1 StringBuilder

Imagine you were concatenating a list of strings, as shown below. What would the running time of this code be? For simplicity, assume that the strings are all the same length (call this x) and that there are n strings.

```
1  String joinWords(String[] words) {
2     String sentence = "";
3     for (String w : words) {
4        sentence = sentence + w;
5     )
6     return sentence;
7  )
```

On each concatenation, a new copy of the string is created, and the two strings are copied over, character by character. The first iteration requires us to copy x characters. The second iteration requires copying 2x characters. The third iteration requires 3x, and so on. The total time therefore is $O(x + 2x + \ldots + nx)$. This reduces to $O(xn^2)$.

Why is it $O(xn^2)$? Because $1 + 2 + \ldots + n$ equals $n(n+1)/2$, or $O(n^2)$.

StringBuilder can help you avoid this problem. StringBuilder simply creates a resizable array of all the strings, copying them back to a string only when necessary.

```
1  String joinWords(String[] words) {
2     StringBuilder sentence = new StringBuilder();
3     for (String w : words) {
4        sentence.append(w);
5     )
6     return sentence.toString();
7  )
```

A good exercise to practice strings, arrays, and general data structures is to implement your own version of StringBuilder, HashTable and ArrayList.

Additional Reading: Hash Table Collision Resolution (pg 522), Rabin-Karp Substring Search (pg 636).

2 ArrayList & Resizable Arrays

In some languages, arrays (often called lists in this case) are automatically resizable. The array or list will grow as you append items. In other languages, like Java, arrays are fixed length. The size is defined when you create the array.

When you need an array-like data structure that offers dynamic resizing, you would usually use an ArrayList. An ArrayList is an array that resizes itself as needed while still providing O(1) access. A typical implementation is that when the array is full, the array doubles in size. Each doubling takes O(n) time, but happens so rarely that its amortized insertion time is still O(1).

```
1  ArrayList<String> merge(String[] words, String[] more) {
2    ArrayList<String> sentence = new ArrayList<String>();
3    for (String w : words) sentence.add(w);
4    for (String w : more) sentence.add(w);
5    return sentence;
6  )
```

This is an essential data structure for interviews. Be sure you are comfortable with dynamically resizable arrays/lists in whatever language you will be working with. Note that the name of the data structure as well as the "resizing factor" (which is 2 in Java) can vary.

Why is the amortized insertion runtime O(1)?

Suppose you have an array of size N. We can work backwards to compute how many elements we copied at each capacity increase. Observe that when we increase the array to K elements, the array was previously half that size. Therefore, we needed to copy $K/2$ elements.

```
final capacity increase    : n/2 elements to copy
previous capacity increase : n/4 elements to copy
previous capacity increase : n/8 elements to copy
previous capacity increase : n/16 elements to copy
...
second capacity increase   : 2 elements to copy
first capacity increase    : 1 element to copy
```

Therefore, the total number of copies to insert N elements is roughly $N/2 + N/4 + N/8 + \ldots + 2 + 1$, which is just less than N.

> If the sum of this series isn't obvious to you, imagine this: Suppose you have a kilometer—long walk to the store. You walk 0.5 kilometers, and then 0.25 kilometers, and then 0.125 kilometers, and so on. You will never exceed one kilometer (although you'll get very close to it).

Therefore, inserting N elements takes O(N) work total. Each insertion is O(1) on average, even though some insertions take O(N) time in the worst case.

Concatenation of Strings using String

```
String joinWords(String[] words){
    String sentence = "";
    for(String w: words){
        sentence = sentence + w;
    }
    return sentence;
}
```

Complexity: $O(n^2)$

Concatenation of Strings using "StringBuilder"

```
String joinWords(String[] words){
    StringBuilder sentence = new
                             StringBuilder();
    for(String w: words){
        sentence.append(w);
    }
    return sentence.toString();
}
```

Merging Strings in Dynamic Arrays.

```
ArrayList<String> merge(String[] words,
                        String[] more) {

    ArrayList<String> sentence =
                        new ArrayList<String>();
    for(String w: words) sentence.add(w);
    for(String w: more) sentence.add(w);
    return sentence;

}
```

3 Hash Tables

A hash table is a data structure that maps keys to values for highly efficient lookup. There are a number of ways of implementing this. Here, we will describe a simple but common implementation.

