Algorithm Design Canvas

Introduction

Some of us have been solving algorithmic problems for fun & profit for almost 15 years. Very early on we developed a systematic process for approaching them.

Meet the Algorithm Design Canvas

Coupled with the right kind of theoretical knowledge, the Algorithm Design Canvas is going to give you everything you need to master algorithm design questions.

What's next?

Now let's start by looking at the Algorithm Design Canvas in more details.

What is the canvas?

The Algorithm Design Canvas captures our process for tackling algorithm design problems.

It's the most convenient way to represent algorithmic thinking. Every algorithmic problem, big or small, easy or hard, should eventually end up as a completed canvas.

The 5 areas of the Canvas

The Algorithm Design Canvas

Problem name: _____

Constraints 🔗

Ideas 💡

Test Cases ✔

Code ‹/›

The Canvas contains 5 major areas: Constraints, Ideas, Complexities, Code, and Tests. Taken together, they capture everything you need to worry about when it comes to algorithm design problems. In further sections, we're going to cover what each area represents, as well as many tips and tricks about filling them in. We'll also work through some examples that let you see the Canvas in action.

Area #1: Constraints

The Constraints area is where you fill in all constraints of the problem. This includes things like how large the input array can be, can the input string contain Unicode characters, is the robot allowed to make diagonal moves in the maze, can the graph have negative edges, etc.

Your very first task when analyzing an algorithm design problem is to uncover all its constraints and write them down in this area.

Area #2: Ideas

After you've identified all constraints, you go into idea generation. Typically during an interview you discuss 1 to 3 ideas. Often times you start with one, explain it to the interviewer, and then move on to a better idea.

Your next goal is to fill in a succinct description of your idea: short and sweet so that any interviewer is able to understand it.

Area #3: Complexities

Each idea has two corresponding Complexity areas: Time and Memory. For every algorithm you describe, you will need to be able to estimate its time and memory complexity. Further in this course we will talk about the Time vs Memory trade-off and why it's key to any algorithm design problem.

Area #4: Code

After you've identified the problem's constraints, discussed a few ideas, analyzed their complexities, and found one that both you and your interviewer think is worth being implemented, you finally go into writing code.

Writing code at interviews is fundamentally different from writing code in your IDE.

Area #5: Tests

Finally, you move on to writing test cases and testing your code. Many people completely ignore this step. This is not smart at all.

That's it.

Now go ahead and print some Canvases!

The best thing to do at this point is to print a bunch of copies of the Canvas: they'll come in handy as you prepare for your technical interview.

Print the Algorithm Design Canvas

We hope you find the Canvas as indispensable as we've found it over the years. Use it wisely, and use it often!

What's next?

Let's continue to the first Canvas area: the often overlooked topic of identifying all constraints of a problem!

Task constraints

You never want to solve a problem that's ill-defined. That's why the very first thing you should turn your head to when it comes to algorithm design problems is the problem's constraints.

The importance of constraints

Developing an algorithm that sorts 50 numbers is going to be fundamentally different from developing an algorithm that sorts 5 billion strings, each one 1 million characters long. So if I told you "design an algorithm that sorts an array", you'd better not start coding a solution right away. Among other things, you need to understand things like what's in the array, and how large the array can get.

You're probably aware that interview problems have a bunch of constraints, which tell you how efficient your solution should be. What is less obvious is that interviewers don't always give you all the information needed. They expect you to be asking clarifying

questions. Don't be alarmed if not everything is clear in a problem statement, this is your chance to make it so.

How do you figure out the right questions?

Generally speaking, think about all the things that matter to your solution. For example, make sure you know the minimum and maximum possible values for any key value in the statement. These may affect the performance of your solution directly. If there are arrays of values you need to know the range of these values, too. They can be numbers, chars, geometric figures, etc. Some of these should come naturally to you once you hear the statement. If it says that there are N numbers and nothing else is mentioned about N, ask how big it can be. Others could come when you start designing your solution. If a given value is important for the complexity of your solution, don't hesitate to ask about it. Never assume things on your own.

The Algorithm Design Canvas helps you collect the needed information at an interview. It has a section where you need to fill in all the important constraints in a problem. If you use it while practicing you will get into the habit of making sure you know everything that you need to design an efficient solution.

Filling in the Constraints box is nothing more than asking the interviewer the right questions. Every problem's different, but to get you started we've compiled a pretty extensive list of common things you should keep an eye on. All of them are included in the Common Constraints Handout. Feel free to download it and print it out: It will come in handy!

Get the [Common Constraints Handout (PDF)](#)

The handout above will help you get a good idea of what is important in most interview problems. In addition to that, you will have to practice with real problems to gain experience. For this you will be able to use the problems we recommend further in the course.

What's next?

Once we know all we need about the problem, let's look at some strategies for designing a solution.

Idea generation

One of the goals of this course is to teach you how to solve new problems. This will be the topic of the current section. We believe that it is much better to learn how to design solutions instead of trying to cover all interview questions that exist. There are a few reasons for that.

First of all, new interview questions get created all the time. It is virtually impossible to be well-prepared for all of them. Second, even if you know the solutions to many problems you need to be able to analyze these solutions, have a discussion about them and be able to tweak them in case the interviewer decides to modify the question in some way.

One final reason is that employers are looking for people who can think of solutions on the spot. This is why it is very useful to learn how to be such a person.

So, how can you achieve that? In this section we will discuss several strategies for solving algorithmic problems. They will give you a small framework for approaching a problem and coming up with a good solution.

Remember that practicing solving problems is the only way to really get in good shape. These strategies are one tool that you can use but you will become good with it only if you devote enough time applying it. Actually, with time you will notice various patterns in the problems and it will become much easier for you to crack them.

The following strategies are popular among people participating in programming contests. Since the algorithmic interview questions are very similar to these, we believe that they are very relevant here, too.

Simplify the task

Let's look at the following question:

"A map of streets is given, which has the shape of a rectangular grid with N columns and M rows. At the intersections of these streets there are people. They all want to meet at one single intersection. The goal is to choose such an intersection, which minimizes the total walking distance of all people. Remember that they can only walk along the streets (the so called "Manhattan distance")."

So, how can we approach this problem?

Imagine that we only have one street and people are at various positions on that street. They will be looking for an optimal place to meet on this street. Can you solve this problem? It turns out to be much easier than the 2D version. You just have to find the

median value of all people's positions on the street and this is an optimal answer. We'll leave it to you to prove it.

Now, if we go back to the original problem we can notice that finding the X and Y coordinates of the meeting point are two independent tasks. This is because of the way people move - Manhattan distance. This leads us to the final conclusion that we have to solve two 1D problems and this will give us the final answer. As we already know how to solve the 1D case, we are done.

This strategy allows you to start thinking about a simpler version of the problem and to draw some conclusions about how to solve the original problem.

Try a few examples

We've noticed that candidates rarely test with examples other than the one given by the interviewer. Sometimes, however, this helps a lot. You may start noticing patterns if you try to solve a few sample inputs that you create. It is ok to tell the interviewer that you would like to try writing down a few examples in order to try to find some pattern. Then do it quickly and see where it leads you.

