

Data Structures & Algorithms Cheat Sheet

Thomas Monson

Essential Patterns

Dynamic Programming

Optimal substructure \implies divide and conquer

Optimal substructure + greedy choice \implies greedy

Optimal substructure + overlapping subproblems
 \implies dynamic programming

Would it be helpful to rephrase a 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 i th 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)**.

```
@functools.lru_cache
```

Arrays

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, $\text{sum}(a[x:y]) = \text{sum}(a[:y]) - \text{sum}(a[:x])$ for $x < y$.

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
```

Searching

Here's some code for a binary 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
```

bisect (binary search)

Sorting

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__`

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 1: Kahn's Algorithm */* see A.1 for code */*

```

Data:  $G = (V, E)$ 
Result:  $L$  (list of  $v \in V$  in topological order)
 $L \leftarrow []$ 
 $S \leftarrow \{v \in V \mid v \text{ has no incoming edges}\}$ 
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 2: DFS Topological Sort */* see A.2 for code */*

Data: $G = (V, E)$ **Result:** L (list of $v \in V$ in topological order) $L \leftarrow []$ **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

Other Stuff

- Helper method recursion (parameter or nonlocal)
- Kadane's algorithm (maximum subarray)
- Knapsack problem (combinatorial optimization)
- Sweep line algorithm (convex hull)
- Backtracking (DFS)
- Sliding window

- LRU Cache (hash map + DLL, OrderedDict)
- Monotonic stack
- Union-find

Useful Python Constructs

Do you need to...

- Count items in a collection?
 \implies `collections.Counter` creates a dictionary of the form
`{element: count}`
- Return a default value for keys not found in a dictionary?
 \implies `collections.defaultdict`
- Get the ASCII value of a character?
 \implies `ord(ch)`
- Reverse a list?
 \implies The fastest method is the “Martian smiley” `[::-1]`

`itertools.combinations`, `itertools.permutations`
`re` (regex)
`enumerate` \rightarrow `count`, `value`
`map`, `filter`, `reduce`, `zip`

Other

- 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 : $\frac{n(n+1)}{2}$
- 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

Potentially Useful Algorithms

- 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)

A Python Code Samples

A.1 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.2 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

```