Data Structures and Algorithms

Contents

1 Pre-requisite Concepts

1.1 Asymptotic Runtime Analysis

1.1.1 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, worse 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 X insertions, this happens every $2, 4, 8, ..., X$ insertions.

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

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,

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

When using Python's standard library sort on an array, we can assume the running time will be $\mathcal{O}(n \log n)$. See section on Timsort for further details.

1.2 Computational Complexity

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. This means that when an answer is YES, it can be proved and a polynomial-time algorithm can verify the proof.

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 sufficiently efficient reduction from one problem to another may 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 which 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 Space–time tradeoff

A space–time 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; 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).

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 (head) of the list
- **tail(L)**: refer to the list consisting of all the components of a list except for its first (tail).

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.

2.1.3 Dynamic Arrays

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.

2.1.4 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.

Python Implementation

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

2.1.5 Self-Organizing Lists

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.

2.1.6 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 $\mathcal{O}(\log n)$ query time. The primary benefits of skip lists 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 removes as 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.

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.

Python Implementation

```
# Using a list
q = []
for item in data:
    q.append(item)

while len(q):
    next_item = queue.pop(0)
    print(next_item)

# The collections.deque module
from collections import deque

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

while len(q):
    next_item = q.popleft()
    print(next_item)
```

2.2.3 Priority Queue

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 such arrival. 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. 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.

2. 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
3. Bounded height priority queue
4. 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.
5. Fibonacci and pairing heaps. These complicated priority queues 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.

Python Implementation

```python
# The queue module
from Queue import PriorityQueue

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

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

# The heapq module
import heapq

q = []
for item in data:
    heapq.heappush(q, (item.priority, item))

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

2.3 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 also has zero or more child nodes and can be recursively defined as a root node of a sub-tree. 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.

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. 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 **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 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 array. For a node corresponding to index i, its left child is stored at index $2i$, and its right child is stored at index $2i + 1$. Going backwards, we can also deduce that the parent of index i (when $i > 1$) is stored at index $\lceil i/2 \rceil$.

Python Implementation

```
# Generic tree
from collections import defaultdict

## a tree is a dict whose default values are trees.
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
```

2.3.1 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 holds for a **max-heap**. We have two key 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 minheap 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.

Python Implementation

```
# Using heapq module
import heapq

listForTree = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]

## min heap
```

```
heapq.heapify(listForTree)
heapq.heappop(minheap)

class MinHeap(object):
  def __init__(self): self.h = []
  def heappush(self, x): heapq.heappush(self.h, x)
  def heappop(self): return heapq.heappop(self.h)
  def __getitem__(self, i): return self.h[i]
  def __len__(self): return len(self.h)

## max heap
heapq._heapify_max(listForTree)
heapq._heappop_max(maxheap)

class MaxHeapObj(object):
  def __init__(self, val): self.val = val
  def __lt__(self, other):
    """Invert comparison logic"""
    return self.val > other.val

class MaxHeap(MinHeap):
  def heappush(self, x): heapq.heappush(self.h, MaxHeapObj(x))
  def heappop(self): return heapq.heappop(self.h).val
  def __getitem__(self, i): return self.h[i].val


# 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)
```

2.3.2 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 (such as a TerminatingTrieNode, which inherits from TrieNode). Or, we could use just a boolean flag terminates within the "parent" node. A node in a trie could have anywhere from 1 through ALPHABET_SIZE + 1 children (or, 0 through ALPHABET_SIZE if a boolean flag is used instead of a * node).

Very commonly, a trie is used to store the entire (English) language for quick prefix lookups. A trie can check if a string is a valid prefix in $\mathcal{O}(K)$ time, where K is the length of the string. Many problems involving lists of valid words leverage a trie as an optimization.

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. 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.
- Longest substring common to a set of strings
- Find the longest palindrome in S

Python Implementation

