BEYOND CRACKING *the* CODING INTERVIEW

Pass Tough Coding Interviews,
Get Noticed, and Negotiate Successfully

With replays & data from over 100k interviews on
interviewing.io

GAYLE LAAKMANN MCDOWELL
MIKE MROCZKA | ALINE LERNER | NIL MAMANO

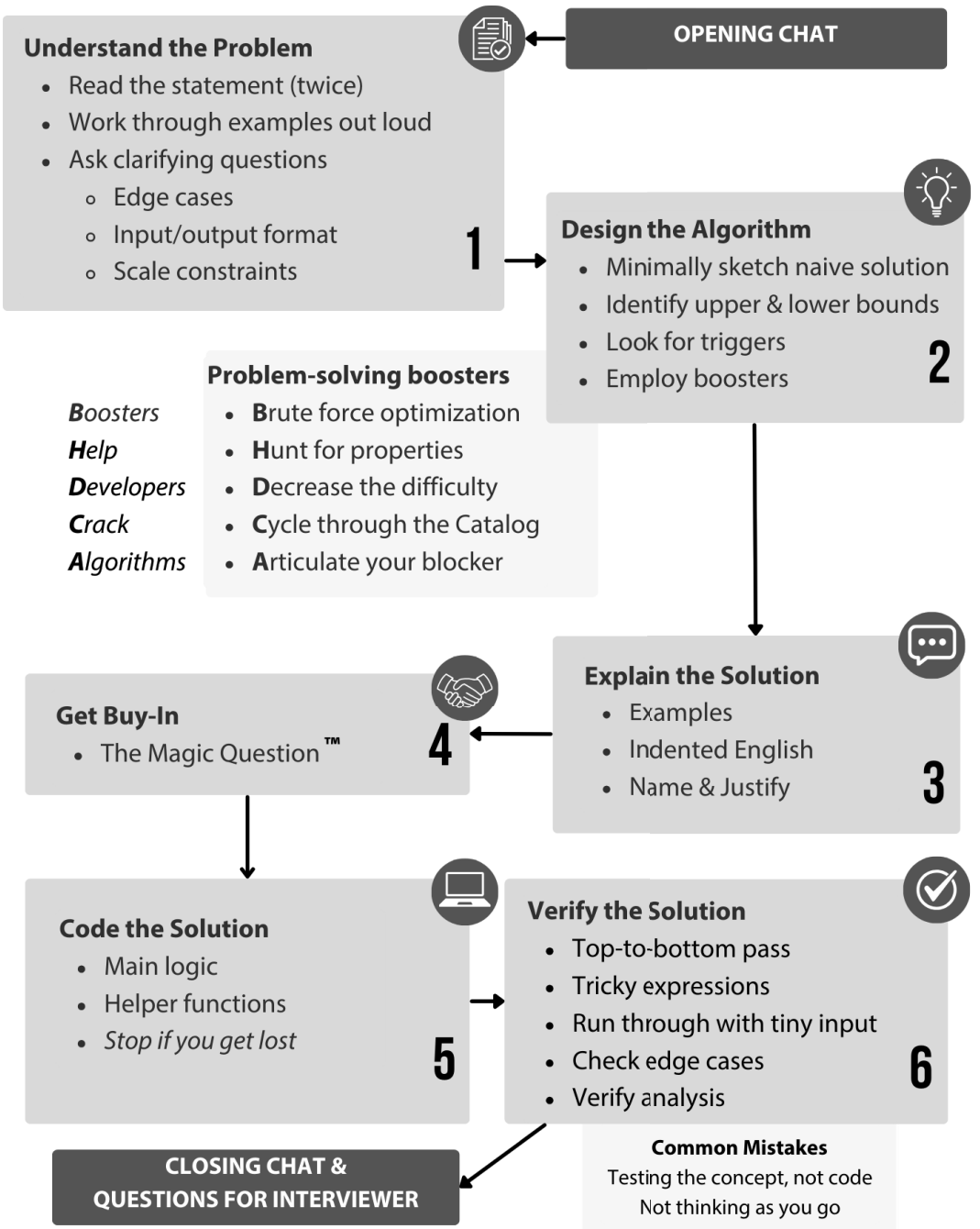
BEYOND
CRACKING
the
CODING INTERVIEW

PASS TOUGH CODING INTERVIEWS,
GET NOTICED, AND NEGOTIATE SUCCESFULLY

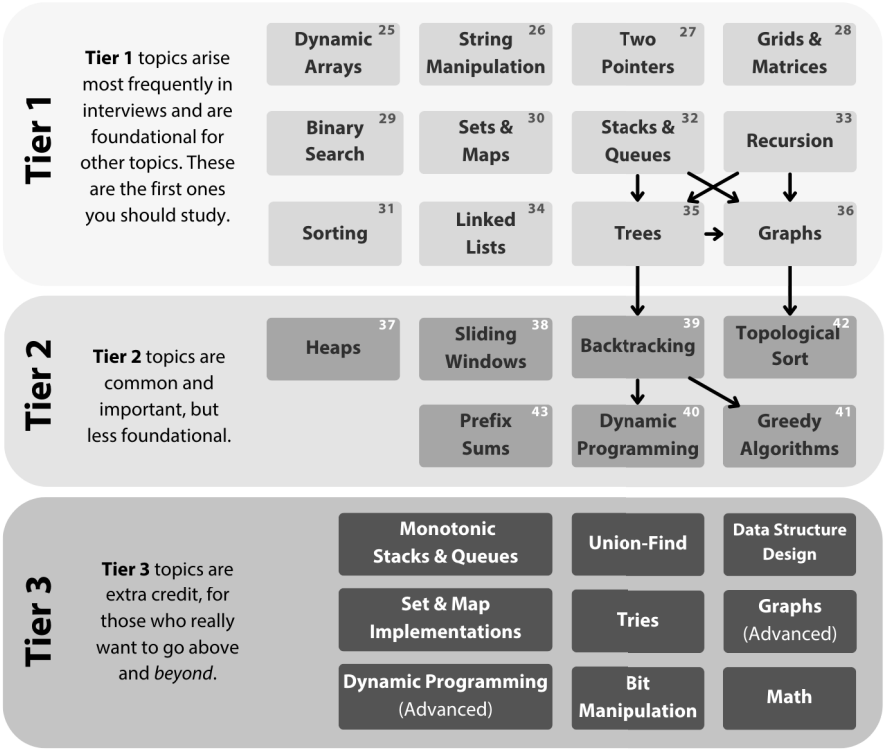
SNEAK PEEK

amazon.com/dp/195570600X

INTERVIEW CHECKLIST



STUDY PLAN



BOOSTERS

If boundary & trigger thinking don't point you to the right approach, start with the brute force

Brute Force Optimization
- Preprocessing Pattern
- Data Structure Pattern
- Skip Unnecessary Work

1

If you need a new approach...

Hunt for Properties
- DIY
- Case Analysis
- Reverse Engineer the Output Pattern
- Sketch a Diagram
- Reframe the Problem

2

If you can't find any approach...

Decrease the Difficulty
- Tackle an Easier Version
- Break Down the Problem

3

Solution might be in your blindspot

Cycle Through the Catalog
- *Think: Could ___ be useful?*

4

If you are still stuck...

Articulate Your Blocker
- Don't Say "Hint"
- Show Your Work

5

Cracking the Coding Interview
189 Programming Questions and Solutions

Cracking the PM Career
The Skills, Frameworks, and Practices to Become a Great Product Manager

Cracking the PM Interview
How to Land a Product Manager Job in Technology

Cracking the Tech Career
Insider Advice on Landing a Job at Google,
Microsoft, Apple, or Any Top Tech Company

BEYOND CRACKING *the* CODING INTERVIEW

GAYLE L. MCDOWELL
MIKE MROCZKA
ALINE LERNER
NIL MAMANO

CareerCup, LLC
Palo Alto, CA

BEYOND CRACKING THE CODING INTERVIEW

Copyright © 2025 by CareerCup.

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from the author or publisher, except by a reviewer who may quote brief passages in a review.

Published by CareerCup, LLC, Palo Alto, CA. Compiled Jun 3, 2025.

For more information, or to enquire about bulk or university copies, contact support@careercup.com.

Please report bugs or issues at beyondctci.com.

978-1955706001 (ISBN 13)

To my favorite coders, Davis and Tobin—
Gayle

To my dog, my wife, and our readers (and not necessarily in that order)—
Mike

To my two wonderful kids (or if I have more, then whichever two are the most wonderful)—
Aline

Als meus pares—
Nil

WHAT'S INSIDE

Get $50 Off on Mock Interviews

Practice anonymously on interviewing.io with FAANG interviewers: bctci.co/discount-X3A4

You can access all of our online materials and bonus chapters here:

bctci.co

Talk with the authors, get help if you're stuck, and geek out with us on Discord.

bctci.co/discord

TECHNICAL README

This chart represents how we see the landscape of interview questions:

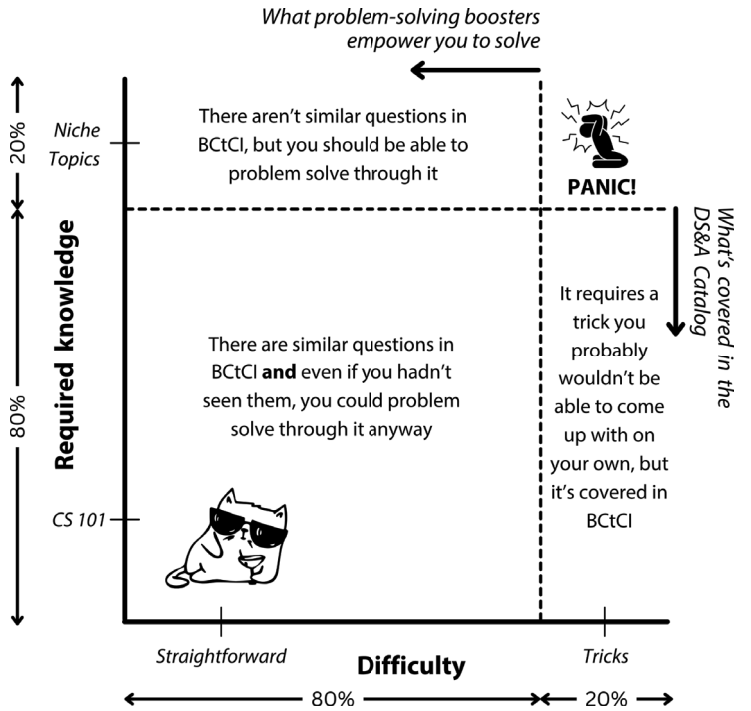


Figure 1. Landscape of Interview Questions, for someone who prepares with BCtCI.

Our goal with this book is twofold:

1. Teach you the 80% most common topics and ideas used in interview problems. That's what Part VII: The Catalog of Technical Topics, is all about. The remaining 20% are *niche topics*.

2. Teach you problem-solving strategies so you can figure out 80% of questions on your own, even if you haven't seen the idea before. This is what the *problem-solving boosters* (Chapter 24: Problem-Solving Boosters, pg 249) are for. The remaining 20% of questions rely on *tricks* (ideas that are really hard to come up with on your own if you haven't seen them before).[1]

Combining these two, after going through the book, you should be able to tackle all but `20% * 20% = 4%` of questions, which are those based on niche topics *and* requiring tricks. But if that happens, you've been truly unlucky.

THE PRINCIPLES AND THE CATALOG

Besides solving problems, we want to help you practice effectively and know how to navigate an interview setting. This is covered in Part VI: Principles of Coding Interviews (pg 166). It includes:

- **Study Plan:** A detailed study plan for how to practice using this book's materials.
- **Universal Rubric:** How you're evaluated by interviewers.
- **Interview Checklist**: Breaking down each step you should take in a coding interview.
- **Big-O Analysis:** In-depth coverage of the "language" of technical interviews.
- **Problem-Solving Strategies:** Boundary thinking, trigger thinking, and problem-solving boosters.

The second part is a **Catalog** of data structures and algorithms topics. We've broken the technical topics into tiers, with Tier 1 being the highest priority.

- **Tier 1:** Essential topics from sets & maps to trees and graphs.
- **Tier 2:** Intermediate topics like heaps, sliding windows, and prefix sums.
- **Tier 3:** Niche (online-only) topics that didn't warrant a spot in the physical book because they don't come up that often (this is where we enter the niche 20% territory). The online-only chapters can be found at bctci.co/bonus.

Chapter 18: How to Practice (pg 170) should be your entry point to the rest of the book.

TOPICS, RECIPES, AND REUSABLE IDEAS

There are three related concepts you'll find as you peruse the Catalog: topics, reusable ideas, and recipes. Here's a quick definition to keep them straight:

- **Topic:** A chapter from the Catalog, like Binary Search.
- **Reusable Idea**: A coding idea that can typically be used across problems (and even across topics). They are tactical tips worth remembering, such as "pass indices, not strings in recursive code to avoid using extra space." You'll typically find them next to the first problem where they are used (look for the ♻ icon).
- **Coding Recipe:** A pseudo-code template related to a specific topic that can be used as a building block to solve similar problems with small tweaks.

Questions, comments, or bugs? Report bugs at bctci.co/bugs or geek out with the authors on Discord: bctci.co/discord.

1 Our mantra? If you encounter something once, it's a trick; if you encounter it repeatedly, it's a tool.

CHAPTER 29

BINARY SEARCH

AI interviewer, replays, and more materials for this chapter at bctci.co/binary-search

▶ ***Prerequisites: None***

When it comes to binary search, software engineers are split in half: One camp thinks it's too basic to be an interview question, and the other dreads it because they always mess up the index manipulation.

The first group overlooks the fact that binary search has many uses beyond the basic "find a value in a sorted array." Far from it, binary search has many non-obvious applications, which we'll cover in this chapter. For the second group, we'll provide a recipe focusing on simplicity and reusability across applications—even the unconventional ones we just foreshadowed.

BINARY SEARCH IS EASY TO MESS UP

Let's start with something simple—the classic binary search setting—and then build up to harder problems.

PROBLEM 29.1 **SEARCH IN SORTED ARRAY**

Given a sorted array of integers, `arr`, and a target value, `target`, return the target's index if it exists in the array or `-1` if it doesn't.

▶ **Example:** `arr = [-2, 0, 3, 4, 7, 9, 11], target = 3`
Output: `2`.

▶ **Example:** `arr = [-2, 0, 3, 4, 7, 9, 11], target = 2`
Output: `-1`.

SOLUTION 29.1 **SEARCH IN SORTED ARRAY**

We assume most engineers are familiar with the basic premise of binary search: two pointers move inward from the ends of a sorted array, closing in on the target by checking if the midpoint is too small or too large.

