Data Structures & Algorithms Cheat Sheet

Thomas Monson

Contents

1 Essential Patterns

1.1 Backtracking

Backtracking is a strategy for solving *combinatorial search* problems, where the goal is to find groupings or arrangements of elements that satisfy certain conditions. Backtracking searches the space of all possible solution states, and this space can be represented by a tree known as a *state-space tree* or *potential search tree*. The nodes of this tree are *partial candidates*. Each partial candidate is the parent of the candidates that differ from it by a single *extension step*, and the leaves of the tree are the partial candidates that cannot be extended further. The tree is searched in depth-first order.

An efficient backtracking algorithm will prune the subtree rooted at a partial candidate that cannot be extended to a valid solution (i.e. it will not bother exploring such a subtree). At each node c, it will check whether c can be extended to a valid solution. If it cannot, it will prune the subtree rooted at c and backtrack from c. Otherwise, it will check whether c itself is a valid solution, and then it will explore the subtree rooted at c.

A backtracking algorithm that does no pruning is equivalent to a *brute-force* or *exhaustive* search of the state-space. The *actual search tree* of a backtracking algorithm is the pruned version of its potential search tree.

In general, backtracking solutions can be implemented with the following template, where `reject(c)` returns whether `c` can be extended to a valid solution, `accept(c)` returns whether `c` is a valid solution, and `get_steps(c)` returns a list of the elements that can be appended to `c` to extend it to a child partial candidate:

```
result = []
def backtrack(c):
    if reject(c):
        return
    if accept(c):
        result.append(c[:])

    for step in get_steps(c):
        c.append(step)
        backtrack(c)
        c.pop()

backtrack([])
```

Note that this template assumes that a valid solution can be extended to another valid solution. If this is not the case, the `accept` conditional should return after appending `c` to `result` (i.e. when valid solutions must be leaves of the potential or actual search tree).

Sometimes a `reject` conditional is unnecessary because an exhaustive search is required. For example, for a backtracking algorithm that finds all possible combinations of `k` numbers in the range `[1, n]`, valid solutions must be leaves and every leaf is a valid solution:

```python
def combine(n: int, k: int) -> list[list[int]]:
    result = []

    def backtrack(state, start):
        if len(state) == k:
            result.append(state[:])
            return

        need = k - len(state)
        remain = n - start + 1
        available = remain - need

        for step in range(start, start + available + 1):
            state.append(step)
            start += 1
            backtrack(state, start)  # Take
            state.pop()  # Not take

    backtrack([], 1)
    return result
```

1.2 Dynamic Programming

(combinatorial optimization)
`@functools.cache`

1.2.1 Would it help to rephrase the problem in order to more easily define its subproblems?

Given an integer array, return the length of the longest strictly increasing subsequence (LIS).

$\equiv$ Return the length of the LIS of an array `a` of length n.
$\equiv$ Return the length of the LIS of `a[0:n]`.

The LIS of `a` must have some first element. If this is the ith element, then the LIS of `a` is equal to the LIS of `a[i:]`, where `a[i]` is the first element of the sequence.

Let `dp[i]` be the length of the LIS of `a[i:]`, where `a[i]` is the first element of the sequence. Return `max(dp)`.

1.2.2 Is the problem a variation of the *knapsack problem*?

Given a set of N items, each with a weight and a value, determine which items to include in a knapsack such that the total weight is less than or equal to the knapsack's capacity W and the total value is as large as possible. Return the total value.

In the **0-1 knapsack problem**, each item can be taken once or not at all.

```
K(n, w) = max(val[n - 1] + K(n - 1, w - wt[n - 1]),
                           K(n - 1, w))
```

In the **unbounded knapsack problem**, each item can be taken an arbitrary number of times.

```
K(n, w) = max(val[n - 1] + K(n, w - wt[n - 1]),
                           K(n - 1, w))
```

Below are three 0-1 knapsack implementations: memoization, 2D tabulation, and 1D tabulation.

```python
def knapsack_memo(wt: list[int], val: list[int], W: int) -> int:
    N = len(wt)
    dp = [[-1 for _ in range(W + 1)] for _ in range(N + 1)]

    def knapsack(n, w):
        if n == 0 or w == 0:
            return 0

        if dp[n][w] == -1:
            if wt[n - 1] > w:
                dp[n][w] = knapsack(n - 1, w)
            else:
                dp[n][w] = max(
                    val[n - 1] + knapsack(n - 1, w - wt[n - 1]),
                    knapsack(n - 1, w)
                )

        return dp[n][w]

    return knapsack(N, W)
```

```python
def knapsack_tab_2d(wt: list[int], val: list[int], W: int) ->
                                int:
    N = len(wt)
    dp = [[0 for _ in range(W + 1)] for _ in range(N + 1)]

    for n in range(1, N + 1):
        for w in range(1, W + 1):
            if wt[n - 1] > w:
                dp[n][w] = dp[n - 1][w]
            else:
                dp[n][w] = max(
                    val[n - 1] + dp[n - 1][w - wt[n - 1]],
                    dp[n - 1][w]
                )

    return dp[N][W]
```

```python
def knapsack_tab_1d(wt: list[int], val: list[int], W: int) ->
                                int:
    N = len(wt)
    dp = [0] * (W + 1)

    for n in range(1, N + 1):
        for w in range(W, wt[n - 1] - 1, -1):
            dp[w] = max(val[n - 1] + dp[w - wt[n - 1]], dp[w])

    return dp[W]
```

1.3 Sets

1.3.1 Do you need to model the partitioning of a set? That is, given a set of items, do you need to group the items into subsets?

