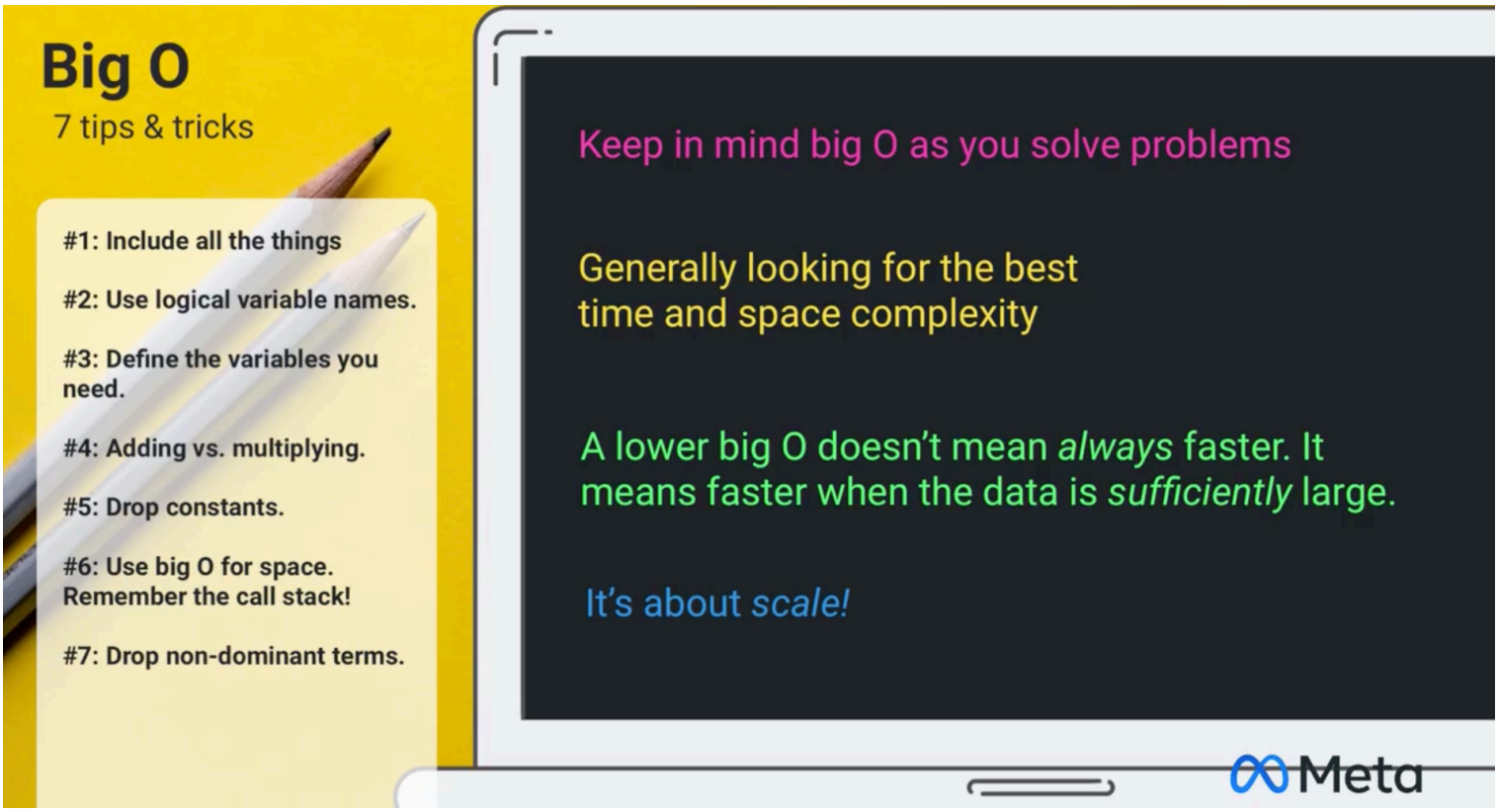