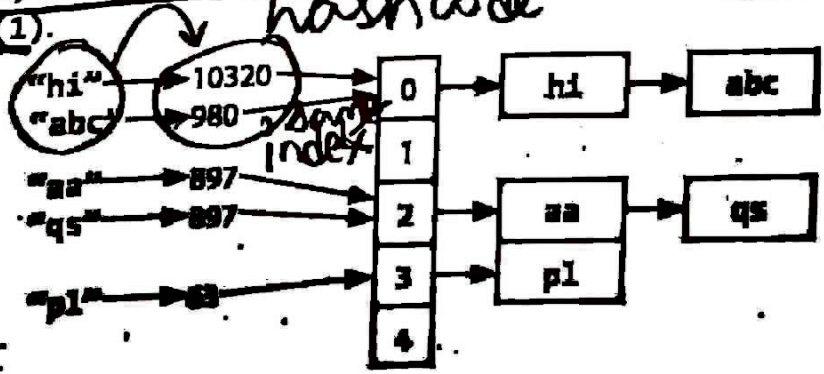
In this simple implementation, we use an array of linked lists and a hash code function. To insert a key (which might be a string or essentially any other data type) and value, we do the following:

1. First, compute the key's hash code, which will usually be an `int` or `long`. Note that two different keys could have the same hash code, as there may be an infinite number of keys and a finite number of ints.
2. Then, map the hash code to an index in the array. This could be done with something like `hash (key) % array_length`. Two different hash codes could, of course, map to the same index.
3. At this index, there is a linked list of keys and values. Store the key and value in this index. We must use a linked list because of collisions: you could have two different keys with the same hash code, or two different hash codes that map to the same index.

key → hashCode → index → LL

To retrieve the value pair by its key, you repeat this process. Compute the hash code from the key, and then compute the index from the hash code. Then, search through the linked list for the value with this key.

If the number of collisions is very high, the worst case runtime is `O(N)`, where `N` is the number of keys. However, we generally assume a good implementation that keeps collisions to a minimum, in which case the lookup time is `O(1)`.



Alternatively, we can implement the hash table with a balanced binary search tree. This gives us an O(log N) lookup time. The advantage of this is potentially using less space, since we no longer allocate a large array. We can also iterate through the keys in order, which can be useful sometimes.

Rabin-Karp Substring Search

The brute force way to search for a substring S in a larger string B takes $O(s(b-s))$ time, where s is the length of S and b is the length of B. We do this by searching through the first `b - s + 1` characters in B and, for each, checking if the next s characters match S.

The Rabin-Karp algorithm optimizes this with a little trick: if two strings are the same, they must have the same hash value. (The converse, however, is not true. Two different strings can have the same hash value.)

Therefore, if we efficiently precompute a hash value for each sequence of s characters within B, we can find the locations of S in $O(b)$ time. We then just need to validate that those locations really do match S.

For example, imagine our hash function was simply the sum of each character (where space = 0, a = 1, b = 2, and so on). If S is `ear` and `B = doe are hearing me`, we'd then just be looking for sequences where the sum is 24 (e + a + r). This happens three times. For each of those locations, we'd check if the string really is `ear`.

char	d	o	e		a	r	e		h	e	a	r	i	n	g		m	e
code:	4	15	5	0	1	18	5	0	8	5	1	18	9	14	7	0	13	5
sum of next 3:	[illegible]	20	6	19	[illegible]	23	13	13	14	[illegible]	28	41	30	21	20	18		

If we computed these sums by doing `hash('doe')`, then `hash('oe ')`, then `hash('e a')`, and so on, we would still be at $O(s(b-s))$ time. longer time

Instead, we compute the hash values by recognizing that `hash('oe ') = hash('doe') - code('d') + code(' ')`. This takes $O(b)$ time to compute all the hashes. optimised

You might argue that, still, in the worst case this will take $O(s(b-s))$ time since many of the hash values could match. That's absolutely true—for this hash function.

In practice, we would use a better *rolling* hash function, such as the Rabin fingerprint. This essentially treats a string like doe as a base 128 (or however many characters are in our alphabet) number.

$$\text{hash('doe')} = \text{code('d')} * 128^2 + \text{code('o')} * 128^1 + \text{code('e')} * 128^0$$

This hash function will allow us to remove the d, shift the o and e, and then add in the space.

$$\text{hash('oe ')} = (\text{hash('doe')} - \text{code('d')} * 128^2) * 128 + \text{code(' ')}$$

This will considerably cut down on the number of false matches. Using a good hash function like this will give us expected time complexity of $O(s + b)$, although the worst case is $O(sb)$.

Usage of this algorithm comes up fairly frequently in interviews, so it's useful to know that you can identify substrings in linear time.

Hash Table Collision Resolution

Essentially any hash table can have collisions. There are a number of ways of handling this.

Chaining with Linked Lists

With this approach (which is the most common), the hash table's array maps to a linked list of items. We just add items to this linked list. As long as the number of collisions is fairly small, this will be quite efficient.

In the worst case, lookup is O(n), where n is the number of elements in the hash table. This would only happen with either some very strange data or a very poor hash function (or both).

Chaining with Binary Search Trees

Rather than storing collisions in a linked list, we could store collisions in a binary search tree. This will bring the worst-case runtime to `O(log n)`.

In practice, we would rarely take this approach unless we expected an extremely nonuniform distribution.

Terminology; Storage of Strings

3.1 Let W be the string ABCD. (*a*) Find the length of W. (*b*) List all substrings of W. (*c*) List all the initial substrings of W.

(a) The number of characters in W is its length, so 4 is the length of W.