You should use a disjoint-set (union-find) forest (see Appendix B.2).

1.4 Arrays

1.4.1 Would it help to know the sum of elements for any subarray in O(n) time?

Computing the **prefix sum** of an array `a` will give you the sum of elements for subarrays `[a[:i] for i in range(1, len(a))]`. By subtracting elements of the prefix sum from each other, you can get the sum of elements for any subarray. That is, `sum(a[x:y]) = sum(a[:y]) - sum(a[:x])` for $x < y$.

1.4.2 Would it help to know if two multisets are permutations of each other?

Fundamental theorem of arithmetic: every integer greater than 1 can be represented uniquely as a product of prime numbers.

You can design a hash function that uses **prime factorization** to map multisets to unique integers. For example, you can map all permutations (anagrams) of a string to a unique integer like so:

```python
def compute_hash(s: str) -> int:
    alphabet_primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
                       31, 37, 41, 43, 47, 53, 59, 61, 67,
                       71, 73, 79, 83, 89, 97, 101]
    h = 1
    for ch in s:
        h *= alphabet_primes[ord(ch) - ord('a')]
    return h
```

1.4.3 Do you need to find the previous/next lesser/greater element for each element in a given array?

You should use a **monotonic stack**. There are four types of monotonic stack: *increasing*, *decreasing*, *non-increasing*, and *non-decreasing*. These stacks are used to find next greater elements, previous greater elements, next lesser elements, and previous lesser elements, and all of these problems can be solved using the template code below:

```
def _find_indicies(arr, op, r):
    stack = []
    result = [-1] * len(arr)
    for i in r:
        while stack and op(arr[i], arr[stack[-1]]):
            result[stack.pop()] = i
        stack.append(i)
    return result
```

Problem Type	Stack Type	Loop Conditional	Direction
Next Greater	Non-Increasing	`curr > top`	$\rightarrow$
Previous Greater	Non-Increasing	`curr > top`	$\leftarrow$
Next Lesser	Non-Decreasing	`curr < top`	$\rightarrow$
Previous Lesser	Non-Decreasing	`curr < top`	$\leftarrow$

If it is preferable to loop from left to right while looking for previous greater elements or previous lesser elements, the template code below can be used instead:

```
def _find_indicies2(arr, op):
    stack = []
    result = [-1] * len(arr)
    for i in range(len(arr)):
        while stack and op(arr[i], arr[stack[-1]]):
            stack.pop()
        if stack:
            result[i] = stack[-1]
        stack.append(i)
    return result
```

Problem Type	Stack Type	Loop Conditional	Direction
Previous Greater	Decreasing	`curr >= top`	$\rightarrow$
Previous Lesser	Increasing	`curr <= top`	$\rightarrow$

1.4.4 Do you need to point to the middle node of a linked list?

```python
def find_mid(head):
    slow, fast = head, head.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

1.4.5 Does the problem involve a changing median?

Consider sorting the array and turning the left half into a max-heap and/or the right half into a min-heap.

1.5 Graphs

1.5.1 Would it help to know the shortest path (or distance) between any two nodes in a graph with non-negative edge weights?

You should use Dijkstra's algorithm (see Appendix A.1).

1.5.2 Do you need to detect a cycle in an undirected graph?

You should use a disjoint-set (union-find) forest (see Appendix B.2). The vertices are the elements of the subsets, and a union of subsets corresponds to an edge between vertices/components. If calling `union(x, y)` does not change the structure of the forest, then you know that `x` and `y` belong to the same component and that an edge between them would produce a cycle.

1.5.3 Do you need to connect verticies together without cycles while minimizing total edge weight?

Such a set of edges is known as the **minimum spanning tree** (MST) of a graph. You should use *Kruskal's algorithm* (see Appendix A.2) or *Prim's algorithm* (see Appendix A.3). The former is slightly preferred for sparse graphs, the latter for dense graphs.

1.5.4 Do you need to traverse every edge of a graph exactly once?

Such a sequence of edges is known as an **Eulerian trail** and finding such a trail was the goal of the famous *Seven Bridges of Königsberg* problem. Similarly, an **Eulerian cycle** is an Eulerian trail that starts and ends at the same vertex.

For connected graphs:

- An undirected graph:
  - Has an Eulerian cycle iff every vertex has even degree.
  - Has an Eulerian trail that is not a cycle iff exactly two verticies have odd degree (the start and end verticies).
- A directed graph:
  - Has an Eulerian cycle iff every vertex has equal in-degree and out-degree.
  - Has an Eulerian trail that is not a cycle iff at most one vertex has `out-degree - in-degree = 1` (the start vertex) and at most one vertex has `in-degree - out-degree = 1` (the end vertex).

To find an Eulerian cycle in a graph, you should use Hierholzer's algorithm:

Algorithm 1: Hierholzer's Algorithm `/* see A.4 for code */`

