10

Solutions to Sorting and Searching

10.1 Sorted Merge: You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

pg 149

SOLUTION

Since we know that A has enough buffer at the end, we won't need to allocate additional space. Our logic should involve simply comparing elements of A and B and inserting them in order, until we've exhausted all elements in A and in B.

The only issue with this is that if we insert an element into the front of A, then we'll have to shift the existing elements backwards to make room for it. It's better to insert elements into the back of the array, where there's empty space.

The code below does just that. It works from the back of A and B, moving the largest elements to the back of A.

```
1  void merge(int[] a, int[] b, int lastA, int lastB) {
2     int indexA = lastA - 1; /* Index of last element in array a */
3     int indexB = lastB - 1; /* Index of last element in array b */
4     int indexMerged = lastB + lastA - 1; /* end of merged array */
5
6     /* Merge a and b, starting from the last element in each */
7     while (indexB >= 0) {
8        /* end of a is > than end of b */
9        if (indexA >= 0 && a[indexA] > b[indexB]) {
10          a[indexMerged] = a[indexA]; // copy element
11          indexA--;
12       } else {
13          a[indexMerged] = b[indexB]; // copy element
14          indexB--;
15       }
16       indexMerged--; // move indices
17    }
18 }
```

Note that you don't need to copy the contents of A after running out of elements in B. They are already in place.

10.2 Group Anagrams: Write a method to sort an array of strings so that all the anagrams are next to each other.

pg 150

SOLUTION

This problem asks us to group the strings in an array such that the anagrams appear next to each other. Note that no specific ordering of the words is required, other than this.

We need a quick and easy way of determining if two strings are anagrams of each other. What defines if two words are anagrams of each other? Well, anagrams are words that have the same characters but in different orders. It follows then that if we can put the characters in the same order, we can easily check if the new words are identical.

One way to do this is to just apply any standard sorting algorithm, like merge sort or quick sort, and modify the comparator. This comparator will be used to indicate that two strings which are anagrams of each other are equivalent.

What's the easiest way of checking if two words are anagrams? We could count the occurrences of the distinct characters in each string and return `true` if they match. Or, we could just sort the string. After all, two words which are anagrams will look the same once they're sorted.

The code below implements the comparator.

```
1  class AnagramComparator implements Comparator<String> {
2    public String sortChars(String s) {
3      char[] content = s.toCharArray();
4      Arrays.sort(content);
5      return new String(content);
6    }
7
8    public int compare(String s1, String s2) {
9      return sortChars(s1).compareTo(sortChars(s2));
10   }
11 }
```

Now, just sort the arrays using this `compareTo` method instead of the usual one.

```
12 Arrays.sort(array, new AnagramComparator());
```

This algorithm will take `O(n log(n))` time.

This may be the best we can do for a general sorting algorithm, but we don't actually need to fully sort the array. We only need to *group* the strings in the array by anagram.

We can do this by using a hash table which maps from the sorted version of a word to a list of its anagrams. So, for example, `acre` will map to the list `{acre, race, care}`. Once we've grouped all the words into these lists by anagram, we can then put them back into the array.

The code below implements this algorithm.

```
1  void sort(String[] array) {
2    HashMapList<String, String> mapList = new HashMapList<String, String>();
3
4    /* Group words by anagram */
5    for (String s : array) {
6      String key = sortChars(s);
7      mapList.put(key, s);
8    }
```

```
9
10    /* Convert hash table to array */
11    int index = 0;
12    for (String key : mapList.keySet()) {
13       ArrayList<String> list = mapList.get(key);
14       for (String t : list) {
15          array[index] = t;
16          index++;
17       }
18    }
19 }
20
21 String sortChars(String s) {
22    char[] content = s.toCharArray();
23    Arrays.sort(content);
24    return new String(content);
25 }
26
27 /* HashMapList<String, Integer> is a HashMap that maps from Strings to
28  * ArrayList<Integer>. See appendix for implementation. */
```

You may notice that the algorithm above is a modification of bucket sort.

10.3 Search in Rotated Array: Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE

Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

pg 150

SOLUTION

If this problem smells like binary search to you, you're right!

In classic binary search, we compare x with the midpoint to figure out if x belongs on the left or the right side. The complication here is that the array is rotated and may have an inflection point. Consider, for example, the following two arrays:

```
Array1: {10, 15, 20,  0,  5}
Array2: {50,  5, 20, 30, 40}
```

Note that both arrays have a midpoint of 20, but 5 appears on the left side of one and on the right side of the other. Therefore, comparing x with the midpoint is insufficient.

However, if we look a bit deeper, we can see that one half of the array must be ordered normally (in increasing order). We can therefore look at the normally ordered half to determine whether we should search the left or right half.

For example, if we are searching for 5 in `Array1`, we can look at the left element (10) and middle element (20). Since 10 < 20, the left half must be ordered normally. And, since 5 is not between those, we know that we must search the right half.

In `Array2`, we can see that since 50 > 20, the right half must be ordered normally. We turn to the middle (20) and right (40) element to check if 5 would fall between them. The value 5 would not; therefore, we search the left half.

The tricky condition is if the left and the middle are identical, as in the example array `{2, 2, 2, 3, 4, 2}`. In this case, we can check if the rightmost element is different. If it is, we can search just the right side. Otherwise, we have no choice but to search both halves.

```
1  int search(int a[], int left, int right, int x) {
2    int mid = (left + right) / 2;
3    if (x == a[mid]) { // Found element
4      return mid;
5    }
6    if (right < left) {
7      return -1;
8    }
9
10   /* Either the left or right half must be normally ordered. Find out which side
11    * is normally ordered, and then use the normally ordered half to figure out
12    * which side to search to find x. */
13   if (a[left] < a[mid]) { // Left is normally ordered.
14     if (x >= a[left] && x < a[mid]) {
15       return search(a, left, mid - 1, x); // Search left
16     } else {
17       return search(a, mid + 1, right, x); // Search right
18     }
19   } else if (a[mid] < a[left]) { // Right is normally ordered.
20     if (x > a[mid] && x <= a[right]) {
21       return search(a, mid + 1, right, x); // Search right
22     } else {
23       return search(a, left, mid - 1, x); // Search left
24     }
25   } else if (a[left] == a[mid]) { // Left or right half is all repeats
26     if (a[mid] != a[right]) { // If right is different, search it
27       return search(a, mid + 1, right, x); // search right
28     } else { // Else, we have to search both halves
29       int result = search(a, left, mid - 1, x); // Search left
30       if (result == -1) {
31         return search(a, mid + 1, right, x); // Search right
32       } else {
33         return result;
34       }
35     }
36   }
37   return -1;
38 }
```

This code will run in `O(log n)` if all the elements are unique. However, with many duplicates, the algorithm is actually `O(n)`. This is because with many duplicates, we will often have to search both the left and right sides of the array (or subarrays).

Note that while this problem is not conceptually very complex, it is actually very difficult to implement flawlessly. Don't feel bad if you had trouble implementing it without a few bugs. Because of the ease of making off-by-one and other minor errors, you should make sure to test your code very thoroughly.

10.4 Sorted Search, No Size: You are given an array-like data structure `Listy` which lacks a size method. It does, however, have an `elementAt(i)` method that returns the element at index `i` in `O(1)` time. If `i` is beyond the bounds of the data structure, it returns `-1`. (For this reason, the data structure only supports positive integers.) Given a `Listy` which contains sorted, positive integers, find the index at which an element x occurs. If x occurs multiple times, you may return any index.

pg 150

SOLUTION

Our first thought here should be binary search. The problem is that binary search requires us knowing the length of the list, so that we can compare it to the midpoint. We don't have that here.

Could we compute the length? Yes!

We know that `elementAt` will return `-1` when `i` is too large. We can therefore just try bigger and bigger values until we exceed the size of the list.

But how much bigger? If we just went through the list linearly—1, then 2, then 3, then 4, and so on—we'd wind up with a linear time algorithm. We probably want something faster than this. Otherwise, why would the interviewer have specified the list is sorted?

It's better to back off exponentially. Try 1, then 2, then 4, then 8, then 16, and so on. This ensures that, if the list has length n, we'll find the length in at most `O(log n)` time.

> Why `O(log n)`? Imagine we start with pointer q at `q = 1`. At each iteration, this pointer q doubles, until q is bigger than the length n. How many times can q double in size before it's bigger than n? Or, in other words, for what value of k does $2^k = n$? This expression is equal when `k = log n`, as this is precisely what `log` means. Therefore, it will take `O(log n)` steps to find the length.

Once we find the length, we just perform a (mostly) normal binary search. I say "mostly" because we need to make one small tweak. If the mid point is -1, we need to treat this as a "too big" value and search left. This is on line 16 below.

There's one more little tweak. Recall that the way we figure out the length is by calling `elementAt` and comparing it to -1. If, in the process, the element is bigger than the value x (the one we're searching for), we'll jump over to the binary search part early.

```
1   int search(Listy list, int value) {
2     int index = 1;
3     while (list.elementAt(index) != -1 && list.elementAt(index) < value) {
4       index *= 2;
5     }
6     return binarySearch(list, value, index / 2, index);
7   }
8
9   int binarySearch(Listy list, int value, int low, int high) {
10    int mid;
11
12    while (low <= high) {
13      mid = (low + high) / 2;
14      int middle = list.elementAt(mid);
15      if (middle > value || middle == -1) {
16        high = mid - 1;
17      } else if (middle < value) {
```

```
18          low = mid + 1;
19        } else {
20          return mid;
21        }
22      }
23      return -1;
24    }
```

It turns out that not knowing the length didn't impact the runtime of the search algorithm. We find the length in `O(log n)` time and then do the search in `O(log n)` time. Our overall runtime is `O(log n)`, just as it would be in a normal array.

10.5 Sparse Search: Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

Input: `ball`, {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}

Output: 4

pg 150

SOLUTION

If it weren't for the empty strings, we could simply use binary search. We would compare the string to be found, `str`, with the midpoint of the array, and go from there.

With empty strings interspersed, we can implement a simple modification of binary search. All we need to do is fix the comparison against `mid`, in case `mid` is an empty string. We simply move `mid` to the closest non-empty string.

The recursive code below to solve this problem can easily be modified to be iterative. We provide such an implementation in the code attachment.

```
1   int search(String[] strings, String str, int first, int last) {
2     if (first > last) return -1;
3     /* Move mid to the middle */
4     int mid = (last + first) / 2;
5
6     /* If mid is empty, find closest non-empty string. */
7     if (strings[mid].isEmpty()) {
8       int left = mid - 1;
9       int right = mid + 1;
10      while (true) {
11        if (left < first && right > last) {
12          return -1;
13        } else if (right <= last && !strings[right].isEmpty()) {
14          mid = right;
15          break;
16        } else if (left >= first && !strings[left].isEmpty()) {
17          mid = left;
18          break;
19        }
20        right++;
21        left--;
22      }
23    }
```

```
24
25    /* Check for string, and recurse if necessary */
26    if (str.equals(strings[mid])) { // Found it!
27       return mid;
28    } else if (strings[mid].compareTo(str) < 0) { // Search right
29       return search(strings, str, mid + 1, last);
30    } else { // Search left
31       return search(strings, str, first, mid - 1);
32    }
33 }
34
35 int search(String[] strings, String str) {
36    if (strings == null || str == null || str == "") {
37       return -1;
38    }
39    return search(strings, str, 0, strings.length - 1);
40 }
```

The worst-case runtime for this algorithm is O(n). In fact, it's impossible to have an algorithm for this problem that is better than O(n) in the worst case. After all, you could have an array of all empty strings except for one non-empty string. There is no "smart" way to find this non-empty string. In the worst case, you will need to look at every element in the array.

Careful consideration should be given to the situation when someone searches for the empty string. Should we find the location (which is an O(n) operation)? Or should we handle this as an error?

There's no correct answer here. This is an issue you should raise with your interviewer. Simply asking this question will demonstrate that you are a careful coder.

10.6 Sort Big File: Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

pg 150

SOLUTION

When an interviewer gives a size limit of 20 gigabytes, it should tell you something. In this case, it suggests that they don't want you to bring all the data into memory.

So what do we do? We only bring part of the data into memory.

We'll divide the file into chunks, which are x megabytes each, where x is the amount of memory we have available. Each chunk is sorted separately and then saved back to the file system.

Once all the chunks are sorted, we merge the chunks, one by one. At the end, we have a fully sorted file.

This algorithm is known as external sort.

10.7 Missing Int: Given an input file with four billion non-negative integers, provide an algorithm to generate an integer that is not contained in the file. Assume you have 1 GB of memory available for this task.

FOLLOW UP

What if you have only 10 MB of memory? Assume that all the values are distinct and we now have no more than one billion non-negative integers.

pg 150

SOLUTION

There are a total of 2^{32}, or 4 billion, distinct integers possible and 2^{31} non-negative integers. Therefore, we know the input file (assuming it is `int`s rather than `long`s) contains some duplicates.

We have 1 GB of memory, or 8 billion bits. Thus, with 8 billion bits, we can map all possible integers to a distinct bit with the available memory. The logic is as follows:

1. Create a bit vector (BV) with 4 billion bits. Recall that a bit vector is an array that compactly stores boolean values by using an array of ints (or another data type). Each int represents 32 boolean values.
2. Initialize BV with all 0s.
3. Scan all numbers (num) from the file and call `BV.set(num, 1)`.
4. Now scan again BV from the 0th index.
5. Return the first index which has a value of 0.

The following code demonstrates our algorithm.

```
long numberOfInts = ((long) Integer.MAX_VALUE) + 1;
byte[] bitfield = new byte [(int) (numberOfInts / 8)];
String filename = ...

void findOpenNumber() throws FileNotFoundException {
   Scanner in = new Scanner(new FileReader(filename));
   while (in.hasNextInt()) {
      int n = in.nextInt ();
      /* Finds the corresponding number in the bitfield by using the OR operator to
       * set the nth bit of a byte (e.g., 10 would correspond to the 2nd bit of
       * index 2 in the byte array). */
      bitfield [n / 8] |= 1 << (n % 8);
   }

   for (int i = 0; i < bitfield.length; i++) {
      for (int j = 0; j < 8; j++) {
         /* Retrieves the individual bits of each byte. When 0 bit is found, print
          * the corresponding value. */
         if ((bitfield[i] & (1 << j)) == 0) {
            System.out.println (i * 8 + j);
            return;
         }
      }
   }
}
```

Follow Up: What if we have only 10 MB memory?

It's possible to find a missing integer with two passes of the data set. We can divide up the integers into blocks of some size (we'll discuss how to decide on a size later). Let's just assume that we divide up the integers into blocks of 1000. So, block 0 represents the numbers 0 through 999, block 1 represents numbers 1000 - 1999, and so on.

Since all the values are distinct, we know how many values we *should* find in each block. So, we search through the file and count how many values are between 0 and 999, how many are between 1000 and 1999, and so on. If we count only 999 values in a particular range, then we know that a missing int must be in that range.

In the second pass, we'll actually look for which number in that range is missing. We use the bit vector approach from the first part of this problem. We can ignore any number outside of this specific range.

The question, now, is what is the appropriate block size? Let's define some variables as follows:

- Let `rangeSize` be the size of the ranges that each block in the first pass represents.
- Let `arraySize` represent the number of blocks in the first pass. Note that `arraySize` = $2^{31}/\texttt{rangeSize}$ since there are 2^{31} non-negative integers.

We need to select a value for `rangeSize` such that the memory from the first pass (the array) and the second pass (the bit vector) fit.

First Pass: The Array

The array in the first pass can fit in 10 megabytes, or roughly 2^{23} bytes, of memory. Since each element in the array is an int, and an int is 4 bytes, we can hold an array of at most about 2^{21} elements. So, we can deduce the following:

$$\texttt{arraySize} = \frac{2^{31}}{\texttt{rangeSize}} \leq 2^{21}$$
$$\texttt{rangeSize} \geq \frac{2^{31}}{2^{21}}$$
$$\texttt{rangeSize} \geq 2^{10}$$

Second Pass: The Bit Vector

We need to have enough space to store `rangeSize` bits. Since we can fit 2^{23} bytes in memory, we can fit 2^{26} bits in memory. Therefore, we can conclude the following:

$$2^{11} <= \texttt{rangeSize} <= 2^{26}$$

These conditions give us a good amount of "wiggle room," but the nearer to the middle that we pick, the less memory will be used at any given time.

The below code provides one implementation for this algorithm.

```
1  int findOpenNumber(String filename) throws FileNotFoundException {
2    int rangeSize = (1 << 20); // 2^20 bits (2^17 bytes)
3
4    /* Get count of number of values within each block. */
5    int[] blocks = getCountPerBlock(filename, rangeSize);
6
7    /* Find a block with a missing value. */
8    int blockIndex = findBlockWithMissing(blocks, rangeSize);
9    if (blockIndex < 0) return -1;
```

```
10
11    /* Create bit vector for items within this range. */
12    byte[] bitVector = getBitVectorForRange(filename, blockIndex, rangeSize);
13
14    /* Find a zero in the bit vector */
15    int offset = findZero(bitVector);
16    if (offset < 0) return -1;
17
18    /* Compute missing value. */
19    return blockIndex * rangeSize + offset;
20 }
21
22 /* Get count of items within each range. */
23 int[] getCountPerBlock(String filename, int rangeSize)
24       throws FileNotFoundException {
25    int arraySize = Integer.MAX_VALUE / rangeSize + 1;
26    int[] blocks = new int[arraySize];
27
28    Scanner in = new Scanner (new FileReader(filename));
29    while (in.hasNextInt()) {
30        int value = in.nextInt();
31        blocks[value / rangeSize]++;
32    }
33    in.close();
34    return blocks;
35 }
36
37 /* Find a block whose count is low. */
38 int findBlockWithMissing(int[] blocks, int rangeSize) {
39    for (int i = 0; i < blocks.length; i++) {
40       if (blocks[i] < rangeSize){
41          return i;
42       }
43    }
44    return -1;
45 }
46
47 /* Create a bit vector for the values within a specific range. */
48 byte[] getBitVectorForRange(String filename, int blockIndex, int rangeSize)
49       throws FileNotFoundException {
50    int startRange = blockIndex * rangeSize;
51    int endRange = startRange + rangeSize;
52    byte[] bitVector = new byte[rangeSize/Byte.SIZE];
53
54    Scanner in = new Scanner(new FileReader(filename));
55    while (in.hasNextInt()) {
56       int value = in.nextInt();
57       /* If the number is inside the block that's missing numbers, we record it */
58       if (startRange <= value && value < endRange) {
59          int offset = value - startRange;
60          int mask = (1 << (offset % Byte.SIZE));
61          bitVector[offset / Byte.SIZE] |= mask;
62       }
63    }
64    in.close();
65    return bitVector;
```

```
66 }
67
68 /* Find bit index that is 0 within byte. */
69 int findZero(byte b) {
70    for (int i = 0; i < Byte.SIZE; i++) {
71       int mask = 1 << i;
72       if ((b & mask) == 0) {
73          return i;
74       }
75    }
76    return -1;
77 }
78
79 /* Find a zero within the bit vector and return the index. */
80 int findZero(byte[] bitVector) {
81    for (int i = 0; i < bitVector.length; i++) {
82       if (bitVector[i] != ~0) { // If not all 1s
83          int bitIndex = findZero(bitVector[i]);
84          return i * Byte.SIZE + bitIndex;
85       }
86    }
87    return -1;
88 }
```

What if, as a follow up question, you are asked to solve the problem with even less memory? In this case, we can do repeated passes using the approach from the first step. We'd first check to see how many integers are found within each sequence of a million elements. Then, in the second pass, we'd check how many integers are found in each sequence of a thousand elements. Finally, in the third pass, we'd apply the bit vector.

10.8 Find Duplicates: You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

pg 151

SOLUTION

We have 4 kilobytes of memory which means we can address up to $8 * 4 * 2^{10}$ bits. Note that $32 * 2^{10}$ bits is greater than 32000. We can create a bit vector with 32000 bits, where each bit represents one integer.

Using this bit vector, we can then iterate through the array, flagging each element v by setting bit v to 1. When we come across a duplicate element, we print it.

```
1  void checkDuplicates(int[] array) {
2     BitSet bs = new BitSet(32000);
3     for (int i = 0; i < array.length; i++) {
4        int num = array[i];
5        int num0 = num - 1; // bitset starts at 0, numbers start at 1
6        if (bs.get(num0)) {
7           System.out.println(num);
8        } else {
9           bs.set(num0);
10       }
11    }
12 }
13
14 class BitSet {
```

```
15    int[] bitset;
16
17    public BitSet(int size) {
18       bitset = new int[(size >> 5) + 1]; // divide by 32
19    }
20
21    boolean get(int pos) {
22       int wordNumber = (pos >> 5); // divide by 32
23       int bitNumber = (pos & 0x1F); // mod 32
24       return (bitset[wordNumber] & (1 << bitNumber)) != 0;
25    }
26
27    void set(int pos) {
28       int wordNumber = (pos >> 5); // divide by 32
29       int bitNumber = (pos & 0x1F); // mod 32
30       bitset[wordNumber] |= 1 << bitNumber;
31    }
32 }
```

Note that while this isn't an especially difficult problem, it's important to implement this cleanly. This is why we defined our own bit vector class to hold a large bit vector. If our interviewer lets us (she may or may not), we could have of course used Java's built in `BitSet` class.

10.9 Sorted Matrix Search: Given an `M x N` matrix in which each row and each column is sorted in ascending order, write a method to find an element.

pg 151

SOLUTION

We can approach this in two ways: a more naive solution that only takes advantage of part of the sorting, and a more optimal way that takes advantage of both parts of the sorting.

Solution #1: Naive Solution

As a first approach, we can do binary search on every row to find the element. This algorithm will be `O(M log(N))`, since there are `M` rows and it takes `O(log(N))` time to search each one. This is a good approach to mention to your interviewer before you proceed with generating a better algorithm.

To develop an algorithm, let's start with a simple example.

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

Suppose we are searching for the element 55. How can we identify where it is?

If we look at the start of a row or the start of a column, we can start to deduce the location. If the start of a column is greater than 55, we know that 55 can't be in that column, since the start of the column is always the minimum element. Additionally, we know that 55 can't be in any columns on the right, since the first element of each column must increase in size from left to right. Therefore, if the start of the column is greater than the element `x` that we are searching for, we know that we need to move further to the left.

For rows, we use identical logic. If the start of a row is bigger than `x`, we know we need to move upwards.

Observe that we can also make a similar conclusion by looking at the ends of columns or rows. If the end of a column or row is less than x, then we know that we must move down (for rows) or to the right (for columns) to find x. This is because the end is always the maximum element.

We can bring these observations together into a solution. The observations are the following:

- If the start of a column is greater than x, then x is to the left of the column.
- If the end of a column is less than x, then x is to the right of the column.
- If the start of a row is greater than x, then x is above that row.
- If the end of a row is less than x, then x is below that row.

We can begin in any number of places, but let's begin with looking at the starts of columns.

We need to start with the greatest column and work our way to the left. This means that our first element for comparison is `array[0][c-1]`, where c is the number of columns. By comparing the start of columns to x (which is 55), we'll find that x must be in columns 0, 1, or 2. We will have stopped at `array[0][2]`.

This element may not be the end of a row in the full matrix, but it is an end of a row of a submatrix. The same conditions apply. The value at `array[0][2]`, which is 40, is less than 55, so we know we can move downwards.

We now have a submatrix to consider that looks like the following (the gray squares have been eliminated).

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

We can repeatedly apply these conditions to search for 55. Note that the only conditions we actually use are conditions 1 and 4.

The code below implements this elimination algorithm.

```
1   boolean findElement(int[][] matrix, int elem) {
2     int row = 0;
3     int col = matrix[0].length - 1;
4     while (row < matrix.length && col >= 0) {
5       if (matrix[row][col] == elem) {
6         return true;
7       } else if (matrix[row][col] > elem) {
8         col--;
9       } else {
10        row++;
11      }
12    }
13    return false;
14  }
```

Alternatively, we can apply a solution that more directly looks like binary search. The code is considerably more complicated, but it applies many of the same learnings.

Solution #2: Binary Search

Let's again look at a simple example.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

We want to be able to leverage the sorting property to more efficiently find an element. So, we might ask ourselves, what does the unique ordering property of this matrix imply about where an element might be located?

We are told that every row and column is sorted. This means that element `a[i][j]` will be greater than the elements in row `i` between columns 0 and `j - 1` and the elements in column `j` between rows 0 and `i - 1`.

Or, in other words:

```
a[i][0] <= a[i][1] <= ... <= a[i][j-1] <= a[i][j]
a[0][j] <= a[1][j] <= ... <= a[i-1][j] <= a[i][j]
```

Looking at this visually, the dark gray element below is bigger than all the light gray elements.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

The light gray elements also have an ordering to them: each is bigger than the elements to the left of it, as well as the elements above it. So, by transitivity, the dark gray element is bigger than the entire square.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

This means that for any rectangle we draw in the matrix, the bottom right hand corner will always be the biggest.

Likewise, the top left hand corner will always be the smallest. The colors below indicate what we know about the ordering of elements (light gray < dark gray < black):

15	20	70	85
20	35	80	95
30	55	95	105
40	80	120	120

Let's return to the original problem: suppose we were searching for the value 85. If we look along the diagonal, we'll find the elements 35 and 95. What does this tell us about the location of 85?

15	20	70	85
25	35	80	95
30	55	95	105
40	80	120	120

85 can't be in the black area, since 95 is in the upper left hand corner and is therefore the smallest element in that square.

85 can't be in the light gray area either, since 35 is in the lower right hand corner of that square.

85 must be in one of the two white areas.

So, we partition our grid into four quadrants and recursively search the lower left quadrant and the upper right quadrant. These, too, will get divided into quadrants and searched.

Observe that since the diagonal is sorted, we can efficiently search it using binary search.

The code below implements this algorithm.

```
1  Coordinate findElement(int[][] matrix, Coordinate origin, Coordinate dest, int x){
2    if (!origin.inbounds(matrix) || !dest.inbounds(matrix)) {
3      return null;
4    }
5    if (matrix[origin.row][origin.column] == x) {
6      return origin;
7    } else if (!origin.isBefore(dest)) {
8      return null;
9    }
10
11   /* Set start to start of diagonal and end to the end of the diagonal. Since the
12    * grid may not be square, the end of the diagonal may not equal dest. */
13   Coordinate start = (Coordinate) origin.clone();
14   int diagDist = Math.min(dest.row - origin.row, dest.column - origin.column);
15   Coordinate end = new Coordinate(start.row + diagDist, start.column + diagDist);
16   Coordinate p = new Coordinate(0, 0);
17
18   /* Do binary search on the diagonal, looking for the first element > x */
19   while (start.isBefore(end)) {
20     p.setToAverage(start, end);
21     if (x > matrix[p.row][p.column]) {
22       start.row = p.row + 1;
23       start.column = p.column + 1;
24     } else {
25       end.row = p.row - 1;
26       end.column = p.column - 1;
27     }
28   }
29
30   /* Split the grid into quadrants. Search the bottom left and the top right. */
31   return partitionAndSearch(matrix, origin, dest, start, x);
32 }
33
34 Coordinate partitionAndSearch(int[][] matrix, Coordinate origin, Coordinate dest,
35                               Coordinate pivot, int x) {
36   Coordinate lowerLeftOrigin = new Coordinate(pivot.row, origin.column);
37   Coordinate lowerLeftDest = new Coordinate(dest.row, pivot.column - 1);
38   Coordinate upperRightOrigin = new Coordinate(origin.row, pivot.column);
39   Coordinate upperRightDest = new Coordinate(pivot.row - 1, dest.column);
40
41   Coordinate lowerLeft = findElement(matrix, lowerLeftOrigin, lowerLeftDest, x);
42   if (lowerLeft == null) {
43     return findElement(matrix, upperRightOrigin, upperRightDest, x);
44   }
```

```
45    return lowerLeft;
46  }
47
48  Coordinate findElement(int[][] matrix, int x) {
49    Coordinate origin = new Coordinate(0, 0);
50    Coordinate dest = new Coordinate(matrix.length - 1, matrix[0].length - 1);
51    return findElement(matrix, origin, dest, x);
52  }
53
54  public class Coordinate implements Cloneable {
55    public int row, column;
56    public Coordinate(int r, int c) {
57      row = r;
58      column = c;
59    }
60
61    public boolean inbounds(int[][] matrix) {
62      return  row >= 0 && column >= 0 &&
63              row < matrix.length && column < matrix[0].length;
64    }
65
66    public boolean isBefore(Coordinate p) {
67      return row <= p.row && column <= p.column;
68    }
69
70    public Object clone() {
71      return new Coordinate(row, column);
72    }
73
74    public void setToAverage(Coordinate min, Coordinate max) {
75      row = (min.row + max.row) / 2;
76      column = (min.column + max.column) / 2;
77    }
78  }
```

If you read all this code and thought, "there's no way I could do all this in an interview!" you're probably right. You couldn't. But, your performance on any problem is evaluated compared to other candidates on the same problem. So while you couldn't implement all this, neither could they. You are at no disadvantage when you get a tricky problem like this.

You help yourself out a bit by separating code out into other methods. For example, by pulling `partitionAndSearch` out into its own method, you will have an easier time outlining key aspects of the code. You can then come back to fill in the body for `partitionAndSearch` if you have time.

10.10 Rank from Stream: Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

EXAMPLE

Stream (in order of appearance): `5, 1, 4, 4, 5, 9, 7, 13, 3`

```
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1
getRankOfNumber(4) = 3
```

pg 151

SOLUTION

A relatively easy way to implement this would be to have an array that holds all the elements in sorted order. When a new element comes in, we would need to shift the other elements to make room. Implementing `getRankOfNumber` would be quite efficient, though. We would simply perform a binary search for n, and return the index.

However, this is very inefficient for inserting elements (that is, the `track(int x)` function). We need a data structure which is good at keeping relative ordering, as well as updating when we insert new elements. A binary search tree can do just that.

Instead of inserting elements into an array, we insert elements into a binary search tree. The method `track(int x)` will run in `O(log n)` time, where n is the size of the tree (provided, of course, that the tree is balanced).

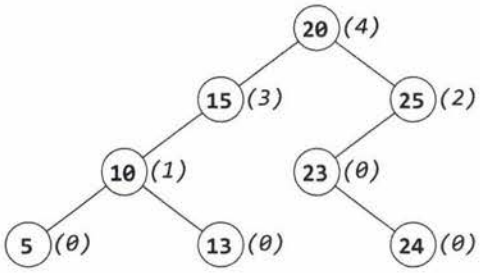
To find the rank of a number, we could do an in-order traversal, keeping a counter as we traverse. The goal is that, by the time we find x, `counter` will equal the number of elements less than x.

As long as we're moving left during searching for x, the counter won't change. Why? Because all the values we're skipping on the right side are greater than x. After all, the very smallest element (with rank of 1) is the leftmost node.

When we move to the right though, we skip over a bunch of elements on the left. All of these elements are less than x, so we'll need to increment `counter` by the number of elements in the left subtree.

Rather than counting the size of the left subtree (which would be inefficient), we can track this information as we add new elements to the tree.

Let's walk through an example on the following tree. In the below example, the value in parentheses indicates the number of nodes in the left subtree (or, in other words, the rank of the node *relative* to its subtree).



Suppose we want to find the rank of 24 in the tree above. We would compare 24 with the root, 20, and find that 24 must reside on the right. The root has 4 nodes in its left subtree, and when we include the root itself, this gives us five total nodes smaller than 24. We set `counter` to 5.

