REUSABLE IDEAS

PROBLEM-SOLVING BOOSTERS

Transform the Input
Often, transforming the input in a preprocessing step is not strictly necessary, but it can simplify the logic of the final solution. If it retains the same time and space complexities, it can be worth doing to simplify your life and reduce the risk of bugs.

Length-26 Array
When the input is a string of lowercase letters, we often can use an array of length `26` where index `0` corresponds to `'a'`, index `1` corresponds to `'b'`, and so on. This is more efficient than using a map data structure.

For instance, if we want to count how many times each letter appears in a string:

```
# We are told that s only contains lowercase letters.
def letter_counts(s):
  counts = [0] * 26
  for c in s:
    index = ord(c) - ord('a')
    counts[index] += 1
  return counts
```

If we want to support all ASCII characters, then we can use an array of length `128` instead (and not subtract `ord('a')`).

STRING MANIPULATION

Building Strings With Dynamic Arrays
Check if strings are mutable in your language of choice. If you need to build a string character by character, and the strings in your language are immutable, put the characters in a dynamic array instead. When you are done, convert the array to a string with the built-in join method.

TWO POINTERS

Searching With Inward Pointers
In problems where we have to find an index or pair of indices in a sorted array, we can often 'discard' the largest value based on the smallest one or vice versa. Such problems are a natural fit for inward pointers.

GRIDS & MATRICES

Directions Array To Visit Grid Neighbors
Navigating through a grid by checking neighbor cells is common in graphs, backtracking, and other topics. We can do this compactly with a `directions` array that contains all the possible offsets to add to the current position to generate its neighbors.

We can easily extend the `directions` array with additional directions (e.g., diagonals) depending on the problem.

Factor Out Validation To A Helper Function
When checking if a cell is valid, embedding the logic directly in your main algorithm can clutter the code. Instead, consider encapsulating the validation logic in a dedicated `is_valid()` function.

Beyond grid cells, this applies anywhere you must validate 'something' before using it.

Handle Wraparounds With Mod
For problems involving 'circular arrays,' where the last element wraps around to the first one, modular arithmetic can simplify indexing logic: the next element after index `i` is always `(i + 1)%n`, where `n` is the length of the array.

BINARY SEARCH

Grid Flattening
If we want to iterate or search through a grid with dimensions `RxC` as if it was a 'normal' array of length `R*C`, we can use the following mapping from grid coordinates to "flattened-grid array" coordinates:

```
[r, c] → r * C + c
```

and the reverse mapping to go from "flattened-grid array" coordinates to grid coordinates:

```
i → [i // C, i % C]
```

For instance, cell `[1, 2]` in Figure 5 (the 9) becomes index `1 * 4 + 2 = 6`, and, conversely, index 6 becomes cell `[6 // 4, 6 % 4] = [1, 2]`.

Exponential Search
Whenever we need to search for a value in a range, but the upper bound (or even lower bound) of the range is unknown, we can find it efficiently with repeated doubling.

This is often useful in the guess-and-check technique (e.g., Problem 29.10: Water Refilling).

SETS & MAPS

Frequency Maps
A *frequency map* is a map from elements to counts. It can be initialized in `O(n)` time and is useful in questions about duplicate detection, counting, and most frequent elements.

Leverage the Input Range
When the input range is small, leveraging it can lead to significantly more efficient solutions. For instance, when it comes to sorting, we may be able to use specialized algorithms like counting sort. Similarly, the difference array technique in Prefix Sums works well for interval problems when the maximum value in the range is constrained—like 24 to represent the hours in a day.

Impose A Canonical Order
If a problem deals with arrays and we want to treat two arrays as the same when they have the same elements regardless of the order (like `[1, 2]` and `[2, 1]`), we can sort them and compare (or hash) their sorted versions rather than comparing them directly.

SORTING

Sort To Track The Smallest/Largest Elements
If we need to repeatedly find the smallest/largest elements of an array without modifying the array itself, we can sort the indices instead, and iterate through the indices.

If we do not have all the elements upfront, it tends to be more efficient to use a heap (Problem 37.3: Top Songs Class).

