

CSE 421 Section 3

Greedy Algorithms

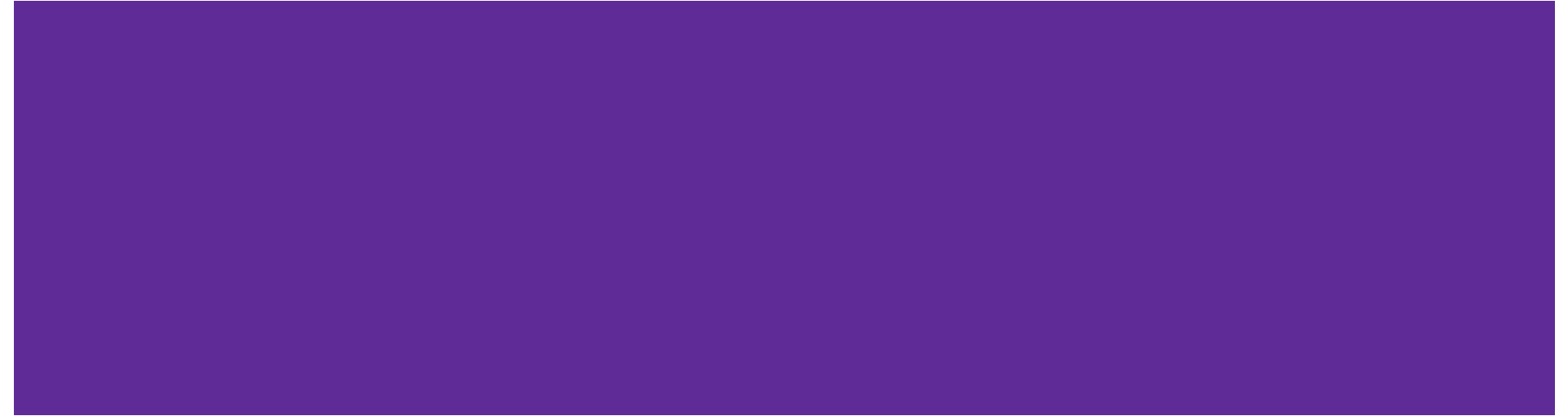
Administrivia



Announcements & Reminders

- HW1
 - If you feel something was graded incorrectly, submit a regrade request!
- HW2
 - Was due yesterday, 1/17
 - 5 late days total
- HW3
 - Due Wednesday, 1/24

How to Approach Writing an Algorithm



Writing an Algorithm

- Throughout this course, you will learn how to write several different kinds of algorithms.
(Ex: greedy algorithms, more kinds to come)
- It can be difficult to understand what a problem actually needs you to do, which makes picking what kind of algorithm might solve the problem challenging as well!
- Today, we will work through a problem using an algorithm-writing strategy that you can apply to all the problems throughout this course and beyond

The Strategy

1. Read and Understand the Problem
2. Generate Examples
3. Come Up with a Baseline
4. Brainstorm and Analyze Possible Algorithms
5. Write an Algorithm
6. Show Your Algorithm is Correct
7. Optimize and Analyze the Run Time

1.1. Read and Understand the Problem



No really, we mean it!

- You can't solve a problem if you don't know what you're supposed to do!
- Remember that problems are written in **mathematical English**, which is likely to be **much more dense** than, say, a paragraph from a novel. You'll have to read more slowly (this advice still applies for our ridiculous word problems).
- As you're reading, **underline anything you don't understand**.
- **Rereading the problem** a few times can often help (it's easier to understand details once you have the big picture in your brain).

Ask Yourself some Questions:

- Are there any **technical terms** in the problem you don't understand?
- What is the **input type**? (Array? Graph? Integer? Something else?)
- What is your **return type**? (Integer? List?)
- Are there any words that look like normal words, but are **secretly technical terms** (like “subsequence” or “list”)? These words sometimes subtly add restrictions to the problem and can be easily missed.

Problem 1 – Line Covering

Your new tow-truck company wants to be prepared to help along the highway during the next snowstorm. You have a list of n doubles, representing mile markers on the highway where you think it is likely someone will need a tow (entrances/exits, merges, rest stops, etc.). To ensure you can help quickly, you want to place your tow-trucks so one is at most 3 miles from **every** marked location. Find the locations which will allow you to place the minimal number of trucks while covering every marked location.

More formally, you will be given an array $A[]$, containing n doubles (in increasing order), representing the locations to cover.