```
Data: $G = (V, E)$
Result: $C$ (a sequence of edges that represents an Eulerian cycle)
$C \longleftarrow$ []
Follow a trail of unused edges starting at $s \in V$ until it returns to $s$,
  appending each edge to $C$                                    /* N1 */
while $\exists\ u \in V \mid u$ is in the trail and has unused adjacent edges do
    $D \longleftarrow$ []
    Follow a trail of unused edges starting at $u$ until it returns to $u$,
      appending each edge to $D$
    Insert $D$ into $C$ before some edge leaving $u$
end
return $C$
```

`N1:` The trail will not get stuck and fail to return to s because all $v \in V$ have even degree $\implies$ when the trail enters another vertex w, w must have an unused edge leaving w.

1.6 Searching

1.6.1 Binary Search

To find a given target (for duplicate targets, return index of first target found in the search):

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

To find the leftmost duplicate target (if target does not exist, return number of elements less than target (rank of target)):

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

`bisect.bisect_left(nums, target)`

To find the rightmost duplicate target (if target does not exist, `(n - right)` is the number of elements greater than target):

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

```
bisect.bisect_right(nums, target) - 1
bisect.bisect(nums, target) - 1
```

1.7 Sorting

1.7.1 Do you need to sort items according to a custom scheme?

- `functools.cmp_to_key`
- Create class and define dunder methods `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`, `__ne__`

1.7.2 Do you need to schedule tasks based on their dependencies?

You can apply **topological sorting** to a directed graph. This will produce a linear ordering of the vertices such that for every directed edge uv from vertex u to vertex v, u comes before v. However, if the graph has cycles, such an ordering does not exist.

There are two main topological sorting algorithms: *Kahn's algorithm* (BFS) and *cycle detection via DFS*. The former cannot visit cycles and detects them by checking for unvisited nodes after traversal. The latter detects cycles by entering the first one it finds and completing a loop.

Algorithm 2: Kahn's Algorithm `/* see A.5 for code */`

```
Data: $G = (V, E)$
Result: $L$ (list of $v \in V$ in topological order)
$L \longleftarrow$ []
$S \longleftarrow \{v \in V \mid v$ has no incoming edges$\}$
while $S$ is not empty do
    remove a node $n$ from $S$
    append $n$ to $L$
    foreach node $m$ with an edge $e$ from $n$ to $m$ do
        remove $e$ from $E$
        if $m$ has no incoming edges then
            add $m$ to $S$
        end
    end
end
if $E$ is empty then
    return $L$
else
    return error                    /* the graph has a cycle */
end
```

Algorithm 3: DFS Topological Sort `/* see A.6 for code */`