Here is a sample problem:

"There are N+1 parking spots, numbered from 0 to N. There are N cars numbered from 1 to N parked in various parking spots with one left empty. Reorder the cars so that car #1 is in spot #1, car #2 is in spot #2 and so on. Spot #0 will remain empty. The only allowed operation is to take a car and move it to the free spot."

This problem is not hard but at first seems intimidating. Write down 5 different examples and try to order the cars on a sheet of paper. See if you can notice a pattern in how it happens.

Think of suitable data structures

For some problems it is more or less apparent that some sort of data structure will do the job. If you start to get this feeling think about the data structures you know about and try to apply them and see if they fit. For example you can consider the data structures we cover in the algorithmic topics section.

Here is an example question:

"Design a data structure, which supports several operations: insert a number with O(logN), return the median element with O(1), delete the median element with O(logN), where N is the number of elements in the data structure."

Here it is pretty obvious that a data structure is involved. What could we use to solve the problem? Which data structure has similar characteristics? Things like arrays, stack and vectors are far from these requirements. Perhaps some sort of a binary tree could do as they usually support logarithmic insert and remove operations.

Heaps do that but they either return the minimum or maximum element, not the median. Hm, but this is very close, if only we could get the median. What if we use two heaps, one stores the one half smallest numbers and the other is for the other half biggest numbers. Then some number in the middle is the median. As mentioned above one of the heaps could hold the maximum and the other the minimum element. This should be enough to return the median in constant time. We will leave the details to you to figure out.

In summary, if you get a feeling that a data structure could be the solution to a problem you are given, think a bit about the ones you know. You may have to combine two or more to get to the correct solution.

Think about related problems that you know.

This is a simple idea and could help if nothing else helps. Since you've been practicing hard for your interviews you surely came across many different problems. If you see a problem and cannot think of a solution, try to remember another problem, which looks like it. If there is such, think if its solution can somehow be adjusted to work for the problem at hand. This can sometimes mislead you but many problems are related, so it could also get you out of the situation.

Read carefully these strategies and try to apply them to the problems you face. In this course there are many recommended problems and you can do that with all of them. The Algorithm Design Canvas will help you write down the ideas that come to you.

TopCoder tutorials also provide a useful resource on this topic, which you may find worth reading: How to Find a Solution.

If you have other strategies for solving interview problems we will be more than happy to hear about them.

Summary

In this section you learned:

- It is important to learn to solve any problem instead of knowing all of them by heart.
- There is a set of well-known strategies for approaching interview problems. We look at them and use some of them for an example problem.

What's next?

Designing solutions to a problem inevitably leads us to talking about time and space complexity. This is a very important topic. The next section will teach you more about it. So, let's dive into some complexity!

Complexity

For every idea you have at an interview, it's super important to be able to evaluate its efficiency. Efficiency is measured by computing the underlying algorithm's time and memory complexity.

Why is complexity so important?

Solving problems is not just about finding a way to compute the correct answer. The solution should work quickly enough and with reasonable memory usage. Otherwise, even the smartest solution is not useful. Therefore, you have to show the interviewer two things:

1. Your solution is good enough in terms of speed and memory.
2. You are capable of evaluating the time and memory complexity of various algorithmic solutions.

Usually, if you fail to evaluate these parameters of your solutions this will reduce your chances of passing a technical interview. The interviewer needs to know that if you get hired, you will be able to choose wisely the solutions that you implement.

Essential readings

If you feel unsure about your knowledge in this topic we have prepared a separate section with some explanations for you. It is called Computational Complexity and comes right after this section.

In it you will find a few lessons and links to very useful resources that will give you more information. In the lessons we provide you will also find some guidance about how to discuss complexities at your tech interviews.

We've also compiled the Complexity Handout, which includes the complexities of many of the best-known algorithms. Like with all handouts, feel free to print it out and reference it. However, keep in mind that there's zero value in you just memorizing these. Instead, read the theoretical articles and make sure you can come up with these complexities yourself.

Download the [Complexity Handout (PDF)](#).

What's next

In the next lesson, we'll finally turn our attention to writing some code! Like with all other areas of the Canvas, there are some rules on how to do it right, as well as some common misconceptions about writing code during interviews. Read on!

Writing the code

At this point, you've already nailed the constraints of a problem, iterated on a few ideas, evaluated their complexities and eventually picked the one you're going to implement. This is a great place to be!

We see many people who jump straight into coding, ignoring all previous steps. This is bad in real life, and it's bad when done at your interview. Never, ever jump straight into coding before having thought about and discussed constraints, ideas and complexities with your interviewer.

Now that we took that out of our system, let's focus on how you code at the interview.

The one thing about coding (other than rushing into it too quickly) that many people struggle with is that coding in your IDE is not the same as coding on a whiteboard / a shared document / using some online system (in short: "outside your IDE"). As engineers, we've become so accustomed to relying on our IDEs (which is not bad), that when presented with a blank sheet of paper, we are lost. While this may have been OK at your day job, at an interview it simply doesn't work.

This is why you need special preparation for "interview coding". Say hello to the big juicy "Code" section of the Canvas, waiting to be filled in. At first it may be a bit tedious to write

your code on paper, but very quickly you get used to it and you become more efficient. And the great news is that being able to code on a sheet of paper will not only boost your interviewing skills - it will also make you a better engineer overall.

Some of the key things to keep in mind when coding outside your IDE:

- **Think before you code.** Especially if you're coding on a sheet of paper (where there's no "undo"), if you're not careful everything could become very messy very quickly
- Coding outside of an IDE does not give you permission to stop abiding by good code style. **Make sure you name your variables properly**, indent nicely, write clean code, etc. etc.
- It's even more important to **decompose your code** into small logical pieces and **NOT copy-paste**
- **Read your code** multiple times before you claim it's ready. You don't have the luxury to compile, see if it compiles, run, see if it runs, and then debug for 4 hours. Your code needs to work off the bat.

That's it. Now say goodbye to your IDE for a while, and off to practicing!

Summary

In this section you learned:

- When should you start coding at the interview?
- Coding for an interview is not like coding in your IDE. How can you prepare for that best?
- Look at important tips for how to write the code. We also write the code for an example problem.

What's next?

Let's wrap up the essential steps of a tech interview with a very important one - testing. You can never be 100% sure that your solution is correct unless you run it through some tests. Doing that has multiple positive effects at an interview. See how to test your solutions.

Testing your code

Once your code is written, you don't just lift your pen, say "I'm ready" and leave. A great next step is to verify it with several small test cases. And what would be even better is if you could tell your interviewer how you would extensively unit test your code beyond these sample tests.

Why?

Showing that you know and care about testing is a great advantage. It demonstrates that you fundamentally understand the important part testing plays in the software development process. It gives the interviewer confidence that once hired, you're not going to start writing buggy code and ship it to production, and instead you're going to write unit tests, review your code, work through example scenarios, etc.