STACKS & QUEUES

Piggybacking Extra Info
When storing elements in a data structure, it may help to store extra fields with information to help you process the element once you fetch it from the data structure. In Python, we can use tuples.

In Problem 32.2: Compress Array By K (pg 5), the count piggybacks along with the number. For additional examples, see the 'node–height queue' recipe for trees, where the height piggybacks along with the node, or Chapter 37: K Most Played, where the title of the song piggybacks in the heap along with the number of plays.

If you think about it, a *map* is just a *set* with extra info piggybacking along with each key.

Reframe Balanced Parentheses As Plot Heights
We can visualize a string of parentheses as a plot that starts at 0 and goes up for each opening parenthesis and down for each closing parenthesis:

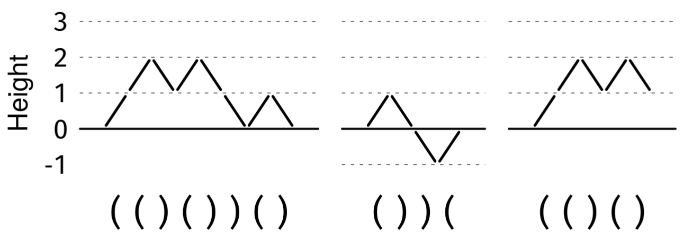

Figure 4. An example of a balanced string (left) and two unbalanced ones (center and right).

The plot for a balanced string creates a valid 'mountain range outline:'

- it never goes below height 0, and
- it ends at height 0.

Balanced parentheses questions are often easier to solve when reframed in terms of this plot.

RECURSION

Pass Indices, Not Copies
By referencing positions within an original array or string using indices, we can eliminate the overhead of slicing and copying, improving time and space efficiency.

Avoiding copies does not only apply to recursion, but it can stack up quickly in recursive algorithms. It is very language dependent, so do some research on your language of choice about how to avoid them (we touched on this in section 'The space complexity of copies'.

Applying Modulo At Each Step
Some problems involve math operations that could result in really large integers that overflow in some languages. As a workaround, many questions ask to return the solution modulo some large prime (we'll use `10`^{9}+7 in this book).

When we see that, we should apply the modulo at each step rather than at the very end, to avoid overflow/working with really large numbers. The neat property of modulo is that, if an operation only involves sums, subtractions, and multiplications (but not divisions), applying it at each step does not change the final output.

E.g., `(10 + 3 + 10 + 6) % 5` is 4, and so is `(10 % 5) + (3 % 5) + (10 % 5) + (6 % 5)`.

Division is the exception: dividing first and then applying modulo yields different results than applying modulo first and then dividing: `(12 / 3) % 5 != (12 % 5) / 3`.

LINKED LISTS

Dummy Node
Adding a dummy node at the head simplifies edge cases by reducing null checks—just don't forget that the real head is `dummy.next`!

Linked List Reversal Break Down
If a problem asks to reverse only a part of a linked list, we can use the break down the problem booster and do it in steps:

1. Find the bounds of the section that need to be reversed.
2. Break that section out of the linked list without losing the before/after parts of the list.
3. Reverse the section (as in Solution 6).
4. Reattach the section.

Each of these steps should be easier to tackle. Using a dummy head will simplify the cases where the real head is part of the section being reversed.

TREES

Finding The Node In The Path With Minimum Depth

A useful property in problems about finding paths in binary trees is that a path has a single node with minimum depth.

Finding this node is often easier than finding the endpoints because there are only `n` candidates as opposed to $O(n^2)$ possible pairs of endpoints. If you can find this node, it's then usually easy to find the rest of the path.

For instance, this property would have helped us solve Problem 24.12: Tree diameter.

GRAPHS

Edge List To Adjacency List

```
def build_adjacency_list(V, edge_list):
  graph = [[] for _ in range(V)]
  for node1, node2 in edge_list:
    graph[node1].append(node2)
    graph[node2].append(node1)
```

For directed graphs, each edge `[node1, node2]` in `edge_list` represents an edge from `node1` to `node2`. Thus, we would add `node2` to the adjacency list of `node1`, but not the other way around.