```
import collections

class TrieNode:
    def __init__(self):
        self.word = False
        self.children = {}

class Trie:
    def __init__(self):
        self.root = TrieNode()
```

```
11
12      def insert(self, word):
13          node = self.root
14          for char in word:
15              if char not in node.children:
16                  node.children[char] = TrieNode()
17              node = node.children[char]
18          node.word = True
19
20      def search(self, word):
21          node = self.root
22          for char in word:
23              if char not in node.children:
24                  return False
25              node = node.children[char]
26          return node.word
27
28      def startsWith(self, prefix):
29          node = self.root
30          for char in prefix:
31              if char not in node.children:
32                  return False
33              node = node.children[char]
34          return True
```

2.3.3 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. Once constructed, several operations can be performed quickly, i.e. locating a substring in S, locating a substring if a certain number of mistakes or edits are allowed, locating matches for a regular expression pattern etc. Suffix trees also provide one of the first linear-time solutions for the longest common substring problem. Though these speedups come at a cost: storing a string's suffix tree typically requires significantly more space than storing the string itself.

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):
    """
    l: iterable of keys
    returns: a list with integer keys
    """
    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
    O(n * log(n)^2)
    """
    n = len(s)
    k = 1
    line = to_int_keys(s)
    while max(line) < n - 1:
        line = to_int_keys_best(
            [a * (n + 1) + b + 1
             for (a, b) in
             zip_longest(line, islice(line, k, None),
                         fillvalue=-1)])
        k <<= 1
    return line
```

2.3.4 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

# 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)
```

2.3.5 Kd-Trees

Kd-trees and related spatial data structures hierarchically decompose 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.

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. 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.

2.4 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

2.4.1 AVL Tree

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 which 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. 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

2.4.2 Red–black Tree

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.

2.4.3 B-tree

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, so that we can make the equivalent of several search steps before another disk access is needed. With B-trees we can access enormous numbers of keys using only a few disk accesses. To get the full benefit from using a B-tree, it is important to understand how the secondary storage device and virtual memory interact, through constants such as page size and virtual/real address space. Cache-oblivious algorithms can mitigate such concerns.

A B-tree of order m is a tree which satisfies the following properties:

1. The root has at least two children if it is not a leaf node.
2. Every non-leaf node (except the root) has at least $\lceil \frac{m}{2} \rceil$ child nodes.
3. All leaves appear in the same level and carry no information.
4. Every node has at most m children.
5. A non-leaf node with k children contains $k - 1$ keys.

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 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.

2.5 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.

There are two common ways to represent a graph: **adjacency lists** and **adjacency matrices**. Every vertex (or node) 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 matrix is an NxN boolean matrix (where N is the number of nodes), where a true value at matrix [i] [j] indicates an edge from node i to node j. (You can also use an integer matrix with Os and 1 s.) In an undirected graph, an adjacency matrix will be symmetric. In a directed graph, it will not (necessarily) be.

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.

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.

Hypergraphs are generalized graphs where each edge may link subsets of more than two vertices. In contrast, in an ordinary graph, an edge connects exactly two vertices. Two basic data structures for hypergraphs are: Incidence matrices, which are analogous to adjacency matrices, and Bipartite incidence structures, which are analogous to adjacency lists

Python Implementation

```
# Directed graph
class Graph(object):
    def __init__(self):
        self.nodes = set()
        self.edges = defaultdict(list)
        self.distances = {}

    def add_node(self, value):
```

```
        self.nodes.add(value)

    def add_edge(self, from_node, to_node, distance):
        self.edges[from_node].append(to_node)
        self.edges[to_node].append(from_node)
        self.distances[(from_node, to_node)] = distance

# Undirected graph using Adjacency Matrix
class Vertex:
    def __init__(self, n):
        self.name = n

class Graph:
    vertices = {}
    edges = []
    edge_indices = {}

    def add_vertex(self, vertex):
        if isinstance(vertex, Vertex) and vertex.name not in self.vertices:
            self.vertices[vertex.name] = vertex
            for row in self.edges:
                row.append(0)
            self.edges.append([0] * (len(self.edges)+1))
            self.edge_indices[vertex.name] = len(self.edge_indices)
            return True
        else:
            return False

    def add_edge(self, u, v, weight=1):
        if u in self.vertices and v in self.vertices:
            self.edges[self.edge_indices[u]][self.edge_indices[v]] = weight
            self.edges[self.edge_indices[v]][self.edge_indices[u]] = weight
            return True
        else:
            return False

    def print_graph(self):
        for v, i in sorted(self.edge_indices.items()):
            print(v + ' ', end='')
            for j in range(len(self.edges)):
                print(self.edges[i][j], end='')
            print(' ')

# Undirected graph using Adjacency Lists
class Vertex:
    def __init__(self, n):
        self.name = n
        self.neighbors = list()

    def add_neighbor(self, v):
        if v not in self.neighbors:
            self.neighbors.append(v)
            self.neighbors.sort()

class Graph:
    vertices = {}

    def add_vertex(self, vertex):
        if isinstance(vertex, Vertex) and vertex.name not in self.vertices:
            self.vertices[vertex.name] = vertex
            return True
        else:
```