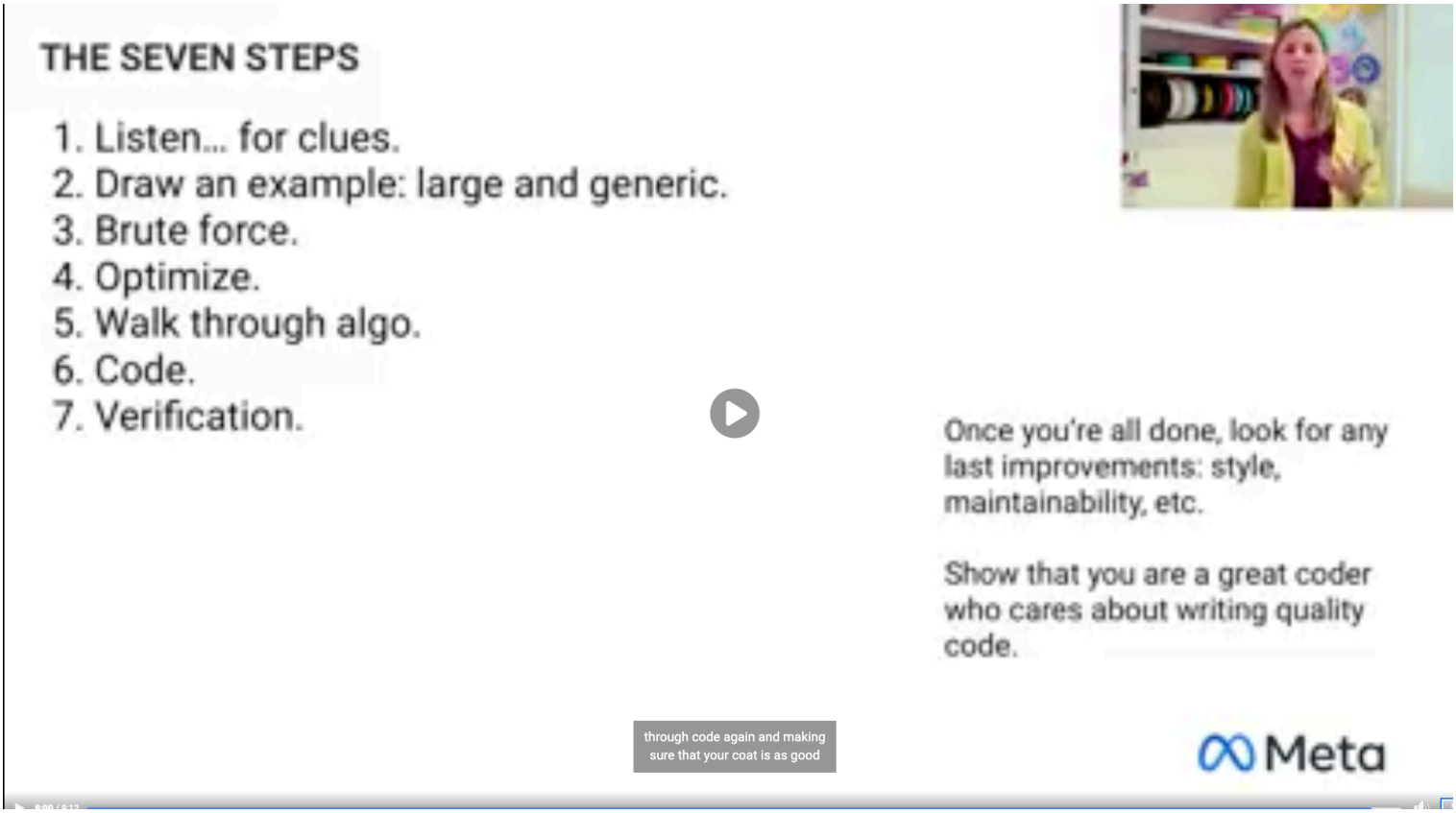
DSA