Sample tests vs Extensive tests

Before we go into what makes a good test case, we're going to make one clarification around testing at interviews. There are two kinds of testing you may be asked to do (or decide to do) at an interview.

We call the first kind **"sample"** tests: these are small examples that you can feed to your code, and run through it line by line to make sure it works. You may want to do this once your code is written, or your interviewer may ask you to do it. The keyword here is "small": not "simple" or "trivial".

Alternatively, your interviewer may ask you to do more **"extensive"** testing, where they ask you to come up with some good test cases for your solution. Typically, you will not be asked to run them step-by-step. It is more of a mental test design exercise to demonstrate whether you're skilled at unit testing your code.

Fundamentally, these are very different. We'll address each kind separately.

Extensive testing

Let's start with the bigger topic: what makes for a good test case. If you were to compile an extensive set of tests for your solution, what should these be? Here is a good list to get you started.

- **Edge cases**: Remember that "Constraints" section that we filled in? It's going to come in very handy. Design cases that make sure the code works when the min and/or max values of the constraints are hit. This includes negative numbers, empty arrays, empty strings, etc.
- Cases where there's **no solution**: To make sure the code does the right thing (hopefully you know what it is)
- **Non-trivial functional tests**: these depend very much on the problem. They would test the internal logic of the solution to make sure the algorithm works correctly.
- **Randomized tests**: this makes sure your code works well in the "average" case, as opposed to only working well on human-generated tests (where there's inherent bias).

- **Load testing**: Test your code with as much data as allowed by the constraints. These test your code against being very slow or taking up too much memory.

A good "test set" is a well-balanced combination of the above types. It will include tests that cover most edge cases, a few non-trivial functional tests, and then a series of random tests. These will make sure the code is solid and functionally correct. Finally, some load tests make sure the algorithm works well on the largest and most resource-demanding tests.

Sample tests

Sample tests are small tests that you run your code on at the interview to make sure it is solid. Now that we've covered the major kinds of tests you may design, which ones should you use as sample tests?

Typically, we stay away from randomized tests and load tests during interviews, for obvious reasons. Instead, we like choosing a small-scale version of a non-trivial functional test, to make sure the code does the right thing. Then, we look at how the code would react to several edge cases, and finally think about whether the code would work well if no solution can be found.

This combination (non-trivial functional + edge + no solution) tends to be the most effective. For the amount of time it takes to design the tests and to run them on your code on a sheet of paper, it gives you the highest certainty in your code.

How to prepare

There are three good practice activities that will significantly improve your testing skills.

1. Number one is to make sure you understand the fundamental kinds of algorithm tests (mentioned above).
2. The second step is to pay attention to the test cases for the TopCoder problems. Each problem has multiple "open" test cases (part of the problem statement). Pay attention to what cases they're trying to cover. Additionally, when you run the System tests you're shown all test cases picked by the problem authors. Make sure you review some of those too.
3. The third step is to always fill in the "Test cases" section of the Algorithm Design Canvas. See what scenarios you can come up with.

No matter how you prepare, the key message here is to not ignore testing your solution. Reading your code once it's written and trying it out with a well-chosen test case is going to do wonders in finding everything from silly typos to actual bugs. Choose the test cases

carefully, so that this does not turn into a huge time sink. You will learn to do that with some practice.

Summary

In this section you learned:

- Why is testing at tech interviews important?
- What tests should you use?
- How can you learn to design the right tests for your solutions?

What's next?

So far we've covered the important steps when solving algorithmic problems at tech interviews. By this time you have a framework for dealing with them from beginning to end. Now you need to boost your skills with some theory and practice we've prepared for you. First, let's look at the best ways to practice.

How to practice?

At this point, you've mastered the theoretical underpinnings of the Algorithm Design Canvas. This is a great achievement and you're already ahead of the majority of applicants.

The next fundamental question is how to most efficiently take what you just learned, and start applying it in practice. Where do you find practice problems, and how do you go about solving them as you're getting ready for your interview?

Read on.

Practice with HiredInTech

In the next sections you will find practice tasks ordered by topic with some learning materials for each topic. You can submit your code solutions and get instant feedback from our grader. Each task also has a solution that you can look at in case you feel stuck or need a hint.

Use online judges

Online judges such as TopCoder are an indispensable tool in your preparation arsenal. You can read as many books as you want or skim through the solutions to dozens of problems, but there's nothing like practicing getting a problem 100% correct.

Online judges automatically grade your solution (and support a large set of programming languages) with some pretty tough tests. They also act as problem banks, additional learning resources, a supportive community, etc.

Here are a few more sites where you can practice solving algorithmic tasks:

- [Firecode.io](#)
- [Codechef](#)
- [Codeforces](#)

Always follow the Canvas

Many people feel tempted to skip certain areas of the Canvas during practice. Sometimes they do this because they think they're saving time, sometimes because they are overconfident and think the problem is too easy.

This is not a good idea. The Canvas is designed to build up all aspects of your interviewing chops, as you never know what your interviewer will want to focus on. By systematically thinking about all areas during practice, when the rubber hits the road you'll be a lot more confident and, ultimately, more successful.

Forget about your IDE

As we discussed in the Code section, writing code in your IDE is fundamentally different from coding outside of it. The only way to truly appreciate the difference and to overcome the related issues is to forget about your IDE.

That's why the "Code" section of the Canvas exists - just write your code there. Alternatively, if you're practicing using TopCoder (highly recommended), just use the "Code" box of the UI.

Read through your code

We all know people whose natural style is to write some code and then immediately try to run it and see if it works on some set of test cases. They spend about 20% writing the code, and then maybe 80% debugging it. While this may work well for them in real life, it simply does not work at interviews. You don't have the luxury of a debugger, nor the luxury of something automatically executing your code.

Instead, what you should do is to carefully think about the code you write, and then once you've written it, to spend some time reviewing it. If you're using TopCoder, make sure

you are 100% confident that your code is going to compile AND run correctly on the tests before actually hitting "Compile" and "Run Tests". This practice of carefully inspecting your code is going to be priceless at interviews.

Summary

In this section, you learned several important tricks for increasing the efficiency and effectiveness of your practice. Remember to:

- Sign up for an automated online judge like TopCoder.
- Always follow the Canvas, even on simpler problems.
- Not use your IDE so that you can train your interview coding skills.
- Read and review your code before compiling and running the automated tests on TopCoder.

What's next?

With these tips in mind let's turn to some algorithmic theory and practice. In the next sections of this course we cover the most important topics and suggest practice tasks that you can solve. Let's start practicing!

Also, in this section there is one last lesson suggesting one more way to practice, which we call the **"blitz rounds"**. You can take a look at these once you feel confident in your task solving skills.

Computational Complexity

Overview

As mentioned earlier in the course, we will devote a section to computational complexity because this is a key thing to consider when designing almost any software solution - from a small program to big software systems. Let's begin with some informal explanations of what it is and why it is so important for your tech interviews.