Pop quiz! Here is an *attempted* solution, but it has a bug. Can you spot it?

```
1  def BUGGED_binary_search(arr, target): # DON'T USE IN INTERVIEWS!
2    l, r = 0, len(arr)
3    while l <= r:
4      mid = (l + r) // 2
5      if mid == target:
6        return mid
7      if target > mid:
8        r = mid+1
9      else:
10       l = mid-1
11   return -1
```

Check the solution in the footnote.[1] Regardless of what you found, the point is that it is easy to miss errors in a binary search implementation.

Here is one way to do it correctly:

```
def binary_search(arr, target):
  n = len(arr)
  if n == 0:
    return -1
  l, r = 0, n - 1
  if arr[l] >= target or arr[r] < target:
    if arr[l] == target:
      return 0
    return -1
  while r - l > 1:
    mid = (l + r) // 2
    if arr[mid] < target:
      l = mid
    else:
      r = mid
  if arr[r] == target:
    return r
  return -1
```

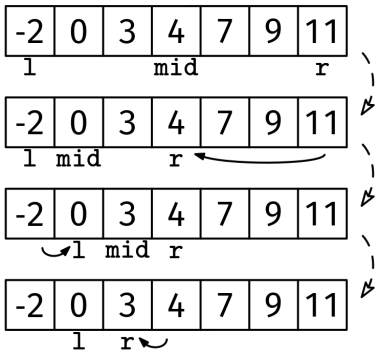


Figure 1. Binary search for target 3.

We'll punt on breaking down this solution until we talk about our transition-point recipe (pg 330).

BINARY SEARCH HAS SURPRISING APPLICATIONS

Imagine that your bike had gotten stolen, and your only chance of getting it back hinged on your ability to explain binary search to a police officer. That is the very situation Tom Whipple, a science journalist, found himself in.[2]

The rack from which the bike went missing was right under a security camera, but the police told him they didn't have the resources to watch many hours of footage just to identify when the bike was stolen. Tom explained that this wouldn't be necessary: they could skip ahead to the middle of the video and check if the bike was still there. If it was, the bike was stolen during the latter half; if not, it was stolen earlier. This could be repeated to quickly narrow down the time of the crime.

In the end, the thief was never caught—the footage was too grainy. Regardless, the story showcases an unconventional use of binary search. We can formalize it into an interview question:

PROBLEM 29.2 CCTV FOOTAGE

You are given an API called `is_stolen(t)` which takes a timestamp as input and returns `True` if the bike is missing at that timestamp and `False` if it is still there. You're also given two timestamps, `t1` and `t2`, representing when you parked the bike and when you found it missing. Return the timestamp when the bike was first missing, minimizing the number of API calls. Assume that `0 < t1 < t2`, `is_stolen(t1)` is `False`, and `is_stolen(t2)` is `True`.

1 We were not completely honest—there isn't just one bug; there are closer to six, depending on how you count them. (1) `r` is initialized out of bounds, (2 & 3) we check `mid` instead of `arr[mid]` (twice!), (4) we update `r` when we should be updating `l`, (5) `l` should be set to `mid+1`, not `mid-1`, and (6) `r` should be set to `mid-1`, not `mid+1`.

2 https://www.thetimes.com/article/i-have-owned-11-bikes-this-is-how-they-were-stolen-d3r553gx3

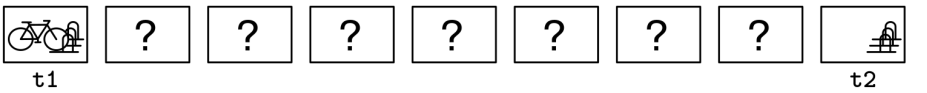


Figure 2.

SOLUTION 29.2 CCTV FOOTAGE

This problem is quite different from Problem 29.1: Search In Sorted Array (pg 326): it doesn't have an array input, and we don't have a target value. In fact, if we tried to use the same binary search code from earlier, we'd have to change almost every line in the algorithm before it would work correctly. That many tweaks make it easy to reintroduce bugs. Nonetheless, we can still use binary search because the range of possible answers can be broken down into two **regions**: (1) **before** the bike was stolen and (2) **after** it was stolen. We are searching for the **transition point** from 'before' to 'after':

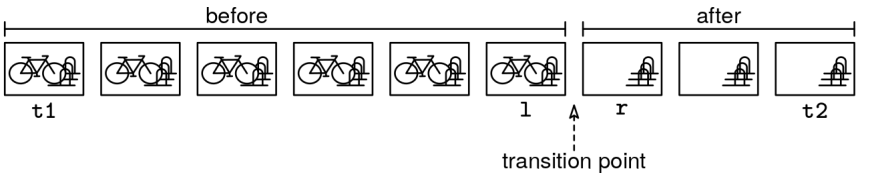


Figure 3.

Initially, we don't know where the transition point is, but we can binary search for it:

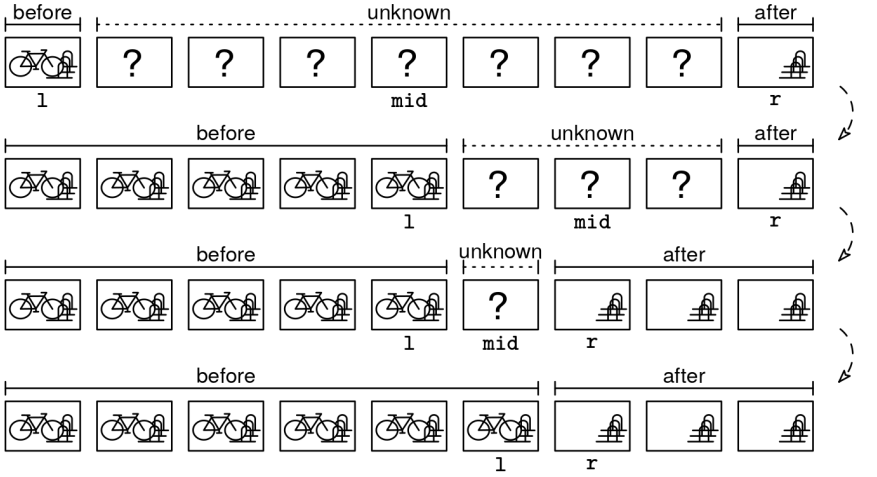


Figure 4.

```
1  def is_before(val):
2    return not is_stolen(val)
3
4  def find_bike(t1, t2):
5    l, r = t1, t2
6    while r - l > 1:
7      mid = (l + r) // 2
8      if is_before(mid):
9        l = mid
10     else:
```

```
11          r = mid
12      return r
```

At the beginning, `l` is in the 'before' region, and `r` is in the 'after' region. From there, `l` never leaves the 'before' region and `r` never leaves the 'after' region, but they end up next to each other: at the end, `l` is the last 'before' and `r` is the first 'after.'

Here is the kicker: **every binary search solution can be reframed as finding a transition point.** For instance, Problem 29.1: Search In Sorted Array can be reframed as finding the "transition point" from elements smaller than `target` to elements greater than or equal to `target`.

If we learn a recipe for finding transition points, we'll be able to use it for every binary search problem. We don't need specialized recipes for various problem types.

TRANSITION-POINT RECIPE

In an important interview—with tensions mounting and anxiety running high—you are not working at total capacity. We joke that you are ~20% dumber than during practice. To counter this, it helps to have a recipe you know well for tricky algorithms like binary search. A good recipe should be easy to remember, have straightforward edge cases, and make it easy to avoid off-by-one errors.

RECIPE 1. TRANSITION-POINT RECIPE

```
transition_point_recipe()
  define is_before(val) to return whether val is 'before'
  initialize l and r to the first and last values in the range
  handle edge cases:
    - the range is empty
    - l is 'after'  (the whole range is 'after')
    - r is 'before' (the whole range is 'before')
  while l and r are not next to each other (r - l > 1)
    mid = (l + r) / 2
    if is_before(mid)
      l = mid
    else
      r = mid
  return l (the last 'before'), r (the first 'after'), or something else,
         depending on the problem
```

The point of the initialization and the initial edge cases is to get to a setting that looks like the first row of Figure 4: `l` *must* be in the 'before' region, and `r` *must* be in the 'after' region. The three edge cases are designed to ensure this.

Once we get to that point, the main `while` loop is *the same for every problem*—no tweaking needed!

The loop has the following **invariants**, which are guarantees that make our lives easier:

- From start to end, `l` is in the 'before' region, and `r` is in the 'after' region. They are never equal and never cross over.[3]
- The midpoint is always strictly between `l` and `r` (`l < mid < r`), which guarantees we always make progress (we don't need to worry about infinite loops).
- When we exit the loop, `l` and `r` are always next to each other.

3 If we were a little more willing to buck conventions, we'd rename from `l` and `r` to b and a, since they always map to 'before' and 'after' values. However, you might get odd looks from an interviewer if you do this!

Something that is typically tricky with binary search is the exit condition of the loop. Here, we keep going until `l` and `r` are next to each other (i.e., until the 'unknown' region in Figure 4 is empty), which happens when `r - l` is `1`. That's why the condition says `r - l > 1`.[4]

Another tricky part is knowing what to return. With this recipe, we just need to reason about the transition point: do we need the final 'before' or the first 'after'?

We recommend starting by defining the `is_before()` function. Keep in mind that, for binary search to work, we must define it in such a way that the search range is *monotonic*: all the 'before' elements must appear before all the 'after' elements. That's why binary search doesn't work on unsorted arrays.

Revisiting Solution 29.1

Here is how we applied the recipe in Solution 29.1: we defined the 'before' region as the elements < `target`, and the 'after' region as the elements ≥ `target`.

In the initialization, we have the three edge cases from the recipe to ensure that `l` is 'before' and `r` is 'after':

```
if n == 0:
  return -1
l, r = 0, n - 1
if arr[l] >= target or arr[r] < target:
  if arr[l] == target:
    return 0
  return -1
```

The `while` loop is just like the recipe, except that we didn't factor out `is_before()` into a helper function:

```
while r - l > 1:
  mid = (l + r) // 2
  if arr[mid] < target:
    l = mid
  else:
    r = mid
```

Finally, when we find the transition point, we consider what that means: `l` is at the largest value smaller than the target, and `r` is at the smallest value greater than or equal to the target. So, if the target is in the array at all, it must be at index `r`.

```
if arr[r] == target:
  return r
return -1
```

What to do at the end depends on how we define the 'before' region. We could have also defined 'before' as "less than or equal to the target," in which case, at the end, we would have to check the element at `l` instead of `r`.

This recipe is a bit like a one-size-fits-all pair of socks. While more concise (but less reusable) implementations may exist for some problems, there is value in needing only one easy-to-remember recipe.

TRANSITION-POINT PROBLEM SET

For each of the following problems:

- **Reframe** it as finding a transition point by defining 'before' and 'after' regions.
- Find the location of `l` and `r` after finding the transition point for the given example input.

4 We could have also written this in other ways, like `r > l + 1`. One way to remember the formula for the number of elements between `l` and `r`, `r-l-1`, is that it looks like a sleepy cat.

- Identify what to return after finding the transition point.

You don't need to code anything yet—focus on the transition logic.

QUESTION 1 **GIT COMMITS**

Find the first commit that fails a test in a sequence of Git commits. We know the test was passing for every commit until it started failing at some point.

```
["pass", "pass", "pass", "pass", "fail", "fail", "fail"]
```

QUESTION 2 **SQUARED TARGET**

Given a sorted array of positive integers and a target value, find the largest number in the array that can be squared and still be less than or equal to the target, if any. Return the number (not its index).

```
[2, 3, 4, 5, 6, 7, 8, 11, 20, 21, 23, 25, 25], target = 36
```

QUESTION 3 **FIRST NON-NEGATIVE**

Return the index of the first non-negative integer in a sorted array (duplicates allowed), if any.

```
[-21, -15, -9, -5, -5, -1, -1, 0, 0, 4, 7, 12, 21]
```

QUESTION 4 **FIRST 'P'**

Find the first word that begins with 'p' in an array of words in dictionary order, if any.

```
["apple", "banana", "peach", "strawberry"]
```

QUESTION 5 **NEAREST ELEMENT**

In a sorted array of integers (duplicates allowed), find the last occurrence of a given target value. If the target does not exist, return the index of the next closest value (it could be smaller or larger than the target).

```
[1, 3, 5, 6, 7, 7, 8, 11, 13, 21], target = 7
```

QUESTION 6 **DECK CUT**

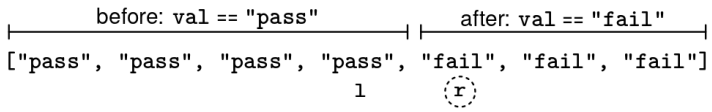
You're given an array that contains each number from 1 to 52 once, representing a deck of playing cards. The deck started in order, but it was then "cut," meaning that a random number of cards was taken from the top (the front of the array) and moved as a block to the bottom (the back of the array). Determine the index where you must "cut" the deck again to return to sorted order (that is, the index with the 52).

```
[36, 37, 38, ..., 50, 51, 52, 1, 2, 3, ..., 33, 34, 35]
```

PROBLEM SET SOLUTIONS

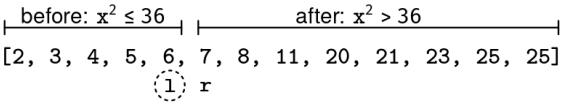
In the solutions below, we circled which of the two pointers we should return at the end.

ANSWER 1 **GIT COMMITS**

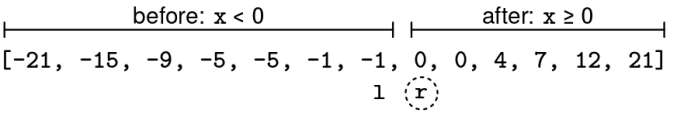


ANSWER 2 **SQUARED TARGET**

We should return `arr[l]`, since the last number in the 'before' region is the largest number that still works.

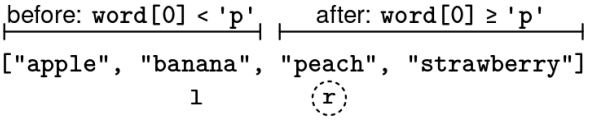


ANSWER 3 **FIRST NON-NEGATIVE**



Including `0` in the 'before' region would be a mistake: if there are multiple zeros, `l` would point to the last one, but the goal is to return the first one.

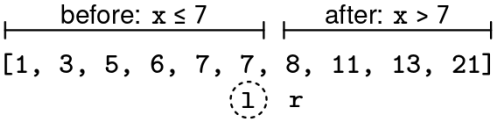
ANSWER 4 **FIRST 'P'**



The 'before' region consists of words that start with a–o and the 'after' region consists of words that start with p–z. If there are words that start with 'p', the first one will be at index r.

Including words that start with 'p' in the 'before' region would be incorrect: if we inserted another word starting with 'p,' like "pear," `l` would point to the last word starting with 'p' rather than the first one.