```
71              return False
72
73      def add_edge(self, u, v):
74          if u in self.vertices and v in self.vertices:
75              self.vertices[u].add_neighbor(v)
76              self.vertices[v].add_neighbor(u)
77              return True
78          else:
79              return False
80
81      def print_graph(self):
82          for key in sorted(list(self.vertices.keys())):
83              print(key + str(self.vertices[key].neighbors))
```

2.6 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. First, we will describe a simple but common implementation 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 ints.
2. Then, map the hash code to an index in the array. This could be done with something like $hash(key) \mod array\_length$. Two different hash codes could map to the same index.
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 require each array element to contain only one key, but to 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 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 a form of clustering occurs where if many items have the same initial hash value, 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 $\mathcal{O}(m)$.

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

2.6.1 Dictionaries

The dictionary data type permits access to data items based on its content. You insert an item into a dictionary so you can retrieve it when you need it. The primary operations a dictionary 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.

2.6.2 Sets

In mathematical terms, a set is an unordered collection of unique objects drawn from a fixed universal set. 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 possible element searching in sublinear time. Finally, printing the elements of a set in a canonical order paradoxically reminds us that order really 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.

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

Although sets are commonly implemented with hashtables, other data structures that can be used:

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.

3 Algorithms and Techniques

3.1 String and Array Sorting

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 repeat this process until we either find x or the subarray has size 0.

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
```

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]
```

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

```
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:
            continue
        A[i], A[min_index] = A[min_index], A[i]
```

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.

Python Implementation

```
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.

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

Python Implementation

```
def _merge_lists(left_sublist, right_sublist):
    i, j = 0, 0
    result = []
    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 = int(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 become 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 0 (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, $E[T_A] = \Theta(n \log n)$.

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
```

3.1.7 Heap Sort

Time Complexity: $\mathcal{O}(n^2)$ 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 max heap from an unordered array A in $\mathcal{O}(n)$.
2. Find the max element A[1] in $\mathcal{O}(1)$.
3. Swap elements A[n] with A[1] so that the max element is at the end of the array in 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 max heap principle but the children won't. This allows us to run heapify in $\mathcal{O}(\log n)$.

Python Implementation

```python
import heapq

def heapsort(A):
    h = []
    for value in A:
        heapq.heappush(h, value)
    return [heapq.heappop(h) for i in range(len(h))]
```

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 direct use in situations where the variation in keys is not significantly greater than the number of items. However, it is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.

Python Implementation

```
def count_sort(arr):
    output = [0 for i in range(256)]
    count = [0 for i in range(256)]
    ans = ["" for _ in arr]

    for i in arr:
        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
    for i in range(len(arr)):
        output[count[ord(arr[i])]-1] = arr[i]
        count[ord(arr[i])] -= 1

    # Copy the output array to arr, so that arr now
    # contains sorted characters
    for i in range(len(arr)):
        ans[i] = output[i]
    return ans
```

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)

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. Then you would start by separating the numbers by units and then putting them together again; next you would separate the numbers by tens and then put them together again; then by hundreds and so on until all the numbers are sorted. Each time you loop, just read the list from left to right. You can also imagine you are separating the numbers into buckets.

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. It's also of note that $\mathcal{O}(f(n))$ really means in order of $K * f(n)$, where K is some arbitrary constant. For radix sort this K happens to be quite large (at least on the order of number of bits in the integers sorted), on the other hand quicksort has one of the lowest K among all sorting algorithms and average complexity of $n \cdot \log(n)$. Thus in real life scenario quicksort will often be faster than radix sort.

Python Implementation

```
def radix_sort(A):
    RADIX = 10
    maxLength = False
    tmp, placement = -1, 1

    while not maxLength:
        maxLength = True
        # declare and initialize buckets
        buckets = [list() for _ in range(RADIX)]

        # split A between lists
        for i in A:
            tmp = i / placement
            buckets[tmp % RADIX].append(i)
            if maxLength and tmp > 0:
                maxLength = False

        # empty lists into A array
        a = 0
        for b in range(RADIX):
            buck = buckets[b]
            for i in buck:
                A[a] = i
                a += 1

        # move to next digit
        placement *= RADIX
```

3.1.10 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 left_right_boundary(self, seq):
    left, right = 0, len(seq) - 1
    while left < right:
        if self.left_condition(left):
            left += 1

        if self.right_condition(right):
            right -= 1

        self.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 pattern (a.k.a. Floyd's Tortoise and Hare Algorithm) is very useful when dealing with cyclic linked lists or arrays. By moving at different speeds, the algorithm proves that the two pointers are going to meet eventually. 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.