Greek letter big lambda

3.21

(b) Any subsequence of characters of W is a substring of W. There are 11 such substrings:

Substrings:	ABCD,	ABC, BCD,	AB, BC, CD,	A, B, C, D,	Λ
Lengths:	4	3	2	1	0

(Here Λ denotes the empty string.)

(c) The initial substrings are ABCD, ABC, AB, A, Λ; that is, both the empty string and those substrings that begin with A.

3.7 PATTERN MATCHING ALGORITHMS

Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T. We assume that the length of P does not exceed the length of T. This section discusses two pattern matching algorithms. We also discuss the complexity of the algorithms so we can compare their efficiencies.

Remark: During the discussion of pattern matching algorithms, characters are sometimes denoted by lowercase letters (a, b, c, ...) and exponents may be used to denote repetition; e.g.,

$a^2b^3ab^2$ for *aabbbabb* and $(cd)^3$ for *cdcdcd*

In addition, the empty string may be denoted by A, the Greek letter lambda, and the concatenation of strings X and Y may be denoted by X·Y or, simply, XY.

3.6 WORD PROCESSING

In earlier times, character data processed by the computer consisted mainly of data items, such as names and addresses. Today the computer also processes printed matter, such as letters, articles and reports. It is in this latter context that we use the term "word processing."

Given some printed text, the operations usually associated with word processing are the following:

(a) *Replacement.* Replacing one string in the text by another.
(b) *Insertion.* Inserting a string in the middle of the text.
(c) *Deletion.* Deleting a string from the text.

Insertion

Suppose in a given text T we want to insert a string S so that S begins in position K. We denote this operation by

T K S

INSERT(text, position, string)

For example,

1 2 3

INSERT ('ABCDEFG', 3, 'XYZ') = 'ABXYZCDEFG'

1 2 3 4 5 6

INSERT ('ABCDEFG', 6, 'XYZ') = 'ABCDEXYZFG'

This INSERT

section as follows:

INSERT(T, K, S) = SUBSTRING(T, 1, K−1) // S // SUBSTRING(T, K, LENGTH(T) − K + 1)

This is, the initial substring of T before the position K, which has length K − 1, is concatenated with the string S, and the result is concatenated with the remaining part of T, which begins in position K and has length LENGTH(T) − (K − 1) = Length (T) − K + 1. (We are assuming implicitly that T is a dyamic variable and that the size of T will not become too large.)

2 **Deletion**

2 Deletion

Suppose in a given text T we want to delete the substring which begins in position K and has length L. We denote this operation by

DELETE(text, position, length)

T K L

For example,

DELETE('ABCDEFG', 4, 2) = 'ABCFG'
DELETE('ABCDEFG', 2, 4) = 'AFG'

We assume that nothing is deleted if position K = 0. Thus

DELETE('ABCDEFG', 0, 2) = 'ABCDEFG'

The importance of this "zero case" is seen later.

The DELETE function can be implemented using the string operations given in the preceding section as follows:

DELETE(T, K, L) =
SUBSTRING(T, 1, K – 1)//SUBSTRING(T, K + L, LENGTH(T) – K – L + 1)

That is, the initial substring of T before position K is concatenated with the terminal substring of T beginning in position K + L. The length of the initial substring is K – 1, and the length of the terminal substring is:

LENGTH(T) – (K + L – 1) = LENGTH(T) – K – L + 1

We also assume that DELETE(T, K, L) = T when K = 0.

Now suppose text T and pattern P are given and we want to delete from T the first occurrence of the pattern P. This can be accomplished by using the above DELETE function as follows:

DELETE(T, INDEX(T, P), LENGTH(P))

That is, in the text T, we first compute INDEX(T, P), the position where P first occurs in T, and then we compute LENGTH(P), the number of characters in P. Recall that when INDEX(T, P) = 0 (i.e., when P does not occur in T) the text T is not changed.

Suppose after reading into the computer a text T and a pattern P, we want to delete every occurrence of the pattern P in the text T. This can be accomplished by repeatedly applying

DELETE(T, INDEX(T, P), LENGTH(P))

until INDEX(T, P) = 0 (i.e., until P does not appear in T). An algorithm which accomplishes this follows.

4. Exit.

We emphasize that after each deletion, the length of T decreases and hence the algorithm must stop. However, the number of times the loop is executed may exceed the number of times P appears in the original text T, as illustrated in the following example.

Example 3.7

The above example shows that when a text T is changed by a deletion, patterns may occur that did not appear originally.

QUICK REVIEW

1. A data type is simple if variables of that type can hold only one value at a time.
2. In a structured data type, each data item is a collection of other data items.
3. An array is a structured data type with a fixed number of components. Every component is of the same type, and components are accessed using their relative positions in the array.
4. Elements of a one-dimensional array are arranged in the form of a list.
5. There is no check on whether an array index is out of bounds.

