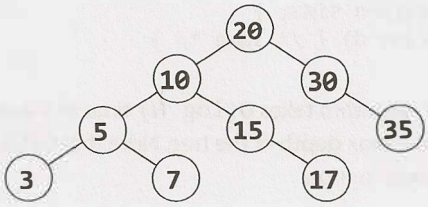
```
1  class TreeNode {
2    private int data;
3    public TreeNode left;
4    public TreeNode right;
5    private int size = 0;
6
7    public TreeNode(int d) {
8      data = d;
9      size = 1;
10   }
11
12   public TreeNode getRandomNode() {
13     int leftSize = left == null ? 0 : left.size();
14     Random random = new Random();
15     int index = random.nextInt(size);
16     if (index < leftSize) {
17       return left.getRandomNode();
18     } else if (index == leftSize) {
19       return this;
20     } else {
21       return right.getRandomNode();
22     }
23   }
24
25   public void insertInOrder(int d) {
26     if (d <= data) {
27       if (left == null) {
28         left = new TreeNode(d);
29       } else {
30         left.insertInOrder(d);
31       }
32     } else {
33       if (right == null) {
34         right = new TreeNode(d);
35       } else {
36         right.insertInOrder(d);
37       }
38     }
39     size++;
40   }
41
42   public int size() { return size; }
43   public int data() { return data; }
44
45   public TreeNode find(int d) {
46     if (d == data) {
47       return this;
48     } else if (d <= data) {
49       return left != null ? left.find(d) : null;
50     } else if (d > data) {
51       return right != null ? right.find(d) : null;
52     }
53     return null;
54   }
55 }
```

In a balanced tree, this algorithm will be $O(\log N)$, where N is the number of nodes.

Option #7 [Fast & Working]

Random number calls can be expensive. If we'd like, we can reduce the number of random number calls substantially.

Imagine we called `getRandomNode` *on the tree below, and then traversed left.*



We traversed left because we picked a number between 0 and 5 (inclusive). When we traverse left, we again pick a random number between 0 and 5. Why re-pick? The first number will work just fine.

But what if we went right instead? We have a number between 7 and 8 (inclusive) but we would need a number between 0 and 1 (inclusive). That's easy to fix: just subtract out `LEFT_SIZE + 1`.

Another way to think about what we're doing is that the initial random number call indicates which node (`i`) to return, and then we're locating the `i`th node in an in-order traversal. Subtracting `LEFT_SIZE + 1` from `i` reflects that, when we go right, we skip over `LEFT_SIZE + 1` nodes in the in-order traversal.

```
1  class Tree {
2    TreeNode root = null;
3
4    public int size() { return root == null ? 0 : root.size(); }
5
6    public TreeNode getRandomNode() {
7      if (root == null) return null;
8
9      Random random = new Random();
10     int i = random.nextInt(size());
11     return root.getIthNode(i);
12   }
13
14   public void insertInOrder(int value) {
15     if (root == null) {
16       root = new TreeNode(value);
17     } else {
18       root.insertInOrder(value);
19     }
20   }
21 }
22
23 class TreeNode {
24   /* constructor and variables are the same. */
25
26   public TreeNode getIthNode(int i) {
27     int leftSize = left == null ? 0 : left.size();
28     if (i < leftSize) {
29       return left.getIthNode(i);
30     } else if (i == leftSize) {
31       return this;
32     } else {
```

```
33          /* Skipping over leftSize + 1 nodes, so subtract them. */
34          return right.getIthNode(i - (leftSize + 1));
35        }
36      }
37
38      public void insertInOrder(int d) { /* same */ }
39      public int size() { return size; }
40      public TreeNode find(int d) { /* same */ }
41  }
```

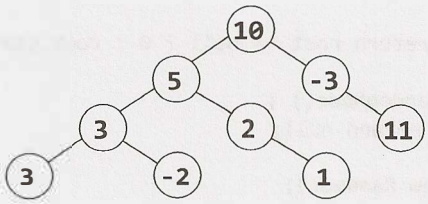
Like the previous algorithm, this algorithm takes O(log N) time in a balanced tree. We can also describe the runtime as O(D), where D is the max depth of the tree. Note that O(D) is an accurate description of the runtime whether the tree is balanced or not.

4.12 Paths with Sum: You are given a binary tree in which each node contains an integer value (which might be positive or negative). Design an algorithm to count the number of paths that sum to a given value. The path does not need to start or end at the root or a leaf, but it must go downwards (traveling only from parent nodes to child nodes).

pg 111

SOLUTION

Let's pick a potential sum—say, 8—and then draw a binary tree based on this. This tree intentionally has a number of paths with this sum.



One option is the brute force approach.

Solution #1: Brute Force

In the brute force approach, we just look at all possible paths. To do this, we traverse to each node. At each node, we recursively try all paths downwards, tracking the sum as we go. As soon as we hit our target sum, we increment the total.

```
1   int countPathsWithSum(TreeNode root, int targetSum) {
2     if (root == null) return 0;
3
4     /* Count paths with sum starting from the root. */
5     int pathsFromRoot = countPathsWithSumFromNode(root, targetSum, 0);
6
7     /* Try the nodes on the left and right. */
8     int pathsOnLeft = countPathsWithSum(root.left, targetSum);
9     int pathsOnRight = countPathsWithSum(root.right, targetSum);
10
11    return pathsFromRoot + pathsOnLeft + pathsOnRight;
12  }
13
14  /* Returns the number of paths with this sum starting from this node. */
```

```
15 int countPathsWithSumFromNode(TreeNode node, int targetSum, int currentSum) {
16   if (node == null) return 0;
17
18   currentSum += node.data;
19
20   int totalPaths = 0;
21   if (currentSum == targetSum) { // Found a path from the root
22     totalPaths++;
23   }
24
25   totalPaths += countPathsWithSumFromNode(node.left, targetSum, currentSum);
26   totalPaths += countPathsWithSumFromNode(node.right, targetSum, currentSum);
27   return totalPaths;
28 }
```

What is the time complexity of this algorithm?

Consider that node at depth d will be "touched" (via `countPathsWithSumFromNode`) by d nodes above it.

In a balanced binary tree, d will be no more than approximately `log N`. Therefore, we know that with N nodes in the tree, `countPathsWithSumFromNode` will be called `O(N log N)` times. The runtime is O(N log N).

We can also approach this from the other direction. At the root node, we traverse to all `N - 1` nodes beneath it (via `countPathsWithSumFromNode`). At the second level (where there are two nodes), we traverse to `N - 3` nodes. At the third level (where there are four nodes, plus three above those), we traverse to `N - 7` nodes. Following this pattern, the total work is roughly:

```
(N - 1) + (N - 3) + (N - 7) + (N - 15) + (N - 31) + ... + (N - N)
```

To simplify this, notice that the left side of each term is always N and the right side is one less than a power of two. The number of terms is the depth of the tree, which is `O(log N)`. For the right side, we can ignore the fact that it's one less than a power of two. Therefore, we really have this:

```
O(N * [number of terms] - [sum of powers of two from 1 through N])
O(N log N - N)
O(N log N)
```

If the value of the sum of powers of two from 1 through N isn't obvious to you, think about what the powers of two look like in binary:

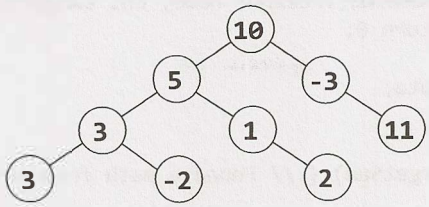
```
  0001
+ 0010
+ 0100
+ 1000
= 1111
```

Therefore, the runtime is `O(N log N)` in a balanced tree.

In an unbalanced tree, the runtime could be much worse. Consider a tree that is just a straight line down. At the root, we traverse to `N - 1` nodes. At the next level (with just a single node), we traverse to `N - 2` nodes. At the third level, we traverse to `N - 3` nodes, and so on. This leads us to the sum of numbers between 1 and N, which is $O(N^2)$.

Solution #2: Optimized

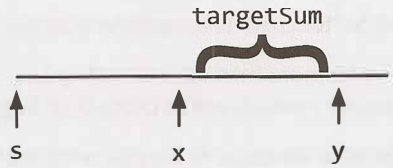
In analyzing the last solution, we may realize that we repeat some work. For a path such as `10 -> 5 -> 3 -> -2`, we traverse this path (or parts of it) repeatedly. We do it when we start with node 10, then when we go to node 5 (looking at 5, then 3, then -2), then when we go to node 3, and then finally when we go to node -2. Ideally, we'd like to reuse this work.



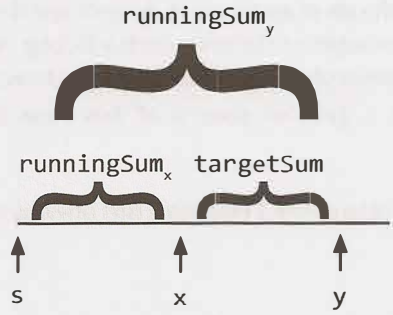
Let's isolate a given path and treat it as just an array. Consider a (hypothetical, extended) path like:

```
10 -> 5 -> 1 -> 2 -> -1 -> -1 -> 7 -> 1 -> 2
```

What we're really saying then is: How many contiguous subsequences in this array sum to a target sum such as 8? In other words, for each `y`, we're trying to find the `x` values below. (Or, more accurately, the number of x values below.)



If each value knows its running sum (the sum of values from `s` through itself), then we can find this pretty easily. We just need to leverage this simple equation: $\texttt{runningSum}_x$ `=` $\texttt{runningSum}_y$ `- targetSum`. We then look for the values of x where this is true.



Since we're just looking for the number of paths, we can use a hash table. As we iterate through the array, build a hash table that maps from a `runningSum` to the number of times we've seen that sum. Then, for each `y`, look up $\texttt{runningSum}_y$ `- targetSum` in the hash table. The value in the hash table will tell you the number of paths with sum `targetSum` that end at `y`.

For example:

```
index:   0    1    2    3    4    5    6    7    8
value:  10 -> 5 -> 1 -> 2 -> -1 -> -1 -> 7 -> 1 -> 2
sum:    10   15   16   18   17   16   23   24   26
```

The value of $\texttt{runningSum}_y$ is 24. If `targetSum` is 8, then we'd look up 16 in the hash table. This would have a value of 2 (originating from index 2 and index 5). As we can see above, indexes 3 through 7 and indexes 6 through 7 have sums of 8.

Now that we've settled the algorithm for an array, let's review this on a tree. We take a similar approach.

We traverse through the tree using depth-first search. As we visit each node:

1. Track its `runningSum`. We'll take this in as a parameter and immediately increment it by `node.value`.
2. Look up `runningSum - targetSum` in the hash table. The value there indicates the total number. Set `totalPaths` to this value.
3. If `runningSum == targetSum`, then there's one additional path that starts at the root. Increment `totalPaths`.
4. Add `runningSum` to the hash table (incrementing the value if it's already there).
5. Recurse left and right, counting the number of paths with sum `targetSum`.
6. After we're done recursing left and right, decrement the value of `runningSum` in the hash table. This is essentially backing out of our work; it reverses the changes to the hash table so that other nodes don't use it (since we're now done with node).

Despite the complexity of deriving this algorithm, the code to implement this is relatively simple.

```
1  int countPathsWithSum(TreeNode root, int targetSum) {
2    return countPathsWithSum(root, targetSum, 0, new HashMap<Integer, Integer>());
3  }
4
5  int countPathsWithSum(TreeNode node, int targetSum, int runningSum,
6                        HashMap<Integer, Integer> pathCount) {
7    if (node == null) return 0; // Base case
8
9    /* Count paths with sum ending at the current node. */
10   runningSum += node.data;
11   int sum = runningSum - targetSum;
12   int totalPaths = pathCount.getOrDefault(sum, 0);
13
14   /* If runningSum equals targetSum, then one additional path starts at root.
15    * Add in this path.*/
16   if (runningSum == targetSum) {
17     totalPaths++;
18   }
19
20   /* Increment pathCount, recurse, then decrement pathCount. */
21   incrementHashTable(pathCount, runningSum, 1); // Increment pathCount
22   totalPaths += countPathsWithSum(node.left, targetSum, runningSum, pathCount);
23   totalPaths += countPathsWithSum(node.right, targetSum, runningSum, pathCount);
24   incrementHashTable(pathCount, runningSum, -1); // Decrement pathCount
25
26   return totalPaths;
27 }
28
29 void incrementHashTable(HashMap<Integer, Integer> hashTable, int key, int delta) {
30   int newCount = hashTable.getOrDefault(key, 0) + delta;
31   if (newCount == 0) { // Remove when zero to reduce space usage
32     hashTable.remove(key);
33   } else {
34     hashTable.put(key, newCount);
35   }
36 }
```