Lets say the loop is of length $m \leq n$: 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 $\mathcal{O}(n)$.

Python Implementation

```
def has_cycle(self, 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
```

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

1. Use two pointers, start and end, to represent a window.
2. Move end to find a valid window.
3. When a valid window is found, move start to find a smaller window.

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 may also use a counter dictionary to maintain current state.

The characteristics of problems that can be solved using the two pointer technique:

1. If the narrow scope of your window is valid, then the wider scope is valid. By definition of the narrow window being a sub problem of the wider problem.
2. If the wider scope of the window is invalid, then the narrow scope is also invalid. By definition of the wider window being a full problem of the given sub problem.
3. If the narrow scope of your window is invalid, it doesn't always mean the wider scope is invalid. You need to increment the pointers to find out (this is why the 2 pointer solution works, after all). In a given narrow window, you cannot guarantee that this window means the solution cannot be found, because the narrow window is just a slice of a given string or subarray. Then, if all sub-problems are solved and the wider problem is also invalid, then we can say that all sub problems or sub strings cannot be formed to solve the problem.

Python Implementation

```python
def basic_sliding_window(self, seq):
    start, end = 0, 0
    while end < len(seq):
        # end grows in the outer loop
        end += 1
        # start grows with some restrict
        while self.start_condition(start):
            # process logic before pointers movement
            self.process_logic1(start, end)
            # start grows in the inner loop
            start += 1
        # or process logic after pointers movement
        self.process_logic2(start, end)

# Using a counter
from collections import Counter

# s - target sequence, p - pattern or restrict sequence
def sliding_window_with_counter(s, p):
    counter = Counter(p)
    start, end, count, res = 0, 0, 0, 0
    while end < len(s):
        counter[s[end]] += 1
        if counter[s[end]] > 1:
            count += 1
        end += 1
        while count > 0:
            # update res here if finding minimum
            # increase start to make it invalid or valid again
            counter[s[start]] -= 1
            # update count based on some condition
            if counter[s[start]] > 0:
```

```
                count -= 1
            start += 1
        # update res here if finding maximum
        res = max(res, end - start)
    return res
```

3.2.4 Single-pass with Lookup Table

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

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 current value to the lookup table with its index.

Python Implementation

```
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 Range Operations on Array

Time Complexity: $\mathcal{O}(Q+n)$, Space Complexity: $\mathcal{O}(1)$, where Q is the number of operations/-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 brute force solution of performing 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 techinque which will give us an algorithm that runs in $\mathcal{O}(n)$. It does this by storing the difference between the current element and the previous element. This means that we add the sum to $A[i]$ showing that $A[i]$ is greater than its previous element by the sum. We subtract sum from $A[j+1]$ to show that $A[j+1]$ is less than $A[j]$ by sum (since $A[j]$ was the last element that was added to sum). By the end of all this, we have an array that shows the difference between every successive element. By adding all the positive differences, we get the value of the maximum element

Python Implementation

```
def arrayOperations(n, operations):
    B = [0] * (n + 1)
    for start, end, incr in operations:
        B[start - 1] += incr
        if stop <= len(B):
            B[stop] -= incr;
    max = cur = 0
    for i in B:
        cur = cur + i;
        if max < cur:
            max = cur;
    return max
```

3.2.6 Kadane's Algorithm

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

The maximum sum subarray 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 algorithmic 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.

1. Use the input vector 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.

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)
```

3.3 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.

3.3.1 KMP Pattern Matching

Time Complexity: $\mathcal{O}(n)$ 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 to $\mathcal{O}(n)$. The basic idea behind KMP's algorithm is that whenever we detect a mismatch after some matches, we know some of the characters in the text of the next window.

Instead of wasting computation on previous matches, we use a precomputed lookup table to skip to the first instance of a match of the starting character.