Big O
7 tips & tricks
#1: Include all the things
#2: Use logical variable names.
#3: Define the variables you need.
#4: Adding vs. multiplying.
#5: Drop constants.
#6: Use big O for space. Remember the call stack!
#7: Drop non-dominant terms.
Keep in mind big O as you solve problems
Generally looking for the best time and space complexity
A lower big O doesn't mean *always* faster. It means faster when the data is *sufficiently* large.
It's about *scale!*
Meta


THE SEVEN STEPS
1. Listen... for clues.
2. Draw an example: large and generic.
3. Brute force.
4. Optimize.
5. Walk through algo.
6. Code.
7. Verification.
Once you're all done, look for any last improvements: style, maintainability, etc.
Show that you are a great coder who cares about writing quality code.
through code again and making sure that your coat is as good
Meta

OPTIMIZING WITH BUD

Try to remove/handle all three

Bottlenecks

Unnecessary work

Duplicated work

OPTIMIZING WITH SPACE AND TIME

Hash tables
Pre-computation
Tries

Consider: upfront work to save yourself time down the road.

For example:
sorting the data, creating a hash table, etc.

BEST CONCEIVABLE RUNTIME

"Best conceivable runtime" is about the *problem*, not a specific algorithm. And it has nothing to do with best case / worse case runtime.

Given the nature of the problem, what is the best runtime you could possibly imagine getting? What runtime could you clearly not beat?

B.C.R. is not necessarily achievable... it's just not beatable!

Therefore:

If O(some step) <= BCR, then it's a "freebie". It won't hurt your runtime!

If O(your algorithm) = BCR, then your runtime is optimal. (But consider optimizing space?)

BEST CONCEIVABLE RUNTIME

"Best conceivable runtime" is about the *problem*, not a specific algorithm. And it has nothing to do with best case / worse case runtime.

Given the nature of the problem, what is the best runtime you could possibly imagine getting? What runtime could you clearly not beat?

B.C.R. is not necessarily achievable... it's just not beatable!

What is the B.C.R.?

I don't know that I can achieve **O(N)**, but I know I can't beat it!

So, **verify(candidate)** is a freebie"

You have a list of people, and some of them know each other. "knows" is one-way; I might know you, but you may not know me.

The only way of knowing if A knows B is by calling ***knows(A, B)*** [provided].

Celebrity = a person who knows no one, but everyone knows them.

Find the celebrity, if any.

OPTIMIZING WITH DIY →Do it yourself

Come up with a good example (large & generic), then figure out the output.

"Reverse engineer" your intuitive thought process.

Pay attention to any short circuiting or tricks you use.

Given two strings, S and B, find all permutations of S within B.

S = abbc
B = babcabcbebbacbabab

WHAT TO EXPECT WHILE CODING

Perfect candidates are awesome, but rare.

Interviewer just wants to make sure you're a good coder.

Pick the language you're most comfortable with.

Talk out loud. Show your interviewer your thought process.

Work towards writing great code. This doesn't mean you have to have things perfect immediately.

Write well-organized, well-written code. Find any bugs, and fix them.

Meta

SIGNAL, NOT PERFECTION

What makes a good code?
- => Readability
- => Correctness
- => Performance
- => Maintainability
- … and so on!

Wait until you're ready to code!

"I understand the basic gist of what I'm going to write" is not enough.

If you get confused while coding, don't power through. Go back to your example / your algorithm and figure out what you need to do.

Some candidates rush into code, when they don't full understand what they're going to do. This causes a lot of mistakes.

Make sure you know exactly what your steps are.

start coding. The other piece

Meta

SIGNAL, NOT PERFECTION

What makes a good code?
- => Readability
- => Correctness
- => Performance
- => Maintainability
- … and so on!

Wait until you're ready to code!
& Wait until your interviewer is ready.

You should know things like:
- Your input/output data types
- Index math
- Helper functions
- Data structures and when they change

can start writing great code in an interview.

CODING: TIPS AND TRICKS

Don't just dive in. Make sure you know the little details. Wait until you and your interviewer are ready.
-> algorithm details, data structures, indices, etc'

Demonstrate that you're a great coder:
-> correctness
-> readability
-> maintainability
-> performance

Use the neat features of your programming language!
-> Ask if you're not sure
-> Can often "make up" reasonable functions (or assume at first and implement later)

Use good style

Think about error cases, boundary checks, etc.