2 Approaching Coding Questions

Coding interviews typically last 30 to 45 minutes and come in a variety of formats. Early in the interview process, coding interviews are often conducted via remote coding assessment tools like HackerRank, Codility, or CoderPad. During final-round onsite interviews, it's typical to write code on a whiteboard. Regardless of the format, the approach outlined below to solve coding interview problems applies.

1

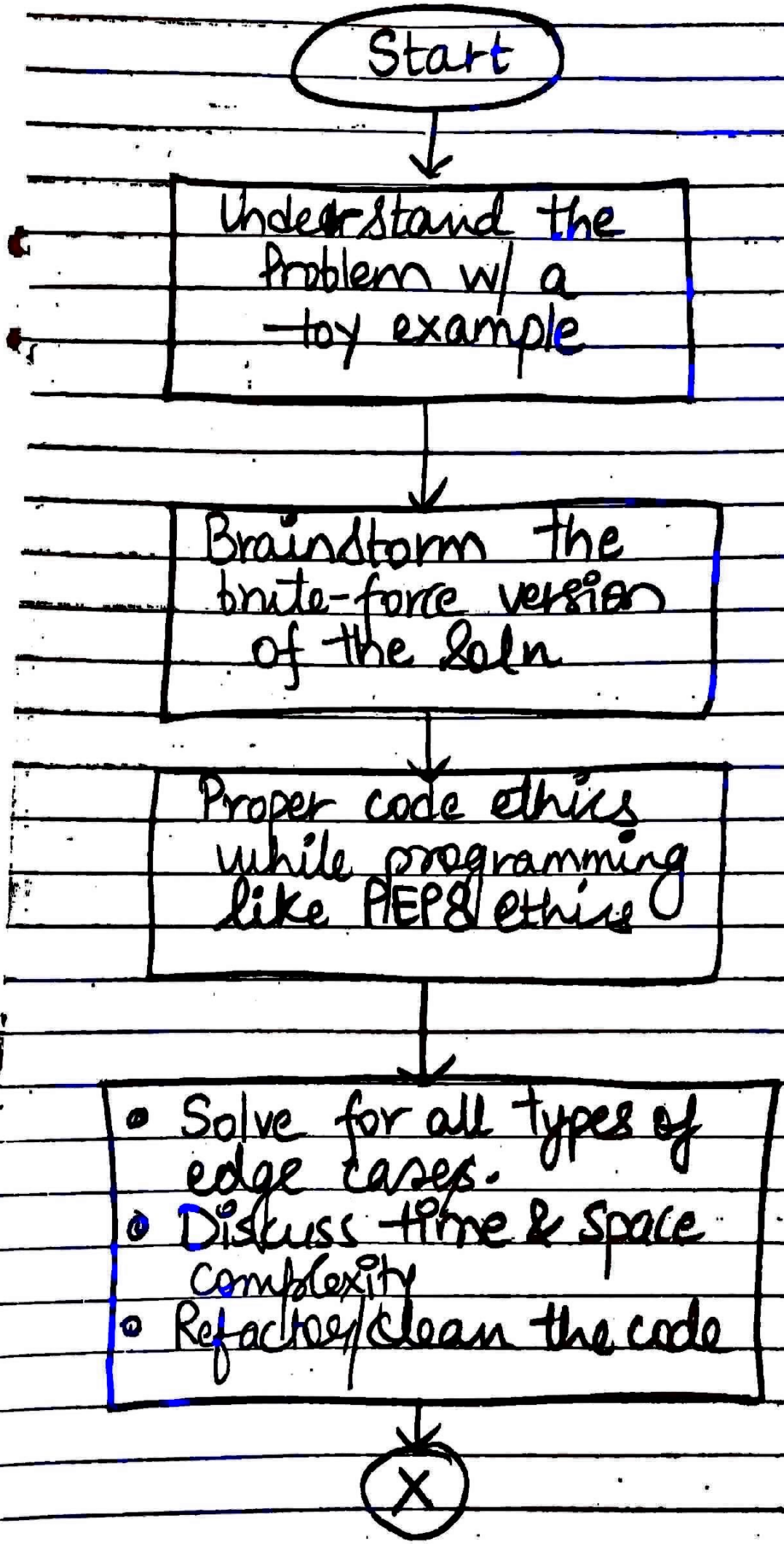
After receiving the problem: Don't jump right into coding. It's crucial first to make sure you are solving the correct problem. Due to language barriers, misplaced assumptions, and subtle nuances that are easy to miss, misunderstanding the problem is a frequent occurrence. To prevent this, make sure to repeat the question back to the interviewer so that the two of you are on the same page. Clarify any assumptions made, like the input format and range, and be sure to ask if the input can be assumed to be non-null or well formed. As a final test to see if you've understood the problem, work through an example input and see if you get the expected output. Only after you've done these steps are you ready to begin solving the problem.