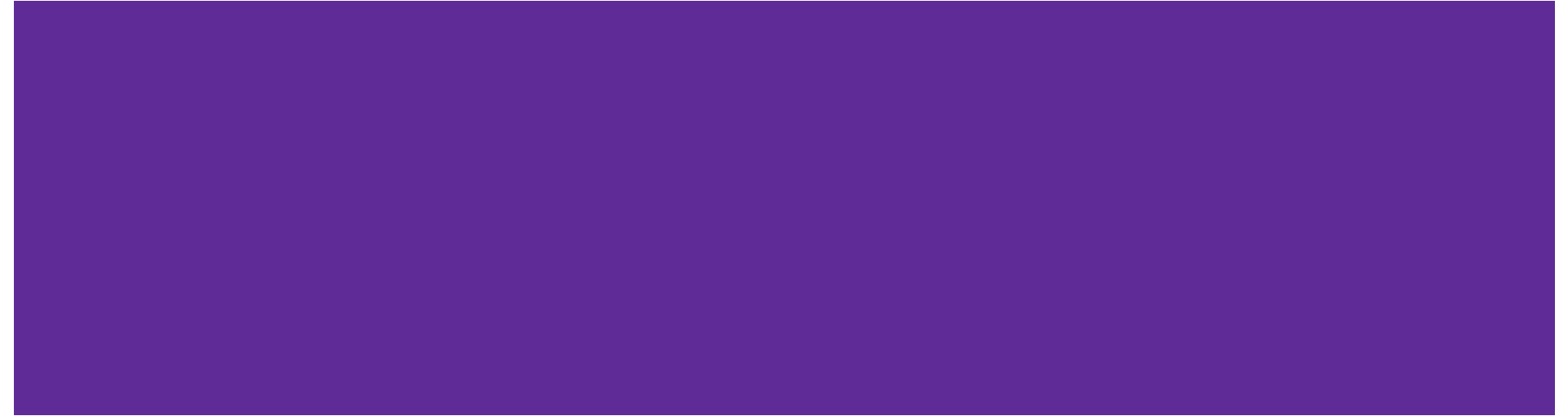
Your task is to produce a list $sites$ containing as few doubles as possible, such that for all i from 1 to n there is a j such that $|A[i] - sites[j]| \leq 3$.

Problem 1.1 – Line Covering

- Are there any **technical terms** in the problem you don't understand?
- What is the **input type**? (Array? Graph? Integer? Something else?)
- What is your **return type**? (Integer? List?)
- Are there any words that look like normal words, but are **secretly technical terms** (like “subsequence” or “list”)? These words sometimes subtly add restrictions to the problem and can be easily missed.

Work reading through this problem with the people around you and answering the four questions, and then we'll go over it together!

1.2. Generate Examples



Examples Help us Understand!

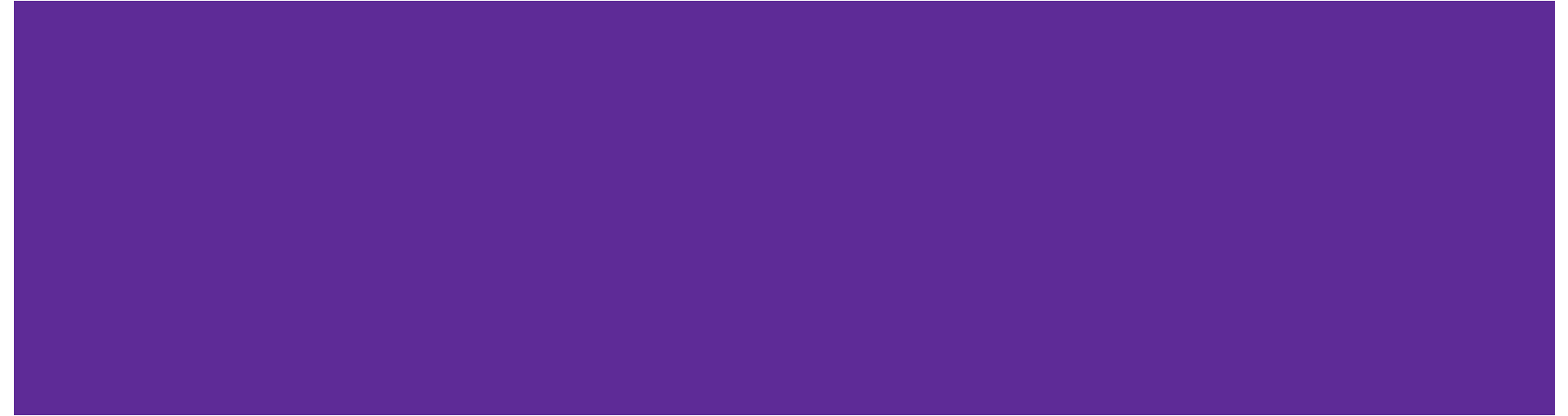
- You should generate two or three sample instances and the correct associated outputs.
- If you're working with others, these instances help make sure you've all interpreted the problem the same way.
- Your second goal is to get better intuition on the problem. You'll have to find the right answer to these instances. In doing so you might notice some patterns that will help you later
- *Note:* You should not think of these examples as debugging examples – null or the empty list is not a good example for this step. You can worry about edge cases at the end, once you have the main algorithm idea. You should be focused on the “normal” (not edge) case.

Problem 1.2 – Line Covering

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

Work through reading this problem with the people around you and answering the four questions, and then we'll go over it together!

