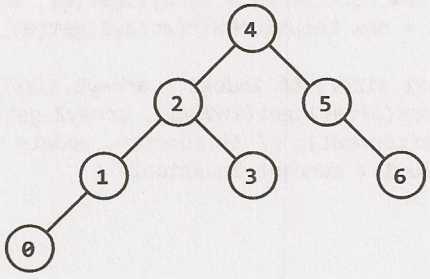
17.12 BiNode: Consider a simple data structure called BiNode, which has pointers to two other nodes. The data structure `BiNode` could be used to represent both a binary tree (where `node1` is the left node and `node2` is the right node) or a doubly linked list (where `node1` is the previous node and `node2` is the next node). Implement a method to convert a binary search tree (implemented with `BiNode`) into a doubly linked list. The values should be kept in order and the operation should be performed in place (that is, on the original data structure).

pg 188

SOLUTION

This seemingly complex problem can be implemented quite elegantly using recursion. You will need to understand recursion very well to solve it.

Picture a simple binary search tree:



The `convert` method should transform it into the below doubly linked list:

```
0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> 6
```

Let's approach this recursively, starting with the root (node 4).

We know that the left and right halves of the tree form their own "sub-parts" of the linked list (that is, they appear consecutively in the linked list). So, if we recursively converted the left and right subtrees to a doubly linked list, could we build the final linked list from those parts?

Yes! We would simply merge the different parts.

The pseudocode looks something like:

```
1  BiNode convert(BiNode node) {
2     BiNode left = convert(node.left);
3     BiNode right = convert(node.right);
4     mergeLists(left, node, right);
5     return left; // front of left
6  }
```

To actually implement the nitty-gritty details of this, we'll need to get the head and tail of each linked list. We can do this several different ways.

Solution #1: Additional Data Structure

The first, and easier, approach is to create a new data structure called `NodePair` which holds just the head and tail of a linked list. The `convert` method can then return something of type `NodePair`.

The code below implements this approach.

```
1  private class NodePair {
```

```
2     BiNode head, tail;
3
4     public NodePair(BiNode head, BiNode tail)  {
5        this.head = head;
6        this.tail = tail;
7     }
8  }
9
10 public NodePair convert(BiNode root) {
11    if (root == null) return null;
12
13    NodePair part1 = convert(root.node1);
14    NodePair part2 = convert(root.node2);
15
16    if (part1 != null) {
17       concat(part1.tail, root);
18    }
19
20    if (part2 != null) {
21       concat(root, part2.head);
22    }
23
24    return new NodePair(part1 == null ? root : part1.head,
25                        part2 == null ? root : part2.tail);
26 }
27
28 public static void concat(BiNode x, BiNode y) {
29    x.node2 = y;
30    y.node1 = x;
31 }
```

The above code still converts the `BiNode` data structure in place. We're just using `NodePair` as a way to return additional data. We could have alternatively used a two-element `BiNode` array to fulfill the same purposes, but it looks a bit messier (and we like clean code, especially in an interview).

It'd be nice, though, if we could do this without these extra data structures—and we can.

Solution #2: Retrieving the Tail

Instead of returning the head and tail of the linked list with `NodePair`, we can return just the head, and then we can use the head to find the tail of the linked list.

```
1  BiNode convert(BiNode root) {
2     if (root == null) return null;
3
4     BiNode part1 = convert(root.node1);
5     BiNode part2 = convert(root.node2);
6
7     if (part1 != null) {
8        concat(getTail(part1), root);
9     }
10
11    if (part2 != null) {
12       concat(root, part2);
13    }
14
15    return part1 == null ? root : part1;
```

```
16 }
17
18 public static BiNode getTail(BiNode node) {
19    if (node == null) return null;
20    while (node.node2 != null) {
21       node = node.node2;
22    }
23    return node;
24 }
```

Other than a call to `getTail`, this code is almost identical to the first solution. It is not, however, very efficient. A leaf node at depth d will be "touched" by the `getTail` method d times (one for each node above it), leading to an $O(N^2)$ overall runtime, where N is the number of nodes in the tree.

Solution #3: Building a Circular Linked List

We can build our third and final approach off of the second one.