Sometimes it can be time consuming to write a bunch of input validation. If so, at least discuss them; this is the most important part.

Meta

WRITING STRUCTURED CODE

Structuring your code makes it more readable, maintainable, etc.
AND makes your life easier

Modularize code top down, filling in the next most interesting piece of code

Ignore things that don't show good signal (but check with your interviewer first)
—> Almost like you're pretending you have functions built in!

Focus on what shows signal

Talk to your interviewer. You don't need to guess at their expectations.

also to make your life as a candidate much much easier.

Meta

VERIFICATION

Your original example is generally *not* a good test case. Good examples are large, and large test cases take a very long time to run through.

Many candidates use their original example (the one they used to come up with the algorithm) as a test case. This isn't a good idea because:

1. This example is too large to test efficiently with.

and

2. You might have not planned for a scenario in your algorithm *due* to this test case.

way. Here's what I recommend instead. First, just do a

Meta

AN EFFICIENT VERIFICATION PROCESS

1. Conceptual walk through.

Walk through your code conceptually. Think through each line. What does it do? Is that the right thing?

Use your problem-solving skills!

AN EFFICIENT VERIFICATION PROCESS

1. Conceptual walk through. FAST!

2. Hot spots. Where are the high risk lines of code? FAST!
   => Math
   => Moving indices
   => Parameters when calling recursion
   => Base cases
   and so on!

3. Test cases:
   a. Small test cases. MEDIUM
   b. Edge cases MEDIUM
      i. Boring edge cases
      ii. Interesting edge cases.

C- Bid Test cases (only if you have time)

Boring edge cases:
- Null
- Empty string
- Single element

Interesting edge cases:
- All punctuation
- Sorted strings
- All duplicates

COMMON MISTAKES

Issue #1:
Verifying algorithm, not code

Issue #2:
Blindly testing, without thinking about what's happening.

Issue #3:
Quick, sloppy fixes.

Don't panic when you find a bug. Don't make the first fix you see.

Instead, think about why the bug is occurring, and how the code should work. Make sure your fix is the *right* one.

COMMUNICATION TIPS

Drive through the problem

Show your thought process.

Ask questions.

Propose an initial solution, but also think about its issues.

Talk while coding, if possible. Explain what you're doing.

COMMUNICATION TIPS

Use examples to explain your approach.

Be open about mistakes.

Admit if you've heard the question before.

Keep trying.

Interviewers want candidates who keep trying to make progress, even when the question is hard.

COMMUNICATION TIPS

Listen to your interviewer.

Make sure you capture the value of their hints.

If your interviewer ever offers the same hint more than one, that's a BIG hint that you missed something... and that it's really important.

COMMUNICATION TIPS

Listen to your interviewer.

Make sure you capture the value of their hints.

Follow your interviewer's path.

It's the journey, not the destination. So show me the journey!

Show me your thought process.

Data Structures:

what?:

A DS is a way of organizing data so that it can be used effectively/efficiently.

Why?:

They are essential ingredients in creating fast and powerful algorithms.

They help to manage and organize data.

They make code cleaner and easier to understand.

Abstract Data Types vs. Data Structures:

An Abstract Data Type (ADT) is an abstraction of a DS which provides only the interface to which a data structure must adhere to.

The interface does not give any specific details about how something should be implemented or in what programming language.

Ex:- (Basically, a mode of transportation from point A to B)

Abstraction (ADT)	Implementation (DS)
List	Dynamic Array Linked List
Queue	Linked List based Queue Array based Queue Stack based Queue

Map | Tree Map
Hash Map / Hash Table

** Computational Complexity Analysis

Big-O Notation

n - size of i/p.

