

When merged together, the lines above make P(3).

Case:  $n > 0$

Generating  $P(n)$  for the general case is just a simple generalization of the above steps. We compute  $P(n-1)$ , clone the results, and then add  $a_n$  to each of these cloned sets.

The following code implements this algorithm:

```
1 ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set, int index) {  
2     ArrayList<ArrayList<Integer>> allsubsets;  
3     if (set.size() == index) { // Base case - add empty set  
4         allsubsets = new ArrayList<ArrayList<Integer>>();  
5         allsubsets.add(new ArrayList<Integer>()); // Empty set  
6     } else {  
7         allsubsets = getSubsets(set, index + 1);  
8         int item = set.get(index);  
9         ArrayList<ArrayList<Integer>> moresubsets =  
10            new ArrayList<ArrayList<Integer>>();  
11            for (ArrayList<Integer> subset : allsubsets) {  
12                ArrayList<Integer> newsubset = new ArrayList<Integer>();  
13                newsubset.addAll(subset); //  
14                newsubset.add(item);  
15                moresubsets.add(newsubset);  
16            }  
17            allsubsets.addAll(moresubsets);  
18        }  
19    return allsubsets;  
20 }
```

This solution will be  $O(n2^n)$  in time and space, which is the best we can do. For a slight optimization, we could also implement this algorithm iteratively.

### Solution #2: Combinatorics

While there's nothing wrong with the above solution, there's another way to approach it.

Recall that when we're generating a set, we have two choices for each element: (1) the element is in the set (the "yes" state) or (2) the element is not in the set (the "no" state). This means that each subset is a sequence of yeses / nos—e.g., "yes, yes, no, no, yes, no"

This gives us  $2^n$  possible subsets. How can we iterate through all possible sequences of "yes" / "no" states for all elements? If each "yes" can be treated as a 1 and each "no" can be treated as a 0, then each subset can be represented as a binary string.

Generating all subsets, then, really just comes down to generating all binary numbers (that is, all integers). We iterate through all numbers from 0 to  $2^n$  (exclusive) and translate the binary representation of the numbers into a set. Easy!

```
1 ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {  
2     ArrayList<ArrayList<Integer>> allsubsets = new ArrayList<ArrayList<Integer>>();  
3     int max = 1 << set.size(); /* Compute  $2^n$  */  
4     for (int k = 0; k < max; k++) {  
5         ArrayList<Integer> subset = convertIntToSet(k, set);  
6         allsubsets.add(subset);  
7     }  
8     return allsubsets;  
9 }
```

```
10
11 ArrayList<Integer> convertIntToSet(int x, ArrayList<Integer> set) {
12     ArrayList<Integer> subset = new ArrayList<Integer>();
13     int index = 0;
14     for (int k = x; k > 0; k >>= 1) {
15         if ((k & 1) == 1) {
16             subset.add(set.get(index));
17         }
18         index++;
19     }
20     return subset;
21 }
```

There's nothing substantially better or worse about this solution compared to the first one.

- 8.5 Recursive Multiply:** Write a recursive function to multiply two positive integers without using the \* operator (or / operator). You can use addition, subtraction, and bit shifting, but you should minimize the number of those operations.

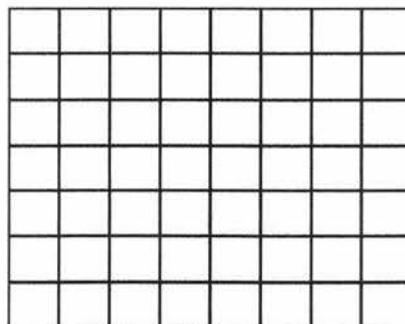
pg 135

### SOLUTION

Let's pause for a moment and think about what it means to do multiplication.

This is a good approach for a lot of interview questions. It's often useful to think about what it really means to do something, even when it's pretty obvious.

We can think about multiplying 8x7 as doing 8+8+8+8+8+8+8 (or adding 7 eight times). We can also think about it as the number of squares in an 8x7 grid.



#### Solution #1

How would we count the number of squares in this grid? We could just count each cell. That's pretty slow, though.

Alternatively, we could count half the squares and then double it (by adding this count to itself). To count half the squares, we repeat the same process.

Of course, this "doubling" only works if the number is in fact even. When it's not even, we need to do the counting/summing from scratch.

```
1 int minProduct(int a, int b) {
2     int bigger = a < b ? b : a;
```

```

3     int smaller = a < b ? a : b;
4     return minProductHelper(smaller, bigger);
5 }
6
7 int minProductHelper(int smaller, int bigger) {
8     if (smaller == 0) { // 0 x bigger = 0
9         return 0;
10    } else if (smaller == 1) { // 1 x bigger = bigger
11        return bigger;
12    }
13
14    /* Compute half. If uneven, compute other half. If even, double it. */
15    int s = smaller >> 1; // Divide by 2
16    int side1 = minProduct(s, bigger);
17    int side2 = side1;
18    if (smaller % 2 == 1) {
19        side2 = minProductHelper(smaller - s, bigger);
20    }
21
22    return side1 + side2;
23 }

```

Can we do better? Yes.

## Solution #2

If we observe how the recursion operates, we'll notice that we have duplicated work. Consider this example:

```

minProduct(17, 23)
    minProduct(8, 23)
        minProduct(4, 23) * 2
        ...
    + minProduct(9, 23)
        minProduct(4, 23)
        ...
    + minProduct(5, 23)
    ...

```

The second call to `minProduct(4, 23)` is unaware of the prior call, and so it repeats the same work. We should cache these results.

```

1  int minProduct(int a, int b) {
2      int bigger = a < b ? b : a;
3      int smaller = a < b ? a : b;
4
5      int memo[] = new int[smaller + 1];
6      return minProduct(smaller, bigger, memo);
7  }
8
9  int minProduct(int smaller, int bigger, int[] memo) {
10     if (smaller == 0) {
11         return 0;
12     } else if (smaller == 1) {
13         return bigger;
14     } else if (memo[smaller] > 0) {
15         return memo[smaller];
16     }
17
18     /* Compute half. If uneven, compute other half. If even, double it. */

```

```
19     int s = smaller >> 1; // Divide by 2
20     int side1 = minProduct(s, bigger, memo); // Compute half
21     int side2 = side1;
22     if (smaller % 2 == 1) {
23         side2 = minProduct(smaller - s, bigger, memo);
24     }
25
26     /* Sum and cache.*/
27     memo[smaller] = side1 + side2;
28     return memo[smaller];
29 }
```

We can still make this a bit faster.

### Solution #3

One thing we might notice when we look at this code is that a call to `minProduct` on an even number is much faster than one on an odd number. For example, if we call `minProduct(30, 35)`, then we'll just do `minProduct(15, 35)` and double the result. However, if we do `minProduct(31, 35)`, then we'll need to call `minProduct(15, 35)` and `minProduct(16, 35)`.

This is unnecessary. Instead, we can do:

$$\text{minProduct}(31, 35) = 2 * \text{minProduct}(15, 35) + 35$$

After all, since  $31 = 2*15+1$ , then  $31 \times 35 = 2*15*35+35$ .

The logic in this final solution is that, on even numbers, we just divide `smaller` by 2 and double the result of the recursive call. On odd numbers, we do the same, but then we also add `bigger` to this result.

In doing so, we have an unexpected “win.” Our `minProduct` function just recurses straight downwards, with increasingly small numbers each time. It will never repeat the same call, so there’s no need to cache any information.

```
1  int minProduct(int a, int b) {
2      int bigger = a < b ? b : a;
3      int smaller = a < b ? a : b;
4      return minProductHelper(smaller, bigger);
5  }
6
7  int minProductHelper(int smaller, int bigger) {
8      if (smaller == 0) return 0;
9      else if (smaller == 1) return bigger;
10
11     int s = smaller >> 1; // Divide by 2
12     int halfProd = minProductHelper(s, bigger);
13
14     if (smaller % 2 == 0) {
15         return halfProd + halfProd;
16     } else {
17         return halfProd + halfProd + bigger;
18     }
19 }
```

This algorithm will run in  $O(\log s)$  time, where  $s$  is the smaller of the two numbers.

- 8.6 Towers of Hanoi:** In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e., each disk sits on top of an even larger one). You have the following constraints:

- (1) Only one disk can be moved at a time.
- (2) A disk is slid off the top of one tower onto another tower.
- (3) A disk cannot be placed on top of a smaller disk.

Write a program to move the disks from the first tower to the last using Stacks.

pg 135

### SOLUTION

This problem sounds like a good candidate for the Base Case and Build approach.



Let's start with the smallest possible example:  $n = 1$ .

Case  $n = 1$ . Can we move Disk 1 from Tower 1 to Tower 3? Yes.

1. We simply move Disk 1 from Tower 1 to Tower 3.

Case  $n = 2$ . Can we move Disk 1 and Disk 2 from Tower 1 to Tower 3? Yes.

1. Move Disk 1 from Tower 1 to Tower 2
2. Move Disk 2 from Tower 1 to Tower 3
3. Move Disk 1 from Tower 2 to Tower 3

Note how in the above steps, Tower 2 acts as a buffer, holding a disk while we move other disks to Tower 3.

Case  $n = 3$ . Can we move Disk 1, 2, and 3 from Tower 1 to Tower 3? Yes.

1. We know we can move the top two disks from one tower to another (as shown earlier), so let's assume we've already done that. But instead, let's move them to Tower 2.
2. Move Disk 3 to Tower 3.
3. Move Disk 1 and Disk 2 to Tower 3. We already know how to do this—just repeat what we did in Step 1.

Case  $n = 4$ . Can we move Disk 1, 2, 3 and 4 from Tower 1 to Tower 3? Yes.

1. Move Disks 1, 2, and 3 to Tower 2. We know how to do that from the earlier examples.
2. Move Disk 4 to Tower 3.
3. Move Disks 1, 2 and 3 back to Tower 3.

Remember that the labels of Tower 2 and Tower 3 aren't important. They're equivalent towers. So, moving disks to Tower 3 with Tower 2 serving as a buffer is equivalent to moving disks to Tower 2 with Tower 3 serving as a buffer.

This approach leads to a natural recursive algorithm. In each part, we are doing the following steps, outlined below with pseudocode:

```
1 moveDisks(int n, Tower origin, Tower destination, Tower buffer) {  
2     /* Base case */  
3     if (n <= 0) return;  
4  
5     /* move top n - 1 disks from origin to buffer, using destination as a buffer. */  
6     moveDisks(n - 1, origin, buffer, destination);  
7  
8     /* move top from origin to destination  
9     moveTop(origin, destination);  
10    /* move top n - 1 disks from buffer to destination, using origin as a buffer. */  
11    moveDisks(n - 1, buffer, destination, origin);  
12 }  
13 }
```

The following code provides a more detailed implementation of this algorithm, using concepts of object-oriented design.

```
1 void main(String[] args) {  
2     int n = 3;  
3     Tower[] towers = new Tower[n];  
4     for (int i = 0; i < 3; i++) {  
5         towers[i] = new Tower(i);  
6     }  
7  
8     for (int i = n - 1; i >= 0; i--) {  
9         towers[0].add(i);  
10    }  
11    towers[0].moveDisks(n, towers[2], towers[1]);  
12 }  
13  
14 class Tower {  
15     private Stack<Integer> disks;  
16     private int index;  
17     public Tower(int i) {  
18         disks = new Stack<Integer>();  
19         index = i;  
20     }  
21  
22     public int index() {  
23         return index;  
24     }  
25  
26     public void add(int d) {  
27         if (!disks.isEmpty() && disks.peek() <= d) {  
28             System.out.println("Error placing disk " + d);  
29         } else {  
30             disks.push(d);  
31         }  
32     }  
33  
34     public void moveTopTo(Tower t) {  
35         int top = disks.pop();  
36         t.add(top);  
37     }  
38 }
```

```

39     public void moveDisks(int n, Tower destination, Tower buffer) {
40         if (n > 0) {
41             moveDisks(n - 1, buffer, destination);
42             moveTopTo(destination);
43             buffer.moveDisks(n - 1, destination, this);
44         }
45     }
46 }
```

Implementing the towers as their own objects is not strictly necessary, but it does help to make the code cleaner in some respects.

- 8.7 Permutations without Dups:** Write a method to compute all permutations of a string of unique characters.

pg 135

## SOLUTION

Like in many recursive problems, the Base Case and Build approach will be useful. Assume we have a string S represented by the characters  $a_1 a_2 \dots a_n$ .

### Approach 1: Building from permutations of first n-1 characters.

*Base Case: permutations of first character substring*

The only permutation of  $a_1$  is the string  $a_1$ . So:

$$P(a_1) = a_1$$

*Case: permutations of  $a_1 a_2$*

$$P(a_1 a_2) = a_1 a_2 \text{ and } a_2 a_1$$

*Case: permutations of  $a_1 a_2 a_3$*

$$P(a_1 a_2 a_3) = a_1 a_2 a_3, a_1 a_3 a_2, a_2 a_1 a_3, a_2 a_3 a_1, a_3 a_1 a_2, a_3 a_2 a_1$$

*Case: permutations of  $a_1 a_2 a_3 a_4$*

This is the first interesting case. How can we generate permutations of  $a_1 a_2 a_3 a_4$  from  $a_1 a_2 a_3$ ?

Each permutation of  $a_1 a_2 a_3 a_4$  represents an ordering of  $a_1 a_2 a_3$ . For example,  $a_2 a_4 a_1 a_3$  represents the order  $a_2 a_1 a_3$ .

Therefore, if we took all the permutations of  $a_1 a_2 a_3$  and added  $a_4$  into all possible locations, we would get all permutations of  $a_1 a_2 a_3 a_4$ :

$$\begin{aligned} a_1 a_2 a_3 &\rightarrow a_4 a_1 a_2 a_3, a_1 a_4 a_2 a_3, a_1 a_2 a_4 a_3, a_1 a_2 a_3 a_4 \\ a_1 a_3 a_2 &\rightarrow a_4 a_1 a_3 a_2, a_1 a_4 a_3 a_2, a_1 a_3 a_4 a_2, a_1 a_3 a_2 a_4 \\ a_3 a_1 a_2 &\rightarrow a_4 a_3 a_1 a_2, a_3 a_4 a_1 a_2, a_3 a_1 a_4 a_2, a_3 a_1 a_2 a_4 \\ a_2 a_1 a_3 &\rightarrow a_4 a_2 a_1 a_3, a_2 a_4 a_1 a_3, a_2 a_1 a_4 a_3, a_2 a_1 a_3 a_4 \\ a_2 a_3 a_1 &\rightarrow a_4 a_2 a_3 a_1, a_2 a_4 a_3 a_1, a_2 a_3 a_4 a_1, a_2 a_3 a_1 a_4 \\ a_3 a_2 a_1 &\rightarrow a_4 a_3 a_2 a_1, a_3 a_4 a_2 a_1, a_3 a_2 a_4 a_1, a_3 a_2 a_1 a_4 \end{aligned}$$

We can now implement this algorithm recursively.

```

1 ArrayList<String> getPerms(String str) {
2     if (str == null) return null;
3
4     ArrayList<String> permutations = new ArrayList<String>();
5     if (str.length() == 0) { // base case
6         permutations.add("");
7     } else {
8         for (String perm : getPerms(str.substring(1))) {
9             for (int i = 0; i < str.length(); i++) {
10                permutations.add(str.charAt(i) + perm);
11            }
12        }
13    }
14    return permutations;
15 }
```

```

7     return permutations;
8 }
9
10    char first = str.charAt(0); // get the first char
11    String remainder = str.substring(1); // remove the first char
12    ArrayList<String> words = getPerms(remainder);
13    for (String word : words) {
14        for (int j = 0; j <= word.length(); j++) {
15            String s = insertCharAt(word, first, j);
16            permutations.add(s);
17        }
18    }
19    return permutations;
20 }
21
22 /* Insert char c at index i in word. */
23 String insertCharAt(String word, char c, int i) {
24     String start = word.substring(0, i);
25     String end = word.substring(i);
26     return start + c + end;
27 }
```

### Approach 2: Building from permutations of all n-1 character substrings.

*Base Case: single-character strings*

The only permutation of  $a_1$  is the string  $a_1$ . So:

$$P(a_1) = a_1$$

*Case: two-character strings*

$$P(a_1a_2) = a_1a_2 \text{ and } a_2a_1.$$

$$P(a_2a_3) = a_2a_3 \text{ and } a_3a_2.$$

$$P(a_1a_3) = a_1a_3 \text{ and } a_3a_1.$$

*Case: three-character strings*

Here is where the cases get more interesting. How can we generate all permutations of three-character strings, such as  $a_1a_2a_3$ , given the permutations of two-character strings?