This approach requires returning the head and tail of the linked list with `BiNode`. We can do this by returning each list as the head of a *circular* linked list. To get the tail, then, we simply call `head.node1`.

```
1  BiNode convertToCircular(BiNode root) {
2     if (root == null) return null;
3
4     BiNode part1 = convertToCircular(root.node1);
5     BiNode part3 = convertToCircular(root.node2);
6
7     if (part1 == null && part3 == null) {
8        root.node1 = root;
9        root.node2 = root;
10       return root;
11    }
12    BiNode tail3 = (part3 == null) ? null : part3.node1;
13
14    /* join left to root */
15    if (part1 == null) {
16       concat(part3.node1, root);
17    } else {
18       concat(part1.node1, root);
19    }
20
21    / *join right to root */
22    if (part3 == null) {
23       concat(root, part1);
24    } else {
25       concat(root, part3);
26    }
27
28    / *join right to left */
29    if (part1 != null && part3 != null) {
30       concat(tail3, part1);
31    }
32
33    return part1 == null ? root : part1;
34 }
35
36 / *Convert list to a circular linked list, then break the circular connection. */
```

```
37 BiNode convert(BiNode root) {
38    BiNode head = convertToCircular(root);
39    head.node1.node2 = null;
40    head.node1 = null;
41    return head;
42 }
```

Observe that we have moved the main parts of the code into `convertToCircular`. The `convert` method calls this method to get the head of the circular linked list, and then breaks the circular connection.

The approach takes `O(N)` time, since each node is only touched an average of once (or, more accurately, `O(1)` times).

17.13 Re-Space: Oh, no! You have accidentally removed all spaces, punctuation, and capitalization in a lengthy document. A sentence like "`I reset the computer. It still didn't boot!`" became "`iresetthecomputeritstilldidntboot`". You'll deal with the punctuation and capitalization later; right now you need to re-insert the spaces. Most of the words are in a dictionary but a few are not. Given a dictionary (a list of strings) and the document (a string), design an algorithm to unconcatenate the document in a way that minimizes the number of unrecognized characters.

EXAMPLE

Input: `jesslookedjustliketimherbrother`

Output: `jess looked just like tim her brother` (7 unrecognized characters)

pg 188

SOLUTION

Some interviewers like to cut to the chase and give you the specific problems. Others, though, like to give you a lot of unnecessary context, like this problem has. It's useful in such cases to boil down the problem to what it's really all about.

In this case, the problem is really about finding a way to break up a string into separate words such that as few characters as possible are "left out" of the parsing.

Note that we do not attempt to "understand" the string. We could just as well parse "`thisisawesome`" to be "`this is a we some`" as we could "`this is awesome`."

Brute Force

The key to this problem is finding a way to define the solution (that is, parsed string) in terms of its subproblems. One way to do this is recursing through the string.

The very first choice we make is where to insert the first space. After the first character? Second character? Third character?

Let's imagine this in terms of a string like `thisismikesfavoritefood`. What is the first space we insert?

- If we insert a space after `t`, this gives us one invalid character.
- After `th` is two invalid characters.
- After `thi` is three invalid characters.
- At `this` we have a complete word. This is zero invalid characters.
- At `thisi` is five invalid characters.
- ... and so on.

After we choose the first space, we can recursively pick the second space, then the third space, and so on, until we are done with the string.

We take the best (fewest invalid characters) out of all these choices and return.

What should the function return? We need both the number of invalid characters in the recursive path as well as the actual parsing. Therefore, we just return both by using a custom-built `ParseResult` class.

```
1  String bestSplit(HashSet<String> dictionary, String sentence) {
2    ParseResult r = split(dictionary, sentence, 0);
3    return r == null ? null : r.parsed;
4  }
5
6  ParseResult split(HashSet<String> dictionary, String sentence, int start) {
7    if (start >= sentence.length()) {
8      return new ParseResult(0, "");
9    }
10
11   int bestInvalid = Integer.MAX_VALUE;
12   String bestParsing = null;
13   String partial = "";
14   int index = start;
15   while (index < sentence.length()) {
16     char c = sentence.charAt(index);
17     partial += c;
18     int invalid = dictionary.contains(partial) ? 0 : partial.length();
19     if (invalid < bestInvalid) { // Short circuit
20       /* Recurse, putting a space after this character. If this is better than
21        * the current best option, replace the best option. */
22       ParseResult result = split(dictionary, sentence, index + 1);
23       if (invalid + result.invalid < bestInvalid) {
24         bestInvalid = invalid + result.invalid;
25         bestParsing = partial + " " + result.parsed;
26         if (bestInvalid == 0) break; // Short circuit
27       }
28     }
29
30     index++;
31   }
32   return new ParseResult(bestInvalid, bestParsing);
33 }
34
35 public class ParseResult {
36   public int invalid = Integer.MAX_VALUE;
37   public String parsed = " ";
38   public ParseResult(int inv, String p) {
39     invalid = inv;
40     parsed = p;
41   }
42 }
```

We've applied two short circuits here.

- Line 22: If the number of current invalid characters exceeds the best known one, then we know this recursive path will not be ideal. There's no point in even taking it.
- Line 30: If we have a path with zero invalid characters, then we know we can't do better than this. We might as well accept this path.

What's the runtime of this? It's difficult to truly describe in practice as it depends on the (English) language.

One way of looking at it is to imagine a bizarre language where essentially all paths in the recursion are taken. In this case, we are making both choices at each character. If there are n characters, this is an $O(2^n)$ runtime.