Imagine that you are given a task to generate all palindromes using a set of N Latin letters. One brute force solution that comes to mind is to generate all permutations of the N letters and to check for each one of them if it's a palindrome. Some people could say that the task is easy because this solution is easy to come up with. However, to generate all permutations of the N letters you would need to generate N! words. For even small values of N this becomes a solution, which will not finish in a reasonable amount of time. Because of that we will need to design a better approach, which can compute the answer much faster.

At tech interviews you will have to compute and explain the time and memory complexity of your solutions quite often. This is something that will show your interviewer that you can evaluate if a solution is feasible given the input size. In real life this is also a very useful skill. Actually, in our opinion, it's a must-have skill for good software engineers.

Two resources that are limited for our programs are time and memory. Of course there are others but these two are usually the main constraints to our solutions. That is why in this section we will focus on time and memory complexity. We will talk about why they are things that you should definitely be able to understand and measure at a tech interview. Of course, we will also provide some explanations and examples to help you understand them better.

Time complexity

Let's look at some examples in order to get a better understanding of time complexity of an algorithm. Later we will also look at memory complexity as this is another limited resource that we have to deal with.

First of all, time complexity will be measured in terms of the input size. As you saw in the example above, N was the number of latin letters to use for building palindromes. We said that all permutations of these N letters are N!. So to generate them we would perform a number of steps that is proportional to N!. Of course, it is also important how many steps it takes to generate each separate permutation. It is possible to do that with a constant amount of steps. We will not go into details how this can be done but trust us.

This means that the number of steps to generate each next permutation does not depend on the size of the input. For each generated permutation we would need to check if it is a palindrome. One way to do it is to compare the first and last letters, then the second and the last but one and so on. This will require a number of steps that is proportional to the number of letters - N. So, for each permutation we will perform that many steps.

With all that in mind we will have to perform a number of steps that is proportional to N * N! in order to execute our brute force solution. This number will be multiplied by some constant but usually when this constant is not too high we don't take it into account. Now that we have quickly analysed the number of steps required, we can clearly see that the number of steps will grow very quickly with increasing values of N. If your interviewer tells you that N can be as high as 100, then there is no use in even considering such a solution. And being able to describe to the interviewer why such a solution is not feasible is also a useful skill.

Let's look at another example. Imagine a block of code, which sorts an array of integers:

```
// An array `arr` with `len` integers in it is sorted.
for (int i = 0; i < len - 1; i++) {
  for (int j = i + 1; j < len; j++) {
    if (arr[i] > arr[j]) {
      int tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
    }
  }
}
```

This algorithm has two nested loops. The outer one goes through the numbers from left to right and finds the number that must be in each position. For the number at position 0 it finds the minimum of all numbers. Then, for the number at position 1 it finds the minimum of all remaining numbers and so on. The question we will answer here is: what is the time complexity of this algorithm?

The outer loop will perform N-1 iterations where N is the number of the numbers to sort. This is our parameter indicating the size of the input. For each iteration of the outer loop

the inner loop will perform a different number of steps. In the first interation it will perform N-1 steps, next it will perform N-2 steps and so on.

Inside the loops there is a comparison and in some cases there will be three operations used for swapping two values. These operations inside the loops take constant time regardless of N. That is why we will be more interested in computing the total number of interations that the two loops will perform. To compute that we just need to sum up the number of iterations of the inner loop: (N-1) + (N-2) + ... + 2 + 1 = N * (N-1) / 2. This is a number proportional to N^2 because if we expand it we will get N^2 / 2 - N / 2. We are always interested in the term with the highest degree and here this is N^2. It is multiplied by a constant - 1/2 - but this does not change the fact that the total number of steps will be proportional to N^2 and as N grows linearly our algorithm's speed will slow down quadratically.

Another important thing to mention is that you are usually interested in finding out the slowest part of your algorithm. Maybe for some task you have a solution that does some preprocessing first taking roughly N*M steps, with N and M being some values identifying the size of your input. But then if your core algorithm performs N*M^2 steps to run, then you can say that this is your actual time complexity becuase it's of higher order than N*M.

There are several formal definitions for how we define computational complexity and they can be used depending on the case. For most tech interviews and real-time examples you will need to use the so called big-O notation. Below we have included links to a few useful resources that will tell you more about big-O and other notations using more or less formal language.

Resources

- [TopCoder's 2-part tutorial](#) on computational complexity is a good place to start, it also covers several notations. From the first part the whole article is useful. You can perhaps skip the section "Finally, formal definitions" if you wish. From the second part, everything up to the section "The substitution method" is worth covering for the purpose of interviews. The rest is more or less optional.
- [This tutorial from Cprogramming.com](#) describes the matter in an easy to understand way
- And [this article](#) uses a lot of examples to give you a better intuition
- From MIT we have [a great lecture handout](#), which has some formal definitions but also has some good examples

Memory complexity

To measure memory complexity of a solution you can use a lot of the things that we already described in this section. The difference is that here you need to measure the maximum amount of memory that is used by your solution at one point in time. Why is that important? Mainly because every time you run a piece of software on a given machine you will only have a limited amount of memory. Go over this amount and you are in trouble. The first most likely effect will be that the OS will start to swap memory to hard disk, which will make the execution much slower and practically useless.

Hopefully, you agree that memory usage is as important as the running speed of an algorithm. You can measure it in similar ways to how you measure the time complexity. For example, let's look again at the example with the permutations of latin letters. First, to hold all the letters you will need memory proportional to N (the number of letters). Then, to generate all permutations you could use the same array and just generate subsequent permutations in it. Checking if a sequence of letters is a palindrome does not require additional memory. We could say that this brute force solution requires memory proportional to N. In terms of memory usage this solution is not bad then.

Let's look at another example. Consider a task in which you need to store information about a network of some kind. The network has nodes and edges that connect them. We will look at more tasks like this one in the section covering graphs but in this lesson let's just focus on how we could represent the network in memory.

One way would be to store a square matrix M with size N x N where N is the number of nodes in the network. In each cell M[i][j] we will have 0 or 1 indicating if there is an edge between nodes i and j. This representation of the network requires that we use memory proportional to N^2.

An alternative approach would be to store for each node a list of the nodes that it is connected to. In this case we can allocate memory that is proportional to the number of edges in the network. If nothing else is specified the number of edges could vary from 0 to N * (N-1). This means that in the worst case the memory used will also be proportional to N^2, like for the previous representation. But if for example we know that the network is quite sparse and doesn't have many edges, the memory used will be less that what we would consume with the first approach.

When designing a solution for a problem at a tech interview you will need to be able to compute the memory complexity, so that you can explain to the interviewer why a solution is good or bad given the input constraints.

How to use it in real life?

Hopefully by now you have built a good understanding of what computation complexity is about. You should be able to compute the time and memory complexity of most algorithmic solutions that you can face at a programming interview or at work.

Don't worry if things are not very clear now. The best thing to do in order to get more proficient in this is, as usually, to practice. Just take different algorithmic problems and their solutions and try to evaluate the time and memory complexity. Then look at some explanations about them to see if you are getting it right. It will take time but once you have this skill mastered, it will be invaluable to you.

