CRACKING THE TOPTAL INTERVIEW

CARLOS ROSO

Table of Contents

1

Introduction

What's Toptal?

For starters, Toptal is a remote work marketplace which claims to accept only 3% of the applicants. As such, the work quality and rates are much, much higher than other non-vetting platforms like Upwork, Freelancer, Freeeup, you name it. Toptal connects you with clients all over the world in engagements that can be full-time, part-time, or hourly. Most of the jobs are likely to be long term. Once you get into the marketplace, you'll probably find your first job in 3 or 4 weeks, given you put good time into polishing your profile.

Why Toptal?

I've gone through tough interviews in most of these top remote platforms. All of them claim to filter out the vast majority of applicants. However, even after taking jobs at some of these marketplaces, I can honestly say Toptal has been the best of them. The fact that you can set your own rate and your weekly availability makes it very flexible for a digital nomad like me. They filter out companies with a rigorous process too, so you're almost guaranteed to work with awesome teams. The cash is top; you get much more than what you would earn in most other remote marketplaces.

Who is this for?

This guide is mainly aimed at software developers (web dev, data science, machine learning, devops, etc) with at least 1 year of experience.

How to prepare?

This guide contains practical tips to ace every step of the interview. First, I dive deep into the English test. This step of the process filters out ~76% of the applicants so it's critical to make it through. Then I go into the most daunting phase: the coding interviews. I cover everything from what to expect to preparation tips. You'll also be able to practice with a handful of exercises and understand the solutions right there. Finally, I talk about the Test Project, which is the last step of the screening. I'll go through a dummy exercise for full stack web development and continue to outline a strategy to deliver a job-winning project in the frontend and backend.

Remember, Toptal is not the only option you have. Once you go through all the preparation, you'll be ready to tackle similar interviews in several other top-remote marketplaces. Big tech companies (Facebook, Amazon, Google) will also ask you to be good at English speaking and algorithms. Please keep that in mind, you're not preparing for just one interview, you're investing hard on your future as a developer.

2 English Interview

This section will help you hone your speaking skills and prepare for the interview. It’s mostly directed for those who are already familiar with the language, who can understand and decently speak English at an intermediate level. It will help you polish your speaking skills and prepare for the interview. If you think you’re still new to English I recommend taking professional lessons or finding expert language advice online before taking your chance at Toptal.

What’s the English Interview?

The English interview is the first phase of the screening process and will assess how well you communicate in speaking English. Toptal calls this the “Language and Personality” test. I’ve witnessed several of these interviews myself and can confidently say the interviewers are very strict with grammar and your ability to communicate well-formed thoughts. It’s not so much about fluency as it is about conveying thoughts in a good way.

As for the personality test, I haven’t seen candidates being rejected for this reason. Based on the description on their website, though, they say they assess "personality traits and look for candidates who are passionate and fully engaged in their work". Keep that in mind but remember the most important part of this interview is how well you can communicate in English.

If you fail this interview you'll probably be put on hold for 3-6 months, depending on how good or bad you do. Don't miss your shot and prepare.

Why do they care about this?

Contrary to popular belief, this is somehow the most restrictive filter of them all. According to Toptal's statistics, 73.6% of the applicants fail to pass this test. This means they are very rigorous about this process. Think about it: most companies within Toptal are American or European companies. As opposed to, say, CrossOver, you'll probably be part of a native in-house team in which you'll be only the remote outsider. You need to have top communication skills and show yourself as a professional expert. This includes, of course, great verbal and written English.

Keep this in mind as a way to fuel your motivation. Even when you don't feel like studying English, think about the number of possibilities that will open up for you if you master this single skill. You're broadening your work scope from your country and language to pretty much all the world with much stronger currencies.

How to prepare?

Preparing for an interview in English is not only about practicing speaking. It doesn't either mean to put a few study hours before the interview. You need to be "living" in English: think in English, write your code and commits in English, watch YouTube English-only videos, listen to English-only podcasts, speak with yourself in English. Now, I'm no language expert, but I can tell you I've helped a dozen people, including myself, with these techniques. You need to practice all 4 areas of language to become decently fluent: reading, writing, listening, and speaking.

Reading

Avoid reading any literature (either online or physical) that's not written in English. Google everything in English. I recommend subscribing to a collection in Medium or Dev with a topic you're interested in. Just read the most upvoted as those are normally easier to digest. Pick up a non-fiction book on Kindle or your bookstore and read it. Become comfortable reading content in English.

Listening

This is critical to your success as a Toptal talent and, in general, as a remote worker for top companies. I recommend watching YouTube videos on TED talks. Put it first with subtitles, and repeat it without subtitles. You should be able to understand 80-90% of what you listen to. Keep doing this, every single day, and you'll greatly improve your listening. I recommend avoiding practicing with songs: it’s much harder to understand (I was told it was a good idea, I picked Eminem, it didn’t work) and it doesn’t depict a formal environment as a remote work setup

Writing

Practice answering any of the sample questions below. This exercise will help you:

1. Practice your writing and speaking (fluency) skills
2. Prepare the questions you will probably be asked in the interviews

Write down the answers to the following questions and memorize them. Practice them over and over again. I highly recommend using pen and paper:

- Tell me about yourself
- How long have you been programming?
- Why would you like to work as a talent in Toptal?
- What have you heard about Toptal?
- What are the technologies you’re most passionate about?
- Tell me about a time when you made a mistake and how you handled it
- What experience do you have with technology X and Y?

Speaking

Personally, I find self-talking to be the most effective way to practice speaking fluency. You won't probably improve pronunciation but you will surely get better at your fluency. Read your answers out loud (from the writing section) to yourself. Do this over and over again. You'll see how, over the weeks, your brain will form neural connections that allow you to think and speak much more fluently.