ANSWER 5 **NEAREST ELEMENT**



Post-processing requires a bit of thought. If the target is in the array, it will be at `l`. We can peek at `arr[l]` and return `l` if it is the target. Otherwise, we need to find the closest value to it, which could be at `l` or `r`. We return either `l` or `r`, based on whether `arr[l]` or `arr[r]` is closer.

ANSWER 6 **DECK CUT**

Trick question! This doesn't require binary search as the answer is always `52 - deck[0]`. Still, we could find the transition point from 52 back to 1 with a binary search. The 'before' region would be numbers ≥ `deck[0]`. The `l` pointer would end up at the 52 and the `r` pointer at the 1. We would return `l`.

VALIDATION & DRAWING ADVICE

Visualizing the binary search in an interview is helpful both for you and your interviewer. Instead of trying to verbally explain what you're doing, show them in the shared editor. Our suggestions are similar to the Two Pointers chapter (pg 296):

- Write each pointer (`l`, `r`, and `m` for the midpoint) on its own line so you can move them independently with ease.
- Writing indices on the top of the array makes it faster to do midpoint calculations.
- You can also draw the transition point between 'before' and 'after.'

Instead of	Try

`[1, 2, 2, 3, 3, 4, 5, 8, 8]`
`^ ^ ^`
`left mid right`
	`0 1 2 3 4 5 6 7 8`
	`[1 2 2 3 3 4|5 8 8]`
	`l`
	`r`
	`m`
	`m = (0+8)/2`

So, which inputs should you validate and visualize? Consider the following edge cases, when applicable:

- The range is empty.
- The range only has 'before' elements.
- The range only has 'after' elements.
- The target is not in the array.
- The target is in the array multiple times.

ANALYSIS

> But officer, it is O(log n)!

In the Big O analysis chapter, we defined $\log_2(n)$ as roughly the number of times we need to halve a number to reach 1. Binary search halves the search range at each step, so binary search converges in `O(log n)` iterations, where `n` is the size of the range (e.g., `t2 - t1` in Problem 29.2: CCTV Footage (pg 327).[5]

Some binary search implementations stop early when `arr[mid] == target`. For simplicity, our recipe doesn't have that, which means that it takes `O(log n)` time even in the best case. That's fine—we mainly care about the worst case.

Don't forget to factor in the time it takes to compute `is_before()` in the runtime calculation if it is not constant!

Finally, the extra space is `O(1)`. Binary search can also be implemented recursively, in which case the extra space increases to `O(log n)` for the call stack.

BINARY SEARCH PROBLEM SET

Try these problems with AI Interviewer: bctci.co/binary-search-problem-set-1

Let's tackle some problems that require creative approaches for using binary search. The transition-point recipe should prove useful!

PROBLEM 29.3 VALLEY BOTTOM

A *valley-shaped* array is an array of integers such that:

- it can be split into a non-empty prefix and a non-empty suffix,
- the prefix is sorted in decreasing order,

5 See page 219 for why we "drop" the base of the logarithm in big O notation.

- the suffix is sorted in increasing order,
- all the elements are unique.

Given a valley-shaped array, `arr`, return the smallest value.

▸ **Example:** `arr = [6, 5, 4, 7, 9]`
Output: `4`

▸ **Example:** `arr = [5, 6, 7]`
Output: `5. The prefix sorted in decreasing order is just [5].`

▸ **Example:** `arr = [7, 6, 5]`
Output: `5. The suffix sorted in increasing order is just [5].`

PROBLEM 29.4 2-ARRAY 2-SUM

You are given two non-empty arrays of integers, `sorted_arr` and `unsorted_arr`. The first one is sorted, but the second is not. The goal is to find one element from each array with sum `0`. If you can find them, return an array with their indices, starting with the element in `sorted_arr`. Otherwise, return `[-1, -1]`. Use `O(1)` *extra space* and do not modify the input.

▸ **Example:** `sorted_arr = [-5, -4, -1, 4, 6, 6, 7]`
`unsorted_arr = [-3, 7, 18, 4, 6]`
Output: `[1, 3]. We can use -4 from the sorted array and 4 from the unsorted array.`

PROBLEM 29.5 TARGET COUNT DIVISIBLE BY K

Given a sorted array of integers, `arr`, a target value, `target`, and a positive integer, k, return whether the number of occurrences of the target in the array is a multiple of k.

▸ **Example:** `arr = [1, 2, 2, 2, 2, 2, 2, 3]`
`target = 2, k = 3`
Output: `True. 2 occurs 6 times, which is a multiple of 3.`

▸ **Example:** `arr = [1, 2, 2, 2, 2, 2, 2, 3]`
`target = 2, k = 4`
Output: `False. 2 occurs 6 times, which is not a multiple of 4.`

▸ **Example:** `arr = [1, 2, 2, 2, 2, 2, 2, 3]`
`target = 4, k = 3`
Output: `True. 4 occurs 0 times, and 0 is a multiple of any number.`

PROBLEM 29.6 RACE OVERTAKING

You are given two arrays of positive integers, p1 and p2, representing players in a racing game. The two arrays are sorted, non-empty, and have the same length, n. The `i`-th element of each array corresponds to where that player was on the track at the `i`-th second of the race. We know that:

1. player 1 started ahead (`p1[0] > p2[0]`),
2. player 2 overtook player 1 *once*, and
3. player 2 remained ahead until the end (`p1[n - 1] < p2[n - 1]`).

Assume the arrays have no duplicates, and that `p1[i] != p2[i]` for any index.

Return the index at which player 2 overtook player 1.

▸ **Example:** `p1 = [2, 4, 6, 8, 10],`
`p2 = [1, 3, 5, 9, 11]`

Output: 3

PROBLEM 29.7 SEARCH IN SORTED GRID

You're given a 2D grid of integers, `grid`, where each row is sorted (without duplicates), and the last value in each row is smaller than the first value in the following row. You are also given a target value, `target`. If the target is in the grid, return an array with its row and column indices. Otherwise, return `[-1, -1]`.

▶ **Example:**
```
target = 4
grid = [[1, 2, 4, 5],
        [6, 7, 8, 9]]
```
Output: `[0, 2]. The number 4 is found in row 0 column 2.`

▶ **Example:**
```
target = 3
grid = [[1, 2, 4, 5],
        [6, 7, 8, 9]]
```
Output: `[-1, -1]`

PROBLEM 29.8 SEARCH IN HUGE ARRAY

We are trying to search for a target integer, `target`, in a sorted array of positive integers (duplicates allowed) that is too big to fit into memory. We can only access the array through an API, `fetch(i)`, which returns the value at index `i` if `i` is within bounds or `-1` otherwise. Using as few calls to the API as possible, return the index of the target, or `-1` if it does not exist. If the target appears multiple times, return any of the indices. There is no API to get the array's length.

PROBLEM SET SOLUTIONS

SOLUTION 29.3 VALLEY BOTTOM

This problem shows that binary search can be used even if the input array is not monotonically sorted.

Intuitively, we want to define the 'before' region as the descending prefix and the 'after' region as the ascending suffix. The tricky part is that an array like `[6, 5, 4, 7, 9]` can be formed in two ways:

- With a descending prefix `[6, 5, 4]` and an ascending suffix `[7, 9]`.
- With a descending prefix `[6, 5]` and an ascending suffix `[4, 7, 9]`.

We need a clear rule for how to define `is_before()`. For instance, if we want the 4 to be in the 'before' region, we can say that a number is in the 'before' region if (a) it is the first element, or (b) it's smaller than the previous element.

This definition is workable, but according to it, an array like `[7, 6, 5]` only contains 'before' elements; we need to check for that case during preprocessing.

With this definition, elements in the 'after' region are always greater than the previous element, so the smallest value in the entire array will be the last one in the 'before' region.

```
1  def valley_min_index(arr):
2    def is_before(i):
3      return i == 0 or arr[i] < arr[i-1]
4    l, r = 0, len(arr)-1
5    if is_before(r):
6      return arr[r]
7    while r - l > 1:
8      mid = (l + r) // 2
```

```
    if is_before(mid):
      l = mid
    else:
      r = mid
  return arr[l]
```

Interestingly, in the variation of this problem where we allow duplicates in the input, binary search does not work: if `mid` lands on a value that is the same as the previous one and the next one, we can't tell if we are in the descending prefix or the ascending suffix.[6]

SOLUTION 29.4 2-ARRAY 2-SUM

This problem, which is a variant of the classic 2-sum problem, shows binary search as a building block of a broader algorithm.

Let `n1` be the length of the sorted array and `n2` the length of the unsorted array. The "only `O(1)` extra space" constraint means that we can't use a map-based solution, which would take `O(n1)` or `O(n2)` space.

Instead, we can iterate through the numbers in the unsorted array and, for each one, binary search for its inverse in the sorted array. The total runtime will be `O(n2 * log n1)`.

```
def two_array_two_sum(sorted_arr, unsorted_arr):
  for i, val in enumerate(unsorted_arr):
    idx = binary_search(sorted_arr, -val)
    if idx != -1:
      return [idx, i]
  return [-1, -1]
```

We omit the binary search step because it is the same as Solution 1.

SOLUTION 29.5 TARGET COUNT DIVISIBLE BY K

The key is to find the first *and* last occurrence of the target, `first` and `last`. If present, the number of occurrences of the target is `last - first + 1`. We can check if this number is multiple of k.

If the target is in the array, we can find `first` and `last` with a binary search for each:

- one defining 'before' as '`< target`' and returning `r`,
- one defining 'before' as '`< target + 1`' and returning `l`.

The runtime is `O(2*log n) = O(log n)`.

SOLUTION 29.6 RACE OVERTAKING

We say an index is 'before' if player 2 has not overtaken player 1 yet. That is:

```
def is_before(i):
  return p1[i] > p2[i]
```

According to the statement, index `0` is 'before' and index `n-1` is 'after,' so we don't need to worry about the initial edge cases. We just need to find the transition point and return `r`.

SOLUTION 29.7 SEARCH IN SORTED GRID

We could solve this problem in two steps:

1. Binary search over the rows to find a single row that may contain the target.
2. Binary search over the row.

6 In fact, for this variant, we cannot do better than `O(n)` time. The array could consist of all 1's and a single `0`, which could be anywhere and can only be found with a linear scan.

While this works, a **trick** that makes the implementation easier is to imagine that we "flatten" the grid into a single, long array with all the rows consecutively:

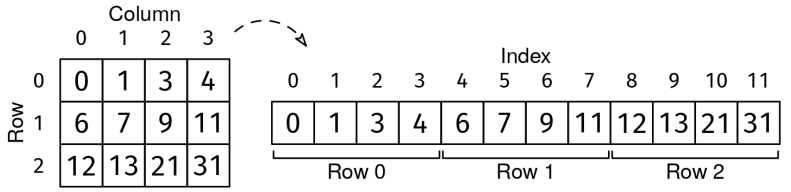


Figure 5.

This would be a sorted array with `R*C` elements. We can binary search over this 'flattened-grid' array without actually creating it. We'd start with `l = 0` and `r = R*C - 1`. To define the `is_before()` function, we must map the 'flattened-grid' array index to the actual grid coordinates on the fly:

```
def is_before(grid, i, target):
  num_cols = len(grid[0])
  row, col = i // num_cols, i % num_cols
  return grid[row][col] < target
```

Once we find the transition point, we have to map `r` back to grid coordinates to check if the `target` is there and return them.

REUSABLE IDEA: GRID FLATTENING

If we want to iterate or search through a grid with dimensions `R`x`C` 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]`.

SOLUTION 29.8 SEARCH IN HUGE ARRAY

Leveraging the **break down the problem** booster, we can break the problem into two. One problem is quickly finding the target in a huge array. Binary search is an obvious choice here, but it leads to the second problem: our left pointer can start at zero, but where do we start our right pointer without knowing the length of the array?

A silly way to solve this would be to keep trying one index after another until the API eventually returns `-1`. Instead, we can double our index at each step. If the length is `n`, we'll reach it in approximately $\log_2(n)$ steps. The rest of the problem is a straightforward application of the transition-point recipe.

```
def find_through_api(target):
  def is_before(idx):
    return fetch(idx) < target
  l, r = 0, 1
  # Step 1: Get the rightmost boundary
  while fetch(r) != -1:
    r *= 2
  # Step 2: Binary search
  # ...
```

The total runtime is O(log n), where n is the size of the huge array.

REUSABLE IDEA: 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, pg 340).

GUESS-AND-CHECK TECHNIQUE

> Since I was unable to come up with any approach,
> I knew the solution was going to be Binary Search.
>
> Anonymous Leetcode User, 2023

As the stolen bike story illustrates, binary search is often used in problems where it is not an obvious choice. Now that we have a solid recipe for any binary search problem, we will discuss the **guess-and-check** technique, which allows us to use binary search on many optimization problems.

Recall that an optimization problem is one where you are asked to find some minimum or maximum value, subject to some constraint. For example, consider the following problem:

PROBLEM 29.9 MIN-SUBARRAY-SUM SPLIT

Given a non-empty array with n positive integers, `arr`, and a number k with $1 \le k \le n$, the goal is to split `arr` into k non-empty subarrays so that the largest sum across all subarrays is minimized. Return the largest sum across all k subarrays after making it as small as possible. Each subarray must contain at least one value.

- **Example:** `arr = [10, 5, 8, 9, 11], k = 3`
  Output: `17. There are six ways of splitting the array into three subarrays. The optimal split is: [10, 5], [8, 9], and [11]. The largest sum among the three subarrays is 17.`
- **Example:** `arr = [10, 10, 10, 10, 10], k = 2`
  Output: `30.`

SOLUTION 29.9 MIN-SUBARRAY-SUM SPLIT

This is an optimization problem because we have a goal and a constraint: we are trying to minimize the largest subarray sum, subject to having at most k subarrays. Without the constraint, we would just put every element in its own subarray.

A naive solution that tries every way of splitting the array into k subarrays would take exponential time.[7] There is a dynamic programming solution that takes O(n*k) time (pg 572).

Here, we'll use a different approach. Given a value, `max_sum`, we can ask:

> Is there a way to split `arr` into k subarrays such that every subarray has sum at most `max_sum`?

- For `max_sum < max(arr)`, the answer is "no" (some numbers are too big to be in a subarray, even by themselves).

7 You need to choose k out of n-1 possible splitting points, so there are (n-1 choose k) options, which is $O((n-1)^k) = O(n^k)$ for any constant value of k. If k is n/2, the number becomes exponential on n ($O(2^n/\sqrt{n})$ to be exact, but you don't need to worry about where that formula comes from). Once k gets larger than n/2, the number of possibilities starts decreasing (if we are picking more than half the points, we can think about picking the points **not** to split at, of which there are fewer than n/2).

- For `max_sum == sum(arr)`, the answer is "yes" (any split will do).

We can **binary search** for the transition point where the answer goes from "no" to "yes" with our **transition point recipe**. The value x corresponding to the first "yes" is the *value* of the optimal solution.