Well, in essence, we just need to "try" each character as the first character and then append the permutations.

$$\begin{aligned} P(a_1a_2a_3) &= \{a_1 + P(a_2a_3)\} + a_2 + P(a_1a_3) + \{a_3 + P(a_1a_2)\} \\ &\{a_1 + P(a_2a_3)\} \rightarrow a_1a_2a_3, a_1a_3a_2 \\ &\{a_2 + P(a_1a_3)\} \rightarrow a_2a_1a_3, a_2a_3a_1 \\ &\{a_3 + P(a_1a_2)\} \rightarrow a_3a_1a_2, a_3a_2a_1 \end{aligned}$$

Now that we can generate all permutations of three-character strings, we can use this to generate permutations of four-character strings.

$$P(a_1a_2a_3a_4) = \{a_1 + P(a_2a_3a_4)\} + \{a_2 + P(a_1a_3a_4)\} + \{a_3 + P(a_1a_2a_4)\} + \{a_4 + P(a_1a_2a_3)\}$$

This is now a fairly straightforward algorithm to implement.

```

1  ArrayList<String> getPerms(String remainder) {
2      int len = remainder.length();
3      ArrayList<String> result = new ArrayList<String>();
4
5      /* Base case. */
```

```

6  if (len == 0) {
7      result.add(""); // Be sure to return empty string!
8      return result;
9  }
10
11
12 for (int i = 0; i < len; i++) {
13     /* Remove char i and find permutations of remaining chars.*/
14     String before = remainder.substring(0, i);
15     String after = remainder.substring(i + 1, len);
16     ArrayList<String> partials = getPerms(before + after);
17
18     /* Prepend char i to each permutation.*/
19     for (String s : partials) {
20         result.add(remainder.charAt(i) + s);
21     }
22 }
23
24 return result;
25 }
```

Alternatively, instead of passing the permutations back up the stack, we can push the prefix down the stack. When we get to the bottom (base case), `prefix` holds a full permutation.

```

1 ArrayList<String> getPerms(String str) {
2     ArrayList<String> result = new ArrayList<String>();
3     getPerms("", str, result);
4     return result;
5 }
6
7 void getPerms(String prefix, String remainder, ArrayList<String> result) {
8     if (remainder.length() == 0) result.add(prefix);
9
10    int len = remainder.length();
11    for (int i = 0; i < len; i++) {
12        String before = remainder.substring(0, i);
13        String after = remainder.substring(i + 1, len);
14        char c = remainder.charAt(i);
15        getPerms(prefix + c, before + after, result);
16    }
17 }
```

For a discussion of the runtime of this algorithm, see Example 12 on page 51.

- 8.8 Permutations with Duplicates:** Write a method to compute all permutations of a string whose characters are not necessarily unique. The list of permutations should not have duplicates.

pg 135

## SOLUTION

This is very similar to the previous problem, except that now we could potentially have duplicate characters in the word.

One simple way of handling this problem is to do the same work to check if a permutation has been created before and then, if not, add it to the list. A simple hash table will do the trick here. This solution will take  $O(n!)$  time in the worst case (and, in fact, in all cases).

While it's true that we can't beat this worst case time, we should be able to design an algorithm to beat this in many cases. Consider a string with all duplicate characters, likeaaaaaaaaaaaaaa. This will take an extremely long time (since there are over 6 billion permutations of a 13-character string), even though there is only one unique permutation.

Ideally, we would like to only create the unique permutations, rather than creating every permutation and then ruling out the duplicates.

We can start with computing the count of each letter (easy enough to get this—just use a hash table). For a string such as aabbhb, this would be:

a->2 | b->4 | c->1

Let's imagine generating a permutation of this string (now represented as a hash table). The first choice we make is whether to use an a, b, or c as the first character. After that, we have a subproblem to solve: find all permutations of the remaining characters, and append those to the already picked "prefix."

$$\begin{aligned} P(a \rightarrow 2 \mid b \rightarrow 4 \mid c \rightarrow 1) &= \{a + P(a \rightarrow 1 \mid b \rightarrow 4 \mid c \rightarrow 1)\} + \\ &\quad \{b + P(a \rightarrow 2 \mid b \rightarrow 3 \mid c \rightarrow 1)\} + \\ &\quad \{c + P(a \rightarrow 2 \mid b \rightarrow 4 \mid c \rightarrow 0)\} \\ P(a \rightarrow 1 \mid b \rightarrow 4 \mid c \rightarrow 1) &= \{a + P(a \rightarrow 0 \mid b \rightarrow 4 \mid c \rightarrow 1)\} + \\ &\quad \{b + P(a \rightarrow 1 \mid b \rightarrow 3 \mid c \rightarrow 1)\} + \\ &\quad \{c + P(a \rightarrow 1 \mid b \rightarrow 4 \mid c \rightarrow 0)\} \\ P(a \rightarrow 2 \mid b \rightarrow 3 \mid c \rightarrow 1) &= \{a + P(a \rightarrow 1 \mid b \rightarrow 3 \mid c \rightarrow 1)\} + \\ &\quad \{b + P(a \rightarrow 2 \mid b \rightarrow 2 \mid c \rightarrow 1)\} + \\ &\quad \{c + P(a \rightarrow 2 \mid b \rightarrow 3 \mid c \rightarrow 0)\} \\ P(a \rightarrow 2 \mid b \rightarrow 4 \mid c \rightarrow 0) &= \{a + P(a \rightarrow 1 \mid b \rightarrow 4 \mid c \rightarrow 0)\} + \\ &\quad \{b + P(a \rightarrow 2 \mid b \rightarrow 3 \mid c \rightarrow 0)\} \end{aligned}$$

Eventually, we'll get down to no more characters remaining.

The code below implements this algorithm.

```
1  ArrayList<String> printPerms(String s) {  
2      ArrayList<String> result = new ArrayList<String>();  
3      HashMap<Character, Integer> map = buildFreqTable(s);  
4      printPerms(map, "", s.length(), result);  
5      return result;  
6  }  
7  
8  HashMap<Character, Integer> buildFreqTable(String s) {  
9      HashMap<Character, Integer> map = new HashMap<Character, Integer>();  
10     for (char c : s.toCharArray()) {  
11         if (!map.containsKey(c)) {  
12             map.put(c, 0);  
13         }  
14         map.put(c, map.get(c) + 1);  
15     }  
16     return map;  
17 }  
18  
19 void printPerms(HashMap<Character, Integer> map, String prefix, int remaining,  
20                  ArrayList<String> result) {  
21     /* Base case. Permutation has been completed. */  
22     if (remaining == 0) {  
23         result.add(prefix);  
24         return;  
25     }  
26  
27     /* Try remaining letters for next char, and generate remaining permutations. */
```

```

28     for (Character c : map.keySet()) {
29         int count = map.get(c);
30         if (count > 0) {
31             map.put(c, count - 1);
32             printPerms(map, prefix + c, remaining - 1, result);
33             map.put(c, count);
34         }
35     }
36 }
```

In situations where the string has many duplicates, this algorithm will run a lot faster than the earlier algorithm.

- 8.9 Paren:** Implement an algorithm to print all valid (i.e., properly opened and closed) combinations of  $n$  pairs of parentheses.

#### EXAMPLE

Input: 3

Output: ((())), ((())), ((())(), ()()), ()()

pg 136

#### SOLUTION

Our first thought here might be to apply a recursive approach where we build the solution for  $f(n)$  by adding pairs of parentheses to  $f(n-1)$ . That's certainly a good instinct.

Let's consider the solution for  $n = 3$ :

((()))      ((()))      ()((()))      ((())())      ()()()

How might we build this from  $n = 2$ ?

(())      ()()

We can do this by inserting a pair of parentheses inside every existing pair of parentheses, as well as one at the beginning of the string. Any other places that we could insert parentheses, such as at the end of the string, would reduce to the earlier cases.

So, we have the following:

```

(()) -> ((())()) /* inserted pair after 1st left paren */
      -> ((())) /* inserted pair after 2nd left paren */
      -> ()((())) /* inserted pair at beginning of string */
()() -> ((())()) /* inserted pair after 1st left paren */
      -> ()((())) /* inserted pair after 2nd left paren */
      -> ()()() /* inserted pair at beginning of string */
```

But wait—we have some duplicate pairs listed. The string ()()() is listed twice.

If we're going to apply this approach, we'll need to check for duplicate values before adding a string to our list.

```

1 Set<String> generateParen(int remaining) {
2     Set<String> set = new HashSet<String>();
3     if (remaining == 0) {
4         set.add("");
5     } else {
6         Set<String> prev = generateParen(remaining - 1);
7         for (String str : prev) {
8             for (int i = 0; i < str.length(); i++) {
```

```
9         if (str.charAt(i) == '(') {
10             String s = insertInside(str, i);
11             /* Add s to set if it's not already in there. Note: HashSet
12              * automatically checks for duplicates before adding, so an explicit
13              * check is not necessary. */
14             set.add(s);
15         }
16     }
17     set.add("()" + str);
18 }
19 }
20 return set;
21 }
22
23 String insertInside(String str, int leftIndex) {
24     String left = str.substring(0, leftIndex + 1);
25     String right = str.substring(leftIndex + 1, str.length());
26     return left + "(" + right;
27 }
```

This works, but it's not very efficient. We waste a lot of time coming up with the duplicate strings.

We can avoid this duplicate string issue by building the string from scratch. Under this approach, we add left and right parens, as long as our expression stays valid.

On each recursive call, we have the index for a particular character in the string. We need to select either a left or a right paren. When can we use a left paren, and when can we use a right paren?

1. *Left Paren*: As long as we haven't used up all the left parentheses, we can always insert a left paren.
2. *Right Paren*: We can insert a right paren as long as it won't lead to a syntax error. When will we get a syntax error? We will get a syntax error if there are more right parentheses than left.

So, we simply keep track of the number of left and right parentheses allowed. If there are left parens remaining, we'll insert a left paren and recurse. If there are more right parens remaining than left (i.e., if there are more left parens in use than right parens), then we'll insert a right paren and recurse.

```
1 void addParen(ArrayList<String> list, int leftRem, int rightRem, char[] str,
2             int index) {
3     if (leftRem < 0 || rightRem < leftRem) return; // invalid state
4
5     if (leftRem == 0 && rightRem == 0) { /* Out of left and right parentheses */
6         list.add(String.valueOf(str));
7     } else {
8         str[index] = '('; // Add left and recurse
9         addParen(list, leftRem - 1, rightRem, str, index + 1);
10
11        str[index] = ')'; // Add right and recurse
12        addParen(list, leftRem, rightRem - 1, str, index + 1);
13    }
14 }
15
16 ArrayList<String> generateParens(int count) {
17     char[] str = new char[count*2];
18     ArrayList<String> list = new ArrayList<String>();
19     addParen(list, count, count, str, 0);
20     return list;
21 }
```

Because we insert left and right parentheses at each index in the string, and we never repeat an index, each string is guaranteed to be unique.

- 8.10 Paint Fill:** Implement the “paint fill” function that one might see on many image editing programs. That is, given a screen (represented by a two-dimensional array of colors), a point, and a new color, fill in the surrounding area until the color changes from the original color.

pg 136

## SOLUTION

First, let’s visualize how this method works. When we call `paintFill` (i.e., “click” paint fill in the image editing application) on, say, a green pixel, we want to “bleed” outwards. Pixel by pixel, we expand outwards by calling `paintFill` on the surrounding pixel. When we hit a pixel that is not green, we stop.

We can implement this algorithm recursively:

```

1 enum Color { Black, White, Red, Yellow, Green }
2
3 boolean PaintFill(Color[][] screen, int r, int c, Color ncolor) {
4     if (screen[r][c] == ncolor) return false;
5     return PaintFill(screen, r, c, screen[r][c], ncolor);
6 }
7
8 boolean PaintFill(Color[][] screen, int r, int c, Color ocolor, Color ncolor) {
9     if (r < 0 || r >= screen.length || c < 0 || c >= screen[0].length) {
10         return false;
11     }
12
13     if (screen[r][c] == ocolor) {
14         screen[r][c] = ncolor;
15         PaintFill(screen, r - 1, c, ocolor, ncolor); // up
16         PaintFill(screen, r + 1, c, ocolor, ncolor); // down
17         PaintFill(screen, r, c - 1, ocolor, ncolor); // left
18         PaintFill(screen, r, c + 1, ocolor, ncolor); // right
19     }
20     return true;
21 }
```

If you used the variable names `x` and `y` to implement this, be careful about the ordering of the variables in `screen[y][x]`. Because `x` represents the *horizontal* axis (that is, it’s left to right), it actually corresponds to the column number, not the row number. The value of `y` equals the number of rows. This is a very easy place to make a mistake in an interview, as well as in your daily coding. It’s typically clearer to use `row` and `column` instead, as we’ve done here.

Does this algorithm seem familiar? It should! This is essentially depth-first search on a graph. At each pixel, we are searching outwards to each surrounding pixel. We stop once we’ve fully traversed all the surrounding pixels of this color.

We could alternatively implement this using breadth-first search.

- 8.11 Coins:** Given an infinite number of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), write code to calculate the number of ways of representing n cents.

pg 136

### SOLUTION

This is a recursive problem, so let's figure out how to compute `makeChange(n)` using prior solutions (i.e., subproblems).

Let's say  $n = 100$ . We want to compute the number of ways of making change for 100 cents. What is the relationship between this problem and its subproblems?

We know that making change for 100 cents will involve either 0, 1, 2, 3, or 4 quarters. So:

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(100 using 1 quarter) +
                  makeChange(100 using 2 quarters) +
                  makeChange(100 using 3 quarters) +
                  makeChange(100 using 4 quarters)
```

Inspecting this further, we can see that some of these problems reduce. For example, `makeChange(100 using 1 quarter)` will equal `makeChange(75 using 0 quarters)`. This is because, if we must use exactly one quarter to make change for 100 cents, then our only remaining choices involve making change for the remaining 75 cents.

We can apply the same logic to `makeChange(100 using 2 quarters)`, `makeChange(100 using 3 quarters)` and `makeChange(100 using 4 quarters)`. We have thus reduced the above statement to the following.