Optimized

Commonly, when we have exponential runtimes for a recursive algorithm, we optimize them through memoization (that is, caching results). To do so, we need to find the common subproblems.

Where do recursive paths overlap? That is, where are the common subproblems?

Let's again imagine the string `thisismikesfavoritefood`. Again, imagine that everything is a valid word.

In this case, we attempt to insert the first space after `t` as well as after `th` (and many other choices). Think about what the next choice is.

```
split(thisismikesfavoritefood) ->
       t  + split(hisismikesfavoritefood)
    OR th + split(isismikesfavoritefood)
    OR ...

split(hisismikesfavoritefood) ->
       h + split(isismikesfavoritefood)
    OR ...

...
```

Adding a space after `t` and `h` leads to the same recursive path as inserting a space after `th`. There's no sense in computing `split(isismikesfavoritefood)` twice when it will lead to the same result.

We should instead cache the result. We do this using a hash table which maps from the current substring to the `ParseResult` object.

We don't actually need to make the current substring a key. The `start` index in the string sufficiently represents the substring. After all, if we were to use the substring, we'd really be using `sentence.substring(start, sentence.length)`. This hash table will map from a start index to the best parsing from that index to the end of the string.

And, since the start index is the key, we don't need a true hash table at all. We can just use an array of `ParseResult` objects. This will also serve the purpose of mapping from an index to an object.

The code is essentially identical to the earlier function, but now takes in a memo table (a cache). We look up when we first call the function and set it when we return.

```
1  String bestSplit(HashSet<String> dictionary, String sentence) {
2    ParseResult[] memo = new ParseResult[sentence.length()];
3    ParseResult r = split(dictionary, sentence, 0, memo);
4    return r == null ? null : r.parsed;
5  }
6
7  ParseResult split(HashSet<String> dictionary, String sentence, int start,
8                    ParseResult[] memo) {
9    if (start >= sentence.length()) {
10     return new ParseResult(0, "");
11   } if (memo[start] != null) {
12     return memo[start];
13   }
```

```
14
15    int bestInvalid = Integer.MAX_VALUE;
16    String bestParsing = null;
17    String partial = "";
18    int index = start;
19    while (index < sentence.length()) {
20       char c = sentence.charAt(index);
21       partial += c;
22       int invalid = dictionary.contains(partial) ? 0 : partial.length();
23       if (invalid < bestInvalid) { // Short circuit
24          /* Recurse, putting a space after this character. If this is better than
25           * the current best option, replace the best option. */
26          ParseResult result = split(dictionary, sentence, index + 1, memo);
27          if (invalid + result.invalid < bestInvalid) {
28             bestInvalid = invalid + result.invalid;
29             bestParsing = partial + " " + result.parsed;
30             if (bestInvalid == 0) break; // Short circuit
31          }
32       }
33
34       index++;
35    }
36    memo[start] = new ParseResult(bestInvalid, bestParsing);
37    return memo[start];
38 }
```

Understanding the runtime of this is even trickier than in the prior solution. Again, let's imagine the truly bizarre case, where essentially everything looks like a valid word.

One way we can approach it is to realize that `split(i)` will only be computed once for each value of `i`. What happens when we call `split(i)`, assuming we've already called `split(i+1)` through `split(n - 1)`?

```
split(i) -> calls:
    split(i + 1)
    split(i + 2)
    split(i + 3)
    split(i + 4)
    ...
    split(n - 1)
```

Each of the recursive calls has already been computed, so they just return immediately. Doing `n - i` calls at $O(1)$ time each takes $O(n - i)$ time. This means that `split(i)` takes $O(i)$ time at most.

We can now apply the same logic to `split(i - 1)`, `split(i - 2)`, and so on. If we make 1 call to compute `split(n - 1)`, 2 calls to compute `split(n - 2)`, 3 calls to compute `split(n - 3)`, ..., n calls to compute `split(0)`, how many calls total do we do? This is basically the sum of the numbers from 1 through n, which is $O(n^2)$.

Therefore, the runtime of this function is $O(n^2)$.

17.14 Smallest K: Design an algorithm to find the smallest K numbers in an array.

pg 188

SOLUTION

There are a number of ways to approach this problem. We will go through three of them: sorting, max heap, and selection rank.

Some of these algorithms require modifying the array. This is something you should discuss with your interviewer. Note, though, that even if modifying the original array is not acceptable, you can always clone the array and modify the clone instead. This will not impact the overall big O time of any algorithm.