Daily Prep Bite

I recommend the following daily 2-hours practice routine. You can adjust it depending on what your weaknesses are:

- **00:00 - 00:30:** Listen to English podcasts or watch YouTube videos in English without subtitles.
- **00:30 - 01:00:** Write down a full paragraph answering any of the sample questions. Paste it on Hemingway. Correct it and revisit again.
- **1:00 - 1:30:** Repeat the answers to your questions out loud. Practice fluency and speak to yourself.
- **1:30 - 2:00:** Get yourself a 30 mins session with any of the free online apps I recommend in the next section.

Useful resources

- If you don’t have it already, you need to install Grammarly right now. This will help you catch any grammar mistakes in your writings.
- Use the Hemingway Editor to double-check your writings. It’ll highlight any hard-to-read sentence, wrong uses of words, etc.
- For speaking, I found some resources online that might be useful for you:
    - Subreddit /r/language_exchange
    - Sites like FluentU or Free4Talk let you find speaking partners
    - Coursera offers a course for English speaking

Disclaimer: I handpicked and curated what I think are the most relevant apps to practice speaking. Having said that, I haven’t used or tried any of these platforms myself.

Final tips

- It’s ok to have a foreign accent, don’t try to mimic native speakers
- Focus on conveying your ideas as best as you can
- Your interviewers will most probably not be English native speakers
- If you’re asked to show your camera, put up your best face. Studies suggest humans are inclined to favor friendly, welcoming faces.

3

Coding Assessment

In this section, you'll learn what the second step of the Toptal interview looks like, what kind of problems you can expect from it, and how to prepare. This is not the most difficult part of the interview but you'll need a lot of preparation to get through.

What's the coding assessment?

The coding assessment is composed of 2 consecutive tests: an online exam and a live interview. The first one consists of 3 algorithm problems. It's normally conducted in Codility where you'll have 90 minutes to solve 3 problems: easy, medium, and hard. The second test will consist of a phone call with an interviewer which will ask you two algorithm questions.

Both steps will filter out 22,8% more applicants which will put you on the top 3.6% percentile. This means you'll be very close to the top 3% if you make it through this filter.

Why do they test algorithms?

You may wonder what's the value in assessing how you solve algorithms when, in fact, you don't do any of that in your day job. The general belief among high tech companies is that, if you can solve logical, abstract problems, then you're smart enough to pick up any language or framework. Whether you love it or hate it, it is what it is and you need to be comfortable with these types of problems. It's an inconvenience but it's yet another gate you need to open to get where you want to be. It's a means to an end.

How to prepare?

The first thing you should know is you'll need to get an absurd amount of preparation. But not any preparation - you just need the right one. It's nonsense to train 10 years for the Olympics when all you wanted was to win your local competition. You will see a dozen online services and books that offer coaching to crack the coding interviews at the big 4 (Facebook, Amazon, Microsoft, Google). The thing is, you don't need such an in-depth preparation to ace the Toptal interview. You need the concepts, but you won't need to pass a System Design interview or even explain what's the BigO complexity of your algorithm. I've passed interviews at one of the big 4 (Facebook, Amazon, Netflix, Google) and it looks very different (not completely) from what Toptal's coding looks like.

Key Concepts

You should first get familiar with the next 3 concepts.

1. **BigO Notation:** To put it in layman terms, BigO is a standard to tell how efficient your algorithm is. It's an indication of how your algorithm performs in time and space.
2. **Hashmaps:** This is by far the most important concept you should master. I've used hashmaps to pass a lot of coding interviews. Their superpower is they can store and fetch data super fast (in O(1) time). Think of an object in JavaScript or a Hashtable in Java. More on this later on.
3. **Strings and lists:** This might come off as a basic topic for you but it's mandatory that you're fluent manipulating strings and lists. Almost any problem will require you to loop over a list or a string. You don't want to be looking for syntax specific methods in the middle of an interview. Make sure you can instantly push an element at the beginning of a list, find the length of a string, reverse a list, map and reduce, and so on. Now, don't just practice the syntax. You need to be able to reason about the BigO time complexity of the algorithms. Interviewers will ask for that. Are you using .reverse()? ok, what's the time cost of doing that?

Quick BigO Recap

BigO is simply a notation that tells you how well or bad your algorithm performs as your input size grows. I've used the following tips to solve and analyze algorithms in all my coding interviews.

1. O(1)

Means your algorithm will take roughly the same amount of time regardless of the input size. Applies normally to math operations and object lookups.

```
// regardless of a or b, a sum is a math operation that takes just
// one clock cycle to complete
const add = (a, b) => a + b;

// no matter of how big your map is (e.g an object in JavaScript),
// a lookup will take 1 cycle to complete
const lookup = (map, key) => map[key];
```

Accessing the bar prop from the foo object like foo.bar is an O(1) operation.

Note: Hashmap lookups are considered to be O(1) in theory regardless of how they solve collisions.

2. O(n)

Means your algorithm run time will grow linearly with the input. Take, for instance, the algorithm to find the max number of a list. Your algorithm necessarily needs to visit every element to know which number is the max. If your list grows 2x, your algorithm will roughly take twice the time to complete.

```
// you need to loop through all the elements to find the max.
function findMax(list) {
  let max = -Infinity;
  for (obj of list) {
    if (obj > max) max = obj
  }
}
```

3. O(nlogn)

This one is easier than what it looks like. Don't worry about the weird logarithm over there. Logs in BigO notation normally means you're dividing something into halves, and each half is divided again into halves. But, the only thing you need to remember is this: O(nlogn) is mostly associated with sorting. Sorting lists are not free operations, you're incurring in costs when using them. Let's see an example of a very inefficient way to get the max of a list:

```
function getMax(list) {
  return list.sort((a, b) => b - a)[0]
}
```

O(n) is better than O(nlogn) because if you multiply a cat (n) by logn, you get a bigger cat (n*logn).