Then, we compare 24 with node 25 and find that 24 must be on the left. The value of `counter` does not update, since we're not "passing over" any smaller nodes. The value of `counter` is still 5.

Next, we compare 24 with node 23, and find that 24 must be on the right. `Counter` gets incremented by just 1 (to 6), since 23 has no left nodes.

Finally, we find 24 and we return `counter`: 6.

Recursively, the algorithm is the following:

```
int getRank(Node node, int x) {
  if x is node.data, return node.leftSize()
  if x is on left of node, return getRank(node.left, x)
  if x is on right of node, return node.leftSize() + 1 + getRank(node.right, x)
}
```

The full code for this is below.

```
RankNode root = null;

void track(int number) {
  if (root == null) {
    root = new RankNode(number);
  } else {
    root.insert(number);
  }
}

int getRankOfNumber(int number) {
  return root.getRank(number);
}


public class RankNode {
  public int left_size = 0;
  public RankNode left, right;
  public int data = 0;
  public RankNode(int d) {
    data = d;
  }

  public void insert(int d) {
    if (d <= data) {
```

```
26          if (left != null) left.insert(d);
27          else left = new RankNode(d);
28          left_size++;
29       } else {
30          if (right != null) right.insert(d);
31          else right = new RankNode(d);
32       }
33    }
34
35    public int getRank(int d) {
36       if (d == data) {
37          return left_size;
38       } else if (d < data) {
39          if (left == null) return -1;
40          else return left.getRank(d);
41       } else {
42          int right_rank = right == null ? -1 : right.getRank(d);
43          if (right_rank == -1) return -1;
44          else return left_size + 1 + right_rank;
45       }
46    }
47 }
```

The `track` method and the `getRankOfNumber` method will both operate in `O(log N)` on a balanced tree and `O(N)` on an unbalanced tree.

Note how we've handled the case in which d is not found in the tree. We check for the -1 return value, and, when we find it, return -1 up the tree. It is important that you handle cases like this.

10.11 Peaks and Valleys: In an array of integers, a "peak" is an element which is greater than or equal to the adjacent integers and a "valley" is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

pg 151

SOLUTION

Since this problem asks us to sort the array in a particular way, one thing we can try is doing a normal sort and then "fixing" the array into an alternating sequence of peaks and valleys.

Suboptimal Solution

Imagine we were given an unsorted array and then sort it to become the following:

```
0  1  4  7  8  9
```

We now have an ascending list of integers.

How can we rearrange this into a proper alternating sequence of peaks and valleys? Let's walk through it and try to do that.

- The `0` is okay.

- The 1 is in the wrong place. We can swap it with either the 0 or 4. Let's swap it with the 0.
  ```
  1  0  4  7  8  9
  ```
- The 4 is okay.
- The 7 is in the wrong place. We can swap it with either the 4 or the 8. Let's swap it with the 4.
  ```
  1  0  7  4  8  9
  ```
- The 9 is in the wrong place. Let's swap it with the 8.
  ```
  1  0  7  4  9  8
  ```

Observe that there's nothing special about the array having these values. The relative order of the elements matters, but all sorted arrays will have the same relative order. Therefore, we can take this same approach on any sorted array.

Before coding, we should clarify the exact algorithm, though.

1. Sort the array in ascending order.
2. Iterate through the elements, starting from index 1 (not 0) and jumping two elements at a time.
3. At each element, swap it with the previous element. Since every three elements appear in the order `small <= medium <= large`, swapping these elements will always put medium as a peak: `medium <= small <= large`.

This approach will ensure that the peaks are in the right place: indexes 1, 3, 5, and so on. As long as the odd-numbered elements (the peaks) are bigger than the adjacent elements, then the even-numbered elements (the valleys) must be smaller than the adjacent elements.

The code to implement this is below.

```
1  void sortValleyPeak(int[] array) {
2    Arrays.sort(array);
3    for (int i = 1; i < array.length; i += 2) {
4      swap(array, i - 1, i);
5    }
6  }
7
8  void swap(int[] array, int left, int right) {
9    int temp = array[left];
10   array[left] = array[right];
11   array[right] = temp;
12 }
```

This algorithm runs in `O(n log n)` time.

Optimal Solution

To optimize past the prior solution, we need to cut out the sorting step. The algorithm must operate on an unsorted array.

Let's revisit an example.

```
9  1  0  4  8  7
```

For each element, we'll look at the adjacent elements. Let's imagine some sequences. We'll just use the numbers 0, 1 and 2. The specific values don't matter.

```
0  1  2
0  2  1       // peak
1  0  2
1  2  0       // peak
2  1  0
```

```
2   0   1
```

If the center element needs to be a peak, then two of those sequences work. Can we fix the other ones to make the center element a peak?

Yes. We can fix this sequence by swapping the center element with the largest adjacent element.

```
0   1   2   -> 0   2   1
0   2   1       // peak
1   0   2   -> 1   2   0
1   2   0   // peak
2   1   0   -> 1   2   0
2   0   1   -> 0   2   1
```

As we noted before, if we make sure the peaks are in the right place then we know the valleys are in the right place.

> We should be a little cautious here. Is it possible that one of these swaps could "break" an earlier part of the sequence that we'd already processed? This is a good thing to worry about, but it's not an issue here. If we're swapping `middle` with `left`, then `left` is currently a valley. `Middle` is smaller than `left`, so we're putting an even smaller element as a valley. Nothing will break. All is good!

The code to implement this is below.

```
1  void sortValleyPeak(int[] array) {
2    for (int i = 1; i < array.length; i += 2) {
3      int biggestIndex = maxIndex(array, i - 1, i, i + 1);
4      if (i != biggestIndex) {
5        swap(array, i, biggestIndex);
6      }
7    }
8  }
9
10 int maxIndex(int[] array, int a, int b, int c) {
11   int len = array.length;
12   int aValue = a >= 0 && a < len ? array[a] : Integer.MIN_VALUE;
13   int bValue = b >= 0 && b < len ? array[b] : Integer.MIN_VALUE;
14   int cValue = c >= 0 && c < len ? array[c] : Integer.MIN_VALUE;
15
16   int max = Math.max(aValue, Math.max(bValue, cValue));
17   if (aValue == max) return a;
18   else if (bValue == max) return b;
19   else return c;
20 }
```

This algorithm takes O(n) time.

11

Solutions to Testing

11.1 Mistake: Find the mistake(s) in the following code:

```
unsigned int i;
for (i = 100; i >= 0; --i)
  printf("%d\n", i);
```

pg 157

SOLUTION

There are two mistakes in this code.

First, note that an `unsigned int` is, by definition, always greater than or equal to zero. The for loop condition will therefore always be true, and it will loop infinitely.

The correct code to print all numbers from 100 to 1, is `i > 0`. If we truly wanted to print zero, we could add an additional `printf` statement after the for loop.

```
1  unsigned int i;
2  for (i = 100; i > 0; --i)
3    printf("%d\n", i);
```

One additional correction is to use %u in place of %d, as we are printing `unsigned int`.

```
1  unsigned int i;
2  for (i = 100; i > 0; --i)
3    printf("%u\n", i);
```

This code will now correctly print the list of all numbers from 100 to 1, in descending order.

11.2 Random Crashes: You are given the source to an application which crashes when it is run. After running it ten times in a debugger, you find it never crashes in the same place. The application is single threaded, and uses only the C standard library. What programming errors could be causing this crash? How would you test each one?

pg 157

SOLUTION

The question largely depends on the type of application being diagnosed. However, we can give some general causes of random crashes.

1. *"Random Variable:"* The application may use some random number or variable component that may not be fixed for every execution of the program. Examples include user input, a random number generated by the program, or the time of day.

2. *Uninitialized Variable:* The application could have an uninitialized variable which, in some languages, may cause it to take on an arbitrary value. The values of this variable could result in the code taking a slightly different path each time.
3. *Memory Leak:* The program may have run out of memory. Other culprits are totally random for each run since it depends on the number of processes running at that particular time. This also includes heap overflow or corruption of data on the stack.
4. *External Dependencies:* The program may depend on another application, machine, or resource. If there are multiple dependencies, the program could crash at any point.

To track down the issue, we should start with learning as much as possible about the application. Who is running it? What are they doing with it? What kind of application is it?

Additionally, although the application doesn't crash in exactly the same place, it's possible that it is linked to specific components or scenarios. For example, it could be that the application never crashes if it's simply launched and left untouched, and that crashes only appear at some point after loading a file. Or, it may be that all the crashes take place within the lower level components, such as file I/O.

It may be useful to approach this by elimination. Close down all other applications on the system. Track resource use very carefully. If there are parts of the program we can disable, do so. Run it on a different machine and see if we experience the same issue. The more we can eliminate (or change), the easier we can track down the issue.

Additionally, we may be able to use tools to check for specific situations. For example, to investigate issue #2, we can utilize runtime tools which check for uninitialized variables.

These problems are as much about your brainstorming ability as they are about your approach. Do you jump all over the place, shouting out random suggestions? Or do you approach it in a logical, structured manner? Hopefully, it's the latter.

11.3 Chess Test: We have the following method used in a chess game: boolean `canMoveTo(int x, int y)`. This method is part of the `Piece` class and returns whether or not the piece can move to position `(x, y)`. Explain how you would test this method.

pg 157

SOLUTION

In this problem, there are two primary types of testing: extreme case validation (ensuring that the program doesn't crash on bad input), and general case testing. We'll start with the first type.

Testing Type #1: Extreme Case Validation

We need to ensure that the program handles bad or unusual input gracefully. This means checking the following conditions:

- Test with negative numbers for x and y
- Test with x larger than the width
- Test with y larger than the height
- Test with a completely full board
- Test with an empty or nearly empty board
- Test with far more white pieces than black

- Test with far more black pieces than white

For the error cases above, we should ask our interviewer whether we want to return false or throw an exception, and we should test accordingly.

Testing Type #2: General Testing:

General testing is much more expansive. Ideally, we would test every possible board, but there are far too many boards. We can, however, perform a reasonable coverage of different boards.

There are 6 pieces in chess, so we can test each piece against every other piece, in every possible direction. This would look something like the below code:

```
1  foreach piece a:
2    for each other type of piece b (6 types + empty space)
3      foreach direction d
4        Create a board with piece a.
5        Place piece b in direction d.
6        Try to move - check return value.
```

The key to this problem is recognizing that we can't test every possible scenario, even if we would like to. So, instead, we must focus on the essential areas.

11.4 No Test Tools: How would you load test a webpage without using any test tools?

pg 157

SOLUTION

Load testing helps to identify a web application's maximum operating capacity, as well as any bottlenecks that may interfere with its performance. Similarly, it can check how an application responds to variations in load.

To perform load testing, we must first identify the performance critical scenarios and the metrics which fulfill our performance objectives. Typical criteria include:

- Response time
- Throughput
- Resource utilization
- Maximum load that the system can bear.

Then, we design tests to simulate the load, taking care to measure each of these criteria.

In the absence of formal testing tools, we can basically create our own. For example, we could simulate concurrent users by creating thousands of virtual users. We would write a multi-threaded program with thousands of threads, where each thread acts as a real-world user loading the page. For each user, we would programmatically measure response time, data I/O, etc.

We would then analyze the results based on the data gathered during the tests and compare it with the accepted values.

11.5 Test a Pen: How would you test a pen?

pg 157

SOLUTION

This problem is largely about understanding the constraints and approaching the problem in a structured manner.

To understand the constraints, you should ask a lot of questions to understand the "who, what, where, when, how and why" of a problem (or as many of those as apply to the problem). Remember that a good tester understands exactly what he is testing before starting the work.

To illustrate the technique in this problem, let us guide you through a mock conversation.

- **Interviewer:** How would you test a pen?
- **Candidate:** Let me find out a bit about the pen. Who is going to use the pen?
- **Interviewer:** Probably children.
- **Candidate:** Okay, that's interesting. What will they be doing with it? Will they be writing, drawing, or doing something else with it?
- **Interviewer:** Drawing.
- **Candidate:** Okay, great. On what? Paper? Clothing? Walls?
- **Interviewer:** On clothing.
- **Candidate:** Great. What kind of tip does the pen have? Felt? Ballpoint? Is it intended to wash off, or is it intended to be permanent?
- **Interviewer:** It's intended to wash off.

Many questions later, you may get to this:

- **Candidate:** Okay, so as I understand it, we have a pen that is being targeted at 5 to 10-year-olds. The pen has a felt tip and comes in red, green, blue and black. It's intended to wash off when clothing is washed. Is that correct?

The candidate now has a problem that is significantly different from what it initially seemed to be. This is not uncommon. In fact, many interviewers intentionally give a problem that seems clear (everyone knows what a pen is!), only to let you discover that it's quite a different problem from what it seemed. Their belief is that users do the same thing, though users do so accidentally.

Now that you understand what you're testing, it's time to come up with a plan of attack. The key here is *structure*.

Consider what the different components of the object or problem, and go from there. In this case, the components might be:

- *Fact check:* Verify that the pen is felt tip and that the ink is one of the allowed colors.
- *Intended use:* Drawing. Does the pen write properly on clothing?
- *Intended use:* Washing. Does it wash off of clothing (even if it's been there for an extended period of time)? Does it wash off in hot, warm and cold water?
- *Safety:* Is the pen safe (non-toxic) for children?
- *Unintended uses:* How else might children use the pen? They might write on other surfaces, so you need to check whether the behavior there is correct. They might also stomp on the pen, throw it, and so on.

You'll need to make sure that the pen holds up under these conditions.

Remember that in any testing question, you need to test both the intended and unintended scenarios. People don't always use the product the way you want them to.

11.6 Test an ATM: How would you test an ATM in a distributed banking system?

pg 157

SOLUTION

The first thing to do on this question is to clarify assumptions. Ask the following questions:

- Who is going to use the ATM? Answers might be "anyone," or it might be "blind people," or any number of other answers.
- What are they going to use it for? Answers might be "withdrawing money," "transferring money," "checking their balance," or many other answers.
- What tools do we have to test? Do we have access to the code, or just to the ATM?

Remember: a good tester makes sure she knows what she's testing!

Once we understand what the system looks like, we'll want to break down the problem into different testable components. These components include:

- Logging in
- Withdrawing money
- Depositing money
- Checking balance
- Transferring money

We would probably want to use a mix of manual and automated testing.

Manual testing would involve going through the steps above, making sure to check for all the error cases (low balance, new account, nonexistent account, and so on).

Automated testing is a bit more complex. We'll want to automate all the standard scenarios, as shown above, and we also want to look for some very specific issues, such as race conditions. Ideally, we would be able to set up a closed system with fake accounts and ensure that, even if someone withdraws and deposits money rapidly from different locations, he never gets money or loses money that he shouldn't.

Above all, we need to prioritize security and reliability. People's accounts must always be protected, and we must make sure that money is always properly accounted for. No one wants to unexpectedly lose money! A good tester understands the system priorities.

12

Solutions to C and C++

12.1 Last K Lines: Write a method to print the last K lines of an input file using C++.

pg 163

SOLUTION

One brute force way could be to count the number of lines (N) and then print from N-K to Nth line. But this requires two reads of the file, which is unnecessarily costly. We need a solution which allows us to read just once and be able to print the last K lines.

We can allocate an array for all K lines and the last K lines we've read in the array. , and so on. Each time that we read a new line, we purge the oldest line from the array.

But—you might ask—wouldn't this require shifting elements in the array, which is also very expensive? No, not if we do it correctly. Instead of shifting the array each time, we will use a circular array.

With a circular array, we always replace the oldest item when we read a new line. The oldest item is tracked in a separate variable, which adjusts as we add new items.

The following is an example of a circular array:

```
step 1 (initially): array = {a, b, c, d, e, f}. p = 0
step 2 (insert g):  array = {g, b, c, d, e, f}. p = 1
step 3 (insert h):  array = {g, h, c, d, e, f}. p = 2
step 4 (insert i):  array = {g, h, i, d, e, f}. p = 3
```

The code below implements this algorithm.

```
1  void printLast10Lines(char* fileName) {
2    const int K = 10;
3    ifstream file (fileName);
4    string L[K];
5    int size = 0;
6
7    /* read file line by line into circular array */
8    /* peek() so an EOF following a line ending is not considered a separate line */
9    while (file.peek() != EOF) {
10     getline(file, L[size % K]);
11     size++;
12   }
13
14   /* compute start of circular array, and the size of it */
15   int start = size > K ? (size % K) : 0;
16   int count = min(K, size);
17
```

```
18    /* print elements in the order they were read */
19    for (int i = 0; i < count; i++) {
20       cout << L[(start + i) % K] << endl;
21    }
22 }
```

This solution will require reading in the whole file, but only ten lines will be in memory at any given point.

12.2 Reverse String: Implement a function void reverse(char* str) in C or C++ which reverses a null-terminated string.

pg 163

SOLUTION

This is a classic interview question. The only "gotcha" is to try to do it in place, and to be careful for the `null` character.

We will implement this in C.

```
1  void reverse(char *str) {
2     char* end = str;
3     char tmp;
4     if (str) {
5        while (*end) { /* find end of the string */
6           ++end;
7        }
8        --end; /* set one char back, since last char is null */
9
10       /* swap characters from start of string with the end of the string, until the
11        * pointers meet in middle. */
12       while (str < end) {
13          tmp = *str;
14          *str++ = *end;
15          *end-- = tmp;
16       }
17    }
18 }
```

This is just one of many ways to implement this solution. We could even implement this code recursively (but we wouldn't recommend it).

12.3 Hash Table vs STL Map: Compare and contrast a hash table and an STL map. How is a hash table implemented? If the number of inputs is small, which data structure options can be used instead of a hash table?

pg 163

SOLUTION

In a hash table, a value is stored by calling a hash function on a key. Values are not stored in sorted order. Additionally, since hash tables use the key to find the index that will store the value, an insert or lookup can be done in amortized `O(1)` time (assuming few collisions in the hash table). In a hash table, one must also handle potential collisions. This is often done by chaining, which means to create a linked list of all the values whose keys map to a particular index.

An STL map inserts the key/value pairs into a binary search tree based on the keys. There is no need to handle collisions, and, since the tree is balanced, the insert and lookup time is guaranteed to be `O(log N)`.

How is a hash table implemented?

A hash table is traditionally implemented with an array of linked lists. When we want to insert a key/value pair, we map the key to an index in the array using a hash function. The value is then inserted into the linked list at that position.

Note that the elements in a linked list at a particular index of the array do not have the same key. Rather, `hashFunction(key)` is the same for these values. Therefore, in order to retrieve the value for a specific key, we need to store in each node both the exact key and the value.

To summarize, the hash table will be implemented with an array of linked lists, where each node in the linked list holds two pieces of data: the value and the original key. In addition, we will want to note the following design criteria:

1. We want to use a good hash function to ensure that the keys are well distributed. If they are not well distributed, then we would get a lot of collisions and the speed to find an element would decline.
2. No matter how good our hash function is, we will still have collisions, so we need a method for handling them. This often means chaining via a linked list, but it's not the only way.
3. We may also wish to implement methods to dynamically increase or decrease the hash table size depending on capacity. For example, when the ratio of the number of elements to the table size exceeds a certain threshold, we may wish to increase the hash table size. This would mean creating a new hash table and transferring the entries from the old table to the new table. Because this is an expensive operation, we want to be careful to not do it too often.

What can be used instead of a hash table, if the number of inputs is small?

You can use an STL map or a binary tree. Although this takes `O(log(n))` time, the number of inputs may be small enough to make this time negligible.

12.4 Virtual Functions: How do virtual functions work in C++?

pg 164

SOLUTION

A virtual function depends on a "vtable" or "Virtual Table." If any function of a class is declared to be virtual, a vtable is constructed which stores addresses of the virtual functions of this class. The compiler also adds a hidden `vptr` variable in all such classes which points to the vtable of that class. If a virtual function is not overridden in the derived class, the vtable of the derived class stores the address of the function in its parent class. The vtable is used to resolve the address of the function when the virtual function is called. Dynamic binding in C++ is performed through the vtable mechanism.

Thus, when we assign the derived class object to the base class pointer, the `vptr` variable points to the vtable of the derived class. This assignment ensures that the most derived virtual function gets called.

Consider the following code.

```
1  class Shape {
2    public:
3     int edge_length;
4     virtual int circumference () {
```

```
5       cout << "Circumference of Base Class\n";
6       return 0;
7     }
8   };
9
10  class Triangle: public Shape {
11    public:
12    int circumference () {
13       cout<< "Circumference of Triangle Class\n";
14       return 3 * edge_length;
15    }
16  };
17
18  void main() {
19    Shape * x = new Shape();
20    x->circumference(); // "Circumference of Base Class"
21    Shape *y = new Triangle();
22    y->circumference(); // "Circumference of Triangle Class"
23  }
```

In the previous example, `circumference` is a virtual function in the `Shape` class, so it becomes virtual in each of the derived classes (`Triangle`, etc). C++ non-virtual function calls are resolved at compile time with static binding, while virtual function calls are resolved at runtime with dynamic binding.

12.5 Shallow vs Deep Copy: What is the difference between deep copy and shallow copy? Explain how you would use each.

pg 164

SOLUTION

A shallow copy copies all the member values from one object to another. A deep copy does all this and also deep copies any pointer objects.

An example of shallow and deep copy is below.

```
1   struct Test {
2     char * ptr;
3   };
4
5   void shallow_copy(Test & src, Test & dest) {
6     dest.ptr = src.ptr;
7   }
8
9   void deep_copy(Test & src, Test & dest) {
10    dest.ptr = (char*)malloc(strlen(src.ptr) + 1);
11    strcpy(dest.ptr, src.ptr);
12  }
```

Note that `shallow_copy` may cause a lot of programming runtime errors, especially with the creation and deletion of objects. Shallow copy should be used very carefully and only when a programmer really understands what he wants to do. In most cases, shallow copy is used when there is a need to pass information about a complex structure without actual duplication of data. One must also be careful with destruction of objects in a shallow copy.

In real life, shallow copy is rarely used. Deep copy should be used in most cases, especially when the size of the copied structure is small.

12.6 Volatile: What is the significance of the keyword "volatile" in C?

pg 164

SOLUTION

The keyword `volatile` informs the compiler that the value of variable it is applied to can change from the outside, without any update done by the code. This may be done by the operating system, the hardware, or another thread. Because the value can change unexpectedly, the compiler will therefore reload the value each time from memory.

A volatile integer can be declared by either of the following statements:

```
int volatile x;
volatile int x;
```

To declare a pointer to a volatile integer, we do the following:

```
volatile int * x;
int volatile * x;
```

A volatile pointer to non-volatile data is rare, but can be done.

```
int * volatile x;
```

If you wanted to declare a volatile variable pointer for volatile memory (both pointer address and memory contained are volatile), you would do the following:

```
int volatile * volatile x;
```

Volatile variables are not optimized, which can be very useful. Imagine this function:

```
1  int opt = 1;
2  void Fn(void) {
3    start:
4      if (opt == 1) goto start;
5      else break;
6  }
```

At first glance, our code appears to loop infinitely. The compiler may try to optimize it to:

```
1  void Fn(void) {
2    start:
3      int opt = 1;
4      if (true)
5      goto start;
6  }
```

This becomes an infinite loop. However, an external operation might write '0' to the location of variable `opt`, thus breaking the loop.

To prevent the compiler from performing such optimization, we want to signal that another element of the system could change the variable. We do this using the `volatile` keyword, as shown below.

```
1  volatile int opt = 1;
2  void Fn(void) {
3    start:
4      if (opt == 1) goto start;
5      else break;
6  }
```

Volatile variables are also useful when multi-threaded programs have global variables and any thread can modify these shared variables. We may not want optimization on these variables.

12.7 Virtual Base Class: Why does a destructor in base class need to be declared virtual?

pg 164

SOLUTION

Let's think about why we have virtual methods to start with. Suppose we have the following code:

```
1  class Foo {
2   public:
3     void f();
4  };
5
6  class Bar : public Foo {
7   public:
8     void f();
9  }
10
11 Foo * p = new Bar();
12 p->f();
```

Calling `p->f()` will result in a call to `Foo::f()`. This is because p is a pointer to Foo, and `f()` is not virtual.

To ensure that `p->f()` will invoke the most derived implementation of `f()`, we need to declare `f()` to be a virtual function.

Now, let's go back to our destructor. Destructors are used to clean up memory and resources. If Foo's destructor were not virtual, then Foo's destructor would be called, even when p is *really* of type Bar.

This is why we declare destructors to be virtual; we want to ensure that the destructor for the most derived class is called.

12.8 Copy Node: Write a method that takes a pointer to a Node structure as a parameter and returns a complete copy of the passed in data structure. The Node data structure contains two pointers to other Nodes.

pg 164

SOLUTION

The algorithm will maintain a mapping from a node address in the original structure to the corresponding node in the new structure. This mapping will allow us to discover previously copied nodes during a traditional depth-first traversal of the structure. Traversals often mark visited nodes—the mark can take many forms and does not necessarily need to be stored in the node.

Thus, we have a simple recursive algorithm:

```
1  typedef map<Node*, Node*> NodeMap;
2
3  Node * copy_recursive(Node * cur, NodeMap & nodeMap) {
4     if (cur == NULL) {
5        return NULL;
6     }
7
8     NodeMap::iterator i = nodeMap.find(cur);
9     if (i != nodeMap.end()) {
10       // we've been here before, return the copy
11       return i->second;
12    }
```

```
13
14    Node * node = new Node;
15    nodeMap[cur] = node; // map current before traversing links
16    node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
17    node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
18    return node;
19  }
20
21  Node * copy_structure(Node * root) {
22    NodeMap nodeMap; // we will need an empty map
23    return copy_recursive(root, nodeMap);
24  }
```

12.9 Smart Pointer: Write a smart pointer class. A smart pointer is a data type, usually implemented with templates, that simulates a pointer while also providing automatic garbage collection. It automatically counts the number of references to a *SmartPointer<T*>* object and frees the object of type T when the reference count hits zero.

pg 164

SOLUTION

A smart pointer is the same as a normal pointer, but it provides safety via automatic memory management. It avoids issues like dangling pointers, memory leaks and allocation failures. The smart pointer must maintain a single reference count for all references to a given object.

This is one of those problems that seems at first glance pretty overwhelming, especially if you're not a C++ expert. One useful way to approach the problem is to divide the problem into two parts: (1) outline the pseudocode and approach and then (2) implement the detailed code.

In terms of the approach, we need a reference count variable that is incremented when we add a new reference to the object and decremented when we remove a reference. The code should look something like the below pseudocode:

```
1   template <class T> class SmartPointer {
2     /* The smart pointer class needs pointers to both the object itself and to the
3      * ref count. These must be pointers, rather than the actual object or ref count
4      * value, since the goal of a smart pointer is that the reference count is
5      * tracked across multiple smart pointers to one object. */
6     T * obj;
7     unsigned * ref_count;
8   }
```

We know we need constructors and a single destructor for this class, so let's add those first.

```
1   SmartPointer(T * object) {
2     /* We want to set the value of T * obj, and set the reference counter to 1. */
3   }
4
5   SmartPointer(SmartPointer<T>& sptr) {
6     /* This constructor creates a new smart pointer that points to an existing
7      * object. We will need to first set obj and ref_count to pointer to sptr's obj
8      * and ref_count. Then, because we created a new reference to obj, we need to
9      * increment ref_count. */
10  }
11
12  ~SmartPointer(SmartPointer<T> sptr) {
13    /* We are destroying a reference to the object. Decrement ref_count. If
```

```
     * ref_count is 0, then free the memory created by the integer and destroy the
     * object. */
}
```

There's one additional way that references can be created: by setting one `SmartPointer` equal to another. We'll want to override the `equal` operator to handle this, but for now, let's sketch the code like this.

```
onSetEquals(SmartPoint<T> ptr1, SmartPoint<T> ptr2) {
  /* If ptr1 has an existing value, decrement its reference count. Then, copy the
   * pointers to obj and ref_count over. Finally, since we created a new
   * reference, we need to increment ref_count. */
}
```

Getting just the approach, even without filling in the complicated C++ syntax, would count for a lot. Finishing out the code is now just a matter of filling the details.

```
template <class T> class SmartPointer {
 public:
  SmartPointer(T * ptr) {
    ref = ptr;
    ref_count = (unsigned*)malloc(sizeof(unsigned));
    *ref_count = 1;
  }

  SmartPointer(SmartPointer<T> & sptr) {
    ref = sptr.ref;
    ref_count = sptr.ref_count;
    ++(*ref_count);
  }

  /* Override the equal operator, so that when you set one smart pointer equal to
   * another the old smart pointer has its reference count decremented and the new
   * smart pointer has its reference count incrememented. */
  SmartPointer<T> & operator=(SmartPointer<T> & sptr) {
    if (this == &sptr) return *this;

    /* If already assigned to an object, remove one reference. */
    if (*ref_count > 0) {
      remove();
    }

    ref = sptr.ref;
    ref_count = sptr.ref_count;
    ++(*ref_count);
    return *this;
  }

  ~SmartPointer() {
    remove(); // Remove one reference to object.
  }

  T getValue() {
    return *ref;
  }

 protected:
  void remove() {
    --(*ref_count);
    if (*ref_count == 0) {
```

```
44          delete ref;
45          free(ref_count);
46          ref = NULL;
47          ref_count = NULL;
48        }
49      }
50
51      T * ref;
52      unsigned * ref_count;
53  };
```

The code for this problem is complicated, and you probably wouldn't be expected to complete it flawlessly.

12.10 Malloc: Write an aligned malloc and free function that supports allocating memory such that the memory address returned is divisible by a specific power of two.

EXAMPLE

`align_malloc(1000,128)` will return a memory address that is a multiple of 128 and that points to memory of size 1000 bytes.

`aligned_free()` will free memory allocated by `align_malloc`.

pg 164

SOLUTION

Typically, with `malloc`, we do not have control over where the memory is allocated within the heap. We just get a pointer to a block of memory which could start at any memory address within the heap.

We need to work with these constraints by requesting enough memory that we can return a memory address which is divisible by the desired value.

Suppose we are requesting a 100-byte chunk of memory, and we want it to start at a memory address that is a multiple of 16. How much extra memory would we need to allocate to ensure that we can do so? We would need to allocate an extra 15 bytes. With these 15 bytes, plus another 100 bytes right after that sequence, we know that we would have a memory address divisible by 16 with space for 100 bytes.

We could then do something like:

```
1  void* aligned_malloc(size_t required_bytes, size_t alignment) {
2    int offset = alignment - 1;
3    void* p = (void*) malloc(required_bytes + offset);
4    void* q = (void*) (((size_t)(p) + offset) & ~(alignment - 1));
5    return q;
6  }
```

Line 4 is a bit tricky, so let's discuss it. Suppose `alignment` is 16. We know that one of the first 16 memory address in the block at p must be divisible by 16. With `(p + 15) & 11...10000` we advance as need to this address. ANDing the last four bits of `p + 15` with `0000` guarantees that this new value will be divisible by 16 (either at the original p or in one of the following 15 addresses).

This solution is *almost* perfect, except for one big issue: how do we free the memory?

We've allocated an extra 15 bytes, in the above example, and we need to free them when we free the "real" memory.

We can do this by storing, in this "extra" memory, the address of where the full memory block begins. We will store this immediately before the aligned memory block. Of course, this means that we now need to allocate even *more* extra memory to ensure that we have enough space to store this pointer.

Therefore, to guarantee both an aligned address and space for this pointer, we will need to allocate an additional `alignment - 1 + sizeof(void*)` bytes.