To implement our `is_before(max_sum)` function, we need to be able to compute the answer to the question. Thankfully, it is much easier than the original problem: we can grow each subarray up until the point where its sum would exceed `max_sum`. At that point, we start a new subarray, and so on. If we need more than k subarrays, the answer is "no." Otherwise, the answer is "yes."

```
# "Is it impossible to split arr into k subarrays, each with sum <= max_sum?"
def is_before(arr, k, max_sum):
  splits_required = get_splits_required(arr, max_sum)
  return splits_required > k

# Returns the minimum number of subarrays with a given maximum sum.
# Assumes that max_sum >= max(arr).
def get_splits_required(arr, max_sum):
  splits_required = 1
  current_sum = 0
  for num in arr:
    if current_sum + num > max_sum:
      splits_required += 1
      current_sum = num  # Start a new subarray with the current number.
    else:
      current_sum += num
  return splits_required

def min_subarray_sum_split(arr, k):
  l, r = max(arr), sum(arr)  # Range for the maximum subarray sum.
  if not is_before(arr, k, l):
    return l
  while r - l > 1:
    mid = (l + r) // 2
    if is_before(arr, k, mid):
      l = mid
    else:
      r = mid
  return r
```

Let S be the sum of `arr`. Binary search will take `O(log S)` steps to converge, and each `is_before()` check takes `O(n)` time. The total runtime is `O(n log S)`. Depending on whether `O(k)` or `O(log S)` is larger, DP or binary search will be better. Neither dominates the other.

To recap, the **guess-and-check technique** involves narrowing in on the value of the optimal solution by **guessing** the midpoint and **checking** whether it's too high or too low. To start, we need lower and upper bounds for the value of the optimal solution (if the bounds are not obvious, exponential search can help).

For minimization problems (like Problem 29.9: Min-Subarray-Sum Split), there is often a transition point where smaller values do not satisfy the constraint, but larger values do. Conversely, for maximization problems, there is often a transition point where larger values do not satisfy the constraint, but smaller values do.

When should I use the guess-and-check technique?

We can try it when we have an optimization problem and finding the optimal value directly is challenging. Ask yourself:

> *"Is it easier to solve the yes/no version of the problem, where we just check if a given value (optimal or not) satisfies the constraint?"*
>
> Think of it like making a deal: You get to solve an easier problem (checking if a specific value satisfies the constraint), but you pay a 'logarithmic tax' in the runtime (to binary searching for the transition point).

We've seen the guess-and-check technique before, in the Boundary Thinking chapter with Problem 22.1: Tunnel Depth (pg 232). It is easier to binary search for the first depth where the tunnel doesn't reach than to try to compute the maximum depth directly.

BOUNDARY THINKING IN ACTION[8] INTERVIEW REPLAY

View Online:	bctci.co/binary-search-replay-1 @ 38:55 - end
The Question:	Return the maximum tunnel depth in a grid.
What You'll See:	The candidate chose a graph traversal after seeing the grid, and the interviewer and candidate discussed multiple solutions and how to avoid getting "tunnel" vision.
Who:	*Interviewer*: Software Engineer at Google *Candidate*: 7 years exp.

GUESS-AND-CHECK PROBLEM SET

Try these problems with AI Interviewer: bctci.co/binary-search-problem-set-2

PROBLEM 29.10 WATER REFILLING

We have an empty container with a capacity of `a` gallons of water and another container with a capacity of `b` gallons. Return how many times you can pour the second container full of water into the first one without overflowing. Assume that `a > b`.

- Constraint: You are not allowed to use the division operation, but you can use still divide by powers of two with the right-shift operator, `>>`. Recall that `x >> 1` is the same as `x // 2`.

▸ **Example:** `a = 18, b = 5`
Output: `3. After pouring 5 gallons three times, the first container will be at 15, and 5 more gallons would make it overflow.`

PROBLEM 29.11 MIN PAGES PER DAY

You have upcoming interviews and have selected specific chapters from BCtCI to read beforehand. Given an array, `page_counts`, where each element represents a chapter's page count, and the number of days, `days`, until your interview, determine the minimum number of pages you must read daily to finish on time. Assume that:

- You must read all the pages of a chapter before moving on to another one.[9]

8 I (Mike) am the interviewer in this particular interview. This question was the opener for the Boundary Thinking chapter and you can see a candidate make the same mistakes we discuss in that chapter and me walking through the boundary thinking mentality.

9 Hypothetically! It's fine to jump around chapters when reading *this actual book*.

- If you finish a chapter on a given day, you practice for the rest of the day and don't start the next chapter until the next day.
- `len(page_counts)` ≤ `days`.

▸ **Example:** `page_counts = [20, 15, 17, 10], days = 14`
Output:
```
5. We can read 5 pages daily and finish all chapters. At a maximum
of 5 pages per day, we spend:
   4 days on the first chapter.
   3 days on the second chapter.
   4 days on the third chapter (stopping when we finish early).
   2 days on the fourth chapter.
   In total, we spent 13 days reading 5 pages a day, which is the
   lowest amount we can read daily and still finish on time.
```

▸ **Example:** `page_counts = [20, 15, 17, 10], days = 5`
Output: `17`

PROBLEM 29.12 TIDE AERIAL VIEW

You are provided a series of aerial-view pictures of the same coastal region, taken a few minutes apart from each other around the time the tide rises. Each picture consists of an `n`x`n` binary grid, where `0` represents a part of the region above water, and `1` represents a part below water.

- The tide appears from the left side and rises toward the right, so, in each picture, for each row, all the `1`'s will be before all the `0`'s.
- Once a region is under water, it stays under water.
- All pictures are different.

Determine which picture shows the most even balance between regions above and below water (i.e., where the number of `1`'s most closely equals the number of `0`'s). In the event of a tie, return the earliest picture.

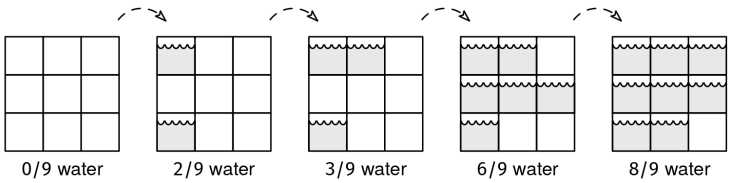


Figure 6. Example input for Problem 9. The empty cells are `0`'s and the cells with water are `1`'s.

▸ **Example:** `The pictures from Figure 6.`
Output:
```
2. The pictures at index 2 and 3 are equally far from having 50%
water. We break the tie by picking the earlier one, 2.
```

PROBLEM SET SOLUTIONS

SOLUTION 29.10 WATER REFILLING

Try to solve this problem manually for `a = 182983` and `b = 90`. Use a calculator if you want, just do not use the division operation. Done? How did you do it? Try to reverse engineer your process and map it to an algorithmic technique.

What you *definitely* didn't do is check sequential multiples of `90` until you reached `182983`. Most people make guesses in increasingly larger jumps until they find a guess that is too large. If we double the guess at

each time, this is *exponential search* (pg 338). Then, they start searching between their closest guess below and above the answer, closing in on the number—something we can do with binary search. Despite the problem not having clear triggers for binary search, it's a natural fit for this thought process.

```
def num_refills(a, b):
  # "Can we pour 'num_pours' times?"
  def is_before(num_pours):
    return num_pours * b <= a

  # Exponential search (repeated doubling until we find an upper bound).
  k = 1
  while is_before(k * 2):
    k *= 2

  # Binary search between k and k*2
  l, r = k, k * 2
  while r-l > 1:
    gap = r - l
    half_gap = gap >> 1  # Bit shift instead of division
    mid = l + half_gap
    if is_before(mid):
      l = mid
    else:
      r = mid
  return l
```

SOLUTION 29.11 MIN PAGES PER DAY

For a given value `daily_limit`, we pose the question:

> Can I finish all the chapters in time reading at most `daily_limit` pages a day?

We can **guess and check** for the answer to this question.

Since we can only finish one chapter per day, the maximum answer is the longest chapter (20 pages in the example). The minimum is 1 page per day. We can binary search between these bounds and simulate reading the guessed amount of pages per day. If the guess allows us to finish within the given days, we try a smaller number. If it takes too many days, we try a larger number.

```
def days_to_finish(page_counts, daily_limit):
  days = 0
  for pages in page_counts:
    days += math.ceil(pages / daily_limit)
  return days

def is_before(page_counts, daily_limit, days):
  return days_to_finish(page_counts, daily_limit) <= days
```

SOLUTION 29.12 TIDE AERIAL VIEW

We can binary search for the transition point where it goes from majority above water to majority underwater. The 'before' pictures are < 0.5 water, and the 'after' pictures are ≥ 0.5 water. The answer will be the last 'before' or the first 'after.'

However, if you constructed the `is_before()` function to just loop through the matrix, counting the number of cells underwater, you missed something! Besides doing a binary search across the range of pictures, we can

speed up the count of underwater cells by *also* doing a binary search on each row: the rows are monotonic, with all 1's followed by all 0's.

```
def get_ones_in_row(row):
  if row[0] == 0:
    return 0
  if row[-1] == 1:
    return len(row)

  def is_before_row(idx):
    return row[idx] == 1

  l, r = 0, len(row)
  while r - l > 1:
    mid = (l + r) // 2
    if is_before_row(mid):
      l = mid
    else:
      r = mid
  return r
```

```
def is_before(picture):
  water = 0
  for row in picture:
    water += get_ones_in_row(row)
  total = len(picture[0])**2
  return water/total < 0.5
```

Checking the number of ones in a row takes `O(log n)` time. Checking the number of ones in an entire grid takes `O(n log n)` time. The total time is `O(n log n log k)`, where k is the number of pictures.

BINARY SEARCH GONE WRONG INTERVIEW REPLAY

View Online:	bctci.co/binary-search-replay-2 @ 2:45 - 26:07:00
The Question:	Write an algorithm to compute the square root of a given non-negative number
What You'll See:	The candidate struggled to implement a working version of binary search, and each change led to further problems with the algorithm.
Who:	*Interviewer*: Software Engineer at Meta *Candidate*: 7 years exp.

CONCLUSIONS

Binary Search triggers: The input is a sorted array/string. The brute force involves repeated linear scans. We are given an optimization problem that's hard to optimize directly.

Keywords: sorted, threshold, range, boundary, find, search, minimum/maximum, first/last, smallest/largest.

Binary search is often a step or a possible optimization in more complicated algorithms. Binary search is so common that it can (and will) be seen alongside almost every other Catalog topic, like Graphs (Problem 36.9: First Time All Connected, pg 468), Sliding Windows (Chapter 38: Longest Repeated Substring, pg 523), and Greedy Algorithms (Problem 41.6: Time Traveler Max Year, pg 593).

The key idea in this chapter is that we can reframe every binary search problem as finding a transition point. This way, we only need one recipe for every scenario—the transition-point recipe (pg 329)—and we can focus our energy on more complicated parts of the code.

At this point, you should be ready to start adding binary search problems to your practice rotation. You can find the problems in this chapter and additional problems in the companion AI interviewer.

ONLINE RESOURCES

Online resources for this chapter include:

- A chance to try each problem in this chapter in AI Interviewer
- Interview replays that show specific mistakes people make with binary search problems
- Full code solutions for every problem in the chapter in multiple programming languages

Try online at bctci.co/binary-search.

CHAPTER 38

SLIDING WINDOWS

AI interviewer, replays, and more materials for this chapter at bctci.co/sliding-windows

▶ ***Prerequisites: None***

In this chapter, we will use the sliding window technique to tackle problems about finding or counting subarrays.[1]

We will use the following setting for problems throughout this chapter: a bookstore is looking at the number of book sales. The sales for each day are stored in an array of non-negative integers called `sales`. We say a *good day* is a day with at least `10` sales, while a *bad day* is a day with fewer than `10` sales. An interviewer could ask questions such as the following:

- Find the most sales in any 7-day period (Problem 1).
- Find the most consecutive days with no bad days (Problem 5).
- Find the longest period of time with at most 3 bad days (Problem 8).
- Find the shortest period of time with more than `20` sales, if any (Problem 14).
- Count the number of subarrays of `sales` with at most `10` bad days (Problem 18).
- Count the number of subarrays of `sales` with exactly `10` bad days (Problem 19).
- Count the number of subarrays of `sales` with at least `10` bad days (Problem 20).

All these questions receive an array as input, `sales`. The first four ask us to find a subarray, while the last three ask us to count subarrays, making them ideal candidates for the sliding window technique. In this chapter, we will cover variants of the sliding window technique to tackle each of the mentioned problems and more.

The basic idea of a sliding window is to consider a subarray (the "window"), marked by left (`l`) and right (`r`) pointers. We move or "slide" the window to the right by increasing the `l` and `r` pointers, all while computing some value about the current window.[2] [3]

1 Beyond DS&A, the term 'sliding window' is also used in network protocols like TCP (https://en.wikipedia.org/wiki/Transmission_Control_Protocol) and in machine learning architectures like convolutional neural networks (https://en.wikipedia.org/wiki/Convolutional_neural_network).

2 A sliding window is a special case of the two-pointer technique. Like in the Two Pointers chapter, we use the terms "pointer" and "index" interchangeably.

3 Sliding windows are usually not useful for problems about subsequences because they don't have a good way of dealing with "skipping" elements. Subsequence problems are more commonly tackled with other techniques that we will see later, like dynamic programming or backtracking.

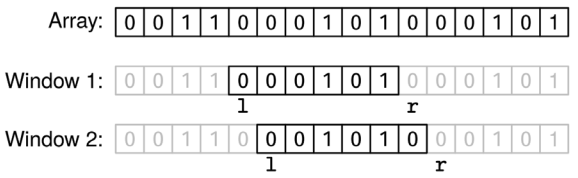


Figure 1. Window 1 is a subarray from `l = 4` (included) to `r = 10` (excluded). We can slide it and get Window 2 by increasing `l` and `r`.

THE ELEMENTS OF A SLIDING WINDOW PROBLEM

Problems where sliding windows may be useful tend to involve the following:

- You have to find a subarray of an input array.
- This subarray must satisfy some *constraint*, which separates the subarrays into *valid* and *invalid*. Examples of constraints:
  - » The length must be k (for some given value k).
  - » The sum must be at least / at most / exactly k.
  - » It must contain or not contain specific elements.
  - » It must not contain repeated elements.
- There is usually an *objective* that makes some subarrays "better" than others. For example:
  - » Maximize/minimize the length of the window.
  - » Maximize/minimize the sum of the elements in the window.
  - » Maximize/minimize the number of distinct elements in the window.
- Less commonly, if there is no objective, the goal may be to count the number of valid subarrays.

For instance, in the first bookstore problem, the constraint is "the length of the subarray must be 7," and the objective is to maximize the sum. In the last one, the constraint is "at least 10 bad days," and there is no objective since it is a counting problem. Can you identify the constraints and objectives for the other bookstore problems?

BRUTE FORCE BASELINE

Most sliding window problems can be solved with a brute force algorithm that checks every subarray one by one. If the subarray is valid, then we check if it's the best one so far.