Now you have good criteria to decide what's most efficient to find a max. Would you use the O(n)? or the O(nlogn) algorithm?

4. O(n^2)

Means your input size affects the algorithm like crazy (not quite as bad as others). If your input size doubles, your run time grows 4x. If your input grows 4x, your run time grows 16x. This normally takes the shape of a nested loop. Let's write a very inefficient way to find if an array has a duplicate.

```
function hasDuplicates(list) {
  for (let [i, el_i] of list.entries()) {
    for (let [j, el_j] of list.entries()) {
      if (i !== j && el_i === el_j) return true;
    }
  }
  return false;
}
```

For every element in the list of size n, we're looping over the whole list of size n. That means we're visiting the elements $n*n = n^2$ times.

5. Final takeaway

No need to dig deeper into this topic to ace an easy to medium interview. Just know that some tasks cost more than others and why. Generally speaking, I recommend you to stick this rule to heart:

```
O(1) < O(n) < O(nlogn) < O(n^2)
```

Recommended resources

- **[Book] Cracking the Coding Interview:** This is the bible for coding interviews. Now, I'd be irresponsible if I just name the book and not explain to you how to study it. This is my advice:
    - **Introduction:** Read all of it but put special emphasis on the section for BigO (section VI on my version).
    - **Data Structures:** Read all of it but put special emphasis on Arrays and Strings. Safely skip over Stacks and Queues, you won't need it. You won't probably need Trees and Graphs either, but I want you to become familiar with recursion and how to think in a recursive way.
    - **Concepts and Algorithms:** Go over chapters 5, 8 (skip Dynamic Programming) and read all the hints in appendix XIII.

- **Codility:** This will be the platform in which you will code the first algorithm assessment. I highly recommend studying their coding lessons, there's a high chance you'll be asked very similar problems from there.

- **LeetCode:** This site contains several problems from real interviews conducted at big companies. You won't necessarily be asked any of those for Toptal, but it’s worth studying with problems classified as easy.

- **HackerRank**, **InterviewCake**: Go over the basic questions. I don’t recommend paying for any of the plans, you won’t need much of what they offer. Consume all their free content, this will give you enough material to study.
- BigO: Checkout the BigO Cheatsheet and watch this video.

How to study

From my experience, there’s just one way to study algorithms. Here’s a 10 step process that will likely help you navigate problems:

1. Turn off your phone. Eliminate any distraction. You will put your brain in beast mode. Don’t dare to look at the solution until you have tried solving the problem for 30 minutes.
2. Take out paper and pen
3. Draw a base mock example to warm up your brain. If you need to deal with strings, write an example string. If the algorithm is about matrices, then draw a 3x3 mock matrix.
4. Start solving the problem as a human. Don’t think abstract at first (see Introspection section).

5. Write down the worst possible solution you can think of. Only pseudocode. No edge cases, no optimization.

6. Stop and think. How do you feel about your solution? Do you feel you can optimize?

7. Optimize. Don’t code just yet. These are my tricks to optimize:
    - Do you think you’re doing unnecessary rework? Are you visiting the data points a lot more than what you think?
    - Are you missing some piece of information from the problem?
    - Important: Can you use an auxiliary data structure, like a hash table, to improve time performance?

8. Cover edge cases. Online assessments are all about covering edge cases because they will be tested automatically. Are you including negative numbers in your reasoning? empty strings? base cases?

9. Code! Don’t write beautiful code. Online tools like Codility or the Toptal interviewers won’t care if your code is clean or not. They will only evaluate if your algorithm works and is efficient.

10. Record your AHA! moment. I’d say this is the most important step in preparation. After finishing your algorithm, write down your biggest realization. Something like “hashmaps took me from $O(n^2)$ to $O(n)$” or “I can create a cache variable to avoid revisiting the list”.

I credit some of this workflow to the book Cracking the Coding Interview.

Introspection. The technique.

I want to dedicate a few paragraphs to this technique which I call introspection. This is the first and most important step to solving an algorithm. It consists of watching your brain from the outside, seeing how it thinks, and how it solves the problem without any abstract thinking. You will likely always come up with the solution in your brain but then struggle to put it in code. This part of the process is essential in helping you overcome this.

Let’s say you need to find the largest pair of numbers in a list. How do you solve this as a human? Your brain doesn’t store anything in variables or even sort the list. Your brain sees the numbers and instantly knows what are the top two numbers. But, is it really instant, though?

Take a step back and think about it. How did you know the first number was not the largest? because you knew you had to see all the numbers before making a decision. This means there’s no better way to do it other than looping through all the numbers (in other words, it’s O(n) at best). Now, you know there are bigger numbers than the one you’re looking at. How? because your brain saw it and remembered it as the biggest number. This could be translated into “storing the biggest in a variable”. You can find the highest number, what about the second highest? What my brain did was to start all over again but ignoring the previous number. Great! we have a working algorithm now: we’ll visit each number and keep a reference of the highest. We'll then remove that number from the list and loop again to find the next highest number.

Is that the best you can do? definitely not but, at least, it's a working solution. It got you moving. It got your neurons up and working - they're warmed now to optimize.

Sample exercises

There's no value in reading the question and going straight to the answer. You'll lose your time. Put on the sweat and try to solve the problem yourself. It'll be uncomfortable but it gets easier with practice. Try to come up, at least, with the worst possible and most inefficient algorithm. Bonus points if you manage to get the best solution.

1. **Poker Chips:** Luigy works in a Casino and he gives customers poker chips in exchange for money. Find the minimum number of chips Luigy can use to match the customer requests. He has chips worth 100, 50, 25, 10, 5, 1. For example, for 126 Luigy should give 3 chips (100, 25, 1).