The code below implements this approach.

```
1  void* aligned_malloc(size_t required_bytes, size_t alignment) {
2    void* p1; // initial block
3    void* p2; // aligned block inside initial block
4    int offset = alignment - 1 + sizeof(void*);
5    if ((p1 = (void*)malloc(required_bytes + offset)) == NULL) {
6      return NULL;
7    }
8    p2 = (void*)(((size_t)(p1) + offset) & ~(alignment - 1));
9    ((void **)p2)[-1] = p1;
10   return p2;
11 }
12
13 void aligned_free(void *p2) {
14   /* for consistency, we use the same names as aligned_malloc*/
15   void* p1 = ((void**)p2)[-1];
16   free(p1);
17 }
```

Let's look at the pointer arithmetic in lines 9 and 15. If we treat p2 as a `void**` (or an array of `void*`'s), we can just look at the index `- 1` to retrieve p1.

In `aligned_free`, we take p2 as the same p2 returned from `aligned_malloc`. As before, we know that the value of p1 (which points to the beginning of the full memory block) was stored just before p2. By freeing p1, we deallocate the whole memory block.

12.11 2D Alloc: Write a function in C called `my2DAlloc` which allocates a two-dimensional array. Minimize the number of calls to `malloc` and make sure that the memory is accessible by the notation `arr[i][j]`.

pg 164

SOLUTION

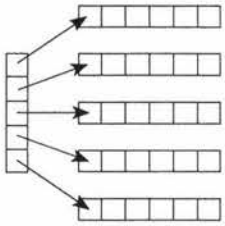
As you may know, a two-dimensional array is essentially an array of arrays. Since we use pointers with arrays, we can use double pointers to create a double array.

The basic idea is to create a one-dimensional array of pointers. Then, for each array index, we create a new one-dimensional array. This gives us a two-dimensional array that can be accessed via array indices.

The code below implements this.

```
1  int** my2DAlloc(int rows, int cols) {
2    int** rowptr;
3    int i;
4    rowptr = (int**) malloc(rows * sizeof(int*));
5    for (i = 0; i < rows; i++)  {
6      rowptr[i] = (int*) malloc(cols * sizeof(int));
7    }
8    return rowptr;
9  }
```

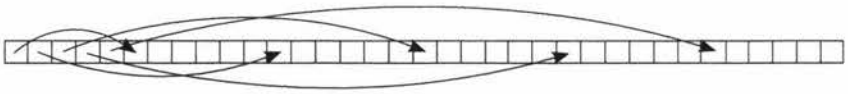
Observe how, in the above code, we've told `rowptr` where exactly each index should point. The following diagram represents how this memory is allocated.



To free this memory, we cannot simply call `free` on `rowptr`. We need to make sure to free not only the memory from the first `malloc` call, but also each subsequent call.

```
1  void my2DDealloc(int** rowptr, int rows) {
2    for (i = 0; i < rows; i++) {
3      free(rowptr[i]);
4    }
5    free(rowptr);
6  }
```

Rather than allocating the memory in many different blocks (one block for each row, plus one block to specify *where* each row is located), we can allocate this in a consecutive block of memory. Conceptually, for a two-dimensional array with five rows and six columns, this would look like the following.



If it seems strange to view the 2D array like this (and it probably does), remember that this is fundamentally no different than the first diagram. The only difference is that the memory is in a contiguous block, so our first five (in this example) elements point elsewhere in the same block of memory.

To implement this solution, we do the following.

```
1  int** my2DAlloc(int rows, int cols) {
2    int i;
3    int header = rows * sizeof(int*);
4    int data = rows * cols * sizeof(int);
5    int** rowptr = (int**)malloc(header + data);
6    if (rowptr == NULL) return NULL;
7
8    int* buf = (int*) (rowptr + rows);
9    for (i = 0; i < rows; i++) {
10     rowptr[i] = buf + i * cols;
11   }
12   return rowptr;
13 }
```

You should carefully observe what is happening on lines 11 through 13. If there are five rows of six columns each, `array[0]` will point to `array[5]`, `array[1]` will point to `array[11]`, and so on.

Then, when we actually call `array[1][3]`, the computer looks up `array[1]`, which is a pointer to another spot in memory—specifically, a pointer to `array[5]`. This element is treated as its own array, and we then get the third (zero-indexed) element from it.

Constructing the array in a single call to `malloc` has the added benefit of allowing disposal of the array with a single `free` call rather than using a special function to free the remaining data blocks.

13

Solutions to Java

13.1 Private Constructor: In terms of inheritance, what is the effect of keeping a constructor private?

pg 167

SOLUTION

Declaring a constructor `private` on class A means that you can only access the (private) constructor if you could also access A's private methods. Who, other than A, can access A's private methods and constructor? A's inner classes can. Additionally, if A is an inner class of Q, then Q's other inner classes can.

This has direct implications for inheritance, since a subclass calls its parent's constructor. The class A can be inherited, but only by its own or its parent's inner classes.

13.2 Return from Finally: In Java, does the finally block get executed if we insert a return statement inside the try block of a try-catch-finally?

pg 167

SOLUTION

Yes, it will get executed. The `finally` block gets executed when the `try` block exits. Even when we attempt to exit within the `try` block (via a `return` statement, a `continue` statement, a `break` statement or any exception), the `finally` block will still be executed.

Note that there are some cases in which the `finally` block will not get executed, such as the following:

- If the virtual machine exits during `try/catch` block execution.
- If the thread which is executing during the `try/catch` block gets killed.

13.3 Final, etc.: What is the difference between `final`, `finally`, and `finalize`?

pg 167

SOLUTIONS

Despite their similar sounding names, `final`, `finally` and `finalize` have very different purposes. To speak in very general terms, `final` is used to control whether a variable, method, or class is "changeable." The `finally` keyword is used in a `try/ catch` block to ensure that a segment of code is always executed. The `finalize()` method is called by the garbage collector once it determines that no more references exist.

Further detail on these keywords and methods is provided below.

final

The final statement has a different meaning depending on its context.

- When applied to a variable (primitive): The value of the variable cannot change.
- When applied to a variable (reference): The reference variable cannot point to any other object on the heap.
- When applied to a method: The method cannot be overridden.
- When applied to a class: The class cannot be subclassed.

finally keyword

There is an optional `finally` block after the `try` block or after the `catch` block. Statements in the `finally` block will always be executed, even if an exception is thrown (except if Java Virtual Machine exits from the `try` block). The `finally` block is often used to write the clean-up code. It will be executed after the `try` and `catch` blocks, but before control transfers back to its origin.

Watch how this plays out in the example below.

```
1  public static String lem() {
2    System.out.println("lem");
3    return "return from lem";
4  }
5
6  public static String foo() {
7    int x = 0;
8    int y = 5;
9    try {
10     System.out.println("start try");
11     int b = y / x;
12     System.out.println("end try");
13     return "returned from try";
14   } catch (Exception ex) {
15     System.out.println("catch");
16     return lem() + " | returned from catch";
17   } finally {
18     System.out.println("finally");
19   }
20 }
21
22 public static void bar() {
23   System.out.println("start bar");
24   String v = foo();
25   System.out.println(v);
26   System.out.println("end bar");
27 }
28
29 public static void main(String[] args) {
30   bar();
31 }
```

The output for this code is the following:

```
1  start bar
```

```
2  start try
3  catch
4  lem
5  finally
6  return from lem | returned from catch
7  end bar
```

Look carefully at lines 3 to 5 in the output. The `catch` block is fully executed (including the function call in the `return` statement), then the `finally` block, and then the function actually returns.

finalize()

The automatic garbage collector calls the `finalize()` method just before actually destroying the object. A class can therefore override the `finalize()` method from the `Object` class in order to define custom behavior during garbage collection.

```
1  protected void finalize() throws Throwable {
2    /* Close open files, release resources, etc */
3  }
```

13.4 Generics vs. Templates: Explain the difference between templates in C++ and generics in Java.

pg 167

SOLUTION

Many programmers consider templates and generics to be essentially equivalent because both allow you to do something like `List<String>`. But, *how* each language does this, and *why*, varies significantly.

The implementation of Java generics is rooted in an idea of "type erasure." This technique eliminates the parameterized types when source code is translated to the Java Virtual Machine (JVM) byte code.

For example, suppose you have the Java code below:

```
1  Vector<String>  vector = new Vector<String>();
2  vector.add(new String("hello"));
3  String  str = vector.get(0);
```

During compilation, this code is re-written into:

```
1  Vector vector = new Vector();
2  vector.add(new String("hello"));
3  String  str = (String) vector.get(0);
```

The use of Java generics didn't really change much about our capabilities; it just made things a bit prettier. For this reason, Java generics are sometimes called "syntactic sugar."

This is quite different from C++. In C++, templates are essentially a glorified macro set, with the compiler creating a new copy of the template code for each type. Proof of this is in the fact that an instance of `MyClass<Foo>` will not share a static variable with `MyClass<Bar>`. Two instances of `MyClass<Foo>`, however, will share a static variable.

To illustrate this, consider the code below:

```
1  /*** MyClass.h ***/
2  template<class T> class MyClass {
3   public:
4     static int val;
5     MyClass(int v) { val = v; }
6  };
7
```

```
/*** MyClass.cpp ***/
template<typename T>
int MyClass<T>::bar;

template class MyClass<Foo>;
template class MyClass<Bar>;

/*** main.cpp ***/
MyClass<Foo> * foo1 = new MyClass<Foo>(10);
MyClass<Foo> * foo2 = new MyClass<Foo>(15);
MyClass<Bar> * bar1 = new MyClass<Bar>(20);
MyClass<Bar> * bar2 = new MyClass<Bar>(35);

int f1 = foo1->val; // will equal 15
int f2 = foo2->val; // will equal 15
int b1 = bar1->val; // will equal 35
int b2 = bar2->val; // will equal 35
```

In Java, static variables are shared across instances of `MyClass`, regardless of the different type parameters.

Java generics and C++ templates have a number of other differences. These include:

- C++ templates can use primitive types, like `int`. Java cannot and must instead use `Integer`.
- In Java, you can restrict the template's type parameters to be of a certain type. For instance, you might use generics to implement a `CardDeck` and specify that the type parameter must extend from `CardGame`.
- In C++, the type parameter can be instantiated, whereas Java does not support this.
- In Java, the type parameter (i.e., the `Foo` in `MyClass<Foo>`) cannot be used for static methods and variables, since these would be shared between `MyClass<Foo>` and `MyClass<Bar>`. In C++, these classes are different, so the type parameter can be used for static methods and variables.
- In Java, all instances of `MyClass`, regardless of their type parameters, are the same type. The type parameters are erased at runtime. In C++, instances with different type parameters are different types.

Remember: Although Java generics and C++ templates look the same in many ways, they are very different.

13.5 TreeMap, HashMap, LinkedHashMap: Explain the differences between `TreeMap`, `HashMap`, and `LinkedHashMap`. Provide an example of when each one would be best.

pg 167

SOLUTION

All offer a key->value map and a way to iterate through the keys. The most important distinction between these classes is the time guarantees and the ordering of the keys.

- `HashMap` offers `O(1)` lookup and insertion. If you iterate through the keys, though, the ordering of the keys is essentially arbitrary. It is implemented by an array of linked lists.
- `TreeMap` offers `O(log N)` lookup and insertion. Keys are ordered, so if you need to iterate through the keys in sorted order, you can. This means that keys must implement the `Comparable` interface. `TreeMap` is implemented by a Red-Black Tree.
- `LinkedHashMap` offers `O(1)` lookup and insertion. Keys are ordered by their insertion order. It is implemented by doubly-linked buckets.

Imagine you passed an empty `TreeMap`, `HashMap`, and `LinkedHashMap` into the following function:

```
1  void insertAndPrint(AbstractMap<Integer, String> map) {
2    int[] array = {1, -1, 0};
3    for (int x : array) {
4      map.put(x, Integer.toString(x));
5    }
6
7    for (int k : map.keySet()) {
8      System.out.print(k + ", ");
9    }
10 }
```

The output for each will look like the results below.

HashMap	LinkedHashMap	TreeMap
(any ordering)	{1, -1, 0}	{-1, 0, 1}

Very important: The output of `LinkedHashMap` and `TreeMap` must look like the above. For `HashMap`, the output was, in my own tests, `{0, 1, -1}`, but it could be any ordering. There is no *guarantee* on the ordering.

When might you need ordering in real life?

- Suppose you were creating a mapping of names to `Person` objects. You might want to periodically output the people in alphabetical order by name. A `TreeMap` lets you do this.
- A `TreeMap` also offers a way to, given a name, output the next 10 people. This could be useful for a "More" function in many applications.
- A `LinkedHashMap` is useful whenever you need the ordering of keys to match the ordering of insertion. This might be useful in a caching situation, when you want to delete the oldest item.

Generally, unless there is a reason not to, you would use `HashMap`. That is, if you need to get the keys back in insertion order, then use `LinkedHashMap`. If you need to get the keys back in their true/natural order, then use `TreeMap`. Otherwise, `HashMap` is probably best. It is typically faster and requires less overhead.

13.6 Object Reflection: Explain what object reflection is in Java and why it is useful.

pg 168

SOLUTION

Object Reflection is a feature in Java that provides a way to get reflective information about Java classes and objects, and perform operations such as:

1. Getting information about the methods and fields present inside the class at runtime.
2. Creating a new instance of a class.
3. Getting and setting the object fields directly by getting field reference, regardless of what the access modifier is.

The code below offers an example of object reflection.

```
1  /* Parameters */
2  Object[] doubleArgs = new Object[] { 4.2, 3.9 };
3
4  /* Get class */
5  Class rectangleDefinition = Class.forName("MyProj.Rectangle");
6
```

```
/* Equivalent: Rectangle rectangle = new Rectangle(4.2, 3.9); */
Class[] doubleArgsClass = new Class[] {double.class, double.class};
Constructor doubleArgsConstructor =
   rectangleDefinition.getConstructor(doubleArgsClass);
Rectangle rectangle = (Rectangle) doubleArgsConstructor.newInstance(doubleArgs);

/* Equivalent: Double area = rectangle.area(); */
Method m = rectangleDefinition.getDeclaredMethod("area");
Double area = (Double) m.invoke(rectangle);
```

This code does the equivalent of:

```
Rectangle rectangle = new Rectangle(4.2, 3.9);
Double area = rectangle.area();
```

Why Is Object Reflection Useful?

Of course, it doesn't seem very useful in the above example, but reflection can be very useful in some cases. Three main reasons are:

1. It can help you observe or manipulate the runtime behavior of applications.
2. It can help you debug or test programs, as you have direct access to methods, constructors, and fields.
3. You can call methods by name when you don't know the method in advance. For example, we may let the user pass in a class name, parameters for the constructor, and a method name. We can then use this information to create an object and call a method. Doing these operations without reflection would require a complex series of if-statements, if it's possible at all.

13.7 Lambda Expressions: There is a class `Country` that has methods `getContinent()` and `getPopulation()`. Write a function `int getPopulation(List<Country> countries, String continent)` that computes the total population of a given continent, given a list of all countries and the name of a continent.

pg 168

SOLUTION

This question really comes in two parts. First, we need to generate a list of the countries in North America. Then, we need to compute their total population.

Without lambda expressions, this is fairly straightforward to do.

```
int getPopulation(List<Country> countries, String continent) {
   int sum = 0;
   for (Country c : countries) {
      if (c.getContinent().equals(continent)) {
         sum += c.getPopulation();
      }
   }
   return sum;
}
```

To implement this with lambda expressions, let's break this up into multiple parts.

First, we use `filter` to get a list of the countries in the specified continent.

```
Stream<Country> northAmerica = countries.stream().filter(
   country -> { return country.getContinent().equals(continent);}
```

```
3   );
```

Second, we convert this into a list of populations using map.

```
1   Stream<Integer> populations = northAmerica.map(
2     c -> c.getPopulation()
3   );
```

Third and finally, we compute the sum using reduce.

```
1   int population = populations.reduce(0, (a, b) -> a + b);
```

This function puts it all together.

```
1   int getPopulation(List<Country> countries, String continent) {
2     /* Filter countries. */
3     Stream<Country> sublist = countries.stream().filter(
4       country -> { return country.getContinent().equals(continent);}
5     );
6
7     /* Convert to list of populations. */
8     Stream<Integer> populations = sublist.map(
9       c -> c.getPopulation()
10    );
11
12    /* Sum list. */
13    int population = populations.reduce(0, (a, b) -> a + b);
14    return population;
15  }
```

Alternatively, because of the nature of this specific problem, we can actually remove the filter entirely. The reduce operation can have logic that maps the population of countries not in the right continent to zero. The sum will effectively disregard countries not within continent.

```
1   int getPopulation(List<Country> countries, String continent) {
2     Stream<Integer> populations = countries.stream().map(
3       c -> c.getContinent().equals(continent) ? c.getPopulation() : 0);
4     return populations.reduce(0, (a, b) -> a + b);
5   }
```

Lambda functions were new to Java 8, so if you don't recognize them, that's probably why. Now is a great time to learn about them, though!

13.8 Lambda Random: Using Lambda expressions, write a function `List<Integer> getRandomSubset(List<Integer> list)` that returns a random subset of arbitrary size. All subsets (including the empty set) should be equally likely to be chosen.

pg 439

SOLUTION

It's tempting to approach this problem by picking a subset size from 0 to N and then generating a random subset of that size.

That creates two issues:

1. We'd have to weight those probabilities. If N > 1, there are more subsets of size N/2 than there are of subsets of size N (of which there is always only one).
2. It's actually more difficult to generate a subset of a restricted size (e.g., specifically 10) than it is to generate a subset of any size.

Instead, rather than generating a subset based on sizes, let's think about it based on elements. (The fact that we're told to use lambda expressions is also a hint that we should think about some sort of iteration or processing through the elements.)

Imagine we were iterating through {1, 2, 3} to generate a subset. Should 1 be in this subset?

We've got two choices: yes or no. We need to weight the probability of "yes" vs. "no" based on the percent of subsets that contain 1. So, what percent of elements contain 1?

For any specific element, there are as many subsets that contain the element as do not contain it. Consider the following:

```
{}          {1}
{2}         {1, 2}
{3}         {1, 3}
{2, 3}      {1, 2, 3}
```

Note how the difference between the subsets on the left and the subsets on the right is the existence of 1. The left and right sides must have the same number of subsets because we can convert from one to the other by just adding an element.

This means that we can generate a random subset by iterating through the list and flipping a coin (i.e., deciding on a 50/50 chance) to pick whether or not each element will be in it.

Without lambda expressions, we can write something like this:

```
1  List<Integer> getRandomSubset(List<Integer> list) {
2    List<Integer> subset = new ArrayList<Integer>();
3    Random random = new Random();
4    for (int item : list) {
5      /* Flip coin. */
6      if (random.nextBoolean()) {
7        subset.add(item);
8      }
9    }
10   return subset;
11 }
```

To implement this approach using lambda expressions, we can do the following:

```
1  List<Integer> getRandomSubset(List<Integer> list) {
2    Random random = new Random();
3    List<Integer> subset = list.stream().filter(
4      k -> { return random.nextBoolean(); /* Flip coin. */
5    }).collect(Collectors.toList());
6    return subset;
7  }
```

Or, we can use a predicate (defined within the class or within the function):

```
1  Random random = new Random();
2  Predicate<Object> flipCoin = o -> {
3    return random.nextBoolean();
4  };
5
6  List<Integer> getRandomSubset(List<Integer> list) {
7    List<Integer> subset = list.stream().filter(flipCoin).
8      collect(Collectors.toList());
9    return subset;
10 }
```

The nice thing about this implementation is that now we can apply the `flipCoin` predicate in other places.

14

Solutions to Databases

Questions 1 through 3 refer to the following database schema:

Apartments	
AptID	int
UnitNumber	varchar(10)
BuildingID	int

Buildings	
BuildingID	int
ComplexID	int
BuildingName	varchar(100)
Address	varchar(500)

Requests	
RequestID	int
Status	varchar(100)
AptID	int
Description	varchar(500)

Complexes	
ComplexID	int
ComplexName	varchar(100)

AptTenants	
TenantID	int
AptID	int

Tenants	
TenantID	int
TenantName	varchar(100)

Note that each apartment can have multiple tenants, and each tenant can have multiple apartments. Each apartment belongs to one building, and each building belongs to one complex.

14.1 Multiple Apartments: Write a SQL query to get a list of tenants who are renting more than one apartment.

pg 172

SOLUTION

To implement this, we can use the HAVING and GROUP BY clauses and then perform an INNER JOIN with Tenants.

```
1  SELECT TenantName
2  FROM Tenants
3  INNER JOIN
4     (SELECT TenantID FROM AptTenants GROUP BY TenantID HAVING count(*) > 1) C
5  ON Tenants.TenantID = C.TenantID
```

Whenever you write a GROUP BY clause in an interview (or in real life), make sure that anything in the SELECT clause is either an aggregate function or contained within the GROUP BY clause.

14.2 Open Requests: Write a SQL query to get a list of all buildings and the number of open requests (`Requests` in which `status` equals 'Open').

pg 173

SOLUTION

This problem uses a straightforward join of `Requests` and `Apartments` to get a list of building IDs and the number of open requests. Once we have this list, we join it again with the `Buildings` table.

```
1  SELECT BuildingName, ISNULL(Count, 0) as 'Count'
2  FROM Buildings
3  LEFT JOIN
4     (SELECT Apartments.BuildingID, count(*) as 'Count'
5      FROM Requests INNER JOIN Apartments
6      ON Requests.AptID = Apartments.AptID
7      WHERE Requests.Status = 'Open'
8      GROUP BY Apartments.BuildingID) ReqCounts
9  ON ReqCounts.BuildingID = Buildings.BuildingID
```

Queries like this that utilize sub-queries should be thoroughly tested, even when coding by hand. It may be useful to test the inner part of the query first, and then test the outer part.

14.3 Close All Requests: Building #11 is undergoing a major renovation. Implement a query to close all requests from apartments in this building.

pg 173

SOLUTION

UPDATE queries, like `SELECT` queries, can have `WHERE` clauses. To implement this query, we get a list of all apartment IDs within building #11 and the list of update requests from those apartments.

```
1  UPDATE Requests
2  SET Status = 'Closed'
3  WHERE AptID IN (SELECT AptID FROM Apartments WHERE BuildingID = 11)
```

14.4 Joins: What are the different types of joins? Please explain how they differ and why certain types are better in certain situations.

pg 173

SOLUTION

`JOIN` is used to combine the results of two tables. To perform a `JOIN`, each of the tables must have at least one field that will be used to find matching records from the other table. The join type defines which records will go into the result set.

Let's take for example two tables: one table lists the "regular" beverages, and another lists the calorie-free beverages. Each table has two fields: the beverage name and its product code. The "code" field will be used to perform the record matching.

Regular Beverages:

Name	Code
Budweiser	BUDWEISER
Coca-Cola	COCACOLA

Name	Code
Pepsi	PEPSI

Calorie-Free Beverages:

Name	Code
Diet Coca-Cola	COCACOLA
Fresca	FRESCA
Diet Pepsi	PEPSI
Pepsi Light	PEPSI
Purified Water	Water

If we wanted to join `Beverage` with `Calorie-Free Beverages`, we would have many options. These are discussed below.

- `INNER JOIN`: The result set would contain only the data where the criteria match. In our example, we would get three records: one with a `COCACOLA` code and two with `PEPSI` codes.
- `OUTER JOIN`: An `OUTER JOIN` will always contain the results of `INNER JOIN`, but it may also contain some records that have no matching record in the other table. `OUTER JOIN`s are divided into the following subtypes:
  - `LEFT OUTER JOIN`, or simply `LEFT JOIN`: The result will contain all records from the left table. If no matching records were found in the right table, then its fields will contain the `NULL` values. In our example, we would get four records. In addition to `INNER JOIN` results, `BUDWEISER` would be listed, because it was in the left table.
  - `RIGHT OUTER JOIN`, or simply `RIGHT JOIN`: This type of join is the opposite of `LEFT JOIN`. It will contain every record from the right table; the missing fields from the left table will be `NULL`. Note that if we have two tables, A and B, then we can say that the statement A `LEFT JOIN` B is equivalent to the statement B `RIGHT JOIN` A. In our example above, we will get five records. In addition to `INNER JOIN` results, `FRESCA` and `WATER` records will be listed.
  - `FULL OUTER JOIN`: This type of join combines the results of the `LEFT` and `RIGHT JOINS`. All records from both tables will be included in the result set, regardless of whether or not a matching record exists in the other table. If no matching record was found, then the corresponding result fields will have a `NULL` value. In our example, we will get six records.

14.5 Denormalization: What is denormalization? Explain the pros and cons.

pg 173

SOLUTION

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database.

By contrast, in a traditional normalized database, we store data in separate logical tables and attempt to minimize redundant data. We may strive to have only one copy of each piece of data in the database.

For example, in a normalized database, we might have a `Courses` table and a `Teachers` table. Each entry in `Courses` would store the `teacherID` for a `Course` but not the `teacherName`. When we need to retrieve a list of all `Courses` with the `Teacher` name, we would do a join between these two tables.

In some ways, this is great; if a teacher changes his or her name, we only have to update the name in one place.

The drawback, however, is that if the tables are large, we may spend an unnecessarily long time doing joins on tables.

Denormalization, then, strikes a different compromise. Under denormalization, we decide that we're okay with some redundancy and some extra effort to update the database in order to get the efficiency advantages of fewer joins.

Cons of Denormalization	Pros of Denormalization
Updates and inserts are more expensive.	Retrieving data is faster since we do fewer joins.
Denormalization can make update and insert code harder to write.	Queries to retrieve can be simpler (and therefore less likely to have bugs), since we need to look at fewer tables.
Data may be inconsistent. Which is the "correct" value for a piece of data?	
Data redundancy necessitates more storage.	

In a system that demands scalability, like that of any major tech companies, we almost always use elements of both normalized and denormalized databases.

14.6 Entity-Relationship Diagram: Draw an entity-relationship diagram for a database with companies, people, and professionals (people who work for companies).

pg 173

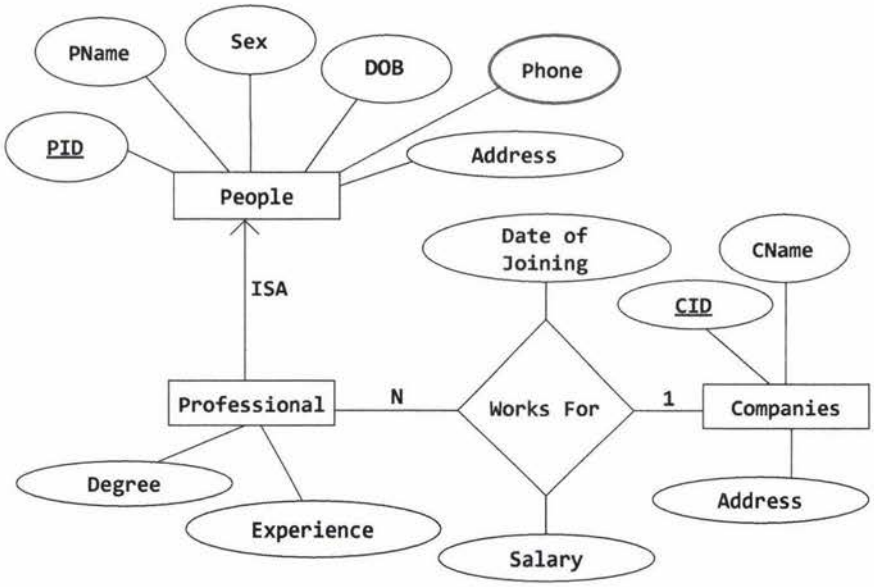
SOLUTION

People who work for `Companies` are `Professionals`. So, there is an `ISA` ("is a") relationship between `People` and `Professionals` (or we could say that a `Professional` is derived from `People`).

Each `Professional` has additional information such as degree and work experiences in addition to the properties derived from `People`.

A `Professional` works for one company at a time (probably—you might want to validate this assumption), but `Companies` can hire many `Professionals`. So, there is a many-to-one relationship between `Professionals` and `Companies`. This "Works For" relationship can store attributes such as an employee's start date and salary. These attributes are defined only when we relate a `Professional` with a `Company`.

A `Person` can have multiple phone numbers, which is why Phone is a multi-valued attribute.



14.7 Design Grade Database: Imagine a simple database storing information for students' grades. Design what this database might look like and provide a SQL query to return a list of the honor roll students (top 10%), sorted by their grade point average.

pg 173

SOLUTION

In a simplistic database, we'll have at least three objects: `Students`, `Courses`, and `CourseEnrollment`. `Students` will have at least a student name and ID and will likely have other personal information. `Courses` will contain the course name and ID and will likely contain the course description, professor, and other information. `CourseEnrollment` will pair `Students` and `Courses` and will also contain a field for `CourseGrade`.

Students	
StudentID	int
StudentName	varchar(100)
Address	varchar(500)

Courses	
CourseID	int
CourseName	varchar(100)
ProfessorID	int

CourseEnrollment	
CourseID	int
StudentID	int
Grade	float
Term	int

This database could get arbitrarily more complicated if we wanted to add in professor information, billing information, and other data.

Using the Microsoft SQL Server TOP ... PERCENT function, we might (incorrectly) first try a query like this:

```
1  SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA,
2                                                    CourseEnrollment.StudentID
3  FROM CourseEnrollment
4  GROUP BY CourseEnrollment.StudentID
5  ORDER BY AVG(CourseEnrollment.Grade)
```

The problem with the above code is that it will return literally the top 10% of rows, when sorted by GPA. Imagine a scenario in which there are 100 students, and the top 15 students all have 4.0 GPAs. The above function will only return 10 of those students, which is not really what we want. In case of a tie, we want to include the students who tied for the top 10% -- even if this means that our honor roll includes more than 10% of the class.

To correct this issue, we can build something similar to this query, but instead first get the GPA cut off.

```
1  DECLARE @GPACutOff float;
2  SET @GPACutOff = (SELECT min(GPA) as 'GPAMin' FROM (
3       SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA
4       FROM CourseEnrollment
5       GROUP BY CourseEnrollment.StudentID
6       ORDER BY GPA desc) Grades);
```

Then, once we have @GPACutOff defined, selecting the students with at least this GPA is reasonably straightforward.

```
1  SELECT StudentName, GPA
2  FROM (SELECT AVG(CourseEnrollment.Grade) AS GPA, CourseEnrollment.StudentID
3        FROM CourseEnrollment
4        GROUP BY CourseEnrollment.StudentID
5        HAVING AVG(CourseEnrollment.Grade) >= @GPACutOff) Honors
6  INNER JOIN Students ON Honors.StudentID = Student.StudentID
```

Be very careful about what implicit assumptions you make. If you look at the above database description, what potentially incorrect assumption do you see? One is that each course can only be taught by one professor. At some schools, courses may be taught by multiple professors.

However, you *will* need to make some assumptions, or you'd drive yourself crazy. Which assumptions you make is less important than just recognizing *that* you made assumptions. Incorrect assumptions, both in the real world and in an interview, can be dealt with *as long as they are acknowledged.*

Remember, additionally, that there's a trade-off between flexibility and complexity. Creating a system in which a course can have multiple professors does increase the database's flexibility, but it also increases its complexity. If we tried to make our database flexible to every possible situation, we'd wind up with something hopelessly complex.

Make your design reasonably flexible, and state any other assumptions or constraints. This goes for not just database design, but object-oriented design and programming in general.