The brute force solution is *correct*, but we'd ideally like a more optimized solution. Before diving into *how* to do this, it can be useful to consider what our upper bound, lower bound, and target runtimes might be (see the Boundary Thinking chapter).

- Upper bound: $O(n^3)$ will be the most common brute force upper bound across sliding window problems, where n is the length of the input array. There are $O(n^2)$ subarrays to search through. For each of those, checking whether it is valid and the best so far in a naive way could take $O(n)$ time.
- Lower bound: if we don't look at every element in the input, we won't even know what some substrings look like, so $O(n)$ is the natural lower bound.
- Target: the sliding window technique often allows us to reach a linear runtime, so we should aim for that.

HOW TO SLIDE A WINDOW

To make things easy to remember, we follow some conventions for initializing and updating all sliding windows in this chapter:

1. The window goes from the element at index `l` **(inclusive)** to the element at index `r` **(exclusive)**. This means:
   - » the window is empty when `l == r`,
   - » `r` points to the first element after the window (if any), and
   - » the length of the window is `r - l`.
2. We always initialize `l` and `r` to `0`, meaning the window starts empty.
3. We *grow* the window by incrementing `r`. We can only grow it when `r < len(arr)`.
4. We *shrink* the window by incrementing `l`. We can only shrink it when `l < r`.
5. We always have `0 ≤ l ≤ r ≤ len(arr)`.

Consistency enables us to predict what our possible off-by-one errors are likely to be. For instance, `r - l` always means "the size of the window," and `l == r` always means "the window is empty," without worrying about off-by-one errors.[4]

ANALYZING SLIDING WINDOWS

Every sliding window consists of a main loop, where, at each iteration, we either grow or shrink the window, or both. Since `r` never decreases and runs from `0` to `n`, our window can only grow `n` times. By this same token, the window can only shrink (by increasing `l`) `n` times. This means that any properly implemented sliding window does at most `O(2n) = O(n)` iterations. To get the total runtime, we need to multiply the number of iterations, `O(n)`, by the time per iteration.

> As long as each iteration grows or shrinks the window (or both), a sliding window algorithm takes `O(n*T)` time, where `n` is the size of the array we are sliding over and `T` is the time per iteration.

Typically, we will be unlikely to reduce the *number* of iterations—we must reach the end of the array—so we should focus on reducing `T`: the time per iteration. We should try to get it down to constant time.

In terms of space analysis, remember that the window is not materialized, it is just identified by the two pointers. So, the space analysis will depend on what other information about the window we need to store.

FIXED-LENGTH WINDOWS

In fixed-length window problems, we have to find a subarray under the constraint that it has a given length. Such problems, which are fairly common, are on the easier side because there are not many subarrays to consider: for a value `k` in the range `1 ≤ k ≤ n`, an array only has `n-k+1 = O(n)` subarrays of length `k`—a lot fewer than $O(n^2)$. Recall the first opening problem:

PROBLEM 38.1 **MOST WEEKLY SALES**

Given an array, `sales`, find the most sales in any 7-day period.

▸ **Example:** `sales = [0, 3, 7, 12, 10, 5, 0, 1, 0, 15, 12, 11, 1]`

4 Our convention is that `l` is inclusive and `r` is exclusive, but this is merely our convention. If you prefer to consider `r` as inclusive, this is equally correct, but be sure to update the little details like the size of the window (which would now be `r - l + 1`). Whatever you do, be explicit about your convention and make sure the little details match.

Output: 44. The 7-day period with the most sales is [5, 0, 1, 0, 15, 12, 11]

▶ **Example:** sales = [0, 3, 7, 12]
Output: 0. There is no 7-day period.

SOLUTION 38.1 MOST WEEKLY SALES

The fact that we are only looking for windows of length 7 gives us a simple strategy for when to grow and shrink the window:

1. Grow the window until it has length 7.
2. Grow and shrink at the same time so that the length stays at 7.

Here is a full solution:

```
1  def most_weekly_sales(sales):
2    l, r = 0, 0
3    window_sum = 0
4    cur_max = 0
5    while r < len(sales):
6      window_sum += sales[r]
7      r += 1
8      if r - l == 7:
9        cur_max = max(cur_max, window_sum)
10       window_sum -= sales[l]
11       l += 1
12   return cur_max
```

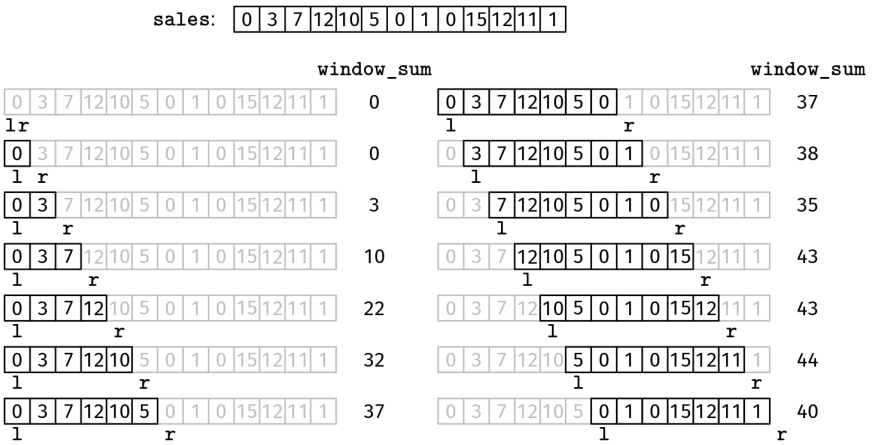


Figure 2. The sliding window of `most_weekly_sales()`.

On top of our window pointers, `l` and `r`, we have:

- `window_sum`: the sum of elements in the window, which corresponds to the objective we have to maximize. The key is to update it whenever the window grows or shrinks and not compute it from scratch at each iteration.
- `cur_max`: where we keep the current maximum we have seen so far.

Each iteration starts by growing the window, which involves two things: updating `window_sum` to reflect that `sales[r]` is now in the window, and increasing `r`. The order of these operations matters!

After growing the window, we check if it is valid, meaning the window length (`r - l`) is 7. If it is valid, we check if it is the best one seen so far and update `cur_max` accordingly.

If the window has a length of 7, we end the iteration by shrinking it so that when we grow it in the next iteration, it will have the right size again. Like growing, shrinking consists of two actions: updating `window_sum` and increasing `l`.

The algorithm ends when the window can no longer grow (`r == len(sales)`).

We can put these ideas together in a general recipe for fixed-length window problems:

RECIPE 1. FIXED-LENGTH WINDOW RECIPE.

```
fixed_length_window(arr, k):
  initialize:
  - l and r to 0 (empty window)
  - data structures to track window info
  - cur_best to 0
  while we can grow the window (r < len(arr))
    grow the window (update data structures and increase r)
    if the window has the correct length (r - l == k)
      update cur_best if needed
      shrink the window (update data structures and increase l)
  return cur_best
```

By "data structures," we mean any information about the window that we need to maintain as we slide it in order to evaluate each window quickly. The data structures that we need change from problem to problem, and they could range from nothing at all to things like sets and maps. In the following problem set, you will have to consider what information to store about the window and how to update it efficiently.

Maintaining information about the window as it slides is a key idea in designing efficient sliding windows.

NESTED LOOPS ARE TOO SLOW FOR SLIDING WINDOW QUESTIONS INTERVIEW REPLAY

View Online:	bctci.co/sliding-windows-replay-1 @ 10:36 - 47:30
The Question:	Given an array of positive numbers and a positive number k, find the maximum sum of any contiguous subarray of size k.
What You'll See:	The candidate struggled to identify the problem as a sliding window problem and coded the brute force instead of an optimal answer.
Who:	*Interviewer*: Software Engineer at FAANG+ *Candidate*: College student

FIXED-LENGTH WINDOWS PROBLEM SET

Try these problems with AI Interviewer: bctci.co/sliding-windows-problem-set-1

We will continue with the bookstore setting. In addition to the `sales` array, we have an array of strings, `best_seller`, with the title of the most sold book for each day.

Constraints:

`sales` and `best_seller` have a length of at most 10^6.
Each book title in `best_seller` has a length of at most 100.

PROBLEM 38.2 MOST SALES IN K DAYS

Given the array `sales` and a number k with `1 ≤ k ≤ len(sales)`, find the most sales in any k-day period. Return the first day of that period (days start at `0`). If there are multiple k-day periods with the most sales, return the first day of the first one.

- **Example:** `sales = [8, 1, 3, 7], k = 2`
  Output: `2. The subarray of length 2 with maximum sum is [3, 7], which starts at index 2.`

PROBLEM 38.3 UNIQUE BEST SELLER STREAK

Given the array `best_seller` and a number k with `1 ≤ k ≤ len(sales)`, return whether there is any k-day period where each day has a *different* best-selling title.

- **Example:** `best_seller = ["book3", "book1", "book3", "book3", "book2", "book3", "book4", "book3"], k = 3`
  Output: `True. There is a 3-day period without a repeated value: ["book2", "book3", "book4"].`
- **Example:** `best_seller = ["book3", "book1", "book3", "book3", "book2", "book3", "book4", "book3"], k = 4`
  Output: `False. There are no 4-day periods without a repeated value.`

PROBLEM 38.4 ENDURING BEST SELLER STREAK

Given the array `best_seller` and a number k with `1 ≤ k ≤ len(sales)`, return whether there is any k-day period where every day has the *same* best-selling title.

- **Example:** `best_seller = ["book3", "book1", "book3", "book3", "book2"], k = 3`
  Output: `False.`
- **Example:** `best_seller = ["book3", "book1", "book3", "book3", "book2"], k = 2`
  Output: `True.`

PROBLEM SET SOLUTIONS

SOLUTION 38.2 MOST SALES IN K DAYS

We can reuse our solution to the previous problem, tweaking it slightly: replacing 7 with k and tracking the position of the best window in addition to its sum.

SOLUTION 38.3 UNIQUE BEST SELLER STREAK

In this problem, we need to check each window of length k for duplicate titles. Checking for duplicates in an array can be done in linear time using a hash map, assuming we can hash each element in constant time (recall that 'checking for duplicates' is a trigger for hash sets and maps).

As mentioned, efficient sliding window algorithms usually *maintain* information about the window. In this case, the information we need is a **frequency map** (the reusable idea from pg 348): a hash map from the titles in the window to the number of times that they appear in the window. So, for a window like `["book3", "book1", "book3"]`, the map would be `{"book3": 2, "book1": 1}`. We can update this map in `O(1)` time whenever the window grows or shrinks.

We remove book titles from the map if their count goes back down to `0`. This way, the size of the map always represents the number of unique keys (book titles) in the window, and the window satisfies the constraint if the map size is k. The extra space of our solution is `O(k)`.[5]

```
def has_unique_k_days(best_seller, k):
  l, r = 0, 0
  window_counts = {}
  while r < len(best_seller):
    if not best_seller[r] in window_counts:
      window_counts[best_seller[r]] = 0
    window_counts[best_seller[r]] += 1
    r += 1
    if r - l == k:
      if len(window_counts) == k:
        return True
      window_counts[best_seller[l]] -= 1
      if window_counts[best_seller[l]] == 0:
        del window_counts[best_seller[l]]
      l += 1
  return False
```

Frequency maps are often useful in sliding window problems.

SOLUTION 38.4 ENDURING BEST SELLER STREAK

This problem can be solved exactly the same way as the previous one, just by changing the window validity condition from `len(window_counts) == k` to `len(window_counts) == 1`. However, this solution requires `O(k)` extra space for the frequency map. Can you think of a constant-space solution? We'll see one in the next section about the next type of sliding windows: resetting windows.

RESETTING WINDOWS

We call the next type of sliding window "resetting windows." It is for problems where a bigger window is usually better, but a *single* element in the array can make the whole window invalid. Our approach will be simple: *grow the window if we can, and otherwise reset it to empty past the problematic element.*

Recall the second opening bookstore problem:

PROBLEM 38.5 LONGEST GOOD DAY STREAK

Given an array, `sales`, find the most consecutive days with no *bad* days (fewer than `10` sales).

▸ **Example:** `sales = [0, 14, 7, 12, 10, 20]`
Output: `3. The subarray [12, 10, 20] has no bad days.`

SOLUTION 38.5 LONGEST GOOD DAY STREAK

This is a resetting window problem because if we encounter a bad day, whatever window we have so far needs to be discarded. This gives us a simple strategy for when to grow and shrink the window:

1. If the next day is *good*, grow the window.
2. If the next day is *bad*, skip it and reset the window.

5 Don't forget that when storing strings in a map, the space complexity of the map is not just the number of strings, as we also need to factor in the length of the strings. For this problem, we said that all the titles would have length at most 100, so the space complexity is `O(100 * k) = O(k)`.

```
def max_no_bad_days(sales):
  l, r = 0, 0
  cur_max = 0
  while r < len(sales):
    can_grow = sales[r] >= 10
    if can_grow:
      r += 1
      cur_max = max(cur_max, r - l)
    else:
      l = r+1
      r = r+1
  return cur_max
```

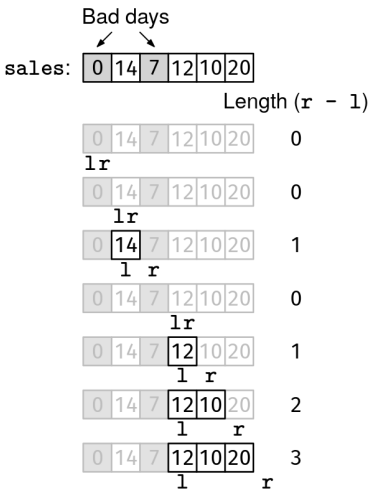


Figure 3. Illustration of the sliding window for `max_no_bad_days()`.

Unlike in the fixed-length window case, a resetting window stays valid throughout the algorithm. We also introduced a `can_grow` variable to decide whether to *grow* or *reset*.[6] When `sales[r]` is a bad day, we reset the window by moving both `l` and `r` past the problematic element.

Once the window cannot grow anymore (`r == len(sales)`), we stop, as we surely won't find a bigger window by shrinking it.

We can put these ideas together in a general recipe for resetting window problems.

RECIPE 2. RESETTING WINDOW RECIPE.

```
resetting_window(arr):
  initialize:
  - l and r to 0 (empty window)
  - data structures to track window info
  - cur_best to 0
  while we can grow the window (r < len(arr))
    if the window is still valid with one more element
      grow the window (update data structures and increase r)
      update cur_best if needed
    else
      reset window and data structures past the problematic element
  return cur_best
```

Now that we have seen two types of sliding windows, it is worth mentioning that problems can fit the criteria for more than one window type.