```
Data: $G = (V, E)$
Result: $L$ (list of $v \in V$ in topological order)
$L \longleftarrow$ []
Function visit(node $n$)
    if $n$ has a permanent mark then
        return
    end
    if $n$ has a temporary mark then
        stop                              /* the graph has a cycle */
    end
    mark $n$ with a temporary mark
    foreach node $m$ with an edge from $n$ to $m$ do
        visit($m$)
    end
    remove temporary mark from $n$
    mark $n$ with a permanent mark
    prepend $n$ to $L$
end
while $\exists$ nodes without a permanent mark do
    select an unmarked node $n$
    visit($n$)
end
return $L$
```

1.8 Bit Manipulation

- $x = y \implies x \oplus y = 0$
- Need to flip x? $x \oplus 1$
- Two's complement...

2 Useful Python Constructs

Do you need to...

- Count items in a collection?

  $\Longrightarrow$ `collections.Counter` creates a dictionary of the form `{element: count}`

- Return a default value for keys not found in a dictionary?

  $\Longrightarrow$ `collections.defaultdict`

- Get the ASCII value of a character?

  $\Longrightarrow$ `ord(ch)`

- Reverse a list?

  $\Longrightarrow$ The fastest method is the "Martian smiley" `[::-1]`

- Determine if a string is a palindrome?

  $\Longrightarrow$ `s == s[::-1]`

- Find the first occurrence of an item in a list?

  $\Longrightarrow$ `next(x for x in lst if ...)`
  (returns `StopIteration` if item is not found)

`itertools.combinations`, `itertools.permutations`
`re` (regex)
`enumerate` $\rightarrow$ `count, value`
`map`, `filter`, `reduce`, `zip`
deep copy, shallow copy
`id`
`contextlib.suppress`?

3 Unorganized

- DFS $\rightarrow$ stack (recursion) $\rightarrow$ LIFO
- BFS $\rightarrow$ queue (iteration) $\rightarrow$ FIFO
- Online tests: have a Python scratchpad open, spam the "Run Tests" button (EAFP > LBYL)

- Number of subarrays of array of size n (also, the sum of an arithmetic progression, $\sum_{i=0}^{n} i$): $\frac{n(n+1)}{2}$
- Python is pass-by-assignment
  - Immutable objects are pass-by-value
  - Mutable objects are pass-by-reference
  - You can rebind the variable in the inner scope, but the outer scope will remain unchanged
- Some DP notes
  - Optimal substructure $\implies$ divide and conquer
  - Optimal substructure + greedy choice $\implies$ greedy
  - Optimal substructure + overlapping subproblems $\implies$ dynamic programming

4 To Do

4.1 Essential Topics

- Intervals?
- Floyd's Tortoise and Hare Cycle Detection Algorithm
- Helper method recursion (parameter or nonlocal)
- Kadane's algorithm (maximum subarray)
- Dijkstra's algorithm (shortest path in weighted graph)
- Sweep line algorithm (convex hull)
- Sliding window
- LRU Cache (hash map + DLL, OrderedDict)
- Monotonic queue/deque (max/min element in sliding window)
- Tries

4.2 Stretch Topics

- Rabin-Karp (string-searching, uses a rolling hash to make approximate comparisons between substring hash and target hash, makes exact comparison if hashes match)
- Sieve of Eratosthenes (find all prime numbers up to a given integer)
- Segment trees and interval trees
- Shortest path problem for graphs with negative edge weights (Bellman-Ford algorithm, Johnson's algorithm)

A Algorithms

A.1 Shortest Paths - Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a source vertex to every other vertex in a directed graph with non-negative edge weights. It is a greedy algorithm that, in each step, chooses to visit the unvisited vertex that is closest to the source. This vertex is necessarily adjacent to a visited vertex, as unvisited verticies that are not adjacent to a visited vertex are at least one additional edge away from the source.

Let there be a set U of verticies that are both unvisited and adjacent to a visited vertex. The algorithm can be described at a high level as follows:

1. Visit the vertex in U that is closest to the source.
2. Add any unvisited neighbors of that vertex to U.
3. Repeat until U is empty.

From an arbitrary vertex u, the algorithm will update the distance between the source s and each neighbor of u, if the path from s to u and the edge from u to the neighbor form the shortest path found so far from s to that neighbor. If only the shortest path between s and some particular $v \in V$ is needed, the algorithm can terminate after visting v.

Algorithm 4: Dijkstra's Algorithm (heap with *decrease-key*)

```
Data: $G = (V, E)$, edge weights $w(e)$ for $e \in E$, source $s \in V$
Result: $D$ (distances from $s$ to every other vertex in $V$),
        $P$ (the predecessor of every $v \in V$ on the shortest path
           from $s$ to $v$)
foreach $v \in V$ do
    $D(v) := \begin{cases} 0 & \text{if } v = s, \\ \infty & \text{otherwise} \end{cases}$
    $P(v) :=$ null
end
$H \longleftarrow$ priority queue of $V$, using D as priorities
       (contains unvisited verticies)
while $H$ is not empty do
    Pop $u$ from $H$ (the unvisited vertex closest to $s$)
    foreach $e = (u, v)$ where $v$ is unvisted do
        if $(s \to u \to v)$ is the shortest path $(s \to v)$ found so far then
            Increase the priority of $v$ in $H$        $(D(v) := D(u) + w(e))$
            Let $v$ connect to $u$ when $v$ is popped             $(P(v) := u)$
        end
    end
end
return $D, P$
```

Note this line from the above pseudocode:

Increase the priority of v in H $\quad (D(v) := D(u) + w(e))$

Increasing the priority of an arbitrary entry in a priority queue requires a heap with a *decrease-key* operation. The Python `heapq` module does not provide this operation, so the pseudocode above must be implemented with a custom heap. With the priority queue implementation given in Appendix B.1, the algorithm can be written as follows:

```python
def dijkstra(adj_list, weight, source):
    distance = {source: 0}
    predecessor = {}
    pq = PriorityQueue()
    for v in adj_list:
        if v != source:
            distance[v] = inf
            predecessor[v] = None
        pq.insert(v, priority=distance[v])

    while pq:
        u = pq.pop()
        for v in adj_list[u]:
            alt = distance[u] + weight[(u, v)]
            if v in pq and alt < distance[v]:
                pq.decrease_key(v, priority=alt)
                distance[v] = alt
                predecessor[v] = u

    return distance, predecessor
```

However, the code required to implement a heap with *decrease-key* is cumbersome, and the algorithm can be written without modifying entries in the priority queue. Instead of initializing the heap with all of the verticies, verticies can be added when they become adjacent to the vertex currently being visited. And when a shorter alternate path to a vertex is found, duplicate entries can be inserted with higher priority. The priority queue will contain stale data, but if we keep track of which verticies have been visited already, then duplicate entries of lower priority can be discarded when popped.

Algorithm 5: Dijkstra's Algorithm (heap with duplicate entries)