15

Solutions to Threads and Locks

15.1 Thread vs. Process: What's the difference between a thread and a process?

pg 179

SOLUTION

Processes and threads are related to each other but are fundamentally different.

A process can be thought of as an instance of a program in execution. A process is an independent entity to which system resources (e.g., CPU time and memory) are allocated. Each process is executed in a separate address space, and one process cannot access the variables and data structures of another process. If a process wishes to access another process' resources, inter-process communications have to be used. These include pipes, files, sockets, and other forms.

A thread exists within a process and shares the process' resources (including its heap space). Multiple threads within the same process will share the same heap space. This is very different from processes, which cannot directly access the memory of another process. Each thread still has its own registers and its own stack, but other threads can read and write the heap memory.

A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads.

15.2 Context Switch: How would you measure the time spent in a context switch?

pg 179

SOLUTION

This is a tricky question, but let's start with a possible solution.

A context switch is the time spent switching between two processes (i.e., bringing a waiting process into execution and sending an executing process into waiting/terminated state). This happens in multitasking. The operating system must bring the state information of waiting processes into memory and save the state information of the currently running process.

In order to solve this problem, we would like to record the timestamps of the last and first instruction of the swapping processes. The context switch time is the difference in the timestamps between the two processes.

Let's take an easy example: Assume there are only two processes, P_1 and P_2.

P_1 is executing and P_2 is waiting for execution. At some point, the operating system must swap P_1 and P_2—let's assume it happens at the Nth instruction of P_1. If $t_{x,k}$ indicates the timestamp in microseconds of the kth instruction of process x, then the context switch would take $t_{2,1} - t_{1,n}$ microseconds.

The tricky part is this: how do we know when this swapping occurs? We cannot, of course, record the time-stamp of every instruction in the process.

Another issue is that swapping is governed by the scheduling algorithm of the operating system and there may be many kernel level threads which are also doing context switches. Other processes could be contending for the CPU or the kernel handling interrupts. The user does not have any control over these extraneous context switches. For instance, if at time $t_{1,n}$ the kernel decides to handle an interrupt, then the context switch time would be overstated.

In order to overcome these obstacles, we must first construct an environment such that after P_1 executes, the task scheduler immediately selects P_2 to run. This may be accomplished by constructing a data channel, such as a pipe, between P_1 and P_2 and having the two processes play a game of ping-pong with a data token.

That is, let's allow P_1 to be the initial sender and P_2 to be the receiver. Initially, P_2 is blocked (sleeping) as it awaits the data token. When P_1 executes, it delivers the token over the data channel to P_2 and immediately attempts to read a response token. However, since P_2 has not yet had a chance to run, no such token is available for P_1 and the process is blocked. This relinquishes the CPU.

A context switch results and the task scheduler must select another process to run. Since P_2 is now in a ready-to-run state, it is a desirable candidate to be selected by the task scheduler for execution. When P_2 runs, the roles of P_1 and P_2 are swapped. P_2 is now acting as the sender and P_1 as the blocked receiver. The game ends when P_2 returns the token to P_1.

To summarize, an iteration of the game is played with the following steps:

1. P_2 blocks awaiting data from P_1.
2. P_1 marks the start time.
3. P_1 sends token to P_2.
4. P_1 attempts to read a response token from P_2. This induces a context switch.
5. P_2 is scheduled and receives the token.
6. P_2 sends a response token to P_1.
7. P_2 attempts read a response token from P_1. This induces a context switch.
8. P_1 is scheduled and receives the token.
9. P_1 marks the end time.

The key is that the delivery of a data token induces a context switch. Let T_d and T_r be the time it takes to deliver and receive a data token, respectively, and let T_c be the amount of time spent in a context switch. At step 2, P_1 records the timestamp of the delivery of the token, and at step 9, it records the timestamp of the response. The amount of time elapsed, T, between these events may be expressed by:

$$T = 2 * (T_d + T_c + T_r)$$

This formula arises because of the following events: P_1 sends a token (3), the CPU context switches (4), P_2 receives it (5). P_2 then sends the response token (6), the CPU context switches (7), and finally P_1 receives it (8).

P_1 will be able to easily compute T, since this is just the time between events 3 and 8. So, to solve for T_c, we must first determine the value of $T_d + T_r$.

How can we do this? We can do this by measuring the length of time it takes P_1 to send and receive a token to itself. This will not induce a context switch since P_1 is running on the CPU at the time it sent the token and will not block to receive it.

The game is played a number of iterations to average out any variability in the elapsed time between steps 2 and 9 that may result from unexpected kernel interrupts and additional kernel threads contending for the CPU. We select the smallest observed context switch time as our final answer.

However, all we can ultimately say that this is an approximation which depends on the underlying system. For example, we make the assumption that P_2 is selected to run once a data token becomes available. However, this is dependent on the implementation of the task scheduler and we cannot make any guarantees.

That's okay; it's important in an interview to recognize when your solution might not be perfect.

15.3 Dining Philosophers: In the famous dining philosophers problem, a bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat, and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.

pg 180

SOLUTION

First, let's implement a simple simulation of the dining philosophers problem in which we don't concern ourselves with deadlocks. We can implement this solution by having `Philosopher` extend `Thread`, and `Chopstick` call `lock.lock()` when it is picked up and `lock.unlock()` when it is put down.

```
1  class Chopstick {
2    private Lock lock;
3
4    public Chopstick() {
5      lock = new ReentrantLock();
6    }
7
8    public void pickUp() {
9      void lock.lock();
10   }
11
12   public void putDown() {
13     lock.unlock();
14   }
15 }
16
17 class Philosopher extends Thread {
18   private int bites = 10;
19   private Chopstick left, right;
20
21   public Philosopher(Chopstick left, Chopstick right) {
22     this.left = left;
23     this.right = right;
24   }
```

```
25
26    public void eat() {
27       pickUp();
28       chew();
29       putDown();
30    }
31
32    public void pickUp() {
33       left.pickUp();
34       right.pickUp();
35    }
36
37    public void chew() { }
38
39    public void putDown() {
40       right.putDown();
41       left.putDown();
42    }
43
44    public void run() {
45       for (int i = 0; i < bites; i++) {
46          eat();
47       }
48    }
49 }
```

Running the above code may lead to a deadlock if all the philosophers have a left chopstick and are waiting for the right one.

Solution #1: All or Nothing

To prevent deadlocks, we can implement a strategy where a philosopher will put down his left chopstick if he is unable to obtain the right one.

```
1  public class Chopstick {
2     /* same as before */
3
4     public boolean pickUp() {
5        return lock.tryLock();
6     }
7  }
8
9  public class Philosopher extends Thread {
10    /* same as before */
11
12    public void eat() {
13       if (pickUp()) {
14          chew();
15          putDown();
16       }
17    }
18
19    public boolean pickUp() {
20       /* attempt to pick up */
21       if (!left.pickUp()) {
22          return false;
23       }
24       if (!right.pickUp()) {
```

```
25          left.putDown();
26          return false;
27       }
28       return true;
29    }
30 }
```

In the above code, we need to be sure to release the left chopstick if we can't pick up the right one—and to not call `putDown()` on the chopsticks if we never had them in the first place.

One issue with this is that if all the philosophers were perfectly synchronized, they could simultaneously pick up their left chopstick, be unable to pick up the right one, and then put back down the left one—only to have the process repeated again.

Solution #2: Prioritized Chopsticks

Alternatively, we can label the chopsticks with a number from `0` to `N - 1`. Each philosopher attempts to pick up the lower numbered chopstick first. This essentially means that each philosopher goes for the left chopstick before right one (assuming that's the way you labeled it), except for the last philosopher who does this in reverse. This will break the cycle.

```
1  public class Philosopher extends Thread {
2     private int bites = 10;
3     private Chopstick lower, higher;
4     private int index;
5     public Philosopher(int i, Chopstick left, Chopstick right) {
6        index = i;
7        if (left.getNumber() < right.getNumber()) {
8           this.lower = left;
9           this.higher = right;
10       } else {
11          this.lower = right;
12          this.higher = left;
13       }
14    }
15
16    public void eat() {
17       pickUp();
18       chew();
19       putDown();
20    }
21
22    public void pickUp() {
23       lower.pickUp();
24       higher.pickUp();
25    }
26
27    public void chew() { ... }
28
29    public void putDown() {
30       higher.putDown();
31       lower.putDown();
32    }
33
34    public void run() {
35       for (int i = 0; i < bites; i++) {
36          eat();
```

```
37        }
38     }
39  }
40
41  public class Chopstick {
42     private Lock lock;
43     private int number;
44
45     public Chopstick(int n) {
46        lock = new ReentrantLock();
47        this.number = n;
48     }
49
50     public void pickUp() {
51        lock.lock();
52     }
53
54     public void putDown() {
55        lock.unlock();
56     }
57
58     public int getNumber() {
59        return number;
60     }
61  }
```

With this solution, a philosopher can never hold the larger chopstick without holding the smaller one. This prevents the ability to have a cycle, since a cycle means that a higher chopstick would "point" to a lower one.

15.4 Deadlock-Free Class: Design a class which provides a lock only if there are no possible deadlocks.

pg 180

SOLUTION

There are several common ways to prevent deadlocks. One of the popular ways is to require a process to declare upfront what locks it will need. We can then verify if a deadlock would be created by issuing these locks, and we can fail if so.

With these constraints in mind, let's investigate how we can detect deadlocks. Suppose this was the order of locks requested:

```
A = {1, 2, 3, 4}
B = {1, 3, 5}
C = {7, 5, 9, 2}
```

This may create a deadlock because we could have the following scenario:

```
A locks 2, waits on 3
B locks 3, waits on 5
C locks 5, waits on 2
```

We can think about this as a graph, where 2 is connected to 3, 3 is connected to 5, and 5 is connected to 2. A deadlock is represented by a cycle. An edge `(w, v)` exists in the graph if a process declares that it will request lock `v` immediately after lock `w`. For the earlier example, the following edges would exist in the graph: `(1, 2)`, `(2, 3)`, `(3, 4)`, `(1, 3)`, `(3, 5)`, `(7, 5)`, `(5, 9)`, `(9, 2)`. The "owner" of the edge does not matter.

This class will need a `declare` method, which threads and processes will use to declare what order they will request resources in. This declare method will iterate through the declare order, adding each contiguous pair of elements (v, w) to the graph. Afterwards, it will check to see if any cycles have been created. If any cycles have been created, it will backtrack, removing these edges from the graph, and then exit.

We have one final component to discuss: how do we detect a cycle? We can detect a cycle by doing a depth-first search through each connected component (i.e., each connected part of the graph). Complex algorithms exist to find all the connected components of a graph, but our work in this problem does not require this degree of complexity.

We know that if a cycle was created, one of our new edges must be to blame. Thus, as long as our depth-first search touches all of these edges at some point, then we know that we have fully searched for a cycle.

The pseudocode for this special case cycle detection looks like this:

```
boolean checkForCycle(locks[] locks) {
  touchedNodes = hash table(lock -> boolean)
  initialize touchedNodes to false for each lock in locks
  for each (lock x in process.locks) {
    if (touchedNodes[x] == false) {
      if (hasCycle(x, touchedNodes)) {
        return true;
      }
    }
  }
  return false;
}

boolean hasCycle(node x, touchedNodes) {
  touchedNodes[r] = true;
  if (x.state == VISITING) {
    return true;
  } else if (x.state == FRESH) {
    ... (see full code below)
  }
}
```

In the above code, note that we may do several depth-first searches, but `touchedNodes` is only initialized once. We iterate until all the values in `touchedNodes` are false.

The code below provides further details. For simplicity, we assume that all locks and processes (owners) are ordered sequentially.

```
class LockFactory {
  private static LockFactory instance;

  private int numberOfLocks = 5; /* default */
  private LockNode[] locks;

  /* Maps from a process or owner to the order that the owner claimed it would
   * call the locks in */
  private HashMap<Integer, LinkedList<LockNode>> lockOrder;

  private LockFactory(int count) { ... }
  public static LockFactory getInstance() { return instance; }

  public static synchronized LockFactory initialize(int count) {
    if (instance == null) instance = new LockFactory(count);
```

```
16      return instance;
17    }
18
19    public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes,
20                            int[] resourcesInOrder) {
21      /* check for a cycle */
22      for (int resource : resourcesInOrder) {
23        if (touchedNodes.get(resource) == false) {
24          LockNode n = locks[resource];
25          if (n.hasCycle(touchedNodes)) {
26            return true;
27          }
28        }
29      }
30      return false;
31    }
32
33    /* To prevent deadlocks, force the processes to declare upfront what order they
34     * will need the locks in. Verify that this order does not create a deadlock (a
35     * cycle in a directed graph) */
36    public boolean declare(int ownerId, int[] resourcesInOrder) {
37      HashMap<Integer, Boolean> touchedNodes = new HashMap<Integer, Boolean>();
38
39      /* add nodes to graph */
40      int index = 1;
41      touchedNodes.put(resourcesInOrder[0], false);
42      for (index = 1; index < resourcesInOrder.length; index++) {
43        LockNode prev = locks[resourcesInOrder[index - 1]];
44        LockNode curr = locks[resourcesInOrder[index]];
45        prev.joinTo(curr);
46        touchedNodes.put(resourcesInOrder[index], false);
47      }
48
49      /* if we created a cycle, destroy this resource list and return false */
50      if (hasCycle(touchedNodes, resourcesInOrder)) {
51        for (int j = 1; j < resourcesInOrder.length; j++) {
52          LockNode p = locks[resourcesInOrder[j - 1]];
53          LockNode c = locks[resourcesInOrder[j]];
54          p.remove(c);
55        }
56        return false;
57      }
58
59      /* No cycles detected. Save the order that was declared, so that we can
60       * verify that the process is really calling the locks in the order it said
61       * it would. */
62      LinkedList<LockNode> list = new LinkedList<LockNode>();
63      for (int i = 0; i < resourcesInOrder.length; i++) {
64        LockNode resource = locks[resourcesInOrder[i]];
65        list.add(resource);
66      }
67      lockOrder.put(ownerId, list);
68
69      return true;
70    }
71
```

```
72    /* Get the lock, verifying first that the process is really calling the locks in
73     * the order it said it would. */
74    public Lock getLock(int ownerId, int resourceID) {
75       LinkedList<LockNode> list = lockOrder.get(ownerId);
76       if (list == null) return null;
77
78       LockNode head = list.getFirst();
79       if (head.getId() == resourceID) {
80          list.removeFirst();
81          return head.getLock();
82       }
83       return null;
84    }
85 }
86
87 public class LockNode {
88    public enum VisitState { FRESH, VISITING, VISITED };
89
90    private ArrayList<LockNode> children;
91    private int lockId;
92    private Lock lock;
93    private int maxLocks;
94
95    public LockNode(int id, int max) { ... }
96
97    /* Join "this" to "node", checking that it doesn't create a cycle */
98    public void joinTo(LockNode node) { children.add(node); }
99    public void remove(LockNode node) { children.remove(node); }
100
101   /* Check for a cycle by doing a depth-first-search. */
102   public boolean hasCycle(HashMap<Integer, Boolean> touchedNodes) {
103      VisitState[] visited = new VisitState[maxLocks];
104      for (int i = 0; i < maxLocks; i++) {
105         visited[i] = VisitState.FRESH;
106      }
107      return hasCycle(visited, touchedNodes);
108   }
109
110   private boolean hasCycle(VisitState[] visited,
111                            HashMap<Integer, Boolean> touchedNodes) {
112      if (touchedNodes.containsKey(lockId)) {
113         touchedNodes.put(lockId, true);
114      }
115
116      if (visited[lockId] == VisitState.VISITING) {
117         /* We looped back to this node while still visiting it, so we know there's
118          * a cycle. */
119         return true;
120      } else if (visited[lockId] == VisitState.FRESH) {
121         visited[lockId] = VisitState.VISITING;
122         for (LockNode n : children) {
123            if (n.hasCycle(visited, touchedNodes)) {
124               return true;
125            }
126         }
127         visited[lockId] = VisitState.VISITED;
```

```
128       }
129       return false;
130    }
131
132    public Lock getLock() {
133       if (lock == null) lock = new ReentrantLock();
134       return lock;
135    }
136
137    public int getId() { return lockId; }
138 }
```

As always, when you see code this complicated and lengthy, you wouldn't be expected to write all of it. More likely, you would be asked to sketch out pseudocode and possibly implement one of these methods.

15.5 Call In Order: Suppose we have the following code:

```
public class Foo {
   public Foo() { ... }
   public void first() { ... }
   public void second() { ... }
   public void third() { ... }
}
```

The same instance of `Foo` will be passed to three different threads. `ThreadA` will call first, `threadB` will call second, and `threadC` will call third. Design a mechanism to ensure that `first` is called before `second` and `second` is called before `third`.

pg 180

SOLUTION

The general logic is to check if `first()` has completed before executing `second()`, and if `second()` has completed before calling `third()`. Because we need to be very careful about thread safety, simple `boolean` flags won't do the job.

What about using a `lock` to do something like the below code?

```
1  public class FooBad {
2     public int pauseTime = 1000;
3     public ReentrantLock lock1, lock2;
4
5     public FooBad() {
6        try {
7           lock1 = new ReentrantLock();
8           lock2 = new ReentrantLock();
9
10          lock1.lock();
11          lock2.lock();
12       } catch (...) { ... }
13    }
14
15    public void first() {
16       try {
17          ...
18          lock1.unlock(); // mark finished with first()
19       } catch (...) { ... }
20    }
```

```
21
22    public void second() {
23       try {
24          lock1.lock(); // wait until finished with first()
25          lock1.unlock();
26          ...
27
28          lock2.unlock(); // mark finished with second()
29       } catch (...) { ... }
30    }
31
32    public void third() {
33       try {
34          lock2.lock(); // wait until finished with third()
35          lock2.unlock();
36          ...
37       } catch (...) { ... }
38    }
39 }
```

This code won't actually quite work due to the concept of *lock ownership*. One thread is actually performing the lock (in the FooBad constructor), but different threads attempt to unlock the locks. This is not allowed, and your code will raise an exception. A lock in Java is owned by the same thread which locked it.

Instead, we can replicate this behavior with semaphores. The logic is identical.

```
1  public class Foo {
2     public Semaphore sem1, sem2;
3
4     public Foo() {
5        try {
6           sem1 = new Semaphore(1);
7           sem2 = new Semaphore(1);
8
9           sem1.acquire();
10          sem2.acquire();
11       } catch (...) { ... }
12    }
13
14    public void first() {
15       try {
16          ...
17          sem1.release();
18       } catch (...) { ... }
19    }
20
21    public void second() {
22       try {
23          sem1.acquire();
24          sem1.release();
25          ...
26          sem2.release();
27       } catch (...) { ... }
28    }
29
30    public void third() {
31       try {
32          sem2.acquire();
```

```
33          sem2.release();
34          ...
35        } catch (...) { ... }
36    }
37  }
```

15.6 Synchronized Methods: You are given a class with `synchronized` method A and a normal method B. If you have two threads in one instance of a program, can they both execute A at the same time? Can they execute A and B at the same time?

pg 180

SOLUTION

By applying the word `synchronized` to a method, we ensure that two threads cannot execute synchronized methods *on the same object instance* at the same time.

So, the answer to the first part really depends. If the two threads have the same instance of the object, then no, they cannot simultaneously execute method A. However, if they have different instances of the object, then they can.

Conceptually, you can see this by considering locks. A synchronized method applies a "lock" on *all* synchronized methods in that instance of the object. This blocks other threads from executing synchronized methods within that instance.

In the second part, we're asked if `thread1` can execute synchronized method A while `thread2` is executing non-synchronized method B. Since B is not synchronized, there is nothing to block `thread1` from executing A while `thread2` is executing B. This is true regardless of whether `thread1` and `thread2` have the same instance of the object.

Ultimately, the key concept to remember is that only one synchronized method can be in execution per instance of that object. Other threads can execute non-synchronized methods on that instance, or they can execute any method on a different instance of the object.

15.7 FizzBuzz: In the classic problem FizzBuzz, you are told to print the numbers from 1 to n. However, when the number is divisible by 3, print "Fizz". When it is divisible by 5, print "Buzz". When it is divisible by 3 and 5, print "FizzBuzz". In this problem, you are asked to do this in a multithreaded way. Implement a multithreaded version of FizzBuzz with four threads. One thread checks for divisibility of 3 and prints "Fizz". Another thread is responsible for divisibility of 5 and prints "Buzz". A third thread is responsible for divisibility of 3 and 5 and prints "FizzBuzz". A fourth thread does the numbers.

pg 180

SOLUTION

Let's start off with implementing a single threaded version of FizzBuzz.

Single Threaded

Although this problem (in the single threaded version) shouldn't be hard, a lot of candidates overcomplicate it. They look for something "beautiful" that reuses the fact that the divisible by 3 and 5 case ("FizzBuzz") seems to resemble the individual cases ("Fizz" and "Buzz").

In actuality, the best way to do it, considering readability and efficiency, is just the straightforward way.

```
1  void fizzbuzz(int n) {
```

```
2     for (int i = 1; i <= n; i++) {
3       if (i % 3 == 0 && i % 5 == 0) {
4         System.out.println("FizzBuzz");
5       } else if (i % 3 == 0) {
6         System.out.println("Fizz");
7       } else if (i % 5 == 0) {
8         System.out.println("Buzz");
9       } else {
10        System.out.println(i);
11      }
12    }
13  }
```

The primary thing to be careful of here is the order of the statements. If you put the check for divisibility by 3 before the check for divisibility by 3 and 5, it won't print the right thing.

Multithreaded

To do this multithreaded, we want a structure that looks something like this:

FizzBuzz Thread	Fizz Thread

if i div by 3 && 5
  print FizzBuzz
  increment i
repeat until i > n
	if i div by only 3
	  print Fizz
	  increment i
	repeat until i > n

Buzz Thread	Number Thread

if i div by only 5
  print Buzz
  increment i
repeat until i > n
	if i not div by 3 or 5
	  print i
	  increment i
	repeat until i > n

The code for this will look something like:

```
1   while (true) {
2     if (current > max) {
3       return;
4     }
5     if (/* divisibility test */) {
6       System.out.println(/* print something */);
7       current++;
8     }
9   }
```

We'll need to add some synchronization in the loop. Otherwise, the value of `current` could change between lines 2 - 4 and lines 5 - 8, and we can inadvertently exceed the intended bounds of the loop. Additionally, incrementing is not thread-safe.

To actually implement this concept, there are many possibilities. One possibility is to have four entirely separate thread classes that share a reference to the current variable (which can be wrapped in an object).

The loop for each thread is substantially similar. They just have different target values for the divisibility checks, and different print values.

	FizzBuzz	Fizz	Buzz	Number
current % 3 == 0	true	true	false	false
current % 5 == 0	true	false	true	false
to print	FizzBuzz	Fizz	Buzz	current

For the most part, this can be handled by taking in "target" parameters and the value to print. The output for the `Number` thread needs to be overwritten, though, as it's not a simple, fixed string.

We can implement a `FizzBuzzThread` class which handles most of this. A `NumberThread` class can extend `FizzBuzzThread` and override the `print` method.

```
1  Thread[] threads = {new FizzBuzzThread(true, true, n, "FizzBuzz"),
2                      new FizzBuzzThread(true, false, n, "Fizz"),
3                      new FizzBuzzThread(false, true, n, "Buzz"),
4                      new NumberThread(false, false, n)};
5  for (Thread thread : threads) {
6    thread.start();
7  }
8
9  public class FizzBuzzThread extends Thread {
10   private static Object lock = new Object();
11   protected static int current = 1;
12   private int max;
13   private boolean div3, div5;
14   private String toPrint;
15
16   public FizzBuzzThread(boolean div3, boolean div5, int max, String toPrint) {
17     this.div3 = div3;
18     this.div5 = div5;
19     this.max = max;
20     this.toPrint = toPrint;
21   }
22
23   public void print() {
24     System.out.println(toPrint);
25   }
26
27   public void run() {
28     while (true) {
29       synchronized (lock) {
30         if (current > max) {
31           return;
32         }
33
34         if ((current % 3 == 0) == div3 &&
35             (current % 5 == 0) == div5) {
36           print();
37           current++;
38         }
39       }
40     }
41   }
42 }
43
44 public class NumberThread extends FizzBuzzThread {
```

```
45    public NumberThread(boolean div3, boolean div5, int max) {
46       super(div3, div5, max, null);
47    }
48
49    public void print() {
50       System.out.println(current);
51    }
52 }
```

Observe that we need to put the comparison of `current` and `max` before the if statement, to ensure the value will only get printed when `current` is less than or equal to `max`.

Alternatively, if we're working in a language which supports this (Java 8 and many other languages do), we can pass in a `validate` method and a `print` method as parameters.

```
1  int n = 100;
2  Thread[] threads = {
3     new FBThread(i -> i % 3 == 0 && i % 5 == 0, i -> "FizzBuzz", n),
4     new FBThread(i -> i % 3 == 0 && i % 5 != 0, i -> "Fizz", n),
5     new FBThread(i -> i % 3 != 0 && i % 5 == 0, i -> "Buzz", n),
6     new FBThread(i -> i % 3 != 0 && i % 5 != 0, i -> Integer.toString(i), n)};
7  for (Thread thread : threads) {
8     thread.start();
9  }
10
11 public class FBThread extends Thread {
12    private static Object lock = new Object();
13    protected static int current = 1;
14    private int max;
15    private Predicate<Integer> validate;
16    private Function<Integer, String> printer;
17    int x = 1;
18
19    public FBThread(Predicate<Integer> validate,
20                    Function<Integer, String> printer, int max) {
21       this.validate = validate;
22       this.printer = printer;
23       this.max = max;
24    }
25
26    public void run() {
27       while (true) {
28          synchronized (lock) {
29             if (current > max) {
30                return;
31             }
32             if (validate.test(current)) {
33                System.out.println(printer.apply(current));
34                current++;
35             }
36          }
37       }
38    }
39 }
```

There are of course many other ways of implementing this as well.

16

Solutions to Moderate

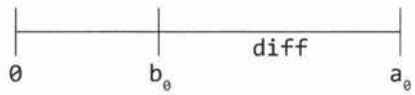
16.1 Number Swapper: Write a function to swap a number in place (that is, without temporary variables).

pg 181

SOLUTION

This is a classic interview problem, and it's a reasonably straightforward one. We'll walk through this using a_0 to indicate the original value of a and b_0 to indicate the original value of b. We'll also use `diff` to indicate the value of a_0 - b_0.

Let's picture these on a number line for the case where a > b.



First, we briefly set a to `diff`, which is the right side of the above number line. Then, when we add b and `diff` (and store that value in b), we get a_0. We now have b = a_0 and a = `diff`. All that's left to do is to set a equal to a_0 - `diff`, which is just b - a.

The code below implements this.

```
1  // Example for a = 9, b = 4
2  a = a - b; // a = 9 - 4 = 5
3  b = a + b; // b = 5 + 4 = 9
4  a = b - a; // a = 9 - 5
```

We can implement a similar solution with bit manipulation. The benefit of this solution is that it works for more data types than just integers.

```
1  // Example for a = 101 (in binary) and b = 110
2  a = a^b; // a = 101^110 = 011
3  b = a^b; // b = 011^110 = 101
4  a = a^b; // a = 011^101 = 110
```

This code works by using XORs. The easiest way to see how this works is by focusing on a specific bit. If we can correctly swap two bits, then we know the entire operation works correctly.

Let's take two bits, x and y, and walk through this line by line.

1. x = x ^ y

   This line essentially checks if x and y have different values. It will result in 1 if and only if x != y.

2. y = x ^ y

Or: `y = {0 if originally same, 1 if different} ^ {original y}`

Observe that XORing a bit with 1 always flips the bit, whereas XORing with 0 will never change it.

Therefore, if we do `y = 1 ^ {original y}` when `x != y`, then y will be flipped and therefore have x's original value.

Otherwise, if `x == y`, then we do `y = 0 ^ {original y}` and the value of y does not change.

Either way, y will be equal to the original value of x.

3. `x = x ^ y`

   Or: `x = {0 if originally same, 1 if different} ^ {original x}`

   At this point, y is equal to the original value of x. This line is essentially equivalent to the line above it, but for different variables.

   If we do `x = 1 ^ {original x}` when the values are different, x will be flipped.

   If we do `x = 0 ^ {original x}` when the values are the same, x will not be changed.

This operation happens for each bit. Since it correctly swaps each bit, it will correctly swap the entire number.

16.2 Word Frequencies: Design a method to find the frequency of occurrences of any given word in a book. What if we were running this algorithm multiple times?

pg 181

SOLUTION

Let's start with the simple case.

Solution: Single Query

In this case, we simply go through the book, word by word, and count the number of times that a word appears. This will take $O(n)$ time. We know we can't do better than that since we must look at every word in the book.

```
1  int getFrequency(String[] book, String word) {
2    word = word.trim().toLowerCase();
3    int count = 0;
4    for (String w : book) {
5      if (w.trim().toLowerCase().equals(word)) {
6        count++;
7      }
8    }
9    return count;
10 }
```

We have also converted the string to lowercase and trimmed it. You can discuss with your interviewer if this is necessary (or even desired).

Solution: Repetitive Queries

If we're doing the operation repeatedly, then we can probably afford to take some time and extra memory to do pre-processing on the book. We can create a hash table which maps from a word to its frequency. The frequency of any word can be easily looked up in $O(1)$ time. The code for this is below.

```
1  HashMap<String, Integer> setupDictionary(String[] book) {
2    HashMap<String, Integer> table =
```

```
3         new HashMap<String, Integer>();
4       for (String word : book) {
5         word = word.toLowerCase();
6         if (word.trim() != "") {
7           if (!table.containsKey(word)) {
8             table.put(word, 0);
9           }
10          table.put(word, table.get(word) + 1);
11        }
12      }
13      return table;
14  }
15
16  int getFrequency(HashMap<String, Integer> table, String word) {
17    if (table == null || word == null) return -1;
18    word = word.toLowerCase();
19    if (table.containsKey(word)) {
20      return table.get(word);
21    }
22    return 0;
23  }
```

Note that a problem like this is actually relatively easy. Thus, the interviewer is going to be looking heavily at how careful you are. Did you check for error conditions?

16.3 Intersection: Given two straight line segments (represented as a start point and an end point), compute the point of intersection, if any.

pg 181

SOLUTION

We first need to think about what it means for two line segments to intersect.