Recall Problem 38.4: "Given the array `best_seller` and a number k with `1 ≤ k ≤ len(sales)`, return whether there is any k-day period where every day has the *same* best-selling title." We can solve it with a fixed-length window like we saw, or with a resetting window:

6 You could skip declaring the variable `can_grow` and put the condition directly in the `if` statement, but the name "`can_grow`" makes it clear what the if/else cases correspond to, so it is extra easy for the interviewer to follow.

We grow the window when it is (a) empty or (b) the next title is the same as every element in the window. We reset the window when the next element is different from the ones in the window. In that case, rather than skipping over the element that is different; we start growing a new window from that new element.

```
def has_enduring_best_seller_streak(best_seller, k):
  l, r = 0, 0
  cur_max = 0
  while r < len(best_seller):
    can_grow = l == r or best_seller[l] == best_seller[r]
    if can_grow:
      r += 1
      if r - l == k:
        return True
    else:
      l = r
  return False
```

This solution improves the extra space to `O(1)`.

RESETTING WINDOWS PROBLEM SET

Try these problems with AI Interviewer: bctci.co/sliding-windows-problem-set-2

PROBLEM 38.6 MAX SUBARRAY SUM

Given a non-empty array `arr` of integers (which can be negative), find the non-empty subarray with the maximum sum and return its sum.

- **Example:** `arr = [1, 2, 3, -2, 1]`
  Output: `6. The subarray with the maximum sum is [1, 2, 3].`
- **Example:** `arr = [1, 2, 3, -2, 7]`
  Output: `11. The subarray with the maximum sum is the whole array.`
- **Example:** `arr = [1, 2, 3, -8, 7]`
  Output: `7. The subarray with the maximum sum is [7].`
- **Example:** `arr = [-2, -3, -4]`
  Output: `-2. The subarray cannot be empty.`

PROBLEM 38.7 LONGEST ALTERNATING SEQUENCE

Given the array `sales`, find the longest sequence of days alternating between good days (at least `10` sales) and bad days (fewer than `10` sales).

- **Example:** `sales = [8, 9, 20, 0, 9]`
  Output: `3. The only good day is day 2, so the subarray [9, 20, 0] alternates from bad to good to bad.`
- **Example:** `arr = [0, 0, 0]`
  Output: `1. Every day is bad, so we cannot find any pair of consecutive days that alternate.`

PROBLEM SET SOLUTIONS

SOLUTION 38.6 MAX SUBARRAY SUM

This is such a classic problem that the resetting window algorithm for it has its own name: *Kadane's algorithm*.

Let's consider the logic for when to grow and shrink our window. If we encounter a positive number, we definitely want to grow the window, as it makes the window sum bigger. If we encounter a negative number, should we keep it and keep growing (as in Example 2), or should we reset the window *past* it (as in Example 3)? The answer depends on the sum of the window elements so far:

- If our current window plus the negative element is still positive, it is worth keeping the current window even with the negative element.
- If the negative element makes the window sum negative, it is not worth keeping; we should reset it.[7]

```
def max_subarray_sum(arr):
  max_val = max(arr)
  if max_val <= 0: # Edge case without positive values.
    return max_val
  l, r = 0, 0
  window_sum = 0
  cur_max = 0
  while r < len(arr):
    can_grow = window_sum + arr[r] >= 0
    if can_grow:
      window_sum += arr[r]
      r += 1
      cur_max = max(cur_max, window_sum)
    else:
      window_sum = 0
      l = r+1
      r = r+1
  return cur_max
```

SOLUTION 38.7 LONGEST ALTERNATING SEQUENCE

This is a resetting window problem because if we find two consecutive days that are both good or both bad, the whole window becomes invalid and we need to reset it (starting from the second of the two consecutive elements of the same type). We can grow the window when (a) it is empty or (b) the next element does not break the "chain" of alternating days (`(sales[r - 1] < 10) != (sales[r] < 10)`).

MAXIMUM WINDOWS

We are going to tackle general *maximization* problems (maximum length, maximum sum, etc.) with what we call maximum windows. *Maximum windows grow when they can and shrink when they must.* They are similar to resetting windows, but when we encounter an element that makes the window invalid, we don't discard the whole window and reset it. Instead, we shrink it element by element (by increasing `l`) until it becomes valid again.

7 In both cases, we need to increment `r`. You can see in our implementation that we could factor out that increment outside of the if/else cases and save a line of code. However, as we mentioned on page 297, when trying to come up with a valid solution, it is easier to think about each case independently first—shared code between the cases can make it harder to reason about the correctness of your code. If you have time, once you are sure your code is correct, you can do a 'clean-up' pass to tidy up the code.

Recall the third opening bookstore problem:

PROBLEM 38.8 MAXIMUM WITH AT MOST 3 BAD DAYS

Given an array `sales`, find the most consecutive days with at most 3 bad days (fewer than `10` sales).

▶ **Example:** `sales = [0, 14, 7, 9, 0, 20, 10, 0, 10]`
Output: `6. There are two 6-day periods with at most 3 bad days, [14, 7, 9, 0, 20, 10] and [9, 0, 20, 10, 0, 10].`

SOLUTION 38.8 MAXIMUM WITH AT MOST 3 BAD DAYS

We can follow this strategy for when to grow and shrink the window:

1. If the next day is *good* or the window contains fewer than 3 bad days, grow the window.
2. Otherwise, shrink it.

Here is a full solution:

```
def max_at_most_3_bad_days(sales):
  l, r = 0, 0
  window_bad_days = 0
  cur_max = 0
  while r < len(sales):
    can_grow = sales[r] >= 10 or window_bad_days < 3
    if can_grow:
      if sales[r] < 10:
        window_bad_days += 1
      r += 1
      cur_max = max(cur_max, r - l)
    else:
      if sales[l] < 10:
        window_bad_days -= 1
      l += 1
  return cur_max
```

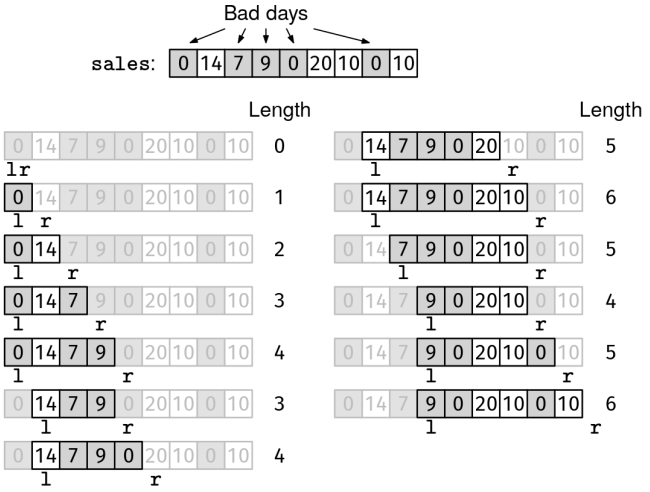


Figure 4. Sliding window for `max_at_most_3_bad_days()`.

Many elements of the solution should look similar to the resetting window recipe, like the `can_grow` variable. The only new part is what happens when we cannot grow the window. We remove *only* the first element in the window (`sales[l]`).

Here is a recipe for maximum windows. Note how, before shrinking the window, we need to check the case where the window is empty (`l == r`)—an empty window cannot be shrunk! For many problems, an empty window is always valid, so we can omit this check (for example, in this problem, an empty window is always valid because it has 0 bad days).

RECIPE 3. MAXIMUM WINDOW RECIPE.

```
maximum_window(arr):
  initialize:
  - l and r to 0 (empty window)
  - data structures to track window info
  - cur_best to 0
  while we can grow the window (r < len(arr))
    if the window would still be valid with one more element
      grow the window (update data structures and increase r)
      update cur_best if needed
    else if the window is empty
      advance both l and r
    else
      shrink the window (update data structures and increase l)
  return cur_best
```

MAXIMUM WINDOWS PROBLEM SET

Try these problems with AI Interviewer: bctci.co/sliding-windows-problem-set-3

Follow the maximum window recipe to tackle the following questions.

PROBLEM 38.9 AD CAMPAIGN BOOST

Imagine that our little bookstore has an array, `projected_sales`, with the projected number of sales per day in the future. We are trying to pick k days for an advertising campaign, which we expect to boost the sales on those specific days by at least `20`. If we pick the days for the advertising campaign correctly, what is the maximum number of consecutive *good* days in a row we can get? (Recall that a good day is a day with at least `10` sales.)

- **Example:** `projected_sales = [5, 0, 20, 0, 5], k = 2`
  Output: `3. The only good day is day 2. We can boost days 0 and 1, days 1 and 3, or days 3 and 4. For instance, if we boost days 0 and 1, the projected sales become [25, 20, 20, 0, 5], with 3 consecutive good days.`
- **Example:** `arr = [0, 10, 0, 10], k = 1`
  Output: `3. We can boost day 2; boosting day 0 is suboptimal.`

PROBLEM 38.10 AD CAMPAIGN WITH SMALL BOOSTS

In the previous problem, what would change if the boost from the advertising campaign was only 5 books instead of `20`? You cannot boost the same day more than once. What is the maximum number of consecutive *good* days in a row we can get?

- **Example:** `projected_sales = [8, 4, 8], k = 3`
  Output: `1. We can boost all 3 days, resulting in [13, 9, 13] projected sales. The max consecutive good days is 1.`
- **Example:** `projected_sales = [10, 5, 8], k = 1`
  Output: `2. We should boost day 1, resulting in [10, 10, 8] projected sales.`

PROBLEM 38.11 BOOSTING DAYS MULTIPLE TIMES

In Problem 38.9, what would change if the boost from the advertising campaign was only 1 book instead of 20, but *you can boost the same day more than once*? What is the maximum number of consecutive *good* days in a row we can get?

- **Example:** `projected_sales = [5, 5, 15, 0, 10], k = 12`
  Output: `3. We can reach 3 consecutive good ways in two ways: boosting days 0 and 1, so both reach 10 sales, or boosting day 3.`
- **Example:** `projected_sales = [5, 5, 15, 0, 10], k = 15`
  Output: `4. We can boost days 1 and 3.`

PROBLEM 38.12 LONGEST PERIOD AT-MOST K DISTINCT

Given an array of strings, `best_seller`, that lists the title of the most sold book for each day, and a number $k \geq 1$, find the maximum consecutive days with at most k *distinct* best-selling books.

- **Example:** `projected_sales = ["book1", "book1", "book2", "book1", "book3", "book1"], k = 2`
  Output: `4. The subarray ["book1", "book1", "book2", "book1"] contains only 2 distinct titles.`
  Constraints: `best_seller has a length of at most` 10^6, `and each book title in best_seller has a length of at most 100.`

PROBLEM SET SOLUTIONS

SOLUTION 38.9 AD CAMPAIGN BOOST

This problem introduces a new dimension: we need to make choices that "modify" the window that we are sliding over. This seems complicated at first since there could be many choices.

A naive solution would be to consider all possible sets of k days we could pick, but that would be very inefficient.[8] The key for this type of problem is usually to use the ***REFRAME THE PROBLEM*** booster. We want to find a way to *reframe it in a way that eliminates the choice aspect*. For our problem, instead of *choosing* k days to turn from bad to good, we can look for the longest window with *at most* k bad days because we can pick those bad days and turn them into good days.

With this reframing, the question becomes just like Problem 38.8, which we solved previously, but with a generic limit of k bad days instead of 3.

When a problem asks you to choose k elements to change (or flip, remove, etc.), the problem can often be reframed in terms of finding a window with at most k elements that need to be changed.

8 If k is a constant, the number of subsets of size k, denoted `(n choose k)`, is $O(n^k)$. The worst case is when k is n/2, as `(n choose n/2)` $= O(2^n/\sqrt{n})$. Once k gets larger than n/2, the number of possibilities starts decreasing. For instance, there are only n subsets of size n-1.

SOLUTION 38.10 AD CAMPAIGN WITH SMALL BOOSTS

We can ***REFRAME THE PROBLEM*** as: "find the longest window with at most k values between 5 and 9 and `0` values less than 5." Then, it becomes a standard maximum window problem.

SOLUTION 38.11 BOOSTING DAYS MULTIPLE TIMES

First, we never want to boost a day beyond `10` projected sales, since we only care about the day being 'good.' The 'cost' of turning a day with x sales into a good day is `max(10-x, 0)`. Thus, we can ***REFRAME THE PROBLEM*** as: "Find the longest window where the sum of `max(10-x, 0)` over each element x in the window is at most k." Then, it becomes a standard maximum window problem.

```
def max_consecutive_with_k_boosts(projected_sales, k):
  l, r = 0, 0
  used_boosts = 0
  cur_max = 0
  while r < len(projected_sales):
    can_grow = used_boosts + max(10 - projected_sales[r], 0) <= k
    if can_grow:
      used_boosts += max(10 - projected_sales[r], 0)
      r += 1
      cur_max = max(cur_max, r - l)
    elif l == r:
      r += 1
      l += 1
    else:
      used_boosts -= max(10 - projected_sales[l], 0)
      l += 1
  return cur_max
```

SOLUTION 38.12 LONGEST PERIOD AT-MOST K DISTINCT

We need to keep track of the number of distinct books in the window. Again, we can use a *frequency map* from book titles in the window to their number of occurrences. When we shrink the window, if a count goes down to `0`, we remove the corresponding key from the map. This way, the size of the map reflects the number of distinct elements in the window, and the window is valid if the map's size is at most k.

```
def max_at_most_k_distinct(best_seller, k):
  l, r = 0, 0
  window_counts = {}
  cur_max = 0
  while r < len(best_seller):
    can_grow = best_seller[r] in window_counts or len(window_counts) + 1 <= k
    if can_grow:
      if not best_seller[r] in window_counts:
        window_counts[best_seller[r]] = 0
      window_counts[best_seller[r]] += 1
      r += 1
      cur_max = max(cur_max, r - l)
    else:
      window_counts[best_seller[l]] -= 1
      if window_counts[best_seller[l]] == 0:
        del window_counts[best_seller[l]]
      l += 1
  return cur_max
```

LIMITATIONS OF MAXIMUM WINDOWS

All the problems we solved with a maximum window (as well as a resetting window) have this property: *Growing an invalid window never makes it valid.*

We call this the *maximum window property*, and it is critical—without it, the maximum window recipe may not work. The intuition is that, without it, we may have to grow through invalid solutions in order to get to the optimal one, making it hard to know when to grow or shrink.

For example, consider a simplified version of Problem 43.7: Longest Subarray With Sum K (pg 618): Given an array of integers, which may be negative, return if any subarray adds up to `0`.

This problem looks like a maximum window problem because it asks for the longest subarray satisfying a constraint, but we can't follow the typical maximum window policy of "grow when you can, shrink when you must."

For instance, if the input starts with `[1, 4, -2, -2, 5, ...]` and we (somehow) grow the window up to `[1, 4, -2, -2]`, should we shrink it in order to find a valid window of length 3, or keep growing because there may be a longer solution with the initial 1, like `[1, 4, -2, -2, 5, -6]`? It is impossible to say.

