Example: Given a smaller string s and a bigger string b, design an algorithm to find all permutations of the shorter string within the longer one. Print the location of each permutation.

Think for a moment about how you'd solve this problem. Note permutations are rearrangements of the string, so the characters in s can appear in any order in b. They must be contiguous though (not split by other characters).

If you're like most candidates, you probably thought of something like: Generate all permutations of s and then look for each in b. Since there are S! permutations, this will take O(S! * B) time, where S is the length of s and B is the length of b.

This works, but it's an extraordinarily slow algorithm. It's actually *worse* than an exponential algorithm. If s has 14 characters, that's over 87 billion permutations. Add one more character into s and we have 15 times more permutations. Ouch!

Approached a different way, you could develop a decent algorithm fairly easily. Give yourself a big example, like this one:

```
s: abbc
b: cbabadcbbabbcbabaabccbabc
```

Where are the permutations of s within b? Don't worry about how you're doing it. Just find them. Even a 12 year old could do this!

(No, really, go find them. I'll wait!)

I've underlined below each permutation.

```
s: abbc
b: cbabadcbbabbcbabaabccbabc
     ____    ____      ____
       ____  ____      ____
             ____
```

Did you find these? How?

Few people—even those who earlier came up with the O(S! * B) algorithm—actually generate all the permutations of abbc to locate those permutations in b. Almost everyone takes one of two (very similar) approaches:

1. Walk through b and look at sliding windows of 4 characters (since s has length 4). Check if each window is a permutation of s.
2. Walk through b. Every time you see a character in s, check if the next four (the length of s) characters are a permutation of s.

Depending on the exact implementation of the "is this a permutation" part, you'll probably get a runtime of either O(B * S), O(B * S log S), or $O(B * S^2)$. None of these are the most optimal algorithm (there is an O(B) algorithm), but it's a lot better than what we had before.

Try this approach when you're solving questions. Use a nice, big example and intuitively—manually, that is—solve it for the specific example. Then, afterwards, think hard about how you solved it. Reverse engineer your own approach.

Be particularly aware of any "optimizations" you intuitively or automatically made. For example, when you were doing this problem, you might have just skipped right over the sliding window with "d" in it, since "d" isn't in abbc. That's an optimization your brain made, and it's something you should at least be aware of in your algorithm.

Optimize & Solve Technique #3: Simplify and Generalize

With Simplify and Generalize, we implement a multi-step approach. First, we simplify or tweak some constraint, such as the data type. Then, we solve this new simplified version of the problem. Finally, once we have an algorithm for the simplified problem, we try to adapt it for the more complex version.

> Example: A ransom note can be formed by cutting words out of a magazine to form a new sentence. How would you figure out if a ransom note (represented as a string) can be formed from a given magazine (string)?

To simplify the problem, we can modify it so that we are cutting *characters* out of a magazine instead of whole words.

We can solve the simplified ransom note problem with characters by simply creating an array and counting the characters. Each spot in the array corresponds to one letter. First, we count the number of times each character in the ransom note appears, and then we go through the magazine to see if we have all of those characters.

When we generalize the algorithm, we do a very similar thing. This time, rather than creating an array with character counts, we create a hash table that maps from a word to its frequency.

Optimize & Solve Technique #4: Base Case and Build

With Base Case and Build, we solve the problem first for a base case (e.g., n = 1) and then try to build up from there. When we get to more complex/interesting cases (often n = 3 or n = 4), we try to build those using the prior solutions.

> Example: Design an algorithm to print all permutations of a string. For simplicity, assume all characters are unique.

Consider a test string `abcdefg`.

```
Case "a" --> {"a"}
Case "ab" --> {"ab", "ba"}
Case "abc" --> ?
```

This is the first "interesting" case. If we had the answer to `P("ab")`, how could we generate `P("abc")`? Well, the additional letter is "c," so we can just stick c in at every possible point. That is:

```
P("abc") = insert "c" into all locations of all strings in P("ab")
P("abc") = insert "c" into all locations of all strings in {"ab","ba"}
P("abc") = merge({"cab", "acb", "abc"}, {"cba", "bca", bac"})
P("abc") = {"cab", "acb", "abc", "cba", "bca", bac"}
```

Now that we understand the pattern, we can develop a general recursive algorithm. We generate all permutations of a string $s_1 \ldots s_n$ by "chopping off" the last character and generating all permutations of $s_1 \ldots s_{n-1}$. Once we have the list of all permutations of $s_1 \ldots s_{n-1}$, we iterate through this list. For each string in it, we insert s_n into every location of the string.