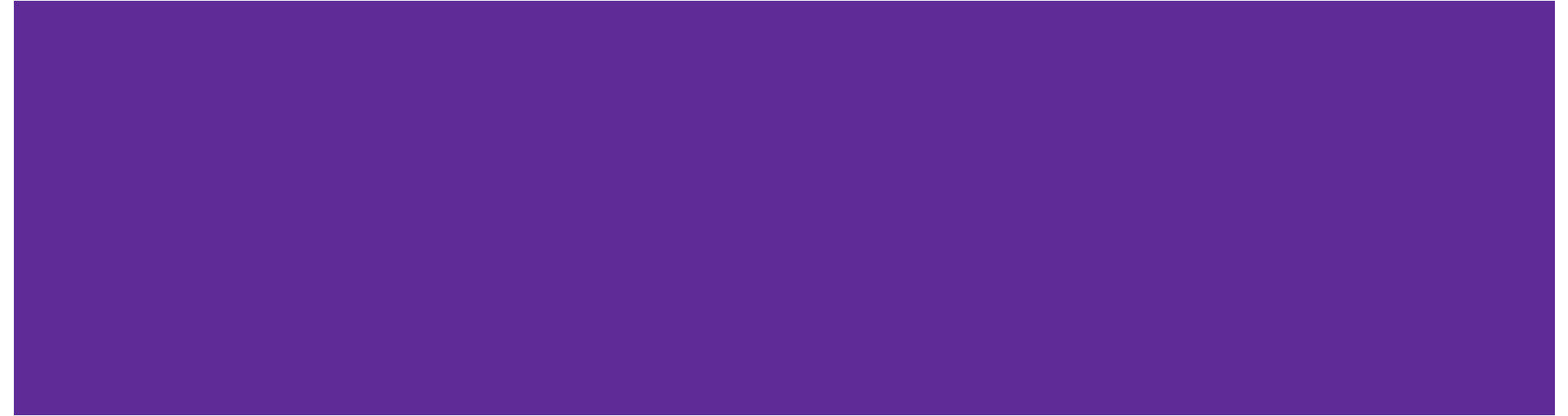
1.3. Come Up with a Baseline



Inefficient but Effective First Attempt

- In a time-constrained setting (like a **technical interview** or an **exam**) you often want a “baseline” algorithm.
- This should be an algorithm that you can implement and will give you the right answer, **even if it might be slow.**
- We’re going to skip this step in this question, but you’ll see it in future examples and in lectures.

1.4. Brainstorm and Analyze Possible Algorithms



Think about Algorithm Possibilities

- It sometimes helps to ask, **“what kind of algorithm could I design?”**
(This week the answer is going to be “a greedy algorithm” because we’re learning about greedy algorithms)
- By the end of the quarter, you’ll have a list of possible techniques.
- Questions to help you pick:
 - Does this problem remind me of any algorithms from class? What technique did we use there?
 - Do I see a modeling opportunity (say a graph we could run an algorithm on, or a stable matching instance we could write)?
 - Is there a way to improve the baseline algorithm (if you have one) to something faster?

Remember: Your First Guess Might Not Be Right!

- You may want to **try multiple different algorithm paradigms** if you're not sure what might work best / be fastest
- Even if you're pretty sure of the algorithm type you want to use, it helps to **brainstorm multiple different possibilities**
- Then, we can test these possibilities, using the examples we came up with in part 2, to see which choice gives the correct output

Problem 1.4 – Line Covering

For this problem today, you should use a greedy algorithm.

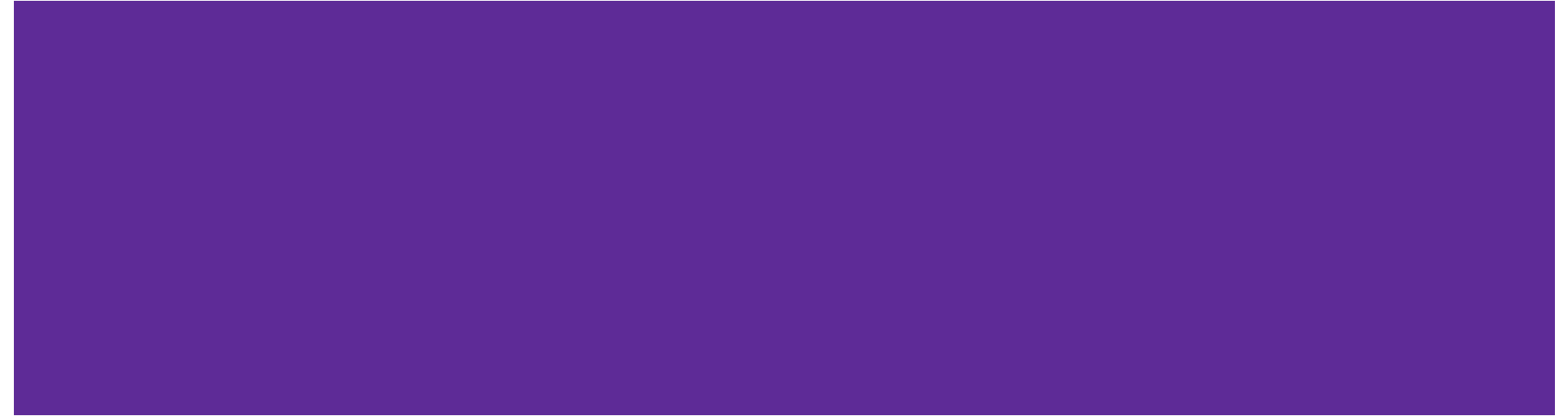
Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Think of some greedy ideas with the people around you, and then we'll go over it together!

Try your Algorithm Ideas on your Examples

- Now that you have some (hopefully) good ideas, you want to test them out on actual example inputs and see if you get the expected output
- Generate a few more instances and narrow down to just one possibility:
 - If you eliminate all of your ideas, go back to the last step and generate a new rule
 - If you can't eliminate down to one choice after multiple examples, try specifically to create an instance where they should behave differently
 - If you still can't make them behave differently, just pick one to try
- This will help you determine which algorithm to pursue!

