or very dense graphs.

2. How dense will your graph be? — If your graph is very dense, meaning that a large fraction of the vertex pairs define edges, there is probably no compelling reason to use adjacency lists. You will be doomed to using $\Theta(n^2)$ space anyway. Indeed, for complete graphs, matrices will be more concise due to the elimination of pointers.

3. Which algorithms will you be implementing? – Certain algorithms are more natural on adjacency matrices (such as all-pairs shortest path) and others favor adjacency lists (such as most DFS-based algorithms). Adjacency matrices win for algorithms that repeatedly ask, "Is (i,j) in G?" However, most graph algorithms can be designed to eliminate such queries.

4. Will you be modifying the graph over the course of your application? – Efficient static graph implementations can be used when no edge insertion/deletion operations will done following initial construction. Indeed, more common than modifying the topology of the graph is modifying the attributes of a vertex or edge of the graph, such as size, weight, label, or color. Attributes are best handled as extra fields in the vertex or edge records of adjacency lists.

Planar graphs are those that can be drawn in the plane so no two edges cross. Planar graphs are always sparse, since any n-vertex planar graph can have at most $3n - 6$ edges, thus they should be represented using adjacency lists. Euler's Formula states $v - e + f = 2$ for all planar graphs, where numbers $v =$ vertices $e =$ edges, and $f =$ faces.

Python Implementation

```
edges = [['A', 'B'], ['B', 'C'], ['C', 'A']]

# Directed graph using an adjacency list
def construct(edges):
    g = collections.defaultdict(list)
    for from, to in edges:
        g[from].append(to)

# Undirected graph using an adjacency list
def construct(edges):
    g = collections.defaultdict(set)
    for from, to in edges:
        g[from].add(to)
        g[to].add(from)

# Undirected graph using an adjacency matrix
def construct(edges):
    nodes = set()
    for e in edges:
        nodes.update({e[0], e[1]})
    n, ordering = len(nodes) list(nodes)
    g = [[0] * n for _ in range(n)]
    for from, to in edges:
        i, j = ordering.index(from), ordering.index(to)
        g[from][to] = 1
```

2.6.1 Flow Networks

A flow network is a directed graph where each edge has a capacity and each edge receives a flow, usually represented as a fraction $flow_i/capacity_i$. A flow must satisfy the restriction that

the amount of flow into a node equals the amount of flow out of it, unless it is a source s, which has only outgoing flow, or sink t, which has only incoming flow. Often we are in search of the max flow of the network using which can be found using the **Ford–Fulkerson algorithm**. Or we may be in search of a **bottleneck** node,

$$bottleneck = min(capacity_i - flow_i \ \ \forall i \text{ in the network}).$$

2.6.2 Union-Find

A union–find data structure (a.k.a disjoint-set union (DSU)) stores a collection of non-overlapping sets. In a graph, a set can be thought of as a tree, i.e. an acyclic and connected subgraph, making union-find a quick method for determining if a graph contains cycles. Disjoint-set data structures play a key role in **Kruskal's algorithm** for finding the **minimum spanning tree** of a graph, which is a subset of the edges of a connected, weighted, undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Union-find can also be used to keep track of the connected components of an undirected graph. The data structure maintains three operations,

1. **makeset(A)** – Create a new size-1 set containing just element A.
2. **find(A)** – Starts at A and returns A's tree root.
3. **union(A, B)** – Finds the root for A and B using the find operation, then sets B's parent to be A, combining the two trees into one.

Two optimizations bring the amortized time for both the union and find operations from $\mathcal{O}(\log N)$ close to $\mathcal{O}(1)$. The find operation limits the number of repeated traversals by keeping track of all the nodes along the path using **path compression**. This is done by storing a reference to a node's parent in a list on the initial find call for all nodes in the traversed path, where the list has size of the node and since its index maps to specific node, we initialize the parent to be itself. In a union operation, we want to ensure the larger set remains the root to ensure the tree depth is minimised, this is known as **union by rank**. To do this, we need to track the size of each set in a list, again where indices correspond to a node.

To perform a sequence of m addition, union, or find operations on a disjoint-set forest with N nodes requires total time $\mathcal{O}(m \cdot \alpha(N))$, where $\alpha(N)$ is the extremely slow-growing inverse Ackermann function, which is effectively considered equal to $\mathcal{O}(1)$.

Python Implementation

```python
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        # We use this to keep track of the size of each set.
        self.rank = [0] * size

    def find(self, x):
        # recursive path compression
        if x != self.parent[x]:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        # Find the parents for x and y.
        px = self.find(x)
```

```
        py = self.find(y)

        # Check if they are already in the same set.
        if px == py:
            return False

        # We want to ensure the larger set remains the root.
        if self.rank[px] > self.rank[py]:
            self.parent[py] = px
            self.rank[px] += self.rank[py]
        else:
            self.parent[px] = py
            self.rank[py] += max(1, self.rank[px])

        # Return true if merge occurred
        return True

    def same_group(self, x, y):
        return self.find(x) == self.find(y)
```

3 Algorithms and Techniques

3.1 Sequence Search and Sort

3.1.1 Binary Search

Time Complexity: $\mathcal{O}(\log n)$ average and worst case. Space Complexity: $\mathcal{O}(1)$

In binary search, we look for an element x in a sorted array by first comparing x to the midpoint of the array. If x is less than the midpoint, then we search the left half of the array. If x is greater than the midpoint, then we search the right half of the array. We then repeat this process, treating the left and right halves as subarrays. Again, we compare x to the midpoint of this subarray and then search either its left or right side. We complete this process when we either find x or the subarray has size 0.

In general, if we can discover some kind of monotonicity, i.e. if condition(index) is True then condition(index + 1) is also True, then we can consider binary search. In this sense, binary search can be thought of as the canonical example of a divide and conquer algorithm. Another notable example is local peak finding in one and two dimensional inputs, where we want to find a number v such that $u < v < w$ where the numbers u, v, w occur in that order. Instead of a linear scan in $\mathcal{O}(n)$, we start with the middle item of the array and compare it with its adjacent elements to determine increasing direction to recurse in, pruning half of the input with every step and finding a peak in $\mathcal{O}(\log n)$.

In some cases we can derive a lower and upper bound on our answer from our input, possibly by finding the sum and the min/max value. Instead of incrementally attempting values in the range of possible values, we should use binary search with backtracking to find a minimum or maximum valid solution.

Instead of searching for an exact target, we may want to return the index of next smallest or next largest index. To do this, we can remove the return on equivalence and return our left pointer instead of -1 after the while loop terminates. For finding the next largest index, we also use an equivalence in the $\geq$ comparator conditional for moving our left pointer forward.

Python Implementation

```
def binary_search(nums, target):
    if len(nums) == 0:
        return -1
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

A = [-14, -10, 2, 108, 108, 243, 285, 285, 285, 401]

# Bisect module implements most binary search use cases
import bisect
## insert into sorted array while maintaining order in O(n)
```

```
bisect.insort(A, 3)
## return index of first occurence of target element O(logn)
bisect.bisect_left(A, -10)
## return index to the right of last occurence of target element O(logn)
bisect.bisect_right(A, -10)

def search_leftmost(nums, target):
    lo, hi = 0, len(nums) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if target > nums[mid]:
            lo = mid + 1
        else: # Equivalence moves search range to the left
            hi = mid - 1
    return lo

def search_rightmost(nums, target):
    lo, hi = 0, len(nums) - 1
    while lo < hi:
        mid = (lo + hi) // 2
        if target >= nums[mid]: # Equivalence moves search range to the right
            lo = mid + 1
        else:
            hi = mid - 1
    return lo
```

3.1.2 Bubble Sort

Time Complexity: $\mathcal{O}(n^2)$ average and worst case. Space Complexity: $\mathcal{O}(1)$

In bubble sort, we start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted. In doing so, the smaller items slowly "bubble" up to the beginning of the list.

Python Implementation

```
def bubble_sort(A):
    for i in range(len(A) - 1, 0, -1):
        for j in range(i):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
    return A
```

3.1.3 Selection Sort

Time Complexity: $\mathcal{O}(n^2)$ average and worst case. Space Complexity: $\mathcal{O}(1)$

Selection sort is the child's algorithm: simple, but inefficient. Find the smallest element using a linear scan and move it to the front (swapping it with the front element). Then, find the second smallest and move it, again doing a linear scan. Continue doing this until all the elements are in place.

Python Implementation

```python
def selection_sort(A):
    for i in range(len(A)):
        min_index = i
        for j in range(i + 1, len(A)):
            if A[j] < A[min_index]:
                min_index = j
        if i != min_index:
            A[i], A[min_index] = A[min_index], A[i]
    return A
```

3.1.4 Insertion Sort

Time Complexity: $\mathcal{O}(n^2)$ average and worst case. Space Complexity: $\mathcal{O}(1)$

Given an array A of size n, iterate i from 1 to n and insert $A[i]$ into a sorted sub array $A[0, i-1]$ until the entire array is sorted. Sorting occurs through pairwise swaps of elements down to their correct positions.

Insertion sort can be useful when streaming real-time data in large chunks and building real-time visualization for these data sources.

Python Implementation

```python
def insertion_sort(A):
    for i in range(1, len(A)):
        n = A[i]
        pos = i
        while pos > 0 and A[pos-1] > n:
            A[pos] = A[pos-1]
            pos -= 1
        A[pos] = n
    return A
```

3.1.5 Merge Sort

Time Complexity: $\mathcal{O}(n \log n)$ average and worst case. Space Complexity: Varies on implementation.

Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single element arrays. It is the "merge" part that does all the heavy lifting.

The merge method operates by copying all the elements from the target array segment into a helper array, keeping track of where the start of the left and right halves should be. We then iterate through the helper, copying the smaller element from each half into the array. At the end, we copy any remaining elements into the target array.

Merge sort can work well with divide-and-conquer approaches if it comes to large amounts of

data stored on different nodes.

$$\begin{aligned} T(n) &= \underbrace{c_1}_{\text{divide}} + \underbrace{2T(n/2)}_{\text{recursion}} + \underbrace{cn}_{\text{merge}} \\ &= (1 + \log n) \cdot cn \\ &= \mathcal{O}(n \log n) \end{aligned}$$

Python Implementation

```
def _merge_lists(left_sublist, right_sublist):
    result = []
    i, j = 0, 0
    while i < len(left_sublist) and j < len(right_sublist):
        if left_sublist[i] <= right_sublist[j]:
            result.append(left_sublist[i])
            i += 1
        else:
            result.append(right_sublist[j])
            j += 1
    result += left_sublist[i:]
    result += right_sublist[j:]
    return result

def merge_sort(A):
    if len(A) <= 1:
        return A
    else:
        midpoint = len(A)//2
        left_sublist = merge_sort(A[:midpoint])
        right_sublist = merge_sort(A[midpoint:])
        return _merge_lists(left_sublist, right_sublist)
```

3.1.6 QuickSort

Time Complexity: $\mathcal{O}(n \log n)$ average, $\mathcal{O}(n^2)$ worst case. Space Complexity: $\mathcal{O}(\log n)$

In quick sort, we pick an element and partition the array, such that all numbers that are less than the partitioning element come before all elements that are greater than it. The partitioning can be performed efficiently through a series of swaps.

If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually be sorted. However, as the partitioned element is not guaranteed to be the median (or anywhere near the median), our sorting could be very slow. This is the reason for the $\mathcal{O}(n^2)$ worst case runtime.

If we allow our algorithm to make random choices, we can turn any input into a "random" input simply by preprocessing it, and then applying the regular quicksort function. To run **randomized quicksort** on an array A with length n, we can define the random variable T_A to be the running time of the algorithm. Now we are considering the probability distribution which the algorithm uses to make its random choices, and not a probability distribution over inputs, $E[T_A] = \Theta(n \log n)$.

Quicksort's divide-and-conquer formulation makes it amenable to parallelization using task parallelism.

A selection algorithm chooses the kth smallest of a list of numbers; this is an easier problem in general than sorting since we don't need to sort the elements in the sublists. **Quickselect** works similarly to quicksort with the difference being that instead of making recursive calls on both sublists, it only makes a single tail-recursive call on the sublist that contains the desired element. This change lowers the average complexity to linear or $\mathcal{O}(n)$ time.