For two infinite lines to intersect, they only have to have different slopes. If they have the same slope, then they must be the exact same line (same y-intercept). That is:

```
slope 1 != slope 2
OR
slope 1 == slope 2 AND intersect 1 == intersect 2
```

For two straight lines to intersect, the condition above must be true, *plus* the point of intersection must be within the ranges of each line segment.

```
extended infinite segments intersect
AND
intersection is within line segment 1 (x and y coordinates)
AND
intersection is within line segment 2 (x and y coordinates)
```

What if the two segments represent the same infinite line? In this case, we have to ensure that some portion of their segments overlap. If we order the line segments by their x locations (start is before end, point 1 is before point 2), then an intersection occurs only if:

```
Assume:
    start1.x < start2.x && start1.x < end1.x && start2.x < end2.x
Then intersection occurs if:
    start2 is between start1 and end1
```

We can now go ahead and implement this algorithm.

```
Point intersection(Point start1, Point end1, Point start2, Point end2) {
   /* Rearranging these so that, in order of x values: start is before end and
    * point 1 is before point 2. This will make some of the later logic simpler. */
   if (start1.x > end1.x) swap(start1, end1);
   if (start2.x > end2.x) swap(start2, end2);
   if (start1.x > start2.x) {
      swap(start1, start2);
      swap(end1, end2);
   }

   /* Compute lines (including slope and y-intercept). */
   Line line1 = new Line(start1, end1);
   Line line2 = new Line(start2, end2);

   /* If the lines are parallel, they intercept only if they have the same y
    * intercept and start 2 is on line 1. */
   if (line1.slope == line2.slope) {
      if (line1.yintercept == line2.yintercept &&
          isBetween(start1, start2, end1)) {
         return start2;
      }
      return null;
   }

   /* Get intersection coordinate. */
   double x = (line2.yintercept - line1.yintercept) / (line1.slope - line2.slope);
   double y = x * line1.slope + line1.yintercept;
   Point intersection = new Point(x, y);

   /* Check if within line segment range. */
   if (isBetween(start1, intersection, end1) &&
       isBetween(start2, intersection, end2)) {
      return intersection;
   }
   return null;
}

/* Checks if middle is between start and end. */
boolean isBetween(double start, double middle, double end) {
   if (start > end) {
      return end <= middle && middle <= start;
   } else {
      return start <= middle && middle <= end;
   }
}

/* Checks if middle is between start and end. */
boolean isBetween(Point start, Point middle, Point end) {
   return isBetween(start.x, middle.x, end.x) &&
          isBetween(start.y, middle.y, end.y);
}

/* Swap coordinates of point one and two. */
void swap(Point one, Point two) {
   double x = one.x;
   double y = one.y;
```

```
57    one.setLocation(two.x, two.y);
58    two.setLocation(x, y);
59  }
60
61  public class Line {
62    public double slope, yintercept;
63
64    public Line(Point start, Point end) {
65      double deltaY = end.y - start.y;
66      double deltaX = end.x - start.x;
67      slope = deltaY / deltaX; // Will be Infinity (not exception) when deltaX = 0
68      yintercept = end.y - slope * end.x;
69    }
70
71  public class Point {
72    public double x, y;
73    public Point(double x, double y) {
74      this.x = x;
75      this.y = y;
76    }
77
78    public void setLocation(double x, double y) {
79      this.x = x;
80      this.y = y;
81    }
82  }
```

For simplicity and compactness (it really makes the code easier to read), we've chosen to make the variables within `Point` and `Line` `public`. You can discuss with your interviewer the advantages and disadvantages of this choice.

16.4 Tic Tac Win: Design an algorithm to figure out if someone has won a game of tic-tac-toe.

pg 181

SOLUTION

At first glance, this problem seems really straightforward. We're just checking a tic-tac-toe board; how hard could it be? It turns out that the problem is a bit more complex, and there is no single "perfect" answer. The optimal solution depends on your preferences.

There are a few major design decisions to consider:

1. Will `hasWon` be called just once or many times (for instance, as part of a tic-tac-toe website)? If the latter is the case, we may want to add pre-processing time to optimize the runtime of `hasWon`.
2. Do we know the last move that was made?
3. Tic-tac-toe is usually on a 3x3 board. Do we want to design for just that, or do we want to implement it as an `NxN` solution?
4. In general, how much do we prioritize compactness of code versus speed of execution vs. clarity of code? Remember: The most efficient code may not always be the best. Your ability to understand and maintain the code matters, too.

Solution #1: If hasWon is called many times

There are only 3^9, or about 20,000, tic-tac-toe boards (assuming a 3x3 board). Therefore, we can represent our tic-tac-toe board as an `int`, with each digit representing a piece (0 means Empty, 1 means Red, 2 means Blue). We set up a hash table or array in advance with all possible boards as keys and the value indicating who has won. Our function then is simply this:

```
1  Piece hasWon(int board) {
2    return winnerHashtable[board];
3  }
```

To convert a board (represented by a char array) to an `int`, we can use what is essentially a "base 3" representation. Each board is represented as $3^0v_0 + 3^1v_1 + 3^2v_2 + \ldots + 3^8v_8$, where v_i is a 0 if the space is empty, a 1 if it's a "blue spot" and a 2 if it's a "red spot."

```
1  enum Piece { Empty, Red, Blue };
2
3  int convertBoardToInt(Piece[][] board) {
4    int sum = 0;
5    for (int i = 0; i < board.length; i++) {
6      for (int j = 0; j < board[i].length; j++) {
7        /* Each value in enum has an integer associated with it. We
8         * can just use that. */
9        int value = board[i][j].ordinal();
10       sum = sum * 3 + value;
11     }
12   }
13   return sum;
14 }
```

Now looking up the winner of a board is just a matter of looking it up in a hash table.

Of course, if we need to convert a board into this format every time we want to check for a winner, we haven't saved ourselves any time compared with the other solutions. But, if we can store the board this way from the very beginning, then the lookup process will be very efficient.

Solution #2: If we know the last move

If we know the very last move that was made (and we've been checking for a winner up until now), then we only need to check the row, column, and diagonal that overlaps with this position.

```
1  Piece hasWon(Piece[][] board, int row, int column) {
2    if (board.length != board[0].length) return Piece.Empty;
3
4    Piece piece = board[row][column];
5
6    if (piece == Piece.Empty) return Piece.Empty;
7
8    if (hasWonRow(board, row) || hasWonColumn(board, column)) {
9      return piece;
10   }
11
12   if (row == column && hasWonDiagonal(board, 1)) {
13     return piece;
14   }
15
16   if (row == (board.length - column - 1) && hasWonDiagonal(board, -1)) {
17     return piece;
18   }
```

```
19
20    return Piece.Empty;
21  }
22
23  boolean hasWonRow(Piece[][] board, int row) {
24    for (int c = 1; c < board[row].length; c++) {
25      if (board[row][c] != board[row][0]) {
26        return false;
27      }
28    }
29    return true;
30  }
31
32  boolean hasWonColumn(Piece[][] board, int column) {
33    for (int r = 1; r < board.length; r++) {
34      if (board[r][column] != board[0][column]) {
35        return false;
36      }
37    }
38    return true;
39  }
40
41  boolean hasWonDiagonal(Piece[][] board, int direction) {
42    int row = 0;
43    int column = direction == 1 ? 0 : board.length - 1;
44    Piece first = board[0][column];
45    for (int i = 0; i < board.length; i++) {
46      if (board[row][column] != first) {
47        return false;
48      }
49      row += 1;
50      column += direction;
51    }
52    return true;
53  }
```

There is actually a way to clean up this code to remove some of the duplicated code. We'll see this approach in a later function.

Solution #3: Designing for just a 3x3 board

If we really only want to implement a solution for a 3x3 board, the code is relatively short and simple. The only complex part is trying to be clean and organized, without writing too much duplicated code.

The code below checks each row, column, and diagonal to see if there is a winner.

```
1   Piece hasWon(Piece[][] board) {
2     for (int i = 0; i < board.length; i++) {
3       /* Check Rows */
4       if (hasWinner(board[i][0], board[i][1], board[i][2])) {
5         return board[i][0];
6       }
7
8       /* Check Columns */
9       if (hasWinner(board[0][i], board[1][i], board[2][i])) {
10        return board[0][i];
11      }
```

```
    }

    /* Check Diagonal */
    if (hasWinner(board[0][0], board[1][1], board[2][2])) {
      return board[0][0];
    }

    if (hasWinner(board[0][2], board[1][1], board[2][0])) {
      return board[0][2];
    }

    return Piece.Empty;
  }

  boolean hasWinner(Piece p1, Piece p2, Piece p3) {
    if (p1 == Piece.Empty) {
      return false;
    }
    return p1 == p2 && p2 == p3;
  }
```

This is an okay solution in that it's relatively easy to understand what is going on. The problem is that the values are hard coded. It's easy to accidentally type the wrong indices.

Additionally, it won't be easy to scale this to an NxN board.

Solution #4: Designing for an NxN board

There are a number of ways to implement this on an NxN board.

Nested For-Loops

The most obvious way is through a series of nested for-loops.

```
Piece hasWon(Piece[][] board) {
  int size = board.length;
  if (board[0].length != size) return Piece.Empty;
  Piece first;

  /* Check rows. */
  for (int i = 0; i < size; i++) {
    first = board[i][0];
    if (first == Piece.Empty) continue;
    for (int j = 1; j < size; j++) {
      if (board[i][j] != first) {
        break;
      } else if (j == size - 1) { // Last element
        return first;
      }
    }
  }

  /* Check columns. */
  for (int i = 0; i < size; i++) {
    first = board[0][i];
    if (first == Piece.Empty) continue;
    for (int j = 1; j < size; j++) {
      if (board[j][i] != first) {
```

```
            break;
         } else if (j == size - 1) { // Last element
            return first;
         }
      }
   }

   /* Check diagonals. */
   first = board[0][0];
   if (first != Piece.Empty) {
      for (int i = 1; i < size; i++) {
         if (board[i][i] != first) {
            break;
         } else if (i == size - 1) { // Last element
            return first;
         }
      }
   }

   first = board[0][size - 1];
   if (first != Piece.Empty) {
      for (int i = 1; i < size; i++) {
         if (board[i][size - i - 1] != first) {
            break;
         } else if (i == size - 1) { // Last element
            return first;
         }
      }
   }

   return Piece.Empty;
}
```

This is, to the say the least, pretty ugly. We're doing nearly the same work each time. We should look for a way of reusing the code.

Increment and Decrement Function

One way that we can reuse the code better is to just pass in the values to another function that increments/decrements the rows and columns. The `hasWon` function now just needs the starting position and the amount to increment the row and column by.

```
class Check {
   public int row, column;
   private int rowIncrement, columnIncrement;
   public Check(int row, int column, int rowI, int colI) {
      this.row = row;
      this.column = column;
      this.rowIncrement = rowI;
      this.columnIncrement = colI;
   }

   public void increment() {
      row += rowIncrement;
      column += columnIncrement;
   }

   public boolean inBounds(int size) {
```

```
17        return row >= 0 && column >= 0 && row < size && column < size;
18     }
19  }
20
21  Piece hasWon(Piece[][] board) {
22     if (board.length != board[0].length) return Piece.Empty;
23     int size = board.length;
24
25     /* Create list of things to check. */
26     ArrayList<Check> instructions = new ArrayList<Check>();
27     for (int i = 0; i < board.length; i++) {
28        instructions.add(new Check(0, i, 1, 0));
29        instructions.add(new Check(i, 0, 0, 1));
30     }
31     instructions.add(new Check(0, 0, 1, 1));
32     instructions.add(new Check(0, size - 1, 1, -1));
33
34     /* Check them. */
35     for (Check instr : instructions) {
36        Piece winner = hasWon(board, instr);
37        if (winner != Piece.Empty) {
38              return winner;
39        }
40     }
41     return Piece.Empty;
42  }
43
44  Piece hasWon(Piece[][] board, Check instr) {
45     Piece first = board[instr.row][instr.column];
46     while (instr.inBounds(board.length)) {
47        if (board[instr.row][instr.column] != first) {
48           return Piece.Empty;
49        }
50        instr.increment();
51     }
52     return first;
53  }
```

The Check function is essentially operating as an iterator.

Iterator

Another way of doing it is, of course, to actually build an iterator.

```
1   Piece hasWon(Piece[][] board) {
2      if (board.length != board[0].length) return Piece.Empty;
3      int size = board.length;
4
5      ArrayList<PositionIterator> instructions = new ArrayList<PositionIterator>();
6      for (int i = 0; i < board.length; i++) {
7         instructions.add(new PositionIterator(new Position(0, i), 1, 0, size));
8         instructions.add(new PositionIterator(new Position(i, 0), 0, 1, size));
9      }
10     instructions.add(new PositionIterator(new Position(0, 0), 1, 1, size));
11     instructions.add(new PositionIterator(new Position(0, size - 1), 1, -1, size));
12
13     for (PositionIterator iterator : instructions) {
14        Piece winner = hasWon(board, iterator);
```

```
      if (winner != Piece.Empty) {
        return winner;
      }
    }
    return Piece.Empty;
  }

  Piece hasWon(Piece[][] board, PositionIterator iterator) {
    Position firstPosition = iterator.next();
    Piece first = board[firstPosition.row][firstPosition.column];
    while (iterator.hasNext()) {
      Position position = iterator.next();
      if (board[position.row][position.column] != first) {
        return Piece.Empty;
      }
    }
    return first;
  }

  class PositionIterator implements Iterator<Position> {
    private int rowIncrement, colIncrement, size;
    private Position current;

    public PositionIterator(Position p, int rowIncrement,
                            int colIncrement, int size) {
      this.rowIncrement = rowIncrement;
      this.colIncrement = colIncrement;
      this.size = size;
      current = new Position(p.row - rowIncrement, p.column - colIncrement);
    }

    @Override
    public boolean hasNext() {
      return current.row + rowIncrement < size &&
             current.column + colIncrement < size;
    }

    @Override
    public Position next() {
      current = new Position(current.row + rowIncrement,
                             current.column + colIncrement);
      return current;
    }
  }

  public class Position {
    public int row, column;
    public Position(int row, int column) {
      this.row = row;
      this.column = column;
    }
  }
```

All of this is potentially overkill, but it's worth discussing the options with your interviewer. The point of this problem is to assess your understanding of how to code in a clean and maintainable way.

16.5 Factorial Zeros: Write an algorithm which computes the number of trailing zeros in n factorial.

pg 181

SOLUTION

A simple approach is to compute the factorial, and then count the number of trailing zeros by continuously dividing by ten. The problem with this though is that the bounds of an `int` would be exceeded very quickly. To avoid this issue, we can look at this problem mathematically.

Consider a factorial like 19!:

```
19! = 1*2*3*4*5*6*7*8*9*10*11*12*13*14*15*16*17*18*19
```

A trailing zero is created with multiples of 10, and multiples of 10 are created with pairs of 5-multiples and 2-multiples.

For example, in 19!, the following terms create the trailing zeros:

```
19! = 2 * ... * 5 * ... * 10 * ... * 15 * 16 * ...
```

Therefore, to count the number of zeros, we only need to count the pairs of multiples of 5 and 2. There will always be more multiples of 2 than 5, though, so simply counting the number of multiples of 5 is sufficient.

One "gotcha" here is 15 contributes a multiple of 5 (and therefore one trailing zero), while 25 contributes two (because `25 = 5 * 5`).

There are two different ways to write this code.

The first way is to iterate through all the numbers from 2 through n, counting the number of times that 5 goes into each number.

```
1  /* If the number is a 5 of five, return which power of 5. For example: 5 -> 1,
2   * 25-> 2, etc. */
3  int factorsOf5(int i) {
4     int count = 0;
5     while (i % 5 == 0) {
6        count++;
7        i /= 5;
8     }
9     return count;
10 }
11
12 int countFactZeros(int num) {
13    int count = 0;
14    for (int i = 2; i <= num; i++) {
15       count += factorsOf5(i);
16    }
17    return count;
18 }
```

This isn't bad, but we can make it a little more efficient by directly counting the factors of 5. Using this approach, we would first count the number of multiples of 5 between 1 and n (which is $n/5$), then the number of multiples of 25 ($n/25$), then 125, and so on.

To count how many multiples of `m` are in n, we can just divide n by `m`.

```
1  int countFactZeros(int num) {
2     int count = 0;
3     if (num < 0) {
4        return -1;
5     }
```

```
6      for (int i = 5; num / i > 0; i *= 5) {
7         count += num / i;
8      }
9      return count;
10  }
```

This problem is a bit of a brainteaser, but it can be approached logically (as shown above). By thinking through what exactly will contribute a zero, you can come up with a solution. You should be very clear in your rules upfront so that you can implement it correctly.

16.6 Smallest Difference: Given two arrays of integers, compute the pair of values (one value in each array) with the smallest (non-negative) difference. Return the difference.

EXAMPLE

Input: {1, 3, 15, 11, 2}, {23, 127, 235, 19, 8}

Output: 3. That is, the pair (11, 8).

pg 181

SOLUTION

Let's start first with a brute force solution.

Brute Force

The simple brute force way is to just iterate through all pairs, compute the difference, and compare it to the current minimum difference.

```
1   int findSmallestDifference(int[] array1, int[] array2) {
2      if (array1.length == 0 || array2.length == 0) return -1;
3
4      int min = Integer.MAX_VALUE;
5      for (int i = 0; i < array1.length; i++) {
6         for (int j = 0; j < array2.length; j++) {
7            if (Math.abs(array1[i] - array2[j]) < min) {
8               min = Math.abs(array1[i] - array2[j]);
9            }
10        }
11     }
12     return min;
13  }
```

One minor optimization we could perform from here is to return immediately if we find a difference of zero, since this is the smallest difference possible. However, depending on the input, this might actually be slower.

This will only be faster if there's a pair with difference zero early in the list of pairs. But to add this optimization, we need to execute an additional line of code each time. There's a tradeoff here; it's faster for some inputs and slower for others. Given that it adds complexity in reading the code, it may be best to leave it out.

With or without this "optimization," the algorithm will take $O(AB)$ time.

Optimal

A more optimal approach is to sort the arrays. Once the arrays are sorted, we can find the minimum difference by iterating through the array.

Consider the following two arrays:

```
A: {1, 2, 11, 15}
B: {4, 12, 19, 23, 127, 235}
```

Try the following approach:

1. Suppose a pointer a points to the beginning of A and a pointer b points to the beginning of B. The current difference between a and b is 3. Store this as the `min`.
2. How can we (potentially) make this difference smaller? Well, the value at b is bigger than the value at a, so moving b will only make the difference larger. Therefore, we want to move a.
3. Now a points to 2 and b (still) points to 4. This difference is 2, so we should update `min`. Move a, since it is smaller.
4. Now a points to 11 and b points to 4. Move b.
5. Now a points to 11 and b points to 12. Update `min` to 1. Move b.

And so on.

```
1  int findSmallestDifference(int[] array1, int[] array2) {
2    Arrays.sort(array1);
3    Arrays.sort(array2);
4    int a = 0;
5    int b = 0;
6    int difference = Integer.MAX_VALUE;
7    while (a < array1.length && b < array2.length) {
8      if (Math.abs(array1[a] - array2[b]) < difference) {
9        difference = Math.abs(array1[a] - array2[b]);
10     }
11
12     /* Move smaller value. */
13     if (array1[a] < array2[b]) {
14       a++;
15     } else {
16       b++;
17     }
18   }
19   return difference;
20 }
```

This algorithm takes `O(A log A + B log B)` time to sort and `O(A + B)` time to find the minimum difference. Therefore, the overall runtime is `O(A log A + B log B)`.

16.7 Number Max: Write a method that finds the maximum of two numbers. You should not use if-else or any other comparison operator.

pg 181

SOLUTION

A common way of implementing a `max` function is to look at the sign of `a - b`. In this case, we can't use a comparison operator on this sign, but we *can* use multiplication.

Let k equal the sign of `a - b` such that if `a - b >= 0`, then k is 1. Else, `k = 0`. Let q be the inverse of k.

We can then implement the code as follows:

```
1  /* Flips a 1 to a 0 and a 0 to a 1 */
2  int flip(int bit) {
```

```
3     return 1^bit;
4   }
5
6   /* Returns 1 if a is positive, and 0 if a is negative */
7   int sign(int a) {
8     return flip((a >> 31) & 0x1);
9   }
10
11  int getMaxNaive(int a, int b) {
12    int k = sign(a - b);
13    int q = flip(k);
14    return a * k + b * q;
15  }
```

This code almost works. It fails, unfortunately, when a - b overflows. Suppose, for example, that a is INT_MAX - 2 and b is -15. In this case, a - b will be greater than INT_MAX and will overflow, resulting in a negative value.

We can implement a solution to this problem by using the same approach. Our goal is to maintain the condition where k is 1 when a > b. We will need to use more complex logic to accomplish this.

When does a - b overflow? It will overflow only when a is positive and b is negative, or the other way around. It may be difficult to specially detect the overflow condition, but we *can* detect when a and b have different signs. Note that if a and b have different signs, then we want k to equal `sign(a)`.

The logic looks like:

```
1   if a and b have different signs:
2     // if a > 0, then b < 0, and k = 1.
3     // if a < 0, then b > 0, and k = 0.
4     // so either way, k = sign(a)
5     let k = sign(a)
6   else
7     let k = sign(a - b) // overflow is impossible
```

The code below implements this, using multiplication instead of if-statements.

```
1   int getMax(int a, int b) {
2     int c = a - b;
3
4     int sa = sign(a); // if a >= 0, then 1 else 0
5     int sb = sign(b); // if b >= 0, then 1 else 0
6     int sc = sign(c); // depends on whether or not a - b overflows
7
8     /* Goal: define a value k which is 1 if a > b and 0 if a < b.
9      * (if a = b, it doesn't matter what value k is) */
10
11    // If a and b have different signs, then k = sign(a)
12    int use_sign_of_a = sa ^ sb;
13
14    // If a and b have the same sign, then k = sign(a - b)
15    int use_sign_of_c = flip(sa ^ sb);
16
17    int k = use_sign_of_a * sa + use_sign_of_c * sc;
18    int q = flip(k); // opposite of k
19
20    return a * k + b * q;
21  }
```

Note that for clarity, we split up the code into many different methods and variables. This is certainly not the most compact or efficient way to write it, but it does make what we're doing much cleaner.

16.8 English Int: Given any integer, print an English phrase that describes the integer (e.g., "One Thousand, Two Hundred Thirty Four").

pg 182

SOLUTION

This is not an especially challenging problem, but it is a somewhat tedious one. The key is to be organized in how you approach the problem—and to make sure you have good test cases.

We can think about converting a number like 19,323,984 as converting each of three 3-digit segments of the number, and inserting "thousands" and "millions" in between as appropriate. That is,

```
convert(19,323,984) = convert(19) + " million " + convert(323) + " thousand " +
                      convert(984)
```

The code below implements this algorithm.

```
1  String[] smalls = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
2    "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen",
3    "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
4  String[] tens = {"", "", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy",
5    "Eighty", "Ninety"};
6  String[] bigs = {"", "Thousand", "Million", "Billion"};
7  String hundred = "Hundred";
8  String negative = "Negative";
9
10 String convert(int num) {
11   if (num == 0) {
12     return smalls[0];
13   } else if (num < 0) {
14     return negative + " " + convert(-1 * num);
15   }
16
17   LinkedList<String> parts = new LinkedList<String>();
18   int chunkCount = 0;
19
20   while (num > 0) {
21     if (num % 1000 != 0) {
22       String chunk = convertChunk(num % 1000) + " " + bigs[chunkCount];
23       parts.addFirst(chunk);
24     }
25     num /= 1000; // shift chunk
26     chunkCount++;
27   }
28
29   return listToString(parts);
30 }
31
32 String convertChunk(int number) {
33   LinkedList<String> parts = new LinkedList<String>();
34
35   /* Convert hundreds place */
36   if (number >= 100) {
37     parts.addLast(smalls[number / 100]);
```

```
38        parts.addLast(hundred);
39        number %= 100;
40     }
41
42     /* Convert tens place */
43     if (number >= 10 && number <= 19) {
44        parts.addLast(smalls[number]);
45     } else if (number >= 20) {
46        parts.addLast(tens[number / 10]);
47        number %= 10;
48     }
49
50     /* Convert ones place */
51     if (number >= 1 && number <= 9) {
52        parts.addLast(smalls[number]);
53     }
54
55     return listToString(parts);
56  }
57  /* Convert a linked list of strings to a string, dividing it up with spaces. */
58  String listToString(LinkedList<String> parts) {
59     StringBuilder sb = new StringBuilder();
60     while (parts.size() > 1) {
61        sb.append(parts.pop());
62        sb.append(" ");
63     }
64     sb.append(parts.pop());
65     return sb.toString();
66  }
```

The key in a problem like this is to make sure you consider all the special cases. There are a lot of them.

16.9 Operations: Write methods to implement the multiply, subtract, and divide operations for integers. The results of all of these are integers. Use only the add operator.

pg 182

SOLUTION

The only operation we have to work with is the add operator. In each of these problems, it's useful to think in depth about what these operations really do or how to phrase them in terms of other operations (either add or operations we've already completed).

Subtraction

How can we phrase subtraction in terms of addition? This one is pretty straightforward. The operation `a - b` is the same thing as `a + (-1) * b`. However, because we are not allowed to use the * (multiply) operator, we must implement a `negate` function.

```
1  /* Flip a positive sign to negative or negative sign to pos. */
2  int negate(int a) {
3     int neg = 0;
4     int newSign = a < 0 ? 1 : -1;
5     while (a != 0) {
6        neg += newSign;
7        a += newSign;
8     }
```

```
9     return neg;
10 }
11
12 /* Subtract two numbers by negating b and adding them */
13 int minus(int a, int b) {
14    return a + negate(b);
15 }
```

The negation of the value k is implemented by adding -1 k times. Observe that this will take O(k) time.

If optimizing is something we value here, we can try to get a to zero faster. (For this explanation, we'll assume that a is positive.) To do this, we can first reduce a by 1, then 2, then 4, then 8, and so on. We'll call this value `delta`. We want a to reach exactly zero. When reducing a by the next `delta` would change the sign of a, we reset `delta` back to 1 and repeat the process.

For example:

```
a:     29   28   26   22   14   13   11    7    6    4    0
delta:  -1   -2   -4   -8   -1   -2   -4   -1   -2   -4
```

The code below implements this algorithm.

```
1  int negate(int a) {
2     int neg = 0;
3     int newSign = a < 0 ? 1 : -1;
4     int delta = newSign;
5     while (a != 0) {
6        boolean differentSigns = (a + delta > 0) != (a > 0);
7        if (a + delta != 0 && differentSigns) { // If delta is too big, reset it.
8           delta = newSign;
9        }
10       neg += delta;
11       a += delta;
12       delta += delta; // Double the delta
13    }
14    return neg;
15 }
```

Figuring out the runtime here takes a bit of calculation.

Observe that reducing a by half takes O(log a) work. Why? For each round of "reduce a by half", the absolute values of a and `delta` always add up to the same number. The values of `delta` and a will converge at $^{a}/_{2}$. Since `delta` is being doubled each time, it will take O(log a) steps to reach half of a.

We do O(log a) rounds.

1. Reducing a to $^{a}/_{2}$ takes O(log a) time.
2. Reducing $^{a}/_{2}$ to $^{a}/_{4}$ takes O(log $^{a}/_{2}$) time.
3. Reducing $^{a}/_{4}$ to $^{a}/_{8}$ takes O(log $^{a}/_{4}$) time.

... As so on, for O(log a) rounds.

The runtime therefore is O(log a + log($^{a}/_{2}$) + log($^{a}/_{4}$) + ...), with O(log a) terms in the expression.

Recall two rules of logs:

- log(xy) = log x + log y
- log($^{x}/_{y}$) = log x - log y.

If we apply this to the above expression, we get:

1. $O(\log a + \log(a/2) + \log(a/4) + \ldots)$
2. $O(\log a + (\log a - \log 2) + (\log a - \log 4) + (\log a - \log 8) + \ldots$
3. $O((\log a)*(\log a) - (\log 2 + \log 4 + \log 8 + \ldots + \log a))$ // $O(\log a)$ terms
4. $O((\log a)*(\log a) - (1 + 2 + 3 + \ldots + \log a))$ // computing the values of logs
5. $O((\log a)*(\log a) - \frac{(\log a)(1+\log a)}{2})$ // apply equation for sum of 1 through k
6. $O((\log a)^2)$ // drop second term from step 5

Therefore, the runtime is $O((\log a)^2)$.

This math is considerably more complicated than most people would be able to do (or expected to do) in an interview. You could make a simplification: You do $O(\log a)$ rounds and the longest round takes $O(\log a)$ work. Therefore, as an upper bound, `negate` takes $O((\log a)^2)$ time. In this case, the upper bound happens to be the true time.

There are some faster solutions too. For example, rather than resetting `delta` to 1 at each round, we could change `delta` to its previous value. This would have the effect of `delta` "counting up" by multiples of two, and then "counting down" by multiples of two. The runtime of this approach would be $O(\log a)$. However, this implementation would require a stack, division, or bit shifting—any of which might violate the spirit of the problem. You could certainly discuss those implementations with your interviewer though.

Multiplication

The connection between addition and multiplication is equally straightforward. To multiply a by b, we just add a to itself b times.

```
1  /* Multiply a by b by adding a to itself b times */
2  int multiply(int a, int b) {
3    if (a < b) {
4      return multiply(b, a); // algorithm is faster if b < a
5    }
6    int sum = 0;
7    for (int i = abs(b); i > 0; i = minus(i, 1)) {
8      sum += a;
9    }
10   if (b < 0) {
11     sum = negate(sum);
12   }
13   return sum;
14 }
15
16 /* Return absolute value */
17 int abs(int a) {
18   if (a < 0) {
19     return negate(a);
20   } else {
21     return a;
22   }
23 }
```

The one thing we need to be careful of in the above code is to properly handle multiplication of negative numbers. If b is negative, we need to flip the value of `sum`. So, what this code really does is:

`multiply(a, b) <-- abs(b) * a * (-1 if b < 0).`

We also implemented a simple `abs` function to help.

Division

Of the three operations, division is certainly the hardest. The good thing is that we can use the `multiply`, `subtract`, and `negate` methods now to implement `divide`.

We are trying to compute x where $x = a/b$. Or, to put this another way, find x where $a = bx$. We've now changed the problem into one that can be stated with something we know how to do: multiplication.

We could implement this by multiplying b by progressively higher values, until we reach a. That would be fairly inefficient, particularly given that our implementation of `multiply` involves a lot of adding.

Alternatively, we can look at the equation $a = xb$ to see that we can compute x by adding b to itself repeatedly until we reach a. The number of times we need to do that will equal x.

Of course, a might not be evenly divisible by b, and that's okay. Integer division, which is what we've been asked to implement, is supposed to truncate the result.

The code below implements this algorithm.

```
1  int divide(int a, int b) throws java.lang.ArithmeticException {
2    if (b == 0) {
3      throw new java.lang.ArithmeticException("ERROR");
4    }
5    int absa = abs(a);
6    int absb = abs(b);
7
8    int product = 0;
9    int x = 0;
10   while (product + absb <= absa) { /* don't go past a */
11     product += absb;
12     x++;
13   }
14
15   if ((a < 0 && b < 0) || (a > 0 && b > 0)) {
16     return x;
17   } else {
18     return negate(x);
19   }
20 }
```

In tackling this problem, you should be aware of the following:

- A logical approach of going back to what exactly multiplication and division do comes in handy. Remember that. All (good) interview problems can be approached in a logical, methodical way!
- The interviewer is looking for this sort of logical work-your-way-through-it approach.
- This is a great problem to demonstrate your ability to write clean code—specifically, to show your ability to reuse code. For example, if you were writing this solution and didn't put `negate` in its own method, you should move it into its own method once you see that you'll use it multiple times.
- Be careful about making assumptions while coding. Don't assume that the numbers are all positive or that a is bigger than b.

16.10 Living People: Given a list of people with their birth and death years, implement a method to compute the year with the most number of people alive. You may assume that all people were born between 1900 and 2000 (inclusive). If a person was alive during any portion of that year, they should be included in that year's count. For example, Person (birth = 1908, death = 1909) is included in the counts for both 1908 and 1909.

pg 182

SOLUTION

The first thing we should do is outline what this solution will look like. The interview question hasn't specified the exact form of input. In a real interview, we could ask the interviewer how the input is structured. Alternatively, you can explicitly state your (reasonable) assumptions.

Here, we'll need to make our own assumptions. We will assume that we have an array of simple `Person` objects:

```
1  public class Person {
2    public int birth;
3    public int death;
4    public Person(int birthYear, int deathYear) {
5      birth = birthYear;
6      death = deathYear;
7    }
8  }
```

We could have also given `Person` a `getBirthYear()` and `getDeathYear()` objects. Some would argue that's better style, but for compactness and clarity, we'll just keep the variables public.

The important thing here is to actually use a `Person` object. This shows better style than, say, having an integer array for birth years and an integer array for death years (with an implicit association of `births[i]` and `deaths[i]` being associated with the same person). You don't get a lot of chances to demonstrate great coding style, so it's valuable to take the ones you get.

With that in mind, let's start with a brute force algorithm.

Brute Force

The brute force algorithm falls directly out from the wording of the problem. We need to find the year with the most number of people alive. Therefore, we go through each year and check how many people are alive in that year.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    int maxAlive = 0;
3    int maxAliveYear = min;
4
5    for (int year = min; year <= max; year++) {
6      int alive = 0;
7      for (Person person : people) {
8        if (person.birth <= year && year <= person.death) {
9          alive++;
10       }
11     }
12     if (alive > maxAlive) {
13       maxAlive = alive;
14       maxAliveYear = year;
15     }
16   }
```

```
17
18    return maxAliveYear;
19  }
```

Note that we have passed in the values for the min year (1900) and max year (2000). We shouldn't hard code these values.

The runtime of this is O(RP), where R is the range of years (100 in this case) and P is the number of people.

Slightly Better Brute Force

A slightly better way of doing this is to create an array where we track the number of people born in each year. Then, we iterate through the list of people and increment the array for each year they are alive.

```
1   int maxAliveYear(Person[] people, int min, int max) {
2     int[] years = createYearMap(people, min, max);
3     int best = getMaxIndex(years);
4     return best + min;
5   }
6
7   /* Add each person's years to a year map. */
8   int[] createYearMap(Person[] people, int min, int max) {
9     int[] years = new int[max - min + 1];
10    for (Person person : people) {
11      incrementRange(years, person.birth - min, person.death - min);
12    }
13    return years;
14  }
15
16  /* Increment array for each value between left and right. */
17  void incrementRange(int[] values, int left, int right) {
18    for (int i = left; i <= right; i++) {
19      values[i]++;
20    }
21  }
22
23  /* Get index of largest element in array. */
24  int getMaxIndex(int[] values) {
25    int max = 0;
26    for (int i = 1; i < values.length; i++) {
27      if (values[i] > values[max]) {
28        max = i;
29      }
30    }
31    return max;
32  }
```

Be careful on the size of the array in line 9. If the range of years is 1900 to 2000 inclusive, then that's 101 years, not 100. That is why the array has size `max - min + 1`.

Let's think about the runtime by breaking this into parts.

- We create an R-sized array, where R is the min and max years.
- Then, for P people, we iterate through the years (Y) that the person is alive.
- Then, we iterate through the R-sized array again.

The total runtime is O(PY + R). In the worst case, Y is R and we have done no better than we did in the first algorithm.

More Optimal

Let's create an example. (In fact, an example is really helpful in almost all problems. Ideally, you've already done this.) Each column below is matched, so that the items correspond to the same person. For compactness, we'll just write the last two digits of the year.

```
birth: 12  20  10  01  10  23  13  90  83  75
death: 15  90  98  72  98  82  98  98  99  94
```

It's worth noting that it doesn't really matter whether these years are matched up. Every birth adds a person and every death removes a person.

Since we don't actually need to match up the births and deaths, let's sort both. A sorted version of the years might help us solve the problem.

```
birth: 01  10  10  12  13  20  23  75  83  90
death: 15  72  82  90  94  98  98  98  98  99
```

We can try walking through the years.

- At year 0, no one is alive.
- At year 1, we see one birth.
- At years 2 through 9, nothing happens.
- Let's skip ahead until year 10, when we have two births. We now have three people alive.
- At year 15, one person dies. We are now down to two people alive.
- And so on.

If we walk through the two arrays like this, we can track the number of people alive at each point.

```
1  int maxAliveYear(Person[] people, int min, int max) {
2    int[] births = getSortedYears(people, true);
3    int[] deaths = getSortedYears(people, false);
4
5    int birthIndex = 0;
6    int deathIndex = 0;
7    int currentlyAlive = 0;
8    int maxAlive = 0;
9    int maxAliveYear = min;
10
11   /* Walk through arrays. */
12   while (birthIndex < births.length) {
13     if (births[birthIndex] <= deaths[deathIndex]) {
14       currentlyAlive++; // include birth
15       if (currentlyAlive > maxAlive) {
16         maxAlive = currentlyAlive;
17         maxAliveYear = births[birthIndex];
18       }
19       birthIndex++; // move birth index
20     } else if (births[birthIndex] > deaths[deathIndex]) {
21       currentlyAlive--; // include death
22       deathIndex++; // move death index
23     }
24   }
25
26   return maxAliveYear;
27 }
28
29 /* Copy birth years or death years (depending on the value of copyBirthYear into
```

```
30   * integer array, then sort array. */
31  int[] getSortedYears(Person[] people, boolean copyBirthYear) {
32    int[] years = new int[people.length];
33    for (int i = 0; i < people.length; i++) {
34      years[i] = copyBirthYear ? people[i].birth : people[i].death;
35    }
36    Arrays.sort(years);
37    return years;
38  }
```

There are some very easy things to mess up here.

On line 13, we need to think carefully about whether this should be a less than (<) or a less than or equals (<=). The scenario we need to worry about is that you see a birth and death in the same year. (It doesn't matter whether the birth and death is from the same person.)

When we see a birth and death from the same year, we want to include the birth *before* we include the death, so that we count this person as alive for that year. That is why we use a <= on line 13.

We also need to be careful about where we put the updating of `maxAlive` and `maxAliveYear`. It needs to be after the `currentAlive++`, so that it takes into account the updated total. But it needs to be before `birthIndex++`, or we won't have the right year.

This algorithm will take `O(P log P)` time, where P is the number of people.

More Optimal (Maybe)

Can we optimize this further? To optimize this, we'd need to get rid of the sorting step. We're back to dealing with unsorted values:

```
birth: 12  20  10  01  10  23  13  90  83  75
death: 15  90  98  72  98  82  98  98  99  94
```

Earlier, we had logic that said that a birth is just adding a person and a death is just subtracting a person. Therefore, let's represent the data using the logic:

```
01: +1      10: +1      10: +1      12: +1      13: +1
15: -1      20: +1      23: +1      72: -1      75: +1
82: -1      83: +1      90: +1      90: -1      94: -1
98: -1      98: -1      98: -1      98: -1      99: -1
```

We can create an array of the years, where the value at `array[year]` indicates how the population changed in that year. To create this array, we walk through the list of people and increment when they're born and decrement when they die.

Once we have this array, we can walk through each of the years, tracking the current population as we go (adding the value at `array[year]` each time).

This logic is reasonably good, but we should think about it more. Does it really work?

One edge case we should consider is when a person dies the same year that they're born. The increment and decrement operations will cancel out to give 0 population change. According to the wording of the problem, this person should be counted as living in that year.

In fact, the "bug" in our algorithm is broader than that. This same issue applies to all people. People who die in 1908 shouldn't be removed from the population count until 1909.

There's a simple fix: instead of decrementing `array[deathYear]`, we should decrement `array[deathYear + 1]`.

```
1   int maxAliveYear(Person[] people, int min, int max) {
```

```
    /* Build population delta array. */
    int[] populationDeltas = getPopulationDeltas(people, min, max);
    int maxAliveYear = getMaxAliveYear(populationDeltas);
    return maxAliveYear + min;
}

/* Add birth and death years to deltas array. */
int[] getPopulationDeltas(Person[] people, int min, int max) {
    int[] populationDeltas = new int[max - min + 2];
    for (Person person : people) {
        int birth = person.birth - min;
        populationDeltas[birth]++;

        int death = person.death - min;
        populationDeltas[death + 1]--;
    }
    return populationDeltas;
}

/* Compute running sums and return index with max. */
int getMaxAliveYear(int[] deltas) {
    int maxAliveYear = 0;
    int maxAlive = 0;
    int currentlyAlive = 0;
    for (int year = 0; year < deltas.length; year++) {
        currentlyAlive += deltas[year];
        if (currentlyAlive > maxAlive) {
            maxAliveYear = year;
            maxAlive = currentlyAlive;
        }
    }

    return maxAliveYear;
}
```

This algorithm takes O(R + P) time, where R is the range of years and P is the number of people. Although O(R + P) might be faster than O(P log P) for many expected inputs, you cannot directly compare the speeds to say that one is faster than the other.

16.11 Diving Board: You are building a diving board by placing a bunch of planks of wood end-to-end. There are two types of planks, one of length `shorter` and one of length `longer`. You must use exactly K planks of wood. Write a method to generate all possible lengths for the diving board.

pg 182

SOLUTION

One way to approach this is to think about the choices we make as we're building a diving board. This leads us to a recursive algorithm.

Recursive Solution

For a recursive solution, we can imagine ourselves building a diving board. We make K decisions, each time choosing which plank we will put on next. Once we've put on K planks, we have a complete diving board and we can add this to the list (assuming we haven't seen this length before).

We can follow this logic to write recursive code. Note that we don't need to track the sequence of planks. All we need to know is the current length and the number of planks remaining.

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2    HashSet<Integer> lengths = new HashSet<Integer>();
3    getAllLengths(k, 0, shorter, longer, lengths);
4    return lengths;
5  }
6
7  void getAllLengths(int k, int total, int shorter, int longer,
8                     HashSet<Integer> lengths) {
9    if (k == 0) {
10     lengths.add(total);
11     return;
12   }
13   getAllLengths(k - 1, total + shorter, shorter, longer, lengths);
14   getAllLengths(k - 1, total + longer, shorter, longer, lengths);
15 }
```

We've added each length to a hash set. This will automatically prevent adding duplicates.

This algorithm takes $O(2^K)$ time, since there are two choices at each recursive call and we recurse to a depth of K.

Memoization Solution

As in many recursive algorithms (especially those with exponential runtimes), we can optimize this through memorization (a form of dynamic programming).

Observe that some of the recursive calls will be essentially equivalent. For example, picking plank 1 and then plank 2 is equivalent to picking plank 2 and then plank 1.

Therefore, if we've seen this `(total, plank count)` pair before then we stop this recursive path. We can do this using a `HashSet` with a key of `(total, plank count)`.

> Many candidates will make a mistake here. Rather than stopping only when they've seen `(total, plank count)`, they'll stop whenever they've seen just `total` before. This is incorrect. Seeing two planks of length 1 is not the same thing as one plank of length 2, because there are different numbers of planks remaining. In memoization problems, be very careful about what you choose for your key.

The code for this approach is very similar to the earlier approach.

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2    HashSet<Integer> lengths = new HashSet<Integer>();
3    HashSet<String> visited = new HashSet<String>();
4    getAllLengths(k, 0, shorter, longer, lengths, visited);
5    return lengths;
6  }
7
8  void getAllLengths(int k, int total, int shorter, int longer,
9                     HashSet<Integer> lengths, HashSet<String> visited) {
10   if (k == 0) {
11     lengths.add(total);
12     return;
13   }
14   String key = k + " " + total;
```

```
15    if (visited.contains(key)) {
16       return;
17    }
18    getAllLengths(k - 1, total + shorter, shorter, longer, lengths, visited);
19    getAllLengths(k - 1, total + longer, shorter, longer, lengths, visited);
20    visited.add(key);
21 }
```

For simplicity, we've set the key to be a string representation of `total` and the current plank count. Some people may argue it's better to use a data structure to represent this pair. There are benefits to this, but there are drawbacks as well. It's worth discussing this tradeoff with your interviewer.

The runtime of this algorithm is a bit tricky to figure out.

One way we can think about the runtime is by understanding that we're basically filling in a table of SUMS x PLANK COUNTS. The biggest possible sum is K * LONGER and the biggest possible plank count is K. Therefore, the runtime will be no worse than $O(K^2 * LONGER)$.

Of course, a bunch of those sums will never actually be reached. How many unique sums can we get? Observe that any path with the same number of each type of planks will have the same sum. Since we can have at most K planks of each type, there are only K different sums we can make. Therefore, the table is really KxK, and the runtime is $O(K^2)$.

Optimal Solution

If you re-read the prior paragraph, you might notice something interesting. There are only K distinct sums we can get. Isn't that the whole point of the problem—to find all possible sums?

We don't actually need to go through all arrangements of planks. We just need to go through all unique sets of K planks (sets, not orders!). There are only K ways of picking K planks if we only have two possible types: {0 of type A, K of type B}, {1 of type A, K-1 of type B}, {2 of type A, K-2 of type B}, ...

This can be done in just a simple for loop. At each "sequence", we just compute the sum.

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2     HashSet<Integer> lengths = new HashSet<Integer>();
3     for (int nShorter = 0; nShorter <= k; nShorter++) {
4        int nLonger = k - nShorter;
5        int length = nShorter * shorter + nLonger * longer;
6        lengths.add(length);
7     }
8     return lengths;
9  }
```

We've used a `HashSet` here for consistency with the prior solutions. This isn't really necessary though, since we shouldn't get any duplicates. We could instead use an `ArrayList`. If we do this, though, we just need to handle an edge case where the two types of planks are the same length. In this case, we would just return an `ArrayList` of size 1.

16.12 XML Encoding: Since XML is very verbose, you are given a way of encoding it where each tag gets mapped to a pre-defined integer value. The language/grammar is as follows:

```
Element   --> Tag Attributes END Children END
Attribute --> Tag Value
END       --> 0
Tag       --> some predefined mapping to int
Value     --> string value
```

For example, the following XML might be converted into the compressed string below (assuming a mapping of `family -> 1, person ->2, firstName -> 3, lastName -> 4, state -> 5`).

```
<family lastName="McDowell" state="CA">
   <person firstName="Gayle">Some Message</person>
</family>
```

Becomes:

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0
```

Write code to print the encoded version of an XML element (passed in `Element` and `Attribute` objects).

pg 182

SOLUTION

Since we know the element will be passed in as an `Element` and `Attribute`, our code is reasonably simple. We can implement this by applying a tree-like approach.

We repeatedly call `encode()` on parts of the XML structure, handling the code in slightly different ways depending on the type of the XML element.

```
1  void encode(Element root, StringBuilder sb) {
2    encode(root.getNameCode(), sb);
3    for (Attribute a : root.attributes) {
4      encode(a, sb);
5    }
6    encode("0", sb);
7    if (root.value != null && root.value != "") {
8      encode(root.value, sb);
9    } else {
10     for (Element e : root.children) {
11       encode(e, sb);
12     }
13   }
14   encode("0", sb);
15 }
16
17 void encode(String v, StringBuilder sb) {
18   sb.append(v);
19   sb.append(" ");
20 }
21
22 void encode(Attribute attr, StringBuilder sb) {
23   encode(attr.getTagCode(), sb);
24   encode(attr.value, sb);
25 }
26
```

```
27 String encodeToString(Element root) {
28    StringBuilder sb = new StringBuilder();
29    encode(root, sb);
30    return sb.toString();
31 }
```

Observe in line 17, the use of the very simple encode method for a string. This is somewhat unnecessary; all it does is insert the string and a space following it. However, using this method is a nice touch as it ensures that every element will be inserted with a space surrounding it. Otherwise, it might be easy to break the encoding by forgetting to append the empty string.

16.13 Bisect Squares: Given two squares on a two-dimensional plane, find a line that would cut these two squares in half. Assume that the top and the bottom sides of the square run parallel to the x-axis.

pg 182

SOLUTION

Before we start, we should think about what exactly this problem means by a "line." Is a line defined by a slope and a y-intercept? Or by any two points on the line? Or, should the line be really a line segment, which starts and ends at the edges of the squares?

We will assume, since it makes the problem a bit more interesting, that we mean the third option: that the line should end at the edges of the squares. In an interview situation, you should discuss this with your interviewer.

This line that cuts two squares in half must connect the two middles. We can easily calculate the slope, knowing that $\text{slope} = \frac{y1 - y2}{x1 - x2}$. Once we calculate the slope using the two middles, we can use the same equation to calculate the start and end points of the line segment.

In the below code, we will assume the origin `(0, 0)` is in the upper left-hand corner.

```
1  public class Square {
2     ...
3     public Point middle() {
4        return new Point((this.left + this.right) / 2.0,
5                         (this.top + this.bottom) / 2.0);
6     }
7
8     /* Return the point where the line segment connecting mid1 and mid2 intercepts
9      * the edge of square 1. That is, draw a line from mid2 to mid1, and continue it
10     * out until the edge of the square. */
11    public Point extend(Point mid1, Point mid2, double size) {
12       /* Find what direction the line mid2 -> mid1 goes. */
13       double xdir = mid1.x < mid2.x ? -1 : 1;
14       double ydir = mid1.y < mid2.y ? -1 : 1;
15
16       /* If mid1 and mid2 have the same x value, then the slope calculation will
17        * throw a divide by 0 exception. So, we compute this specially. */
18       if (mid1.x == mid2.x) {
19          return new Point(mid1.x, mid1.y + ydir * size / 2.0);
20       }
21
22       double slope = (mid1.y - mid2.y) / (mid1.x - mid2.x);
23       double x1 = 0;
24       double y1 = 0;
25
```

```
26      /* Calculate slope using the equation (y1 - y2) / (x1 - x2).
27       * Note: if the slope is "steep" (>1) then the end of the line segment will
28       * hit size / 2 units away from the middle on the y axis. If the slope is
29       * "shallow" (<1) the end of the line segment will hit size / 2 units away
30       * from the middle on the x axis. */
31      if (Math.abs(slope) == 1) {
32         x1 = mid1.x + xdir * size / 2.0;
33         y1 = mid1.y + ydir * size / 2.0;
34      } else if (Math.abs(slope) < 1) { // shallow slope
35         x1 = mid1.x + xdir * size / 2.0;
36         y1 = slope * (x1 - mid1.x) + mid1.y;
37      } else { // steep slope
38         y1 = mid1.y + ydir * size / 2.0;
39         x1 = (y1 - mid1.y) / slope + mid1.x;
40      }
41      return new Point(x1, y1);
42   }
43
44   public Line cut(Square other) {
45      /* Calculate where a line between each middle would collide with the edges of
46       * the squares */
47      Point p1 = extend(this.middle(), other.middle(), this.size);
48      Point p2 = extend(this.middle(), other.middle(), -1 * this.size);
49      Point p3 = extend(other.middle(), this.middle(), other.size);
50      Point p4 = extend(other.middle(), this.middle(), -1 * other.size);
51
52      /* Of above points, find start and end of lines. Start is farthest left (with
53       * top most as a tie breaker) and end is farthest right (with bottom most as
54       * a tie breaker. */
55      Point start = p1;
56      Point end = p1;
57      Point[] points = {p2, p3, p4};
58      for (int i = 0; i < points.length; i++) {
59         if (points[i].x < start.x ||
60             (points[i].x == start.x && points[i].y < start.y)) {
61            start = points[i];
62         } else if (points[i].x > end.x ||
63                   (points[i].x == end.x && points[i].y > end.y)) {
64            end = points[i];
65         }
66      }
67
68      return new Line(start, end);
69   }
```

The main goal of this problem is to see how careful you are about coding. It's easy to glance over the special cases (e.g., the two squares having the same middle). You should make a list of these special cases before you start the problem and make sure to handle them appropriately. This is a question that requires careful and thorough testing.

16.14 Best Line: Given a two-dimensional graph with points on it, find a line which passes the most number of points.

pg 183

SOLUTION

This solution seems quite straightforward at first. And it is—sort of.

We just "draw" an infinite line (that is, not a line segment) between every two points and, using a hash table, track which line is the most common. This will take $O(N^2)$ time, since there are N^2 line segments.

We will represent a line as a slope and y-intercept (as opposed to a pair of points), which allows us to easily check to see if the line from `(x1, y1)` to `(x2, y2)` is equivalent to the line from `(x3, y3)` to `(x4, y4)`.

To find the most common line then, we just iterate through all lines segments, using a hash table to count the number of times we've seen each line. Easy enough!

However, there's one little complication. We're defining two lines to be equal if the lines have the same slope and y-intercept. We are then, furthermore, hashing the lines based on these values (specifically, based on the slope). The problem is that floating point numbers cannot always be represented accurately in binary. We resolve this by checking if two floating point numbers are within an `epsilon` value of each other.

What does this mean for our hash table? It means that two lines with "equal" slopes may not be hashed to the same value. To solve this, we will round the slope down to the next epsilon and use this `flooredSlope` as the hash key. Then, to retrieve all lines that are *potentially* equal, we will search the hash table at three spots: `flooredSlope`, `flooredSlope - epsilon`, and `flooredSlope + epsilon`. This will ensure that we've checked out all lines that might be equal.

```
1  /* Find line that goes through most number of points. */
2  Line findBestLine(GraphPoint[] points) {
3     HashMapList<Double, Line> linesBySlope = getListOfLines(points);
4     return getBestLine(linesBySlope);
5  }
6
7  /* Add each pair of points as a line to the list. */
8  HashMapList<Double, Line> getListOfLines(GraphPoint[] points) {
9     HashMapList<Double, Line> linesBySlope = new HashMapList<Double, Line>();
10    for (int i = 0; i < points.length; i++) {
11       for (int j = i + 1; j < points.length; j++) {
12          Line line = new Line(points[i], points[j]);
13          double key = Line.floorToNearestEpsilon(line.slope);
14          linesBySlope.put(key, line);
15       }
16    }
17    return linesBySlope;
18 }
19
20 /* Return the line with the most equivalent other lines. */
21 Line getBestLine(HashMapList<Double, Line> linesBySlope) {
22    Line bestLine = null;
23    int bestCount = 0;
24
25    Set<Double> slopes = linesBySlope.keySet();
26
27    for (double slope : slopes) {
```

```
      ArrayList<Line> lines = linesBySlope.get(slope);
      for (Line line : lines) {
         /* count lines that are equivalent to current line */
         int count = countEquivalentLines(linesBySlope, line);

         /* if better than current line, replace it */
         if (count > bestCount) {
            bestLine = line;
            bestCount = count;
            bestLine.Print();
            System.out.println(bestCount);
         }
      }
   }
   return bestLine;
}

/* Check hashmap for lines that are equivalent. Note that we need to check one
 * epsilon above and below the actual slope since we're defining two lines as
 * equivalent if they're within an epsilon of each other. */
int countEquivalentLines(HashMapList<Double, Line> linesBySlope, Line line) {
   double key = Line.floorToNearestEpsilon(line.slope);
   int count = countEquivalentLines(linesBySlope.get(key), line);
   count += countEquivalentLines(linesBySlope.get(key - Line.epsilon), line);
   count += countEquivalentLines(linesBySlope.get(key + Line.epsilon), line);
   return count;
}

/* Count lines within an array of lines which are "equivalent" (slope and
 * y-intercept are within an epsilon value) to a given line */
int countEquivalentLines(ArrayList<Line> lines, Line line) {
   if (lines == null) return 0;

   int count = 0;
   for (Line parallelLine : lines) {
      if (parallelLine.isEquivalent(line)) {
         count++;
      }
   }
   return count;
}

public class Line {
   public static double epsilon = .0001;
   public double slope, intercept;
   private boolean infinite_slope = false;

   public Line(GraphPoint p, GraphPoint q) {
      if (Math.abs(p.x - q.x) > epsilon) { // if x's are different
         slope = (p.y - q.y) / (p.x - q.x); // compute slope
         intercept = p.y - slope * p.x; // y intercept from y=mx+b
      } else {
         infinite_slope = true;
         intercept = p.x; // x-intercept, since slope is infinite
      }
   }
```

```
84
85    public static double floorToNearestEpsilon(double d) {
86       int r = (int) (d / epsilon);
87       return ((double) r) * epsilon;
88    }
89
90    public boolean isEquivalent(double a, double b) {
91       return (Math.abs(a - b) < epsilon);
92    }
93
94    public boolean isEquivalent(Object o) {
95       Line l = (Line) o;
96       if (isEquivalent(l.slope, slope) && isEquivalent(l.intercept, intercept) &&
97           (infinite_slope == l.infinite_slope)) {
98          return true;
99       }
100      return false;
101   }
102 }
103
104 /* HashMapList<String, Integer> is a HashMap that maps from Strings to
105  * ArrayList<Integer>. See appendix for implementation. */
```

We need to be careful about the calculation of the slope of a line. The line might be completely vertical, which means that it doesn't have a y-intercept and its slope is infinite. We can keep track of this in a separate flag (`infinite_slope`). We need to check this condition in the equals method.

16.15 Master Mind: The Game of Master Mind is played as follows:

The computer has four slots, and each slot will contain a ball that is red (R), yellow (Y), green (G) or blue (B). For example, the computer might have RGGB (Slot #1 is red, Slots #2 and #3 are green, Slot #4 is blue).

You, the user, are trying to guess the solution. You might, for example, guess YRGB.

When you guess the correct color for the correct slot, you get a "hit." If you guess a color that exists but is in the wrong slot, you get a "pseudo-hit." Note that a slot that is a hit can never count as a pseudo-hit.

For example, if the actual solution is RGBY and you guess GGRR, you have one hit and one pseudo-hit.

Write a method that, given a guess and a solution, returns the number of hits and pseudo-hits.

pg 183

SOLUTION

This problem is straightforward, but it's surprisingly easy to make little mistakes. You should check your code *extremely* thoroughly, on a variety of test cases.

We'll implement this code by first creating a frequency array which stores how many times each character occurs in `solution`, excluding times when the slot is a "hit." Then, we iterate through `guess` to count the number of pseudo-hits.

The code below implements this algorithm.

```
1  class Result {
2     public int hits = 0;
```

```
3     public int pseudoHits = 0;
4
5     public String toString() {
6        return "(" + hits + ", " + pseudoHits + ")";
7     }
8  }
9
10 int code(char c) {
11    switch (c) {
12    case 'B':
13       return 0;
14    case 'G':
15       return 1;
16    case 'R':
17       return 2;
18    case 'Y':
19       return 3;
20    default:
21       return -1;
22    }
23 }
24
25 int MAX_COLORS = 4;
26
27 Result estimate(String guess, String solution) {
28    if (guess.length() != solution.length()) return null;
29
30    Result res = new Result();
31    int[] frequencies = new int[MAX_COLORS];
32
33    /* Compute hits and build frequency table */
34    for (int i = 0; i < guess.length(); i++) {
35       if (guess.charAt(i) == solution.charAt(i)) {
36          res.hits++;
37       } else {
38          /* Only increment the frequency table (which will be used for pseudo-hits)
39           * if it's not a hit. If it's a hit, the slot has already been "used." */
40          int code = code(solution.charAt(i));
41          frequencies[code]++;
42       }
43    }
44
45    /* Compute pseudo-hits */
46    for (int i = 0; i < guess.length(); i++) {
47       int code = code(guess.charAt(i));
48       if (code >= 0 && frequencies[code] > 0 &&
49            guess.charAt(i) != solution.charAt(i)) {
50          res.pseudoHits++;
51          frequencies[code]--;
52       }
53    }
54    return res;
55 }
```

Note that the easier the algorithm for a problem is, the more important it is to write clean and correct code. In this case, we've pulled `code(char c)` into its own method, and we've created a `Result` class to hold the result, rather than just printing it.

16.16 Sub Sort: Given an array of integers, write a method to find indices `m` and `n` such that if you sorted elements `m` through `n`, the entire array would be sorted. Minimize `n - m` (that is, find the smallest such sequence).

EXAMPLE

Input: `1, 2, 4, 7, 10, 11, 7, 12, 6, 7, 16, 18, 19`

Output: `(3, 9)`

pg 183

SOLUTION

Before we begin, let's make sure we understand what our answer will look like. If we're looking for just two indices, this indicates that some middle section of the array will be sorted, with the start and end of the array already being in order.

Now, let's approach this problem by looking at an example.

```
1, 2, 4, 7, 10, 11, 8, 12, 5, 6, 16, 18, 19
```

Our first thought might be to just find the longest increasing subsequence at the beginning and the longest increasing subsequence at the end.

```
left:   1, 2, 4, 7, 10, 11
middle: 8, 12
right:  5, 6, 16, 18, 19
```

These subsequences are easy to generate. We just start from the left and the right sides, and work our way inward. When an element is out of order, then we have found the end of our increasing/decreasing subsequence.

In order to solve our problem, though, we would need to be able to sort the middle part of the array and, by doing just that, get all the elements in the array in order. Specifically, the following would have to be true:

```
/* all items on left are smaller than all items in middle */
min(middle) > end(left)

/* all items in middle are smaller than all items in right */
max(middle) < start(right)
```

Or, in other words, for all elements:

```
left < middle < right
```

In fact, this condition will *never* be met. The middle section is, by definition, the elements that were out of order. That is, it is *always* the case that `left.end > middle.start` and `middle.end > right.start`. Thus, you cannot sort the middle to make the entire array sorted.

But, what we can do is *shrink* the left and right subsequences until the earlier conditions are met. We need the left part to be smaller than all the elements in the middle and right side, and the right part to be bigger than all the elements on the left and right side.

Let `min` equal `min(middle and right side)` and `max` equal `max(middle and left side)`. Observe that since the right and left sides are already in sorted order, we only actually need to check their start or end point.

On the left side, we start with the end of the subsequence (value 11, at element 5) and move to the left. The value `min` equals 5. Once we find an element `i` such that `array[i] < min`, we know that we could sort the middle and have that part of the array appear in order.

Then, we do a similar thing on the right side. The value max equals 12. So, we begin with the start of the right subsequence (value 6) and move to the right. We compare the max of 12 to 6, then 7, then 16. When reach 16, we know that no elements smaller than 12 could be after it (since it's an increasing subsequence). Thus, the middle of the array could now be sorted to make the entire array sorted.

The following code implements this algorithm.