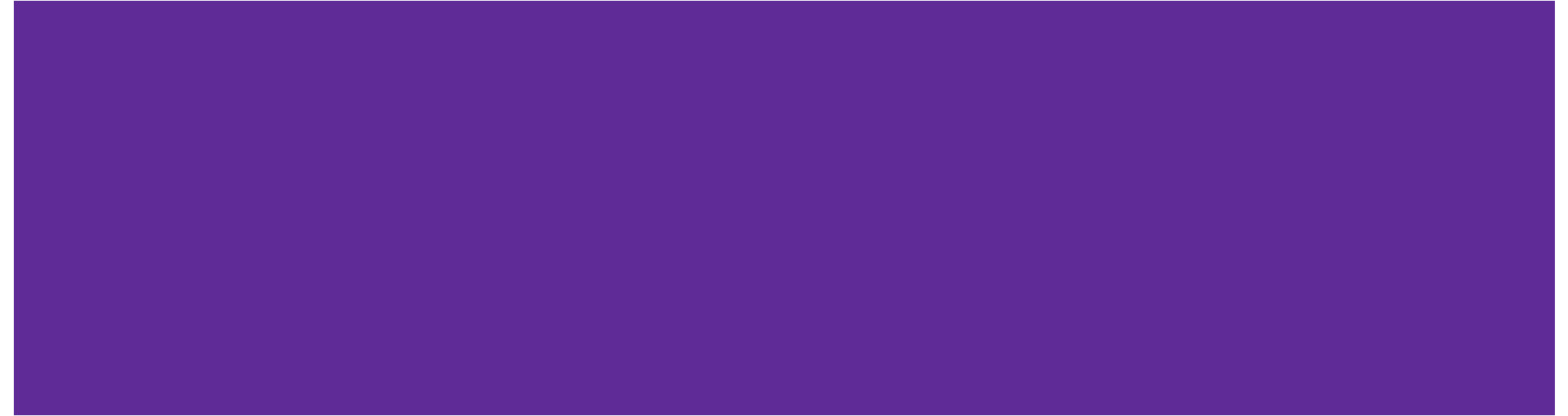
1.5. Write an Algorithm



Write That Pseudocode!

- Now that you've determined which of your ideas is (probably) correct, you need to formalize your algorithm into pseudocode
- Some pseudocode tips:
 - Pseudocode should be somewhere **between** a paragraph and actual (java/python/c/whatever) code
 - You can have English phrases and sentences in your pseudocode!
 - It can be helpful to name the variables you want to use in a paragraph before your actual pseudocode
 - You don't need to be very specific about data structures
 - You should (sparingly) add comments to help explain anything that may be confusing

1.6. Show Your Algorithm is Correct



Writing the Algo isn't Enough...

We Need to Prove that it Works!

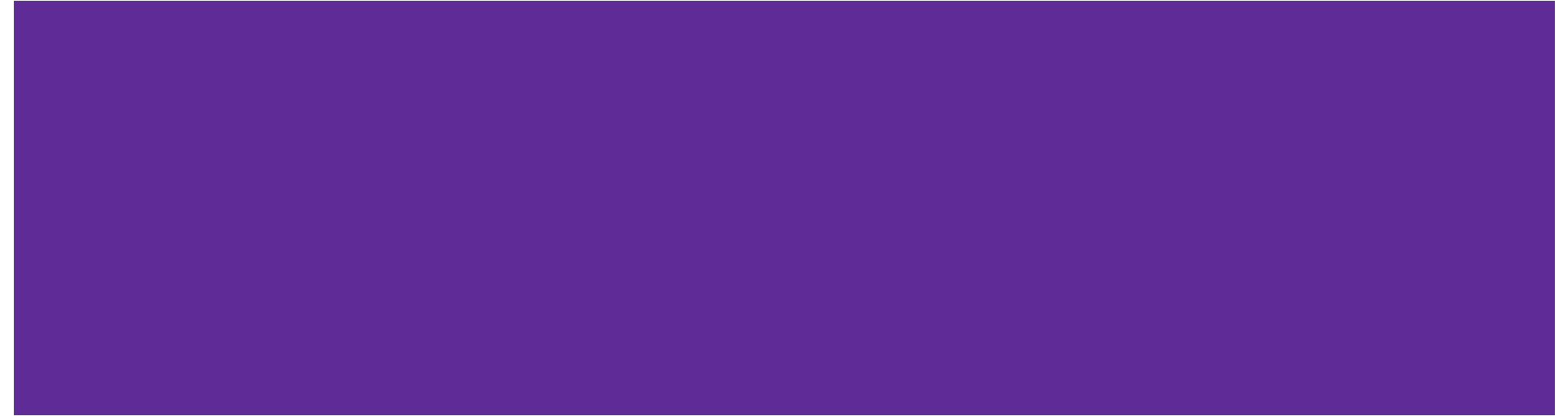
- In general, you'll be often writing some kind of induction proof, or proving some implications to show that your algorithm is correct
- For greedy algorithms specifically, we have three common proof strategies:
 - greedy stays ahead
 - exchange arguments
 - structural arguments

Problem 1.6 – Line Covering

Write a proof of correctness.

Try to use one of those greedy proof techniques with the people around you, and then we'll go over it together!