```
makeChange(100) = makeChange(100 using 0 quarters) +
                  makeChange(75 using 0 quarters) +
                  makeChange(50 using 0 quarters) +
                  makeChange(25 using 0 quarters) +
                  1
```

Note that the final statement from above, `makeChange(100 using 4 quarters)`, equals 1. We call this "fully reduced."

Now what? We've used up all our quarters, so now we can start applying our next biggest denomination: dimes.

Our approach for quarters applies to dimes as well, but we apply this for *each* of the four of five parts of the above statement. So, for the first part, we get the following statements:

```
makeChange(100 using 0 quarters) = makeChange(100 using 0 quarters, 0 dimes) +
                                    makeChange(100 using 0 quarters, 1 dime) +
                                    makeChange(100 using 0 quarters, 2 dimes) +
                                    ...
                                    makeChange(100 using 0 quarters, 10 dimes)
```

```
makeChange(75 using 0 quarters) = makeChange(75 using 0 quarters, 0 dimes) +
                                    makeChange(75 using 0 quarters, 1 dime) +
                                    makeChange(75 using 0 quarters, 2 dimes) +
                                    ...
                                    makeChange(75 using 0 quarters, 7 dimes)
```

```
makeChange(50 using 0 quarters) = makeChange(50 using 0 quarters, 0 dimes) +
                                    makeChange(50 using 0 quarters, 1 dime) +
                                    makeChange(50 using 0 quarters, 2 dimes) +
```

```

    ...
    makeChange(50 using 0 quarters, 5 dimes)

makeChange(25 using 0 quarters) = makeChange(25 using 0 quarters, 0 dimes) +
                                makeChange(25 using 0 quarters, 1 dime) +
                                makeChange(25 using 0 quarters, 2 dimes)

```

Each one of these, in turn, expands out once we start applying nickels. We end up with a tree-like recursive structure where each call expands out to four or more calls.

The base case of our recursion is the fully reduced statement. For example, `makeChange(50 using 0 quarters, 5 dimes)` is fully reduced to 1, since 5 dimes equals 50 cents.

This leads to a recursive algorithm that looks like this:

```

1  int makeChange(int amount, int[] denoms, int index) {
2      if (index >= denoms.length - 1) return 1; // last denom
3      int denomAmount = denoms[index];
4      int ways = 0;
5      for (int i = 0; i * denomAmount <= amount; i++) {
6          int amountRemaining = amount - i * denomAmount;
7          ways += makeChange(amountRemaining, denoms, index + 1);
8      }
9      return ways;
10 }
11
12 int makeChange(int n) {
13     int[] denoms = {25, 10, 5, 1};
14     return makeChange(n, denoms, 0);
15 }

```

This works, but it's not as optimal as it could be. The issue is that we will be recursively calling `makeChange` several times for the same values of `amount` and `index`.

We can resolve this issue by storing the previously computed values. We'll need to store a mapping from each pair (`amount`, `index`) to the precomputed result.

```

1  int makeChange(int n) {
2      int[] denoms = {25, 10, 5, 1};
3      int[][] map = new int[n + 1][denoms.length]; // precomputed vals
4      return makeChange(n, denoms, 0, map);
5  }
6
7  int makeChange(int amount, int[] denoms, int index, int[][] map) {
8      if (map[amount][index] > 0) { // retrieve value
9          return map[amount][index];
10     }
11     if (index >= denoms.length - 1) return 1; // one denom remaining
12     int denomAmount = denoms[index];
13     int ways = 0;
14     for (int i = 0; i * denomAmount <= amount; i++) {
15         // go to next denom, assuming i coins of denomAmount
16         int amountRemaining = amount - i * denomAmount;
17         ways += makeChange(amountRemaining, denoms, index + 1, map);
18     }
19     map[amount][index] = ways;
20     return ways;
21 }

```

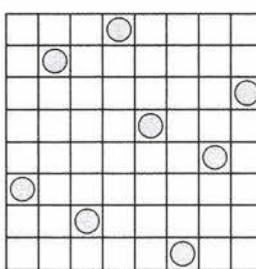
Note that we've used a two-dimensional array of integers to store the previously computed values. This is simpler, but takes up a little extra space. Alternatively, we could use an actual hash table that maps from amount to a new hash table, which then maps from denom to the precomputed value. There are other alternative data structures as well.

- 8.12 Eight Queens:** Write an algorithm to print all ways of arranging eight queens on an 8x8 chess board so that none of them share the same row, column, or diagonal. In this case, "diagonal" means all diagonals, not just the two that bisect the board.

pg 136

### SOLUTION

We have eight queens which must be lined up on an 8x8 chess board such that none share the same row, column or diagonal. So, we know that each row and column (and diagonal) must be used exactly once.



A "Solved" Board with 8 Queens

Picture the queen that is placed last, which we'll assume is on row 8. (This is an okay assumption to make since the ordering of placing the queens is irrelevant.) On which cell in row 8 is this queen? There are eight possibilities, one for each column.

So if we want to know all the valid ways of arranging 8 queens on an 8x8 chess board, it would be:

ways to arrange 8 queens on an 8x8 board =  
ways to arrange 8 queens on an 8x8 board with queen at (7, 0) +  
ways to arrange 8 queens on an 8x8 board with queen at (7, 1) +  
ways to arrange 8 queens on an 8x8 board with queen at (7, 2) +  
ways to arrange 8 queens on an 8x8 board with queen at (7, 3) +  
ways to arrange 8 queens on an 8x8 board with queen at (7, 4) +  
ways to arrange 8 queens on an 8x8 board with queen at (7, 5) +  
ways to arrange 8 queens on an 8x8 board with queen at (7, 6) +  
ways to arrange 8 queens on an 8x8 board with queen at (7, 7)

We can compute each one of these using a very similar approach:

ways to arrange 8 queens on an 8x8 board with queen at (7, 3) =  
ways to ... with queens at (7, 3) and (6, 0) +  
ways to ... with queens at (7, 3) and (6, 1) +  
ways to ... with queens at (7, 3) and (6, 2) +  
ways to ... with queens at (7, 3) and (6, 4) +  
ways to ... with queens at (7, 3) and (6, 5) +  
ways to ... with queens at (7, 3) and (6, 6) +  
ways to ... with queens at (7, 3) and (6, 7)

Note that we don't need to consider combinations with queens at (7, 3) and (6, 3), since this is a violation of the requirement that every queen is in its own row, column and diagonal.

Implementing this is now reasonably straightforward.

```

1 int GRID_SIZE = 8;
2
3 void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
4     if (row == GRID_SIZE) { // Found valid placement
5         results.add(columns.clone());
6     } else {
7         for (int col = 0; col < GRID_SIZE; col++) {
8             if (checkValid(columns, row, col)) {
9                 columns[row] = col; // Place queen
10                placeQueens(row + 1, columns, results);
11            }
12        }
13    }
14 }
15
16 /* Check if (row1, column1) is a valid spot for a queen by checking if there is a
17 * queen in the same column or diagonal. We don't need to check it for queens in
18 * the same row because the calling placeQueen only attempts to place one queen at
19 * a time. We know this row is empty. */
20 boolean checkValid(Integer[] columns, int row1, int column1) {
21     for (int row2 = 0; row2 < row1; row2++) {
22         int column2 = columns[row2];
23         /* Check if (row2, column2) invalidates (row1, column1) as a
24          * queen spot. */
25
26         /* Check if rows have a queen in the same column */
27         if (column1 == column2) {
28             return false;
29         }
30
31         /* Check diagonals: if the distance between the columns equals the distance
32          * between the rows, then they're in the same diagonal. */
33         int columnDistance = Math.abs(column2 - column1);
34
35         /* row1 > row2, so no need for abs */
36         int rowDistance = row1 - row2;
37         if (columnDistance == rowDistance) {
38             return false;
39         }
40     }
41     return true;
42 }
```

Observe that since each row can only have one queen, we don't need to store our board as a full 8x8 matrix. We only need a single array where  $\text{column}[r] = c$  indicates that row  $r$  has a queen at column  $c$ .

- 8.13 Stack of Boxes:** You have a stack of  $n$  boxes, with widths  $w_1$ , heights  $h_1$ , and depths  $d_1$ . The boxes cannot be rotated and can only be stacked on top of one another if each box in the stack is strictly larger than the box above it in width, height, and depth. Implement a method to compute the height of the tallest possible stack. The height of a stack is the sum of the heights of each box.

pg 136

### SOLUTION

To tackle this problem, we need to recognize the relationship between the different subproblems.

#### Solution #1

Imagine we had the following boxes:  $b_1, b_2, \dots, b_n$ . The biggest stack that we can build with all the boxes equals the max of (biggest stack with bottom  $b_1$ , biggest stack with bottom  $b_2$ , ..., biggest stack with bottom  $b_n$ ). That is, if we experimented with each box as a bottom and built the biggest stack possible with each, we would find the biggest stack possible.

But, how would we find the biggest stack with a particular bottom? Essentially the same way. We experiment with different boxes for the second level, and so on for each level.

Of course, we only experiment with valid boxes. If  $b_s$  is bigger than  $b_1$ , then there's no point in trying to build a stack that looks like  $\{b_1, b_s, \dots\}$ . We already know  $b_1$  can't be below  $b_s$ .

We can perform a small optimization here. The requirements of this problem stipulate that the lower boxes must be strictly greater than the higher boxes in all dimensions. Therefore, if we sort (descending order) the boxes on a dimension—any dimension—then we know we don't have to look backwards in the list. The box  $b_1$  cannot be on top of box  $b_s$ , since its height (or whatever dimension we sorted on) is greater than  $b_s$ 's height.

The code below implements this algorithm recursively.

```
1 int createStack(ArrayList<Box> boxes) {  
2     /* Sort in descending order by height. */  
3     Collections.sort(boxes, new BoxComparator());  
4     int maxHeight = 0;  
5     for (int i = 0; i < boxes.size(); i++) {  
6         int height = createStack(boxes, i);  
7         maxHeight = Math.max(maxHeight, height);  
8     }  
9     return maxHeight;  
10 }  
11  
12 int createStack(ArrayList<Box> boxes, int bottomIndex) {  
13     Box bottom = boxes.get(bottomIndex);  
14     int maxHeight = 0;  
15     for (int i = bottomIndex + 1; i < boxes.size(); i++) {  
16         if (boxes.get(i).canBeAbove(bottom)) {  
17             int height = createStack(boxes, i);  
18             maxHeight = Math.max(height, maxHeight);  
19         }  
20     }  
21     maxHeight += bottom.height;  
22     return maxHeight;  
23 }  
24  
25 class BoxComparator implements Comparator<Box> {
```

```

26     @Override
27     public int compare(Box x, Box y){
28         return y.height - x.height;
29     }
30 }
```

The problem in this code is that it gets very inefficient. We try to find the best solution that looks like  $\{b_3, b_4, \dots\}$  even though we may have already found the best solution with  $b_4$  at the bottom. Instead of generating these solutions from scratch, we can cache these results using memoization.

```

1  int createStack(ArrayList<Box> boxes) {
2     Collections.sort(boxes, new BoxComparator());
3     int maxHeight = 0;
4     int[] stackMap = new int[boxes.size()];
5     for (int i = 0; i < boxes.size(); i++) {
6         int height = createStack(boxes, i, stackMap);
7         maxHeight = Math.max(maxHeight, height);
8     }
9     return maxHeight;
10 }
11
12 int createStack(ArrayList<Box> boxes, int bottomIndex, int[] stackMap) {
13     if (bottomIndex < boxes.size() && stackMap[bottomIndex] > 0) {
14         return stackMap[bottomIndex];
15     }
16
17     Box bottom = boxes.get(bottomIndex);
18     int maxHeight = 0;
19     for (int i = bottomIndex + 1; i < boxes.size(); i++) {
20         if (boxes.get(i).canBeAbove(bottom)) {
21             int height = createStack(boxes, i, stackMap);
22             maxHeight = Math.max(height, maxHeight);
23         }
24     }
25     maxHeight += bottom.height;
26     stackMap[bottomIndex] = maxHeight;
27     return maxHeight;
28 }
```

Because we're only mapping from an index to a height, we can just use an integer array for our "hash table."

Be very careful here with what each spot in the hash table represents. In this code, `stackMap[i]` represents the tallest stack with box `i` at the bottom. Before pulling the value from the hash table, you have to ensure that box `i` can be placed on top of the current bottom.

It helps to keep the line that recalls from the hash table symmetric with the one that inserts. For example, in this code, we recall from the hash table with `bottomIndex` at the start of the method. We insert into the hash table with `bottomIndex` at the end.

### Solution #2

Alternatively, we can think about the recursive algorithm as making a choice, at each step, whether to put a particular box in the stack. (We will again sort our boxes in descending order by a dimension, such as height.)

First, we choose whether or not to put box 0 in the stack. Take one recursive path with box 0 at the bottom and one recursive path without box 0. Return the better of the two options.

Then, we choose whether or not to put box 1 in the stack. Take one recursive path with box 1 at the bottom and one path without box 1. Return the better of the two options.

We will again use memoization to cache the height of the tallest stack with a particular bottom.

```
1 int createStack(ArrayList<Box> boxes) {
2     Collections.sort(boxes, new BoxComparator());
3     int[] stackMap = new int[boxes.size()];
4     return createStack(boxes, null, 0, stackMap);
5 }
6
7 int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {
8     if (offset >= boxes.size()) return 0; // Base case
9
10    /* height with this bottom */
11    Box newBottom = boxes.get(offset);
12    int heightWithBottom = 0;
13    if (bottom == null || newBottom.canBeAbove(bottom)) {
14        if (stackMap[offset] == 0) {
15            stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);
16            stackMap[offset] += newBottom.height;
17        }
18        heightWithBottom = stackMap[offset];
19    }
20
21    /* without this bottom */
22    int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);
23
24    /* Return better of two options. */
25    return Math.max(heightWithBottom, heightWithoutBottom);
26 }
```

Again, pay close attention to when you recall and insert values into the hash table. It's typically best if these are symmetric, as they are in lines 15 and 16-18.

**8.14 Boolean Evaluation:** Given a boolean expression consisting of the symbols 0 (false), 1 (true), & (AND), | (OR), and ^ (XOR), and a desired boolean result value `result`, implement a function to count the number of ways of parenthesizing the expression such that it evaluates to `result`. The expression should be fully parenthesized (e.g., `(0)^1`) but not extraneously (e.g., `((0))^1`).

EXAMPLE

```
countEval("1^0|0|1", false) -> 2
countEval("0&0&0&1^1|0", true) -> 10
```

pg 136

### SOLUTION

As in other recursive problems, the key to this problem is to figure out the relationship between a problem and its subproblems.

#### Brute Force

Consider an expression like `0^0&0^1|1` and the target result `true`. How can we break down `countEval(0^0&0^1|1, true)` into smaller problems?

We could just essentially iterate through each possible place to put a parenthesis.

```
countEval(0^0&0^1|1, true) =
    countEval(0^0&0^1|1 where paren around char 1, true)
+ countEval(0^0&0^1|1 where paren around char 3, true)
+ countEval(0^0&0^1|1 where paren around char 5, true)
+ countEval(0^0&0^1|1 where paren around char 7, true)
```

Now what? Let's look at just one of those expressions—the paren around char 3. This gives us  $(0^0) \& (0^1)$ .

In order to make that expression true, both the left and right sides must be true. So:

```
left = "0^0"
right = "0^1|1"
countEval(left & right, true) = countEval(left, true) * countEval(right, true)
```

The reason we multiply the results of the left and right sides is that each result from the two sides can be paired up with each other to form a unique combination.

Each of those terms can now be decomposed into smaller problems in a similar process.

What happens when we have an “|” (OR)? Or an “^” (XOR)?

If it's an OR, then either the left or the right side must be true—or both.