For problems without the maximum window property, it is better to ditch the sliding window approach entirely and think of different approaches. A linear-time algorithm may also be less realistic (although it is possible for this particular problem using prefix sums, pg 618).

Sliding windows often don't work with negative values.[9]

Finally, there are also problems that have the maximum window property, meaning that the maximum window recipe gives the optimal answer, but it is just really hard to implement efficiently. Here is an example of a classic problem:

PROBLEM 38.13 LONGEST REPEATED SUBSTRING

Given a string, `s`, return the longest substring that appears more than once in `s` (overlapping is allowed) or the empty string if there is none.

- **Example:** `s = "murmur"`
  Output: `"mur"`
- **Example:** `s = "murmurmur"`
  Output: `"murmur"`
- **Example:** `s = "aaaa"`
  Output: `"aaa"`

SOLUTION 38.13 LONGEST REPEATED SUBSTRING

This can be seen as a maximum window problem because we are looking for the longest window with some property. Further, it has the maximum window property: if a substring is not repeated, it won't suddenly become repeated if we make it longer.

The challenge for this problem is assessing whether a substring is valid or not is not easy because it depends on what is *outside* the window instead of what is inside of it.

For problems like this, where the maximum window recipe works but is hard to implement efficiently, we recommend trying something other than sliding windows.

9 Kadane's algorithm (Solution 38.6: Max Subarray Sum (pg 518) for the maximum subarray sum problem is an exception.

For this particular problem, we can try the **guess-and-check technique** we learned in the Binary Search chapter (pg 338). That is, we can try to *binary search over the length of the optimal window*. This approach starts by asking: "If I somehow knew the length of the optimal window, would that make the problem easier?" The answer is often yes because then we can use the fixed-length window recipe, which is the most straightforward one. In this case, the fixed-length window version of the problem is "For a given k, is there a substring of length k that appears more than once?" If we can solve this efficiently, we can then binary search for the transition point in the range of values of k where the answer goes from "yes" to "no". The fixed-length window version can be solved in `O(n)` time using a rolling hash (bctci.co/set-and-map-implementations, Rolling Hash Algorithm section), leading to `O(n log n)` total time.

When a problem has the maximum window property, but you cannot find an efficient way to check if the window is valid or evaluate the window, consider using the guess-and-check technique. For an 'extra' factor of `O(log n)` in the runtime, it turns the problem into a potentially easier fixed-length window problem.

MINIMUM WINDOWS

Minimum window problems are the opposite of maximum window problems. We try to find a window as short as possible, but the constraint restricts how small valid windows can be. We'll use 'minimum windows', which *grow when they must and shrink when they can*.

Recall the fourth opening bookstore problem:

PROBLEM 38.14 SHORTEST PERIOD WITH OVER 20 SALES

Given an array, `sales`, return the length of the shortest period of time with over `20` sales, or `-1` if there isn't any.

- **Example:** `sales = [5, 10, 15, 5, 10]`
  Output: `2. The subarray [10, 15] has over 20 sales.`
- **Example:** `sales = [5, 10, 4, 5, 10]`
  Output: `4. [5, 10, 4, 5] and [10, 4, 5, 10] have over 20 sales.`
- **Example:** `sales = [5, 5, 5, 5]`
  Output: `-1. There is no subarray with more than 20 sales.`

SOLUTION 38.14 SHORTEST PERIOD WITH OVER 20 SALES

This is a minimum window problem because we are trying to find a window as short as possible.

For minimum window problems, the empty window is invalid (in this problem, because it has fewer than `20` sales). We need to grow it until it becomes valid, similar to how we did for fixed-length window problems. We can follow this strategy for when to grow and shrink the window:

1. If the window has `20` sales or fewer: grow it.
2. Otherwise, shrink it to look for a shorter one with over `20` sales.

```
1  def shortest_over_20_sales(sales):
2    l, r = 0, 0
3    window_sum = 0
4    cur_min = math.inf
5    while True:
6      must_grow = window_sum <= 20
7      if must_grow:
8        if r == len(sales):
9          break
10       window_sum += sales[r]
11       r += 1
12     else:
13       cur_min = min(cur_min, r - l)
14       window_sum -= sales[l]
15       l += 1
16   if cur_min == math.inf:
17     return -1
18   return cur_min
```

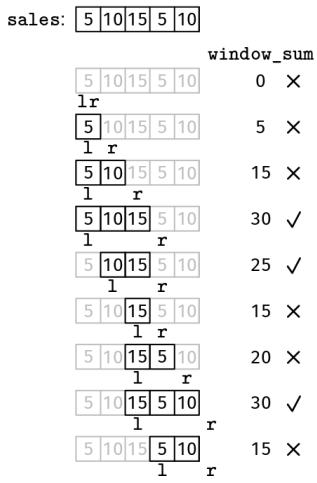


Figure 5. Illustration of the sliding window of `shortest_over_20_sales()`.

Unlike the other recipes, we initialize the result (`cur_min`) to infinity because we update it by taking the minimum. At the end, we need to check if it is still infinity, which means that we didn't find any valid windows. It is a common mistake to forget this final check!

In the main loop, we start each iteration by declaring a variable `must_grow` (instead of `can_grow` for maximum windows) which indicates if the *current* window is invalid. If we must grow, there is one edge case to consider: if `r == len(sales)`, we ran out of elements to grow, so we break out of the loop. We have this edge case for minimum windows but not maximum windows because the while-loop condition is different: we don't stop as soon as `r` gets to the end because it might still be possible to make the window smaller and get a better answer.

If `must_grow` is false, then we have a valid window, so we update the current minimum *first* and then shrink the window to see if we can make it even smaller.

We can put these ideas together in a general recipe for minimum window problems.

RECIPE 4. MINIMUM WINDOW RECIPE.

```
minimum_window(arr):
  initialize:
  - l and r to 0 (empty window)
  - data structures to track window info
  - cur_best to infinity
  while true
    if the window must grow to become valid
      if the window cannot grow (r == len(arr))
        break
      grow the window (update data structures and increase r)
    else
      update cur_best if needed
      shrink the window (update data structures and increase l)
  return cur_best
```

Recall that maximum windows only work for problems that have what we call the maximum window property? For the minimum window recipe to work, we need the opposite property: **Shrinking an invalid window never makes it valid**. We call this the **minimum window property**. It means that if the current window is invalid, we definitely need to grow it.

SUCCESSFULLY SOLVES MINIMUM SLIDING WINDOW PROBLEM		INTERVIEW REPLAY
View Online:	bctci.co/sliding-windows-replay-2 @ 4:15 - 43:23	
The Question:	Find the smallest substring in `s` containing all characters of `t` (including duplicates).	
What You'll See:	The candidate successfully applied a minimum sliding window solution.	
Who:	*Interviewer*: Staff Software Engineer at Meta *Candidate*: 1 year exp.	
Outcome:	The candidate got the job at Amazon!	

MINIMUM WINDOWS PROBLEM SET

Try these problems with AI Interviewer: bctci.co/sliding-windows-problem-set-4

PROBLEM 38.15 SHORTEST WITH ALL LETTERS

Given a string, `s1`, and a shorter but non-empty string, `s2`, return the length of the shortest substring of `s1` that has every letter in `s2` (as many times as they appear in `s2`). If there is no such substring, return `-1`.

- **Example:** `s1 = "helloworld", s2 = "well"`
  Output: `5. The substring "ellow" in s1 has all the letters in s2.`
- **Example:** `s1 = "helloworld", s2 = "weelll"`
  Output: `-1. s1 does not have 2 e's.`

PROBLEM 38.16 SMALLEST RANGE WITH K ELEMENTS

Given an array of integers, `arr`, and an integer k with `1 ≤ k ≤ len(arr)`, return a pair of values, `[low, high]`, with `low ≤ high`, representing the smallest range such that there are at least k elements in `arr` with values at least `low` and at most `high`. If there are multiple valid answers, return any of them.

- **Example:** `arr = [1, 2, 5, 7, 8], k = 3`
  Output: `[5, 8]. The range has 3 elements in arr (5, 7, and 8) and it is smaller than other ranges with 3 elements, such as [1, 5].`
- **Example:** `arr = [5, 5, 2, 2, 8, 8], k = 3`
  Output: `[2, 5]. The range has 4 elements in arr (5, 5, 2, and 2) and there is no smaller range with at least 3 elements. [5, 8] is also a valid answer.`
- **Example:** `arr = [0], k = 1`
  Output: `[0, 0].`

PROBLEM 38.17 STRONG START AND ENDING

We have an array, `projected_sales`, with the number of book sales we expect each day of the fall season. We would like to start and close the season strong. We want to have as many consecutive good days as possible starting from day `0` and as many consecutive good days as possible ending on the last day (a *good day* is a day with at least `10` sales). We can pick k days to boost with advertising, which we expect to boost the sales on those specific days by at least `20`. What's the maximum number of combined initial good days and final good days we can have?

- **Example:** `projected_sales = [10, 0, 0, 0, 10, 0, 0, 10], k = 2`
  Output: `5. We should boost days 5 and 6 so that the projected sales after boosting are [10, 0, 0, 0, 10, 20, 20, 10]. This way, we have 1 initial and 4 final good days.`
- **Example:** `arr = [0, 10, 0, 10], k = 1`
  Output: `3. We can boost either day 0 or day 2.`

PROBLEM SET SOLUTIONS

SOLUTION 38.15 SHORTEST WITH ALL LETTERS

This is a minimum window problem since we have to minimize the window length. A window is valid if the count for each letter is at least as big as the count in `s2`.

The key information we need to maintain about the window is a *frequency map* counting how many times each letter from `s2` is missing. We can also keep a separate count of the number of distinct letters that are missing. When this counter is at `0`, the window is valid.

```
def shortest_with_all_letters(s1, s2):
  l, r = 0, 0
  missing = {}
  for c in s2:
    if not c in missing:
      missing[c] = 0
    missing[c] += 1
  distinct_missing = len(missing)
  cur_min = math.inf
  while True:
    must_grow = distinct_missing > 0
    if must_grow:
      if r == len(s1):
        break
      if s1[r] in missing:
        missing[s1[r]] -= 1
```

```
        if missing[s1[r]] == 0:
          distinct_missing -= 1
      r += 1
    else:
      cur_min = min(cur_min, r - l)
      if s1[l] in missing:
        missing[s1[l]] += 1
        if missing[s1[l]] == 1:
          distinct_missing += 1
      l += 1
  return cur_min if cur_min != math.inf else -1
```

SOLUTION 38.16 SMALLEST RANGE WITH K ELEMENTS

This is an interesting problem because we are not looking for a subarray of the input but rather a window on the *values* in the array. Nonetheless, we can still use a minimum window to find this range.

First, we sort the input since we do not care about the original order, and it will help us find values in `arr` that are close together. After sorting, the problem can be *REFRAMED* as follows:

Find the window containing at least k elements, minimizing the difference between its maximum and minimum. Now that we have a constraint and an objective for the window, this is a standard minimum window problem.

Thanks to sorting, the minimum and maximum in the window are easy to calculate as they are the first and last elements. Sorting is the runtime bottleneck in this solution, so it takes `O(n log n)` time instead of `O(n)` as usual.

```
def smallest_range_with_k_elements(arr, k):
  arr.sort()
  l, r = 0, 0
  best_low, best_high = 0, math.inf
  while True:
    must_grow = (r - l) < k
    if must_grow:
      if r == len(arr):
        break
      r += 1
    else:
      if arr[r - 1] - arr[l] < best_high - best_low:
        best_low, best_high = arr[l], arr[r - 1]
      l += 1
  return [best_low, best_high]
```

SOLUTION 38.17 STRONG START AND ENDING

It is not obvious at all how a problem about maximizing prefix and suffix lengths is related to minimum windows, but we will show a clever trick to ***REFRAME THE PROBLEM*** into a minimum window problem.

What is between a prefix and a suffix? A window! Finding a prefix and a suffix is the twin problem of finding the subarray between them. The next question is: "If we look for a subarray instead of for a prefix and a suffix, what property should the subarray have?"

Let's say `projected_sales` has `B` bad days in total. We know we can flip `k` of them into good days, so we will end up with `B - k` bad days (if `B - k` is `0` or negative, we can convert all bad days into good days, so

the answer is the length of the input array). If we can find the *smallest* window with `B - k` bad days, then we can flip every bad day *outside* the window and maximize the prefix and suffix without bad days.

With this reframing, the problem becomes a standard minimum window problem.

EXTRA CREDIT: COUNTING PROBLEMS

By this point, we have seen many sliding window problems in which the constraint that the window must satisfy is of the form "at most/at least/exactly k of something." In this section, we talk about how to count the number of subarrays under a constraint like that, including the final three opening bookstore problems.

AT-MOST-K COUNTING

PROBLEM 38.18 COUNT SUBARRAYS WITH AT MOST K BAD DAYS

Given an array, `sales`, count the number of subarrays with at most k bad days (days with fewer than `10` sales).

▶ **Example:** `sales = [0, 20, 5], k = 1`
Output: `5. [20] has 0 bad days, and [0], [0, 20], [20, 5], and [5] have 1 bad day each.`

SOLUTION 38.18 COUNT SUBARRAYS WITH AT MOST K BAD DAYS

We can leverage an interesting property about the maximum window recipe: if a problem has the maximum window property (pg 523), whenever we grow the window by adding an element `arr[r - 1]`, the new window is the longest valid window that ends at `arr[r - 1]`. This means that the valid subarrays ending at `arr[r - 1]` are those starting at `arr[l]`, `arr[l + 1]`, and so on, up to `arr[r - 1]` itself. Thus, there are `r - l` valid subarrays ending `arr[r - 1]`.

As we saw in the Problem-Solving Boosters chapter, we can often leverage properties into algorithmic ideas. In this case, we can follow the maximum window recipe—as if we were trying to find the longest window with at most k bad days—and whenever we grow the window by adding an element `arr[r - 1]`, we add `r - l` to a running count of valid subarrays.

Here is a full solution based on this idea:

```
def count_at_most_k_bad_days(sales, k):
  l, r = 0, 0
  window_bad_days = 0
  count = 0
  while r < len(sales):
    can_grow = sales[r] >= 10 or
               window_bad_days < k
    if can_grow:
      if sales[r] < 10:
        window_bad_days += 1
      r += 1
      count += r - l
    else:
      if sales[l] < 10:
        window_bad_days -= 1
      l += 1
  return count
```

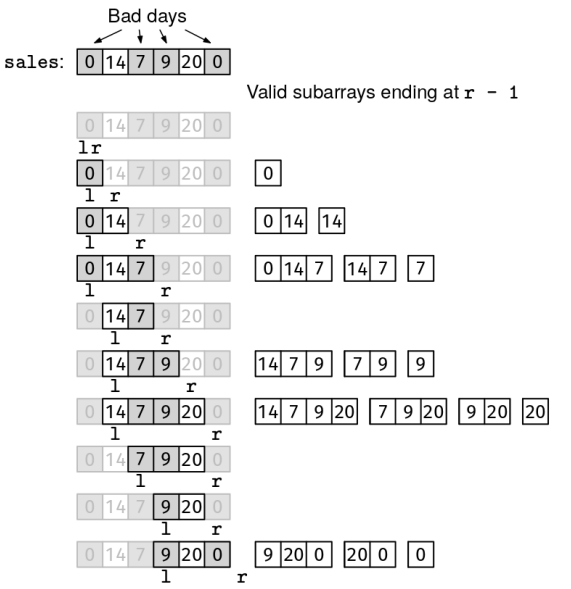


Figure 6. Sliding window for `count_at_most_k_bad_days(sales, 2)`. Every time we grow the window, we show all the valid subarrays that we add to the running count (16 in total).