1.7. Optimize and Analyze the Run Time



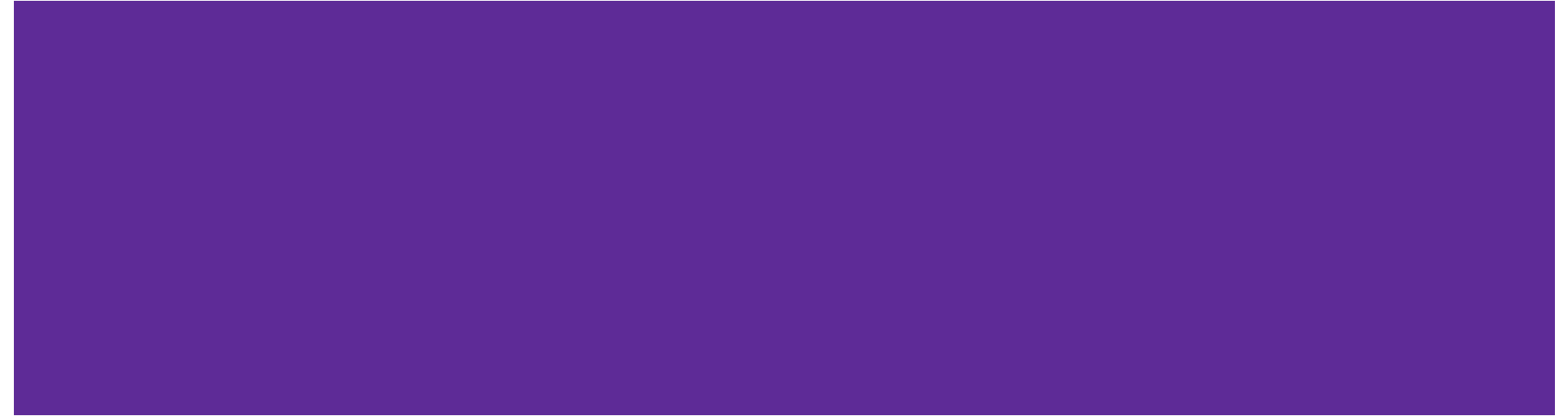
Just Like Back in 332...

- Make your algorithm as efficient as possible.
- Flesh out any pseudocode you've written with enough detail to analyze the running time (do you need particular data structures?).
- Write and justify the big-O running time.
 - Can you make your code more efficient?
 - Can you give a reason why you shouldn't expect the code to be any faster?

Problem 1.7 – Line Covering

Write the big-O of your code and justify the running time with a few sentences.

2.1. Read and Understand the Problem



Problem 2 – Another Greedy Algorithm

You have a set, A , of (possibly overlapping) intervals. You wish to choose a subset B of the intervals to cover the full set. Here, cover means every interval X in A is contained in some interval Y in B .

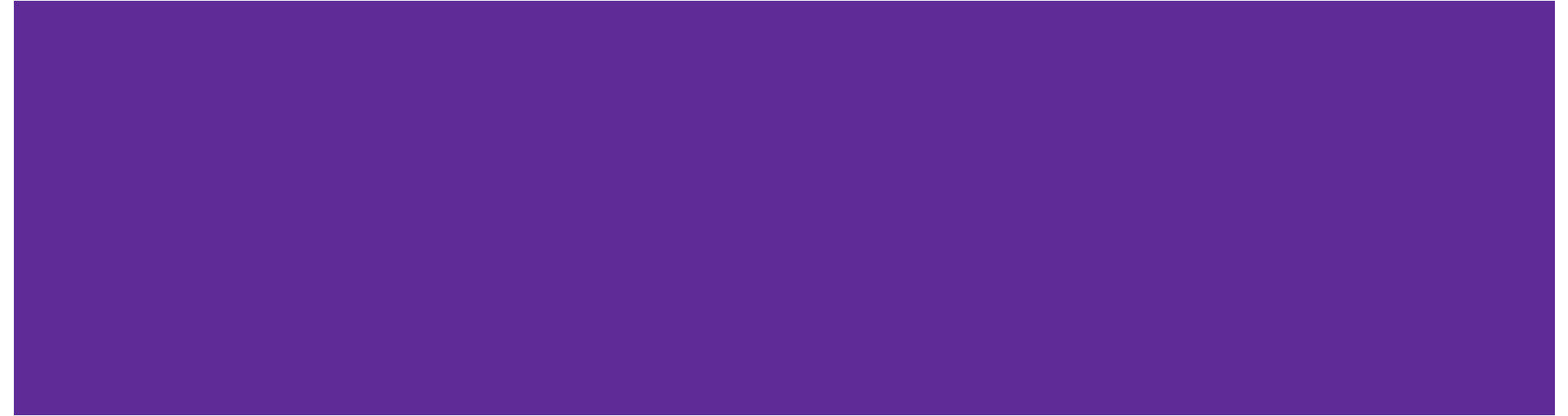
Describe (and prove correct) an algorithm which gives you a cover with the fewest intervals.

Problem 2.1 – Another Greedy Algorithm

- Are there any **technical terms** in the problem you don't understand?
- What is the **input type**? (Array? Graph? Integer? Something else?)
- What is your **return type**? (Integer? List?)

Work reading through this problem with the people around you and answering the questions, and then we'll go over it together!