```
Data: $G = (V, E)$, edge weights $w(e)$ for $e \in E$, source $s \in V$
Result: $D$ (distances from $s$ to every other vertex in $V$),
        $P$ (the predecessor of every $v \in V$ on the shortest path
           from $s$ to $v$)
foreach $v \in V$ do
    $D(v) := \begin{cases} 0 & \text{if } v = s, \\ \infty & \text{otherwise} \end{cases}$
    $P(v) :=$ null
end
$H \longleftarrow$ priority queue containing $s$, highest priority
       (contains unvisited verticies adjacent to visited verticies)
while there are unvisited verticies do
    Pop $u$ from $H$ until $u$ is unvisited (discard duplicates)
       (break if $H$ is empty)
    Mark $u$ as visited
    foreach $e = (u, v)$ where $v$ is unvisited do
        if $(s \to u \to v)$ is the shortest path $(s \to v)$ found so far then
            Insert $v$ into $H$ with priority $D(u) + w(e)$
                                                  $(D(v) := D(u) + w(e))$
            Let $v$ connect to $u$ when $v$ is popped      $(P(v) := u)$
        end
    end
end
return $D, P$
```

Because the heap no longer requires *decrease-key*, the `heapq` module is sufficient:

```python
def dijkstra(adj_list, weight, source):
    distance = {v: inf for v in adj_list}; distance[source] = 0
    predecessor = {}
    pq = [(0, source)]
    visited = set()

    while len(visited) < len(adj_list):
        try:
            while (u := heapq.heappop(pq)[1]) in visited: pass
        except IndexError:
            break  # graph is not strongly connected
        visited.add(u)
        for v in adj_list[u]:
            alt = distance[u] + weight[(u, v)]
            if v not in visited and alt < distance[v]:
                heapq.heappush(pq, (alt, v))
                distance[v] = alt
                predecessor[v] = u

    return distance, predecessor
```

If a graph is not strongly connected, then there may not be a path from the source vertex to every other vertex in the graph. In this case, the algorithm will return a distance of ∞ for any vertex not reachable from the source.

Dijkstra's algorithm is designed for directed graphs with non-negative edge weights. For graphs with negative edge weights, one should use the Bellman-Ford algorithm or Johnson's algorithm instead.

Dijkstra's algorithm uses the same core procedure as Prim's algorithm.

A.2 Minimum Spanning Trees - Kruskal's Algorithm

Kruskal's algorithm finds a minimum spanning forest (MSF) of an undirected, weighted graph. If the graph is connected, the MSF is a minimum spanning tree (MST). It is a greedy algorithm that, in each step, adds to a tree the lightest edge that will not form a cycle. It uses a disjoint-set forest to detect whether adding an edge will form a cycle.

Algorithm 6: Kruskal's Algorithm

```
**Data:** $G = (V, E)$ with edge weights $w(e)$ for $e \in E$
**Result:** $T$ (a set of edges that represents an MST)
$T \longleftarrow$ []
$D \longleftarrow$ disjoint-set forest of $V$
Sort $E$ by weight, increasing
foreach $(u, v) \in E$ do
    if $u$ and $v$ do not belong to the same disjoint set in $D$ then
        Add $(u, v)$ to $T$
        Union $u$ and $v$ in $D$
    end
end
return $T$
```

See Appendix B.2 for an implementation of a disjoint-set forest with weighted union and collapsing find. Given an edge list representing a graph and a weight function, Kruskal's algorithm can be implemented as follows:

```python
def kruskal(edges, weight):
    verticies = set()
    for u, v in edges:
        verticies.add(u)
        verticies.add(v)
    verticies = list(verticies)
    n = len(verticies)

    vertex_to_index = {}
    for i in range(n):
        vertex_to_index[verticies[i]] = i

    edges.sort(key=weight)
    uf = DisjointSet(n)

    mst = []
    for u, v in edges:
        i, j = vertex_to_index[u], vertex_to_index[v]
        if uf.union(i, j):
            mst.append((u, v))

    return mst
```

Note that the values in the disjoint-set forest correspond to the indicies of the vertex list, but the disjoint sets in the forest represent groupings of the verticies themselves.

A.3 Minimum Spanning Trees - Prim's Algorithm

Prim's algorithm finds a minimum spanning tree (MST) of an undirected, weighted, connected graph. Some variants can find the minimum spanning forest (MSF) of a disconnected graph (more on this below). It is a greedy algorithm that, in each step, adds to a tree the lightest edge that will connect a vertex in the tree to a vertex not in the tree. It can be described at a high level by the following three steps:

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

To find the minimum-weight edge from a tree vertex to a non-tree vertex, it would help to have a heap or priority queue. The heap could contain every non-tree vertex (initially, every vertex) and prioritize them based on the minimum cost to connect them to the tree (weight of lightest edge connecting to tree). Cost would be finite for adjacent verticies, infinite otherwise. As long as we keep track of which tree vertex provides that min-cost connection for each non-tree vertex, we will know which edges to add to the tree.

Algorithm 7: Prim's Algorithm (heap with *decrease-key*)

```
Data: $G = (V, E)$ with edge weights $w(e)$ for $e \in E$
Result: $T$ (a set of edges that represents an MST)
foreach $v \in V$ do
    $C(v) \coloneqq \infty$ (cost of cheapest connection to $v$)
    $P(v) \coloneqq$ null (vertex that provides cheapest connection to $v$)
end
$H \longleftarrow$ priority queue of $V$, using $C$ as priorities
         (contains the verticies not yet in the tree)
while $H$ is not empty do
    Pop $u$ from $H$ (the cheapest vertex to connect to the tree)
    foreach $e = (u, v)$ where $v$ is not yet in the tree do
        if $e$ is the cheapest path to $v$ found so far then
            Increase the priority of $v$ in $H$          ($C(v) \coloneqq w(e)$)
            Let $v$ connect to $u$ when $v$ is popped    ($P(v) \coloneqq u$)
        end
    end
end
$T \longleftarrow \{(P(v), v) \; \forall \, v \in V \mid P(v)$ is not null$\}$
return $T$
```

Note this line from the above pseudocode:

$$\text{Increase the priority of } v \text{ in } H \qquad (C(v) \coloneqq w(e))$$

Increasing the priority of an arbitrary entry in a priority queue requires a heap with a *decrease-key* operation. The Python `heapq` module does not provide this operation, so the pseudocode above must be implemented with a custom heap. With the priority queue implementation given in Appendix B.1, the algorithm can be written as follows:

```python
def prim(adj_list, weight):
    prev = {}
    pq = PriorityQueue([[inf, v] for v in adj_list])
    while pq:
        u = pq.pop()
        for v in adj_list[u]:
            if v in pq and (w := weight[(u, v)]) < pq.
                                                get_priority(v):
                pq.decrease_key(v, priority=w)
                prev[v] = u

    return [(prev[v], v) for v in prev]
```

However, the code required to implement a heap with *decrease-key* is cumbersome, and the algorithm can be written without modifying entries in the priority queue. Instead of initializing the heap with all of the verticies, verticies can be added when they become adjacent to the tree. And when costs to connect are lowered, duplicate entries can be inserted with higher priority. The priority queue will contain stale data, but if we keep track of which verticies have been added to the tree, then duplicate entries of lower priority can be discarded when popped.

Algorithm 8: Prim's Algorithm (heap with duplicate entries)

```
Data: $G = (V, E)$ with edge weights $w(e)$ for $e \in E$
Result: $T$ (a set of edges that represents an MST)
foreach $v \in V$ do
    $C(v) := \infty$ (cost of cheapest connection to $v$)
    $P(v) :=$ null (vertex that provides cheapest connection to $v$)
end
$H \longleftarrow$ priority queue containing arbitrary $v \in V$, highest priority
        (contains the verticies adjacent to but not yet in the tree and
        also possibly stale duplicates of verticies in the tree)
while there are verticies that are not in the tree do
    Pop $u$ from $H$ until $u$ is not in the tree (discard duplicates)
    Add $u$ to the tree
    foreach $e = (u, v)$ where $v$ is not yet in the tree do
        if $e$ is the cheapest path to $v$ found so far then
            Insert $v$ into $H$ with priority $w(e)$      $(C(v) := w(e))$
            Let $v$ connect to $u$ when $v$ is popped    $(P(v) := u)$
        end
    end
end
$T \longleftarrow \{(P(v), v) \; \forall \, v \in V \mid P(v)$ is not null$\}$
return $T$
```

Because the heap no longer requires *decrease-key*, the `heapq` module is sufficient:

```
def prim(adj_list, weight):
    cost = {v: inf for v in adj_list}
    prev = {}
    pq = [(0, next(iter(adj_list)))]
    tree = set()  # visited

    while len(tree) < len(adj_list):
        while (u := heapq.heappop(pq)[1]) in tree: pass
        tree.add(u)
        for v in adj_list[u]:
            if v not in tree and (w := weight[(u, v)]) < cost[v]
                                                            :
                heapq.heappush(pq, (w, v))
                cost[v] = w
                prev[v] = u

    return [(prev[v], v) for v in prev]
```

Note that the first implementation of Prim's algorithm will find the MSF for a disconnected graph. The second implementation will only find the MST of the connected component that it starts in, but the function could be called on every connected component in order to find the MSF.

Prim's algorithm uses the same core procedure as Dijkstra's algorithm.

A.4 Eulerian Cycle Detection - Hierholzer's Algorithm

Hierholzer's algorithm finds an Eulerian cycle in a directed graph where such a cycle is assumed to exist. Below are two variants: the former returns a list of verticies, the latter returns a list of edges. These functions can also be used to find an Eulerian trail, provided that `s` is set to the vertex with `out-degree - in-degree = 1`, if such a vertex exists.