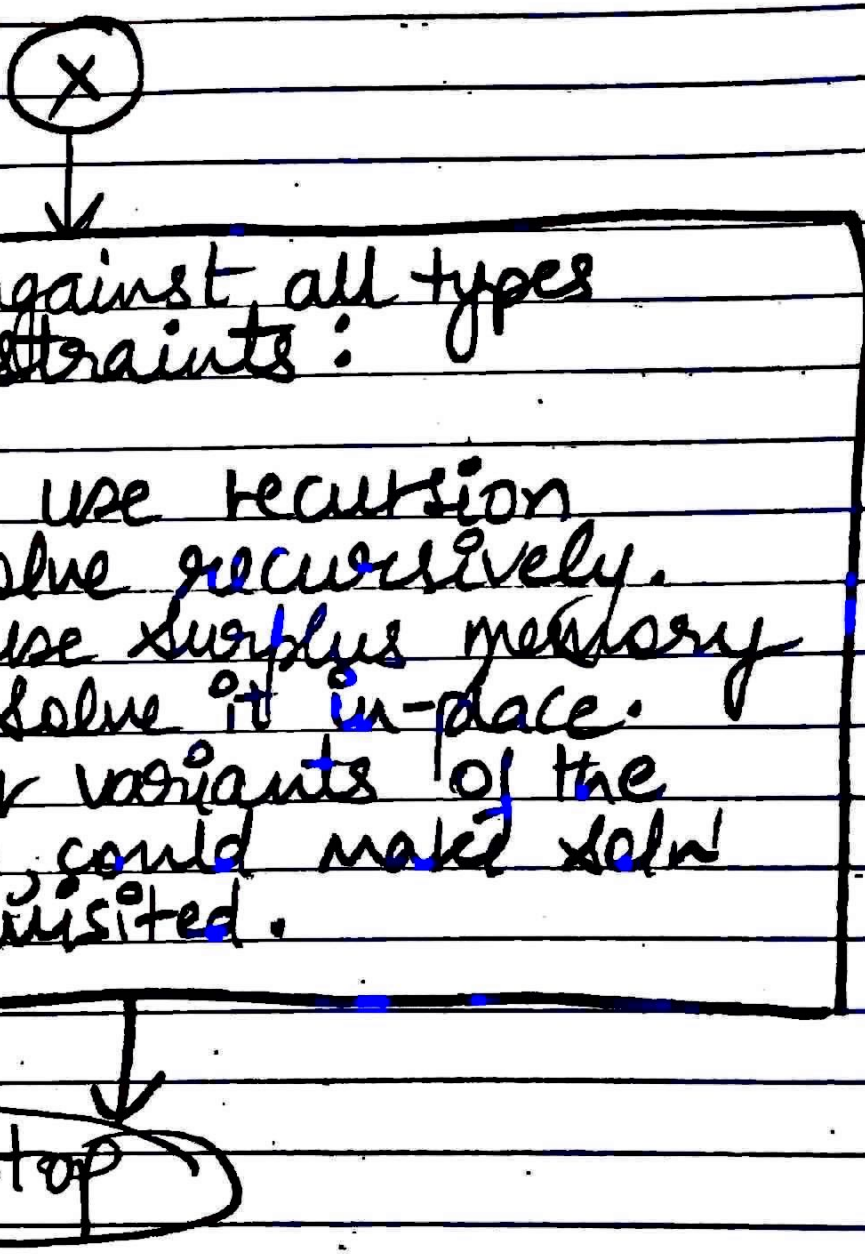
2 ***When brainstorming a solution:*** First, explain at a high level how you could tackle the question. This usually means discussing the brute-force solution. Then, try to gain an intuition for why this brute-force solution might be inefficient, and how you could improve upon it. If you're able to land on a more optimal approach, articulate how and why this new solution is better than the first brute-force solution provided. Only after you've settled on a solution is it time to begin coding.

3 ***When coding the solution:*** Explain what you are coding. Don't just sit there typing away, leaving your interviewer in the dark. Because coding interviews often let you pick the language you write code in, you're expected to be proficient in the programming language you chose. As such, avoid pseudocode in favor of proper compilable code. While there is time pressure, don't take many shortcuts when coding. Use clear variable names and follow good code organization principles. Write well-styled code — for example, following PEP 8 guidelines when coding in Python. While you are allowed to cut some corners, like assuming a helper method exists, be explicit about it and offer to fix this later on.

4 ***After you're done coding:*** Make sure there are no mistakes or edge cases you didn't handle. Then write and execute test cases to prove you solved the problem.

At this point, the interviewer should dictate which direction the interview heads. They may ask about the time and space complexity of your code. Sometimes they may ask you to refactor and clean the code, especially if you cut some corners while coding the solution. They may also extend the problem, often with a new constraint. For example, they may ask you not to use recursion and instead tell you to solve the problem recursively. Or, they might ask you to not use surplus memory and instead solve the problem in place. Sometimes, they may pose a tougher variant of the problem as a follow-up, which might require starting the problem-solving process all over again.


Start
Understand the problem w/ a toy example
Brainstorm the brute-force version of the soln
Proper code ethics while programming like PEP8 ethics
• Solve for all types of edge cases.
• Discuss time & space complexity
• Refactor/clean the code
X


X
Solve against all types of constraints:
• Don't use recursion but solve recursively.
• Don't use surplus memory and solve it in-place.
• Tougher variants of the problem could make soln be revisited.
Stop

3 Replacement

Suppose in a given text T we want to replace the first occurrence of a pattern P_1 by a pattern P_2. We will denote this operation by

REPLACE(text, $pattern_1$, $pattern_2$)

For example

REPLACE('XABYABZ', 'AB', 'C') = 'XCYABZ'
REPLACE('XABYABZ' , 'BA', 'C') = 'XABYABZ'

In the second case, the pattern BA does not occur, and hence there is no change.

We note that this REPLACE function can be expressed as a deletion followed by an insertion if we use the preceding DELETE and INSERT functions. Specifically, the REPLACE function can be executed by using the following three steps:

K := INDEX(T, P_1)
T := DELETE(T, K, LENGTH(P_1))
INSERT(T, K, P_2)

The first two steps delete P_1 from T, and the third step inserts P_2 in the position K from which P_1 was deleted.

Suppose a text T and patterns P and Q are in the memory of a computer. Suppose we want to replace every occurrence of the pattern P in T by the pattern Q. This might be accomplished by repeatedly applying

REPLACE(T, P, Q)

until INDEX(T, P) = 0 (i.e., until P does not appear in T). An algorithm which does this follows.

3.5 STRING OPERATIONS

Although a string may be viewed simply as a sequence or linear array of characters, there is a fundamental difference in use between strings and other types of arrays. Specifically, groups of consecutive elements in a string (such as words, phrases and sentences), called *substrings*, may be units unto themselves. Furthermore, the basic units of access in a string are usually these substrings, not individual characters.

Consider, for example, the string

'TO BE OR NOT TO BE'

We may view the string as the 18-character sequence T, O, □, B, ..., E. However, the substrings TO, BE, OR, ... have their own meaning.

On the other hand, consider an 18-element linear array of 18 integers,

4, 8, 6, 15, 9, 5, 4, 13, 8, 5, 11, 9, 9, 13, 7, 10, 6, 11

The basic unit of access in such an array is usually an individual element. Groups of consecutive elements normally do not have any special meaning.

For the above reason, various string operations have been developed which are not normally used with other kinds of arrays. This section discusses these string-oriented operations. The next section shows how these operations are used in word processing. (Unless otherwise stated or implied, we assume our character-type variables are dynamic and have a variable length determined by the context in which the variable is used.)

(1) Substring

(1) name of string
(2) position of Substring's 1st character

Accessing a substring from a given string requires three pieces of information: (1) the name of the string or the string itself, (2) the position of the first character of the substring in the given string and (3) the length of the substring or the position of the last character of the substring. We call this operation SUBSTRING. Specifically, we write (3) Length of substring

SUBSTRING(string, initial, length)

S K L

to denote the substring of a string S beginning in a position K and having a length L.

2 Indexing

Indexing, also called *pattern matching*, refers to finding the position where a string pattern P first appears in a given string text T. We call this operation INDEX and write

```
INDEX(text, pattern)
```

Index(T, P)

If the pattern P does not appear in the text T, then INDEX is assigned the value 0. The arguments "text" and "pattern" can be either string constants or string variables.

Observe the reverse order of the arguments in UCSD Pascal.

3 Concatenation

Let S_1 and S_2 be strings. Recall (Sec. 3.2) that the *concatenation* of S_1 and S_2, which we denote by $S_1 /// S_2$, is the string consisting of the characters of S_1 followed by the characters of S_2.

4 Length

The number of characters in a string is called its length. We will write

LENGTH(string)

for the length of a given string. Thus

LENGTH('COMPUTER') = 8 LENGTH('0') = 0 LENGTH(' ') = 0

Some of the programming languages denote this function as follows:

PL/1:	LENGTH(string)
BASIC:	LEN(string)
UCSD Pascal:	LENGTH(string)
SNOBOL:	SIZE(string)

FORTRAN and standard Pascal, which use fixed-length string variables, do not have any built-in LENGTH functions for strings. However, such variables may be viewed as having variable length if one ignores all trailing blanks. Accordingly, one could write a subprogram LENGTH in these languages so that

Historically, computers were first used for processing numerical data. Today, computers are frequently used for processing nonnumerical data, called *character data*. This chapter discusses how such data are stored and processed by the computer.

One of the primary applications of computers today is in the field of word processing. Such processing usually involves some type of pattern matching, as in checking to see if a particular word S appears in a given text T. We discuss this pattern matching problem in detail and, moreover, present two different pattern matching algorithms. The complexity of these algorithms is also investigated.

Computer terminology usually uses the term "string" for a sequence of characters rather than the term "word," since "word" has another meaning in computer science. For this reason, many texts sometimes use the expression "string processing," "string manipulation" or "text editing" instead of the expression "word processing."