Complexities ordered in from smallest to largest

Constant time : $O(1)$

logarithmic " : $O(\log(n))$

linear " : $O(n)$

Linearithmic " : $O(n \log(n))$

Quadric " : $O(n^2)$

Cubic " : $O(n^3)$

Exponential " : $O(b^n)$, $b > 1$

Factorial " : $O(n!)$

** Big O properties:

$O(n+c) = O(n)$

$O(cn) = O(n)$, $c > 0$

Big O Ex:

```
i = 0
while i < n  Do
```

```
        j = 0
    while j < 3*n Do
        j = j+1
    j = 0
        while j < 2*n Do
            j = j+1
    i = i+1
```

$\therefore f(n) = n*(3n+2n) = 5n^2$

$O(f(n)) = O(n^2)$

(2)

```
i = 0
while i < 3*n Do
    j := 10
    while j <= 50 Do
        j = j+1
    j = 0
    while j < n*n*n Do
        j = j+2
    i = i+1
```

$f(n) = 3n*(40 + n^3/2) = 3n/40 + 3n^4/2$

$O(f(n)) = O(n^4)$ (because n^4 is the dominant n)

- finding all subsets of a set $\Rightarrow O(2^n)$
  finding " permutation of a string $\Rightarrow O(n!)$

  Sorting using mergesort $\Rightarrow O(n \log(n))$

  Iterating over all the cells in a matrix of size n by $m \Rightarrow O(nm)$

Static and Dynamic Array:

Static Array: A static array is a fixed length container containing n elements indexable from the range $[0, n-1]$.

indexable → each slot/index in the array can be referenced with a number.

- When and where is Static Array used?

1. Storing and accessing sequential data.
2. Temporarily storing objects.
3. Used by IO routines as buffers.
4. Lookup tables and inverse lookup tables.
5. Can be used to return multiple values from a function
6. Used in dynamic programming to cache answers to subproblems.

~~Possibilities~~

Complexity:

	Static Array	Dynamic Array
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	N/A	O(n)
Appending	N/A	O(1)
Deletion	N/A	O(n)

Dynamic Array: can grow and shrink in size.

How can we implement Dynamic Array?

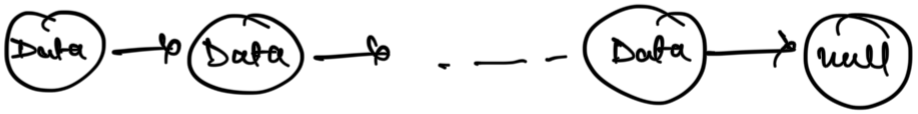
A) one way is to use a static array.

1. Create a static array with an initial capacity.
2. Add elements to the underlying static array, keeping track of the no. of elements.
3. If adding another element will exceed the capacity, then create a new static array with twice the capacity and copy the original elements into it.

Singly and Doubly Linked List

Singly and Doubly linked list

- A linked list is a sequential list of nodes that hold data which point to other nodes also containing data.

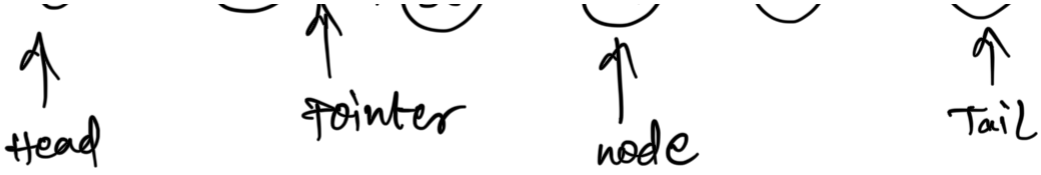


The last node is always a null node.

Where are linked lists used?

- Used in many list, Queue & Stack implementations
- Great for creating circular lists.
- Can easily model real world objects such as trains.
- Used in separate chaining, which is present certain Hashtable implementations to deal with hashing collisions.
- Often used in the implementation of adjacency lists for graphs.

Terminology:



Singly LL:-

Doubly LL:-

SLL & DLL Pros and Cons:

	Pros	Cons
SLL	Use less memory Simpler implement-ation	Cannot easily access previous elements
DLL	Can be traversed backwards	Takes 2x memory

** Implementation Details: