More common patterns

$O(n) string building$

In languages like Python and Java, strings are immutable. So building up strings incrementally will require repeated copying that will take: `1 + 2 + 3 + .. + n` steps which is $O(n^2)$. Instead, in Python you can use a list and Java you can use the StringBuilder class.

Python

1. Declare a list
2. When building the string, add the characters to the list. This is $O(1)$ per operation. Across `n` operations, it will cost $O(n)$ in total
3. Once finished, convery the list to a string using `"".join(list)`. This is $O(n)$
4. In total, it cost us $O(n + n) = O(2n) = O(n)$

```
def build_string(s):
    arr = []
    for c in s:
        arr.append(c)

    return "".join(arr)
```

Java

1. Use the StringBuilder class
2. When building the string, add the characters to the list. This is $O(1)$ per operation. Across `n` operations, it will cost $O(n)$ in total
3. Once finished, convert the list to a string using `StringBuilder.toString()`. This is $O(n)$
4. In total, it cost us $O(n + n) = O(2n) = O(n)$

```
public String buildString(String s) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < s.length(); i++) {
        sb.append(s.charAt(i));
    }
```

```
        return sb.toString();
    }
```

C++ and Javascript

Simply using `+=` is fine when bulding strings.

Definitions

Subarrays/substrings

A subarray or substring is a contiguous section of an array or string.

If a problem has explicit constraints such as:

- Sum greater than or less than `k`
- Limits on what is contained, such as the maximum ok `k` unique elements or no duplicatees allowed

And/or asks for:

- Minimum or maximum length
- Number of subarrays/strings
- Max or minimum sum

Think about using a sliding window (but this is just a general guideline).

If a problem's input is an integer array and you find yourself needing to calculate multiple subarray sums, consider building a prefix sum.

The size of a subarray between `i` and `j` (inclusive) is `j - i + 1`. This is also the number of subarrays that end at `j`, starting from `i` or later.

Subsequences

A subsequence is a set of elements of an array/string that keeps the same relative order but doesn't need to be contiguous.

For example, subsequences of `[1, 2, 3, 4]` include: `[1, 3]`, `[4]`, `[2, 3]`, but not `[3, 2]`, `[5]`, `[4, 1]`.

Dynamic programming is used to solve a lot of subsequence problems. But from what we've learned so far, the most commonm one associated with subsequences is two pointers when two input arrays/strings are given. Because prefix sum and sliding window represent subarrays/substrings, they are not applicable here.

Subsets

A subset is any set of elements from the original array or string. The order doesn't matter and neither do the elements being beside each other. For example, given `[1, 2, 3, 4]`, all of these are subsets: `[3, 2]`, `[4, 1, 2]`, `[]b1`. Note: subsets that contain the same elements are considered the same, so `[1, 2, 4]` is the same subset as `[4, 1, 2]`.

The difference between a subsequence and a subset is that order matters for a subsequence.

We will see subsets being used in the backtracking chapter.

One thing to note is that if a problem involves subsequences, but the order of the subsequences doesn't actually matter (let's say it wants the sum of subsequences), then you can treat it the same as a subset. A useful things that you can do when dealing with subsets that you can't do with subsequences is that you can sort the input, since the order doesn't matter.