Data Structures & Algorithms Cheat Sheet

Thomas Monson

Contents

1 Essential Patterns

1.1 Dynamic Programming

1.1.1 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)`.

1.1.2 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)
```

```
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]
```

```
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]
```

(combinatorial optimization)

```
@functools.lru_cache
```

1.2 Backtracking

(combinatorial search)

Backtracking solutions can be implemented with the following template:

```
result = []
```

```
def is_valid_state(state):
    return True

def get_candidates(state):
    return []

def backtrack(state):
    if is_valid_state(state):
        result.append(state[:])
        return

    for candidate in get_candidates(state):
        state.append(candidate)
        backtrack(state)
        state.pop()

backtrack([])
```

```
result = []

def is_valid_state(state):
    if len(state) == k:
        return True

def get_candidates(state, start):
    need = k - len(state)
    remain = n - start + 1
    available = remain - need

    return list(range(start, start + available + 1))

def backtrack(state, start):
    if is_valid_state(state):
        result.append(state[:])
        return

    for candidate in get_candidates(state, start):
        state.append(candidate)
        start += 1
        backtrack(state, start)
        state.pop()

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

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:

```
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:

```python
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`	→
Previous Greater	Non-Increasing	`curr > top`	←
Next Lesser	Non-Decreasing	`curr < top`	→
Previous Lesser	Non-Decreasing	`curr < top`	←

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:

```python
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`	→
Previous Lesser	Increasing	`curr <= top`	→

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
```

1.5 Graphs

1.5.1 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.

1.5.2 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.1 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.2 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.3 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]`

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

3 Unorganized

- DFS → stack (recursion) → LIFO
- BFS → queue (iteration) → 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

- Heap Structure
- 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)
- Backtracking (DFS) ("the best solutions often model the problem in some way that allows them to quickly prune state prefixes that cannot lead to solutions")
- 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)
- Kruskal's algorithm and Prim's algorithm (minimum spanning tree)
- Sieve of Eratosthenes (find all prime numbers up to a given integer)
- Segment trees and interval trees

A Algorithm Code

A.1 Eulerian Cycle Detection - Hierholzer's Algorithm

Below are two functions that find the Eulerian cycle in a directed graph where such a cycle is assumed to exist. 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.2 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.3 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

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

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 x:
        #     x = 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]
```

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