2. **The wedding:** Lina and Carlos are getting married. They both have an invitee list. Find out if they both want to invite the exact same people.

3. **Duplicate characters:** Find the top most repeated character in a string. For instance, given a string 'hello world', the top most

repeated character is ‘l’ because it can be found 3 times in the string.

4. **Balanced Brackets:** Given a mathematical operation, find out if the brackets are placed such that the equation is valid. There are 3 types of brackets: (,[,{. For example, `([4*{2+3}])` is balanced, but `{[3}` is unbalanced.

Solutions

1. Poker Chips

This is a problem you are likely solving on a daily basis when handling money. If someone is paying you and you need to give change, you normally optimize and don’t give all your coins. You probably find your biggest coins and see how many of them would sum up to a close number. Then you find your next biggest coins and add them to the previous coins so that they won’t surpass the total change. It’s almost unconscious, but this tells you something about how to solve this problem.

Let’s then follow that rationale to solve this problem:

1. Say you want to add chips up to 273 USD. What you would do in a normal scenario is first to find the biggest coin you have because, otherwise, you’d end up using more coins than necessary. Now, how do you find how many of those coins you can max use? Let’s

divide 273USD over 100USD to find out. This is 2.73 which means you can use 2 chips of 100USD and you'd be left with 73USD to fill.

2. So, what now? you start the problem again trying to fill 73USD but this time without the 100USD chip. You use the second-best of your options which is the 50USD chip. 73USD/50USD yields to 1.46 which means you can only use 1 coin of 50USD. How much do we have left? it's not 46USD. It's just the same 73USD minus 1 chip of 50USD which means we're left with 23USD now.

3. The next big coin is 25USD. Divide 23USD into 25USD and you get 0.92. This means you can't even use a 25USD coin. It's straightforward as 25 > 23 and you don't want to give away your money.

4. Let's try 10USD now. 23USD/10USD = 2.3. Use 2 coins of 10USD and you're left with 23 - 2*10 = 3USD.

5. Your last coin is 1USD. 3USD/1USD = 3 coins.

The answer to this problem would then be 8 (for chips 100, 100, 50, 10, 10, 1, 1, 1).

Implementation

```
function getNumChips(val) {
  let chips = [100, 50, 25, 10, 5, 1]
  let numChips = 0
  for (chip of chips) {
    let num = Math.floor(val / chip)
    val -= chip * num
    numChips += num
  }
  return numChips
}
```

Edge cases

After you have a working solution, think about some edge cases. This will help you polish the algorithm. We'll go through some of them, implementing them in code is left as an exercise for the reader.

- **Negative values:** ask your interviewer what you should do in this case as it totally depends on the spec. I'd go with an exception, as returning 0 is not quite accurate - not even with 0 coins you can fulfill a negative number.

- **Decimals:** ask your interviewer if the input range is only integers or if doubles are allowed too. You might want to round the number before going into the loop.

BigO Analysis

Think about this. For 273USD you solved the problem in 5 steps. Each one involved a mathematical operation. Mathematical operations are O(1) in time, it can be computed immediately by the machine. Now, how many steps would you need if the amount was 60.520 USD? You would divide by 100, find the max number of 100USD chips and then continue with 50USD. You see where this is going, it's the same 5 steps, the same 5 computations. If small and large numbers need just 5 calculations to solve the problem, this means we have a constant time of O(5). Now, as per the theory, for any k > 1, O(k*1) = O(1). With k=5 we have O(5) which means our algorithm is O(1) in time.

You might think, how come this is O(1) when you have a `for` cycle? Don't be confused by that. Space complexity only tells how the algorithm performs depending on the size of the input. We showed that it's the same for small and large inputs. Also, see that the loop is always constant over 5 elements. We could have easily done this without the loop, just did it for convenience.

AHA moment

You were asked to minimize the number of chips to fill a total amount but you end up maximizing the USD per chip in each step. This is your AHA moment: minimizing globally means maximizing locally. This is not true all the time but it's definitely a trick that can get you unblocked.

2. The wedding

This might seem like an easy problem at first. Verifying if two lists are equal is something you might have done a few hundreds of times before. But, are you doing it optimally?

Let’s see how your brain does this naturally. Take the first element from list 1 and find it in list 2. Now take the second element from list 1 and find it in list 2. Every time you find a matching pair, put a stroke over them. If you find an item that exists in list 1 but not in list 2, then you know they are not identical. If all the items in list 1 exist in list 2, but there are unstroked items in list 2, then both lists are not identical. If all items in list 1 are found in list 2 and there’s no item unmarked, then you can say both lists are identical.

We got it! We have a working algorithm, that’s the first step. We did the first introspection to recognize how our brain works. Now, is that the best we could do? Let’s get more technical and analyze the time complexity.

For every item in list 1, we’re looping in list 2 to find a matching pair. Be careful, this doesn’t mean our algorithm is $O(n^2)$. Indeed we have nested loops but, what does ‘n’ represent? Both lists can grow indistinguishable. If n and m are the sizes of list 1 and 2, correspondingly, then our algorithm is O(n*m) in time.

Optimization 1: Avoid rework

Let's optimize a bit. We're doing a lot of rework because we shouldn't visit the items in list 2 that have been found already. What if you skip them? You would then visit fewer elements each time you find a matching pair. Take an item from list 1, find it in list 2. Next time you won't look over m elements but instead over m - 1. Then, after that, you'll loop over m - 2, and so on. This means that the number of times you visit the elements in list 2 is m + (m-1) + (m-2) + ... + 1.