```
countEval(left | right, true) = countEval(left, true) * countEval(right, false)
                                + countEval(left, false) * countEval(right, true)
                                + countEval(left, true) * countEval(right, true)
```

If it's an XOR, then the left or the right side can be true, but not both.

```
countEval(left ^ right, true) = countEval(left, true) * countEval(right, false)
                                + countEval(left, false) * countEval(right, true)
```

What if we were trying to make the result `false` instead? We can switch up the logic from above:

```
countEval(left & right, false) = countEval(left, true) * countEval(right, false)
                                + countEval(left, false) * countEval(right, true)
                                + countEval(left, false) * countEval(right, false)
countEval(left | right, false) = countEval(left, false) * countEval(right, false)
countEval(left ^ right, false) = countEval(left, false) * countEval(right, false)
                                + countEval(left, true) * countEval(right, true)
```

Alternatively, we can just use the same logic from above and subtract it out from the total number of ways of evaluating the expression.

```
totalEval(left) = countEval(left, true) + countEval(left, false)
totalEval(right) = countEval(right, true) + countEval(right, false)
totalEval(expression) = totalEval(left) * totalEval(right)
countEval(expression, false) = totalEval(expression) - countEval(expression, true)
```

This makes the code a bit more concise.

```
1 int countEval(String s, boolean result) {
2     if (s.length() == 0) return 0;
3     if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
4
5     int ways = 0;
6     for (int i = 1; i < s.length(); i += 2) {
7         char c = s.charAt(i);
8         String left = s.substring(0, i);
9         String right = s.substring(i + 1, s.length());
10
11        /* Evaluate each side for each result. */
12        int leftTrue = countEval(left, true);
13        int leftFalse = countEval(left, false);
14        int rightTrue = countEval(right, true);
```

```
15     int rightFalse = countEval(right, false);
16     int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18     int totalTrue = 0;
19     if (c == '^') { // required: one true and one false
20         totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21     } else if (c == '&') { // required: both true
22         totalTrue = leftTrue * rightTrue;
23     } else if (c == '|') { // required: anything but both false
24         totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25             leftTrue * rightFalse;
26     }
27
28     int subWays = result ? totalTrue : total - totalTrue;
29     ways += subWays;
30 }
31
32     return ways;
33 }
34
35 boolean stringToBool(String c) {
36     return c.equals("1") ? true : false;
37 }
```

Note that the tradeoff of computing the `false` results from the `true` ones, and of computing the `{leftTrue, rightTrue, leftFalse, and rightFalse}` values upfront, is a small amount of extra work in some cases. For example, if we're looking for the ways that an AND (`&`) can result in `true`, we never would have needed the `leftFalse` and `rightFalse` results. Likewise, if we're looking for the ways that an OR (`|`) can result in `false`, we never would have needed the `leftTrue` and `rightTrue` results.

Our current code is blind to what we do and don't actually need to do and instead just computes all of the values. This is probably a reasonable tradeoff to make (especially given the constraints of whiteboard coding) as it makes our code substantially shorter and less tedious to write. Whichever approach you make, you should discuss the tradeoffs with your interviewer.

That said, there are more important optimizations we can make.

### Optimized Solutions

If we follow the recursive path, we'll note that we end up doing the same computation repeatedly.

Consider the expression `0^0&0^1|1` and these recursion paths:

- Add parens around char 1. `(0)^((0)&(0^1|1))`
  - » Add parens around char 3. `(0)^((0)&(0^1|1))`
- Add parens around char 3. `(0^0)&(0^1|1)`
  - » Add parens around char 1. `((0)^((0)))&(0^1|1)`

Although these two expressions are different, they have a similar component: `(0^1|1)`. We should reuse our effort on this.

We can do this by using memoization, or a hash table. We just need to store the result of `countEval(expression, result)` for each expression and result. If we see an expression that we've calculated before, we just return it from the cache.

```
1 int countEval(String s, boolean result, HashMap<String, Integer> memo) {
2     if (s.length() == 0) return 0;
3     if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
```

```

4  if (memo.containsKey(result + s)) return memo.get(result + s);
5
6  int ways = 0;
7
8  for (int i = 1; i < s.length(); i += 2) {
9      char c = s.charAt(i);
10     String left = s.substring(0, i);
11     String right = s.substring(i + 1, s.length());
12     int leftTrue = countEval(left, true, memo);
13     int leftFalse = countEval(left, false, memo);
14     int rightTrue = countEval(right, true, memo);
15     int rightFalse = countEval(right, false, memo);
16     int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);
17
18     int totalTrue = 0;
19     if (c == '^') {
20         totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
21     } else if (c == '&') {
22         totalTrue = leftTrue * rightTrue;
23     } else if (c == '|') {
24         totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
25                     leftTrue * rightFalse;
26     }
27
28     int subWays = result ? totalTrue : total - totalTrue;
29     ways += subWays;
30 }
31
32 memo.put(result + s, ways);
33 return ways;
34 }
```

The added benefit of this is that we could actually end up with the same substring in multiple parts of the expression. For example, an expression like  $0^1^0 \& 0^1^0$  has two instances of  $0^1^0$ . By caching the result of the substring value in a memoization table, we'll get to reuse the result for the right part of the expression after computing it for the left.

There is one further optimization we can make, but it's far beyond the scope of the interview. There is a closed form expression for the number of ways of parenthesizing an expression, but you wouldn't be expected to know it. It is given by the Catalan numbers, where  $n$  is the number of operators:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

We could use this to compute the total ways of evaluating the expression. Then, rather than computing `leftTrue` and `leftFalse`, we just compute one of those and calculate the other using the Catalan numbers. We would do the same thing for the right side.

# 9

---

## Solutions to System Design and Scalability

---

- 9.1 Stock Data:** Imagine you are building some sort of service that will be called by up to 1,000 client applications to get simple end-of-day stock price information (open, close, high, low). You may assume that you already have the data, and you can store it in any format you wish. How would you design the client-facing service that provides the information to client applications? You are responsible for the development, rollout, and ongoing monitoring and maintenance of the feed. Describe the different methods you considered and why you would recommend your approach. Your service can use any technologies you wish, and can distribute the information to the client applications in any mechanism you choose.

pg 144

### SOLUTION

From the statement of the problem, we want to focus on how we actually distribute the information to clients. We can assume that we have some scripts that magically collect the information.

We want to start off by thinking about what the different aspects we should consider in a given proposal are:

- *Client Ease of Use:* We want the service to be easy for the clients to implement and useful for them.
- *Ease for Ourselves:* This service should be as easy as possible for us to implement, as we shouldn't impose unnecessary work on ourselves. We need to consider in this not only the cost of implementing, but also the cost of maintenance.
- *Flexibility for Future Demands:* This problem is stated in a "what would you do in the real world" way, so we should think like we would in a real-world problem. Ideally, we do not want to overly constrain ourselves in the implementation, such that we can't be flexible if the requirements or demands change.
- *Scalability and Efficiency:* We should be mindful of the efficiency of our solution, so as not to overly burden our service.

With this framework in mind, we can consider various proposals.

### Proposal #1

One option is that we could keep the data in simple text files and let clients download the data through some sort of FTP server. This would be easy to maintain in some sense, since files can be easily viewed and backed up, but it would require more complex parsing to do any sort of query. And, if additional data were added to our text file, it might break the clients' parsing mechanism.

### Proposal #2

We could use a standard SQL database, and let the clients plug directly into that. This would provide the following benefits:

- Facilitates an easy way for the clients to do query processing over the data, in case there are additional features we need to support. For example, we could easily and efficiently perform a query such as “return all stocks having an open price greater than N and a closing price less than M.”
- Rolling back, backing up data, and security could be provided using standard database features. We don’t have to “reinvent the wheel,” so it’s easy for us to implement.
- Reasonably easy for the clients to integrate into existing applications. SQL integration is a standard feature in software development environments.

What are the disadvantages of using a SQL database?

- It’s much heavier weight than we really need. We don’t necessarily need all the complexity of a SQL backend to support a feed of a few bits of information.
- It’s difficult for humans to be able to read it, so we’ll likely need to implement an additional layer to view and maintain the data. This increases our implementation costs.
- Security: While a SQL database offers pretty well defined security levels, we would still need to be very careful to not give clients access that they shouldn’t have. Additionally, even if clients aren’t doing anything “malicious,” they might perform expensive and inefficient queries, and our servers would bear the costs of that.

These disadvantages don’t mean that we shouldn’t provide SQL access. Rather, they mean that we should be aware of the disadvantages.

### Proposal #3

XML is another great option for distributing the information. Our data has fixed format and fixed size: company\_name, open, high, low, closing price. The XML could look like this:

```
1  <root>
2      <date value="2008-10-12">
3          <company name="foo">
4              <open>126.23</open>
5              <high>130.27</high>
6              <low>122.83</low>
7              <closingPrice>127.30</closingPrice>
8          </company>
9          <company name="bar">
10             <open>52.73</open>
11             <high>60.27</high>
12             <low>50.29</low>
13             <closingPrice>54.91</closingPrice>
14         </company>
15     </date>
16     <date value="2008-10-11"> . . . </date>
17 </root>
```

The advantages of this approach include the following:

- It’s very easy to distribute, and it can also be easily read by both machines and humans. This is one reason that XML is a standard data model to share and distribute data.
- Most languages have a library to perform XML parsing, so it’s reasonably easy for clients to implement.

- We can add new data to the XML file by adding additional nodes. This would not break the client's parser (provided they have implemented their parser in a reasonable way).
- Since the data is being stored as XML files, we can use existing tools for backing up the data. We don't need to implement our own backup tool.

The disadvantages may include:

- This solution sends the clients all the information, even if they only want part of it. It is inefficient in that way.
- Performing any queries on the data requires parsing the entire file.

Regardless of which solution we use for data storage, we could provide a web service (e.g., SOAP) for client data access. This adds a layer to our work, but it can provide additional security, and it may even make it easier for clients to integrate the system.

However—and this is a pro and a con—clients will be limited to grabbing the data only how we expect or want them to. By contrast, in a pure SQL implementation, clients could query for the highest stock price, even if this wasn't a procedure we "expected" them to need.

So which one of these would we use? There's no clear answer. The pure text file solution is probably a bad choice, but you can make a compelling argument for the SQL or XML solution, with or without a web service.

The goal of a question like this is not to see if you get the "correct" answer (there is no single correct answer). Rather, it's to see how you design a system, and how you evaluate trade-offs.

- 9.2 Social Network:** How would you design the data structures for a very large social network like Facebook or LinkedIn? Describe how you would design an algorithm to show the shortest path between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

pg 145

### SOLUTION

A good way to approach this problem is to remove some of the constraints and solve it for that situation first.

#### Step 1: Simplify the Problem—Forget About the Millions of Users

First, let's forget that we're dealing with millions of users. Design this for the simple case.

We can construct a graph by treating each person as a node and letting an edge between two nodes indicate that the two users are friends.

If I wanted to find the path between two people, I could start with one person and do a simple breadth-first search.

Why wouldn't a depth-first search work well? First, depth-first search would just find a path. It wouldn't necessarily find the shortest path. Second, even if we just needed any path, it would be very inefficient. Two users might be only one degree of separation apart, but I could search millions of nodes in their "subtrees" before finding this relatively immediate connection.

Alternatively, I could do what's called a bidirectional breadth-first search. This means doing two breadth-first searches, one from the source and one from the destination. When the searches collide, we know we've found a path.

In the implementation, we'll use two classes to help us. BFSData holds the data we need for a breadth-first search, such as the `isVisited` hash table and the `toVisit` queue. PathNode will represent the path as we're searching it, storing each Person and the `previousNode` we visited in this path.

```

1  LinkedList<Person> findPathBiBFS(HashMap<Integer, Person> people, int source,
2          int destination) {
3      BFSData sourceData = new BFSData(people.get(source));
4      BFSData destData = new BFSData(people.get(destination));
5
6      while (!sourceData.isFinished() && !destData.isFinished()) {
7          /* Search out from source. */
8          Person collision = searchLevel(people, sourceData, destData);
9          if (collision != null) {
10              return mergePaths(sourceData, destData, collision.getID());
11          }
12
13          /* Search out from destination. */
14          collision = searchLevel(people, destData, sourceData);
15          if (collision != null) {
16              return mergePaths(sourceData, destData, collision.getID());
17          }
18      }
19      return null;
20  }
21
22 /* Search one level and return collision, if any. */
23 Person searchLevel(HashMap<Integer, Person> people, BFSData primary,
24                     BFSData secondary) {
25     /* We only want to search one level at a time. Count how many nodes are
26      * currently in the primary's level and only do that many nodes. We'll continue
27      * to add nodes to the end. */
28     int count = primary.toVisit.size();
29     for (int i = 0; i < count; i++) {
30         /* Pull out first node. */
31         PathNode pathNode = primary.toVisit.poll();
32         int personId = pathNode.getPerson().getID();
33
34         /* Check if it's already been visited. */
35         if (secondary.visited.containsKey(personId)) {
36             return pathNode.getPerson();
37         }
38
39         /* Add friends to queue. */
40         Person person = pathNode.getPerson();
41         ArrayList<Integer> friends = person.getFriends();
42         for (int friendId : friends) {
43             if (!primary.visited.containsKey(friendId)) {
44                 Person friend = people.get(friendId);
45                 PathNode next = new PathNode(friend, pathNode);
46                 primary.visited.put(friendId, next);
47                 primary.toVisit.add(next);
48             }
49         }
50     }
51     return null;
52 }
53

```

```
54 /* Merge paths where searches met at connection. */
55 LinkedList<Person> mergePaths(BFSData bfs1, BFSData bfs2, int connection) {
56     PathNode end1 = bfs1.visited.get(connection); // end1 -> source
57     PathNode end2 = bfs2.visited.get(connection); // end2 -> dest
58     LinkedList<Person> pathOne = end1.collapse(false);
59     LinkedList<Person> pathTwo = end2.collapse(true); // reverse
60     pathTwo.removeFirst(); // remove connection
61     pathOne.addAll(pathTwo); // add second path
62     return pathOne;
63 }
64
65 class PathNode {
66     private Person person = null;
67     private PathNode previousNode = null;
68     public PathNode(Person p, PathNode previous) {
69         person = p;
70         previousNode = previous;
71     }
72
73     public Person getPerson() { return person; }
74
75     public LinkedList<Person> collapse(boolean startsWithRoot) {
76         LinkedList<Person> path = new LinkedList<Person>();
77         PathNode node = this;
78         while (node != null) {
79             if (startsWithRoot) {
80                 path.addLast(node.person);
81             } else {
82                 path.addFirst(node.person);
83             }
84             node = node.previousNode;
85         }
86         return path;
87     }
88 }
89
90 class BFSData {
91     public Queue<PathNode> toVisit = new LinkedList<PathNode>();
92     public HashMap<Integer, PathNode> visited =
93         new HashMap<Integer, PathNode>();
94
95     public BFSData(Person root) {
96         PathNode sourcePath = new PathNode(root, null);
97         toVisit.add(sourcePath);
98         visited.put(root.getID(), sourcePath);
99     }
100
101    public boolean isFinished() {
102        return toVisit.isEmpty();
103    }
104 }
```

Many people are surprised that this is faster. Some quick math can explain why.

Suppose every person has  $k$  friends, and node S and node D have a friend C in common.

- Traditional breadth-first search from S to D: We go through roughly  $k+k*k$  nodes: each of S's  $k$  friends, and then each of their  $k$  friends.

- Bidirectional breadth-first search: We go through  $2k$  nodes: each of S's  $k$  friends and each of D's  $k$  friends. Of course,  $2k$  is much less than  $k+k^2$ .

Generalizing this to a path of length  $q$ , we have this:

- BFS:  $O(k^q)$
- Bidirectional BFS:  $O(k^{q/2} + k^{q/2})$ , which is just  $O(k^{q/2})$

If you imagine a path like A->B->C->D->E where each person has 100 friends, this is a big difference. BFS will require looking at 100 million ( $100^4$ ) nodes. A bidirectional BFS will require looking at only 20,000 nodes ( $2 \times 100^2$ ).

A bidirectional BFS will generally be faster than the traditional BFS. However, it requires actually having access to both the source node and the destination nodes, which is not always the case.

## Step 2: Handle the Millions of Users

When we deal with a service the size of LinkedIn or Facebook, we cannot possibly keep all of our data on one machine. That means that our simple Person data structure from above doesn't quite work—our friends may not live on the same machine as we do. Instead, we can replace our list of friends with a list of their IDs, and traverse as follows:

- For each friend ID: int machine\_index = getMachineIDForUser(personID);
- Go to machine #machine\_index
- On that machine, do: Person friend = getPersonWithID(person\_id);

The code below outlines this process. We've defined a class Server, which holds a list of all the machines, and a class Machine, which represents a single machine. Both classes have hash tables to efficiently lookup data.