Base Case and Build algorithms often lead to natural recursive algorithms.

Optimize & Solve Technique #5: Data Structure Brainstorm

This approach is certainly hacky, but it often works. We can simply run through a list of data structures and try to apply each one. This approach is useful because solving a problem may be trivial once it occurs to us to use, say, a tree.

> Example: Numbers are randomly generated and stored into an (expanding) array. How would you keep track of the median?

Our data structure brainstorm might look like the following:

- Linked list? Probably not. Linked lists tend not to do very well with accessing and sorting numbers.
- Array? Maybe, but you already have an array. Could you somehow keep the elements sorted? That's probably expensive. Let's hold off on this and return to it if it's needed.
- Binary tree? This is possible, since binary trees do fairly well with ordering. In fact, if the binary search tree is perfectly balanced, the top might be the median. But, be careful—if there's an even number of elements, the median is actually the average of the middle two elements. The middle two elements can't both be at the top. This is probably a workable algorithm, but let's come back to it.
- Heap? A heap is really good at basic ordering and keeping track of max and mins. This is actually interesting—if you had two heaps, you could keep track of the bigger half and the smaller half of the elements. The bigger half is kept in a min heap, such that the smallest element in the bigger half is at the root. The smaller half is kept in a max heap, such that the biggest element of the smaller half is at the root. Now, with these data structures, you have the potential median elements at the roots. If the heaps are no longer the same size, you can quickly "rebalance" the heaps by popping an element off the one heap and pushing it onto the other.

Note that the more problems you do, the more developed your instinct on which data structure to apply will be. You will also develop a more finely tuned instinct as to which of these approaches is the most useful.

Best Conceivable Runtime (BCR)

Considering the best conceivable runtime can offer a useful hint for some problem.

The best conceivable runtime is, literally, the *best* runtime you could *conceive* of a solution to a problem having. You can easily prove that there is no way you could beat the BCR.

For example, suppose you want to compute the number of elements that two arrays (of length A and B) have in common. You immediately know that you can't do that in better than $O(A + B)$ time because you have to "touch" each element in each array. $O(A + B)$ is the BCR.

Or, suppose you want to print all pairs of values within an array. You know you can't do that in better than $O(N^2)$ time because there are N^2 pairs to print.

Be careful though! Suppose your interviewer asks you to find all pairs with sum k within an array (assuming all distinct elements). Some candidates who have not fully mastered the concept of BCR will say that the BCR is $O(N^2)$ because you have to look at N^2 pairs.

That's not true. Just because you want all pairs with a particular sum doesn't mean you have to look at *all* pairs. In fact, you don't.

What's the relationship between the Best Conceivable Runtime and Best Case Runtime? Nothing at all! The Best Conceivable Runtime is for a *problem* and is largely a function of the inputs and outputs. It has no particular connection to a specific algorithm. In fact, if you compute the Best Conceivable Runtime by thinking about what *your* algorithm does, you're probably doing something wrong. The Best Case Runtime is for a specific algorithm (and is a mostly useless value).

Note that the best conceivable runtime is not necessarily achievable. It says only that you can't do *better* than it.

An Example of How to Use BCR

Question: Given two sorted arrays, find the number of elements in common. The arrays are the same length and each has all distinct elements.

Let's start with a good example. We'll underline the elements in common.

```
A: 13  27  35  40  49  55  59
B: 17  35  39  40  55  58  60
```

A brute force algorithm for this problem is to start with each element in A and search for it in B. This takes $O(N^2)$ time since for each of N elements in A, we need to do an $O(N)$ search in B.

The BCR is $O(N)$, because we know we will have to look at each element at least once and there are $2N$ total elements. (If we skipped an element, then the value of that element could change the result. For example, if we never looked at the last value in B, then that 60 could be a 59.)

Let's think about where we are right now. We have an $O(N^2)$ algorithm and we want to do better than that—potentially, but not necessarily, as fast as $O(N)$.

```
Brute Force:          O(N^2)
Optimal Algorithm:    ?
BCR:                  O(N)
```

What is between $O(N^2)$ and $O(N)$? Lots of things. Infinite things actually. We could theoretically have an algorithm that's $O(N \log(\log(\log(\log(N)))))$. However, both in interviews and in real life, that runtime doesn't come up a whole lot.