```
1  void findUnsortedSequence(int[] array) {
2    // find left subsequence
3    int end_left = findEndOfLeftSubsequence(array);
4    if (end_left >= array.length - 1) return; // Already sorted
5
6    // find right subsequence
7    int start_right = findStartOfRightSubsequence(array);
8
9    // get min and max
10   int max_index = end_left; // max of left side
11   int min_index = start_right; // min of right side
12   for (int i = end_left + 1; i < start_right; i++) {
13     if (array[i] < array[min_index]) min_index = i;
14     if (array[i] > array[max_index]) max_index = i;
15   }
16
17   // slide left until less than array[min_index]
18   int left_index = shrinkLeft(array, min_index, end_left);
19
20   // slide right until greater than array[max_index]
21   int right_index = shrinkRight(array, max_index, start_right);
22
23   System.out.println(left_index + " " + right_index);
24 }
25
26 int findEndOfLeftSubsequence(int[] array) {
27   for (int i = 1; i < array.length; i++) {
28     if (array[i] < array[i - 1]) return i - 1;
29   }
30   return array.length - 1;
31 }
32
33 int findStartOfRightSubsequence(int[] array) {
34   for (int i = array.length - 2; i >= 0; i--) {
35     if (array[i] > array[i + 1]) return i + 1;
36   }
37   return 0;
38 }
39
40 int shrinkLeft(int[] array, int min_index, int start) {
41   int comp = array[min_index];
42   for (int i = start - 1; i >= 0; i--) {
43     if (array[i] <= comp) return i + 1;
44   }
45   return 0;
46 }
47
48 int shrinkRight(int[] array, int max_index, int start) {
49   int comp = array[max_index];
50   for (int i = start; i < array.length; i++) {
```

```
51        if (array[i] >= comp) return i - 1;
52      }
53      return array.length - 1;
54  }
```

Note the use of other methods in this solution. Although we could have jammed it all into one method, it would have made the code a lot harder to understand, maintain, and test. In your interview coding, you should prioritize these aspects.

16.17 Contiguous Sequence: You are given an array of integers (both positive and negative). Find the contiguous sequence with the largest sum. Return the sum.

EXAMPLE

Input: `2, -8, 3, -2, 4, -10`

Output: `5 (i.e., {3, -2, 4})`

pg 183

SOLUTION

This is a challenging problem, but an extremely common one. Let's approach this by looking at an example:

2 3 -8 -1 2 4 -2 3

If we think about our array as having alternating sequences of positive and negative numbers, we can observe that we would never include only part of a negative subsequence or part of a positive sequence. Why would we? Including part of a negative subsequence would make things unnecessarily negative, and we should just instead not include that negative sequence at all. Likewise, including only part of a positive subsequence would be strange, since the sum would be even bigger if we included the whole thing.

For the purposes of coming up with our algorithm, we can think about our array as being a sequence of alternating negative and positive numbers. Each number corresponds to the sum of a subsequence of positive numbers of a subsequence of negative numbers. For the array above, our new reduced array would be:

5 -9 6 -2 3

This doesn't give away a great algorithm immediately, but it does help us to better understand what we're working with.

Consider the array above. Would it ever make sense to have `{5, -9}` in a subsequence? No. These numbers sum to `-4`, so we're better off not including either number, or possibly just having the sequence be just `{5}`).

When would we want negative numbers included in a subsequence? Only if it allows us to join two positive subsequences, each of which have a sum greater than the negative value.

We can approach this in a step-wise manner, starting with the first element in the array.

When we look at 5, this is the biggest sum we've seen so far. We set `maxSum` to 5, and `sum` to 5. Then, we consider -9. If we added it to `sum`, we'd get a negative value. There's no sense in extending the subsequence from 5 to -9 (which "reduces" to a sequence of just -4), so we just reset the value of `sum`.

Now, we consider 6. This subsequence is greater than 5, so we update both `maxSum` and `sum`.

Next, we look at -2. Adding this to 6 will set `sum` to 4. Since this is still a "value add" (when adjoined to another, bigger sequence), we *might* want `{6, -2}` in our max subsequence. We'll update `sum`, but not `maxSum`.

Finally, we look at 3. Adding 3 to `sum` (4) gives us 7, so we update `maxSum`. The max subsequence is therefore the sequence `{6, -2, 3}`.

When we look at this in the fully expanded array, our logic is identical. The code below implements this algorithm.

```
1  int getMaxSum(int[] a) {
2    int maxsum = 0;
3    int sum = 0;
4    for (int i = 0; i < a.length; i++) {
5      sum += a[i];
6      if (maxsum < sum) {
7        maxsum = sum;
8      } else if (sum < 0) {
9        sum = 0;
10     }
11   }
12   return maxsum;
13 }
```

If the array is all negative numbers, what is the correct behavior? Consider this simple array: `{-3, -10, -5}`. You could make a good argument that the maximum sum is either:

1. -3 (if you assume the subsequence can't be empty)
2. 0 (the subsequence has length 0)
3. `MINIMUM_INT` (essentially, the error case).

We went with option #2 (`maxSum = 0`), but there's no "correct" answer. This is a great thing to discuss with your interviewer; it will show how detail-oriented you are.

16.18 Pattern Matching: You are given two strings, `pattern` and `value`. The `pattern` string consists of just the letters a and b, describing a pattern within a string. For example, the string `catcatgocatgo` matches the pattern `aabab` (where `cat` is a and `go` is b). It also matches patterns like `a`, `ab`, and `b`. Write a method to determine if `value` matches `pattern`.

pg 183

SOLUTION

As always, we can start with a simple brute force approach.

Brute Force

A brute force algorithm is to just try all possible values for a and b and then check if this works.

We could do this by iterating through all substrings for a and all possible substrings for b. There are $O(n^2)$ substrings in a string of length n, so this will actually take $O(n^4)$ time. But then, for each value of a and b, we need to build the new string of this length and compare it for equality. This building/comparison step takes $O(n)$ time, giving an overall runtime of $O(n^5)$.

```
1  for each possible substring a
2    for each possible substring b
3      candidate = buildFromPattern(pattern, a, b)
4      if candidate equals value
5        return true
```

Ouch.

One easy optimization is to notice that if the pattern starts with 'a', then the a string must start at the beginning of `value`. (Otherwise, the b string must start at the beginning of `value`.) Therefore, there aren't $O(n^2)$ possible values for a; there are $O(n)$.

The algorithm then is to check if the pattern starts with a or b. If it starts with b, we can "invert" it (flipping each 'a' to a 'b' and each 'b' to an 'a') so that it starts with 'a'. Then, iterate through all possible substrings for a (each of which must begin at index 0) and all possible substrings for b (each of which must begin at some character after the end of a). As before, we then compare the string for this pattern with the original string.

This algorithm now takes $O(n^4)$ time.

There's one more minor (optional) optimization we can make. We don't actually need to do this "inversion" if the string starts with 'b' instead of 'a'. The `buildFromPattern` method can take care of this. We can think about the first character in the pattern as the "main" item and the other character as the alternate character. The `buildFromPattern` method can build the appropriate string based on whether 'a' is the main character or alternate character.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    int size = value.length();
5    for (int mainSize = 0; mainSize < size; mainSize++) {
6      String main = value.substring(0, mainSize);
7      for (int altStart = mainSize; altStart <= size; altStart++) {
8        for (int altEnd = altStart; altEnd <= size; altEnd++) {
9          String alt = value.substring(altStart, altEnd);
10         String cand = buildFromPattern(pattern, main, alt);
11         if (cand.equals(value)) {
12           return true;
13         }
14       }
15     }
16   }
17   return false;
18 }
19
20 String buildFromPattern(String pattern, String main, String alt) {
21   StringBuffer sb = new StringBuffer();
22   char first = pattern.charAt(0);
23   for (char c : pattern.toCharArray()) {
24     if (c == first) {
25       sb.append(main);
26     } else {
27       sb.append(alt);
28     }
29   }
30   return sb.toString();
31 }
```

We should look for a more optimal algorithm.

Optimized

Let's think through our current algorithm. Searching through all values for the main string is fairly fast (it takes $O(n)$ time). It's the alternate string that is so slow: $O(n^2)$ time. We should study how to optimize that.

Suppose we have a pattern like aabab and we're comparing it to the string catcatgocatgo. Once we've picked "cat" as the value for a to try, then the a strings are going to take up nine characters (three a strings with length three each). Therefore, the b strings must take up the remaining four characters, with each having length two. Moreover, we actually know exactly where they must occur, too. If a is cat, and the pattern is aabab, then b must be go.

In other words, once we've picked a, we've picked b too. There's no need to iterate. Gathering some basic stats on pattern (number of as, number of bs, first occurrence of each) and iterating through values for a (or whichever the main string is) will be sufficient.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    char mainChar = pattern.charAt(0);
5    char altChar = mainChar == 'a' ? 'b' : 'a';
6    int size = value.length();
7
8    int countOfMain = countOf(pattern, mainChar);
9    int countOfAlt = pattern.length() - countOfMain;
10   int firstAlt = pattern.indexOf(altChar);
11   int maxMainSize = size / countOfMain;
12
13   for (int mainSize = 0; mainSize <= maxMainSize; mainSize++) {
14     int remainingLength = size - mainSize * countOfMain;
15     String first = value.substring(0, mainSize);
16     if (countOfAlt == 0 || remainingLength % countOfAlt == 0) {
17       int altIndex = firstAlt * mainSize;
18       int altSize = countOfAlt == 0 ? 0 : remainingLength / countOfAlt;
19       String second = countOfAlt == 0 ? "" :
20                       value.substring(altIndex, altSize + altIndex);
21
22       String cand = buildFromPattern(pattern, first, second);
23       if (cand.equals(value)) {
24         return true;
25       }
26     }
27   }
28   return false;
29 }
30
31 int countOf(String pattern, char c) {
32   int count = 0;
33   for (int i = 0; i < pattern.length(); i++) {
34     if (pattern.charAt(i) == c) {
35       count++;
36     }
37   }
38   return count;
39 }
40
41 String buildFromPattern(...) {  /* same as before */ }
```

This algorithm takes $O(n^2)$, since we iterate through O(n) possibilities for the main string and do O(n) work to build and compare the strings.

Observe that we've also cut down the possibilities for the main string that we try. If there are three instances of the main string, then its length cannot be any more than one third of value.

Optimized (Alternate)

If you don't like the work of building a string only to compare it (and then destroy it), we can eliminate this.

Instead, we can iterate through the values for a and b as before. But this time, to check if the string matches the pattern (given those values for a and b), we walk through `value`, comparing each substring to the first instance of the a and b strings.

```
1  boolean doesMatch(String pattern, String value) {
2    if (pattern.length() == 0) return value.length() == 0;
3
4    char mainChar = pattern.charAt(0);
5    char altChar = mainChar == 'a' ? 'b' : 'a';
6    int size = value.length();
7
8    int countOfMain = countOf(pattern, mainChar);
9    int countOfAlt = pattern.length() - countOfMain;
10   int firstAlt = pattern.indexOf(altChar);
11   int maxMainSize = size / countOfMain;
12
13   for (int mainSize = 0; mainSize <= maxMainSize; mainSize++) {
14     int remainingLength = size - mainSize * countOfMain;
15     if (countOfAlt == 0 || remainingLength % countOfAlt == 0) {
16       int altIndex = firstAlt * mainSize;
17       int altSize = countOfAlt == 0 ? 0 : remainingLength / countOfAlt;
18       if (matches(pattern, value, mainSize, altSize, altIndex)) {
19         return true;
20       }
21     }
22   }
23   return false;
24 }
25
26 /* Iterates through pattern and value. At each character within pattern, checks if
27  * this is the main string or the alternate string. Then checks if the next set of
28  * characters in value match the original set of those characters (either the main
29  * or the alternate. */
30 boolean matches(String pattern, String value, int mainSize, int altSize,
31                 int firstAlt) {
32   int stringIndex = mainSize;
33   for (int i = 1; i < pattern.length(); i++) {
34     int size = pattern.charAt(i) == pattern.charAt(0) ? mainSize : altSize;
35     int offset = pattern.charAt(i) == pattern.charAt(0) ? 0 : firstAlt;
36     if (!isEqual(value, offset, stringIndex, size)) {
37       return false;
38     }
39     stringIndex += size;
40   }
41   return true;
42 }
43
44 /* Checks if two substrings are equal, starting at given offsets and continuing to
45  * size. */
46 boolean isEqual(String s1, int offset1, int offset2, int size) {
47   for (int i = 0; i < size; i++) {
48     if (s1.charAt(offset1 + i) != s1.charAt(offset2 + i)) {
49       return false;
```

```
50        }
51     }
52     return true;
53  }
```

This algorithm will still take $O(n^2)$ time, but the benefit is that it can short circuit when matches fail early (which they usually will). The previous algorithm must go through all the work to build the string before it can learn that it has failed.

16.19 Pond Sizes: You have an integer matrix representing a plot of land, where the value at that location represents the height above sea level. A value of zero indicates water. A pond is a region of water connected vertically, horizontally, or diagonally. The size of the pond is the total number of connected water cells. Write a method to compute the sizes of all ponds in the matrix.

EXAMPLE

Input:

```
0 2 1 0
0 1 0 1
1 1 0 1
0 1 0 1
```

Output: 2, 4, 1 (in any order)

pg 184

SOLUTION

The first thing we can try is just walking through the array. It's easy enough to find water: when it's a zero, that's water.

Given a water cell, how can we compute the amount of water nearby? If the cell is not adjacent to any zero cells, then the size of this pond is 1. If it is, then we need to add in the adjacent cells, plus any water cells adjacent to those cells. We need to, of course, be careful to not recount any cells. We can do this with a modified breadth-first or depth-first search. Once we visit a cell, we permanently mark it as visited.

For each cell, we need to check eight adjacent cells. We could do this by writing in lines to check up, down, left, right, and each of the four diagonal cells. It's even easier, though, to do this with a loop.

```
1  ArrayList<Integer> computePondSizes(int[][] land) {
2     ArrayList<Integer> pondSizes = new ArrayList<Integer>();
3     for (int r = 0; r < land.length; r++) {
4        for (int c = 0; c < land[r].length; c++) {
5           if (land[r][c] == 0) { // Optional. Would return anyway.
6              int size = computeSize(land, r, c);
7              pondSizes.add(size);
8           }
9        }
10    }
11    return pondSizes;
12 }
13
14 int computeSize(int[][] land, int row, int col) {
15    /* If out of bounds or already visited. */
16    if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
17        land[row][col] != 0) { // visited or not water
18       return 0;
19    }
```

```
20    int size = 1;
21    land[row][col] = -1; // Mark visited
22    for (int dr = -1; dr <= 1; dr++) {
23       for (int dc = -1; dc <= 1; dc++) {
24          size += computeSize(land, row + dr, col + dc);
25       }
26    }
27    return size;
28 }
```

In this case, we marked a cell as visited by setting its value to -1. This allows us to check, in one line (`land[row][col] != 0`), if the value is valid dry land or visited. In either case, the value will be zero.

You might also notice that the for loop iterates through nine cells, not eight. It includes the current cell. We could add a line in there to not recurse if `dr == 0` and `dc == 0`. This really doesn't save us much. We'll execute this if-statement in eight cells unnecessarily, just to avoid one recursive call. The recursive call returns immediately since the cell is marked as visited.

If you don't like modifying the input matrix, you can create a secondary `visited` matrix.

```
1  ArrayList<Integer> computePondSizes(int[][] land) {
2     boolean[][] visited = new boolean[land.length][land[0].length];
3     ArrayList<Integer> pondSizes = new ArrayList<Integer>();
4     for (int r = 0; r < land.length; r++) {
5        for (int c = 0; c < land[r].length; c++) {
6           int size = computeSize(land, visited, r, c);
7           if (size > 0) {
8              pondSizes.add(size);
9           }
10       }
11    }
12    return pondSizes;
13 }
14
15 int computeSize(int[][] land, boolean[][] visited, int row, int col) {
16    /* If out of bounds or already visited. */
17    if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
18        visited[row][col] || land[row][col] != 0) {
19       return 0;
20    }
21    int size = 1;
22    visited[row][col] = true;
23    for (int dr = -1; dr <= 1; dr++) {
24       for (int dc = -1; dc <= 1; dc++) {
25          size += computeSize(land, visited, row + dr, col + dc);
26       }
27    }
28    return size;
29 }
```

Both implementations are $O(WH)$, where W is the width of the matrix and H is the height.

> Note: Many people say "$O(N)$" or "$O(N^2)$", as though N has some inherent meaning. It doesn't. Suppose this were a square matrix. You could describe the runtime as $O(N)$ or $O(N^2)$. Both are correct, depending on what you mean by N. The runtime is $O(N^2)$, where N is the length of one side. Or, if N is the number of cells, it is $O(N)$. Be careful by what you mean by N. In fact, it might be safer to just not use N at all when there's any ambiguity as to what it could mean.

Some people will miscompute the runtime to be $O(N^4)$, reasoning that the `computeSize` method could take as long as $O(N^2)$ time and you might call it as much as $O(N^2)$ times (and apparently assuming an NxN matrix, too). While those are both basically correct statements, you can't just multiply them together. That's because as a single call to `computeSize` gets more expensive, the number of times it is called goes down.

For example, suppose the very first call to `computeSize` goes through the entire matrix. That might take $O(N^2)$ time, but then we never call `computeSize` again.

Another way to compute this is to think about how many times each cell is "touched" by either call. Each cell will be touched once by the `computePondSizes` function. Additionally, a cell might be touched once by each of its adjacent cells. This is still a constant number of touches per cell. Therefore, the overall runtime is $O(N^2)$ on an NxN matrix or, more generally, $O(WH)$.

16.20 T9: On old cell phones, users typed on a numeric keypad and the phone would provide a list of words that matched these numbers. Each digit mapped to a set of 0 - 4 letters. Implement an algorithm to return a list of matching words, given a sequence of digits. You are provided a list of valid words (provided in whatever data structure you'd like). The mapping is shown in the diagram below:

1	**2** abc	**3** def
4 ghi	**5** jkl	**6** mno
7 pqrs	**8** tuv	**9** wxyz
	0	

EXAMPLE

Input: 8733

Output: `tree, used`

pg 184

SOLUTION

We could approach this in a couple of ways. Let's start with a brute force algorithm.

Brute Force

Imagine how you would solve the problem if you had to do it by hand. You'd probably try every possible value for each digit with all other possible values.

This is exactly what we do algorithmically. We take the first digit and run through all the characters that map to that digit. For each character, we add it to a `prefix` variable and recurse, passing the prefix downward. Once we run out of characters, we print `prefix` (which now contains the full word) if the string is a valid word.

We will assume the list of words is passed in as a `HashSet`. A `HashSet` operates similarly to a hash table, but rather than offering key->value lookups, it can tell us if a word is contained in the set in $O(1)$ time.

```
1  ArrayList<String> getValidT9Words(String number, HashSet<String> wordList) {
2    ArrayList<String> results = new ArrayList<String>();
3    getValidWords(number, 0, "", wordList, results);
4    return results;
5  }
6
```

```
7  void getValidWords(String number, int index, String prefix,
8                     HashSet<String> wordSet, ArrayList<String> results) {
9    /* If it's a complete word, print it. */
10   if (index == number.length() && wordSet.contains(prefix)) {
11     results.add(prefix);
12     return;
13   }
14
15   /* Get characters that match this digit. */
16   char digit = number.charAt(index);
17   char[] letters = getT9Chars(digit);
18
19   /* Go through all remaining options. */
20   if (letters != null) {
21     for (char letter : letters) {
22       getValidWords(number, index + 1, prefix + letter, wordSet, results);
23     }
24   }
25 }
26
27 /* Return array of characters that map to this digit. */
28 char[] getT9Chars(char digit) {
29   if (!Character.isDigit(digit)) {
30     return null;
31   }
32   int dig = Character.getNumericValue(digit) - Character.getNumericValue('0');
33   return t9Letters[dig];
34 }
35
36 /* Mapping of digits to letters. */
37 char[][] t9Letters = {null, null, {'a', 'b', 'c'}, {'d', 'e', 'f'},
38   {'g', 'h', 'i'}, {'j', 'k', 'l'}, {'m', 'n', 'o'}, {'p', 'q', 'r', 's'},
39   {'t', 'u', 'v'},  {'w', 'x', 'y', 'z'}
40 };
```

This algorithm runs in $O(4^N)$ time, where N is the length of the string. This is because we recursively branch four times for each call to `getValidWords`, and we recurse until a call stack depth of N.

This is very, very slow on large strings.

Optimized

Let's return to thinking about how you would do this, if you were doing it by hand. Imagine the example of `33835676368` (which corresponds to `development`). If you were doing this by hand, I bet you'd skip over solutions that start with `fftf` [3383], as no valid words start with those characters.

Ideally, we'd like our program to make the same sort of optimization: stop recursing down paths which will obviously fail. Specifically, if there are no words in the dictionary that start with `prefix`, stop recursing.

The `Trie` data structure (see "Tries (Prefix Trees)" on page 105) can do this for us. Whenever we reach a string which is not a valid prefix, we exit.

```
1  ArrayList<String> getValidT9Words(String number, Trie trie) {
2    ArrayList<String> results = new ArrayList<String>();
3    getValidWords(number, 0, "", trie.getRoot(), results);
4    return results;
5  }
6
```

```
void getValidWords(String number, int index, String prefix, TrieNode trieNode,
                   ArrayList<String> results) {
   /* If it's a complete word, print it. */
   if (index == number.length()) {
      if (trieNode.terminates()) { // Is complete word
         results.add(prefix);
      }
      return;
   }

   /* Get characters that match this digit */
   char digit = number.charAt(index);
   char[] letters = getT9Chars(digit);

   /* Go through all remaining options. */
   if (letters != null) {
      for (char letter : letters) {
         TrieNode child = trieNode.getChild(letter);
         /* If there are words that start with prefix + letter,
          * then continue recursing. */
         if (child != null) {
            getValidWords(number, index + 1, prefix + letter, child, results);
         }
      }
   }
}
```

It's difficult to describe the runtime of this algorithm since it depends on what the language looks like. However, this "short-circuiting" will make it run much, much faster in practice.

Most Optimal

Believe or not, we can actually make it run even faster. We just need to do a little bit of preprocessing. That's not a big deal though. We were doing that to build the trie anyway.

This problem is asking us to list all the words represented by a particular number in T9. Instead of trying to do this "on the fly" (and going through a lot of possibilities, many of which won't actually work), we can just do this in advance.

Our algorithm now has a few steps:

Pre-Computation:

1. Create a hash table that maps from a sequence of digits to a list of strings.
2. Go through each word in the dictionary and convert it to its T9 representation (e.g., `APPLE -> 27753`). Store each of these in the above hash table. For example, 8733 would map to `{used, tree}`.

Word Lookup:

1. Just look up the entry in the hash table and return the list.

That's it!

```
/* WORD LOOKUP */
ArrayList<String> getValidT9Words(String numbers,
                                  HashMapList<String, String> dictionary) {
   return dictionary.get(numbers);
}

```

```
/* PRECOMPUTATION */

/* Create a hash table that maps from a number to all words that have this
 * numerical representation. */
HashMapList<String, String> initializeDictionary(String[] words) {
  /* Create a  hash table that maps from a letter to the digit */
  HashMap<Character, Character> letterToNumberMap = createLetterToNumberMap();

  /* Create word -> number map. */
  HashMapList<String, String> wordsToNumbers = new HashMapList<String, String>();
  for (String word : words) {
    String numbers = convertToT9(word, letterToNumberMap);
    wordsToNumbers.put(numbers, word);
  }
  return wordsToNumbers;
}

/* Convert mapping of number->letters into letter->number. */
HashMap<Character, Character> createLetterToNumberMap() {
  HashMap<Character, Character> letterToNumberMap =
    new HashMap<Character, Character>();
  for (int i = 0; i < t9Letters.length; i++) {
    char[] letters = t9Letters[i];
    if (letters != null) {
      for (char letter : letters) {
        char c = Character.forDigit(i, 10);
        letterToNumberMap.put(letter, c);
      }
    }
  }
  return letterToNumberMap;
}

/* Convert from a string to its T9 representation. */
String convertToT9(String word, HashMap<Character, Character> letterToNumberMap) {
  StringBuilder sb = new StringBuilder();
  for (char c : word.toCharArray()) {
    if (letterToNumberMap.containsKey(c)) {
      char digit = letterToNumberMap.get(c);
      sb.append(digit);
    }
  }
  return sb.toString();
}

char[][] t9Letters = /* Same as before */

/* HashMapList<String, Integer> is a HashMap that maps from Strings to
 * ArrayList<Integer>. See appendix for implementation. */
```

Getting the words that map to this number will run in $O(N)$ time, where N is the number of digits. The $O(N)$ comes in during the hash table look up (we need to convert the number to a hash table). If you know the words are never longer than a certain max size, then you could also describe the runtime as $O(1)$.

Note that it's easy to think, "Oh, linear—that's not that fast." But it depends what it's linear *on*. Linear on the length of the word is extremely fast. Linear on the length of the dictionary is not so fast.

16.21 Sum Swap: Given two arrays of integers, find a pair of values (one value from each array) that you can swap to give the two arrays the same sum.

EXAMPLE

Input: `{4, 1, 2, 1, 1, 2}` and `{3, 6, 3, 3}`

Output: `{1, 3}`

pg 184

SOLUTION

We should start by trying to understand what exactly we're looking for.

We have two arrays and their sums. Although we likely aren't given their sums upfront, we can just act like we are for now. After all, computing the sum is an O(N) operation and we know we can't beat O(N) anyway. Computing the sum, therefore, won't impact the runtime.

When we move a (positive) value a from array A to array B, then the sum of A drops by a and the sum of B increases by a.

We are looking for two values, a and b, such that:

```
sumA - a + b = sumB - b + a
```

Doing some quick math:

```
2a - 2b = sumA - sumB
a - b = (sumA - sumB) / 2
```

Therefore, we're looking for two values that have a specific target difference: `(sumA - sumB) / 2`.

Observe that because that the target must be an integer (after all, you can't swap two integers to get a non-integer difference), we can conclude that the difference between the sums must be even to have a valid pair.

Brute Force

A brute force algorithm is simple enough. We just iterate through the arrays and check all pairs of values.

We can either do this the "naive" way (compare the new sums) or by looking for a pair with that difference.

Naive approach:

```
1  int[] findSwapValues(int[] array1, int[] array2) {
2    int sum1 = sum(array1);
3    int sum2 = sum(array2);
4
5    for (int one : array1) {
6      for (int two : array2) {
7        int newSum1 = sum1 - one + two;
8        int newSum2 = sum2 - two + one;
9        if (newSum1 == newSum2) {
10         int[] values = {one, two};
11         return values;
12       }
13     }
14   }
15
16   return null;
17 }
```

Target approach:

```
1  int[] findSwapValues(int[] array1, int[] array2) {
```

```
    Integer target = getTarget(array1, array2);
    if (target == null) return null;

    for (int one : array1) {
       for (int two : array2) {
          if (one - two == target) {
             int[] values = {one, two};
             return values;
          }
       }
    }

    return null;
}

Integer getTarget(int[] array1, int[] array2) {
    int sum1 = sum(array1);
    int sum2 = sum(array2);

    if ((sum1 - sum2) % 2 != 0) return null;
    return (sum1 - sum2) / 2;
}
```

We've used an `Integer` (a boxed data type) as the return value for `getTarget`. This allows us to distinguish an "error" case.

This algorithm takes $O(AB)$ time.

Optimal Solution

This problem reduces to finding a pair of values that have a particular difference. With that in mind, let's revisit what the brute force does.

In the brute force, we're looping through A and then, for each element, looking for an element in B which gives us the "right" difference. If the value in A is 5 and the target is 3, then we must be looking for the value 2. That's the only value that could fulfill the goal.

That is, rather than writing `one - two == target`, we could have written `two == one - target`. How can we more quickly find an element in B that equals `one - target`?

We can do this very quickly with a hash table. We just throw all the elements in B into a hash table. Then, iterate through A and look for the appropriate element in B.

```
int[] findSwapValues(int[] array1, int[] array2) {
    Integer target = getTarget(array1, array2);
    if (target == null) return null;
    return findDifference(array1, array2, target);
}

/* Find a pair of values with a specific difference. */
int[] findDifference(int[] array1, int[] array2, int target) {
    HashSet<Integer> contents2 = getContents(array2);
    for (int one : array1) {
       int two = one - target;
       if (contents2.contains(two)) {
          int[] values = {one, two};
          return values;
       }
```

```
    }

    return null;
  }

  /* Put contents of array into hash set. */
  HashSet<Integer> getContents(int[] array) {
    HashSet<Integer> set = new HashSet<Integer>();
    for (int a : array) {
      set.add(a);
    }
    return set;
  }
```

This solution will take O(A+B) time. This is the Best Conceivable Runtime (BCR), since we have to at least touch every element in the two arrays.

Alternate Solution

If the arrays are sorted, we can iterate through them to find an appropriate pair. This will require less space.

```
int[] findSwapValues(int[] array1, int[] array2) {
  Integer target = getTarget(array1, array2);
  if (target == null) return null;
  return findDifference(array1, array2, target);
}

int[] findDifference(int[] array1, int[] array2, int target) {
  int a = 0;
  int b = 0;

  while (a < array1.length && b < array2.length) {
    int difference = array1[a] - array2[b];
    /* Compare difference to target. If difference is too small, then make it
     * bigger by moving a to a bigger value. If it is too big, then make it
     * smaller by moving b to a bigger value. If it's just right, return this
     * pair. */
    if (difference == target) {
      int[] values = {array1[a], array2[b]};
      return values;
    } else if (difference < target) {
      a++;
    } else {
      b++;
    }
  }

  return null;
}
```

This algorithm takes O(A + B) time but requires the arrays to be sorted. If the arrays aren't sorted, we can still apply this algorithm but we'd have to sort the arrays first. The overall runtime would be O(A log A + B log B).

16.22 Langton's Ant: An ant is sitting on an infinite grid of white and black squares. It initially faces right. At each step, it does the following:

(1) At a white square, flip the color of the square, turn 90 degrees right (clockwise), and move forward one unit.

(2) At a black square, flip the color of the square, turn 90 degrees left (counter-clockwise), and move forward one unit.

Write a program to simulate the first K moves that the ant makes and print the final board as a grid. Note that you are not provided with the data structure to represent the grid. This is something you must design yourself. The only input to your method is K. You should print the final grid and return nothing. The method signature might be something like `void printKMoves(int K)`.

pg 185

SOLUTION

At first glance, this problem seems very straightforward: create a grid, remember the ant's position and orientation, flip the cells, turn, and move. The interesting part comes in how to handle an infinite grid.

Solution #1: Fixed Array

Technically, since we're only running the first K moves, we do have a max size for the grid. The ant cannot move more than K moves in either direction. If we create a grid that has width 2K and height 2K (and place the ant at the center), we know it will be big enough.

The problem with this is that it's not very extensible. If you run K moves and then want to run another K moves, you might be out of luck.

Additionally, this solution wastes a good amount of space. The max might be K moves in a particular dimension, but the ant is probably going in circles a bit. You probably won't need all this space.

Solution #2: Resizable Array

One thought is to use a resizable array, such as Java's `ArrayList` class. This allows us to grow an array as necessary, while still offering `O(1)` amortized insertion.

The problem is that our grid needs to grow in two dimensions, but the `ArrayList` is only a single array. Additionally, we need to grow "backward" into negative values. The `ArrayList` class doesn't support this.

However, we take a similar approach by building our own resizable grid. Each time the ant hits an edge, we double the size of the grid in that dimension.

What about the negative expansions? While conceptually we can talk about something being at negative positions, we cannot actually access array indices with negative values.

One way we can handle this is to create "fake indices." Let us treat the ant as being at coordinates `(-3, -10)`, but track some sort of offset or delta to translate these coordinates into array indices.

This is actually unnecessary, though. The ant's location does not need to be publicly exposed or consistent (unless, of course, indicated by the interviewer). When the ant travels into negative coordinates, we can double the size of the array and just move the ant and all cells into the positive coordinates. Essentially, we are relabeling all the indices.

This relabeling will not impact the big O time since we have to create a new matrix anyway.