```

1  class Server {
2      HashMap<Integer, Machine> machines = new HashMap<Integer, Machine>();
3      HashMap<Integer, Integer> personToMachineMap = new HashMap<Integer, Integer>();
4
5      public Machine getMachineWithId(int machineID) {
6          return machines.get(machineID);
7      }
8
9      public int getMachineIDForUser(int personID) {
10         Integer machineID = personToMachineMap.get(personID);
11         return machineID == null ? -1 : machineID;
12     }
13
14     public Person getPersonWithID(int personID) {
15         Integer machineID = personToMachineMap.get(personID);
16         if (machineID == null) return null;
17
18         Machine machine = getMachineWithId(machineID);
19         if (machine == null) return null;
20
21         return machine.getPersonWithID(personID);
22     }
23 }
24
25 class Person {
```

```
26  private ArrayList<Integer> friends = new ArrayList<Integer>();  
27  private int personID;  
28  private String info;  
29  
30  public Person(int id) { this.personID = id; }  
31  public String getInfo() { return info; }  
32  public void setInfo(String info) { this.info = info; }  
33  public ArrayList<Integer> getFriends() { return friends; }  
34  public int getID() { return personID; }  
35  public void addFriend(int id) { friends.add(id); }  
36 }
```

There are more optimizations and follow-up questions here than we could possibly discuss, but here are just a few possibilities.

### Optimization: Reduce machine jumps

Jumping from one machine to another is expensive. Instead of randomly jumping from machine to machine with each friend, try to batch these jumps—e.g., if five of my friends live on one machine, I should look them up all at once.

### Optimization: Smart division of people and machines

People are much more likely to be friends with people who live in the same country as they do. Rather than randomly dividing people across machines, try to divide them by country, city, state, and so on. This will reduce the number of jumps.

### Question: Breadth-first search usually requires “marking” a node as visited. How do you do that in this case?

Usually, in BFS, we mark a node as visited by setting a `visited` flag in its node class. Here, we don't want to do that. There could be multiple searches going on at the same time, so it's a bad idea to just edit our data.

Instead, we could mimic the marking of nodes with a hash table to look up a node id and determine whether it's been visited.

### Other Follow-Up Questions:

- In the real world, servers fail. How does this affect you?
- How could you take advantage of caching?
- Do you search until the end of the graph (infinite)? How do you decide when to give up?
- In real life, some people have more friends of friends than others, and are therefore more likely to make a path between you and someone else. How could you use this data to pick where to start traversing?

These are just a few of the follow-up questions you or the interviewer could raise. There are many others.

### 9.3 Web Crawler:

If you were designing a web crawler, how would you avoid getting into infinite loops?

pg 145

### SOLUTION

The first thing to ask ourselves in this problem is how an infinite loop might occur. The simplest answer is that, if we picture the web as a graph of links, an infinite loop will occur when a cycle occurs.

To prevent infinite loops, we just need to detect cycles. One way to do this is to create a hash table where we set  $\text{hash}[v]$  to true after we visit page  $v$ .

We can crawl the web using breadth-first search. Each time we visit a page, we gather all its links and insert them at the end of a queue. If we've already visited a page, we ignore it.

This is great—but what does it mean to visit page  $v$ ? Is page  $v$  defined based on its content or its URL?

If it's defined based on its URL, we must recognize that URL parameters might indicate a completely different page. For example, the page [www.careercup.com/page?pid=microsoft-interview-questions](http://www.careercup.com/page?pid=microsoft-interview-questions) is totally different from the page [www.careercup.com/page?pid=google-interview-questions](http://www.careercup.com/page?pid=google-interview-questions). But, we can also append URL parameters arbitrarily to any URL without truly changing the page, provided it's not a parameter that the web application recognizes and handles. The page [www.careercup.com?foobar=hello](http://www.careercup.com?foobar=hello) is the same as [www.careercup.com](http://www.careercup.com).

"Okay, then," you might say, "let's define it based on its content." That sounds good too, at first, but it also doesn't quite work. Suppose I have some randomly generated content on the [careercup.com](http://careercup.com) home page. Is it a different page each time you visit it? Not really.

The reality is that there is probably no perfect way to define a "different" page, and this is where this problem gets tricky.

One way to tackle this is to have some sort of estimation for degree of similarity. If, based on the content and the URL, a page is deemed to be sufficiently similar to other pages, we *deprioritize* crawling its children. For each page, we would come up with some sort of signature based on snippets of the content and the page's URL.

Let's see how this would work.

We have a database which stores a list of items we need to crawl. On each iteration, we select the highest priority page to crawl. We then do the following:

1. Open up the page and create a signature of the page based on specific subsections of the page and its URL.
2. Query the database to see whether anything with this signature has been crawled recently.
3. If something with this signature has been recently crawled, insert this page back into the database at a low priority.
4. If not, crawl the page and insert its links into the database.

Under the above implementation, we never "complete" crawling the web, but we will avoid getting stuck in a loop of pages. If we want to allow for the possibility of "finishing" crawling the web (which would clearly happen only if the "web" were actually a smaller system, like an intranet), then we can set a minimum priority that a page must have to be crawled.

This is just one, simplistic solution, and there are many others that are equally valid. A problem like this will more likely resemble a conversation with your interviewer which could take any number of paths. In fact, the discussion of this problem could have taken the path of the very next problem.

- 9.4 Duplicate URLs:** You have 10 billion URLs. How do you detect the duplicate documents? In this case, assume "duplicate" means that the URLs are identical.

pg 145

### SOLUTION

Just how much space do 10 billion URLs take up? If each URL is an average of 100 characters, and each character is 4 bytes, then this list of 10 billion URLs will take up about 4 terabytes. We are probably not going to hold that much data in memory.

But, let's just pretend for a moment that we were miraculously holding this data in memory, since it's useful to first construct a solution for the simple version. Under this version of the problem, we would just create a hash table where each URL maps to `true` if it's already been found elsewhere in the list. (As an alternative solution, we could sort the list and look for the duplicate values that way. That will take a bunch of extra time and offers few advantages.)

Now that we have a solution for the simple version, what happens when we have all 4000 gigabytes of data and we can't store it all in memory? We could solve this either by storing some of the data on disk or by splitting up the data across machines.

#### **Solution #1: Disk Storage**

If we stored all the data on one machine, we would do two passes of the document. The first pass would split the list of URLs into 4000 chunks of 1 GB each. An easy way to do that might be to store each URL  $u$  in a file named  $\langle x \rangle .txt$  where  $x = \text{hash}(u) \% 4000$ . That is, we divide up the URLs based on their hash value (modulo the number of chunks). This way, all URLs with the same hash value would be in the same file.

In the second pass, we would essentially implement the simple solution we came up with earlier: load each file into memory, create a hash table of the URLs, and look for duplicates.

#### **Solution #2: Multiple Machines**

The other solution is to perform essentially the same procedure, but to use multiple machines. In this solution, rather than storing the data in file  $\langle x \rangle .txt$ , we would send the URL to machine  $x$ .

Using multiple machines has pros and cons.

The main pro is that we can parallelize the operation, such that all 4000 chunks are processed simultaneously. For large amounts of data, this might result in a faster solution.

The disadvantage though is that we are now relying on 4000 different machines to operate perfectly. That may not be realistic (particularly with more data and more machines), and we'll need to start considering how to handle failure. Additionally, we have increased the complexity of the system simply by involving so many machines.

Both are good solutions, though, and both should be discussed with your interviewer.

- 9.5 Cache:** Imagine a web server for a simplified search engine. This system has 100 machines to respond to search queries, which may then call out using `processSearch(string query)` to another cluster of machines to actually get the result. The machine which responds to a given query is chosen at random, so you cannot guarantee that the same machine will always respond to the same request. The method `processSearch` is very expensive. Design a caching mechanism to cache the results of the most recent queries. Be sure to explain how you would update the cache when data changes.

pg 145

### SOLUTION

Before getting into the design of this system, we first have to understand what the question means. Many of the details are somewhat ambiguous, as is expected in questions like this. We will make reasonable assumptions for the purposes of this solution, but you should discuss these details—in depth—with your interviewer.

#### Assumptions

Here are a few of the assumptions we make for this solution. Depending on the design of your system and how you approach the problem, you may make other assumptions. Remember that while some approaches are better than others, there is no one “correct” approach.

- Other than calling out to `processSearch` as necessary, all query processing happens on the initial machine that was called.
- The number of queries we wish to cache is large (millions).
- Calling between machines is relatively quick.
- The result for a given query is an ordered list of URLs, each of which has an associated 50 character title and 200 character summary.
- The most popular queries are extremely popular, such that they would always appear in the cache.

Again, these aren’t the *only* valid assumptions. This is just one reasonable set of assumptions.

#### System Requirements

When designing the cache, we know we’ll need to support two primary functions:

- Efficient lookups given a key.
- Expiration of old data so that it can be replaced with new data.

In addition, we must also handle updating or clearing the cache when the results for a query change. Because some queries are very common and may permanently reside in the cache, we cannot just wait for the cache to naturally expire.

#### Step 1: Design a Cache for a Single System

A good way to approach this problem is to start by designing it for a single machine. So, how would you create a data structure that enables you to easily purge old data and also efficiently look up a value based on a key?

- A linked list would allow easy purging of old data, by moving “fresh” items to the front. We could implement it to remove the last element of the linked list when the list exceeds a certain size.

- A hash table allows efficient lookups of data, but it wouldn't ordinarily allow easy data purging.

How can we get the best of both worlds? By merging the two data structures. Here's how this works:

Just as before, we create a linked list where a node is moved to the front every time it's accessed. This way, the end of the linked list will always contain the stalest information.

In addition, we have a hash table that maps from a query to the corresponding node in the linked list. This allows us to not only efficiently return the cached results, but also to move the appropriate node to the front of the list, thereby updating its "freshness."

For illustrative purposes, abbreviated code for the cache is below. The code attachment provides the full code for this part. Note that in your interview, it is unlikely that you would be asked to write the full code for this as well as perform the design for the larger system.

```
1  public class Cache {  
2      public static int MAX_SIZE = 10;  
3      public Node head, tail;  
4      public HashMap<String, Node> map;  
5      public int size = 0;  
6  
7      public Cache() {  
8          map = new HashMap<String, Node>();  
9      }  
10  
11     /* Moves node to front of linked list */  
12     public void moveToFront(Node node) { ... }  
13     public void moveToFront(String query) { ... }  
14  
15     /* Removes node from linked list */  
16     public void removeFromLinkedList(Node node) { ... }  
17  
18     /* Gets results from cache, and updates linked list */  
19     public String[] getResults(String query) {  
20         if (!map.containsKey(query)) return null;  
21  
22         Node node = map.get(query);  
23         moveToFront(node); // update freshness  
24         return node.results;  
25     }  
26  
27     /* Inserts results into linked list and hash */  
28     public void insertResults(String query, String[] results) {  
29         if (map.containsKey(query)) { // update values  
30             Node node = map.get(query);  
31             node.results = results;  
32             moveToFront(node); // update freshness  
33             return;  
34         }  
35  
36         Node node = new Node(query, results);  
37         moveToFront(node);  
38         map.put(query, node);  
39  
40         if (size > MAX_SIZE) {  
41             map.remove(tail.query);  
42             removeFromLinkedList(tail);  
43         }  
44     }
```