Python Implementation

```
class KMP:
    def partial(self, pattern):
        """ Calculate partial match table: String -> [Int]"""
        ret = [0]
        for i in range(1, len(pattern)):
            j = ret[i - 1]
            while j > 0 and pattern[j] != pattern[i]:
                j = ret[j - 1]
            ret.append(j + 1 if pattern[j] == pattern[i] else j)
        return ret

    def search(self, T, P):
        """
        KMP search main algorithm: String -> String -> [Int]
        Return all the matching position of pattern string P in T
        """
        partial, ret, j = self.partial(P), [], 0

        for i in range(len(T)):
            while j > 0 and T[i] != P[j]:
                j = partial[j - 1]
            if T[i] == P[j]: j += 1
            if j == len(P):
                ret.append(i - (j - 1))
                j = partial[j - 1]

        return ret
```

3.3.2 Rabin–Karp

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.

3.3.3 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.

Python Implementation

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

    intervals.sort(key=lambda x: x[0])
    merged = []
    start = intervals[0][0]
    end = intervals[0][1]

    for i in range(1, len(intervals)):
        interval = intervals[i]
        if interval[0] <= end:  # overlapping intervals
            end = max(interval[1], end)
        else:  # non-overlapping interval, add the previous interval and reset
            merged.append([start, end])
            start = interval[0]
            end = interval[1]

    merged.append([start, end])  # add the last interval
    return merged
```

3.4 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.

Python Implementation

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

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

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

3.5 Heap Use Cases

3.5.1 Top K Numbers

The best data structure to keep track of top K elements is a heap. If we iterate through an array, one element at a time, and keep 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. We may use a min-heap for this.

Python Implementation

```python
import heapq

# For maintaining a class
class KthLargestHeap:
    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:
        min_heap = []
        for i in range(k):
            heapq.heappush(min_heap, nums[i])
        for i in range(k, len(nums)):
            if nums[i] > min_heap[0]:
                heapq.heappop(min_heap)
                heapq.heappush(min_heap, nums[i])
        return min_heap[0]
```

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

```python
from heapq import *

```

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

    def addNum(self, num):
        small, large = self.heaps
        #convert a min heap to a max heap. heapq only has 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 Graph Traversal

3.6.1 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.

In BFS, node x visits each of x'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.

Python Implementation

```python
from collections import deque

def bfs(matrix):
    # Check for an empty graph.
    if not matrix:
        return []

    rows, cols = len(matrix), len(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()
            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))
```

```
25
26      for i in range(rows):
27          for j in range(cols):
28              traverse(i, j)
```

3.6.2 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.

Python Implementation

```
1  def dfs(matrix):
2      # Check for an empty graph.
3      if not matrix:
4          return []
5
6      rows, cols = len(matrix), len(matrix[0])
7      visited = set()
8      directions = ((0, 1), (0, -1), (1, 0), (-1, 0))
9
10     def traverse(i, j):
11         if (i, j) in visited:
12             return
13
14         visited.add((i, j))
15         # Traverse neighbors.
16         for direction in directions:
17             next_i, next_j = i + direction[0], j + direction[1]
18             if 0 <= next_i < rows and 0 <= next_j < cols:
19                 # Add in your question-specific checks.
20                 traverse(next_i, next_j)
21
22     for i in range(rows):
23         for j in range(cols):
24             traverse(i, j)
```

3.6.3 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. Then, in a traditional breadth-first search we visit $\mathcal{O}(k^d)$ nodes while bidirectional search visits $\mathcal{O}(k^{d/2})$

3.6.4 Dijkstra's algorithm

Time Complexity: $\Theta(|E| + |V| \log |V|)$ worst case

An algorithm for finding the shortest paths between nodes in a graph. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. Let the node at which we are starting be called the initial node and the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

Djikstra's algorithm prioritizes shorter distances and edges with low weights in its path discovery process.

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbour B has length 2, then the distance to B through A will be $6 + 2 = 8$. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, the current value will be kept.
4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

The code for maintaining the visited set can be simplified by using a priority queue.

Python Implementation

```
from collections import defaultdict
from heapq import *

def dijkstra(edges, f, t):
    g = defaultdict(list)
    for l,r,c in edges:
        g[l].append((c,r))

    q, seen, mins = [(0,f,())], set(), {f: 0}
    while q:
        (cost,v1,path) = heappop(q)
        if v1 not in seen:
            seen.add(v1)
            path = (v1, path)
            if v1 == t: return (cost, path)
```

```
16
17              for c, v2 in g.get(v1, ()):
18                  if v2 in seen: continue
19                  prev = mins.get(v2, None)
20                  next = cost + c
21                  if prev is None or next < prev:
22                      mins[v2] = next
23                      heappush(q, (next, v2, path))
24
25      return float("inf")
```

3.6.5 A*

A* (pronounced "A-star") is a graph traversal and path search algorithm. It is a minor extension of Djikstra'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 Graph Analysis Methods

3.7.1 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. A topological ordering is possible if and only 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.

An 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 which 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. Since each edge and node is visited once, the algorithm runs in linear time.

Python Implementation

```python
from collections import deque
GRAY, BLACK = 0, 1

def topological_sort(graph):
    order, enter, state = deque(), set(graph), {}

    def dfs(node):
        state[node] = GRAY
        for k in graph.get(node, ()):
            sk = state.get(k, None)
            if sk == GRAY:
                raise ValueError("cycle")
            if sk == BLACK:
                continue
            enter.discard(k)
            dfs(k)
        order.appendleft(node)
        state[node] = BLACK

    while enter:
        dfs(enter.pop())
    return order
```

3.8 Recursive Problems

Although there is some overlap, recursive solutions can be thought of in terms of the following categories,

- Iteration – instead of iterating with a for loop, a recursive call stack is used to iterate over an array/list. This often simplifies the code.
- Sub-problems – A catch-all for generic and classic recursion problems in which a solution to the larger problem can be derived from solutions of the sub-problems.
- Selection – Problems that can be solved by finding all valid combinations of a set of inputs. A brute-force approach that finds all possible combinations and then validates based on the requirements of problem statement. Can be further optimized to continuously validate at every decision step, dropping invalid branches of combinations using backtracking or caching relevant computations using dynamic programming.
- Ordering (Permutations) – Similar to selection but the ordering of the selections matters.
- Divide and Conquer – Similar to sub-problems, but instead of solving for n sub-problems, we split the input into two halves and recursively validate the sub-problem dropping one half at each recursion. Commonly found in searching and sorting problems.
- Depth-first search – A common technique used in tree/graph structures, where a recursive call is used instead of using stack ADT.

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.

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.

All recursive algorithms can be implemented iteratively, although sometimes the code to do so is much more complex. Drawing the recursive calls as a tree is a useful way to figure out the runtime of a recursive algorithm.

3.8.1 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 at what they've done to see if they could optimise globally. This is the main difference between Greedy and Dynamic Programming.

Even though a greedy algorithm follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum, there are cases where locally optimal solutions or maxima are not the global optimal solution which will cause the algorithm to product incorrect solutions. Nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Greedy algorithms are only ideal for problems which have optimal substructure. A problem is said to have **optimal substructure** if an optimal solution can be constructed efficiently from optimal solutions of its subproblem. 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.

3.8.2 Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate 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.

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?

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid solution.

The difference between backtracking and depth-first search is that backtracking traverses in the solution space whereas DFS traverses in data structure space, pruned of DFS.

Python Implementation

```
def backtracking(self, data, candidate):
    # pruning
    if self.reject(data, candidate):
        return

    # reach the end
    if self.accept(data, candidate):
        return self.output(data, candidate)

    # drill down
    for cur_candidate in self.all_extension(data, candidate):
        # or you can choose to prune here, recursion depth - 1
        if not self.should_to_be_pruned(cur_candidate):
            self.backtracking(data, cur_candidate)
```

3.8.3 Dynamic Programming & Memoization

Dynamic programming (DP) is a general, powerful algorithm design technique. It is mostly just a matter of taking a recursive algorithm and finding the overlapping subproblems (that is, the repeated calls). You then cache those results for future recursive calls. Alternatively, you can study the pattern of the recursive calls and implement something iterative. You still cache previous work. A dynamical 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.

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. 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.

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 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.

A memoized function only recurses the first time it's called with the memoized call costing $\Theta(1)$. 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.

1. Define subproblems
2. Guess (part of the solution)
3. Relate subproblem solutions (with a recurrence)
4. Construct an algorithm by recursion and memoization (need acyclic DAG) or building a DP table bottom up (need topological order)
5. Solve original problem. Time = number of subproblems multiplied by running time/ subproblem

Defining subproblems for strings, sequences:

- suffixes x[i:] for all i. $\mathcal{O}(n)$
- prefixes x[:i] for all i. $\mathcal{O}(n)$
- substrings x[i:j] for all $i \leq j$ $\mathcal{O}(n^2)$

Topological order, i.e. order in which subproblems are executed, should be from smallest to largest.

Two kinds of guessing:

- Which subproblems to use to solve bigger subproblem.
- Add more subproblems to guess, remember more features of the solution variations.

Python Implementation