```
1  public class Grid {
2    private boolean[][] grid;
```

```
3     private Ant ant = new Ant();
4
5     public Grid() {
6        grid = new boolean[1][1];
7     }
8
9     /* Copy old values into new array, with an offset/shift applied to the row and
10     * columns. */
11    private void copyWithShift(boolean[][] oldGrid, boolean[][] newGrid,
12                               int shiftRow, int shiftColumn) {
13       for (int r = 0; r < oldGrid.length; r++) {
14          for (int c = 0; c < oldGrid[0].length; c++) {
15             newGrid[r + shiftRow][c + shiftColumn] = oldGrid[r][c];
16          }
17       }
18    }
19
20    /* Ensure that the given position will fit on the array. If necessary, double
21     * the size of the matrix, copy the old values over, and adjust the ant's
22     * position so that it's in a positive range. */
23    private void ensureFit(Position position) {
24       int shiftRow = 0;
25       int shiftColumn = 0;
26
27       /* Calculate new number of rows. */
28       int numRows = grid.length;
29       if (position.row < 0) {
30          shiftRow = numRows;
31          numRows *= 2;
32       } else if (position.row >= numRows) {
33          numRows *= 2;
34       }
35
36       /* Calculate new number of columns. */
37       int numColumns = grid[0].length;
38       if (position.column < 0) {
39          shiftColumn = numColumns;
40          numColumns *= 2;
41       } else if (position.column >= numColumns) {
42          numColumns *= 2;
43       }
44
45       /* Grow array, if necessary. Shift ant's position too. */
46       if (numRows != grid.length || numColumns != grid[0].length) {
47          boolean[][] newGrid = new boolean[numRows][numColumns];
48          copyWithShift(grid, newGrid, shiftRow, shiftColumn);
49          ant.adjustPosition(shiftRow, shiftColumn);
50          grid = newGrid;
51       }
52    }
53
54    /* Flip color of cells. */
55    private void flip(Position position) {
56       int row = position.row;
57       int column = position.column;
58       grid[row][column] = grid[row][column] ? false : true;
```

```
    }

    /* Move ant. */
    public void move() {
        ant.turn(grid[ant.position.row][ant.position.column]);
        flip(ant.position);
        ant.move();
        ensureFit(ant.position); // grow
    }

    /* Print board. */
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int r = 0; r < grid.length; r++) {
            for (int c = 0; c < grid[0].length; c++) {
                if (r == ant.position.row && c == ant.position.column) {
                    sb.append(ant.orientation);
                } else if (grid[r][c]) {
                    sb.append("X");
                } else {
                    sb.append("_");
                }
            }
            sb.append("\n");
        }
        sb.append("Ant: " + ant.orientation + ". \n");
        return sb.toString();
    }
}
```

We pulled the Ant code into a separate class. The nice thing about this is that if we need to have multiple ants for some reason, we can easily extend the code to support this.

```
public class Ant {
    public Position position = new Position(0, 0);
    public Orientation orientation = Orientation.right;

    public void turn(boolean clockwise) {
        orientation = orientation.getTurn(clockwise);
    }

    public void move() {
        if (orientation == Orientation.left) {
            position.column--;
        } else if (orientation == Orientation.right) {
            position.column++;
        } else if (orientation == Orientation.up) {
            position.row--;
        } else if (orientation == Orientation.down) {
            position.row++;
        }
    }

    public void adjustPosition(int shiftRow, int shiftColumn) {
        position.row += shiftRow;
        position.column += shiftColumn;
    }
}
```

`Orientation` is also its own `enum`, with a few useful functions.

```
public enum Orientation {
   left, up, right, down;

   public Orientation getTurn(boolean clockwise) {
      if (this == left) {
         return clockwise ? up : down;
      } else if (this == up) {
         return clockwise ? right : left;
      } else if (this == right) {
         return clockwise ? down : up;
      } else { // down
         return clockwise ? left : right;
      }
   }

   @Override
   public String toString() {
      if (this == left) {
         return "\u2190";
      } else if (this == up) {
         return "\u2191";
      } else if (this == right) {
         return "\u2192";
      } else { // down
         return "\u2193";
      }
   }
}
```

We've also put `Position` into its own simple class. We could just as easily track the row and column separately.

```
public class Position {
   public int row;
   public int column;

   public Position(int row, int column) {
      this.row = row;
      this.column = column;
   }
}
```

This works, but it's actually more complicated than is necessary.

Solution #3: HashSet

Although it may seem "obvious" that we would use a matrix to represent a grid, it's actually easier not to do that. All we actually need is a list of the white squares (as well as the ant's location and orientation).

We can do this by using a HashSet of the white squares. If a position is in the hash set, then the square is white. Otherwise, it is black.

The one tricky bit is how to print the board. Where do we start printing? Where do we end?

Since we will need to print a grid, we can track what should be top-left and bottom-right corner of the grid. Each time the ant moves, we compare the ant's position to the most top-left position and most bottom-right position, updating them if necessary.

```
1  public class Board {
2    private HashSet<Position> whites = new HashSet<Position>();
3    private Ant ant = new Ant();
4    private Position topLeftCorner = new Position(0, 0);
5    private Position bottomRightCorner = new Position(0, 0);
6
7    public Board() { }
8
9    /* Move ant. */
10   public void move() {
11     ant.turn(isWhite(ant.position)); // Turn
12     flip(ant.position); // flip
13     ant.move(); // move
14     ensureFit(ant.position);
15   }
16
17   /* Flip color of cells. */
18   private void flip(Position position) {
19     if (whites.contains(position)) {
20       whites.remove(position);
21     } else {
22       whites.add(position.clone());
23     }
24   }
25
26   /* Grow grid by tracking the most top-left and bottom-right positions.*/
27   private void ensureFit(Position position) {
28     int row = position.row;
29     int column = position.column;
30
31     topLeftCorner.row = Math.min(topLeftCorner.row, row);
32     topLeftCorner.column = Math.min(topLeftCorner.column, column);
33
34     bottomRightCorner.row = Math.max(bottomRightCorner.row, row);
35     bottomRightCorner.column = Math.max(bottomRightCorner.column, column);
36   }
37
38   /* Check if cell is white. */
39   public boolean isWhite(Position p) {
40     return whites.contains(p);
41   }
42
43   /* Check if cell is white. */
44   public boolean isWhite(int row, int column) {
45     return whites.contains(new Position(row, column));
46   }
47
48   /* Print board. */
49   public String toString() {
50     StringBuilder sb = new StringBuilder();
51     int rowMin = topLeftCorner.row;
52     int rowMax = bottomRightCorner.row;
53     int colMin = topLeftCorner.column;
54     int colMax = bottomRightCorner.column;
55     for (int r = rowMin; r <= rowMax; r++) {
56       for (int c = colMin; c <= colMax; c++) {
```

```
57            if (r == ant.position.row && c == ant.position.column) {
58               sb.append(ant.orientation);
59            } else if (isWhite(r, c)) {
60               sb.append("X");
61            } else {
62               sb.append("_");
63            }
64         }
65         sb.append("\n");
66      }
67      sb.append("Ant: " + ant.orientation + ". \n");
68      return sb.toString();
69   }
```

The implementation of `Ant` and `Orientation` is the same.

The implementation of `Position` gets updated slightly, in order to support the `HashSet` functionality. The position will be the key, so we need to implement a `hashCode()` function.

```
1  public class Position {
2     public int row;
3     public int column;
4
5     public Position(int row, int column) {
6        this.row = row;
7        this.column = column;
8     }
9
10    @Override
11    public boolean equals(Object o) {
12       if (o instanceof Position) {
13          Position p = (Position) o;
14          return p.row == row && p.column == column;
15       }
16       return false;
17    }
18
19    @Override
20    public int hashCode() {
21       /* There are many options for hash functions. This is one. */
22       return (row * 31) ^ column;
23    }
24
25    public Position clone() {
26       return new Position(row, column);
27    }
28 }
```

The nice thing about this implementation is that if we do need to access a particular cell elsewhere, we have consistent row and column labeling.

16.23 Rand7 from Rand5: Implement a method `rand7()` given `rand5()`. That is, given a method that generates a random number between 0 and 4 (inclusive), write a method that generates a random number between 0 and 6 (inclusive).

pg 186

SOLUTION

To implement this function correctly, we must have each of the values between 0 and 6 returned with 1/7th probability.

First Attempt (Fixed Number of Calls)

As a first attempt, we might try generating all numbers between 0 and 9, and then mod the resulting value by 7. Our code for it might look something like this:

```
1  int rand7() {
2    int v = rand5() + rand5();
3    return v % 7;
4  }
```

Unfortunately, the above code will not generate the values with equal probability. We can see this by looking at the results of each call to `rand5()` an d the return result of the `rand7()` function.

1st Call	2nd Call	Result	1st Call	2nd Call	Result
0	0	0	2	3	5
0	1	1	2	4	6
0	2	2	3	0	3
0	3	3	3	1	4
0	4	4	3	2	5
1	0	1	3	3	6
1	1	2	3	4	0
1	2	3	4	0	4
1	3	4	4	1	5
1	4	5	4	2	6
2	0	2	4	3	0
2	1	3	4	4	1
2	2	4			

Each individual row has a 1 in 25 chance of occurring, since there are two calls to `rand5()` and each distributes its results with $1/5$ th probability. If you count up the number of times each number occurs, you'll note that this `rand7()` function will return 4 with $5/25$ th probability but return 0 with just $3/25$ th probability. This means that our function has failed; the results do not have probability $1/7$ th.

Now, imagine we modify our function to add an if-statement, to change the constant multiplier, or to insert a new call to `rand5()`. We will still wind up with a similar looking table, and the probability of getting any one of those rows will be $1/5^k$, where k is the number of calls to `rand5()` in that row. Different rows may have different number of calls.

The probability of winding up with the result of the `rand7()` function being, say, 6 would be the sum of the probabilities of all rows that result in 6. That is:

$$P(\texttt{rand7()} = 6) = 1/5^i + 1/5^j + \ldots + 1/5^m$$

We know that, in order for our function to be correct, this probability must equal $1/7$. This is impossible though. Because 5 and 7 are relatively prime, no series of reciprocal powers of 5 will result in $1/7$.

Does this mean the problem is impossible? Not exactly. Strictly speaking, it means that, as long as we can list out the combinations of `rand5()` results that will result in a particular value of `rand7()`, the function will not give well distributed results.

We can still solve this problem. We just have to use a while loop, and realize that there's no telling just how many turns will be required to return a result.

Second Attempt (Nondeterministic Number of Calls)

As soon as we've allowed for a while loop, our work gets much easier. We just need to generate a range of values where each value is equally likely (and where the range has at least seven elements). If we can do this, then we can discard the elements greater than the previous multiple of 7, and mod the rest of them by 7. This will get us a value within the range of 0 to 6, with each value being equally likely.

In the below code, we generate the range 0 through 24 by doing `5 * rand5() + rand5()`. Then, we discard the values between 21 and 24, since they would otherwise make `rand7()` unfairly weighted towards 0 through 3. Finally, we mod by 7 to give us the values in the range 0 to 6 with equal probability.

Note that because we discard values in this approach, we have no guarantee on the number of `rand5()` calls it may take to return a value. This is what is meant by a *nondeterministic* number of calls.

```
1  int rand7() {
2    while (true) {
3      int num = 5 * rand5() + rand5();
4      if (num < 21) {
5        return num % 7;
6      }
7    }
8  }
```

Observe that doing `5 * rand5() + rand5()` gives us exactly one way of getting each number in its range (0 to 24). This ensures that each value is equally probable.

Could we instead do `2 * rand5() + rand5()`? No, because the values wouldn't be equally distributed. For example, there would be three ways of getting a 6 (`6 = 2 * 1 + 4`, `6 = 2 * 2 + 2`, and `6 = 2 * 3 + 0`) but only one way of getting a 0 (`0=2*0+0`). The values in the range are not equally probable.

There *is* a way that we can use `2 * rand5()` and still get an identically distributed range, but it's much more complicated. See below.

```
1  int rand7() {
2    while (true) {
3      int r1 = 2 * rand5(); /* evens between 0 and 9 */
4      int r2 = rand5(); /* used later to generate a 0 or 1 */
5      if (r2 != 4) { /* r2 has extra even num-discard the extra */
6        int rand1 = r2 % 2; /* Generate 0 or 1 */
7        int num = r1 + rand1; /* will be in the range 0 to 9 */
8        if (num < 7) {
9          return num;
10       }
11     }
12   }
13 }
```

In fact, there is an infinite number of ranges we can use. The key is to make sure that the range is big enough and that all values are equally likely.

16.24 Pairs with Sum: Design an algorithm to find all pairs of integers within an array which sum to a specified value.

pg 185

SOLUTION

Let's start with a definition. If we're trying to find a pair of numbers that sums to z, the *complement* of x will be z - x (that is, the number that can be added to x to make z). For example, if we're trying to find a pair of numbers that sums to 12, the complement of -5 would be 17.

Brute Force

A brute force solution is to just iterate through all pairs and print the pair if its sum matches the target sum.

```
1  ArrayList<Pair> printPairSums(int[] array, int sum) {
2    ArrayList<Pair> result = new ArrayList<Pair>();
3    for (int i = 0 ; i < array.length; i++) {
4      for (int j = i + 1; j < array.length; j++) {
5        if (array[i] + array[j] == sum) {
6          result.add(new Pair(array[i], array[j]));
7        }
8      }
9    }
10   return result;
11 }
```

If there are duplicates in the array (e.g., {5, 6, 5}), it might print the same sum twice. You should discuss this with your interviewer.

Optimized Solution

We can optimize this with a hash map, where the value in the hash map reflects the number of "unpaired" instances of a key. We walk through the array. At each element x, check how many unpaired instances of x's complement preceded it in the array. If the count is at least one, then there is an unpaired instance of x's complement. We add this pair and decrement x's complement to signify that this element has been paired. If the count is zero, then increment the value of x in the hash table to signify that x is unpaired.

```
1  ArrayList<Pair> printPairSums(int[] array, int sum) {
2    ArrayList<Pair> result = new ArrayList<Pair>();
3    HashMap<Integer, Integer> unpairedCount = new HashMap<Integer, Integer>();
4    for (int x : array) {
5      int complement = sum - x;
6      if (unpairedCount.getOrDefault(complement, 0) > 0) {
7        result.add(new Pair(x, complement));
8        adjustCounterBy(unpairedCount, complement, -1); // decrement complement
9      } else {
10       adjustCounterBy(unpairedCount, x, 1); // increment count
11     }
12   }
13   return result;
14 }
15
```

```
16 void adjustCounterBy(HashMap<Integer, Integer> counter, int key, int delta) {
17    counter.put(key, counter.getOrDefault(key, 0) + delta);
18 }
```

This solution will print duplicate pairs, but will not reuse the same instance of an element. It will take O(N) time and O(N) space.

Alternate Solution

Alternatively, we can sort the array and then find the pairs in a single pass. Consider this array:

{-2, -1, 0, 3, 5, 6, 7, 9, 13, 14}.

Let `first` point to the head of the array and `last` point to the end of the array. To find the complement of `first`, we just move `last` backwards until we find it. If `first + last < sum`, then there is no complement for `first`. We can therefore move `first` forward. We stop when `first` is greater than `last`.

Why must this find all complements for `first`? Because the array is sorted and we're trying progressively smaller numbers. When the sum of `first` and `last` is less than the sum, we know that trying even smaller numbers (as `last`) won't help us find a complement.

Why must this find all complements for `last`? Because all pairs must be made up of a `first` and a `last`. We've found all complements for `first`, therefore we've found all complements of `last`.

```
1  void printPairSums(int[] array, int sum) {
2     Arrays.sort(array);
3     int first = 0;
4     int last = array.length - 1;
5     while (first < last) {
6        int s = array[first] + array[last];
7        if (s == sum) {
8           System.out.println(array[first] + " " + array[last]);
9           first++;
10          last--;
11       } else {
12          if (s < sum) first++;
13          else last--;
14       }
15    }
16 }
```

This algorithm takes O(N log N) time to sort and O(N) time to find the pairs.

Note that since the array is presumably unsorted, it would be equally fast in terms of big O to just do a binary search at each element for its complement. This would give us a two-step algorithm, where each step is O(N log N).

16.25 LRU Cache: Design and build a "least recently used" cache, which evicts the least recently used item. The cache should map from keys to values (allowing you to insert and retrieve a value associated with a particular key) and be initialized with a max size. When it is full, it should evict the least recently used item. You can assume the keys are integers and the values are strings.

pg 185

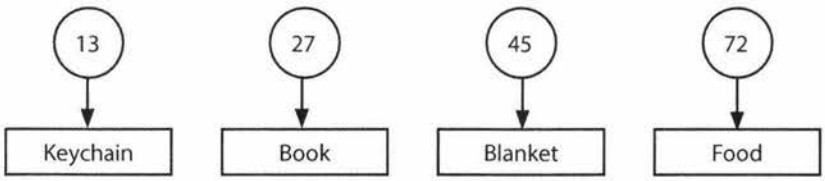
SOLUTION

We should start off by defining the scope of the problem. What exactly do we need to achieve?

- **Inserting Key, Value Pair:** We need to be able to insert a (key, value) pair.

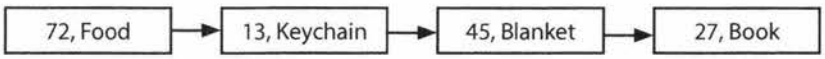
- **Retrieving Value by Key:** We need to be able to retrieve the value using the key.
- **Finding Least Recently Used:** We need to know the least recently used item (and, likely, the usage ordering of all items).
- **Updating Most Recently Used:** When we retrieve a value by key, we need to update the order to be the most recently used item.
- **Eviction:** The cache should have a max capacity and should remove the least recently used item when it hits capacity.

The (key, value) mapping suggests a hash table. This would make it easy to look up the value associated with a particular key.



Unfortunately, a hash table usually would not offer a quick way to remove the most recently used item. We could mark each item with a timestamp and iterate through the hash table to remove the item with the lowest timestamp, but that can get quite slow (O(N) for insertions).

Instead, we could use a linked list, ordered by the most recently used. This would make it easy to mark an item as the most recently used (just put it in the front of the list) or to remove the least recently used item (remove the end).

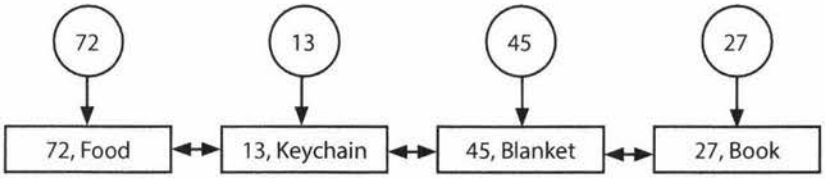


Unfortunately, this does not offer a quick way to look up an item by its key. We could iterate through the linked list and find the item by key. But this could get very slow (O(N) for retrieval).

Each approach does half of the problem (different halves) very well, but neither approach does both parts well.

Can we get the best parts of each? Yes. By using both!

The linked list looks as it did in the earlier example, but now it's a doubly linked list. This allows us to easily remove an element from the middle of the linked list. The hash table now maps to each linked list node rather than the value.



The algorithms now operate as follows:

- **Inserting Key, Value Pair:** Create a linked list node with key, value. Insert into head of linked list. Insert key -> node mapping into hash table.
- **Retrieving Value by Key:** Look up node in hash table and return value. Update most recently used item

(see below).

- **Finding Least Recently Used:** Least recently used item will be found at the end of the linked list.
- **Updating Most Recently Used:** Move node to front of linked list. Hash table does not need to be updated.
- **Eviction:** Remove tail of linked list. Get key from linked list node and remove key from hash table.

The code below implements these classes and algorithms.

```
1  public class Cache {
2    private int maxCacheSize;
3    private HashMap<Integer, LinkedListNode> map =
4      new HashMap<Integer, LinkedListNode>();
5    private LinkedListNode listHead = null;
6    public LinkedListNode listTail = null;
7
8    public Cache(int maxSize) {
9      maxCacheSize = maxSize;
10   }
11
12   /* Get value for key and mark as most recently used. */
13   public String getValue(int key) {
14     LinkedListNode item = map.get(key);
15     if (item == null) return null;
16
17     /* Move to front of list to mark as most recently used. */
18     if (item != listHead) {
19       removeFromLinkedList(item);
20       insertAtFrontOfLinkedList(item);
21     }
22     return item.value;
23   }
24
25   /* Remove node from linked list. */
26   private void removeFromLinkedList(LinkedListNode node) {
27     if (node == null)  return;
28
29     if (node.prev != null) node.prev.next = node.next;
30     if (node.next != null) node.next.prev = node.prev;
31     if (node == listTail) listTail = node.prev;
32     if (node == listHead) listHead = node.next;
33   }
34
35   /* Insert node at front of linked list. */
36   private void insertAtFrontOfLinkedList(LinkedListNode node) {
37     if (listHead == null) {
38       listHead = node;
39       listTail = node;
40     } else {
41       listHead.prev = node;
42       node.next = listHead;
43       listHead = node;
44     }
45   }
46
47   /* Remove key/value pair from cache, deleting from hashtable and linked list. */
48   public boolean removeKey(int key) {
```

```
49        LinkedListNode node = map.get(key);
50        removeFromLinkedList(node);
51        map.remove(key);
52        return true;
53     }
54
55     /* Put key, value pair in cache. Removes old value for key if necessary. Inserts
56      * pair into linked list and hash table.*/
57     public void setKeyValue(int key, String value) {
58        /* Remove if already there. */
59        removeKey(key);
60
61        /* If full, remove least recently used item from cache. */
62        if (map.size() >= maxCacheSize && listTail != null) {
63           removeKey(listTail.key);
64        }
65
66        /* Insert new node. */
67        LinkedListNode node = new LinkedListNode(key, value);
68        insertAtFrontOfLinkedList(node);
69        map.put(key, node);
70     }
71
72     private static class LinkedListNode {
73        private LinkedListNode next, prev;
74        public int key;
75        public String value;
76        public LinkedListNode(int k, String v) {
77           key = k;
78           value = v;
79        }
80     }
81  }
```

Note that we've chosen to make `LinkedListNode` an inner class of `Cache`, since no other classes should need access to this class and really should only exist within the scope of `Cache`.

16.26 Calculator: Given an arithmetic equation consisting of positive integers, +, -, * and / (no parentheses), compute the result.

EXAMPLE

Input: `2*3+5/6*3+15`

Output: `23.5`

pg 185

SOLUTION

The first thing we should realize is that the dumb thing—just applying each operator left to right—won't work. Multiplication and division are considered "higher priority" operations, which means that they have to happen before addition.

For example, if you have the simple expression 3+6*2, the multiplication must be performed first, and then the addition. If you just processed the equation left to right, you would end up with the incorrect result, 18, rather than the correct one, 15. You know all of this, of course, but it's worth really spelling out what it means.

Solution #1

We can still process the equation from left to right; we just have to be a little smarter about how we do it. Multiplication and division need to be grouped together such that whenever we see those operations, we perform them immediately on the surrounding terms.

For example, suppose we have this expression:

```
2 - 6 - 7*8/2 + 5
```

It's fine to compute 2 - 6 immediately and store it into a `result` variable. But, when we see `7*(something)`, we know we need to fully process that term before adding it to the result.

We can do this by reading left to right and maintaining two variables.

- The first is `processing`, which maintains the result of the current cluster of terms (both the operator and the value). In the case of addition and subtraction, the cluster will be just the current term. In the case of multiplication and division, it will be the full sequence (until you get to the next addition or subtraction).
- The second is the `result` variable. If the next term is an addition or subtraction (or there is no next term), then `processing` is applied to `result`.

On the above example, we would do the following:

1. Read +2. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
   ```
   processing = {+, 2} --> null
   result = 0          --> 2
   ```
2. Read -6. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
   ```
   processing = {-, 6} --> null
   result = 2          --> -4
   ```
3. Read -7. Apply it to `processing`. Observe next sign is a *. Continue.
   ```
   processing = {-, 7}
   result = -4
   ```
4. Read *8. Apply it to `processing`. Observe next sign is a /. Continue.
   ```
   processing = {-, 56}
   result = -4
   ```
5. Read /2. Apply it to `processing`. Observe next sign is a +, which terminates this multiplication and division cluster. Apply processing to `result`. Clear `processing`.
   ```
   processing = {-, 28}    --> null
   result = -4         --> -32
   ```
6. Read +5. Apply it to `processing`. Apply processing to `result`. Clear `processing`.
   ```
   processing = {+, 5} --> null
   result = -32        --> -27
   ```

The code below implements this algorithm.

```
1  /* Compute the result of the arithmetic sequence. This works by reading left to
2   * right and applying each term to a result. When we see a multiplication or
3   * division, we instead apply this sequence to a temporary variable. */
4  double compute(String sequence) {
5    ArrayList<Term> terms = Term.parseTermSequence(sequence);
6    if (terms == null) return Integer.MIN_VALUE;
7
8    double result = 0;
9    Term processing = null;
10   for (int i = 0; i < terms.size(); i++) {
```

```
11      Term current = terms.get(i);
12      Term next = i + 1 < terms.size() ? terms.get(i + 1) : null;
13
14      /* Apply the current term to "processing". */
15      processing = collapseTerm(processing, current);
16
17      /* If next term is + or -, then this cluster is done and we should apply
18       * "processing" to "result". */
19      if (next == null || next.getOperator() == Operator.ADD
20          || next.getOperator() == Operator.SUBTRACT) {
21        result = applyOp(result, processing.getOperator(), processing.getNumber());
22        processing = null;
23      }
24    }
25
26    return result;
27  }
28
29  /* Collapse two terms together using the operator in secondary and the numbers
30   * from each. */
31  Term collapseTerm(Term primary, Term secondary) {
32    if (primary == null) return secondary;
33    if (secondary == null) return primary;
34
35    double value = applyOp(primary.getNumber(), secondary.getOperator(),
36                   secondary.getNumber());
37    primary.setNumber(value);
38    return primary;
39  }
40
41  double applyOp(double left, Operator op, double right) {
42    if (op == Operator.ADD) return left + right;
43    else if (op == Operator.SUBTRACT) return left - right;
44    else if (op == Operator.MULTIPLY) return left * right;
45    else if (op == Operator.DIVIDE) return left / right;
46    else return right;
47  }
48
49  public class Term {
50    public enum Operator {
51      ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
52    }
53
54    private double value;
55    private Operator operator = Operator.BLANK;
56
57    public Term(double v, Operator op) {
58      value = v;
59      operator = op;
60    }
61
62    public double getNumber() { return value; }
63    public Operator getOperator() { return operator; }
64    public void setNumber(double v) { value = v; }
65
66    /* Parses arithmetic sequence into a list of Terms. For example, 3-5*6 becomes
```

```
67      * something like: [{BLANK,3}, {SUBTRACT, 5}, {MULTIPLY, 6}].
68      * If improperly formatted, returns null. */
69     public static ArrayList<Term> parseTermSequence(String sequence) {
70       /* Code can be found in downloadable solutions. */
71     }
72  }
```

This takes O(N) time, where N is the length of the initial string.

Solution #2

Alternatively, we can solve this problem using two stacks: one for numbers and one for operators.

```
2 - 6 - 7 * 8 / 2 + 5
```

The processing works as follows:

- Each time we see a number, it gets pushed onto `numberStack`.
- Operators get pushed onto `operatorStack`—as long as the operator has higher priority than the current top of the stack. If `priority(currentOperator) <= priority(operatorStack.top())`, then we "collapse" the top of the stacks:
  - » Collapsing: pop two elements off `numberStack`, pop an operator off `operatorStack`, apply the operator, and push the result onto `numberStack`.
  - » Priority: addition and subtraction have equal priority, which is lower than the priority of multiplication and division (also equal priority).

  This collapsing continues until the above inequality is broken, at which point `currentOperator` is pushed onto `operatorStack`.
- At the very end, we collapse the stack.

Let's see this with an example: `2 - 6 - 7 * 8 / 2 + 5`

	action	numberStack	operatorStack
2	numberStack.push(2)	2	[empty]
-	operatorStack.push(-)	2	-
6	numberStack.push(6)	6, 2	-
-	collapseStacks [2 - 6] operatorStack.push(-)	-4 -4	[empty] -
7	numberStack.push(7)	7, -4	-
*	operatorStack.push(*)	7, -4	*, -
8	numberStack.push(8)	8, 7, -4	*, -
/	collapseStack [7 * 8] numberStack.push(/)	56, -4 56, -4	- /, -
2	numberStack.push(2)	2, 56, -4	/, -
+	collapseStack [56 / 2] collapseStack [-4 - 28] operatorStack.push(+)	28, -4 -32 -32	- [empty] +
5	numberStack.push(5)	5, -32	+
	collapseStack [-32 + 5]	-27	[empty]
	return -27		

The code below implements this algorithm.

```
1  public enum Operator {
2    ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
3  }
4
5  double compute(String sequence) {
6    Stack<Double> numberStack = new Stack<Double>();
7    Stack<Operator> operatorStack = new Stack<Operator>();
8
9    for (int i = 0; i < sequence.length(); i++) {
10     try {
11       /* Get number and push. */
12       int value = parseNextNumber(sequence, i);
13       numberStack.push((double) value);
14
15       /* Move to the operator. */
16       i += Integer.toString(value).length();
17       if (i >= sequence.length()) {
18         break;
19       }
20
21       /* Get operator, collapse top as needed, push operator. */
22       Operator op = parseNextOperator(sequence, i);
23       collapseTop(op, numberStack, operatorStack);
24       operatorStack.push(op);
25     } catch (NumberFormatException ex) {
26       return Integer.MIN_VALUE;
27     }
28   }
29
30   /* Do final collapse. */
31   collapseTop(Operator.BLANK, numberStack, operatorStack);
32   if (numberStack.size() == 1 && operatorStack.size() == 0) {
33     return numberStack.pop();
34   }
35   return 0;
36 }
37
38 /* Collapse top until priority(futureTop) > priority(top). Collapsing means to pop
39  * the top 2 numbers and apply the operator popped from the top of the operator
40  * stack, and then push that onto the numbers stack.*/
41 void collapseTop(Operator futureTop, Stack<Double> numberStack,
42                  Stack<Operator> operatorStack) {
43   while (operatorStack.size() >= 1 && numberStack.size() >= 2) {
44     if (priorityOfOperator(futureTop) <=
45         priorityOfOperator(operatorStack.peek())) {
46       double second = numberStack.pop();
47       double first = numberStack.pop();
48       Operator op = operatorStack.pop();
49       double collapsed = applyOp(first, op, second);
50       numberStack.push(collapsed);
51     } else {
52       break;
53     }
54   }
```

```
}

/* Return priority of operator. Mapped so that:
 *     addition == subtraction < multiplication == division. */
int priorityOfOperator(Operator op) {
   switch (op) {
      case ADD: return 1;
      case SUBTRACT: return 1;
      case MULTIPLY: return 2;
      case DIVIDE: return 2;
      case BLANK: return 0;
   }
   return 0;
}

/* Apply operator: left [op] right. */
double applyOp(double left, Operator op, double right) {
   if (op == Operator.ADD) return left + right;
   else if (op == Operator.SUBTRACT) return left - right;
   else if (op == Operator.MULTIPLY) return left * right;
   else if (op == Operator.DIVIDE) return left / right;
   else return right;
}

/* Return the number that starts at offset. */
int parseNextNumber(String seq, int offset) {
   StringBuilder sb = new StringBuilder();
   while (offset < seq.length() && Character.isDigit(seq.charAt(offset))) {
      sb.append(seq.charAt(offset));
      offset++;
   }
   return Integer.parseInt(sb.toString());
}

/* Return the operator that occurs as offset. */
Operator parseNextOperator(String sequence, int offset) {
   if (offset < sequence.length()) {
      char op = sequence.charAt(offset);
      switch(op) {
         case '+': return Operator.ADD;
         case '-': return Operator.SUBTRACT;
         case '*': return Operator.MULTIPLY;
         case '/': return Operator.DIVIDE;
      }
   }
   return Operator.BLANK;
}
```

This code also takes O(N) time, where N is the length of the string.

This solution involves a lot of annoying string parsing code. Remember that getting all these details out is not that important in an interview. In fact, your interviewer might even let you assume the expression is passed in pre-parsed into some sort of data structure.

Focus on modularizing your code from the beginning and "farming out" tedious or less interesting parts of the code to other functions. You want to focus on getting the core `compute` function working. The rest of the details can wait!