Try to remember this for your interview because it throws a lot of people off. Runtime is not a multiple choice question. Yes, it's very common to have a runtime that's $O(\log N)$, $O(N)$, $O(N \log N)$, $O(N^2)$ or $O(2^N)$. But you shouldn't assume that something has a particular runtime by sheer process of elimination. In fact, those times when you're confused about the runtime and so you want to take a guess—those are the times when you're most likely to have a non-obvious and less common runtime. Maybe the runtime is $O(N^2K)$, where N is the size of the array and K is the number of pairs. Derive, don't guess.

Most likely, we're driving towards an $O(N)$ algorithm or an $O(N \log N)$ algorithm. What does that tell us?

If we imagine our current algorithm's runtime as $O(N \times N)$, then getting to $O(N)$ or $O(N \times \log N)$ might mean reducing that second $O(N)$ in the equation to $O(1)$ or $O(\log N)$.

This is one way that BCR can be useful. We can use the runtimes to get a "hint" for what we need to reduce.

That second O(N) comes from searching. The array is sorted. Can we search in a sorted array in faster than O(N) time?

Why, yes. We can use binary search to find an element in a sorted array in O(log N) time.

We now have an improved algorithm: O(N log N).

```
Brute Force:         O(N²)
Improved Algorithm: O(N log N)
Optimal Algorithm:  ?
BCR:                 O(N)
```

Can we do even better? Doing better likely means reducing that O(log N) to O(1).

In general, we cannot search an array—even a sorted array—in better than O(log N) time. This is *not* the general case though. We're doing this search over and over again.

The BCR is telling us that we will never, ever have an algorithm that's faster than O(N). Therefore, any work we do in O(N) time is a "freebie"—it won't impact our runtime.

Re-read the list of optimization tips on page 64. Is there anything that can help us?

One of the tips there suggests precomputing or doing upfront work. Any upfront work we do in O(N) time is a freebie. It won't impact our runtime.

> This is another place where BCR can be useful. Any work you do that's less than or equal to the BCR is "free," in the sense that it won't impact your runtime. You might want to eliminate it eventually, but it's not a top priority just yet.

Our focus is still on reducing search from O(log N) to O(1). Any precomputation that's O(N) or less is "free."

In this case, we can just throw everything in B into a hash table. This will take O(N) time. Then, we just go through A and look up each element in the hash table. This look up (or search) is O(1), so our runtime is O(N).

Suppose our interviewer hits us with a question that makes us cringe: Can we do better?

No, not in terms of runtime. We have achieved the fastest possible runtime, therefore we cannot optimize the big O time. We could potentially optimize the space complexity.

> This is another place where BCR is useful. It tells us that we're "done" in terms of optimizing the runtime, and we should therefore turn our efforts to the space complexity.

In fact, even without the interviewer prompting us, we should have a question mark with respect to our algorithm. We would have achieved the exact same runtime if the data wasn't sorted. So why did the interviewer give us sorted arrays? That's not unheard of, but it is a bit strange.

Let's turn back to our example.

```
A: 13  27  35  40  49  55  59
B: 17  35  39  40  55  58  60
```

We're now looking for an algorithm that:

- Operates in O(1) space (probably). We already have an O(N) space algorithm with optimal runtime. If we want to use less additional space, that probably means no additional space. Therefore, we need to drop the hash table.

- Operates in O(N) time (probably). We'll probably want to at least match the current best runtime, and we know we can't beat it.
- Uses the fact that the arrays are sorted.

Our best algorithm that doesn't use extra space was the binary search one. Let's think about optimizing that. We can try walking through the algorithm.

1. Do a binary search in B for A[0] = 13. Not found.
2. Do a binary search in B for A[1] = 27. Not found.
3. Do a binary search in B for A[2] = 35. Found at B[1].
4. Do a binary search in B for A[3] = 40. Found at B[5].
5. Do a binary search in B for A[4] = 49. Not found.
6. ...

Think about BUD. The bottleneck is the searching. Is there anything unnecessary or duplicated?

It's unnecessary that A[3] = 40 searched over all of B. We know that we just found 35 at B[1], so 40 certainly won't be before 35.

Each binary search should start where the last one left off.

In fact, we don't need to do a binary search at all now. We can just do a linear search. As long as the linear search in B is just picking up where the last one left off, we know that we're going to be operating in linear time.

1. Do a linear search in B for A[0] = 13. Start at B[0] = 17. Stop at B[0] = 17. Not found.
2. Do a linear search in B for A[1] = 27. Start at B[0] = 17. Stop at B[1] = 35. Not found.
3. Do a linear search in B for A[2] = 35. Start at B[1] = 35. Stop at B[1] = 35. Found.
4. Do a linear search in B for A[3] = 40. Start at B[2] = 39. Stop at B[3] = 40. Found.
5. Do a linear search in B for A[4] = 49. Start at B[3] = 40. Stop at B[4] = 55. Found.
6. ...