Python Implementation

```python
import random

def partition(arr, start, end, pivot_mode):
  if pivot_mode == 'first':
    pivot = arr[start]
  else:
    pivot_index = random.randrange(start, end)
    pivot = arr[pivot_index]
    arr[pivot_index], arr[start] = arr[start], arr[pivot_index] # place the
    pivot at the start
  i = start + 1
  for j in range(start + 1, end + 1):
    if arr[j] < pivot:
      arr[i], arr[j] = arr[j], arr[i]
      i += 1
  arr[start], arr[i-1] = arr[i-1], arr[start]
  return i-1

def quicksort(arr, start, end, pivot_mode='random'):
    if start < end:
      split = partition(arr, start, end, pivot_mode)
      quicksort(arr, start, split-1, pivot_mode)
      quicksort(arr, split+1, end, pivot_mode)
    return arr

# finds the kth ordered position in an unsorted array  of distinct elements
def findKthLargest(nums, k):
    def quickselect(nums, start, end, k):
        if start == end:
            return nums[start]
        pivot_index = partition(nums, start, end)
        if pivot_index == k:
            return nums[k]
        elif k < pivot_index:
            return quickselect(nums, start, pivot_index-1, k)
        else:
            return quickselect(nums, pivot_index+1, end, k)

    return quickselect(nums, 0, len(nums)-1, k-1)
```

3.1.7 Heap Sort

Time Complexity: $\mathcal{O}(n \log n)$ average and worst case. Space Complexity: $\mathcal{O}(n)$

Given a heap, we can extract a sorted list of the elements in the heap simply by repeatedly calling Remove and adding the items to a list. In particular, the Heap Sort algorithm does the following:

1. Build a min-heap from an unordered array A in $\mathcal{O}(n)$.

2. Find the min element A[0] in $\mathcal{O}(1)$.
3. Swap elements A[n] with A[0] so that the min element is at the end of the array in $\mathcal{O}(1)$.
4. Extract node n from the array and decrement the heap size in $\mathcal{O}(1)$.
5. The new node may violate the min heap principle but the children won't. This allows us to run heapify in $\mathcal{O}(\log n)$.

Python Implementation

```
import heapq

def heapsort(A):
    heapq.heapify(A)
    return [heapq.heappop(A) for _ in range(len(A))]
```

3.1.8 Counting Sort

Time Complexity: $\mathcal{O}(n+k)$ average and worst case, Space Complexity: $\mathcal{O}(n+k)$ (where k is the range of the non-negative key values.)

Instead of using comparison operations as in previous sorting algorithms, counting sort uses integer sorting and relies on a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (a kind of hashing), then doing some arithmetic to calculate the position of each object in the output sequence.

Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for use in situations where the variation in keys is not significantly greater than the number of items.

Counting sort is a **stable** sorting algorithm, meaning the order in which identical values appear in the sorted result will be the same as in the unsorted array. Because of this property, it is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.

1. In an auxiliary array with the size of range of keys, add counts of the number in its corresponding index. Note, the index of the auxiliary array represents the numeric or ordinal value in the original array. Knowing the count of repeated numeric values will help us find the contiguous indices needed to store each number.
2. Iterate over the auxiliary array, add the number to left of the current number, generating a cumulative sum. This will later give us the starting position of the current number since the counts of previous numbers will need be allocated in the indices to the left in a sorted array.
3. Shift all the elements of the auxiliary array to right with the left-most value being zero. This will give us our finalized zero-indexed mapping of array indices (representing a numeric value) to a value representing the starting index in a sorted array.

If the resulting sorted array does not need to be returned then a variant of counting sort can be used in which frequencies of characters are counted and stored in a bitmap of size equal to the max of the input constraints, then only the bitmap is processed while ignoring any values of 0.

Python Implementation

```python
def count_sort_ascii(A):
    output, count = [0] * 256, [0] * 256,
    for i in A:
        count[ord(i)] += 1
    # Change count[i] so that count[i] now contains actual
    # position of this character in output array
    for i in range(256):
        count[i] += count[i-1]
    # Build the output character array while shifting position
    for i in range(len(A)):
        output[count[ord(A[i])] - 1] = A[i]
        count[ord(A[i])] -= 1
    # Copy the output array to A, so that A now
    # contains sorted characters
    for i in range(len(A)):
        A[i] = output[i]
    return A

def count_sort_digits(A, exponent=1):
    n = len(A)
    output, count = [0] * n, [0] * 10
    # Store count of occurrences
    for i in range(n):
        index = A[i] // exponent
        count[index % 10] += 1
    # Change count[i] so that count[i] now contains actual
    # position of this digit in output array
    for i in range(1, 10):
        count[i] += count[i-1]
    # Build the output digit array while shifting position
    for i in range(n):
        output[count[i % 10] - 1] = A[i]
        count[i % 10] -= 1
    # Copy the output array to A, so that A now
    # contains sorted digits
    for i in range(n):
        A[i] = output[i]
    return A
```

3.1.9 Radix Sort

Time Complexity: $\mathcal{O}(w \cdot n)$ average and worst case, Space Complexity: $\mathcal{O}(w + n)$ (where w is the number of bits required to store each key and n is the length of the array to be sorted)

Radix is a Latin word for "root" which can be considered a synonym for an arithmetical base, where decimal is base 10. For simplicity, say you want to use the decimal radix ($= 10$) for sorting. Radix sort, sometimes called bucket sort, makes use of the stable sorting property by iteratively applying counting sort on a single digit of all the elements, i.e. by sorting the numbers by tens and then putting them together again; then by hundreds and so on, which will eventually produce an array sorted in ascending order.

Radix sort can be applied to data that can be sorted lexicographically, i.e integers, words, playing cards, etc. Unlike radix sort, quicksort is universal, while radix sort is only useful for fixed length integer keys. w can be interpreted as the length of the longest value in an array of length n. If $k = n$, then $\mathcal{O}(k * n) = \mathcal{O}(n^2)$. We see that radix sort will only outperform

quicksort when the longest value can be interpreted in less values (digits) than the size of the given array.

Python Implementation

```
# Radix sort using counting sort as subroutine
def radix_sort(arr):
    # Find the maximum number in O(N) to determine number of digits
    max_num = max(arr)
    # Do counting sort for every digit. exp is 10^i where i is the current digit
     number
    exp = 1
    while max_num/exp > 0:
        count_sort_digits(arr, exp)
        exp *= 10
```

3.1.10 Bucket Sort

Time complexity: $\mathcal{O}(n^2)$ worst case, $\mathcal{O}\left(n + \frac{n^2}{k} + k\right) \approx \mathcal{O}(n)$ average case when $k \approx n$ where k is the number of buckets. Space complexity: $\mathcal{O}(n \cdot k)$

1. Set up an array of initially empty bucket
2. Scatter: Go over the original array, putting each object in its bucket
3. Sort each non-empty bucket
4. Gather: Visit the buckets in order and put all elements back into the original array

Radix and bucket sort are two useful generalizations of counting sort. They have a lot in common with counting sort and with each other but exchange time-space tradeoffs.

- Counting sort – Simple buckets, simple processing, memory overhead
- Radix sort – Simple buckets, sophisticated processing, speed overhead (and still need additional static memory)
- Bucket sort – Sophisticated buckets, simple processing, requires dynamic memory, performant on average

Python Implementation

```
def bucket_sort_frequency(s):
    counter = collections.Counter(s)
    buckets = collections.defaultdict(list)

    for key, value in counter.items():
        buckets[value].append(key)

    ret = []
    for count in range(len(s), 0, -1):
        for char in bucket[count]:
            ret += [char] * count
    return ret
```

3.1.11 Cycle Sort

Time Complexity: $\mathcal{O}(n)$ worst case, Space Complexity: $\mathcal{O}(1)$

Cycle sort is an **in-place**, unstable sorting algorithm. It is often useful if we are allowed to modify the input array and want to bring down the space complexity of our algorithm from $\mathcal{O}(N)$ to $\mathcal{O}(1)$, while keeping the time complexity at $\mathcal{O}(N)$. However, it will only produce a correctly sorted array if its elements are in the range of 0 and the length of the array. It can be useful when determining missing positive values in that range since there will be a mismatch between index and the value being stored.

The high-level idea is to move every number to its corresponding index in the array. The original number in the target index is swapped, then we update our target index and use the same approach until the target index matches its value, is out of range, or we have processed all indices in the array.

Python Implementation

```
def cycle_sort(arr):
    for idx, val in enumerate(arr):
        while arr[val] != val:
            arr[val], arr[idx] = arr[idx], arr[val]
    return arr
```

3.1.12 Timsort

Time Complexity: $\mathcal{O}(n)$ best-case, $\mathcal{O}(n \log n)$ average and worst case, Memory $\mathcal{O}(n)$.

Timsort is a hybrid stable sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. The algorithm finds subsequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently. This is done by merging runs until certain criteria are fulfilled. Timsort has been Python's standard sorting algorithm since version 2.3.

3.2 Array Analysis Methods

3.2.1 Two Pointer Technique

Time complexity: $\mathcal{O}(n^2)$, Space Complexity: $\mathcal{O}(1)$

The two pointer technique uses two references to values in a given array to check if they satisfy a condition, otherwise the pointers usually move towards the middle of the array being iterated over.

Python Implementation

```
def two_pointers(seq):
    left, right = 0, len(seq) - 1
    while left < right:
        if _left_condition(left):
            left += 1
        if _right_condition(right):
```

```
            right -= 1
        _validate(left, right)
```

3.2.2 Fast and Slow Pointers

Time Complexity: $\mathcal{O}(n)$, Space Complexity: $\mathcal{O}(1)$

The fast and slow pointers/runners technique (a.k.a. Floyd's Tortoise and Hare Algorithm) is useful when dealing with cyclic linked lists. By moving at different rates, the algorithm proves that the two pointers are either going to meet eventually or one will reach the end of the list. That is, the fast pointer should catch the slow pointer once both the pointers are in a cyclic loop.

If the list has n nodes, then in $\leq n$ steps, either the fast pointer will find the end of the list, or there is a loop and the slow pointer will be in the loop. Suppose the loop is of length $m \leq n$, then once the slow pointer is in the loop, both the fast and slow pointers will be stuck in the loop forever. Each step, the distance between the fast and the slow pointers will increase by 1. When the distance is divisible by m, then the fast and slow pointers will be on the same node and the algorithm terminates. The distance will reach a number divisible by m in $\leq m$ steps. So, getting the slow pointer to the loop, and then getting the fast and slow pointers to meet takes $\leq n + m \leq 2n$ steps, and that is in $\mathcal{O}(n)$.

Fast and slow pointers can also be used to find the midpoint of an acyclic linked list since the second pointer will reach the end while the first pointer reaches the middle. To determine if a linked list has palindromic symmetry, we can find the midpoint, then reverse the second half until the end. From here, we can traverse from the head to the middle and from the end backwards, comparing each node to check if they are equal.

A variation of this technique can be used to find the intersection of two linked lists. Instead of detecting a cycle, we are given a linked list with two heads the intersect and want to find the point of intersection. We again use two pointers, one at the first head and the other at the second, but move both forward at the same rate until they are equal. The trick being that when either pointer reaches the end, we change the pointers reference to be the head of the opposite starting point and traverse a second time. Eventually the two pointers will either meet at a single node or both have None reference.

Python Implementation

```
def has_cycle(head):
    try:
        slow, fast = head, head.next
        while slow is not fast:
            slow = slow.next
            fast = fast.next.next
        return True
    except AttributeError:
        return False

def middleNode(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

```python

def reverseList(head):
    prev = None
    while head:
        next_node = head.next
        head.next = prev
        prev = head
        head = next_node
    return prev

def intersectionNode(headA, headB):
        if headA is None or headB is None:
            return None
        pa = headA
        pb = headB
        while pa is not pb:
            pa = headB if pa is None else pa.next
            pb = headA if pb is None else pb.next
        return pa
```

3.2.3 Sliding Window Technique

Time Complexity: $\mathcal{O}(n)$, Space Complexity: $\mathcal{O}(k)$, where k is the length of a pattern or restrict sequence

Sliding windows are commonly used to find a match or maximum/minimum in a given subarray or substring. It uses two pointers as the boundary of a sliding window to traverse and can also use a counter dictionary to track the state of the current window. A counter may keep track of the number of elements in the current window, while a dictionary can be used to track the last seen indices of the elements in the current window.

1. Define two pointers, start and end, to represent the sliding window.
2. Move end to search for a valid window.
3. When a valid window is found, move start to find a smaller valid window, continuing until the global smallest valid window is found.

A possible variation is to use two sliding windows. This may be necessary when a question asks to find the number of valid sub-sequence of *exactly* K elements.[2] Then we want to find $exactly(K) = atMost(K) - atMost(K-1)$ and may define two sliding windows, one which counts subsequences with length $\leq k$ and another with length $\leq k-1$.

Another variation involves finding the maximum value in windows of size K for each starting index of a given array. This can be done in $\mathcal{O}(Nk)$ time using the standard sliding window technique, however, we can reduce this to $\mathcal{O}(N)$ by using a double-ended queue (deque) that is kept monotonically decreasing. After popping all smaller elements from the back of the queue, we append the next smallest element. We can then pop from the front of the queue to remove elements that are no longer in our current window. This way the queue remains decreasing and stores the max element of the range at queue

Python Implementation

[2] https://leetcode.com/problems/subarrays-with-k-different-integers/

```python
# Using a counter
from collections import Counter
def sliding_window_with_counter(sequence, max_length):
    ans = start = 0
    count = collections.Counter()
    for end, val in enumerate(sequence):
        count[val] += 1
        # while invalid condition, reduce counter and shorten window
        while len(count) >= max_length:
            count[sequence[start]] -= 1
            if count[sequence[start]] == 0:
                del count[sequence[start]]
            start += 1
        ans = max(ans, end - start + 1)
    return ans