This last equation is known as a triangular number and can be expressed as $\frac{m(m+1)}{2}$. If you solve the parenthesis, this turns into $\frac{m^2+m}{2}$. As you can see, there's a quadratic term m^2 which means we're still dealing with quadratic times. In BigO notation this means our optimized algorithm is $O(n^2)$. Now, is this really the best we can do? Do you not feel like we're still doing rework by looping the list again and again on every iteration?

Optimization 2: Sort

We could also think about sorting both lists. If you sorted list 1 and list 2 then you could verify that all the values in the same indices are identical. Take an index `i` from 0 to list 1 length and verify that `list1[i] == list2[i]`. Is this better than the previous alternative? Let's see.

Sorting list 1 is O(nlogn) and sorting list 2 is O(mlogm). After this, you loop over all the items one last time. This means the final complexity in time is O(nlogn + mlogm + n). 'n' is the least dominant term here so we can ignore it (why? because n multiplied by logn is always greater than simply n). This means our algorithm is O(nlogn + mlogm). We went from n^2 to nlogn which means that, in fact, we did much better with this approach. But, do you not feel like sorting is a bit overkill for such a simple task? let's see if we can do better.

Optimization 3: Enter hashmaps

Let's try what I call "the copper bullet". It's not a silver bullet as it won't work for every problem but it's definitely useful for a big chunk of them. **Enter hashmaps**. As reviewed in previous sections, a hashmap is a data structure that lets you write and read in O(1) time. Let's see how to leverage this fact to improve our algorithm:

1. Loop over all elements of list 1. Store each element in a hashmap.
    - The key will be the invitee name and the value will be 1.
    - If a name already exists then increase the number by one.
2. Loop over all elements of list 2. Treat each element as the key of the hashmap and try to find its value.
    - If a value is found, then decrease the value by 1.
    - If no value is found, it means it wasn't put on the map, so both lists are not equal.
3. Loop over the values in the hashmap.
    - If some value is different than zero, it means one list has a value that appears more or fewer times than in the other list. This yields both lists not being identical.

  - If all values are zero, it means both lists are identical

Let’s put all this into code and see what our algorithm looks like:

```
function verifyAllEqual(list1, list2) {
  // fail fast if both lists are not the same length
  if (list1.length !== list2.length) return false;

  let map = {};

  // loop through list 1, fill the hashmap, count characters
  for (num of list1) {
    if (map[num] == null) {
      map[num] = 1;
    } else {
      map[num]++;
    }
  }

  // loop through list 2, decrease character count on hashmap
  for (num of list2) {
    if (map[num] == null) return false;
    map[num]--;
  }

  // if any count is greater than 0, both lists are not identical
  for (val in map) {
    if (map[val] !== 0) return false;
  }

  return true;
}
```

Edge cases

After you have a working solution, think about some edge cases. This will help you polish the algorithm. We'll go through some of them, implementing them in code is left as an exercise for the reader.

- How well does your algorithm handle empty lists? What if both lists are empty?
- What if one of the lists is much bigger than the other? would you find it better to start looping over the short one? or the large one?

BigO Analysis

We did BigO time complexity analysis for all the previous algorithms. We'll focus now on the last optimization:

1. We first loop over all elements of list 1 and visit each element just once. This is O(n).
2. When we finish that, we go over list 1 and do the same. This is O(m).
3. Finally, we loop over the values in the hashmap which, in the worst-case scenario with no duplicates, has n elements (i.e all values from list 1 stored in the map). This yields to another O(n).
4. The time complexity is then O(n + m + n) => O(2n + m) => O(n + m).

5. Now, notice that the second loop terminates quickly when no matching element is found in list 1. This means that, at most, you only loop through n elements in the second list, even when its length is ‘m’. This would finally yield a time complexity of O(n + n) => O(2n).

6. In BigO notation we only deal with the shape of the curve (constant, linear, quadratic, etc) so O(k * n) => O(n). Our algorithm is O(n) in time.

How do you know this is the best you can do? To know that both lists are equal you should, at least, know what are the elements in the list. How do you know them? by visiting them. This means you should, at least, visit every element which is exactly O(n). You can’t do better than that, because you can’t leave any element unvisited.

AHA moment

We realized we were doing rework as we were visiting each element a lot of times. But, what really took us to the next level, was the hashmap. Hashmaps are really the copper bullet for most algorithms as the hashing power lets you write and read in O(1).

3. Duplicate characters

Almost anyone can solve this problem in their mind for short sentences. Let’s see how our brain works, no algorithms involved. You start skimming through the words and get a feel of which characters are candidates for repetition. How do you know that? because your short-term memory helps you with retention until something becomes more and more familiar. But, still, though, you can't easily tell if a character has been repeated 5 or 6 times until you count them. You need to keep a record of what characters are repeated and how many times.

Following the rationale above, we can outline a rough working algorithm:

- For every character, visit the rest of the string and count how the number of times it's repeated.
- Keep track of the characters and the number of visits
- Finish the algorithm by finding the max of the counts

For the last exercise, you already know you can use maps to store key-value pairs. We can then leverage this data structure to keep out character count. Let's see how this looks like in pseudo-code:

```
counter = map()

# build counter
for i = 0 and i < input.size # for every character...
  char_i = input[i]
  if !counter.has(char_i) # ...if we haven't seen this character
before...
    for j = i + 1 and j < input.size # ...visit the rest of the
string...
      char_j = input[j]
      if char_i == char_j: # coincidence found? update counter
        counter.char_i++
# find max
top_repeated = ''
for char in counter:
  if counter.get(char) > counter.get(top_repeated):
    top_repeated = char

return top_repeated
```

Optimization 1: Avoid rework

Our first would work just fine. Let's see how expensive it is. For every character, we loop through every other character. This means our first run will visit N elements. The next iteration will visit N-1. The next one will be N-2. You get it. Hence, the number of steps our algorithm needs to terminate is N + (N-1) + (N-2) + ... + 1. As shown in problem #2, this behavior means our algorithm runs in quadratic time: O(n^2).