This algorithm is very similar to merging two sorted arrays. It operates in O(N) time and O(1) space.

We have now reached the BCR and have minimal space. We know that we cannot do better.

> This is another way we can use BCR. If you ever reach the BCR and have O(1) additional space, then you know that you can't optimize the big O time or space.

Best Conceivable Runtime is not a "real" algorithm concept, in that you won't find it in algorithm textbooks. But I have found it personally very useful, when solving problems myself, as well as while coaching people through problems.

If you're struggling to grasp it, make sure you understand big O time first (page 38). You need to master it. Once you do, figuring out the BCR of a problem should take literally seconds.

Handling Incorrect Answers

One of the most pervasive—and dangerous—rumors is that candidates need to get every question right. That's not quite true.

First, responses to interview questions shouldn't be thought of as "correct" or "incorrect." When I evaluate how someone performed in an interview, I never think, "How many questions did they get right?" It's not a binary evaluation. Rather, it's about how optimal their final solution was, how long it took them to get there, how much help they needed, and how clean was their code. There is a range of factors.

Second, your performance is evaluated *in comparison to other candidates*. For example, if you solve a question optimally in 15 minutes, and someone else solves an easier question in five minutes, did that person do better than you? Maybe, but maybe not. If you are asked really easy questions, then you might be expected to get optimal solutions really quickly. But if the questions are hard, then a number of mistakes are expected.

Third, many—possibly most—questions are too difficult to expect even a strong candidate to immediately spit out the optimal algorithm. The questions I tend to ask would take strong candidates typically 20 to 30 minutes to solve.

In evaluating thousands of hiring packets at Google, I have only once seen a candidate have a "flawless" set of interviews. Everyone else, including the hundreds who got offers, made mistakes.

When You've Heard a Question Before

If you've heard a question before, admit this to your interviewer. Your interviewer is asking you these questions in order to evaluate your problem-solving skills. If you already know the question, then you aren't giving them the opportunity to evaluate you.

Additionally, your interviewer may find it highly dishonest if you don't reveal that you know the question. (And, conversely, you'll get big honesty points if you do reveal this.)

The "Perfect" Language for Interviews

At many of the top companies, interviewers aren't picky about languages. They're more interested in how well you solve the problems than whether you know a specific language.

Other companies though are more tied to a language and are interested in seeing how well you can code in a particular language.

If you're given a choice of languages, then you should probably pick whatever language you're most comfortable with.

That said, if you have several good languages, you should keep in mind the following.

Prevalence

It's not required, but it is ideal for your interviewer to know the language you're coding in. A more widely known language can be better for this reason.

Language Readability

Even if your interviewer doesn't know your programming language, they should hopefully be able to basically understand it. Some languages are more naturally readable than others, due to their similarity to other languages.

For example, Java is fairly easy for people to understand, even if they haven't worked in it. Most people have worked in something with Java-like syntax, such as C and C++.

However, languages such as Scala or Objective C have fairly different syntax.

Potential Problems

Some languages just open you up to potential issues. For example, using C++ means that, in addition to all the usual bugs you can have in your code, you can have memory management and pointer issues.

Verbosity

Some languages are more verbose than others. Java for example is a fairly verbose language as compared with Python. Just compare the following code snippets.

Python:

```
1  dict = {"left": 1, "right": 2, "top": 3, "bottom": 4};
```

Java:

```
1  HashMap<String, Integer> dict = new HashMap<String, Integer>().
2  dict.put("left", 1);
3  dict.put("right", 2);
4  dict.put("top", 3);
5  dict.put("bottom", 4);
```

However, some of the verbosity of Java can be reduced by abbreviating code. I could imagine a candidate on a whiteboard writing something like this:

```
1  HM<S, I> dict = new HM<S, I>().
2  dict.put("left", 1);
3  ...      "right", 2
4  ...      "top", 3
5  ...      "bottom", 4
```

The candidate would need to explain the abbreviations, but most interviewers wouldn't mind.

Ease of Use

Some operations are easier in some languages than others. For example, in Python, you can very easily return multiple values from a function. In Java, the same action would require a new class. This can be handy for certain problems.

Similar to the above though, this can be mitigated by just abbreviating code or presuming methods that you don't actually have. For example, if one language provides a function to transpose a matrix and another language doesn't, this doesn't necessarily make the first language much better to code in (for a problem that needs such a function). You could just assume that the other language has a similar method.