# Using monotonic double-ended queue
def maxSlidingWindow(nums, k):
    out, dq = [], deque()
    for i in range(len(nums)):
        # Pop all elements at back of queue that are less than current element
        while dq and dq[-1][0] < nums[i]:
            dq.pop()
        dq.append((nums[i], i))
        # At least one full window was added to queue
        if i >= k-1:
            # Pop front while element index is out of window
            while dq and dq[0][1] < i - k + 1:
                dq.popleft()
            #Add largest element to answer
            if dq: out.append(dq[0][0])
    return out
```

3.2.4 Single-pass with Lookup Table

Time Complexity: $\mathcal{O}(n)$, Space Complexity: $\mathcal{O}(n)$

A lookup table is an example of making a time-space tradeoff. This technique is useful when searching an array for a pair of values, though the method can be extrapolated to triples and more. Given an array and a target we first initialize a lookup table, usually as a hash table or dictionary. Next, we iterate over the array checking if the complement to the current value needed to satisfy the target exists in our lookup table. If it exists, meaning we've visited the complement before, we retrieve the value from the pointer in the lookup and return it with the current index. Otherwise we add the compliment with the current index as a key value pair in the lookup table.

The key takeaway is that when finding pairs to meet a condition, we typically require nested for loops to run $\mathcal{O}(\frac{n(n-1)}{2}) = \mathcal{O}(n^2)$ comparisons. By instead storing and searching a hashmap only for the desired pairing, we reduce our operations to $\mathcal{O}(n)$. When we need to find triples we can reduce 3 for loops and a running time of $\mathcal{O}(n^3)$ to only two loops and running time of $\mathcal{O}(n^2)$. Similar optimizations hold for larger groups, but it's likely optimal to sort the array in $\mathcal{O}(n \log n)$ and use multiple pointers instead.

Python Implementation

```python
def single_pass_lookup(nums, target):
```

```
    lookup = {}
    for i, v in enumerate(nums):
        if target - v in lookup:
            return i, lookup[target - v]
        lookup[v] = i
    raise ValueError('Target not in list.')
```

3.2.5 Prefix Sums and Kadane's Algorithm

Time Complexity: $\mathcal{O}(n)$, Space Complexity: $\mathcal{O}(1)$

The maximum subarray sum problem is the task of finding a contiguous subarray with the largest sum within a given one-dimensional array $A[1...n]$ of numbers.

This problem can be solved using several different techniques, including brute force, divide and conquer, dynamic programming, and reduction to shortest paths. Kadane's algorithm can be viewed as a trivial example of dynamic programming which will be visited in more detail later, and makes use of in-place **prefix sums**. For each number in a sequence, its corresponding prefix sum, also known as cumulative sum, is the sum of all previous numbers in the sequence plus the number itself.

1. Use the input array of nums to store the candidate subarrays sum (i.e. the greatest contiguous sum so far).
2. Ignore cumulative negatives, as they don't contribute positively to the sum.
3. Return the max value of the mutated nums array, which will be the maximum contiguous subararry sum.

Python Implementation

```
def maxSubArray(self, nums: List[int]) -> int:
    for i in range(1, len(nums)):
        if nums[i-1] > 0:
            nums[i] += nums[i-1]
    return max(nums)
```

3.2.6 Prefix Sums with Binary Search

Time Complexity: $\mathcal{O}(n)$, Space Complexity: $\mathcal{O}(n)$

Similar to Kadane's algorithm, we can create an alternative representation of the given array of numbers which will speed up finding the solution. The trick is to relate the original list of numbers to a corresponding list of prefix sums using their index. This is similar to counting sort, where we used cumulative sums in a bitmap to determine sorted indices.

If all numbers are positive, then we see that the list of prefix sums would be strictly monotonically increasing. If we are given a list of positive numbers and we want to find which offset or range a new number corresponds to, we can convert the numbers (which represent some kind of weight) to list of relative offsets (i.e. prefix sums). Our task is then to fit the target offset into the list so that the ascending order is maintained using binary search ($\mathcal{O}(\log(n))$).

Python Implementation

```python
def prefix_sum(self, weights: List[int], target: int) -> int:
    cumulative, prefix_sums = 0, []
    for w in weights:
        cumulative += w
        prefix_sums.append(cumulative)

    low, high = 0, len(prefix_sums)
    while low <= high:
        mid = low + (high - low) // 2
        if target > prefix_sums[mid]:
            low = mid + 1
        else:
            high = mid - 1
    return low
    # Alternatively using itertools and bisect modules
    prefix_sums = list(itertools.accumulate(arr))
    return bisect.bisect_left(prefix_sums, target)
```

3.3 Intervals

3.3.1 Range Operations on Array

Time Complexity: $\mathcal{O}(Q+n)$, Space Complexity: $\mathcal{O}(n)$, where Q is the number of operations or queries

Given an array A of 0's of size n, perform Q operations or queries by incrementing values in the subarray $A[L:R]$ by 1. A naive solution of simulating all the given operations will result in time complexity of $\mathcal{O}(Q \cdot n)$. However, using a numerical method we are able to reduce the time complexity to $\mathcal{O}(Q+n)$. This technique involves creating a secondary array B and only incrementing the value at the left endpoint, L by 1 and decrementing the value at index $R+1$ by -1. After repeating this process for all queries, to find the true desired value of $A[i]$ we can find the prefix sum of B from $B[0:i]$.

If instead we want to find the maximum in the array after performing Q range operations, we can modify the above technique which will give us an algorithm that runs in $\mathcal{O}(n)$. Again, for each range interval we increment the left pointer by 1 and decrement the right by -1. By the end of all this, we have an array that shows the difference between every successive element. From here, we iterate over the array while maintaining a running sum and keeping track of the maximum of the sums.

In the case where furthest right value is very large and the number of queries is small, the linear and space complexity will suffer. For example, when we want to analyze the number of overlapping intervals, instead of storing and iterating over an auxiliary array with a length corresponding to the latest end time, we can use a Counter that increments start times and decrements end times. Then, by sorting and iterating over the keys, we reduce the space complexity to $\mathcal{O}(Q)$ but increase time complexity to $\mathcal{O}(Q \log Q)$.

Python Implementation

```python
def max_after_operations(n, operations):
    B = [0] * (n + 1)
    for start, end, incr in operations:
        B[start - 1] += incr
        if stop <= len(B):
```

```
            B[stop] -= incr;
    max_value = cur = 0
    for i in B:
        cur = cur + i;
        max_value = max(max_value, cur)
    return max_value

import collections
def minMeetingRooms(intervals):
    mp = collections.Counter()
    for item in intervals:
        mp[item.start] += 1
        mp[item.end] -= 1
    ans = cumulative = 0
    for i in sorted(mp.keys()):
        cumulative += mp[i]
        ans = max(ans, cumulative)
    return ans
```

3.3.2 Merge Intervals

Time Complexity: $\mathcal{O}(n \log n)$, Space Complexity: $\mathcal{O}(n)$

An interval problem has an input of a 2D array in which each nested array represents a start and an end value. The interval can also be represented as an object with start and end attributes.

Given two intervals A and B, there will be six different ways the two intervals can relate to each other:

1. A and B do not overlap, A before B
2. A and B overlap, B ends after A
3. A completely overlaps B
4. A and B overlap, A ends after B
5. A and B do not overlap, B before A

If A.start $\leq$ B.start, only 1, 2 and 3 are possible from the above scenarios. Our goal is to merge the intervals whenever they overlap.

For unweighted job scheduling, see the greedy programming section. For weighted job scheduling, see dynamic programming.

Python Implementation

```
def merge_intervals(self, intervals):
    if len(intervals) < 2: return intervals

    # sort on start values
    intervals.sort(key=lambda x: x[0])
    merged = []
    prev_start, prev_end = intervals[0]

    for i in range(1, len(intervals)):
        cur_start, cur_end = intervals[i]

        # overlapping intervals, new starts before previous ends
```

```
        if cur_start <= prev_end: # equivalence implies ranges are inclusive
            prev_end = max(prev_end, cur_end)
        # non-overlapping interval, add the previous interval and update
        else:
            merged.append([prev_start, prev_end])
            prev_start, prev_end = cur_start, cur_end
    # add the last interval
    merged.append([prev_start, prev_end])
    return merged
```

3.4 String Analysis Methods

A string is a sequence of ASCII characters. Many analysis techniques that apply to arrays can also be used on string inputs when they are interpreted as character arrays. Note, the space required for a string counter is $\mathcal{O}(1)$ not $\mathcal{O}(n)$. This is because the upper bound is the range of characters, which is usually a fixed constant of 26 or 256 for all ASCII characters. Typically, slicing and concatenating strings require $\mathcal{O}(n)$ time. Instead, use start and end indices to demarcate a substring.

3.4.1 KMP Pattern Matching

Time Complexity: $\mathcal{O}(n+k)$, Space Complexity: $\mathcal{O}(k)$, where n is the length of the string and k is the length of the pattern

KMP (Knuth–Morris–Pratt) pattern matching improves the worst case complexity of a naive approach of $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. The basic idea behind KMP's algorithm is that whenever we detect a mismatch after some matches, we already know some of the characters in the text of the next window. Instead of wasting computation on previous matches, we use a pre-computed lookup table to skip to the first instance of a match of the starting character.

Values in the longest proper suffix (LPS) array record the max length of matching substrings that exists as both a prefix and suffix, where the suffix ends at the current index. For example, the LPS array of patter "aabaac" is [0, 1, 0, 1, 2, 0].

Python Implementation

```
class KMP:
    def build_lps(self, pattern):
        m = len(pattern)
        lps = [0] * m
        i, j = 0, 1
        while j < m:
            if pattern[i] == pattern[j]:
                lps[j] = i + 1
                i, j = i + 1, j + 1
            elif i != 0:
                i = lps[i - 1]
            else:
                lps[j] = 0
                j += 1
        return lps

    def search(self, text, pattern):
        if not text or not pattern: return 0
```

```
        lps_array = self.build_lps(pattern)
        n, m = len(text), len(pattern)
        i, j = 0, 0
        while i < n:
            # current characters match, move to the next characters
            if text[i] == pattern[j]:
                i, j = i + 1, j + 1
            # current characters don't match
            else:
                if j > 0:  # try start with previous longest prefix
                    j = lps_array[j - 1]
                # 1st character of pattern doesn't match character in text
                # go to the next character in text
                else:
                    i += 1
            # whole pattern matches text, match is found
            if j == m:
                return i - m
        # no match was found
        return -1
```

3.4.2 Rabin–Karp

Time Complexity: $\mathcal{O}(n+k)$, Space Complexity: $\mathcal{O}(k)$, where n is the length of the string and k is the length of the pattern

The Rabin–Karp algorithm is a string-searching algorithm that uses hashing to find an exact match of a pattern string in a text. It uses a rolling hash to quickly filter out positions of the text that cannot match the pattern, and then checks for a match at the remaining positions. It can be useful when the size of the query string is much smaller than the full text as it reduces a naive quadratic worst case of $\mathcal{O}(n \cdot k)$ to a linear worst case runtime.

Python Implementation

```
class RHash():
    def __init__(self, s):
        self.d = 256
        self.q = 101
        self.h = pow(self.d, len(s)-1, self.q)
        self.val = 0
        for c in s:
            self.append(c)
    def append(self, c):
            self.val = (self.val * self.d + ord(c)) % self.q
    def rmleft(self, c):
            self.val = (self.val - self.h * ord(c) + self.q) % self.q
    def __eq__(self, other):
        return self.val == other.val

def strStr(self, haystack, needle):
    hl, nl = len(haystack), len(needle)
    if not needle:
        return 0
    if not haystack or nl > hl:
        return -1

    nh, hh = RHash(needle), RHash(haystack[:nl])
    if nh == hh and haystack[:nl] == needle:
```

```
        return 0

    for i in range(nl, hl):
        hh.rmleft(haystack[i-nl])
        hh.append(haystack[i])
        if hh == nh and haystack[i-nl+1:i+1] == needle:
            return i - nl + 1
    return -1
```

3.4.3 Non-Sequential Analysis with Stack

Time Complexity: $\mathcal{O}(n)$, Space Complexity: $\mathcal{O}(n)$

Given a sequence, we are tasked with processing it so that subsequences are interpreted recursively in a non-standard left-to-right order. This is done in mathematical expressions because of a non-sequential order of operations. It can be further complicated with parenthesis, where we need to delay processing the main expression until we are done evaluating the sub-expressions within parenthesis. To introduce this delay, we use a stack.

A stack can also be useful for processing structured text that uses rules based on special end or beginning delimiters of substrings. If we only care about whether the structured is valid, i.e. has a correct start and end delimiters, then we can reduce the stack into constant number of counter integers.

Python Implementation