```
44     }
45 }
```

### Step 2: Expand to Many Machines

Now that we understand how to design this for a single machine, we need to understand how we would design this when queries could be sent to many different machines. Recall from the problem statement that there's no guarantee that a particular query will be consistently sent to the same machine.

The first thing we need to decide is to what extent the cache is shared across machines. We have several options to consider.

#### *Option 1: Each machine has its own cache.*

A simple option is to give each machine its own cache. This means that if "foo" is sent to machine 1 twice in a short amount of time, the result would be recalled from the cache on the second time. But, if "foo" is sent first to machine 1 and then to machine 2, it would be treated as a totally fresh query both times.

This has the advantage of being relatively quick, since no machine-to-machine calls are used. The cache, unfortunately, is somewhat less effective as an optimization tool as many repeat queries would be treated as fresh queries.

#### *Option 2: Each machine has a copy of the cache.*

On the other extreme, we could give each machine a complete copy of the cache. When new items are added to the cache, they are sent to all machines. The entire data structure—linked list and hash table—would be duplicated.

This design means that common queries would nearly always be in the cache, as the cache is the same everywhere. The major drawback however is that updating the cache means firing off data to N different machines, where N is the size of the response cluster. Additionally, because each item effectively takes up N times as much space, our cache would hold much less data.

#### *Option 3: Each machine stores a segment of the cache.*

A third option is to divide up the cache, such that each machine holds a different part of it. Then, when machine i needs to look up the results for a query, machine i would figure out which machine holds this value, and then ask this other machine (machine j) to look up the query in j's cache.

But how would machine i know which machine holds this part of the hash table?

One option is to assign queries based on the formula  $\text{hash(query)} \% N$ . Then, machine i only needs to apply this formula to know that machine j should store the results for this query.

So, when a new query comes in to machine i, this machine would apply the formula and call out to machine j. Machine j would then return the value from its cache or call `processSearch(query)` to get the results. Machine j would update its cache and return the results back to i.

Alternatively, you could design the system such that machine j just returns null if it doesn't have the query in its current cache. This would require machine i to call `processSearch` and then forward the results to machine j for storage. This implementation actually increases the number of machine-to-machine calls, with few advantages.

### Step 3: Updating results when contents change

Recall that some queries may be so popular that, with a sufficiently large cache, they would permanently be cached. We need some sort of mechanism to allow cached results to be refreshed, either periodically or “on-demand” when certain content changes.

To answer this question, we need to consider when results would change (and you need to discuss this with your interviewer). The primary times would be when:

1. The content at a URL changes (or the page at that URL is removed).
2. The ordering of results change in response to the rank of a page changing.
3. New pages appear related to a particular query.

To handle situations #1 and #2, we could create a separate hash table that would tell us which cached queries are tied to a specific URL. This could be handled completely separately from the other caches, and reside on different machines. However, this solution may require a lot of data.

Alternatively, if the data doesn’t require instant refreshing (which it probably doesn’t), we could periodically crawl through the cache stored on each machine to purge queries tied to the updated URLs.

Situation #3 is substantially more difficult to handle. We could update single word queries by parsing the content at the new URL and purging these one-word queries from the caches. But, this will only handle the one-word queries.

A good way to handle Situation #3 (and likely something we’d want to do anyway) is to implement an “automatic time-out” on the cache. That is, we’d impose a time out where no query, regardless of how popular it is, can sit in the cache for more than x minutes. This will ensure that all data is periodically refreshed.

### Step 4: Further Enhancements

There are a number of improvements and tweaks you could make to this design depending on the assumptions you make and the situations you optimize for.

One such optimization is to better support the situation where some queries are very popular. For example, suppose (as an extreme example) a particular string constitutes 1% of all queries. Rather than machine i forwarding the request to machine j every time, machine i could forward the request just once to j, and then i could store the results in its own cache as well.

Alternatively, there may also be some possibility of doing some sort of re-architecture of the system to assign queries to machines based on their hash value (and therefore the location of the cache), rather than randomly. However, this decision may come with its own set of trade-offs.

Another optimization we could make is to the “automatic time out” mechanism. As initially described, this mechanism purges any data after X minutes. However, we may want to update some data (like current news) much more frequently than other data (like historical stock prices). We could implement timeouts based on topic or based on URLs. In the latter situation, each URL would have a time out value based on how frequently the page has been updated in the past. The time out for the query would be the minimum of the time outs for each URL.

These are just a few of the enhancements we can make. Remember that in questions like this, there is no single correct way to solve the problem. These questions are about having a discussion with your interviewer about design criteria and demonstrating your general approach and methodology.

- 9.6 Sales Rank:** A large eCommerce company wishes to list the best-selling products, overall and by category. For example, one product might be the #1056th best-selling product overall but the #13th best-selling product under "Sports Equipment" and the #24th best-selling product under "Safety." Describe how you would design this system.

pg 145

### SOLUTION

Let's first start off by making some assumptions to define the problem.

#### Step 1: Scope the Problem

First, we need to define what exactly we're building.

- We'll assume that we're only being asked to design the components relevant to this question, and not the entire eCommerce system. In this case, we might touch the design of the frontend and purchase components, but only as it impacts the sales rank.
- We should also define what the sales rank means. Is it total sales over all time? Sales in the last month? Last week? Or some more complicated function (such as one involving some sort of exponential decay of sales data)? This would be something to discuss with your interviewer. We will assume that it is simply the total sales over the past week.
- We will assume that each product can be in multiple categories, and that there is no concept of "subcategories."

This part just gives us a good idea of what the problem, or scope of features, is.

#### Step 2: Make Reasonable Assumptions

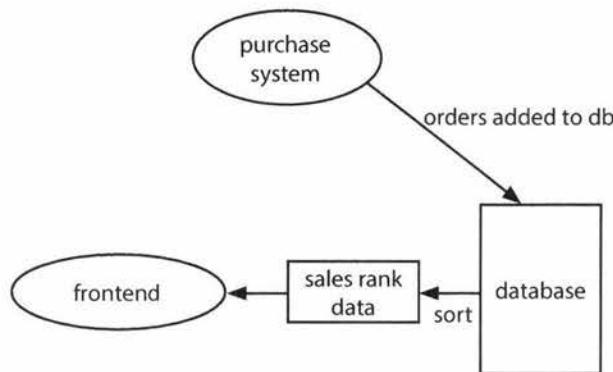
These are the sorts of things you'd want to discuss with your interviewer. Because we don't have an interviewer in front of us, we'll have to make some assumptions.

- We will assume that the stats do not need to be 100% up-to-date. Data can be up to an hour old for the most popular items (for example, top 100 in each category), and up to one day old for the less popular items. That is, few people would care if the #2,809,132th best-selling item should have actually been listed as #2,789,158th instead.
- Precision is important for the most popular items, but a small degree of error is okay for the less popular items.
- We will assume that the data should be updated every hour (for the most popular items), but the time range for this data does not need to be precisely the last seven days (168 hours). If it's sometimes more like 150 hours, that's okay.
- We will assume that the categorizations are based strictly on the origin of the transaction (i.e., the seller's name), not the price or date.

The important thing is not so much which decision you made at each possible issue, but whether it occurred to you that these are assumptions. We should get out as many of these assumptions as possible in the beginning. It's possible you will need to make other assumptions along the way.

#### Step 3: Draw the Major Components

We should now design just a basic, naive system that describes the major components. This is where you would go up to a whiteboard.



In this simple design, we store every order as soon as it comes into the database. Every hour or so, we pull sales data from the database by category, compute the total sales, sort it, and store it in some sort of sales rank data cache (which is probably held in memory). The frontend just pulls the sales rank from this table, rather than hitting the standard database and doing its own analytics.

#### Step 4: Identify the Key Issues

##### *Analytics are Expensive*

In the naive system, we periodically query the database for the number of sales in the past week for each product. This will be fairly expensive. That's running a query over all sales for all time.

Our database just needs to track the total sales. We'll assume (as noted in the beginning of the solution) that the general storage for purchase history is taken care of in other parts of the system, and we just need to focus on the sales data analytics.

Instead of listing every purchase in our database, we'll store just the total sales from the last week. Each purchase will just update the total weekly sales.

Tracking the total sales takes a bit of thought. If we just use a single column to track the total sales over the past week, then we'll need to re-compute the total sales every day (since the specific days covered in the last seven days change with each day). That is unnecessarily expensive.

Instead, we'll just use a table like this.

Prod ID	Total	Sun	Mon	Tues	Wed	Thurs	Fri	Sat

This is essentially like a circular array. Each day, we clear out the corresponding day of the week. On each purchase, we update the total sales count for that product on that day of the week, as well as the total count.

We will also need a separate table to store the associations of product IDs and categories.

Prod ID	Category ID

To get the sales rank per category, we'll need to join these tables.

### *Database Writes are Very Frequent*

Even with this change, we'll still be hitting the database very frequently. With the amount of purchases that could come in every second, we'll probably want to batch up the database writes.

Instead of immediately committing each purchase to the database, we could store purchases in some sort of in-memory cache (as well as to a log file as a backup). Periodically, we'll process the log / cache data, gather the totals, and update the database.

We should quickly think about whether or not it's feasible to hold this in memory. If there are 10 million products in the system, can we store each (along with a count) in a hash table? Yes. If each product ID is four bytes (which is big enough to hold up to 4 billion unique IDs) and each count is four bytes (more than enough), then such a hash table would only take about 40 megabytes. Even with some additional overhead and substantial system growth, we would still be able to fit this all in memory.

After updating the database, we can re-run the sales rank data.

We need to be a bit careful here, though. If we process one product's logs before another's, and re-run the stats in between, we could create a bias in the data (since we're including a larger timespan for one product than its "competing" product).

We can resolve this by either ensuring that the sales rank doesn't run until all the stored data is processed (difficult to do when more and more purchases are coming in), or by dividing up the in-memory cache by some time period. If we update the database for all the stored data up to a particular moment in time, this ensures that the database will not have biases.

### *Joins are Expensive*

We have potentially tens of thousands of product categories. For each category, we'll need to first pull the data for its items (possibly through an expensive join) and then sort those.

Alternatively, we could just do one join of products and categories, such that each product will be listed once per category. Then, if we sorted that on category and then product ID, we could just walk the results to get the sales rank for each category.

Prod ID	Category	Total	Sun	Mon	Tues	Wed	Thurs	Fri	Sat
1423	sportseq	13	4	1	4	19	322	32	232
1423	safety	13	4	1	4	19	322	32	232

Rather than running thousands of queries (one for each category), we could sort the data on the category first and then the sales volume. Then, if we walked those results, we would get the sales rank for each category. We would also need to do one sort of the entire table on just sales number, to get the overall rank.

We could also just keep the data in a table like this from the beginning, rather than doing joins. This would require us to update multiple rows for each product.

### *Database Queries Might Still Be Expensive*

Alternatively, if the queries and writes get very expensive, we could consider forgoing a database entirely and just using log files. This would allow us to take advantage of something like MapReduce.

Under this system, we would write a purchase to a simple text file with the product ID and time stamp. Each category has its own directory, and each purchase gets written to all the categories associated with that product.

We would run frequent jobs to merge files together by product ID and time ranges, so that eventually all purchases in a given day (or possibly hour) were grouped together.