Solution 1: Sorting

We can sort the elements in ascending order and then take the first million numbers from that.

```
1  int[] smallestK(int[] array, int k) {
2    if (k <= 0 || k > array.length) {
3      throw new IllegalArgumentException();
4    }
5
6    /* Sort array. */
7    Arrays.sort(array);
8
9    /* Copy first k elements. */
10   int[] smallest = new int[k];
11   for (int i = 0; i < k; i++) {
12     smallest[i] = array[i];
13   }
14   return smallest;
15 }
```

The time complexity is O(n log(n)).

Solution 2: Max Heap

We can use a max heap to solve this problem. We first create a max heap (largest element at the top) for the first million numbers.

Then, we traverse through the list. On each element, if it's smaller than the root, we insert it into the heap and delete the largest element (which will be the root).

At the end of the traversal, we will have a heap containing the smallest one million numbers. This algorithm is O(n log(m)), where m is the number of values we are looking for.

```
1  int[] smallestK(int[] array, int k) {
2    if (k <= 0 || k > array.length) {
3      throw new IllegalArgumentException();
4    }
5
6    PriorityQueue<Integer> heap = getKMaxHeap(array, k);
7    return heapToIntArray(heap);
8  }
9
10 /* Create max heap of smallest k elements. */
11 PriorityQueue<Integer> getKMaxHeap(int[] array, int k) {
12   PriorityQueue<Integer> heap =
```

```
13        new PriorityQueue<Integer>(k, new MaxHeapComparator());
14     for (int a : array) {
15        if (heap.size() < k) { // If space remaining
16           heap.add(a);
17        } else if (a < heap.peek()) { // If full and top is small
18           heap.poll(); // remove highest
19           heap.add(a); // insert new element
20        }
21     }
22     return heap;
23  }
24
25  /* Convert heap to int array. */
26  int[] heapToIntArray(PriorityQueue<Integer> heap) {
27     int[] array = new int[heap.size()];
28     while (!heap.isEmpty()) {
29        array[heap.size() - 1] = heap.poll();
30     }
31     return array;
32  }
33
34  class MaxHeapComparator implements Comparator<Integer> {
35      public int compare(Integer x, Integer y) {
36          return y - x;
37      }
38  }
```

Java's uses the `PriorityQueue` class to offer heap-like functionality. By default, it operates as a min heap, with the smallest element on the top. To switch it to the biggest element on the top, we can pass in a different comparator.

Approach 3: Selection Rank Algorithm (if elements are unique)

Selection Rank is a well-known algorithm in computer science to find the `i`th smallest (or largest) element in an array in linear time.

If the elements are unique, you can find the `i`th smallest element in expected `O(n)` time. The basic algorithm operates like this:

1. Pick a random element in the array and use it as a "pivot." Partition elements around the pivot, keeping track of the number of elements on the left side of the partition.
2. If there are exactly `i` elements on the left, then you just return the biggest element on the left.
3. If the left side is bigger than `i`, repeat the algorithm on just the left part of the array.
4. If the left side is smaller than `i`, repeat the algorithm on the right, but look for the element with rank `i - leftSize`.

Once you have found the `i`th smallest element, you know that all elements smaller than this will be to the left of this (since you've partitioned the array accordingly). You can now just return the first `i` elements.

The code below implements this algorithm.

```
1  int[] smallestK(int[] array, int k) {
2     if (k <= 0 || k > array.length) {
3        throw new IllegalArgumentException();
4     }
5
```

```
    int threshold = rank(array, k - 1);
    int[] smallest = new int[k];
    int count = 0;
    for (int a : array) {
      if (a <= threshold) {
        smallest[count] = a;
        count++;
      }
    }
    return smallest;
}

/* Get element with rank. */
int rank(int[] array, int rank) {
    return rank(array, 0, array.length - 1, rank);
}

/* Get element with rank between left and right indices. */
int rank(int[] array, int left, int right, int rank) {
    int pivot = array[randomIntInRange(left, right)];
    int leftEnd = partition(array, left, right, pivot);
    int leftSize = leftEnd - left + 1;
    if (rank == leftSize - 1) {
      return max(array, left, leftEnd);
    } else if (rank < leftSize) {
      return rank(array, left, leftEnd, rank);
    } else {
      return rank(array, leftEnd + 1, right, rank - leftSize);
    }
}

/* Partition array around pivot such that all elements <= pivot come before all
 * elements > pivot. */
int partition(int[] array, int left, int right, int pivot) {
    while (left <= right) {
      if (array[left] > pivot) {
        /* Left is bigger than pivot. Swap it to the right side, where we know it
         * should be. */
        swap(array, left, right);
        right--;
      } else if (array[right] <= pivot) {
        /* Right is smaller than the pivot. Swap it to the left side, where we know
         * it should be. */
        swap(array, left, right);
        left++;
      } else {
        /* Left and right are in correct places. Expand both sides. */
        left++;
        right--;
      }
    }
    return left - 1;
}

/* Get random integer within range, inclusive. */
int randomIntInRange(int min, int max) {
```