```
def basic_calculate(self, s: str) -> int:
    num, stack, sign = 0, [], "+"
    for i in range(len(s)):
        if s[i].isdigit():
            num = num * 10 + int(s[i])
        if s[i] in "+-*/" or i == len(s) - 1:
            if sign == "+":
                stack.append(num)
            elif sign == "-":
                stack.append(-num)
            elif sign == "*":
                stack.append(stack.pop()*num)
            else:
                stack.append(int(stack.pop()/num))
            num = 0
            sign = s[i]
    return sum(stack)
```

3.4.4 Edit Distance

Edit distance is a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other.

Different types of edit distance allow different sets of string operations. For instance:

1. The **Levenshtein distance** allows deletion, insertion and substitution.
2. The **Longest common subsequence (LCS) distance** allows only insertion and deletion, not substitution.

3. The **Hamming distance** allows only substitution, hence, it only applies to strings of the same length.
4. The **Damerau–Levenshtein distance** allows insertion, deletion, substitution, and the transposition of two adjacent characters.
5. The **Jaro distance** allows only transposition.

3.5 Heap Use Cases

3.5.1 K Largest or Smallest Numbers

The best data structure to keep track of the largest or smallest K elements (in ascending sorted order) in streaming data is a heap. If we iterate through an array or read from a stream of data one element at a time, we can keep the K-th largest element in a heap such that each time we find a larger number than the smallest number in the heap, we do two things:

1. Take out the smallest number from the heap.
2. Insert the larger number into the heap.

This will ensure that we always have top K largest numbers in the heap. To easily remove the smallest value we may use a min-heap. Note, largest in in terms of ascending order from low to high values.

Heaps can also be used for maintaining order when selecting the largest/smallest elements to be processed first and then possibly modifying and re-inserting them for a secondary process. See greedy algorithm section.

Python Implementation

```python
import heapq

# For maintaining a class
class KthLargest:
    def __init__(self, k: int, nums):
        self.pq, self.k = [], k
        for n in nums:
            self.add(n)

    def add(self, val: int) -> int:
        heapq.heappush(self.pq, val)
        if len(self.pq) > self.k:
            heapq.heappop(self.pq)
        return self.pq[0]

# For one-time calculation
def findKthLargest(self, nums: List[int], k: int) -> int:
    return heapq.nlargest(k, nums)[-1]
    # Use sort for large values of k
    ## return sorted(nums)[k]
```

3.5.2 Two Heaps (Median of Data Stream)

If we maintain two heaps, we can keep track of the bigger half and the smaller half of a stream of data. The bigger half is kept in a min heap, such that the smallest element in the bigger half is at the root. The smaller half is kept in a max heap, such that the biggest element of the smaller half is at the root. Now, with these data structures, we have the potential median elements at the roots. If the heaps are no longer the same size, we can easily re-balance the heaps by popping an element off the one heap and pushing it onto the other.

Python Implementation

```
from heapq import heappush, heappushpop, heappop

class MedianFinder:
    def __init__(self):
        self.heaps = [], []

    def addNum(self, num):
        small, large = self.heaps
        # Invert values when pushing to or popping from the small max-heap.
    heapq uses a min heap by default.
        heappush(small, -heappushpop(large, num))
        if len(large) < len(small):
            heappush(large, -heappop(small))

    def findMedian(self):
        small, large = self.heaps
        if len(large) > len(small):
            return float(large[0])
        return (large[0] - small[0]) / 2.0
```

3.6 Tree Traversal

Given a binary tree, an **in-order traversal** (LNR) means to visit the left branch, then the current node, and finally, the right branch. A **pre-order traversal** (NLR) visits the current node before its child nodes, i.e. the root is always the first node visited. A **post-order traversal** (LRN) visits the current node after its child nodes, i.e. the root node is always the last node visited. A **breadth-first search** of a tree, a.k.a. **level order traversal**, iteratively visits all the nodes at the same height from left to right.

Recall, for a binary search tree an in-order traversal will result in an increasing sequence. Note, the successor of a node could be in either one of the ancestors or the right child, and the predecessor could be in either the ancestors or left child. Many tree analysis problems can be reduced to a sequence problem by first performing a traversal and storing the values into an auxiliary array, though some care may need to be taken to capture the the empty nodes with a special None value. Common sequence techniques that apply to tree problems include prefix sums and pattern matching.

A tree traversal visiting the root first will provide a valid topological ordering, i.e. pre-order, reverse in-order. On general DAGs possibly containing converging siblings, reverse post order (LRN) can be used to find a topological ordering.

Python Implementation

```python
def inorder(root):
    if root:
        inorder(root.left)
        print(root.val)
        inorder(root.right)

def inorder_iterative(root):
    res, stack = [], []
    while stack or root:
        if root:
            stack.append(root)
            root = root.left
        else:
            node = stack.pop()
            res.append(node.val)
            root = node.right
    return res

def preorder(root):
    if root:
        print(root.val)
        preorder(root.left)
        preorder(root.right)

def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.val)

def levelorder(root):
    q, level = collections.deque(root), 0
    while q:
        level += 1
        for _ in range(len(q)):
            node = q.popleft()
            print(node.val)
            if node.left:
                q.append(node.left)
            if node.right:
                q.append(node.right)
    return level

def verticalorder(root):
    columnTable = defaultdict(list)
    queue = deque([(root, 0)])
    while queue:
        node, column = queue.popleft()
        if node is not None:
            columnTable[column].append(node.val)
            queue.append((node.left, column - 1))
            queue.append((node.right, column + 1))
    return [columnTable[x] for x in sorted(columnTable.keys())]
```

3.7 Graph Traversal

3.7.1 Depth-First Search

Time Complexity: $\mathcal{O}(|V| + |E|)$ worst case, Space Complexity: $\mathcal{O}(|V|)$

In depth-first search (DFS), we start at the root (or another arbitrarily selected node) and explore each branch completely before moving on to the next branch. That is, we go deep first before we go wide.

Note that pre-order and other forms of tree traversal are a form of DFS. The key difference is that when implementing this algorithm for a graph, we must check if the node has been visited. If we don't, we risk getting stuck in an infinite loop. This is because graphs may not be acyclic and do not make distinctions between parent and child edges. We may also be able to reduce space by marking nodes in a graph as visited.

When a graph is implemented with an Adjacency List, the time complexity of DFS will be $\mathcal{O}(|V|+|E|)$, but when implemented with an Adjacency Matrix or if the graph is dense, it will be $\mathcal{O}(|V|^2)$. The complexity difference occurs due to the fact that in an Adjacency Matrix we must iterate through all possible adjacent edges to find existing outgoing edges, which takes $O(|V|)$ time summed over $|V|$ vertices. So $\mathcal{O}(|V|+|E|) = \mathcal{O}(|V|+|V|^2) = \mathcal{O}(|V|^2)$.

Python Implementation

```python
# DFS on graph represented as an adjacency matrix
def dfs(adj_matrix):
    if not adj_matrix: return []
    rows, cols = len(adj_matrix), len(adj_matrix[0])
    visited = set()
    directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

    def traverse(i, j):
        if (i, j) in visited:
            return
        visited.add((i, j))
        # Traverse neighbors.
        for direction in directions:
            next_i, next_j = i + direction[0], j + direction[1]
            if 0 <= next_i < rows and 0 <= next_j < cols:
                # Add in your question-specific checks.
                traverse(next_i, next_j)

    for i in range(rows):
        for j in range(cols):
            traverse(i, j)

# DFS on graph represented as an adjacency list
adj_list = {
    'A' : ['B','C'],
    'B' : ['A', 'C'],
    'C' : ['A', 'B'],
}
def dfs(adj_list):
    visited = set()
    def traverse(node):
        if node not in visited:
            visited.add(node)
            for neighbour in adj_list[node]:
                traverse(neighbour)
    for node in adj_list.keys():
        traverse(node)
```

3.7.2 Breadth-First Search

Time Complexity: $\mathcal{O}(|V| + |E|)$ worst case, Space Complexity: $\mathcal{O}(|V|)$

In a breadth-first search (BFS), we start at the root (or another arbitrarily selected node) and explore each neighbor before going on to any of their children. That is, we go wide before we go deep. A node x visits each of it's neighbors before visiting any of their neighbors. You can think of this as searching level by level out from x. An iterative solution involving a **queue** usually works best, this means that we should avoid using recursion, which will have an implicit stack LIFO ordering.

In an unweighted graph, since BFS explores all neighbors at the same depth, it can be used to find the shortest path between a start and a target node. In general, BFS often has similar function to DFS, but only BFS is optimal as a **single-source shortest path** algorithm in an unweighted graph.

Again, we find that when implemented with an Adjacency List, the time complexity of DFS will be $\mathcal{O}(|V|+|E|)$, but when implemented with an Adjacency Matrix or if the graph is dense, it will be $\mathcal{O}(|V|^2)$.

Python Implementation

```python
from collections import deque

# BFS on graph represented as an adjacency matrix
def bfs(adj_matrix):
    if not adj_matrix: return []
    rows, cols = len(adj_matrix), len(adj_matrix[0])
    visited = set()
    directions = ((0, 1), (0, -1), (1, 0), (-1, 0))

    def traverse(i, j):
        queue = deque([(i, j)])
        while queue:
            curr_i, curr_j = queue.popleft() #alt: pop(0)
            if (curr_i, curr_j) not in visited:
                visited.add((curr_i, curr_j))
                # Traverse neighbors.
                for direction in directions:
                    next_i, next_j = curr_i + \
                        direction[0], curr_j + direction[1]
                    if 0 <= next_i < rows and 0 <= next_j < cols:
                        # Add in your question-specific checks.
                        queue.append((next_i, next_j))

    for i in range(rows):
        for j in range(cols):
            traverse(i, j)

# BFS on graph represented by an adjacency list
def bfs(adj_list):
    visited = set()

    def traverse(node):
        queue = deque([node])
        while queue:
            cur = queue.popleft()
            for neighbor in adj_list[cur]:
                if neighbor not in visited:
```

```
                    visited.add(neighbor)
                    queue.append(neighbor)

    for node in adj_list:
        if node not in visited:
            traverse(node)
```

3.7.3 Bidirectional Search

Bidirectional search is used to find the shortest path between a source and destination node. It operates by essentially running two simultaneous breadth-first searches, one from each node. When their searches collide, we have found a path. If every node has at most k adjacent nodes and the shortest path from node s to node t has length d. Then, in a traditional breadth-first search we visit $\mathcal{O}(k^d)$ nodes while bidirectional search visits $\mathcal{O}(k^{d/2})$

3.7.4 Dijkstra's Shortest Path Algorithm

Time Complexity: $\mathcal{O}(|E| + |V| \log |V|)$ worst case

An algorithm for finding the shortest paths between nodes in a positive weighted, directed, acyclic graph. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. In breadth-first search we find the path with the fewest edges, but in Dijkstra's algorithm, we assign a number or weight to each segment, giving us the path with the smallest total weight.

Let the node at which we are starting be called the initial node and consider the distance to be from the initial node to a given node Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them iteratively, prioritizing shorter distances and edges with low weights in its path discovery process.

Similar to breadth first search, we use a queue for maintaining our traversal order, The code is further simplified by maintaining the already visited set with a min-priority queue (min-heap) and uses two dictionaries (hashmaps) for storing the graph as an adjacency list and for storing the minimum distances.

1. **Initialize data structures** –

   Convert graph to adjacency list in order to store node to (weight, neighbor) tuples as key-value pairs.

   Create a min-heap/min-priority queue of unvisited nodes, initialized with starting tuple of (0 cumulative distance, start node, empty path).

   Create a minimum distance hashmap and set the distance of the initial node to zero and exclude or set all other nodes to infinity.

   Create an empty set to track visited nodes.

2. **Enter top level while loop** –

   While the unvisited priority queue is non-empty, pop and unpack the vertex with the smallest distance and set it to the current node. Check that the current node has not already been visited and add to the visited set. Append the current node to the front of

the existing path. If the current node matches the desired end target node, we can return the path and cost.

3. **Enter inner for loop** –

   Loop over (weight, neighbor) tuples of the current node and check that they are not already in the visited set. Calculate the new distance by adding the neighbor's edge weight to the cumulative distance. Check if the neighbor already has a min value in our distance hash map and if the newly calculated distance is smaller than its existing value, save the smaller value to the hashmap and push the (new cost, neighbor, new path) tuple to the priority queue.

4. **Otherwise return infinity** –

   When the unvisited queue has depleted and the target node was not found, return failure response.

Python Implementation

```python
from collections import defaultdict
import heapq

edges = [
    ("A", "B", 7),
    ("A", "D", 5),
    ("B", "C", 8),
    ("D", "E", 15),
]

def dijkstra(edges, start, end):
    graph = defaultdict(list)
    for l, r, weight in edges:
        graph[l].append((weight, r))