There is one more aspect to computational complexity and how it helps us evaluate different software solutions. Let's say that you can determine the time complexity of a solution. You also know the expected inputs. How do you decide if the solution will run fast enough?

If the time complexity of a solution is $O(N\text{^}2)$ and N can be as big as 10,000, is that good or not?

First of all, it depends on the time constraints that you are under. Secondly, you need to consider the machine(s) where the solution will run. With time you will start to build better intuition about how quick a solution will really be given it's time complexity and the input size. This won't happen immediately, it takes practice, but for sure, you'll get better at this as you work through various examples.

However, knowing the time or memory complexity of different solutions to the same problem also gives you the ability to compare them and decide which one is better, or what trade-offs are required.

With memory it's probably somewhat easier to evaluate and still it depends. For example the programming language maybe not allow you to compute the exact memory used. If you implement stuff in C it will probably be easier to know exactly how much memory will be used. If you use languages offering higher level abstractions like C++ (with STL for example) or Ruby, Java, Python, you may need to experiment to see how much memory is consumed.

And still if you know the memory complexity, you will be able to have a pretty good expectation about the memory used. This is much better than nothing.

Now with this knowledge under the belt you can continue to the next sections, which cover specific algorithmic topics.

At the technical interviews

Now that you've mastered the theory behind time and memory complexity, let's talk about how to best apply what you've learned at your interview.

Computing and discussing the complexity of a solution will come up very often at tech interviews. You should be able to identify the time and memory complexity of a solution fairly fast. It should not be taking you a long time. The answer should come to you almost instantly. Keep in mind that no complicated math is expected to be needed here. Don't expect that someone will require you to use the Master Theorem, for example. Most likely, with very few exceptions, we are talking standard complexities here.

Don't worry if you don't feel confident enough right now. Training is the best way to become really smooth when it comes to computing the complexity of a solution. As you solve problems using the Algorithm Design Canvas, always make sure to fill in the box about complexity for each solution you design. Don't skip this vital step. Doing this will help you get better.

One more thing: practice shows that usually time complexity is somewhat more discussed. However, you must remember that very often there is a tradeoff between time and memory. A solution can be tweaked to use more memory and become faster and vice versa. Because of that, both complexities are important. You may even be in a situation where your solution can be pushed in any of the two directions. In such cases, it's wise to ask the interviewer which resource is more valuable. The answer will hopefully allow you to decide how to shape your solution.

Summary

In this section you learned:

- Why time and memory complexity are so important and where they fit in the coding interview.
- Given an algorithm, how to figure out its complexity.
- Given an algorithm's complexity, build up the intuition about whether that complexity is "sufficient": would it work fast enough and consume reasonable amounts of memory given the algorithm's constraints.

Dynamic Programming

Overview

Dynamic programming is a very widely used technique. You may have even been applying it without knowing it. Its applications vary from very simple ones to more complicated cases. Nevertheless, many people wonder if they even need that for the interviews. Our experience shows that it is likely that you would get a problem requiring you to design a solution using dynamic programming. Overall, it is useful knowledge for many software engineers. Depending on what you work on it can come very handy sometimes.

How to proceed with this section?

There are plenty of articles and books out there covering the topic very thoroughly. If you are new to dynamic programming our advice is to read these overview lessons and then go through some more in depth articles available online. We will offer useful links at the end of this lesson. Once you feel confident with your knowledge about dynamic programming you should continue to the practice tasks.

About dynamic programming

So, what is dynamic programming? In short, it's a method, which allows you to solve a problem by breaking it down into smaller sub-problems. With dynamic programming you will try to identify a way to combine the answers for smaller versions of a problem to compute the answer for a bigger version. If you find such a recurrent dependency you would be able to start with some very trivial cases, which you know the answer to, and subsequently compute the answers of bigger and bigger versions of the problem until you reach the version that you were initially interested in.

The above method works well if you store the answers for smaller versions of the problem and reuse the computed values when computing for bigger versions of the problem. This technique is called "memoization".

Resources

- [A great tutorial on dynamic programming from TopCoder](#)

- If you are still curious and want to go in more depth, [the Wikipedia article on dynamic programming](#) can give you that

Trivial example

Let's start with a very simple example to illustrate the above words. Think about the [Fibonacci numbers](#). How do we compute the 100th such number: by taking the sum of the 98th and 99th. This can be considered a very simple example of breaking down a task into smaller tasks. In order to get the answer for the problem with size N we need to solve for the problems with sizes N-1 and N-2. If we go down like that we will reach the trivial problems with sizes 1 and 2, which we know the answers for. They come from the very definition of Fibonacci numbers: F(1) = 1 and F(2) = 1. In some cases it may be stated that the first two values are 0 and 1, which is not so important for our example.

This means that if we compute the answers for problems with increasing sizes we will eventually get to the answer for F(100).

Let's see how memoization works for us here. The definition for the Fibonacchi numbers looks like this: F(n) = F(n-1) + F(n-2). If we write a program, which makes recursive calls to get the answer for smaller tasks we may get in trouble very quickly. This is because each new call will spawn two new calls. This means that at the top most level of the recursion there will be one call, at level two there will be two calls, at level 3 - there will be 4 calls and so on. At every next level there will be twice as many calls as the level above it. Of course, at some level the recursion will be reaching one of the trivial cases (F(1) and F(2)) but for high enough indexes the levels at which we reach the trivial versions of the problem will be very far away from the initial call.

Here is pseudocode for how this solution would look like:

```
def fibonacci(N)
  if N <= 2 return 1;
  return fibonacci(N-1) + fibonacci(N-2)
end
```

As a small homework task you can try to compute the number of recursive function calls that we happen for different values of N. This will help you observe the exponential growth of these calls as N increases linearly.

So, this implementation of dynamic programming won't be efficient enough if we want to find Fibonacci numbers with too high indexes, like 100 for example. Why is that? The main reason is that the same Fibonacci number will get computed over and over again.

This is because each Fibonacci number, except the first one, is used to compute two other numbers. This is a very common situation in different problems requiring dynamic programming - the same sub-problem is a building block for computing more than one bigger sub-problem. Instead of re-computing the value each time we need it we could store it once and reuse it from memory when we need it. This is what memoization is about.

For Fibonacci numbers let's start computing the answers from the bottom up. Initially we store F(1) and F(2) and them continue with computing for F(3), F(4), F(5) and so on. Pseudocode will look like that:

```
F(1) = 1
F(2) = 1

for i = 3 to N
  F(i) = F(i-1) + F(i-2)
end
```

This way we compute the answer to each sub-problem only once. Of course, we could stick to our initial recursive solution starting from the top and going down. But we need to make sure that we cache the computed values instead of computing them over and over again. Here is some code:

```
F(1) = 1
F(2) = 1

def fibonacci(N)
  if F(N) is not stored
    F(N) = fibonacci(N-1) + fibonacci(N-2)
  end

  return F(N)
end
```

This solution stores the computed values in memory and reuses them if they were already computed. If a value has not yet been computed we make recursive calls to get the answer for the smaller sub-problems. However, since no value is computed more than once the complexity is now linear, instead of the exponential growth of spawned tasks that we observed with the recursive solution not using memoization.