Can we do better? Do you not feel like we're doing some rework over here? We're running an inner loop to count coincidences, is that necessary? Can't we just keep a counter map with the number of coincidences of each character? Think about it for a second. Your brain is implicitly keeping a map with the number of times it sees a letter. Can't we just do the same? We're effectively changing the problem from "find the most repeated char" to "count the number of occurrences for each char".

Let's see how our final algorithm looks like. Code snippet is shown in JavaScript.

```
function findTopRepeated(sentence) {
  // Build counter map
  const counterMap = {}
  for (const char of sentence) {
    if (counterMap[char]) {
      counterMap[char]++
    } else {
      counterMap[char] = 1
    }
  }

  // Find max
  let top = ''
  for (const char in counterMap) {
    if (!counterMap[top] || counterMap[char] > counterMap[top]) {
      top = char
    }
  }
  return top
}
```

Optimization 2: Merging loops

At this point, you have the most performant solution in terms of BigO behavior (we discuss this in-depth later). Now, is it really necessary to run two separate loops? can we keep track of the topmost repeated char while we count the coincidences? Yes, it makes no difference. You'll end up counting them all anyways. Let's see what it looks like now.

```
function findTopRepeated(sentence) {
  // Build counter map and keep a running max
  const counterMap = {}
  let top = ''

  for (const char of sentence) {
    if (counterMap[char]) {
      counterMap[char]++
    } else {
      counterMap[char] = 1
    }

    if (!counterMap[top] || counterMap[char] > counterMap[top]) {
      top = char
    }
  }

  return top
}
```

Now, this one feels much better. We went from 2 loops to just 1 in no time, it's almost free profit.

Edge cases

This is an exercise for you. Think about what edge cases you can encounter and solve them. What happens if there's no character repeated? The problem didn't say anything about it, it's up to you how to solve it. What happens with blank spaces? will that count as a character? if not, then deal with it, make sure you don't count empty characters. You can even clean the sentence before the algorithm, it'll make it run faster.

BigO Analysis

Just by looking at the code, we can tell we're no longer in the quadratic realm because we got rid of the nested loops. We're looping through the sentence just once, char by char. At first, we had 2 loops which, in the worst-case scenario (no char repeated), would end up being O(2n). As per BigO properties, O(k*n) is equivalent to O(n) so we have linear time. Please remember BigO is a mere measure of how your algorithm performs relative to how the input grows. It doesn't imply anything about the absolute run time. If O(n) takes 100ms, O(2n) will take 200ms. This is irrelevant for performance analysis, though. After our second optimization, we got rid of the second loop and got an effective, O(n) linear time.

To find the space complexity we need to think about the data structures we introduced to solve the problem. In this case, we're saving a counter map which, in the worst case, would keep a copy of all the characters in the input string. Does this mean we have O(n) space complexity? Not so fast. The maximum size of our map is the number of characters in the alphabet. Let's say it's 27. Therefore our map will have no more than 27 key-value pairs. The space complexity is then O(27), which is effectively equivalent to O(1).

AHA moment

You might think the AHA moment for this problem was using the hashmap, same as problem #2. But, it was actually when we switched the problem and paraphrased it in a different way; that's when we found the most performant solution. We stopped thinking about "finding the topmost repeated" and started thinking about "logging the number of occurrences per character". Then we got ourselves in front of a well-known problem: finding the max number in a list.

Remember this: Always try to boil problems down to subproblems that you already know how to solve. In the future, you might find problems that require you to find repeated elements in lists. You already know it can be done in O(n) with hashmaps. That's going to be familiar and will put you one step ahead.

4. Balanced Brackets

Let's first draw two basic problems to get us warmed up.

- **Balanced:** `(4+[1-(-2)])`
- **Unbalanced:** `[{(5+2)]`

We'll put names to the brackets just for convenience. () is type P, [] is type Q, and {} is type R.

Right. Let's now try to solve the problem in our brains. Our eyes ignore all the opening brackets and focus only on the closing ones. I get to the first closing bracket of type P and think "ok, this one should close the last open bracket I saw". I keep moving and see another closing bracket, this time of type Q. "right, this one should close the last open bracket I saw, and that last bracket should be of type Q". I also tend to skip those inner brackets that have already been closed, because they are already resolved anyways.

Let’s try to implement this same algorithm in pseudocode. Notice we’re somehow keeping a list in our memory. This means we need to add the brackets to a list. Also, notice we tend to skip the inner brackets that have been resolved. How do we translate this to code? well, “skipping it” means ignoring it, so we don’t really need the open brackets that have been resolved; we’ll remove resolved brackets.

```
brackets = []
for every c in equation
  if c is an open bracket
    brackets.add(c)
  if c is a closed bracket
    if type of c == type of last(brackets)
      brackets.remove_last()
```

For the equation to be valid, all brackets should have been resolved. Also, as you saw, we’re removing resolved brackets from the list. This means that, for all the brackets to be balanced, we should expect our list to have length 0 at the end of the algorithm. Otherwise, there was a bracket that didn’t get closed. This is a great start. It won’t cover all the edge cases (like an equation with only closing brackets), but it’ll be fine for a first iteration.

Can we find out something more optimal? think about BigO time complexity for a minute. Our algorithm is looping through all the characters once. This means our algorithm is O(n). For someone to tell if an equation is valid or not, they need to look at the whole equation. No character should be skipped. This means there's nothing better than "looking at every character". Therefore, we can be confident there's no better way to do this than O(n).

Enter Stacks

This is a classic example to get you started with Stacks. A Stack is a LIFO (last in, first out) data structure that lets you push and pop elements in O(1). The elements are pushed at the end of the list and are also popped from the end of the list. Think like a stack of poker chips. Remember we were pushing and removing brackets from the end of a list? Well, it seems like we can make it cleaner with a Stack.

Let's assume we have a data structure called Stack with methods push and pop which will add and remove from the list tail, correspondingly. We'll also have a last method which will give us the last element without removing it from the list. Assume we have auxiliary functions `isOpeningBracket` and `isClosingBracket` which will validate if a character is opening or closing, correspondingly. Finally, we'll need a function isValidPair which takes two characters and validates if they are any of (), {} or []. This is how our algorithm would look like in real code.

```
function validateBrackets(equation) {
  const stack = new Stack();
  for (letter of equation) {
    if (isOpeningBracket(letter)) {
      stack.push(letter);
    }
    if (isClosingBracket(letter)) {
      if (isValidPair(stack.last(), letter)) {
        stack.pop();
      }
    }
  }
  return stack.length === 0;
}
```

Note: JavaScript has `push` and `pop` methods in vanilla Arrays so we don't really need to implement the Stack data structure. This is just for explanation purposes.

Edge cases

After you write your first algorithm, make sure to come up with contrived examples with weird edge cases. Let's look at this one, for instance:]]]]]]. How would our algorithm behave for such an unbalanced expression? There's no opening bracket so our stack will always be empty. Every time the loop lands on a closing bracket it will check if it's a valid pair with the last stack element, which is null. This will, of course, be false and therefore we pass on to the next iteration. At the end of the algorithm, our stack length is indeed 0, so our algorithm will classify this as true.

Let's fix this. Check this out: Every closing bracket is a decisive point in our algorithm. If it doesn't pass our test then we should terminate and return `false` right away. Did you get it? There's no point in looping any further if one closing bracket didn't match its opening counterpart. With this in mind, let's modify our algorithm to fail early when a valid pair is not found.

```javascript
function validateBrackets(equation) {
  const stack = new Stack();
  for (c of equation) {
    if (isOpeningBracket(c)) {
      stack.push(c);
    }
    if (isClosingBracket(c) && !isValidPair(stack.pop(), c)) {
      return false;
    }
  }
  return stack.length === 0;
}
```

Notice how we don't need 'last' anymore. We pop right away. If it's a valid bracket, we were going to remove it from the stack anyway; if it's not valid, we terminate the algorithm.

BigO Analysis

As analyzed earlier, our algorithm is O(n) in time because it visits each element exactly once. We also showed there's no better and more performant way to solve this problem. What's the space complexity? The worst-case scenario is we find an equation with only opening brackets. This means we'd end up with a stack of n elements. Our algorithm is then O(n) in space.

AHA moment

The aha moment is realizing we don’t need resolved pairs in the equation anymore. When we don’t need anything, we’re effectively removing it. When you see yourself ignoring something in your brain, that probably means removing it all over. In our case, it all led to us using a Stack and popping elements all over.

4
Test Project

The general belief is that, when you get to this point, you're pretty much a Toptaler already. I've heard this should be a straightforward step for someone who just went through the full application and cleared all the filters. I, personally, don't think this is the case.

The advantage, though, is that you have plenty of time to work on this by yourself. The drawback is you may not know it all and you'll need to research a lot. But that's fun too. Let's see what to expect and how to prepare for this step.

What's the test project?

This is the last step of the process. It consists of 2 parts. First, they will send you a project spec and you will have 2 weeks to build it as if it was for a client. Second, you'll interview real-time with a toptaler to showcase your project and support all your decisions. They will ask you all kinds of questions ranging from fundamentals to project-specific stuff.

This step is supposed to filter out 0.2% more candidates. This means 32 out of 1000 people get to this point, and 30 of them will pass. Now, I need to emphasize this: do not underestimate this phase. I myself have referred overconfident people who play hard to get to this point only to see them fail. Make sure you deliver world-class work here.

Why do they ask me for this?

This can easily be the most important, critical part of the process. This is where you show off all your experience and your true engineering skills. Solving a real-world problem is the only way you have to demonstrate you can write good code. You'll be pushed to think about all aspects of an app in a clean, maintainable, and scalable way. It's ultimately very similar to what you'll be doing on a daily basis.

How to prepare

It can be daunting at first to tackle a problem of this magnitude. You will probably feel unqualified from the very get-go. You might be good at frontend development but not feel confident about building an API. Or, you might be strong on the backend but lack UI/UX skills to make something look decent. This is normal, you can't excel at everything.

The key here is to know what to study and where to focus. In the next few sections, I'll point out the main key concepts you should study in order to deliver a great project. We'll start reviewing a sample project and we'll go piece by piece discussing what's expected from you and what you should learn.

Sample Exercise

I'll focus here on what you'll probably be applying for: web development. The project you'll have to build is probably either a full-stack or a frontend-only project. It depends on whether you use a backend-as-a-service to build your API (ie Firebase), or create your own backend using your preferred language. You'll probably be given a very vague definition of a problem and this will be on purpose: they want to test how well you adapt to uncertainty.

Let's see what a problem statement looks like. You must build an application to rate books:

- Users can add and remove books on their account
- New users must sign up with email and password. Email confirmation is not mandatory.
- Existing users log in via email and password
- Every book consists of a name, author and a numeric rating (from 1 to 5)
- Users can update the rating of a book on their account
- Users can filter their books by rating

- Create an admin role whose dashboard shows all the books from all users
- Admins can only view ratings but not edit them
- Admins can filter books by created date
- Clean and good looking UI/UX is expected
- The API must be Restful. All actions should be executed via API.
- Optional. Use a backend-as-a-service, like Firebase, if you don't know how to implement a backend. In any case, your interviewer will probably assess your knowledge in APIs and make you test it via Postman or cURL.

You must deliver a fully functional app as if it was for a client. You'll probably be given a Gitlab repo with access to push your code. They won't review your commit messages but I recommend to be professional with that. It's the little details that count.

We'll go over what you should focus on both the frontend and the backend.

Frontend

- It's expected that you build a **SPA** (Single Page Application) so you'll need to rely on some framework like Angular, React, or Vue.

- Make sure to use a battle-tested, proven **UI library** like Material components. All the major frameworks have a library for this: Angular Material, Material-UI (React), and Vue Material.

- Use a common **CSS framework**. This is not the time to build your own framework. Just use whatever works like Bootstrap or Tailwind.

- Implement **good routing.** I recommend every page in your app to have an individual reachable URL, even login and signup.

- Write **authentication guards**. They should automatically route your users to login/signup when accessing forbidden routes. Make sure you understand this concept and talk about it in the interview.

- **Write E2E tests**. At least 2 or 3, they are not normally mandatory but will show you think about maintainable code. I hugely recommend doing this with Cypress. It's the fastest and easiest way to write E2E tests.

- Implement **network interceptors** to add your auth tokens in a clean way. You can achieve this with HTTP interceptors (Angular), Axios interceptors (React) for VueResource (Vue).

- Make sure your app is mobile **responsive**. This will tell your interviewer you think mobile-first, which is critical in the web scene today.

- Test in at least **3 major browsers**. Leveraging a UI component library and a CSS framework is a great way to mitigate risks with this. I recommend testing in Chrome, Firefox, and either Safari or Edge, depending on your OS.

- **Go modular.** Make sure you build independent components with very clear inputs and outputs. If your component is growing too large, consider dividing it into smaller components. This is critical because your interviewer will ask you to add features in the live screening. If you have spaghetti code all over the codebase, you'll find it very hard to do this fast.

- **Good folder architecture.** Make sure you show a clean codebase with well-defined folders: models, utils, shared, components, assets.

- **Use TypeScript.** It's first-class in Angular but, if you work on React or Vue, I highly recommend using the TypeScript starter. This will make your code look much cleaner. You can also tell your interviewer this makes it more maintainable in the long term.

- **Styling.** Avoid inline styles in your HTML <div style="color: red">. Style classes only .my-class { color: blue }, not HTML tags directly. If using CSS-in-JS, use semantic names for your styles.

- Always aim to do **sorting and filtering on the backend**. Doing this in the frontend will probably tell your interviewer you don't think enough about scalability.

Backend

- Use your **favorite framework and programming language** to build the API. I recommend sticking to the known ones: Python, Java, NodeJS, PHP, .NET, Go.

- No matter what you do with your API, always go **Restful**. Research about how a truly Restful API should look like. Avoid having routes like `POST /api/add-book` or `GET /api/filter-by-rating?:rating`.

- Make sure you understand the difference between URL **parameters** (`/users/:id`) and **query strings** (`/users?sortBy=:order`) and when to use them.

- Handle **authentication** properly. Do not store plain text passwords. Look for 'salt hashing', learn to manage secrets in the server. Study this over and over again until you understand how clean auth works.

- Handle **authorization** properly. Look online on how to best organize roles in a database with an authorization scheme.

- **Choose your DB** properly. You'll have to support your decision in the interview. Whether you go with SQL or NoSQL, make sure you read the differences beforehand and be ready to answer questions like "why MySQL and not Postgres? why is this app better for a NoSQL?"

- **Write unit tests**. If working on NodeJS there are several resources online on how to test your Mongoose controllers with Mocha and Sinon. If working with Python/Django, there's a complete section on Testing in their docs. Remember: just do a few tests, don't spend too much here. You can say you focused on the critical components and you moved on. This will at least show you care about testing; it's critical in picturing you as an expert.

- **ORM**. It's highly advised to leverage an ORM like Mongoose or Django models to manage DB objects. I wouldn't recommend having plain SQL statements on your codebase, at least for this project. You won't probably be asked to show off your SQL skills so having them can be detrimental.

- **Dev vs Prod settings**. You won't be asked to deploy your project, don't spend time on it. However, make sure you understand what configs can be made public and which ones should be secrets. This typically involves DB passwords, API Keys, or any other credentials. This data should not be committed to your repo and should instead be kept secret in the prod server. Again, you don't have to do any of this, but it's critical to understand this concept.

- Learn to use **Postman**. You'll be asked to open this, or cURL, in your interview. Be prepared to create a POST request with authorization headers, to create a JSON payload, or to analyze the headers of a GET request.

- Use **middlewares**. When you use middlewares you show you understand the different layers of an app in the backend. You can use middlewares to verify authentication or authorization, manage sessions, append headers, write logs, cookie management, etc.

- **Headers**. Using advanced headers will make you look ahead of the curve. Believe or not, most devs won't ever go into configuring security headers as they trust their framework so much. I'd stick with headers like CSP and disabling X-powered-by. Make sure to talk about this in your interview.

Tips for the interview

You can't just build a great project and go unprepared for the call. I recommend these specific steps to prepare your screening:

1. Make sure you can run the project right after the first git clone.

2. They will probably ask you to use the existing project on your local, so have it up and running for the call.

3. Get Postman (or your favorite HTTP client) ready for the interview.

4. Be confident but also authentic. If you don't know a concept just say you know something about it, mention what you know, but be willing to say "I don't know at this time, I would just search for that".

5. Practice some dummy questions before the interview: add a new role, add a new object to the DB, add a new data filter, add a new dialog on the frontend, etc.

6. Always think you’re presenting to a customer. Don’t take it as yet another interview but as a demo to a real client.