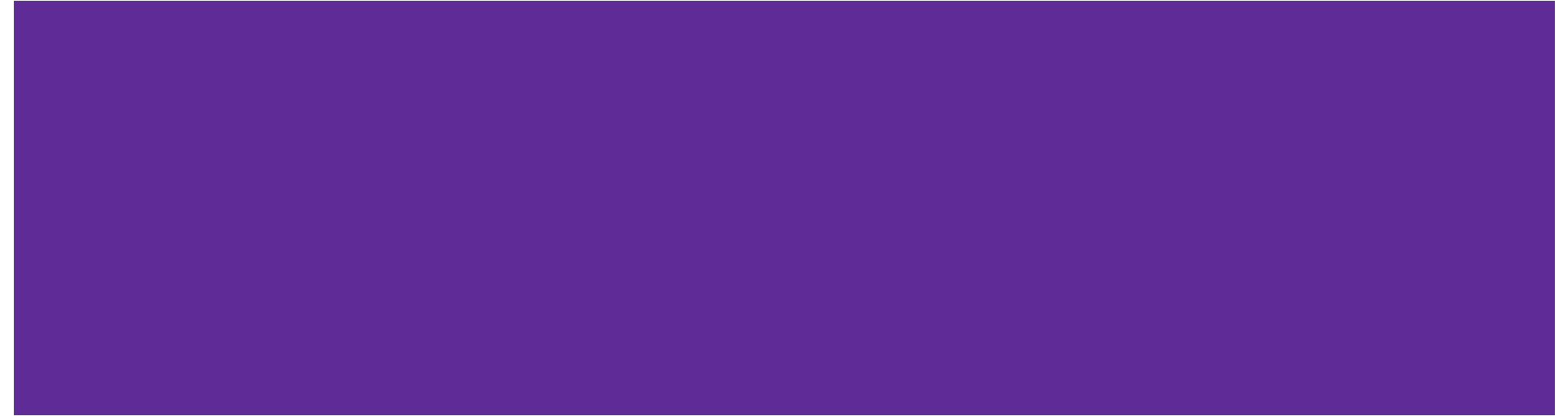
2.2. Generate Examples



Problem 2.2 – Another Greedy Algorithm

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

2.3. Come Up with a Baseline



Inefficient but Effective First Attempt

- In a time-constrained setting (like a **technical interview** or an **exam**) you often want a “baseline” algorithm.
- This should be an algorithm that you can implement and will give you the right answer, **even if it might be slow.**
- What would be a baseline for this question?

2.4. Brainstorm and Analyze Possible Algorithms



Problem 2.4 – Another Greedy Algorithm

For this problem today, you should use a greedy algorithm.

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Think of some greedy ideas with the people around you, and then we'll go over it together!

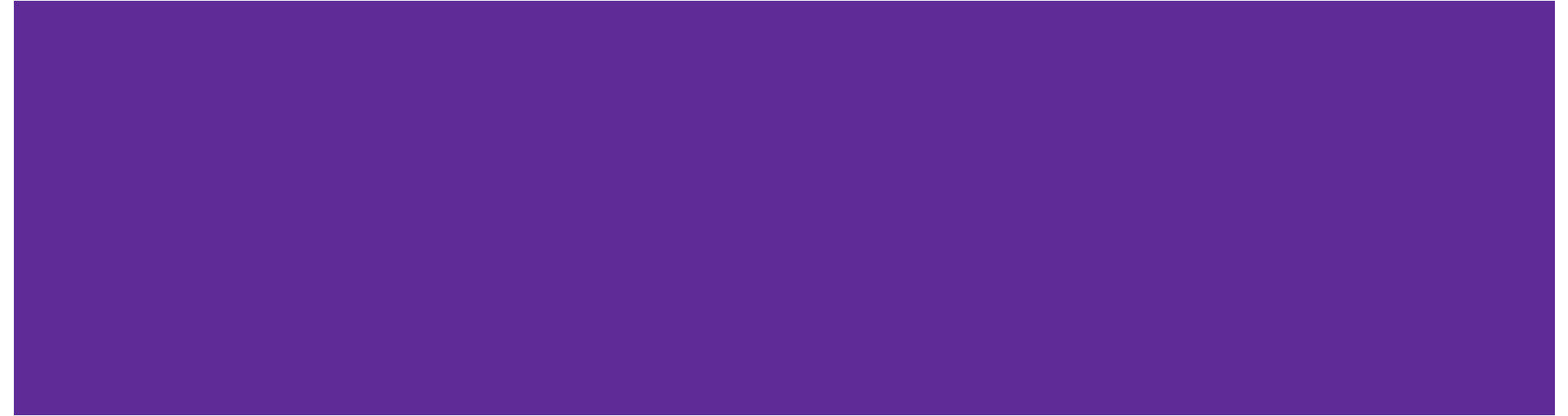
2.5. Write an Algorithm



Write That Pseudocode!

- Now that you've determined which of your ideas is (probably) correct, you need to formalize your algorithm into pseudocode
- Some pseudocode tips:
 - Pseudocode should be somewhere **between** a paragraph and actual (java/python/c/whatever) code
 - You can have English phrases and sentences in your pseudocode!
 - It can be helpful to name the variables you want to use in a paragraph before your actual pseudocode
 - You don't need to be very specific about data structures
 - You should (sparingly) add comments to help explain anything that may be confusing

2.6. Show Your Algorithm is Correct



Problem 2.6 – Another Greedy Algorithm

Write a proof of correctness.

Try to use one of those greedy proof techniques with the people around you, and then we'll go over it together!

2.7. Optimize and Analyze the Run Time



Problem 2.7 – Another Greedy Algorithm

Write the big-O of your code and justify the running time with a few sentences.

3.1. Read and Understand the Problem



Problem 3 – Art Comissions

You've just started a new one-person art company. You've convinced n of your friends to each put $\$c$ of their current money to supporting your dreams by commissioning you to make art. It takes you one month to finish a commissioned piece (you are only working in your limited free-time). One of your friends will pay you at the beginning of the month to make their artwork. While waiting for you to start their commission, your friends are going to place their money into bank accounts, which earn small (and varying!) rates of interest.