▸ What Good Coding Looks Like

You probably know by now that employers want to see that you write "good, clean" code. But what does this really mean, and how is this demonstrated in an interview?

Broadly speaking, good code has the following properties:

- **Correct:** The code should operate correctly on all expected and unexpected inputs.
- **Efficient:** The code should operate as efficiently as possible in terms of both time and space. This "efficiency" includes both the asymptotic (big O) efficiency and the practical, real-life efficiency. That is, a

constant factor might get dropped when you compute the big O time, but in real life, it can very much matter.

- **Simple:** If you can do something in 10 lines instead of 100, you should. Code should be as quick as possible for a developer to write.
- **Readable:** A different developer should be able to read your code and understand what it does and how it does it. Readable code has comments where necessary, but it implements things in an easily understandable way. That means that your fancy code that does a bunch of complex bit shifting is not necessarily *good* code.
- **Maintainable:** Code should be reasonably adaptable to changes during the life cycle of a product and should be easy to maintain by other developers, as well as the initial developer.

Striving for these aspects requires a balancing act. For example, it's often advisable to sacrifice some degree of efficiency to make code more maintainable, and vice versa.

You should think about these elements as you code during an interview. The following aspects of code are more specific ways to demonstrate the earlier list.

Use Data Structures Generously

Suppose you were asked to write a function to add two simple mathematical expressions which are of the form $Ax^a + Bx^b + \ldots$ (where the coefficients and exponents can be any positive or negative real number). That is, the expression is a sequence of terms, where each term is simply a constant times an exponent. The interviewer also adds that she doesn't want you to have to do string parsing, so you can use whatever data structure you'd like to hold the expressions.

There are a number of different ways you can implement this.

Bad Implementation

A bad implementation would be to store the expression as a single array of doubles, where the kth element corresponds to the coefficient of the x^k term in the expression. This structure is problematic because it could not support expressions with negative or non-integer exponents. It would also require an array of 1000 elements to store just the expression x^{1000}.

```
1  int[] sum(double[] expr1, double[] expr2) {
2    ...
3  }
```

Less Bad Implementation

A slightly less bad implementation would be to store the expression as a set of two arrays, `coefficients` and `exponents`. Under this approach, the terms of the expression are stored in any order, but "matched" such that the `i`th term of the expression is represented by `coefficients[i] * x^exponents[i]`.

Under this implementation, if `coefficients[p] = k` and `exponents[p] = m`, then the pth term is kx^m. Although this doesn't have the same limitations as the earlier solution, it's still very messy. You need to keep track of two arrays for just one expression. Expressions could have "undefined" values if the arrays were of different lengths. And returning an expression is annoying because you need to return two arrays.

```
1  ??? sum(double[] coeffs1, double[] expon1, double[] coeffs2, double[] expon2) {
2    ...
3  }
```

Good Implementation

A good implementation for this problem is to design your own data structure for the expression.

```
class ExprTerm {
   double coefficient;
   double exponent;
}

ExprTerm[] sum(ExprTerm[] expr1, ExprTerm[] expr2) {
   ...
}
```

Some might (and have) argued that this is "over-optimizing." Perhaps so, perhaps not. Regardless of whether you think it's over-optimizing, the above code demonstrates that you think about how to design your code and don't just slop something together in the fastest way possible.

Appropriate Code Reuse

Suppose you were asked to write a function to check if the value of a binary number (passed as a string) equals the hexadecimal representation of a string.

An elegant implementation of this problem leverages code reuse.

```
boolean compareBinToHex(String binary, String hex) {
   int n1 = convertFromBase(binary, 2);
   int n2 = convertFromBase(hex, 16);
   if (n1 < 0 || n2 < 0) {
      return false;
   }
   return n1 == n2;
}

int convertFromBase(String number, int base) {
   if (base < 2 || (base > 10 && base != 16)) return -1;
   int value = 0;
   for (int i = number.length() - 1; i >= 0; i--) {
      int digit = digitToValue(number.charAt(i));
      if (digit < 0 || digit >= base) {
         return -1;
      }
      int exp = number.length() - 1 - i;
      value += digit * Math.pow(base, exp);
   }
   return value;
}

int digitToValue(char c) { ... }
```

We could have implemented separate code to convert a binary number and a hexadecimal code, but this just makes our code harder to write and harder to maintain. Instead, we reuse code by writing one `convertFromBase` method and one `digitToValue` method.

Modular

Writing modular code means separating isolated chunks of code out into their own methods. This helps keep the code more maintainable, readable, and testable.