Multiple dimensions

So far we illustrated the basic idea of dynamic programming - that the answer for a given problem can be computed using the answers for smaller sub-problems. We also showed that there is no need to compute the answer for the same sub-problem more than once and that we can store the computed answer and reuse it.

The Fibonacci problem was defined by only one dimension - the index of the number. However, dynamic programming can be applied to problems defined by more than one parameter. One very popular problem that can be solved using dynamic programming is the "0-1 Knapsack problem".

In this problem we have a set of N items each with a given weight and value ($W[i]$ and $V[i]$). We are given a knapsack with a maximum capacity of C. Our goal is to select, which items to put into the knapsack, so that their total weight is not more than C and their total value is as large as possible.

To solve this task we need to define what a problem is and how one problem's answer can be computed using smaller sub-problems' answers.

Let's assume we order the items in some way, so that we have item 1, item 2, and so on. We can describe a problem with two parameters - the maximum capacity allowed and the highest index of an item to consider. These values for the final problem to solve are N for the highest index and C for the capacity. Now the question is: how do we break it down into smaller sub-problems?

For a given version of the problem described by some values N and C we have two options:

1. Put item with index N into the knapsack and reduce the capacity to $C - W[N]$
2. Don't put the item with index N into the knapsack and remain with capacity C

In the first case we will be interested in the answer for the problem with parameters $N-1$, $C - W[N]$. This answer will give us the maximum obtainable value when considering the first $N-1$ items and limiting the total weight capacity to $C - W[N]$. To this answer we can add the value of item N since we decided to put it in the knapsack.

In the second case we will be interested in the answer for the problem with parameters $N-1$, C. This answer will give us the maximum obtainable value when considering the first $N-1$ items and limiting the total weight capacity to C. To this answer we won't add any value because we have decided no to put item N into the knapsack.

Since we need to make a choice between adding and not adding item N into the knapsack, we should take the maximum between these two possible answers:

F(N, C) = max(F(N-1, C-W[N]) + V[N], F(N-1, C))

Here is our recursive dependency. Of course, we need to think about the base cases now.

The base values will be the all F(1, *) and F(*, 0). The first are these for which we are only allowed to take the first item with any capacity in the range [0, C]. The second group are the sub-problems in which the maximum weight capacity is 0.

For F(1, *) we can say that the answer is 0 for these sub-problems in which the allowed capacity is less than W[1] and it is equal to V[1] for the rest of the cases in which the maximum capacity can hold item 1's weight.

For all F(*, 0) we know that the answer is 0 because there is no free capacity to hold any elements.

The answer that we are looking for is F(N, C).

For the Fibonacci numbers we showed that the answer can either be computed by going "bottom-up" or "top-down". The first means that we start from the base cases and proceed to the final problem. In this task this would mean to first compute F(1, *), then F(2, *), F(3, *) and so on, until we reach F(N, *). For this approach note that for computing F(i, *) we only need the values from F(i-1, *). So, we don't need to store the earlier values all the time. This could help us save a lot of memory.

The second method, "top-down", would involve a recursive approach, in which we store and reuse the values once they are computed once. This is the memoization technique. Without it we would be computing the same value many times and this would make our solution quite inefficient.

Conclusions

This is a brief introduction to dynamic programming. We showed through examples what we mean by breaking down a problem into sub-problems. There are usually different ways to implement the computations. One would be "bottom-up" where we start from the base cases and compute the values until we reach the desired value. Another would be "top-down" in which we recursively compute the answers for smaller problems, on demand,

but try to store the computed valued in order not to compute them multiple times. This technique is usually called "memoization".

Sometimes, especially for "bottom-up" implementations it is possible to store only one part of the computed values at a time and free the memory for other parts once they have served their job in the computations.

Sorting and Search

Overview

Sorting is one of the fundamental topics. Nowadays, most programming languages have utilities, which allow us to sort items efficiently. So, in real life we rarely have to implement sorting. However, this topic is not just about pure sorting of items. Moreover, it is always helpful to have an idea about how these algorithms work and what to expect when we use them. This is why questions related to them can be expected at interviews.

In the practice tasks you will see that sometimes you would need to perform sorting in non-standard ways. Also, sometimes to solve a task one needs to make observations related to the sorted order of elements in a sequence. The examples could go on and on but let's talk a bit more about sorting.

Popular sorting algorithms

You are probably already familiar with different techniques for sorting elements. It is not impossible that sometimes at an interview you would just need to write code, which orders a set of elements by some value. For such cases it is always good to be able to quickly implement something from scratch. Probably the simplest and easiest to write sorting algorithms are "selection sort" and "bubble sort". Both algorithms have time complexity $O(N^2)$, where N is the number of elements to sort. We will not go in details about how these algorithms operate because there are plenty of excellent resources out there covering this. The Wikipedia article on sorting linked at the bottom of this lesson is a good start. It tells you about some other interesting algorithms as well.

There are some other more efficient algorithms in terms of their speed. Quick sort, merge sort and heap sort are probably the most famous ones.

One interesting aspect of these algorithms is their running time for different inputs. For example it's worth mentioning that *selection sort* performs $O(N^2)$ operations regardless of the input. However, *bubble sort* could be very quick if given input that is already sorted and it will take $O(N^2)$ if the output is sorted in reverse for example. That may be an important feature in some situations. It's worth considering such differences between seemingly similar algorithms. Sometimes interviewers may be curious to test your knowledge in this regard.

Another such example is *quick sort*, which depending on the implementation could perform O(N^2) operations in the worst case. At the same time *heap sort* and *merge sort* always have time complexity O(N*logN) regardless of the input.

Resources

- Wikipedia [Sorting algorithms article](#)
- TopCoder [tutorial on sorting](#)

Binary search

One very important benefit of having sorted elements is the ability to quickly search through them. If you have a sequence of sorted elements you can use binary search to find if it includes a given element or to find the proper place of a new element in the sorted sequence with time complexity O(logN). For big enough sequences this is much better than the linear approach of searching through the whole sequence until you find the answer.

Resources

- TopCoder [tutorial on binary search](#)

Mathematics

Overview

Mathematics is a very wide topic and there are many fields in it that could be covered. There are also plenty of different tasks that one could face that require math in one form or another. In this section we will try to outline some very popular ideas that may be helpful at tech interviews. Let's focus on topics and practice tasks that are likely to be used at the interviews. We will only briefly mention different problems and techniques, so that you can get a better sense of what's important.

Representing numbers in different bases

This is fundamental to computer science as a whole. It's important for everyone in the industry to be familiar with how a number is represented in different bases and how to convert between these representations. The most often used bases are probably 2, 8, 10, 16. Base 10 is what we mainly use nowadays and the others happen to be degrees of 2 and have application in computer science.

Integer number factorization

Breaking down an integer into the product of smaller integers is something that is often useful for solving various algorithmic problems. It usually boils down to finding the prime divisors of an integer as they can be the basis for finding all divisors of a number, for example.

Prime numbers

Prime numbers play a major role in mathematics. In computer science they are probably most widely used in cryptography. For your tech interviews and everyday work as a computer scientist it's good to know how to efficiently find all prime divisors of an integer. Also, checking if a number is prime or finding the Nth prime number are fundamental tasks with multiple applications.

Greatest common divisor (GCD)

The GCD of two integers gives you the greatest integer that divides both integers. It can help you when working with fractions. You will see that in the practice tasks. Or, for example, it could be useful when solving some sort of a fitting problem where smaller

objects need to fit into bigger ones and we don't know how many times. There is a efficient algorithm for computing GCD and it will be covered in more detail in the practice tasks.

Geometry

This is a huge topic on its own. Such tasks should be given less often at regular tech interviews. Well, if you happen to apply for a more specialised position, say in computer graphics, you may need to prepare more extensively here. However, some popular tasks that come to mind are finding the area of a triangle or a shape with more sides (convex vs non-convex), finding intersections of rectangles or the total area covered by overlapping rectangles, computing the convex hull of a set of points and many more. Take a look at the link to the TopCoder tutorial below for some additional examples.

Resources

- TopCoder [tutorial on mathematics](#)
- TopCoder [tutorial on prime numbers, factorization and euler function](#)

String Algorithms

Overview

Engineers need to work with strings all the time. Imagine how much information in the form of text is processed at companies like Google, Facebook, and Microsoft. Such problems are popular at interviews and that's why we have a separate section for them.

There are some often seen string problems, which people face. Here we will look at some of them without going into too many details about the different algorithms. As for the other algorithmic topics this lesson only serves as a quick overview that will hopefully help you in finding more detailed resources.

Popular string tasks

There are plenty of tasks involving strings that may come to mind. It's hard to cover them all and this is not practical at all. What is better is to practice working with strings by solving as many string problems as possible. We will try to add more practice tasks in this section with time. Meanwhile, you can also just search the Internet and you are guaranteed to find tons of shared tech interview tasks involving strings.

In this section we've covered some more interesting topics that can be useful for solving a bigger class of problems.

String hashing

This is a technique often used when you need to use strings as keys pointing to a set of corresponding values. For example, if you are writing code, which analyses text and needs to record some data for each word found in the text? A good approach would be to compute a hash value for each found word. Then, when a new word is processed its hash can be used to find the already computed values for it, if such already exist. In many programming languages there are already implemented data structures giving you this functionality by using a hash table of some sort.

A good hashing function usually is one, which computes a polynomial from the symbols of the input string modulo a prime number. For example, if you have a string with symbols $S = S_1 S_2 \dots S_n$, the hash function would compute $H(S) = (S_1 * A\char94(n-1) + S_2 * A\char94(n-2) + \dots + S_{n-1} * A + S_n) \bmod P$. Some presentation of the symbols as numbers needs to be used and a value for A is to be chosen. The prime number P gives you the size of the resulting set of values.

It's important to keep in mind that collisions are possible. Each time when there is a match of hash values you need to check if the string hashed really is the one that we are using as input.

One very useful property of this kind of hash function is that if you compute it for some string $S = S_1 S_2 \dots S_n$ you can compute in constant time the hash function value for a string $S' = S_2 S_3 \dots S_n S_{n+1}$. This is the case because $H(S') = (H(S) - S_1 * A\char94(n-1)) * A + S_{n+1}$ if we ignore the fact that there is also a modulo operation. However, this is easily handled. We will leave you to figure this out as a small homework assignment.

It turns out that this "sliding window" property of the hash function can be used in some pattern matching algorithms. We will look into this topic next.

Pattern matching

A very popular string problem is to find one, some or all occurrences of one string within another. Usually the string to search for is called a *pattern*. One example task would be given a text T (with length Lt) and a pattern P (with length Lp) to find all the starting positions of substrings in T that are equal to P.

Once given such a task probably the most straight forward solution would be to try matching the string P against all substrings in T with the length of P. For example, if given $T = mississippi$ and $P = issi$ you would first check against the first substring miss and this would produce a mismatch on the very first symbol. However, the next substring issi will match P. Then you continue with the next substring - ssis - a mismatch again. There will be one more match with issi starting at symbol number 5 and all the rest of the substrings will not match.

This approach is easy to implement but it's worst-case running time can be terrible - O(Lt * Lp). Sometimes Lt and Lp could be quite big rendering this approach not suitable. Still, you need to evaluate the task at hand and the possible inputs. For random strings it may be the case that in most of the cases there is a mismatch between P and the substrings of T in the first 1 or 2 symbols. If the matches are very few, this would mean that your running time will be much better than the expected worst-case. However, if your solution needs

to solve an input in which P is contained in T many times you may need to think of a better algorithm. A very extreme and simple example is one where T contains the same latin letter many times and P is constructed of this letter only.

More efficient pattern matching

It was mentioned in a previous lesson that a hashing function could be used in string pattern matching due to its "sliding window" property. This is the idea of the *Rabin-Karp algorithm*. In the resources below we've included links to Wikipedia and TopCoder articles about it. It is relatively easy to implement and offers an improvement over the previous algorithm but it's worst-case running time is still O(Lt * Lp).

That is why it's worth considering some more advanced algorithms. A very nice algorithm worth learning and implementing is the *Knuth-Morris-Pratt* algorithm. The algorithm first constructs a table with some data based on the pattern with time complexity O(Lp). Then for the search of the pattern within the text it uses the data from the table and this takes O(Lt). These are worst-case running times. We encourage you to learn this algorithm and you will also have the chance to apply it in one of the practice problems in this section. The links about it from Wikipedia and Topcoder, found below, can be useful.

Some other techniques worth mentioning are the *Aho–Corasick algorithm*, *suffix trees* and *suffix arrays*.

Resources

- TopCoder on [Rabin-Karp and Knuth-Morris-Pratt Algorithms](#)
- Wikipedia on [Rabin-Karp](#)
- Wikipedia on [Knuth-Morris-Pratt](#)
- Wikipedia on [Aho–Corasick algorithm](#)
- Wikipedia on [suffix arrays](#)
- Wikipedia on [suffix trees](#)

Graph Theory

Overview

In this section we will look into graph theory and mainly things that you need to know in order to recognise and solve such problems, both at interviews and at work.

As with other sections in this course it won't be possible to go into details about various algorithms and techniques. One reason is that to do that we would need to write a book, not a single lesson. Our focus are the practice tasks. Another reason is that nowadays there are plenty of free resources on this and other topics that one can use to learn. We provide some links in the lessons and there are many more.

Unlike some other categories of algorithms it may not be immediately obvious that a task requires some sort of graph representation. This may be especially true for someone with little experience in solving such problems. Let's first look at some entities that can be represented with graphs.

What things do graphs represent?

Some very natural examples are the transportation networks of cities or countries. Even the flights performed in the whole world could form a graph where airports are the nodes and flights are the connecting edges.

Another very popular situation is when there is a grid of some sort and the cells in it can be represented as nodes while the edges exist between cells that share a side. As an example take a spreadsheet application in which one needs to compute values in cells that are based on the values in other cells.

Various relationships between people could be described as graphs, too. Think about social networks and how people in them can be nodes in a graph and the relationships between them as edges.

Representations in code

In order to code graph algorithms one needs to know how to represent them in memory. There are some more widely used representations and each one has advantages and disadvantages compared to the others. Hence, it's good to know which one to use for a given problem. This may be a crucial decision for an interview problem.

One popular representation is a matrix M with size n x n where n is the number of nodes in the graph. The cell M[i][j] will contain a value indicating whether there is an edge going from node i to node j. This representation will work for graphs with or without weights on their edges. It will also work for graphs with or without directed edges. Of course, for undirected graphs the matrix will be symmetric because M[i][j] = M[j][i] for all i, j.

The matrix representation takes O(n^2) memory, which may not be optimal in some cases but it could turn out to be very fast to use for certain algorithms. Also, for dense graphs (with many edges) this representation may happen to have similar memory usage as other representations.

Another representation is one, which has a list of neighbours for each node. This means that for each node Vi in a graph G we will have to store the identifiers of the nodes, which are connected to Vi by an edge. With this representation you could save a lot of memory compared to the matrix approach for sparse graphs in which the consumed memory will be O(m) where m is the number of edges in the graph.

Finally, one more representation that could be useful in some cases is a list of edges. This means that you would just store a list of all edges as pairs of nodes and the attributes associated with each edge. This would again require memory that is O(m) but could be more suitable for a given set of problems.

Now let's look at some popular classes of graph algorithms.

Traversals

Graph traversals are a very popular set of problems in which some entity is represented by a graph and the nodes in this graph need to be traversed in some order. Sometimes one needs to find if there is a way to get from node Vi to another node Vj following the existing edges in the graph. Two popular algorithms for this are Depth-first search (DFS) and Breadth-first search (BFS). After learning and practicing the different representations of graphs these two algorithms are a must for people learning to code graph algorithms.

In summary, DFS is about starting from one node and recursively continuing to the nodes that are connected to it by edges and have not yet been visited by the traversal.

The idea of BFS is different, starting from a node it first visits all its neighbours and then does the same for all of them in some order.

On one hand these algorithms both traverse the edges of a graph that are reachable from the starting node. On the other hand, due to the differences in the algorithms approaches, they have some major differences.

As mentioned above DFS takes a somewhat recursive approach, a simple pseudocode will look like that:

```
def dfs(node)
  mark node as visited

  for next_node in neighbours(node)
    if not visited(next_node)
    dfs(next_node)
    end
end
end
```

If implementing it recursively you need to be aware of the stack depth. If, for example, the graph is like a chain of nodes consecutively connected by edges and the number of nodes is high enough your program may crash due to consuming too many stack frames. Of course, it is possible to avoid this problem by implementing the recursive approach iteratively.

The time complexity of DFS is $O(m)$ where m is the number of edges.

With BFS the solution is usually implemented using a queue and is iterative. There is one very useful effect of running BFS - when starting from a given node, it will get to all reachable nodes with the minimum possible number of hops (edges). This is useful in problems in which one needs to find the minimum path in terms of edges between two or more nodes in a graph.

This is pseudocode for BFS:

```
def bfs(node)
  queue.add(node)
  mark node as visited
  distance[node] = 0
```

```
  while not queue.empty
   top_node = queue.pop
   for next_node in neighbours(top_node)
     if not visited(next_node)
     queue.add(next_node)
       mark next_node as visited
     distance[node] = distance[top_node] + 1
     end
   end
end
end
```

The time complexity of BFS is $O(m)$ where m is the number of edges in the graph.

We encourage you to read more about these algorithms and you will also have a chance to apply them in the practice tasks in this section.

Practice Ideas

Blitz rounds

Topical Practice

Most people spend their time doing topical practice. They pick a topic (say "string algorithms" or "graph problems"), and start going through problems related to that topic.

Up to a point, this is quite useful. It can also be particularly good if you know there is some topic where you're markedly weaker.

But at interviews, you simply don't know if a problem is a graph problem, a data structures problem, or a dynamic programming problem. You will need to train your intuition to quickly discern between the large pool of potential ideas and deduce the topic yourself. How?

Introducing: Blitz Rounds

We do this by what we call "Blitz" rounds at HiredInTech. A "Blitz" round is a focused block of practice (say, 2.5 hours), where you work on a pool of problems you know nothing about. You set up a fixed amount of time per problem (say, 30 minutes), and try to go through all areas of the Canvas in that time. Then, you move on to the next one. After finishing 3 or 5 such problems, you can take a break, crack open a beer or pour yourself some wine, and give yourself a pat on the back.

If you're using TopCoder, taking 250 or 500 problems from Division 2 and pooling them together is a great way to construct a Blitz session. If you are working through a book, it's a bit more difficult to be "surprised" by what topic each problem is from, because books tend to organize problems by topic. This is another reason we strongly recommend switching your practice to HiredInTech's practice tasks or online judges at some point.

4 Sample Blitz Rounds

To get you started, we've assembled for you two blitz rounds with five TopCoder problems each. You can use the Arena for submitting your solutions as you did for previous sections.

In addition to that there are two more optional blitz rounds with problems from the book ["Cracking the Coding Interview: Fifth Edition"](). For them you can use a printed copy of the [Canvas](), a sheet of paper or a whiteboard. Again give yourself no more than 30 minutes per question. Code for the solved problems of the book is found [here]()

TopCoder Blitz Round 1

- Problem 1 - [Statement]() - [Solution]()
- Problem 2 - [Statement]() - [Solution]()
- Problem 3 - [Statement]() - [Solution]()
- Problem 4 - [Statement]() - [Solution]()
- Problem 5 - [Statement]() - [Solution]()

TopCoder Blitz Round 2

- Problem 1 - [Statement]() - [Solution]()
- Problem 2 - [Statement]() - [Solution]()
- Problem 3 - [Statement]() - [Solution]()
- Problem 4 - [Statement]() - [Solution]()
- Problem 5 - [Statement]() - [Solution]()

"Cracking the Coding Interview" Blitz Round 1

Questions: 4.3, 5.2, 17.8, 17.12, 18.7

"Cracking the Coding Interview" Blitz Round 2

Questions: 4.7, 7.6, 1.6, 17.6, 18.12

Summary

In this section, we focused on the importance of a second kind of practice that we call "Blitz rounds". Blitz rounds involve picking a set of 3 to 5 problems, setting a time limit per problem (20 to 30 minutes), and sequentially filling all areas of the Canvas for each problem. Combined with what we call "Topical" practice, blitz rounds increase your confidence, teach you to manage your time better, and make you faster and more fluent when working through problems.

What's next?

If you've reached this far, covered all the lessons carefully and did the recommended problems, you should be in a pretty good shape and deserve congratulations!