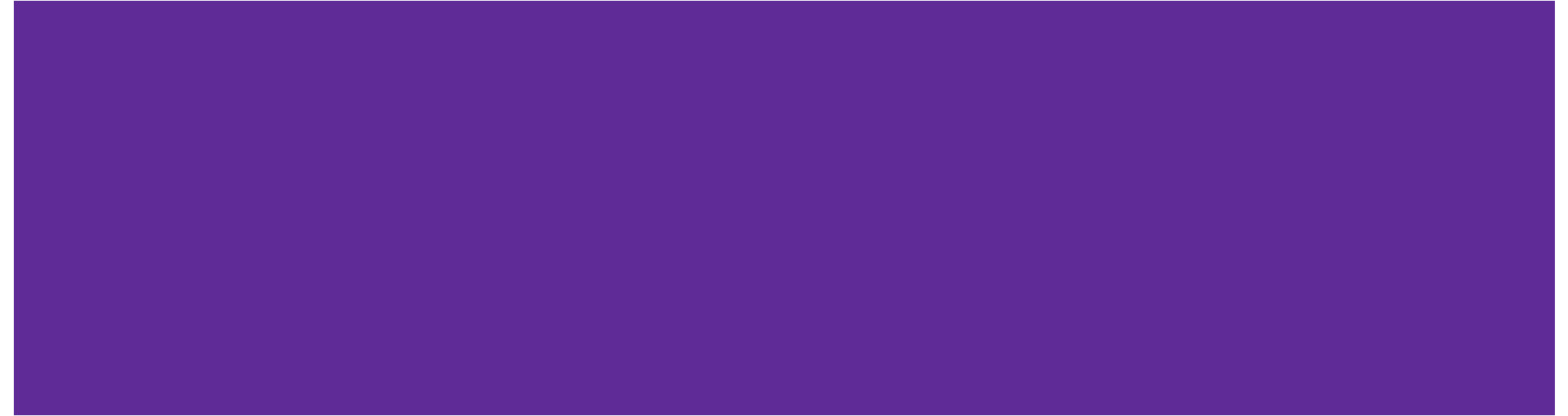
Friend i earns interest at the rate of r_i , compounding monthly. I.e., the amount in their bank account is r_i times what it was at the start of the last month (until they withdraw their money; $r_i > 1$). Your friends generously decide to pay you both the principal and the interest earned at the time you start their commission. Describe an ordering to take the commissions that will maximize the amount you are paid (you may assume you know the r_i for each of your friends).

Problem 2.1 – Another Greedy Algorithm

- Are there any **technical terms** in the problem you don't understand?
- What is the **input type**? (Array? Graph? Integer? Something else?)
- What is your **return type**? (Integer? List?)

Work reading through this problem with the people around you and answering the questions, and then we'll go over it together!