The runtime for this algorithm is $O(N)$, where N is the number of nodes in the tree. We know it is $O(N)$ because we travel to each node just once, doing $O(1)$ work each time. In a balanced tree, the space complexity is $O(\log N)$ due to the hash table. The space complexity can grow to $O(n)$ in an unbalanced tree.

5

Solutions to Bit Manipulation

5.1 **Insertion:** You are given two 32-bit numbers, N and M, and two bit positions, i and j. Write a method to insert M into N such that M starts at bit j and ends at bit i. You can assume that the bits j through i have enough space to fit all of M. That is, if M = 10011, you can assume that there are at least 5 bits between j and i. You would not, for example, have j = 3 and i = 2, because M could not fully fit between bit 3 and bit 2.

EXAMPLE

Input: N = 10000000000, M = 10011, i = 2, j = 6

Output: N = 10001001100

pg 115

SOLUTION

This problem can be approached in three key steps:

1. Clear the bits j through i in N
2. Shift M so that it lines up with bits j through i
3. Merge M and N.

The trickiest part is Step 1. How do we clear the bits in N? We can do this with a mask. This mask will have all 1s, except for 0s in the bits j through i. We create this mask by creating the left half of the mask first, and then the right half.

```
1  int updateBits(int n, int m, int i, int j) {
2    /* Create a mask to clear bits i through j in n. EXAMPLE: i = 2, j = 4. Result
3     * should be 11100011. For simplicity, we'll use just 8 bits for the example. */
4    int allOnes = ~0; // will equal sequence of all 1s
5
6    // 1s before position j, then 0s. left = 11100000
7    int left = allOnes << (j + 1);
8
9    // 1's after position i. right = 00000011
10   int right = ((1 << i) - 1);
11
12   // All 1s, except for 0s between i and j. mask = 11100011
13   int mask = left | right;
14
15   /* Clear bits j through i then put m in there */
16   int n_cleared = n & mask; // Clear bits j through i.
17   int m_shifted = m << i; // Move m into correct position.
```

```
18
19    return n_cleared | m_shifted; // OR them, and we're done!
20 }
```

In a problem like this (and many bit manipulation problems), you should make sure to thoroughly test your code. It's extremely easy to wind up with off-by-one errors.

5.2 **Binary to String:** Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation. If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR."

pg 116

SOLUTION

NOTE: When otherwise ambiguous, we'll use the subscripts x_2 and x_{10} to indicate whether x is in base 2 or base 10.

First, let's start off by asking ourselves what a non-integer number in binary looks like. By analogy to a decimal number, the binary number 0.101_2 would look like:

$$0.101_2 = 1 * {}^1/_{2^1} + 0 * {}^1/_{2^2} + 1 * {}^1/_{2^3}.$$

To print the decimal part, we can multiply by 2 and check if 2n is greater than or equal to 1. This is essentially "shifting" the fractional sum. That is:

$$\begin{aligned} r &= 2_{10} * n \\ &= 2_{10} * 0.101_2 \\ &= 1 * {}^1/_{2^0} + 0 * {}^1/_{2^1} + 1 * {}^1/_{2^2} \\ &= 1.01_2 \end{aligned}$$

If r >= 1, then we know that n had a 1 right after the decimal point. By doing this continuously, we can check every digit.

```
1  String printBinary(double num) {
2    if (num >= 1 || num <= 0) {
3      return "ERROR";
4    }
5
6    StringBuilder binary = new StringBuilder();
7    binary.append(".");
8    while (num > 0) {
9      /* Setting a limit on length: 32 characters */
10     if (binary.length() >= 32) {
11       return "ERROR";
12     }
13
14     double r = num * 2;
15     if (r >= 1) {
16       binary.append(1);
17       num = r - 1;
18     } else {
19       binary.append(0);
20       num = r;
21     }
22   }
23   return binary.toString();
24 }
```