```
# It takes n steps to reach to the top of a set of stairs. Each time you can
    either climb 1 or 2 steps. In how many distinct ways can you climb to the
    top?

def climbStairs(n):
        dp = [1, 1]
        for i in range(2, n + 1):
            dp.append(dp[i-1] + dp[i-2])
        return dp[n]

# Compute the fewest number of coins that are needed to sum to an amount

def coinChange(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], -1][dp[-1] == MAX]

```

```python
# Given a knapsack with a maximum weight capacity and a list of items with value and weights, maximize the amount of value we can fit within the knapsacks weight capacity.

def knapsack(capacity, weight, values, n):
    if n == 0 or capacity == 0 :
        return 0
    # If weight is higher than capacity then it is not included
    if (weight[n-1] > capacity):
        return knapsack(capacity, weight, values, n-1)
    # return either nth item being included or not
    else:
        return max(
            values[n-1] + knapsack(capacity-weight[n-1], weight, values, n-1),
            knapsack(capacity, weight, values, n-1)


# Given an unsorted array of integers, find the length of longest increasing subsequence.

def lengthOfLIS(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 longestCommonSubsequence(text1: str, text2: str) -> int:

    ## similar to memoization, in recursive calls the decorator doesn't have to recompute but retrieves from the cache
    @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)
```

3.9 Numerical Problems

3.9.1 Bit Manipulation

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word. Computer programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization. For most other tasks, modern programming languages allow the

programmer to work directly with abstractions instead of bits that represent those abstractions. Source code that does bit manipulation makes use of the bitwise operations: AND, OR, XOR, NOT, and bit shifts.

Bit manipulation, in some cases, can obviate or reduce the need to loop over a data structure and can give many-fold speed ups, as bit manipulations are processed in parallel, but the code can become more difficult to write and maintain.

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

- & (and)
- | (or)
- $\sim$ (not)
- $\wedge$ (exclusive-or, xor)
- a << b, a >> b (shift operators)

The exclusive-or operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-or, with the operator of a caret, performs the exclusive-or operation on each pair of bits. Exclusive-or is commonly abbreviated XOR.

- Set union: $A|B$
- Set intersection: $A \ \& \ B$
- Set subtraction: $A \ \& \sim B$
- Set negation: ALL BITS $\wedge A$ or $\sim A$
- Set bit: $A| = 1 << \text{bit}$
- Clear bit: $A \ \& = \sim (1 << \text{bit})$
- Test bit: $(A\&1 << \text{bit})! = 0$
- Extract last bit: $A \ \& \ -A$ or $A \ \& \sim (A-1)$ or $x \wedge (x \ \& \ (x-1))$
- Remove last bit: $A \ \& \ (A-1)$
- Get all 1-bits: ~ 0

Two's complement is a mathematical operation on binary numbers, and is an example of a radix complement. It is used in computing as a method of signed number representation. 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 $010 + 110 = 1000$. The two's complement is calculated by inverting the digits and adding one.

Python Implementation

```
import operator

# Given an array containing n distinct numbers taken from [0, n], find the one
    that is missing from the array.

def missingNumber(self, nums):
    return reduce(operator.xor, nums + range(len(nums)+1))
```

3.10 Combinatorial Problems

View combinatorics notebook[1] for more details.

3.10.1 Permuations

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)
```

3.10.2 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*.

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

Python Implementation

```
import itertools

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

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

[1] https://github.com/lukepereira/latex-ci

3.10.3 n-th Partial Sum

This counting formula can be used to count the number of contiguous substrings in a 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 l: [
    l[m: n + 1]
    for m in range(len(l))
    for n in range(m, len(l))
]
contiguous_sublist_count = len(contiguous_sublists([0] * n ))

assert iterative_sum_count == nth_partial_sum == contiguous_sublist_count
```

3.10.4 Lattice Paths

A sequence of ordered pairs $(m_1, n_1), (m_2, n_2), \cdots, (m_t, n_t)$ such that either:

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
    return math.factorial(m + n - 2)/(math.factorial(n - 1) * math.factorial(m -
     1))
```

3.10.5 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,))]
```

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