6. In C++, an array index starts with 0.
7. An array index can be any expression that evaluates to a nonnegative integer. The value of the index must always be less than the size of the array.
8. There are no aggregate operations on arrays, except for the input/output of character arrays (C-strings).
9. Arrays can be initialized during their declaration. If there are fewer initial values than the array size, the remaining elements are initialized to 0.
10. The base address of an array is the address of the first array component. For example, if `list` is a one-dimensional array, the base address of `list` is the address of `list[0]`.

11. When declaring a one-dimensional array as a formal parameter, you usually omit the array size. If you specify the size of a one-dimensional array in the formal parameter declaration, the compiler will ignore the size.

12. In a function call statement, when passing an array as an actual parameter, you use only its name.

13. As parameters to functions, arrays are passed by reference only.

14. Because as parameters, arrays are passed by reference only, when declaring an array as a formal parameter, you do not use the symbol & after the data type.

15. A function cannot return a value of type array.

13.5.4 Search Engine Indexing

The World Wide Web contains a huge collection of text documents (Web pages). Information about these pages are gathered by a program called a ***Web crawler***, which then stores this information in a special dictionary database. A Web ***search engine*** allows users to retrieve relevant information from this database, thereby identifying relevant pages on the Web containing given keywords. In this section, we present a simplified model of a search engine.

Inverted Files

The core information stored by a search engine is a dictionary, called an ***inverted index*** or ***inverted file***, storing key-value pairs (w, L), where w is a word and L is a collection of pages containing word w. The keys (words) in this dictionary are called ***index terms*** and should be a set of vocabulary entries and proper nouns as large as possible. The elements in this dictionary are called ***occurrence lists*** and should cover as many Web pages as possible.

We can efficiently implement an inverted index with a data structure consisting of the following:

1. An array storing the occurrence lists of the terms (in no particular order).
2. A compressed trie for the set of index terms, where each leaf stores the index of the occurrence list of the associated term.

The reason for storing the occurrence lists outside the trie is to keep the size of the

A *trie* (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching. The main application for tries is in information retrieval. Indeed, the name "trie" comes from the word "re*trie*val." In an information retrieval application, such as a search for a certain DNA sequence in a genomic database, we are given a collection S of strings, all defined using the same alphabet. The primary query operations that tries support are pattern matching and *prefix matching*. The latter operation involves being given a string X, and looking for all the strings in S that contain X as a prefix.

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	0(1)	0(n)	0(n)	0(n)	0(1)	0(n)	0(n)	0(n)	0(n)
Stack	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(n)
Queue	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(n)
Linked List	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(n)
Hash Map	N/A	0(1)	0(1)	0(1)	N/A	0(n)	0(n)	0(n)	0(n)
Binary Search Tree	0(log (n))	0(log (n))	0(log (n))	0(log (n))	0(n)	0(n)	0(n)	0(n)	0(n)

Arrays

An array is a series of consecutive elements stored sequentially in memory. Arrays are optimal for accessing elements at particular indices, with an O(1) access and index time. However, they are slower for searching and deleting a specific value, with an O(N) runtime, unless sorted. An array's simplicity makes it one of the most commonly used data structures during coding interviews.

HashMap: The HashMap collection is widely used, both in interviews and in the real world. We've provided a snippet of the syntax below.

```
1  HashMap<String, String> map = new HashMap<String, String>();
2  map.put("one", "uno");
3  map.put("two", "dos");
4  System.out.println(map.get("one"));
```

Before your interview, make sure you're very comfortable with the above syntax. You'll need it.

Java's collection framework is incredibly useful, and you will see it used throughout this book. Here are some of the most useful items:

`ArrayList`: An `ArrayList` is a dynamically resizing array, which grows as you insert elements.

```
1  ArrayList<String> myArr = new ArrayList<String>();
2  myArr.add("one");
3  myArr.add("two");
4  System.out.println(myArr.get(0)); /* prints <one> */
```

`Vector`: A `vector` is very similar to an `ArrayList`, except that it is synchronized. Its syntax is almost identical as well.

```
1 Vector<String> myVect = new Vector<String>();
2 myVect.add("one");
3 myVect.add("two");
4 System.out.println(myVect.get(0));
```

LinkedList: LinkedList is, of course, Java's built-in LinkedList class. Though it rarely comes up in

3.1 Contiguous vs. Linked Data Structures

Data structures can be neatly classified as either *contiguous* or *linked*, depending upon whether they are based on arrays or pointers:

- *Contiguously-allocated structures* are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- *Linked data structures* are composed of distinct chunks of memory bound together by *pointers*, and include lists, trees, and graph adjacency lists.

together by *pointers*, and include lists, trees, and graph adjacency lists.

In this section, we review the relative advantages of contiguous and linked data structures. These tradeoffs are more subtle than they appear at first glance, so I encourage readers to stick with me here even if you may be familiar with both types of structures.

3.1.1 Arrays

3.1.1 Arrays

The *array* is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its *index* or (equivalently) address.