Alternatively, rather than multiplying the number by two and comparing it to 1, we can compare the number to .5, then .25, and so on. The code below demonstrates this approach.

```
String printBinary2(double num) {
   if (num >= 1 || num <= 0) {
      return "ERROR";
   }

   StringBuilder binary = new StringBuilder();
   double frac = 0.5;
   binary.append(".");
   while (num > 0) {
      /* Setting a limit on length: 32 characters */
      if (binary.length() > 32) {
         return "ERROR";
      }
      if (num >= frac) {
         binary.append(1);
         num -= frac;
      } else {
         binary.append(0);
      }
      frac /= 2;
   }
   return binary.toString();
}
```

Both approaches are equally good; choose the one you feel most comfortable with.

Either way, you should make sure to prepare thorough test cases for this problem—and to actually run through them in your interview.

5.3 **Flip Bit to Win:** You have an integer and you can flip exactly one bit from a 0 to a 1. Write code to find the length of the longest sequence of 1s you could create.

EXAMPLE

Input: 1775 (or: 11011101111)

Output: 8

pg 116

SOLUTION

We can think about each integer as being an alternating sequence of 0s and 1s. Whenever a 0s sequence has length one, we can potentially merge the adjacent 1s sequences.

Brute Force

One approach is to convert an integer into an array that reflects the lengths of the 0s and 1s sequences. For example, 11011101111 would be (reading from right to left) $[0_0, 4_1, 1_0, 3_1, 1_0, 2_1, 21_0]$. The subscript reflects whether the integer corresponds to a 0s sequence or a 1s sequence, but the actual solution doesn't need this. It's a strictly alternating sequence, always starting with the 0s sequence.

Once we have this, we just walk through the array. At each 0s sequence, then we consider merging the adjacent 1s sequences if the 0s sequence has length 1.

```
int longestSequence(int n) {
```

```
    if (n == -1) return Integer.BYTES * 8;
    ArrayList<Integer> sequences = getAlternatingSequences(n);
    return findLongestSequence(sequences);
  }

  /* Return a list of the sizes of the sequences. The sequence starts off with the
     number of 0s (which might be 0) and then alternates with the counts of each
     value.*/
  ArrayList<Integer> getAlternatingSequences(int n) {
    ArrayList<Integer> sequences = new ArrayList<Integer>();

    int searchingFor = 0;
    int counter = 0;

    for (int i = 0; i < Integer.BYTES * 8; i++) {
      if ((n & 1) != searchingFor) {
        sequences.add(counter);
        searchingFor = n & 1; // Flip 1 to 0 or 0 to 1
        counter = 0;
      }
      counter++;
      n >>>= 1;
    }
    sequences.add(counter);

    return sequences;
  }

  /* Given the lengths of alternating sequences of 0s and 1s, find the longest one
   * we can build. */
  int findLongestSequence(ArrayList<Integer> seq) {
    int maxSeq = 1;

    for (int i = 0; i < seq.size(); i += 2) {
      int zerosSeq = seq.get(i);
      int onesSeqRight = i - 1 >= 0 ? seq.get(i - 1) : 0;
      int onesSeqLeft = i + 1 < seq.size() ? seq.get(i + 1) : 0;

      int thisSeq = 0;
      if (zerosSeq == 1) { // Can merge
        thisSeq = onesSeqLeft + 1 + onesSeqRight;
      } if (zerosSeq > 1) { // Just add a zero to either side
        thisSeq = 1 + Math.max(onesSeqRight, onesSeqLeft);
      } else if (zerosSeq == 0) { // No zero, but take either side
        thisSeq = Math.max(onesSeqRight, onesSeqLeft);
      }
      maxSeq = Math.max(thisSeq, maxSeq);
    }

    return maxSeq;
  }
```

This is pretty good. It's $O(b)$ time and $O(b)$ memory, where b is the length of the sequence.