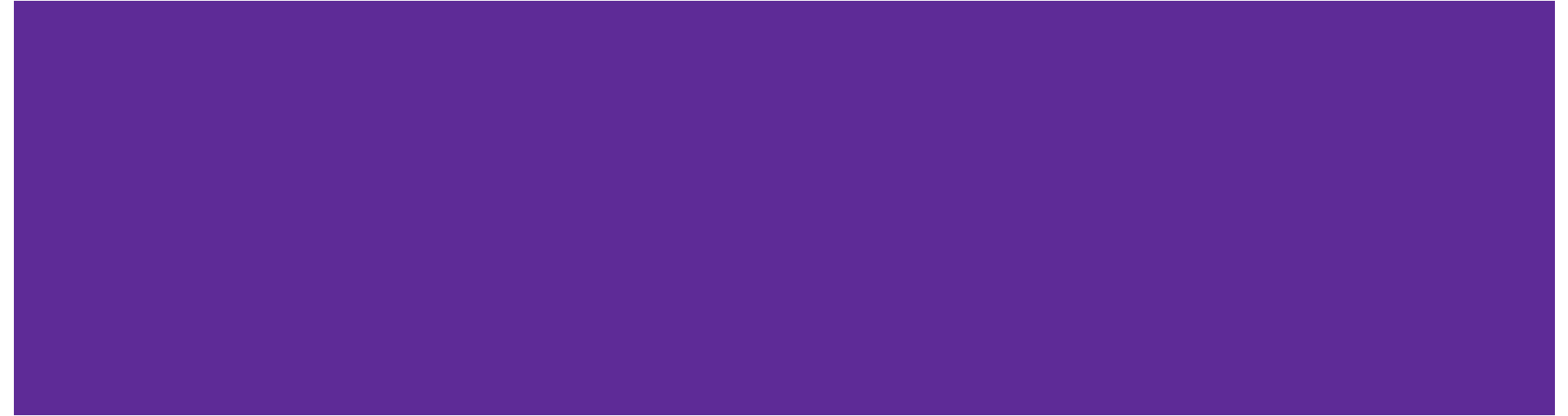
3.2. Generate Examples



Problem 3.2 – Art Comissions

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

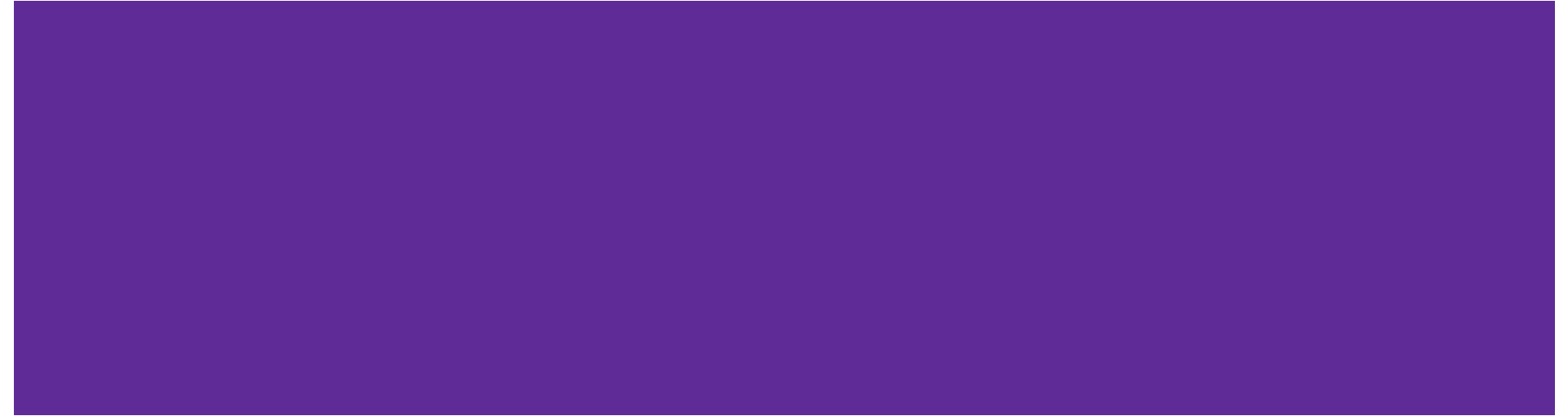
3.3. Come Up with a Baseline



Inefficient but Effective First Attempt

- In a time-constrained setting (like a **technical interview** or an **exam**) you often want a “baseline” algorithm.
- This should be an algorithm that you can implement and will give you the right answer, **even if it might be slow.**
- What would be a baseline for this question?

3.4. Brainstorm and Analyze Possible Algorithms



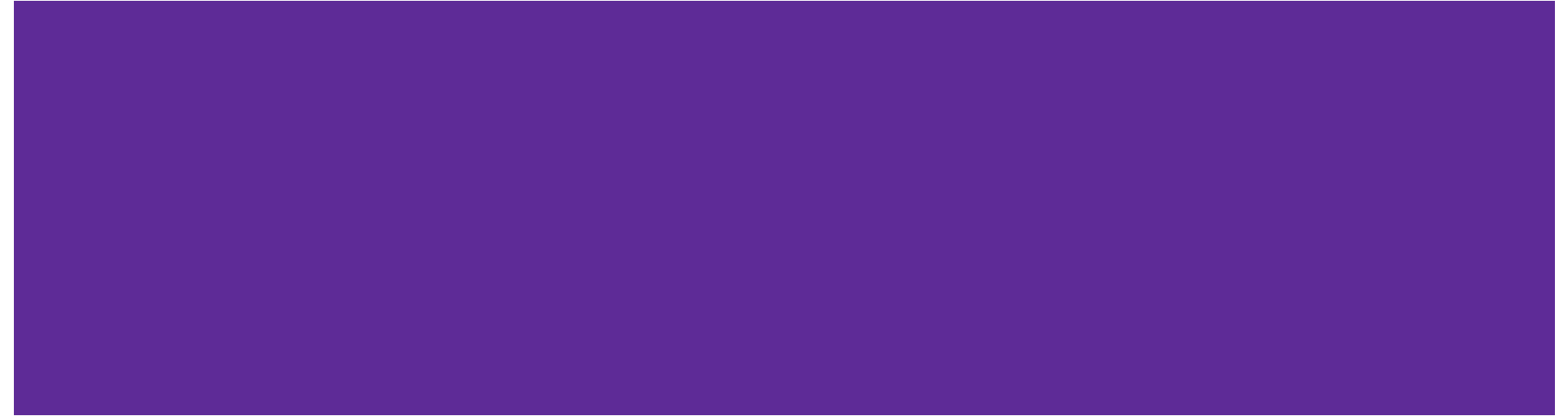
Problem 3.4 – Art Comissions

For this problem today, you should use a greedy algorithm.

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Think of some greedy ideas with the people around you, and then we'll go over it together!

3.5. Write an Algorithm



Write That Pseudocode!

- Now that you've determined which of your ideas is (probably) correct, you need to formalize your algorithm into pseudocode
- Some pseudocode tips:
 - Pseudocode should be somewhere **between** a paragraph and actual (java/python/c/whatever) code
 - You can have English phrases and sentences in your pseudocode!
 - It can be helpful to name the variables you want to use in a paragraph before your actual pseudocode
 - You don't need to be very specific about data structures
 - You should (sparingly) add comments to help explain anything that may be confusing

3.6. Show Your Algorithm is Correct



Problem 3.6 – Art Comissions

Write a proof of correctness.

Try to use one of those greedy proof techniques with the people around you, and then we'll go over it together!

3.7. Optimize and Analyze the Run Time



Problem 3.7 – Art Comissions

Write the big-O of your code and justify the running time with a few sentences.

That's All, Folks!

Thanks for coming to section this week!
Any questions?