    # queue of tuples: cumulative cost, vertex, path
    q = [(0, start, ())]
    visited = set()
    mins = {start: 0}

    while q:
        (cost, v1, path) = heapq.heappop(q)
        if v1 not in visited:
            visited.add(v1)
            path = (v1, path)
            if v1 == end: return (cost, path)

            for weight, v2 in graph.get(v1, ()):
                if v2 in visited: continue

                next_cost = cost + weight
                prev_cost = mins.get(v2, None)
                if prev_cost is None or next_cost < prev_cost:
                    mins[v2] = next_cost
                    heapq.heappush(q, (next_cost, v2, path))

    return float("inf")
```

3.7.5 A*

A* (pronounced "A-star") is a graph traversal and path search algorithm. It is a minor extension of Dijkstra's algorithm that builds in a heuristic for remaining distance used to indicate the relevance of paths which should be tried first.

One important aspect of A* is $F = G + H$. The F, G, and H variables are in our Node class and get calculated every time we create a new node.

- F is the total cost of the node.
- G is the distance between the current node and the start node.
- H is the heuristic — estimated distance from the current node to the end node.

A major practical drawback is its $\mathcal{O}(b^d)$ space complexity, as it stores all generated nodes in memory.

3.7.6 Bellman-Ford Shortest Path Algorithm

Time Complexity: $\mathcal{O}(|E||V|)$ worst case. Space Complexity: $\mathcal{O}(|V|)$.

The Bellman-Ford (BF) algorithm is a Single Source Shortest Path (SSSP) algorithm, i.e. it can find the shortest path from one node to any other. It is slower than Dijkstra's algorithm, but is capable of handling graph's with negative edges, in particular negative weighted cycles.

1. Let S be the start node and D be an array of length $|V|$ to record distances. Initialized every entry in D to ∞
2. Set $D[S] = 0$. Traverse the graph from adjacencies of S in any order.
3. Relax each edge $V-1$ times, i.e. $D[edge.to] = min(D[edge.to], D[edge.from] + edge.cost)$.
4. After traversing the graph $V-1$ times, repeat previous step and if any edge is still updated to a new minimum then there exists a negative cycle. So we set that node to $-\infty$.

3.7.7 Floyd-Warshall All-Pairs Shortest Path Algorithm

Time Complexity: $\mathcal{O}(|V|^3)$ worst case. Space Complexity: $\mathcal{O}(|V|^2)$.

The Floyd-Warshall (FW) algorithm is an **all-pairs shortest path** algorithm. This means it can find the shortest path between all pairs of nodes. With its high cubic time complexity, it's generally only ideal for graphs with less than a couple hundred nodes.

FW works well with a 2D adjacency matrix with ∞ representing no adjacency. The main idea of FW is to build up all possible intermediary paths between node i and j to find the optimal path. We use dynamic programming, covered in more detail later, to cache previous optimal solutions in an $n \times n$ memo table.

1. Define dp table where $dp[i][j] =$ shortest path from i to j routing through nodes $\{0, 1, \ldots, k\}$.
2. Iterate over 3 nested loops k, i, j to fill the dp table.

3. Transitions will be, $dp[i][j] = m[i][j]$ if $k = 0$. Otherwise $dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j])$. where we compute the solution for k in place, saving us a dimension of space.

   Note, the first term in the minimization is reusing the previous best distance routing through $0, \ldots, k-1$. The second term essentially measures the path from i to k plus the path from k to j.

4. If negative cycles are possible, add a subroutine to detect them

5. Return dp, the 2D matrix containing the shortest path pairs.

3.8 Graph Analysis Methods

3.8.1 Topological Sort

Time Complexity: $\mathcal{O}(|V| + |E|)$ worst case, Space Complexity: $\mathcal{O}(|V|)$

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. It can be useful for determining the the order of dependencies in anything that has a directed graph representation. A topological ordering is only possible if the graph has no directed cycles, that is, if it is a **directed acyclic graph** (DAG). Any DAG has at least one topological ordering, and algorithms are known for constructing a topological ordering of any DAG in linear time. A simple way to understand a topological ordering is to visualize the graph with each node placed in a row such that edges are only directed to the right. In general, topological orderings are not unique.

One algorithm for topological sorting is based on depth-first search. Simply put, run DFS and output the reverse of the finishing times of vertices, where finishing time corresponds to number of steps taken by DFS. The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort or the node has no outgoing edges (i.e. a leaf node). Each node n gets prepended to the output list L only after considering all other nodes which depend on n (all descendants of n in the graph). Specifically, when the algorithm adds node n, we are guaranteed that all nodes that depend on n are already in the output list L: they were added to L either by the recursive call to visit() which ended before the call to visit n, or by a call to visit() which started even before the call to visit n. We mark nodes with a state or color to detect cycles in the graph while it is being processed. Since each edge and node is visited once, the algorithm runs in linear time.

An alternative to the DFS approach is known as **Kahn's algorithm** and corresponds to BFS. Along with an adjacency list of outgoing edges, we will need to maintain a list of the counts of incoming edges, a.k.a its indegree or prerequisites. We select nodes that have no incoming edges and add that subset in any order to the back of the queue. To process a node, we pop from the front of the queue, add the node's value to our resulting ordering list, then we emulate removing the node from the graph by decrementing the corresponding indegree count of their neighbors. We repeat this process until no nodes are left or we detect a cycle. If not all nodes are in the output, there must be a cycle so no valid ordering exists. Nodes with no incoming nor outgoing edges can be included in any position of the final ordering, though are usually placed at the beginning.

Python Implementation

```python
def dfs_topological_sort(adj_list):
    order = []
    visiting, visited = set([]), set([])

    def dfs(node):
        if node in visiting: # A cycle was detected
            return False
        visiting.add(node)
        for neighbor in adj_list[node]:
            if neighbor in visited:
                continue
            if not dfs(neighbor):
                return False
        visiting.remove(node)
        visited.add(node)
        order.append(node)
        return True

    for node in adj_list:
        if node in visited:
            continue
        if not dfs(node):
            return []
    return order

from collections import deque, Counter
def khans_topological_sort(adj_list):
    # initialize indegree counter
    # this can be done with an array from 0-n or a counter
    # be sure to include nodes with 0 count
    in_degree = Counter(({node : 0 for node in adj_list.keys()})

    for node in adj_list
        for neighbor in adj_list[node]:
            in_degree[neighbor] += 1

    # repeatedly pick off nodes with an indegree of 0
    output = []
    queue = collections.deque([c for c in in_degree if in_degree[c] == 0])

    while queue:
        cur = queue.popleft()
        output.append(cur)
        # decrement indegree of neighbors, append them to queue when 0
        for neighbor in adj_list[c]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # If not all nodes are in the output, there must be a cycle so
    # no valid ordering exists.
    if len(output) < len(in_degree):
        return []
    return output
```

3.8.2 Tarjan's Strongly Connected Component Algorithm

Time Complexity: $\mathcal{O}(|E| + |V|)$ worst case.

A **strongly connected component** (SCC) can be thought of as self-contained cycles within a directed graph where every vertex in a given cycle can reach every other vertex in the same cycle. A SCC is analogous to a connected component in an undirected graph.

A **low-link** of a node is the smallest node id reachable from that node where the id corresponds to a rank obtained by performing a DFS with some starting node being 0. The low-link value is highly dependent on the order of traversal in a DFS, which is random.

To handle the random traversal order of a DFS, Tarjan's algorithm maintains a stack of valid nodes from which to update low-link values from. Nodes are added to the stack of valid nodes as they're explored for the first time and are removed each time a complete SCC is found. To update node u's low-link value to node v's low-link value, there must be a path of edges from u to v and v must be on the stack. Then we find that each strongly connected component will contain the same low-link value.

1. Mark the id of each node as unvisited.
2. Perform a DFS from an arbitrary node. Upon visiting a node, assign it an id and a low-link value. Mark current nodes as visited and add them to a seen stack.
3. On DFS callback, i.e. after reaching a leaf or already vising all neighbors, we begin backtracking. If the previous node is on the seen stack, then minimize the current node's low-link value with the last node's low-link vaue. This allows low-link values to propegate through cycles being tracked in the stack.
4. After visiting all neighbors, if the current node started a connected component, i.e. if its id equals its low-link value, then pop all nodes off the stack until the current node is reached. This will remove all the nodes associated with the current connected component and reset our stack reference.
5. Pick another node at random and continue this process until all nodes are visited.

Python Implementation

```python
# Return all critical connections (SCC) in the network in any order.
def tarjan(connections):
    g = collections.defaultdict(list)
    for u, v in connections:
        g[u].append(v)
        g[v].append(u)

    N = len(connections)
    # n nodes numbered from 0 to n-1, these lists could also be dictionaries
    lev = [None] * N
    low = [None] * N

    def dfs(node, parent, level):
        # already visited
        if lev[node] is not None:
            return

        lev[node] = low[node] = level
        for neighbor in g[node]:
            if not lev[neighbor]:
                dfs(neighbor, node, level + 1)

        # minimal level in the neighbors including self excluding the parent
        cur = min([level] + [
```

```
            low[neighbor]
            for neighbor in g[node]
            if neighbor != parent
        ])
        low[node] = cur

    dfs(0, None, 0)

    ans = []
    for u, v in connections:
        if low[u] > lev[v] or low[v] > lev[u]:
            ans.append([u, v])
    return ans
```

3.8.3 Kruskal's Minimum Spanning Tree Algorithm

Time Complexity: $\mathcal{O}(|E| \log |V|)$, Space Complexity: Varies on implementation

Given an undirected, connected graph, a **Minimum Spanning Tree** (MST) is a subset of the edges which connects all vertices together (without creating cycles) while minimizing the total edge cost. It is possible for graphs to have multiple MSTs. In an unconnected graph, multiple sets of MSTs will form a **Minimum Spanning Forrest** (MSF).

1. Sort edges by ascending edge weight. This can be done by adding the (weight, to, from) tuples into a min-heap.
2. Walk through sorted edges and look at the two nodes the edges belong to. If the nodes are already unified we don't include this edge, otherwise we include and unify the nodes. Unified edges are groupings of sub-trees in the graph. The algorithm makes use of the *union-find* data structure to efficiently determine if an edge belongs to an existing group. See data structure section for UF implementation details.
3. The algorithm terminates when every edge has been processed or all the vertices have been unified. If there are disconnected components, we cannot find a MST.

Python Implementation

```
import heapq
def kruskal(n, edges):
    uf = UnionFind(size=n)
    heap = []
    count, cost = n, 0

    for a, b, weight in edges:
        heapq.heappush(heap, (weight, a, b))

    while heap:
        weight, a, b = heapq.heappop(heap)
        if uf.parent(a) != uf.parent(b):
            uf.union(a, b)
            count -=1
            cost += weight

    # if there are multiple connected components we cannot form a MST
    if count > 1:
        return  float('inf')

```

```
21      return cost
```

3.8.4 Prim's Minimum Spanning Tree Algorithm

Time Complexity: $\mathcal{O}(|E|\log(|V|)$, Space Complexity: Varies on implementation

Prim's algorithm is a greedy MST algorithm that works well on dense graphs. On these graphs, it meets or rivals other popular MST/MSF algorithms like Kruskal's and Boruvka's. The lazy version of Prim's algorithm uses a priority queue and has running time of $\mathcal{O}(|E|\log(|E|))$, while the eager version has running time of $\mathcal{O}(|E|\log(|V|))$

Lazy Prim's MST

1. Maintain a min Priority Queue (PQ) that sorts edges based on min edge cost. This will be used to determine the next node to visit and the edges used to get there.
2. Start the algorithm on any node s. Mark s as visited and iterate over all edges of s, adding them to PQ.
3. While PQ is not empty and an MST has not been formed, dequeue the next cheapest edge from PQ. If the dequeued edge is stale, i.e. it has already be visited, then skip it and poll again. Otherwise mark the current node as visited and add it to the MST.
4. Iterate over the new current node's edges and add all its edges to the PQ. Do not add edges to the PQ which point to already visited nodes.

In the eager version, instead of adding edges to the priority queue as we iterate over the edges of a node, we update the destinations node's most promising incoming edge. We do this using an indexed priorty queue (IPQ).

3.9 Recursive Problems

We can interpret recursive solutions using the following categories:

- A **bottom-up** approach is often the most intuitive recursive pattern. We start with knowing how to solve the problem for a simple case, like a list with only one element. Then we figure out how to solve the problem for two elements, then for three elements, and so on. The key here is to think about how you can build the solution for one case off of the previous case (or multiple previous cases).
- The **top-down** approach can be more complex since it's less concrete. But sometimes, it's the best way to think about the problem. In these problems, we think about how we can divide the problem for case N into subproblems. Be careful of overlap between the cases.
- The **half-and-half** approach: in addition to top-down and bottom-up approaches, it's often effective to divide the data set in half. For example, binary search works with a "half-and-half" approach. When we look for an element in a sorted array, we first figure out which half of the array contains the value. Then we recurse and search for it in that half.

Recall, Python has a default maximum call stack of 1000. A tail recursive algorithm (i.e. one where the recursive step is the very last statement in the function) can always be converted into an iterative one. In Python, an iterative version is almost always faster than the equivalent tail recursion because Python deliberately lacks a feature called **tail call optimization**, not making the trade-off for speed gains in favour of keeping debugging information.

A problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems. For example, consider the recursive generation of the nth Fibonacci number.

A problem is said to have **optimal substructure** if an optimal solution can be constructed efficiently from optimal solutions of its subproblem. Another way to view this is that when given the optimal solution, the solution of the sub-problems will also be optimal. For example: if we know a path (A, B, C, D) is the shortest possible distance between A and D, then the shortest path from B to D must also pass through C.

3.9.1 The Decision Tree (DAG) Model

Many problems involve finding a solution by processing the state space, i.e. the space of all possible results, in a search for a solution that is both correct and optimal. An algorithm processes the state space by making a series of decisions and completes when it knows it has found the best resulting set of decisions. We can think of this decision space in terms of a graph structure in which each node being traversed is an inclusion to our resulting decision sequence which is represented by a path. Then, each of the outgoing edges in the graph indicates a reachable link from one state to another.

Recall that a **directed acyclic graph (DAG)** is a directed graph that does not contain any cycles. This can be understood as a state space in which no decision chosen from a unique subset of choices can be reached again or included multiple times, i.e. there are no infinite loops.

1. **Depth-First Search** – In the most straightforward case, we are given a generic state space represented as a directed graph that may or may not be acyclic and we want to determine whether two states are reachable from one another.

   DFS is the ideal solution since no heuristic is available as the intermediary states do not reveal information about the location of the desired state, except when it is found or unreachable as a result visiting all neighbors or reaching a leaf node without neighbors. Although overlapping subproblems exist, i.e. many pathways may branch from a shared path, we can not optimize the final result through a heuristic made in the algorithm's local decisions, so there is no optimal substructure. Despite doing some bookkeeping to prevent duplicate work in overlapping subproblems, we are mostly blindly searching the unvisted state space.

2. **Backtracking** – Similar to DFS, this approach can be used when there are overlapping subproblems and no optimal substructure exists, i.e. we can not derive optimal solutions to subproblems even when given a global optimal solution. Additionally, any local cost analysis will not be enough to help us determine which path to choose from our future choices. However, when analyzing a current state, a heuristic exists that determines if

a path cannot possibly be appended to produce a valid solution, so that a path can be avoided or pruned without first reaching its leaves as is necessary in DFS.

3. **Greedy Algorithm** – This approach is ideal when overlapping subproblems exist and a cost analysis exists in which making optimal local decisions results in finding the globally optimal result. That is, the cost analysis heuristic is independent of previous states and only involves processing the subset of decisions at the current local state.
4. **Dynamic Programming** – Similar to the greedy algorithm, this approach is ideal when there are overlapping subproblems and optimal substructure exists, i.e. given the optimal solution, sub-problems will also be optimal. Although a greedy heuristic may sometimes solve problems with optimal substructure, a DP approach is more reliable when cost analysis is largely dependent on previous decision states.

3.9.2 Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, notably constraint satisfaction problems that incrementally builds candidates to the solutions and abandons a candidate as soon as it determines that they cannot possibly be completed to a valid solution. It is useful for exhaustive recursive problems in which the solution must follow some constraints. We may define a policy for recursion and when a computation does not meet the constraints, we halt or backtrack on the exhaustive recursion. The call stack remembers our previous choices and decides what choice to make next. Thus, there are three key things to keep in mind:

1. Our choice – What choice do we make at each call of the function? Recursion expresses this decision
2. Our constraints – When do we stop following a certain path?
3. Our goal – What's our target? What are we trying to find?

We can think of the run-time of a backtracking algorithm as $\mathcal{O}(b^d)$, where b is the branching factor and d is the maximum depth of recursion.

DFS is a special type of backtracking where the process of backtracking only takes place in the leaf nodes, whereas general backtracking algorithms can also preemptively reject failing branches in the state space tree. Thus, DFS maintains the entire tree structure while backtracking creates a pruned tree.

The backtracking pattern can be useful for generating combinatoric choices in polynomial time, i.e. permutations, powersets, subsets, combinations, etc.

Python Implementation