```
/sportsequipment  
1423,Dec 13 08:23-Dec 13 08:23,1  
4221,Dec 13 15:22-Dec 15 15:45,5  
...  
/safety  
1423,Dec 13 08:23-Dec 13 08:23,1  
5221,Dec 12 03:19-Dec 12 03:28,19  
...
```

To get the best-selling products within each category, we just need to sort each directory.

How do we get the overall ranking? There are two good approaches:

- We could treat the general category as just another directory, and write every purchase to that directory. That would mean a lot of files in this directory.
- Or, since we'll already have the products sorted by sales volume order for each category, we can also do an N-way merge to get the overall rank.

Alternatively, we can take advantage of the fact that the data doesn't need (as we assumed earlier) to be 100% up-to-date. We just need the most popular items to be up-to-date.

We can merge the most popular items from each category in a pairwise fashion. So, two categories get paired together and we merge the most popular items (the first 100 or so). After we have 100 items in this sorted order, we stop merging this pair and move onto the next pair.

To get the ranking for all products, we can be much lazier and only run this work once a day.

One of the advantages of this is that it scales nicely. We can easily divide up the files across multiple servers, as they aren't dependent on each other.

### Follow Up Questions

The interviewer could push this design in any number of directions.

- Where do you think you'd hit the next bottlenecks? What would you do about that?
- What if there were subcategories as well? So items could be listed under "Sports" and "Sports Equipment" (or even "Sports" > "Sports Equipment" > "Tennis" > "Rackets")?
- What if data needed to be more accurate? What if it needed to be accurate within 30 minutes for all products?

Think through your design carefully and analyze it for the tradeoffs. You might also be asked to go into more detail on any specific aspect of the product.

- 9.7 Personal Financial Manager:** Explain how you would design a personal financial manager (like Mint.com). This system would connect to your bank accounts, analyze your spending habits, and make recommendations.

pg 145

### SOLUTION

The first thing we need to do is define what it is exactly that we are building.

### Step 1: Scope the Problem

Ordinarily, you would clarify this system with your interviewer. We'll scope the problem as follows:

- You create an account and add your bank accounts. You can add multiple bank accounts. You can also add them at a later point in time.
- It pulls in all your financial history, or as much of it as your bank will allow.
- This financial history includes outgoing money (things you bought or paid for), incoming money (salary and other payments), and your current money (what's in your bank account and investments).
- Each payment transaction has a "category" associated with it (food, travel, clothing, etc.).
- There is some sort of data source provided that tells the system, with some reliability, which category a transaction is associated with. The user might, in some cases, override the category when it's improperly assigned (e.g., eating at the cafe of a department store getting assigned to "clothing" rather than "food").
- Users will use the system to get recommendations on their spending. These recommendations will come from a mix of "typical" users ("people generally shouldn't spend more than X% of their income on clothing"), but can be overridden with custom budgets. This will not be a primary focus right now.
- We assume this is just a website for now, although we could potentially talk about a mobile app as well.
- We probably want email notifications either on a regular basis, or on certain conditions (spending over a certain threshold, hitting a budget max, etc.).
- We'll assume that there's no concept of user-specified rules for assigning categories to transactions.

This gives us a basic goal for what we want to build.

### Step 2: Make Reasonable Assumptions

Now that we have the basic goal for the system, we should define some further assumptions about the characteristics of the system.

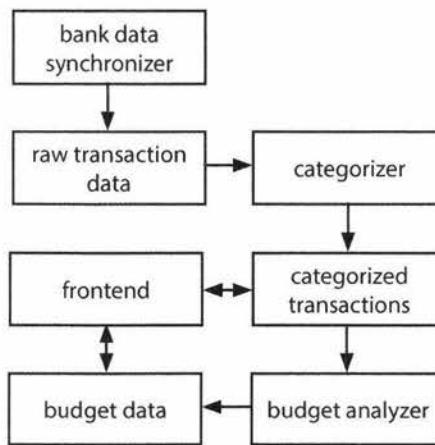
- Adding or removing bank accounts is relatively unusual.
- The system is write-heavy. A typical user may make several new transactions daily, although few users would access the website more than once a week. In fact, for many users, their primary interaction might be through email alerts.
- Once a transaction is assigned to a category, it will only be changed if the user asks to change it. The system will never reassign a transaction to a different category "behind the scenes", even if the rules change. This means that two otherwise identical transactions could be assigned to different categories if the rules changed in between each transaction's date. We do this because it may confuse users if their spending per category changes with no action on their part.
- The banks probably won't push data to our system. Instead, we will need to pull data from the banks.
- Alerts on users exceeding budgets probably do not need to be sent instantaneously. (That wouldn't be realistic anyway, since we won't get the transaction data instantaneously.) It's probably pretty safe for them to be up to 24 hours delayed.

It's okay to make different assumptions here, but you should explicitly state them to your interviewer.

### Step 3: Draw the Major Components

The most naive system would be one that pulls bank data on each login, categorizes all the data, and then analyzes the user's budget. This wouldn't quite fit the requirements, though, as we want email notifications on particular events.

We can do a bit better.



With this basic architecture, the bank data is pulled at periodic times (hourly or daily). The frequency may depend on the behavior of the users. Less active users may have their accounts checked less frequently.

Once new data arrives, it is stored in some list of raw, unprocessed transactions. This data is then pushed to the categorizer, which assigns each transaction to a category and stores these categorized transactions in another datastore.

The budget analyzer pulls in the categorized transactions, updates each user's budget per category, and stores the user's budget.

The frontend pulls data from both the categorized transactions datastore as well as from the budget datastore. Additionally, a user could also interact with the frontend by changing the budget or the categorization of their transactions.

### Step 4: Identify the Key Issues

We should now reflect on what the major issues here might be.

This will be a very data-heavy system. We want it to feel snappy and responsive, though, so we'll want as much processing as possible to be asynchronous.

We will almost certainly want at least one task queue, where we can queue up work that needs to be done. This work will include tasks such as pulling in new bank data, re-analyzing budgets, and categorizing new bank data. It would also include re-trying tasks that failed.

These tasks will likely have some sort of priority associated with them, as some need to be performed more often than others. We want to build a task queue system that can prioritize some task types over others, while still ensuring that all tasks will be performed eventually. That is, we wouldn't want a low priority task to essentially "starve" because there are always higher priority tasks.

One important part of the system that we haven't yet addressed will be the email system. We could use a task to regularly crawl user's data to check if they're exceeding their budget, but that means checking every

single user daily. Instead, we'll want to queue a task whenever a transaction occurs that potentially exceeds a budget. We can store the current budget totals by category to make it easy to understand if a new transaction exceeds the budget.

We should also consider incorporating the knowledge (or assumption) that a system like this will probably have a large number of inactive users—users who signed up once and then haven't touched the system since. We may want to either remove them from the system entirely or deprioritize their accounts. We'll want some system to track their account activity and associate priority with their accounts.

The biggest bottleneck in our system will likely be the massive amount of data that needs to be pulled and analyzed. We should be able to fetch the bank data asynchronously and run these tasks across many servers. We should drill a bit deeper into how the categorizer and budget analyzer work.

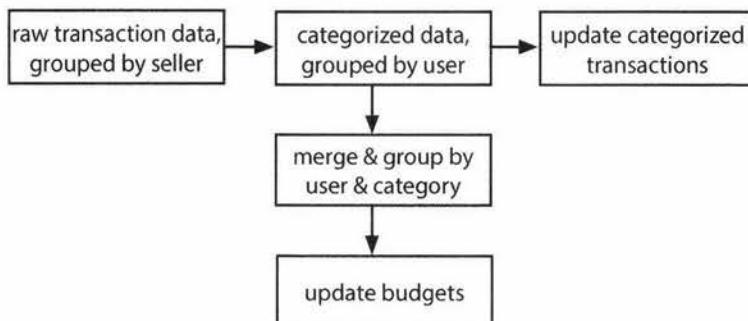
#### *Categorizer and Budget Analyzer*

One thing to note is that transactions are not dependent on each other. As soon as we get a transaction for a user, we can categorize it and integrate this data. It might be inefficient to do so, but it won't cause any inaccuracies.

Should we use a standard database for this? With lots of transactions coming in at once, that might not be very efficient. We certainly don't want to do a bunch of joins.

It may be better instead to just store the transactions to a set of flat text files. We assumed earlier that the categorizations are based on the seller's name alone. If we're assuming a lot of users, then there will be a lot of duplicates across the sellers. If we group the transaction files by seller's name, we can take advantage of these duplicates.

The categorizer can do something like this:



It first gets the raw transaction data, grouped by seller. It picks the appropriate category for the seller (which might be stored in a cache for the most common sellers), and then applies that category to all those transactions.

After applying the category, it re-groups all the transactions by user. Then, those transactions are inserted into the datastore for this user.

before categorizer	after categorizer
amazon/ user121,\$5.43,Aug 13 user922,\$15.39,Aug 27 ... comcast/ user922,\$9.29,Aug 24 user248,\$40.13,Aug 18 ...	user121/ amazon,shopping,\$5.43,Aug 13 ... user922/ amazon,shopping,\$15.39,Aug 27 comcast,utilities,\$9.29,Aug 24 ... user248/ comcast,utilities,\$40.13,Aug 18 ...

Then, the budget analyzer comes in. It takes the data grouped by user, merges it across categories (so all Shopping tasks for this user in this timespan are merged), and then updates the budget.

Most of these tasks will be handled in simple log files. Only the final data (the categorized transactions and the budget analysis) will be stored in a database. This minimizes writing and reading from the database.

#### User Changing Categories

The user might selectively override particular transactions to assign them to a different category. In this case, we would update the datastore for the categorized transactions. It would also signal a quick recomputation of the budget to decrement the item from the old category and increment the item in the other category.

We could also just recompute the budget from scratch. The budget analyzer is fairly quick as it just needs to look over the past few weeks of transactions for a single user.

#### Follow Up Questions

- How would this change if you also needed to support a mobile app?
- How would you design the component which assigns items to each category?
- How would you design the recommended budgets feature?
- How would you change this if the user could develop rules to categorize all transactions from a particular seller differently than the default?

**9.8 Pastebin:** Design a system like Pastebin, where a user can enter a piece of text and get a randomly generated URL for public access.

pg 145

#### SOLUTION

---

We can start with clarifying the specifics of this system.

##### Step 1: Scope the Problem

- The system does not support user accounts or editing documents.
- The system tracks analytics of how many times each page is accessed.
- Old documents get deleted after not being accessed for a sufficiently long period of time.
- While there isn't true authentication on accessing documents, users should not be able to "guess" docu-

ment URLs easily.

- The system has a frontend as well as an API.
- The analytics for each URL can be accessed through a “stats” link on each page. It is not shown by default, though.

### Step 2: Make Reasonable Assumptions

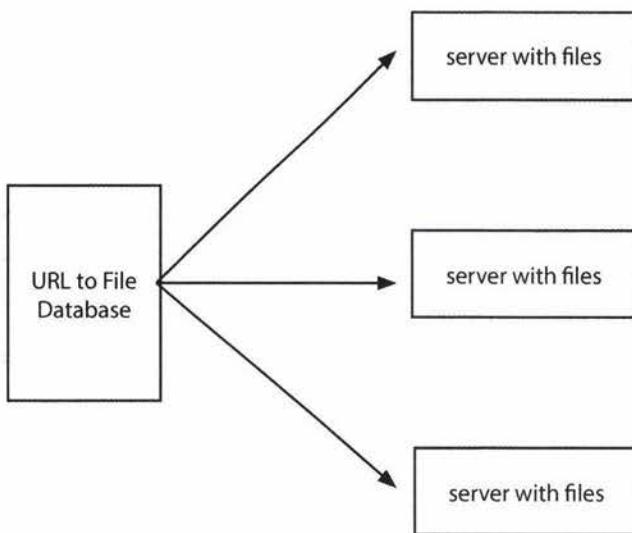
- The system gets heavy traffic and contains many millions of documents.
- Traffic is not equally distributed across documents. Some documents get much more access than others.

### Step 3: Draw the Major Components

We can sketch out a simple design. We'll need to keep track of URLs and the files associated with them, as well as analytics for how often the files have been accessed.

How should we store the documents? We have two options: we can store them in a database or we can store them on a file. Since the documents can be large and it's unlikely we need searching capabilities, storing them on a file is probably the better choice.

A simple design like this might work well:



Here, we have a simple database that looks up the location (server and path) of each file. When we have a request for a URL, we look up the location of the URL within the datastore and then access the file.

Additionally, we will need a database that tracks analytics. We can do this with a simple datastore that adds each visit (including timestamp, IP address, and location) as a row in a database. When we need to access the stats of each visit, we pull the relevant data in from this database.

### Step 4: Identify the Key Issues

The first issue that comes to mind is that some documents will be accessed much more frequently than others. Reading data from the filesystem is relatively slow compared with reading from data in memory. Therefore, we probably want to use a cache to store the most recently accessed documents. This will ensure

that items accessed very frequently (or very recently) will be quickly accessible. Since documents cannot be edited, we will not need to worry about invalidating this cache.

We should also potentially consider sharding the database. We can shard it using some mapping from the URL (for example, the URL's hash code modulo some integer), which will allow us to quickly locate the database which contains this file.

In fact, we could even take this a step further. We could skip the database entirely and just let a hash of the URL indicate which server contains the document. The URL itself could reflect the location of the document. One potential issue from this is that if we need to add servers, it could be difficult to redistribute the documents.

### *Generating URLs*

We have not yet discussed how to actually generate the URLs. We probably do not want a monotonically increasing integer value, as this would be easy for a user to "guess." We want URLs to be difficult to access without being provided the link.

One simple path is to generate a random GUID (e.g., 5d50e8ac-57cb-4a0d-8661-bcdee2548979). This is a 128-bit value that, while not strictly guaranteed to be unique, has low enough odds of a collision that we can treat it as unique. The drawback of this plan is that such a URL is not very "pretty" to the user. We could hash it to a smaller value, but then that increases the odds of collision.

We could do something very similar, though. We could just generate a 10-character sequence of letters and numbers, which gives us  $36^{10}$  possible strings. Even with a billion URLs, the odds of a collision on any specific URL are very low.

This is not to say that the odds of a collision over the whole system are low. They are not. Any one specific URL is unlikely to collide. However, after storing a billion URLs, we are very likely to have a collision at some point.

Assuming that we aren't okay with periodic (even if unusual) data loss, we'll need to handle these collisions. We can either check the datastore to see if the URL exists yet or, if the URL maps to a specific server, just detect whether a file already exists at the destination.

When a collision occurs, we can just generate a new URL. With  $36^{10}$  possible URLs, collisions would be rare enough that the lazy approach here (detect collisions and retry) is sufficient.

### *Analytics*

The final component to discuss is the analytics piece. We probably want to display the number of visits, and possibly break this down by location or time.

We have two options here:

- Store the raw data from each visit.
- Store just the data we know we'll use (number of visits, etc.).

You can discuss this with your interviewer, but it probably makes sense to store the raw data. We never know what features we'll add to the analytics down the road. The raw data allows us flexibility.

This does not mean that the raw data needs to be easily searchable or even accessible. We can just store a log of each visit in a file, and back this up to other servers.

One issue here is that this amount of data could be substantial. We could potentially reduce the space usage considerably by storing data only probabilistically. Each URL would have a `storage_probability` associated with it. As the popularity of a site goes up, the `storage_probability` goes down. For example, a popular document might have data logged only one out of every ten times, at random. When we look up the number of visits for the site, we'll need to adjust the value based on the probability (for example, by multiplying it by 10). This will of course lead to a small inaccuracy, but that may be acceptable.

The log files are not designed to be used frequently. We will want to also store this precomputed data in a datastore. If the analytics just displays the number of visits plus a graph over time, this could be kept in a separate database.

URL	Month and Year	Visits
12ab31b92p	December 2013	242119
12ab31b92p	January 2014	429918
...	...	...

Every time a URL is visited, we can increment the appropriate row and column. This datastore can also be sharded by the URL.

As the stats are not listed on the regular pages and would generally be of less interest, it should not face as heavy of a load. We could still cache the generated HTML on the frontend servers, so that we don't continuously reaccess the data for the most popular URLs.

### Follow-Up Questions

- How would you support user accounts?
- How would you add a new piece of analytics (e.g., referral source) to the stats page?
- How would your design change if the stats were shown with each document?

# 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  }
```

```
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 ints rather than longs) 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.

```

1  long numberOfInts = ((long) Integer.MAX_VALUE) + 1;
2  byte[] bitfield = new byte [(int) (numberOfInts / 8)];
3  String filename = ...
4
5  void findOpenNumber() throws FileNotFoundException {
6      Scanner in = new Scanner(new FileReader(filename));
7      while (in.hasNextInt()) {
8          int n = in.nextInt ();
9          /* Finds the corresponding number in the bitfield by using the OR operator to
10           * set the nth bit of a byte (e.g., 10 would correspond to the 2nd bit of
11           * index 2 in the byte array). */
12          bitfield [n / 8] |= 1 << (n % 8);
13      }
14
15     for (int i = 0; i < bitfield.length; i++) {
16         for (int j = 0; j < 8; j++) {
17             /* Retrieves the individual bits of each byte. When 0 bit is found, print
18             * the corresponding value. */
19             if ((bitfield[i] & (1 << j)) == 0) {
20                 System.out.println (i * 8 + j);
21                 return;
22             }
23         }
24     }
25 }
```

### 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  $\text{arraySize} = \frac{2^{31}}{\text{rangeSize}}$  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:

$$\begin{aligned}\text{arraySize} &= \frac{2^{31}}{\text{rangeSize}} \leq 2^{21} \\ \text{rangeSize} &\geq \frac{2^{31}}{2^{21}} \\ \text{rangeSize} &\geq 2^{10}\end{aligned}$$

#### 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:

$$2^{11} \leq \text{rangeSize} \leq 2^{26}$$

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

```
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  $M \times N$  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  $\text{array}[0][c - 1]$ , 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  $\text{array}[0][2]$ .

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  $\text{array}[0][2]$ , 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:

$$\begin{aligned} a[i][0] &\leq a[i][1] \leq \dots \leq a[i][j-1] \leq a[i][j] \\ a[0][j] &\leq a[1][j] \leq \dots \leq a[i-1][j] \leq a[i][j] \end{aligned}$$

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     /* Set start to start of diagonal and end to the end of the diagonal. Since the  
11        * grid may not be square, the end of the diagonal may not equal dest. */  
12     Coordinate start = (Coordinate) origin.clone();  
13     int diagDist = Math.min(dest.row - origin.row, dest.column - origin.column);  
14     Coordinate end = new Coordinate(start.row + diagDist, start.column + diagDist);  
15     Coordinate p = new Coordinate(0, 0);  
16  
17     /* Do binary search on the diagonal, looking for the first element > x */  
18     while (start.isBefore(end)) {  
19         p.setToAverage(start, end);  
20         if (x > matrix[p.row][p.column]) {  
21             start.row = p.row + 1;  
22             start.column = p.column + 1;  
23         } else {  
24             end.row = p.row - 1;  
25             end.column = p.column - 1;  
26         }  
27     }  
28  
29     /* Split the grid into quadrants. Search the bottom left and the top right. */  
30     return partitionAndSearch(matrix, origin, dest, start, x);  
31 }  
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 }
```

```
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).

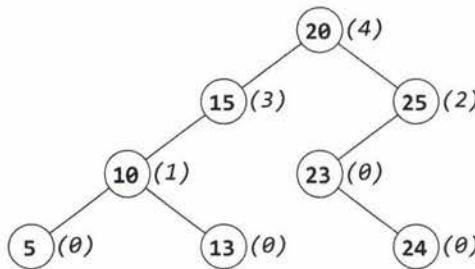
To find the rank of a number, we could do an in-order traversal, keeping a counter as we traverse. The goal is that, by the time we find  $x$ , counter will equal the number of elements less than  $x$ .

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:

```

1 int getRank(Node node, int x) {
2     if x is node.data, return node.leftSize()
3     if x is on left of node, return getRank(node.left, x)
4     if x is on right of node, return node.leftSize() + 1 + getRank(node.right, x)
5 }
```

The full code for this is below.

```

1 RankNode root = null;
2
3 void track(int number) {
4     if (root == null) {
5         root = new RankNode(number);
6     } else {
7         root.insert(number);
8     }
9 }
10
11 int getRankOfNumber(int number) {
12     return root.getRank(number);
13 }
14
15
16 public class RankNode {
17     public int left_size = 0;
18     public RankNode left, right;
19     public int data = 0;
20     public RankNode(int d) {
21         data = d;
22     }
23
24     public void insert(int d) {
25         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.

- Sort the array in ascending order.
- Iterate through the elements, starting from index 1 (not 0) and jumping two elements at a time.
- 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:

```
1 unsigned int i;
2 for (i = 100; i >= 0; --i)
3     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  $N$ th 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     Node * node = new Node;
14     nodeMap[cur] = node; // map current before traversing links
15     node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
16     node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
17     return node;
18 }
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
```

```

14     * ref_count is 0, then free the memory created by the integer and destroy the
15     * object. */
16 }

```

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.

```

1 onSetEquals(SmartPoint<T> ptr1, SmartPoint<T> ptr2) {
2     /* If ptr1 has an existing value, decrement its reference count. Then, copy the
3      * pointers to obj and ref_count over. Finally, since we created a new
4      * reference, we need to increment ref_count. */
5 }

```

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.

```

1 template <class T> class SmartPointer {
2 public:
3     SmartPointer(T * ptr) {
4         ref = ptr;
5         ref_count = (unsigned*)malloc(sizeof(unsigned));
6         *ref_count = 1;
7     }
8
9     SmartPointer(SmartPointer<T> & sptr) {
10        ref = sptr.ref;
11        ref_count = sptr.ref_count;
12        ++(*ref_count);
13    }
14
15    /* Override the equal operator, so that when you set one smart pointer equal to
16     * another the old smart pointer has its reference count decremented and the new
17     * smart pointer has its reference count incremented. */
18    SmartPointer<T> & operator=(SmartPointer<T> & sptr) {
19        if (this == &sptr) return *this;
20
21        /* If already assigned to an object, remove one reference. */
22        if (*ref_count > 0) {
23            remove();
24        }
25
26        ref = sptr.ref;
27        ref_count = sptr.ref_count;
28        ++(*ref_count);
29        return *this;
30    }
31
32    ~SmartPointer() {
33        remove(); // Remove one reference to object.
34    }
35
36    T getValue() {
37        return *ref;
38    }
39
40 protected:
41     void remove() {
42         --(*ref_count);
43         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 needed 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 my2DAalloc 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.

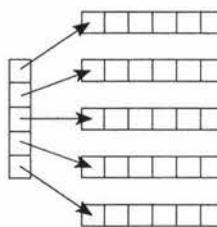
The basic idea is to create a one-dimensional array of pointers. Then, for each array index, we create a new one-dimensional array. This gives us a two-dimensional array that can be accessed via array indices.

