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 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)`.

```
@functools.lru_cache
```

Sets

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).

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, `sum(a[x:y]) = sum(a[:y]) - 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
```

Graphs

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.

Searching

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
```

```python
    return right - 1
```

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

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 ∈ V in topological order)
L ⟵ []
S ⟵ {v ∈ V | 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 2: DFS Topological Sort `/* see A.2 for code */`

```
Data: G = (V, E)
Result: L (list of v ∈ V in topological order)
L ⟵ []
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 ∃ 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)
- 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 stack

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` → `count, value`
`map`, `filter`, `reduce`, `zip`
deep copy, shallow copy

Other

- 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: $\frac{n(n+1)}{2}$
- 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)
- Sieve of Eratosthenes (find all prime numbers up to a given integer)

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
```

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]
```