```
def backtrack(candidate):
    if find_solution(candidate):
        return candidate
    # iterate all possible candidates.
    for next_candidate in list_of_candidates:
        if is_valid(next_candidate):
            # try this partial candidate solution
            place(next_candidate)
            # given the candidate, explore further.
            backtrack(next_candidate)
```

```
            # backtrack/prune
            remove(next_candidate)

def powerset(nums):
    result, partial = [], []
    def backtrack(idx):
        if idx == len(nums):
            result.append(partial[:])
            return
        partial.append(nums[idx])
        backtrack(idx + 1)
        partial.pop()
        backtrack(idx + 1)
    backtrack(0)
    return result

## Generate all combinations of well-formed parentheses with n pairs.
def generate_parenthesis(n):
    ans = []
    def backtrack(S='', opened=0, closed=0):
        if len(S) == 2 * n:
            ans.append(S)
            return
        if opened < n:
            backtrack(S + "(", opened + 1, closed)
        if closed < opened:
            backtrack( S + ")", opened, closed + 1)
    backtrack()
    return ans

assert generate_parenthesis(3) == [
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

3.9.3 Greedy Algorithms

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. They never look backwards to see if they could optimise globally. This is the main difference between Greedy and Dynamic Programming.

Greedy algorithms are only ideal for problems which have optimal substructure. Typically, a greedy algorithm is used to solve a problem with optimal substructure if it can be proven by induction that it is optimal at each step. Otherwise, provided the problem exhibits overlapping subproblems, then dynamic programming is preferable. If there are no appropriate greedy algorithms and the problem fails to exhibit overlapping subproblems, often a lengthy but straightforward search of the solution space is the best alternative. If the problem is NP-Complete, a greedy algorithm is likely the best approximation function.

A common aspect of greedy algorithms is to sort the choices and then selecting them in the appropriate order. This will often be the bottleneck step, bounding the algorithm to $\mathcal{O}(n \log n)$. In the unweighted interval/job scheduling problem, we first sort intervals by their end times in ascending order i.e. we schedule them based on earliest finishing times.

3.9.4 Dynamic Programming & Memoization

Dynamic programming (DP) corresponds to a careful bruteforce approach, taking an exponential algorithm and making it polynomial. The basic idea of dynamic programming is to take a problem, split it into subproblems, solve the subproblems, and re-use the solutions to the subproblems. A dynamic programming solution can only be used if the problem possesses the optimal substructure property, i.e. its global optimal solution can be constructed efficiently from optimal solutions of its subproblems. Recall, overlapping subproblems exist if the problem can be broken down into subproblems which are reused several times.

Memoization refers to the technique of caching and reusing previously computed results. Some people call top-down dynamic programming "memoization" and only use "dynamic programming" to refer to bottom-up work. A memoized function only recurses the first time it's called with the future cached calls costing $\mathcal{O}(1)$. In general, the time complexity will be the number of subproblems needed to be solved multiplied by the running time per subproblem. We no longer need to count recursions or the call stack. It's important to note that we want to perform an exhaustive search with the values we're caching. Doing this recursively with memoization can be more obvious when the cached values are in the recursive function header. The typical procedure for a memoized approach will be to instantiate a cache table in the driver function, where features or arguments will map to a previously computed value. Then define a recursive function using those args which will first check for success and failure base cases and return appropriate values, then will check if the given args already exist as keys in the cache table, returning the mapped value if they do. Otherwise it will recurse with modified/reduced args and derive a value from the recursion, adding it to our cache and then returning it. The driver function will call the recursive function with the desired arguments of the original problem and return its results.

A bottom-up solution uses **tabulation** to only store the relevant calls needed for future computations. With tabulation, we have to come up with an ordering which is often less intuitive than memoized solutions. If all sub-problems must be solved at least once, a bottom-up tabulated dynamic programming algorithm usually outperforms a top-down memoized algorithm by a constant factor. The typical procedure for a tabulated approach involves defining a memo table with dimensions equal to the number of features and then initializing the base case values in the relevant rows or columns of the table. The function will then iteratively update the tables entries according to a state transition rule, where the order and range of iterations over states occurs in a way so that previous states referenced by the transition rule will already exist. We then return the index of the table corresponding to the state values of the original problem.

A useful strategy for solving dynamic programming problems is as follows:

1. Define subproblems. Subproblems for sequences will be one of the following:
   (a) Suffixes – x[i:] for all i. Time complexity: $\mathcal{O}(n)$
   (b) Prefixes – x[:i] for all i. Time complexity: $\mathcal{O}(n)$
   (c) Subsequences – x[i:j] for all $i \leq j$. Time complexity: $\mathcal{O}(n^2)$
2. Guess part of the solution. There are two kinds of guessing:
   (a) Which existing subproblems to use to solve bigger subproblem.
   (b) Or add more subproblems to guess, remember more features of the solution variations.

3. Relate subproblem solutions with a recurrence, i.e. define the state transitions.
4. Construct an algorithm by recursion and memoization (need acyclic DAG) or building a DP table bottom up (need topological order). Note: The topological order, i.e. the order in which subproblems are executed, should be from smallest to largest.
5. Solve original problem. The runtime will be the number of subproblems multiplied by the running time per subproblem.

Unordered Choices Without Repetitions in 1D

Given a set of choices, we are tasked with including or excluding members of this set in any order, typically with the aim of maximizing, minimizing, or exactly matching a specific target, cumulative sum, or product of these choices under some constraint. Although the choices can be sorted, the constraint prevents a greedy algorithm from being correct. In general, our actual choices are unimportant and only their cumulative value matters and needs to be tracked.

Thus the features of the recursive states are the current choice being considered and the remaining constraint which produce the ideal cumulative value for a given state. The state transitions generally iterate over the different elements taking a max or min of their inclusion or exclusion. Subproblems are either prefixes or suffixes of the given choices since ordering is irrelevant.

Notable Problems: 0/1 Knapsack, Partition Equal Subset Sum

Python Implementation

```python
# Maximize the amount of value we can fit within the knapsacks weight capacity
from functools import lru_cache
def knapsack_recursive(capacity, weights, values):
    @lru_cache(None)
    def dp(capacity, idx):
        if idx == 0 or capacity == 0:
            return 0
        if (weights[i-1] > capacity):
            return dp(capacity, idx-1)
        else:
            return max(
                values[idx-1] + dp(capacity - weights[i-1], idx - 1),
                dp(capacity, idx - 1)
            )
    return dp(capacity, len(values))

def knapsack_bottom_up(item_weights, item_values, capacity):
    n = len(item_weights)
    dp = [[0] * (capacity + 1) for _ in range(n)]
    for i in range(1, n):
        for w in range(1, capacity + 1):
            wi = item_weights[i]
            vi = item_values[i]
            if wi <= w:
                dp[i][w] = max([dp[i - 1][w - wi] + vi, dp[i - 1][w]])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n - 1][capacity]
```

Unordered Choices With Repetitions in 1D

As in the previous case we are given a set of choices and make unordered selections, but now we are able to repeat our selections any number of times. Similarly we might want to optimize under a constraint, exactly match a target value, or count the number of combinations that can match the target.

Again, the features will be the current choice being considered, and the remaining constraint. However, instead of iterating over the different choices and taking a max or min of their inclusion or exclusion once per transition, the state transitions will iterate over all choices for every state, allowing for repetitions.

Notable Problems: Coin Change (Minimize Coins), Coin Change (Count Ways), Word Break II

Python Implementation

```python
# Compute the fewest number of coins that are needed to sum to an amount
def coin_change_min(coins, amount):
    MAX = float("inf")
    dp = [0] + [MAX] * amount
    for i in range(1, amount + 1):
        dp[i] = min(dp[i - c] if i - c >= 0 else MAX for c in coins) + 1
    return dp[-1] if dp[-1] != MAX else -1

# Count the number of combinations of the given coins can sum to the amount
def coin_change_ways(amount, coins):
    dp = [0] * (amount + 1)
    for coin in coins:
        for x in range(coin, amount + 1):
            dp[x] += dp[x - coin]
    return dp[amount]
```

Ordered Choices in 1D Sequences

We are given an ordered sequence and are typically given an implicit set of choices and an implicit constraint on which choices can be made in a given state. In a sense, we can view this category of problem as a state machine. Since there is only one dimension, processing a sequence from left to right can be interpreted as moving forward in time while making decisions at each step, generally with the goal of optimizing a cumulative sum or product. As with the unordered 1D case, we can iterate over our choices using their prefixes or suffixes.

The difference between this and the knapsack problem is that the order of our decisions matters and is usually subject to some constraints which often rely on how adjacent values can interact with each other. Because of this we may want to pre-process our given sequence using cumulative prefix or suffix sums or products. When dealing with products, it can be useful to track both a cumulative min so far as well as the desired cumulative max, because the min can be inverted after multiplying with a negative value and become the new maximum.

Notable Problems: Climb stairs, Buy and Sell Stocks, House Robbers, Maximum Product Subarray

Python Implementation

```python
# Count distinct ways to reach the top of a set of stairs climbing 1 or 2 steps at a time
def climb_stairs(n): ## Fibonacci
    if n <= 1: return n
    dp = [1, 2]
    for i in range(3, n+1):
        curr = dp[0] + dp[1]
        dp[0], dp[1] = dp[1], curr
    return dp[1]

# House Robbers II: Max profit without robbing two adjacent houses in a circular street
def rob_II(nums):
    if len(nums) == 0 or nums is None: return 0
    if len(nums) == 1: return nums[0]

    def rob_I(nums):
        t1 = t2 = 0
        for current in nums:
            temp = t1
            t1 = max(current + t2, t1)
            t2 = temp
        return t1

    return max(rob_I(nums[:-1]), rob_I(nums[1:]))

# Best Time to Buy and Sell Stock III (K=2 transactions only)
def maxProfit(prices: List[int]) -> int:
    @functools.lru_cache(None)
    def recursive(idx, k, holding):
        if k == 0 and not holding or idx == len(prices):
            return 0
        if holding:
            memo[args] = max(
                recursive(idx + 1, k, holding),
                prices[idx] + recursive(idx + 1, k - 1, not holding),
            )
        else:
            memo[args] = max(
                recursive(idx + 1, k, holding),
                -prices[idx] + recursive(idx + 1, k, not holding),
            )
        return memo[args]
    return recursive(0, 2, 0)
```

Ordered Choices in 2D Sequences (Subsequences)

As with before, we are given an input where order matters but instead of only traversing left to right in one dimension, there are two degrees of freedom and each choice has two dimensions. Consequentially, we can think of interacting with the state space in one of two ways: as searching for subsequences of a given sequence, or as traversing or path-finding in a graph using DFS. As such, the input we're given might be in the form of a one or two-dimensional array.