The code below implements this.

```

1 int** my2DAalloc(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 }
```

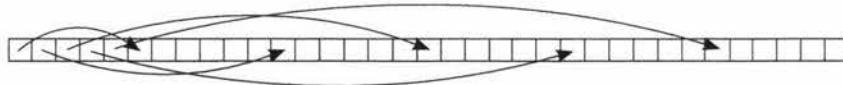
Observe how, in the above code, we've told rowptr where exactly each index should point. The following diagram represents how this memory is allocated.



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

```

```
8  /*** MyClass.cpp ***/
9  template<typename T>
10 int MyClass<T>::bar;
11
12 template class MyClass<Foo>;
13 template class MyClass<Bar>;
14
15 /*** main.cpp ***/
16 MyClass<Foo> * foo1 = new MyClass<Foo>(10);
17 MyClass<Foo> * foo2 = new MyClass<Foo>(15);
18 MyClass<Bar> * bar1 = new MyClass<Bar>(20);
19 MyClass<Bar> * bar2 = new MyClass<Bar>(35);
20
21 int f1 = foo1->val; // will equal 15
22 int f2 = foo2->val; // will equal 15
23 int b1 = bar1->val; // will equal 35
24 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:

- Getting information about the methods and fields present inside the class at runtime.
- Creating a new instance of a class.
- 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

```

```
7  /* Equivalent: Rectangle rectangle = new Rectangle(4.2, 3.9); */
8  Class[] doubleArgsClass = new Class[] {double.class, double.class};
9  Constructor doubleArgsConstructor =
10    rectangleDefinition.getConstructor(doubleArgsClass);
11 Rectangle rectangle = (Rectangle) doubleArgsConstructor.newInstance(doubleArgs);
12
13 /* Equivalent: Double area = rectangle.area(); */
14 Method m = rectangleDefinition.getDeclaredMethod("area");
15 Double area = (Double) m.invoke(rectangle);
```

This code does the equivalent of:

```
1  Rectangle rectangle = new Rectangle(4.2, 3.9);
2  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.

```
1  int getPopulation(List<Country> countries, String continent) {
2      int sum = 0;
3      for (Country c : countries) {
4          if (c.getContinent().equals(continent)) {
5              sum += c.getPopulation();
6          }
7      }
8      return sum;
9  }
```

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.

```
1  Stream<Country> northAmerica = countries.stream().filter(
2      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 JOINS 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

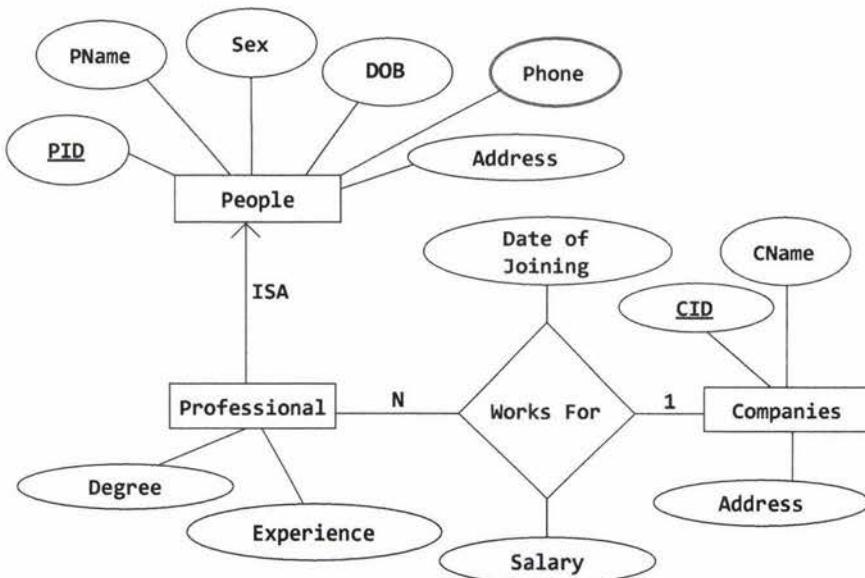
### SOLUTION

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 timestamp 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  $T_c$ , 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          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    public void eat() {
26        pickUp();
27        chew();
28        putDown();
29    }
30
31    public void pickUp() {
32        left.pickUp();
33        right.pickUp();
34    }
35
36    public void chew() { }
37
38    public void putDown() {
39        right.putDown();
40        left.putDown();
41    }
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}
```

```
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:

```

1 boolean checkForCycle(locks[] locks) {
2     touchedNodes = hash table(lock -> boolean)
3     initialize touchedNodes to false for each lock in locks
4     for each (lock x in process.locks) {
5         if (touchedNodes[x] == false) {
6             if (hasCycle(x, touchedNodes)) {
7                 return true;
8             }
9         }
10    }
11    return false;
12 }

13
14 boolean hasCycle(node x, touchedNodes) {
15     touchedNodes[r] = true;
16     if (x.state == VISITING) {
17         return true;
18     } else if (x.state == FRESH) {
19         ... (see full code below)
20     }
21 }
```

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.

```

1 class LockFactory {
2     private static LockFactory instance;
3
4     private int numberofLocks = 5; /* default */
5     private LockNode[] locks;
6
7     /* Maps from a process or owner to the order that the owner claimed it would
8      * call the locks in */
9     private HashMap<Integer, LinkedList<LockNode>> lockOrder;
10
11    private LockFactory(int count) { ... }
12    public static LockFactory getInstance() { return instance; }
13
14    public static synchronized LockFactory initialize(int count) {
15        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  $\text{diff}$  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  $\text{diff}$ , which is the right side of the above number line. Then, when we add  $b$  and  $\text{diff}$  (and store that value in  $b$ ), we get  $a_0$ . We now have  $b = a_0$  and  $a = \text{diff}$ . All that's left to do is to set  $a$  equal to  $a_0 - \text{diff}$ , 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.  $x = x \wedge y$

This line essentially checks if  $x$  and  $y$  have different values. It will result in 1 if and only if  $x \neq y$ .

2.  $y = x \wedge y$

Or:  $y = \{0 \text{ if originally same, 1 if different}\} \wedge \{\text{original } y\}$

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

Therefore, if we do  $y = 1 \wedge \{\text{original } y\}$  when  $x \neq y$ , then  $y$  will be flipped and therefore have  $x$ 's original value.

Otherwise, if  $x == y$ , then we do  $y = 0 \wedge \{\text{original } y\}$  and the value of  $y$  does not change.

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

3.  $x = x \wedge y$

Or:  $x = \{0 \text{ if originally same, 1 if different}\} \wedge \{\text{original } x\}$

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  $x = 1 \wedge \{\text{original } x\}$  when the values are different,  $x$  will be flipped.

If we do  $x = 0 \wedge \{\text{original } x\}$  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.

```

1 Point intersection(Point start1, Point end1, Point start2, Point end2) {
2     /* Rearranging these so that, in order of x values: start is before end and
3      * point 1 is before point 2. This will make some of the later logic simpler. */
4     if (start1.x > end1.x) swap(start1, end1);
5     if (start2.x > end2.x) swap(start2, end2);
6     if (start1.x > start2.x) {
7         swap(start1, start2);
8         swap(end1, end2);
9     }
10
11    /* Compute lines (including slope and y-intercept). */
12    Line line1 = new Line(start1, end1);
13    Line line2 = new Line(start2, end2);
14
15    /* If the lines are parallel, they intercept only if they have the same y
16     * intercept and start 2 is on line 1. */
17    if (line1.slope == line2.slope) {
18        if (line1.yintercept == line2.yintercept &&
19            isBetween(start1, start2, end1)) {
20            return start2;
21        }
22        return null;
23    }
24
25    /* Get intersection coordinate. */
26    double x = (line2.yintercept - line1.yintercept) / (line1.slope - line2.slope);
27    double y = x * line1.slope + line1.yintercept;
28    Point intersection = new Point(x, y);
29
30    /* Check if within line segment range. */
31    if (isBetween(start1, intersection, end1) &&
32        isBetween(start2, intersection, end2)) {
33        return intersection;
34    }
35    return null;
36 }
37
38 /* Checks if middle is between start and end. */
39 boolean isBetween(double start, double middle, double end) {
40     if (start > end) {
41         return end <= middle && middle <= start;
42     } else {
43         return start <= middle && middle <= end;
44     }
45 }
46
47 /* Checks if middle is between start and end. */
48 boolean isBetween(Point start, Point middle, Point end) {
49     return isBetween(start.x, middle.x, end.x) &&
50            isBetween(start.y, middle.y, end.y);
51 }
52
53 /* Swap coordinates of point one and two. */
54 void swap(Point one, Point two) {
55     double x = one.x;
56     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     }
83 }
```

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  $N \times N$  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  $3^0v_0 + 3^1v_1 + 3^2v_2 + \dots + 3^8v_8$ , 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     return Piece.Empty;
20 }
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        }
12    }
```

```

12     }
13
14     /* Check Diagonal */
15     if (hasWinner(board[0][0], board[1][1], board[2][2])) {
16         return board[0][0];
17     }
18
19     if (hasWinner(board[0][2], board[1][1], board[2][0])) {
20         return board[0][2];
21     }
22
23     return Piece.Empty;
24 }
25
26 boolean hasWinner(Piece p1, Piece p2, Piece p3) {
27     if (p1 == Piece.Empty) {
28         return false;
29     }
30     return p1 == p2 && p2 == p3;
31 }
```

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.

```

1  Piece hasWon(Piece[][] board) {
2      int size = board.length;
3      if (board[0].length != size) return Piece.Empty;
4      Piece first;
5
6      /* Check rows. */
7      for (int i = 0; i < size; i++) {
8          first = board[i][0];
9          if (first == Piece.Empty) continue;
10         for (int j = 1; j < size; j++) {
11             if (board[i][j] != first) {
12                 break;
13             } else if (j == size - 1) { // Last element
14                 return first;
15             }
16         }
17     }
18
19     /* Check columns. */
20     for (int i = 0; i < size; i++) {
21         first = board[0][i];
22         if (first == Piece.Empty) continue;
23         for (int j = 1; j < size; j++) {
24             if (board[j][i] != first) {
```

```
25         break;
26     } else if (j == size - 1) { // Last element
27         return first;
28     }
29 }
30 }
31 /* Check diagonals. */
32 first = board[0][0];
33 if (first != Piece.Empty) {
34     for (int i = 1; i < size; i++) {
35         if (board[i][i] != first) {
36             break;
37         } else if (i == size - 1) { // Last element
38             return first;
39         }
40     }
41 }
42 }
43 first = board[0][size - 1];
44 if (first != Piece.Empty) {
45     for (int i = 1; i < size; i++) {
46         if (board[i][size - i - 1] != first) {
47             break;
48         } else if (i == size - 1) { // Last element
49             return first;
50         }
51     }
52 }
53 }
54
55 return Piece.Empty;
56 }
```

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.

```
1 class Check {
2     public int row, column;
3     private int rowIncrement, columnIncrement;
4     public Check(int row, int column, int rowI, int colI) {
5         this.row = row;
6         this.column = column;
7         this.rowIncrement = rowI;
8         this.columnIncrement = colI;
9     }
10
11    public void increment() {
12        row += rowIncrement;
13        column += columnIncrement;
14    }
15
16    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);

```

```
15     if (winner != Piece.Empty) {
16         return winner;
17     }
18 }
19 return Piece.Empty;
20 }
21
22 Piece hasWon(Piece[][] board, PositionIterator iterator) {
23     Position firstPosition = iterator.next();
24     Piece first = board[firstPosition.row][firstPosition.column];
25     while (iterator.hasNext()) {
26         Position position = iterator.next();
27         if (board[position.row][position.column] != first) {
28             return Piece.Empty;
29         }
30     }
31     return first;
32 }
33
34 class PositionIterator implements Iterator<Position> {
35     private int rowIncrement, colIncrement, size;
36     private Position current;
37
38     public PositionIterator(Position p, int rowIncrement,
39                             int colIncrement, int size) {
40         this.rowIncrement = rowIncrement;
41         this.colIncrement = colIncrement;
42         this.size = size;
43         current = new Position(p.row - rowIncrement, p.column - colIncrement);
44     }
45
46     @Override
47     public boolean hasNext() {
48         return current.row + rowIncrement < size &&
49                current.column + colIncrement < size;
50     }
51
52     @Override
53     public Position next() {
54         current = new Position(current.row + rowIncrement,
55                               current.column + colIncrement);
56         return current;
57     }
58 }
59
60 public class Position {
61     public int row, column;
62     public Position(int row, int column) {
63         this.row = row;
64         this.column = column;
65     }
66 }
```

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:

$$19! = 2 * \dots * 5 * \dots * 10 * \dots * 15 * 16 * \dots$$

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 or 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  $\frac{n}{5}$ ), then the number of multiples of 25 ( $\frac{n}{25}$ ), 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:

- 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*.
- 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*.
- 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.
- Now *a* points to 11 and *b* points to 4. Move *b*.
- 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  $a - b \geq 0$ , 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  $\text{INT\_MAX} - 2$  and  $b$  is  $-15$ . In this case,  $a - b$  will be greater than  $\text{INT\_MAX}$  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  $\text{sign}(a)$ .

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 *= 2; // 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  $\frac{a}{2}$ . 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.

- Reducing  $a$  to  $\frac{a}{2}$  takes  $O(\log a)$  time.
  - Reducing  $\frac{a}{2}$  to  $\frac{a}{4}$  takes  $O(\log \frac{a}{2})$  time.
  - Reducing  $\frac{a}{4}$  to  $\frac{a}{8}$  takes  $O(\log \frac{a}{4})$  time.
- ... As so on, for  $O(\log a)$  rounds.

The runtime therefore is  $O(\log a + \log(\frac{a}{2}) + \log(\frac{a}{4}) + \dots)$ , with  $O(\log a)$  terms in the expression.

Recall two rules of logs:

- $\log(xy) = \log x + \log y$
- $\log(\frac{x}{y}) = \log x - \log y$

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

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

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  $X = \frac{a}{b}$ . 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.ArithmaticException {
2     if (b == 0) {
3         throw new java.lang.ArithmaticException("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     return maxAliveYear;
18 }
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  $\text{max} - \text{min} + 1$ .

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	99	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 ( $\leq$ ). 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  $\leq$  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) {

```

```
2  /* Build population delta array. */
3  int[] populationDeltas = getPopulationDeltas(people, min, max);
4  int maxAliveYear = getMaxAliveYear(populationDeltas);
5  return maxAliveYear + min;
6 }
7
8 /* Add birth and death years to deltas array. */
9 int[] getPopulationDeltas(Person[] people, int min, int max) {
10    int[] populationDeltas = new int[max - min + 2];
11    for (Person person : people) {
12        int birth = person.birth - min;
13        populationDeltas[birth]++;
14
15        int death = person.death - min;
16        populationDeltas[death + 1]--;
17    }
18    return populationDeltas;
19 }
20
21 /* Compute running sums and return index with max. */
22 int getMaxAliveYear(int[] deltas) {
23    int maxAliveYear = 0;
24    int maxAlive = 0;
25    int currentlyAlive = 0;
26    for (int year = 0; year < deltas.length; year++) {
27        currentlyAlive += deltas[year];
28        if (currentlyAlive > maxAlive) {
29            maxAliveYear = year;
30            maxAlive = currentlyAlive;
31        }
32    }
33
34    return maxAliveYear;
35 }
```

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  $K \times K$ , 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.