```
from collections import deque

# Returns the vertex sequence of the Eulerian cycle
def hierholzer_vertices(graph: dict[str, list[str]], s: str) ->
                                 list[str]:
    cycle = deque()
    def dfs(v):
        while graph[v]:
            dfs(graph[v].pop())
        cycle.appendleft(v)

    dfs(s)
    return cycle


# Returns the Eulerian cycle
def hierholzer_edges(graph: dict[str, set[str]], s: str) -> list
                                [str]:
    cycle = deque()
    def dfs(src, dst):
        while graph[dst]:
            dfs(dst, graph[dst].pop())
        cycle.appendleft([src, dst])
        while graph[src]:
            dfs(src, graph[src].pop())

    dfs(s, graph[s].pop())
    return cycle
```

A.5 Topological Sorting - Kahn's Algorithm

```
def find_order_bfs(adj_list: list[list[int]],
                   in_degrees: list[int]) -> list[int]:
    # 1. Create list of start nodes
    queue = deque()
    for n, d in enumerate(in_degrees):
        if d == 0:
            queue.append(n)

    topo_order = []
    while queue:
        # 2. Add a start node n to the topological ordering
        n = queue.popleft()
        topo_order.append(n)

        # 3. Remove edges from n to its neighbors
        #    Add neighbors of in-degree 0 to start node list
        for m in adj_list[n]:
            in_degrees[m] -= 1
            if in_degrees[m] == 0:
                queue.append(m)
    return topo_order if len(topo_order) == len(adj_list) else [
        ]
```

A.6 Topological Sorting - DFS Cycle Detection

```
def find_order_dfs(adj_list: list[list[int]]) -> list[int]:
    visited = set()
    dfs_tree = set()
    topo_order = []

    def has_cycle(n):
        if n in visited:  # path already explored
            return False
        if n in dfs_tree:  # cycle detected
            return True

        dfs_tree.add(n)
        for m in adj_list[n]:
            if has_cycle(m):
                return True

        dfs_tree.remove(n)
        visited.add(n)
        topo_order.append(n)
        return False

    for n in range(len(adj_list)):
        if has_cycle(n):
            return []
    return topo_order
```

B Data Structures

B.1 Heaps and Priority Queues

A **heap** is a tree-based data structure that satisfies the heap property:

- For a min-heap, every parent is less than or equal to its children.
- For a max-heap, every parent is greater than or equal to its children.

Heaps are useful when you need direct access to the smallest or largest element in a mutable or dynamic collection.

Heaps are usually implemented with arrays. For a binary heap, the node stored at index 0 is the root, and a node stored at index i has children at indicies $2i + 1$ and $2i + 2$ and a parent at $\lfloor (i - 1)/2 \rfloor$.

A min-heap has three essential operations:

Operation	**Time**	**Python Function**
Heapify	$O(n)$	`heapq.heapify(heap)`
Insert	$O(\log n)$	`heapq.heappush(heap, item)`
Extract-min	$O(\log n)$	`heapq.heappop(heap)`

- *Heapify*: for all non-leaf nodes, from the last ($i = \lfloor n/2 \rfloor - 1$) to the root, sift the node down (for a max of one swap)
- *Insert*: append the new item to the array, sift it up
- *Extract-min*: swap the root r with the last element of the array x, pop r from the end of the array, sift x down, return r

The Python `heapq` module provides two more heap operations that improve efficiency when you push-then-pop or pop-then-push. The amount of sift operations required is reduced from two to one:

Operation	**Time**	**Python Function**
Push-pop	$O(\log n)$	`heapq.heappushpop(heap, item)`
Replace	$O(\log n)$	`heapq.heapreplace(heap, item)`

- *Push-pop*: if the new item x is less than the root r, return x; else, save r, overwrite the first element of the array with x, sift x down, return r
- *Replace*: append the new item to the array, call *extract-min*

Heaps are often used to implement **priority queues**. A priority queue is an abstract data type similar to a queue or stack. Each element in a priority queue has an associated *priority*, and elements with high priority are dequeued before elements with low priority. A priority queue has two essential operations: *insert with priority* and *extract element with highest priority*.

There are other operations that a priority queue could have. For example, if a priority queue is being used to organize a set of tasks, it might be useful to increase the priority of a task as circumstances change. To do this, the priority queue (or, technically, the underlying heap) would need a *decrease-key* operation (assuming that smaller keys correspond to higher priorities).

The heapq module does not provide this operation; see the code below for a priority queue implementation that has *decrease-key*:

```
class PriorityQueue:
    def __init__(self, items=[]):
        self.pq = items
        self.index = {t: i for (i, [_, t]) in enumerate(items)}
        self._heapify()

    def __len__(self):
        return len(self.pq)

    def __contains__(self, task):
        return task in self.index

    def insert(self, task, priority=0):
        self.pq.append([priority, task])
        self.index[task] = len(self.pq) - 1
        self._sift_up(len(self.pq) - 1)

    def pop(self):
        if len(self.pq):
            self.pq[0], self.pq[-1] = self.pq[-1], self.pq[0]
            self.index[self.pq[0][1]] = 0
            _, task = self.pq.pop()
            del self.index[task]
            self._sift_down(0)
            return task
        raise KeyError('Pop from empty priority queue')

    def decrease_key(self, task, priority=0):
        if self.get_priority(task) <= priority:
            raise ValueError('New priority is not less than old
                                                priority')
        c = self.index[task]
        self.pq[c][0] = priority
        self._sift_up(c)
    def get_priority(self, task):
        if task not in self.index:
            raise KeyError('Task not in priority queue')
        return self.pq[self.index[task]][0]

    def _heapify(self):
        for p in range(len(self.pq) // 2 - 1, -1, -1):
            self._sift_down(p)

    def _sift_up(self, c):
        p = (c - 1) // 2
        if c > 0 and self.pq[c] < self.pq[p]:
            self._swap(c, p)
            self._sift_up(p)

    def _sift_down(self, p):
        n = len(self.pq)
        cl, cr = 2 * p + 1, 2 * p + 2
        if cl >= n:
            return
        if cr >= n:
            cr = cl
        c = cl if self.pq[cl] < self.pq[cr] else cr

        if self.pq[c] < self.pq[p]:
            self._swap(c, p)
            self._sift_down(c)

    def _swap(self, i, j):
        self.index[self.pq[i][1]] = j
        self.index[self.pq[j][1]] = i
        self.pq[i], self.pq[j] = self.pq[j], self.pq[i]
```

B.2 Disjoint-Set (Union-Find) Forest

A disjoint-set forest models the partitioning of a set. Initially, each element of the set belongs to a subset where it is the only member. Two subsets can be united into a single subset that contains the elements of each. The union of a set with itself is itself.

These subsets are represented as trees in the structure, and the structure has two operations on these trees: `union(x, y)` and `find(x)`, where `x` and `y` are elements of the set. When `union(x, y)` is called, the subset that `x` belongs to is united with the subset that `y` belongs to. Structurally, the root of one tree becomes the child of the other tree's root. If `x` and `y` belong to the same set, the structure does not change. When `find(x)` is called, the root of the tree that `x` belongs to is returned. This is the "representative member" of the set, a kind of "name" for the set.

The following class arbitrarily chooses `x` to be the parent of `y` upon their union. Here, the parent of a root is itself, but 0 would also be a fine choice.

```
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))
        # self.parent = [0] * n

    def union(self, x, y):
        self.parent[self.find(y)] = self.find(x)

    def find(self, x):
        return x if x == self.parent[x] else self.find(self.
                                        parent[x])
        # while self.parent[x]:
        #     x = self.parent[x]
        # return x
```

The following class implements two enhancements known as *weighted union* and *collapsing find*. The parent of a root is now a negative number whose absolute value corresponds to the tree's *weight* or *rank*. When `union(x, y)` is called, where `x`'s tree has greater weight than `y`'s tree, the weight of `y`'s tree will be added to the weight of `x`'s tree, and the root of `y`'s tree will point to the root of `x`'s tree. This ensures that the united tree is more balanced. `find(x)` now sets the parent of any node on the path from `x` to the representative member of the tree to the representative member. Initially, `find(x)` is $O(\log n)$, but subsequent calls are $O(1)$.

```
class DisjointSet:
    def __init__(self, n):
        self.parent = [-1] * n

    def union(self, x, y):
        rx, ry = self.find(x), self.find(y)
        if rx == ry:
            return False
        elif self.parent[rx] < self.parent[ry]:
            self.parent[rx] += self.parent[ry]
            self.parent[ry] = rx
        else:
            self.parent[ry] += self.parent[rx]
            self.parent[rx] = ry
        return True

    def find(self, x):
        if self.parent[x] < 0:
            return x
        self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
```

Note that the elements of a disjoint set could be represented with something other than integers (so `self.parent` would be a dictionary), but weighted union would not be able to be implemented as it is above.

C Glossary

C.1 Graph Theory

- *Walk*: a sequence of edges which joins a sequence of vertices
- *Trail*: a walk in which all edges are distinct
- *Cycle*: a trail that begins and ends at the same vertex
- *Path*: a trail in which all verticies are distinct
- A graph is *strongly connected* if every vertex is reachable from every other vertex