Notable Problems: Longest Increasing Subsequence (LIS), Longest Common Subsequence (LCS)

Python Implementation

```python
# Given an unsorted array of integers, find the length of longest increasing subsequence.
def lis(nums):
    n = len(nums)
    if not n: return 0
    dp = [1] * n
    for i in range(1, n):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

# Given two strings text1 and text2, return the length of their longest common subsequence.
import functools
def lcs_recursive(text1: str, text2: str) -> int:
    @functools.lru_cache(None)
    def helper(i,j):
        if i < 0 or j < 0:
            return 0
        if text1[i] == text2[j]:
            return helper(i-1, j-1) + 1
        return max(helper(i-1, j), helper(i, j-1))
    return helper(len(text1) - 1, len(text2) - 1)

def lcs_bottom_up(X, Y):
    m, n = len(X), len(Y)
    L = [[None] * (n + 1) for i in range(m + 1)]
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])
    return L[m][n]
```

Ordered Choices in 2D+ Sequences (DFS)

As mentioned above, we can interpret the state space as a graph or 2D grid and can use DFS with memoization to exhastively search for paths. DP becomes valid when constraints are given on the directions we're allowed to traverse in, i.e. only moving up and right, creating optimal substructure and allowing us to build up a solution from subproblems. Sometimes we can reduce our DP table to single array by cumulatively storing row and column information. We may also be able to reduce space by using the grid to overwrite memoized values in-place. DFS can generalize to larger dimensions as long as the constraints on traversal provide optimal substructure.

Notable Problems: Unique paths in grid, Unique paths below diagonal (Catalan Number), Minimum/Maximum path weight, Unique paths with obstacles, Minimum Knight Moves

Python Implementation

```python
# Unique paths in grid
def unique_paths_2D_memo(self, m, n):
    if not m or not n:
```

```
        return 0
    dp = [[1] * n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[-1][-1]

def unique_paths_1d_memo(self, m, n):
    if not m or not n:
        return 0
    dp = [1] * n
    for _ in range(1, m):
        for j in range(1, n):
            dp[j] += dp[j-1]
    return dp[-1]

# Minimum path sum
def minPathSum(self, grid):
    m, n = len(grid), len(grid[0])
    for i in range(1, n):
        grid[0][i] += grid[0][i-1]
    for i in range(1, m):
        grid[i][0] += grid[i-1][0]
    for i in range(1, m):
        for j in range(1, n):
            grid[i][j] += min(grid[i-1][j], grid[i][j-1])
    return grid[-1][-1]
```

Unordered Choices in 2D Sequences (Binary Search)

Again, we're given a two-dimensional input but are now able to reorder the subarrays. A common trick is to first sort the subarrays by their first dimensional value and possibly inversely sort on their second dimensional value.

Now that we have a sorted 2D array of choices, we can make use of binary search to find target values. This may further reduce a polynomial exhaustive search runtime to a logarithmic one as is done when finding the length of longest increasing subsequence (LIS). Since we are using binary search, we can't initialize the dp array with empty or placeholder values and instead need to iteratively append to it.

Notable Problems: Weighted Interval Scheduling, Russian Doll Envelopes, Rectangular Block Stacking, Length of Longest Increasing Subsequence

A closely related version of this category is a **minimax** problem where we aim to minimize the possible loss for a worst case (maximum loss) scenario. This is generally in the form of determining the minimum number of guesses or the minimum cost required to perform a binary search.

Super egg drop, Guessing game with cost

Python Implementation

```
# Weighted Interval Scheduling
import bisect
def weightedJobScheduling(startTime, endTime, profit):
    # Similar to unweighted greedy approach, we sort on earliest finish time
    jobs = sorted(zip(startTime, endTime, profit), key=lambda v: v[1])
```

```
    dp = [[0, 0]]
    for s, e, p in jobs:
        i = bisect.bisect(dp, [s + 1]) - 1
        if dp[i][1] + p > dp[-1][1]:
            dp.append([e, dp[i][1] + p])
    return dp[-1][1]

def length_of_LIS(nums):
    dp = []
    for num in nums:
        idx = bisect.bisect(dp, num)
        if idx == len(dp):
            dp.append(num)
        elif dp[idx] >= num:
            dp[idx] = num
    return len(dp)

# Russian Doll Envelopes
def maxEnvelopes(arr):
    # sort increasing in first dimension and decreasing in second
    arr.sort(key=lambda x: (x[0], -x[1]))
    return length_of_LIS([i[1] for i in arr])

# Given K eggs and N floors find the min number of moves needed to guarantee you
    find the lowest floor needed to break the egg
def superEggDrop(self, K: int, N: int) -> int:
        @lru_cache(None)
        def dfs(eggs, floors):
            if eggs == 1 or floors in [0, 1]:
                return floors
            lo, hi = 0, floors
            while lo < hi:
                mid = (lo + hi) // 2
                left, right = dfs(eggs - 1, mid-1), dfs(eggs, floors - mid)
                if left < right:
                    lo = mid + 1
                else:
                    hi = mid
            res = 1 + max(dfs(eggs-1, lo-1), dfs(eggs, floors-lo))
            return res
        return dfs(K, N)
```

Turn-Based Games (Double-ended queue)

The main idea is to use tuples to store the game results of two people in the two-dimensional DP output. In general, there are three states: the starting index i, the ending index j, and the player whose turn it is. The state transitions decide between two choices: the leftmost index, or the rightmost index. In some cases, we can use a sliding window by modifying the array to be circular using modular arithmetic on indices in a doubled range while maintaining up-to-date cumulative values using decrements.

Notable Problems: Nim Game, Stone Game, Line of Wines, Binary Tree Coloring Game, Maximum Points You Can Obtain from Cards

Python Implementation

```
def stoneGame(piles: List[int]) -> bool:
    n = len(piles)
```

```
    dp = [[0] * n for i in range(n)]
    for i in range(n):
        dp[i][i] = piles[i]
    for d in range(1, n):
        for i in range(n - d):
            dp[i][i + d] = max(
                piles[i] - dp[i + 1][i + d],
                piles[i + d] - dp[i][i + d - 1],
            )
    return dp[0][-1] > 0

def maxScore_dp(cardPoints, limit):
        if limit <= len(cardPoints): return sum(cardPoints)
        @lru_cache(None)
        def dfs(i, j, limit):
            if limit == 0: return 0
            res = max(
                cardPoints[i] + dfs(i + 1, j, limit - 1),
                cardPoints[j] + dfs(i, j - 1, limit - 1),
            )
            return res
        return dfs(0, len(cardPoints) - 1, limit)

def maxScore_sw(cardPoints, limit):
        ans = win = 0
        L = len(cardPoints)
        for i in range(L - limit, L + limit):
            win += cardPoints[i % L]
            if i >= L:
                win -= cardPoints[(i - limit) % L]
            ans = max(win, ans)
        return ans
```

3.10 Numerical Methods

3.10.1 Bit Manipulation and Set Operations

1 byte is comprised of 8 bits. Any integer or character can be represented using bits, which we call its binary form (containing only 1 or 0) or its base 2 representation.

For example, $14 = (1110)_2 = 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$

At the heart of bit manipulation are the bit-wise operators:

- **AND** – &

  $A = 5 = (101)_2$, $B = 3 = (011)_2$

  $A \ \& \ B = (101)_2 \ \& \ (011)_2 = (001)_2 = 1$

- **OR** – |

  $A = 5 = (101)_2, B = 3 = (011)_2$

  $A|B = (101)_2 \mid (011)_2 = (111)_2 = 7$

- **NOT** – $\sim$

  $N = 5 = (101)_2$

$\sim N = \sim 5 = \sim (101)_2 = (010)_2 = 2$

- **XOR** – $\wedge$

  The exclusive-or operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but 0 if both are 1.

  $A = 5 = (101)_2$, $B = 3 = (011)_2$

  $A \wedge B = (101)_2 \wedge (011)_2 = (110)_2 = 6$

- **SHIFT** – a << b, a >> b

  Left shift operator shifts some number of bits to the left and appends 0 at the end. Left shift is equivalent to multiplying the bit pattern with 2^k (if we are shifting k bit).

  $1 << n = 2^n$

  Right shift operator shifts some number of bits, to the right and appends 1 at the end. Right shift is equivalent to dividing the bit pattern by 2^k (if we are shifting k bits).

  $16 >> 4 = 1$

- **Set a bit** – $A| = 1 <<$ bit

  $(1 << n)$ will return a number with only nth bit set. So if we OR it with x it will set the nth bit of x.

- **Clear bit** – $A \ \& = \sim (1 <<$ bit$)$

- **Test bit** – $(A \ \& \ 1 <<$ bit$)! = 0$

- **Extract last bit** – $A \ \& \ -A$ or $A \ \& \sim (A - 1)$ or $x \wedge (x \ \& \ (x - 1))$

  $(-x)$ is the two's complement of x and will have all the bits flipped that are on the left of the rightmost 1 in x. So $x \ \& \ (-x)$ will return rightmost 1.

- **Remove last bit** – $A \ \& \ (A - 1)$

- **Get all 1-bits** – ~ 0

A big advantage of bit manipulation is that it can help to iterate over all the subsets of an N-element set. If we represent each element in a subset with a bit, which can be either 0 or 1, we can use a bit array to denote whether a corresponding element belongs to this given subset or not. Then, each bit pattern will represent a possible subset and set operations can be performed with bit operations. Basic operations are outlined below,

- **Set union** – $A|B$
- **Set intersection** – $A \ \& \ B$
- **Set subtraction** – $A \ \& \sim B$
- **Set negation** – ALL BITS $\wedge A$ or $\sim A$

Basic use cases of bit manipulations are given below,

1. **Check if a given number x is a power of 2** –

   The binary representation of $(x - 1)$ will have all the same bits as x except for the rightmost 1 in x and all the bits to the right of the rightmost 1.

Thus, x & $(x-1)$ will have all the bits equal to x except for the rightmost 1 in x.

If the number is neither zero nor a power of two, it will have 1 in more than one place. So if x is a power of 2 then x & $(x-1)$ will be 0.

$x = 4 = (100)_2$

$x - 1 = 3 = (011)_2$

x & $(x-1) = 4$ & $3 = (100)_2$ & $(011)_2 = (000)_2$

2. **Count the number of ones in the binary representation of the given number** –

   Recall, $(x-1)$ has the rightmost 1 and all bits to the right of it are flipped in comparison to x. So, setting $x = x$ & $(x-1)$ will reduce the number of 1 bits by 1. We may do this repeatedly while incrementing a counter until $x = 0$.

3. **Find the largest power of 2 which is less than or equal to the given number N** –

   Change all the bits which are at the right side of the most significant digit to 1. Then the number will become $x + (x-1) = 2 * x - 1$, where x is the required answer.

   For a 16 bit integer, $N = N|(N >> 1); N = N|(N >> 2); N = N|(N >> 4); N = N|(N >> 8);$ return $(N+1) >> 1$

Two's complement is a mathematical operation on binary numbers, and is an example of a radix complement. The two's complement is calculated by inverting the digits and adding one. The two's complement of an N-bit number is defined as its complement with respect to 2^N. For instance, for the three-bit number 010, the two's complement is 110, because 010 + 110 = 1000.

Python Implementation

```
a = set(['a', 'b', 'c', 'd'])
b = set(['c', 'd', 'e', 'f'])
c = set(['a', 'c'])

## Union
print(a | b)
print(a.union(["foo", "bar"]))

## Intersection
print(a & b)
print(a.intersection(["b"]))

## Difference
print(a - b)
print(a.difference(["foo"]))

## Subset
print(c < a)
print(a.issubset(["a", "b", "c", "d", "e", "f"]))

## Symmetric Difference
print(a ^ b) # {'e', 'a', 'b', 'f'}
print(a.symmetric_difference(["a", "b", "e"]))

print(a.issuperset(["b", "c"]))
print(a.isdisjoint(["y", 'z']))
```

```
a.intersection_update(["a", "c", "z"])
print(a)

# Given an array containing n distinct numbers taken from [0, n], find the one
    that is missing from the array.
import operator
def missingNumber(self, nums):
    return reduce(operator.xor, nums + range(len(nums)+1))

# Use a 26-bit bitmask to indicate which lower case latin characters are inside
    the string.
def get_mask(word):
    mask = 0
    for c in set(word):
      mask |= (1 << (ord(c) - ord('a')))
    return mask

def has_common_chars(word_1, word_2):
    mask_1, mask_2 = map(get_mask, [word_1, word_2])
    return mask_1 & mask_2 > 0
```

3.11 Combinatorial Methods

View combinatorics notebook[3] for more details.