Note that the code is exactly the same as `max_at_most_3_bad_days()` from the previous section except for computing `count` instead of `cur_max` (and k instead of 3).

If a problem has the maximum window property, the maximum window recipe also works for At-Most-K counting problems. Whenever we add an element to the window, we add to the running count all the valid subarrays ending at that element.

Fortunately, At-Most-K counting problems are a bit of a 'freebie' because we can reuse the maximum window recipe. In the next section, we'll see a trick that makes Exactly-K counting problems equally easy!

As mentioned, this only works if the problem has the maximum window property. For instance, if we allow negative numbers in the array in this problem, we cannot use the algorithm anymore.

EXACTLY-K COUNTING

PROBLEM 38.19 COUNT SUBARRAYS WITH EXACTLY K BAD DAYS

Given an array, `sales`, count the number of subarrays with *exactly* k bad days (days with fewer than `10` sales).

▸ **Example:** `sales = [0, 20, 5], k = 1`
Output: `4. The subarrays [0], [0, 20], [20, 5], and [5] have 1 bad day each.`

SOLUTION 38.19 COUNT SUBARRAYS WITH EXACTLY K BAD DAYS

This time, we will start with the solution and then break it down:

```
def count_exactly_k_bad_days(sales, k):
  if k == 0:
    return count_at_most_k_bad_days(sales, 0)
  return count_at_most_k_bad_days(sales, k) -
```

```
5            count_at_most_k_bad_days(sales, k-1)
```

We re-used the At-Most-K counting function `count_at_most_k_bad_days()` from the previous section.

For `k == 0`, 'at most 0' and 'exactly 0' are the same. For `k == 1`, the code works because the number of subarrays with exactly `1` bad day is equal to the number of subarrays with at most `1` bad day (meaning `0` or `1` bad days) minus the number of subarrays with at most `0` bad days. This applies to any higher k as well!

For example, Figure 7 shows an example `sales` array and all 16 subarrays with at most 2 bad days. Of those, `10` have at most 1 bad day, meaning there are `16 - 10 = 6` subarrays with **exactly** 2 bad days. By the same logic, there are `10 - 2 = 8` subarrays with exactly 1 bad day.

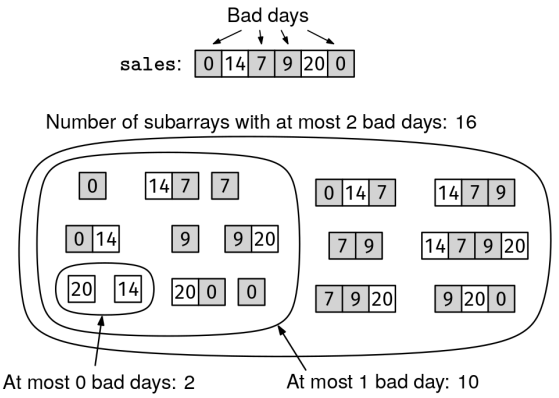


Figure 7. The set of all subarrays with at most 2 bad days, which contains the set of subarrays with at most 1 bad day, which contains the set of subarrays with no bad days.

Since we need to make two calls to the At-Most-K solution, the runtime is twice as long, which is still `O(n)`.

Exactly-K counting problems can also be solved in linear time with prefix sums: see Problem 38.6: Max Subarray Sum (pg 517). The prefix-sums approach works *even if the array has negative numbers.*

AT-LEAST-K COUNTING

PROBLEM 38.20 COUNT SUBARRAYS WITH AT LEAST K BAD DAYS

Given an array, `sales`, count the number of subarrays with *at least* k bad days (days with fewer than `10` sales).

▸ **Example:** `sales = [0, 20, 5], k = 1`
Output: `5. The subarrays [0], [0, 20], [20, 5], and [5] have 1 bad day each, and the subarray [0, 20, 5] has 2.`

SOLUTION 38.20 COUNT SUBARRAYS WITH AT LEAST K BAD DAYS

We will see how to reuse the At-Most-K counting solution once again. First, we need to know the total number of subarrays.

What's the total number of subarrays?

An array of length n has `(n+1)*n/2` non-empty subarrays.

Each subarray is defined by a pair of indices, `[start, end]`. There are n^2 `[i, j]` pairs where `i` and `j` are valid indices (`0 ≤ i, j < n`). Of those, there are n pairs where `i == j`, which define single-element subarrays. Of the remaining n^2 `- n` pairs, half have `i < j` and half have `i > j`.

The latter half does not identify valid subarrays, so we don't count them. In total, we have `n + (n² - n)/2 = (n+1)*n/2` subarrays.

The subarrays with at least k bad days are those with k bad days, those with k+1 bad days, those with k+2 bad days, and so on. Therefore, to count the subarrays with at least k bad days, we can start with the count of *all* subarrays (of which there are `n*(n+1)/2`) and subtract the number of subarrays with at most `k-1` bad days. Something we already saw how to compute!

```
def count_at_least_k_bad_days(sales, k):
  n = len(sales)
  total_subarrays = n*(n+1)//2
  if k == 0:
    return total_subarrays
  return total_subarrays - count_at_most_k_bad_days(sales, k-1)
```

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

REUSABLE IDEA: 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.

1. 'Exactly k' is equivalent to 'at most k' minus 'at most `k - 1`'.
2. '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.

There are counting problems that do not fit into any of the At-Most-K / Exactly-K / At-Least-K categories. We should tackle such problems on a case-by-case basis. Here is an example:

PROBLEM 38.21 COUNT SUBARRAYS WITH GOOD START AND ENDING

Given an array, `sales`, return the number of subarrays that start and end on a good day (a day with at least `10` sales).

SOLUTION 38.21 COUNT SUBARRAYS WITH GOOD START AND ENDING

For this particular problem, the key property to leverage is that each distinct pair of good days in `sales` contributes 1 to the final count. Thus, the answer is `g*(g + 1)/2`, where `g` is the number of good days in `sales`.

COUNTING PROBLEM SET

Try these problems with AI Interviewer: bctci.co/sliding-windows-problem-set-5

PROBLEM 38.22 COUNT SUBARRAYS WITH DROPS

Given an array of integers, `arr`, and an integer k, count how many subarrays have (1) at most k drops, (2) exactly k drops, and (3) at least k drops. A *drop* is a sequence of two consecutive numbers where the first is larger than the second.

▸ **Example:** `arr = [1, 2, 3], k = 1`
Output: `(1) 6. The array has no drops, so every subarray has 0 drops.`

(2) 0. The array has no drops.
(3) 0. The array has no drops.

▶ **Example:** arr = [3, 2, 1], k = 1
Output: (1) 5. [3, 2] and [2, 1] have 1 drop and [3], [2], and [1] have 0 drops.
(2) 2. [3, 2] and [2, 1] have exactly 1 drop.
(3) 3. [3, 2] and [2, 1] have 1 drop and [3, 2, 1] has 2 drops.

PROBLEM 38.23 COUNT SUBARRAYS WITH BAD DAYS IN RANGE

Given the array sales and two numbers k1 and k2 with 0 ≤ k1 ≤ k2, count the number of subarrays with *at least* k1 bad days and *at most* k2 bad days (days with fewer than 10 sales).

▶ **Example:** sales = [0, 20, 5], k1 = 2, k2 = 2
Output: 1. [0, 20, 5] has 2 bad days.

▶ **Example:** sales = [0, 20, 5], k1 = 1, k2 = 2
Output: 5. [0, 20, 5] has 2 bad days, and [0], [0, 20], [20, 5], and [5] have 1 bad day.

PROBLEM 38.24 COUNT SUBARRAYS WITH ALL REMAINDERS

Given an array of positive integers, arr, return the number of subarrays that have at least one of each of the following:

- a multiple of 3,
- a number with remainder 1 when divided by 3, and
- a number with remainder 2 when divided by 3.

▶ **Example:** arr = [9, 8, 7]
Output: 1. [9, 8, 7] counts because 9 % 3 is 0, 7 % 3 is 1, and 8 % 3 is 2.

▶ **Example:** arr = [1, 2, 3, 4, 5]
Output: 6. The subarrays are [1, 2, 3], [2, 3, 4], [3, 4, 5], [1, 2, 3, 4], [2, 3, 4, 5], and [1, 2, 3, 4, 5].

▶ **Example:** arr = [1, 3, 4, 6, 7, 9]
Output: 0. There are no numbers with remainder 2 when divided by 3.

PROBLEM 38.25 COUNT GOOD SUBARRAYS WITH AT LEAST K SALES

Given an array, sales, and a positive integer k, return the number of subarrays with no bad days and at least k total sales (bad days are days with fewer than 10 sales).

▶ **Example:** arr = [15, 20, 5, 30, 25], k = 50
Output: 1. The subarrays with no bad days are [15], [15, 20], [20], [30], [30, 25], and [25]. Of those, only [30, 25] has at least 50 sales.

PROBLEM SET SOLUTIONS

SOLUTION 38.22 COUNT SUBARRAYS WITH DROPS

This is a direct application of the techniques we discussed.

```
1  def count_at_most_k_drops(arr, k):
2    l, r = 0, 0
```

```
  window_drops = 0
  count = 0
  while r < len(arr):
    can_grow = r == 0 or arr[r] >= arr[r-1] or window_drops < k
    if can_grow:
      if r > 0 and arr[r] < arr[r-1]:
        window_drops += 1
      r += 1
      count += r - l
    else:
      if arr[l] > arr[l+1]:
        window_drops -= 1
      l += 1
  return count

def count_exactly_k_drops(arr, k):
  if k == 0:
    return count_at_most_k_drops(arr, 0)
  return count_at_most_k_drops(arr, k) - count_at_most_k_drops(arr, k-1)

def count_at_least_k_drops(arr, k):
  n = len(arr)
  total_count = n*(n+1)//2
  if k == 0:
    return total_count
  return total_count - count_at_most_k_drops(arr, k-1)
```

SOLUTION 38.23 COUNT SUBARRAYS WITH BAD DAYS IN RANGE

The same idea for Exactly-K counting problems also applies to a range:[10]

```
def count_bad_days_range(sales, k1, k2):
  if k1 == 0:
    return count_at_most_k_bad_days(sales, k2)
  return count_at_most_k_bad_days(sales, k2) -
           count_at_most_k_bad_days(sales, k1-1)
```

SOLUTION 38.24 COUNT SUBARRAYS WITH ALL REMAINDERS

Every number has a remainder of `0`, `1`, or `2` when divided by 3. The count of subarrays that have a number from each group is equal to the total number of subarrays (`n*(n+1)/2`) minus the subarrays that have numbers from at most 2 of those groups. For the latter, we use the at-most-k counting technique.

```
def count_all_3_groups(arr):
  n = len(arr)
  total_count = n * (n + 1) // 2
  return total_count - count_at_most_2_groups(arr)

def count_at_most_2_groups(arr):
  l, r = 0, 0
  window_counts = {}
  count = 0
  while r < len(arr):
    can_grow = arr[r] % 3 in window_counts or len(window_counts) < 2
```

10 This is the same logic behind how we use prefix sums to answer range sum queries (pg 612).

```
    if can_grow:
      if not arr[r] % 3 in window_counts:
        window_counts[arr[r] % 3] = 0
      window_counts[arr[r] % 3] += 1
      r += 1
      count += r - l
    else:
      window_counts[arr[l] % 3] -= 1
      if window_counts[arr[l] % 3] == 0:
        del window_counts[arr[l] % 3]
      l += 1
  return count
```

We could have made the code a bit more efficient by replacing the dictionary with an array of length 3, since the set of keys that we are using is `{0, 1, 2}`.

SOLUTION 38.25 COUNT GOOD SUBARRAYS WITH AT LEAST K SALES

We can use the 'break down the problem' booster:

1. First, we can use a resetting window to find all maximal subarrays without any bad days. Then, we can focus on each subarray we found without worrying about bad days.
2. For each subarray, `sub`, from Step 1, we need to count the number of subarrays of `sub` with at least `k` total sales. To do this, we can use the trick for At-Least-K counting: counting the total number of subarrays in `sub` and subtracting the number of subarrays in `sub` with at most `k-1` total sales.

KEY TAKEAWAYS

If you want to try using a sliding window, your first goal should be identifying the *constraint* and the *objective*. Based on those, you can choose the most appropriate window type: see the fixed-length window recipe (pg 513), the resetting window recipe (pg 516), the maximum window recipe (pg 520), and the minimum window recipe (pg 526).

Once you choose a recipe, the next question is:

▸ What information do I need about the window to check its *validity* and *evaluation* efficiently?

Based on the answer, we will pick which window data structures to *maintain* as we slide the window (and remember, if you can't find appropriate window data structures, you can always try the binary search **guess-and-check technique** (pg 338).

Finally, we implement the sliding window. We recommend following some conventions such as those described on page 511. Here are some edge cases to keep an eye for:

- Make sure to consider the case where no valid window is found, especially for minimization problems.
- Make sure not to shrink an empty window or grow a window past the end of the array.

Even better, update your own personal Bug List (pg 172) with the edge cases that *you* tend to forget about.

Counting problems are less common, but we can use the trick to adapt the maximum window recipe to At-Most-K counting problems (pg 530). For Exactly-K/At-Least-K counting problems, see the reusable idea for transforming them to At-Most-K problems (pg 532).

For maximization problems without the maximum window property (pg 523) or minimization problems without the minimum window property (pg 526), sliding windows may be the wrong technique! Consider other techniques commonly used on subarray problems, like two pointers, prefix sums, and dynamic programming.

Sliding window triggers: The input type is just an array of numbers or a string, and maybe a number. The lower bound is linear.

Keywords: subarray, substring, length, contiguous, consecutive, range, longest, or shortest.

At this point, you should be ready to start adding sliding window problems to your practice rotation. You can find the problems in this chapter and additional problems in the companion AI interviewer.

ONLINE RESOURCES

Online resources for this chapter include:

- A chance to try each problem in this chapter in AI Interviewer
- Interview replays that show specific mistakes people make with sliding windows problems
- Full code solutions for every problem in the chapter in multiple programming languages

Try online at bctci.co/sliding-windows.