A good analogy likens an array to a street full of houses, where each array element is equivalent to a house, and the index is equivalent to the house number. Assuming all the houses are equal size and numbered sequentially from 1 to n, we can compute the exact position of each house immediately from its address.[1]

Advantages of contiguously-allocated arrays include:

- *Constant-time access given the index* – Because the index of each element maps directly to a particular memory address, we can access arbitrary data items instantly provided we know the index.
- *Space efficiency* – Arrays consist purely of data, so no space is wasted with links or other formatting information. Further, end-of-record information is not needed because arrays are built from fixed-size records.
- *Memory locality* – A common programming idiom involves iterating through all the elements of a data structure. Arrays are good for this because they exhibit excellent memory locality. Physical continuity between successive data accesses helps exploit the high-speed *cache memory* on modern computer architectures.

The downside of arrays is that we cannot adjust their size in the middle of a program's execution. Our program will fail soon as we try to add the $(n+$

[1] Houses in Japanese cities are traditionally numbered in the order they were built, not by their physical location. This makes it extremely difficult to locate a Japanese address without a detailed map.

1)st customer, if we only allocate room for n records. We can compensate by allocating extremely large arrays, but this can waste space, again restricting what our programs can do.

Actually, we *can* efficiently enlarge arrays as we need them, through the miracle of *dynamic arrays*. Suppose we start with an array of size 1, and double its size from m to $2m$ each time we run out of space. This doubling process involves allocating a new contiguous array of size $2m$, copying the contents of the old array to the lower half of the new one, and returning the space used by the old array to the storage allocation system.

The apparent waste in this procedure involves the recopying of the old contents on each expansion. How many times might an element have to be recopied after a total of n insertions? Well, the first inserted element will have been recopied when the array expands after the first, second, fourth, eighth, ... insertions. It will take $\log_2 n$ doublings until the array gets to have n positions. However, most elements do not suffer much upheaval. Indeed, the $(n/2+1)$st through nth elements will move at most once and might never have to move at all.

If half the elements move once, a quarter of the elements twice, and so on, the total number of movements M is given by

partial sum — larger ∞ Sum — converges to 2

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n \sum_{i=1}^{\lg n} i/2^i \leq n \sum_{i=1}^{\infty} i/2^i = 2n$$

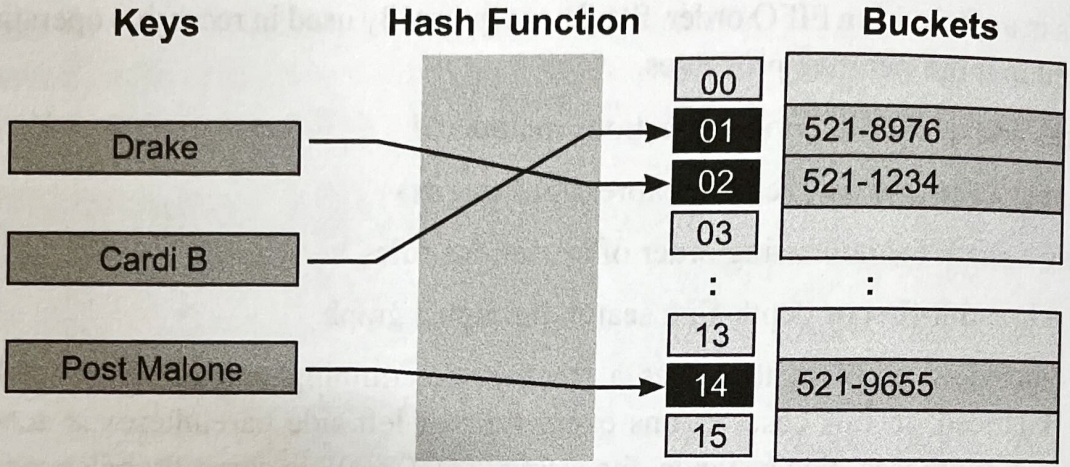
Infinite series.

Thus, each of the n elements move only two times on average, and the total work of managing the dynamic array is the same $O(n)$ as it would have been if a single array of sufficient size had been allocated in advance!

The primary thing lost using dynamic arrays is the guarantee that each array access takes constant time *in the worst case*. Now all the queries will be fast, except for those relatively few queries triggering array doubling. What we get instead is a promise that the nth array access will be completed quickly enough that the *total* effort expended so far will still be $O(n)$. Such *amortized* guarantees arise frequently in the analysis of data structures.

Hash Maps

A hash map stores key-value pairs. For every key, a hash map uses a hash function to compute an index, which locates the bucket where that key's corresponding value is stored. In Python, a dictionary offers support for key-value pairs and has the same functionality as a hash map.



While a hash function aims to map each key to a unique index, there will sometimes be "collisions" where different keys have the same index. In general, when you use a good hash function, expect the elements to be distributed evenly throughout the hash map. Hence, lookups, insertions, or deletions for a key take constant time.

Due to their optimal runtime properties, hash maps make a frequent appearance in coding interview questions.

Common h