Path Reconstruction

In DFS when we mark a node as visited, we can also track their *predecessor* in the DFS. The mapping from each node to its predecessor defines a 'DFS visit tree' rooted at the starting node. The predecessor of each node is the parent in this DFS tree (the starting node has no predecessor). Given any node, `node`, reachable from the starting node, `start`, we can follow a chain of predecessors to find a path from `node` to `start`.

Every chain of predecessors leads to the starting node, so we can keep going until we reach `start`:

```
path = [node]
while path[-1] != start:
  path.append(predecessors[path[-1]])
```

Path reconstruction also works the BFS, in which case it doesn't just find any path, it finds the shortest path.

Computing averages

When computing averages, it is often easiest to focus on computing the numerator and denominator separately and dividing at the very end.

Multisource BFS

In `O(V+E)` time, we can find the distance from every node to its closest node in a designated subset with multisource BFS. It's like a normal BFS except for the initialization: we start by putting all the designated nodes in the queue and setting their distances to `0`.

HEAPS

Heap Size Restriction

Depending on the problem, restricting the size of the heap to the size of the output can save space and even time.

For a 'largest `k`' problem, you can use either a max-heap or a restricted-size min-heap.

- Max-heap: dump all elements in the max-heap with heapify and pop the largest `k`. This takes `O(n + (k log n))` time and `O(n)` space.
- Min-heap: add the elements one by one to a min-heap of the `k` largest elements so far. This takes `O(n log k)` time and `O(k)` space.

There is an interesting trade-off between the time and space of the two solutions.

Two-Heap Median Tracking

Tracking the median in a dynamic dataset is possible by keeping the lower half in a max-heap and the upper half in a min-heap.

Each addition takes `O(log n)` time, and returning the median takes `O(1)` time.

M-way Merge
We can efficiently merge `m` sorted lists into a single sorted list by tracking `m` pointers in a heap pointing to the first non-added element from each list.

SLIDING WINDOWS

Transform Exactly-K/At-Least-K Counting to At-Most-K Counting
If the At-Most-K version of a counting problem has the maximum window property, it can be reused to solve the Exactly-K and At-Least-K versions.

- 'Exactly `k`' is equivalent to 'at most `k`' minus 'at most `k - 1`'.
- 'At Least `k`' is equivalent to 'total count' (`n*(n+1)/2`) minus 'at most `k - 1`'.

The At-Most-K version can be solved by tweaking the maximum window template, as in Solution 18.

BACKTRACKING

Modify -> Recurse -> Undo
When you make a mess, clean it up. In backtracking, if you modify a global variable before calling `visit()` recursively, you must do the opposite change immediately upon returning from `visit()`. Code how you make the mess and how you clean it up symmetrically. For example, if you append to an array right before your recursive call, pop from it right after (not in the recursive function).[2]

Extracting And Adding Digits
In some problems, we need to handle integers digit by digit. The following loop extracts all the digits of `x` and returns them as an array, from least to most significant:

```
arr = []
while x > 0:
  arr.append(x % 10)  # Extract the last digit.
  x = x // 10  # Remove the last digit.
```

If `x = 123`, we'll get `arr = [3, 2, 1]`. We can reassemble `x` like this (note that we reverse `arr`):

```
x = 0
for digit in reversed(arr):
  x = x * 10 + digit
```

The same idea applies if we need to handle integers bit by bit instead.

DYNAMIC PROGRAMMING

Solution Reconstruction
DP problems asking for the *value* of the solution are easier than problems that ask for the solution itself.

A practical way to return full solutions to DP problems instead of just their values is to store them in the `memo` table. It can piggyback along with the value, if the value is also needed (see 'Reusable Idea: Piggybacking Extra Info'). However, this increases the time and space complexity.

The more efficient approach is to start by implementing the recurrence relation for values only. Then, we reconstruct the solution step by step, essentially following a path down the decision tree. The key is to use the function for the value-based recurrence relation to determine the best choice at each step. This is analogous to how we build our partial solution in backtracking, except we only go down one path, so we don't need to do it recursively.

2 You can avoid having to worry about cleanups by making copies of the parent's state for each child. This can simplify the code but it increases the time and space we use at each node.