3.11.1 Permutations

1. Order of items matters.
2. Counts do not include duplication or removals of items.
3. Collection of counts could be stored in *arrays*.

$$P(n, m) = \frac{n!}{(n-m)!}$$

Python Implementation

```
from itertools import permutations

# Get all permutations of [1, 2, 3]
perm = permutations([1, 2, 3])

# Get all permutations of length 2
perm = permutations([1, 2, 3], 2)

# Backtracking
def permute(nums):
    res, partial = [], []
    def backtrack(nums):
        if not nums:
            res.append(partial[:])
            return
        for i in range(len(nums)):
```

[3] https://github.com/lukepereira/notebooks

```
            partial.append(nums[i])
            backtrack(nums[:i] + nums[i+1:])
            partial.pop()
    backtrack(nums)
    return res
```

3.11.2 Combinations

1. Order of items doesn't matter.
2. Counts do not include duplication or removals of items.
3. Collection of counts could be stored in *sets*.

$$C(n,k) = \binom{n}{k} = \frac{P(n,k)}{k!} = \frac{n!}{k!(n-k)!} = \binom{n}{n-k}$$

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k+1}$$

Python Implementation

```
from itertools import combinations, combinations_with_replacement

# Get all combinations of length 2
comb = combinations([1, 2, 3], 2)

# Get all combinations with an element-to-itself combinations included
comb = combinations_with_replacement([1, 2, 3])

import math

# Count n choose k combinations (Python v3.8)
n, k  = 10, 3
math.comb(n, k)

# < Python v3.8
math.factorial(n) / (math.factorial(k) * math.factorial(n-k)
```

3.11.3 Cartesian Product

The Cartesian product of two sets A and B, is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$.

```
import itertools
A = [1, 2, 3]
B = [4, 5, 6]

# product of two iterables taking one element from each
## Using nested for loop
for a in A:
    for b in B:
        results.append((a, b))

```

```
## Using itertools module
list(itertools.product(A, B))

# product of iterable with itself of size 2
list(itertools.product(A, repeat=2))
```

3.11.4 n-th Partial Sum

This counting formula can be used to count the number of contiguous substrings in a string or contiguous subarrays in an array.

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Python Implementation

```
n = 10
nth_partial_sum = (n * (n + 1)) / 2
iterative_sum_count = sum([i for i in range(1, n + 1)])
contiguous_sublists = lambda arr: [
    arr[m: n + 1]
    for m in range(len(arr))
    for n in range(m, len(arr))
]
contiguous_sublist_count = len(contiguous_sublists([0] * n ))

assert iterative_sum_count == nth_partial_sum == contiguous_sublist_count
```

3.11.5 Derangement

A derangement is a permutation of the elements of a set such that no element appears in its original position. In other words, a derangement is a permutation that has no fixed points.

$!n$ (n subfactorial) is the number of derangements – n-permutations where all of the n elements change their initial places.

$$!n = (n-1)(!(n-1) + !(n-2))$$

$$!n = n! \sum_{i=0}^{n} \frac{(-1)^i}{i!}$$

Python Implementation

```
import random

def random_derangement(arr):
    while True:
        for j in range(len(arr) - 1, -1, -1):
            p = random.randint(0, j)
            if arr[p] == j:
                break
            else:
                arr[j], arr[p] = arr[p], arr[j]
        else:
```

```
            if arr[0] != 0:
                return arr
```

3.11.6 Fibonacci Numbers

A recursively defined sequence used to derive the golden ratio and other naturally occurring patterns and sequences.

$$F_0 = 0,\ F_1 = 1 \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for } n > 1.$$

The first few Fibonacci numbers are: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

Python Implementation

```
# Fibonacci series using Dynamic Programming
def fibonacci(n):
    if n <= 1: return n
    dp = [1, 2]
    for i in range(3, n+1):
        curr = dp[0] + dp[1]
        dp[0], dp[1] = dp[1], curr
    return dp[1]
```

3.11.7 Lattice Paths

A sequence of ordered pairs $(m_1, n_1), (m_2, n_2), \cdots, (m_t, n_t)$ such that a coordinate moves one unit either horizontally or vertically from its previous coordinate, i.e.:

1. $m_{i+1} = m_i + 1$ and $n_{i+1} = n_i$
2. $m_{i+1} = m_i$ and $n_{i+1} = n_i + 1$.

The construction of lattice paths forms a bijection with X-strings where $X = \{H, V\}$ with H, V encoding horizontal or vertical moves on a grid. The number of lattice paths from (m_1, n_1) to (m_2, n_2) is,

$$\binom{m_2 - m_1 + m_2 - m_1}{m_2 - m_1}.$$

Python Implementation

```
import math

def unique_paths(m, n):
    if not m or not n: return 0
    numerator = math.factorial(m + n - 2)
    denominator = (math.factorial(n - 1) * math.factorial(m - 1))
    return numerator / denominator
```

3.11.8 Catalan Numbers

Catalan numbers are a sequence of natural numbers that occur in various counting problems,

1. The number of lattice paths from $(0, 0)$ to (n, n) that do not go above the diagonal line $y = x$ (a.k.a Dyck paths).
2. Forming a bijection with up/down movements as characters X, Y, shows that C_n counts the number of Dyck words.
3. The number of valid arrangement of n pairs of opening and closing parenthesis.
4. Re-interpreting the parenthesis as binary operators, i.e. associative multiplication orders, can count the number of possible orderings.
5. We may again re-interpret the binary operations count as being equivalent to the number of unique rooted full binary tree structures with n + 1 leaves.
6. The number of ways a convex polygon of $n+2$ sides can split into triangles by connecting vertices.

Catalan number from binomial coefficients,

$$C(n) = \frac{1}{n+1}\binom{2n}{n}.$$

Catalan number from recursive definition,

$$C_0 = 1, \quad C_{n+1} = \frac{2(2n+1)}{n+2} \cdot C_n.$$

The first few Catalan numbers are: $1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$

The time complexity of brute-force computation of a Catalan number is $\mathcal{O}(3^n)$ which can be reduced to $\mathcal{O}(2^n)$ if recursive calls are memoized.

Python Implementation

```
import math
def catalan_direct(n):
    first_term = 1 / (n + 1)
    second_term = math.comb(2*n, n) # in python 3.8 and above
    second_term = math.factorial(2*n) / (math.factorial(n) * math.factorial(2*n-
    n))
    return int(first_term * second_term)

def catalan_recursive(n):
    C = 1
    for i in range(0, n):
        C = C * 2*(2*i+1)/(i+2)
    return int(C)

def catalan_dp(n):
    if n <= 1: return 1
    dp = [[0] * (n) for _ in range(n)]
    for i in range(n):
        dp[0][i] = 1
    for i in range(1, n):
```

```
        for j in range(1, n):
          if i <= j:
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[-1][-1]
```

3.11.9 Stars and Bars

The number of ways to put n identical objects into k labeled boxes is,

$$\binom{n+k-1}{n}.$$

We can use this for various counting problems, i.e the number of non-negative integer sums, the number of lower-bound integer sums, etc.

Python Implementation

```
import itertools

def stars_and_bars(n, k):
    for c in itertools.combinations(range(n + k - 1), k - 1):
        yield [b - a - 1 for a, b in zip((-1,) + c, c + ( n + k - 1,))]
```

3.12 Geometric Problems

3.12.1 K Nearest Neighbors

3.12.2 Convex Hulls

3.12.3 Count Rectangles

3.12.4 Area of Histograms

3.12.5 Newton's Method

4 Appendix

4.1 Powers of 2 Table

Power of 2	Exact Value (X)	Approx. Value	X Bytes into MB, GB, etc.
7	128		
8	256		
10	1024	1 thousand	1 KB
16	65,536		64 KB
20	1,048,576	1 million	1 MB
30	1,073,741,824	1 billion	1 GB
32	4,294,967,296		4 GB
40	1,099,511,627,776	1 trillion	1 TB

4.2 Array Sorting Algorithms Table

Array Sorting Algorithms

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

4.3 Single-Source Shortest Path Table

Graph type	Algorithm	Time Complexity	Sparse $E = \mathcal{O}(V)$	Dense $E = \mathcal{O}(V^2)$
Unweighted	BFS	$\mathcal{O}(V + E)$	$\mathcal{O}(V)$	$\mathcal{O}(V^2)$
Positive Weights	Dijkstra	$\mathcal{O}(V \log V + E)$	$\mathcal{O}(V \log V)$	$\mathcal{O}(V^2)$
General Weights	Bellman-Ford	$\mathcal{O}(VE)$	$\mathcal{O}(V^2)$	$\mathcal{O}(V^2)$
DAGs	Top. Sort + B-F	$\mathcal{O}(V + E)$	$\mathcal{O}(V)$	$\mathcal{O}(V^2)$

4.4 Algorithm Optimization Checklist

1. Consider making a time-space trade off, usually in the form of a hash-table or cached results.
2. Data Structure Brainstorm. Linked List, Stack, Queue, Priority Queue, Heap, Dictionary, Set, Binary tree, Graph, etc.
3. Consider Best Conceivable Runtime (BCR). Try to derive an approach from an ideal upper bound on the solution.
4. Simplify and Generalize. Simplify problem statement then attempt to generalize solution to original problem.
5. Look for BUD (bottlenecks, unnecessary work, duplicated work).
6. DIY (Do It Yourself). Design an algorithm around how you would solve an analogous real-word scenario without programming.

4.5 Whiteboard Interview Checklist

1. **Restate and reduce problem**
   (a) Carefully read the problem. Spend some time simplifying and re-stating problem in your own words. This helps solidify your understanding, plus translating the problem into its most essential form may reveal possible reductions.
   (b) Clarify requirements, constraints on input and any other assumptions you may need or want to make. For example, what is the format of the input and output, can the function receive invalid inputs, how large will the input be, etc.
   (c) Can you pre-process the input or re-interpret the desired output to simplify or reduce the problem to a simpler or familiar one? This may involve sorting a sequence, converting structured data into a graph, generating a prefix sum array, etc.
   (d) What are problematic or challenging areas that might arise from certain inputs? For example, an input that causes a naive approach to traverse the decision tree toward an incorrect solution. These inputs can be converted into sufficiently complex test cases later.
   (e) What is the best conceivable run time? If it's not yet obvious, this can be examined later during the optimization phase.
   (f) What are your intuitions about possible solutions? You can revisit these ideas later.

2. **State brute-force solution**
   (a) Give an overview of the approach.
   (b) Find the time and space complexity.
   (c) If you don't have any immediate ideas for an optimized solution, spend time elaborating on the brute force algorithm, otherwise mention that we can do better and can move on.
3. **Optimize previous approach or introduce new, better approach**
   (a) Brainstorm using Algorithm Optimization Checklist. Avoid getting stuck on memory recall for too long, even if you recognize the problem.
   (b) Give an overview of the approach. Find the time and space complexity.
   (c) Repeat or expand. Always spend extra time considering alternative approaches before implementing a solution.
   (d) When out of ideas or if able to match the best conceivable run time with low/linear space complexity, prompt the interviewer for approval: "If you're happy with this approach, I can go into the finer details and begin implemention."
4. **Consider granular implementation details**
   (a) Describe edge cases, i.e. empty input, invalid inputs, large inputs.
   (b) Consider boundary conditions if dealing with iterations, array pointers, or ranges.
   (c) Describe sufficiently complex test case if none are given, consider problem areas.
   (d) Consider minor optimizations to general approach, i.e. short-circuiting, more performant data structures, caching.
   (e) If using recursion, note the limitations of relying on the call stack. Default limit in Python is 1000.
5. **Implement**
   (a) Handle base/empty cases, i.e. empty inputs, invalid input error checks.
   (b) Use descriptive variable and function names.
   (c) Add inline comments when necessary.
   (d) Do not repeat yourself (DRY).
   (e) Use modular code when possible.
   (f) Follow coding principles (Correct, Efficient, Simple, Readable, Maintainable).
6. **Validate and test with code walkthrough**
   (a) Scan over code and double check for any syntax errors.
   (b) Walk through code for correctness using sufficiently complex test cases. Ensure boundary conditions don't cause errors.
   (c) Clean up code if possible.

(d) Consider further optimizations if time complexity is not BCR and space complexity is not constant.

References

[1] Steven S. Skiena. 2008. The Algorithm Design Manual (2nd. ed.). Springer Publishing Company, Incorporated.

[2] Erik Demaine, Srini Devadas. Introduction to Algorithms. Fall 2011. Massachusetts Institute of Technology: MIT OpenCouseWare, https://ocw.mit.edu/. License: Creative Commons

[3] David Liu, Data Structures and Analysis: Lecture Notes for CSC263, Department of Computer Science, University of Toronto

[4] McDowell, Gayle Laakmann, Cracking The Coding Interview: 150 Programming Questions and Solutions. Palo Alto, CA :CareerCup, LLC, 2011.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.

[6] Keller, M.T. and Trotter, W.T., Applied Combinatorics, Open Textbook Library, ISBN9781534878655