```
  private ArrayList<LinkedListNode<K, V>> arr;
  public Hasher(int capacity) {
    /* Create list of linked lists at a particular size. Fill list with null
     * values, as it's the only way to make the array the desired size. */
    arr = new ArrayList<LinkedListNode<K, V>>();
    arr.ensureCapacity(capacity); // Optional optimization
    for (int i = 0; i < capacity; i++) {
      arr.add(null);
    }
  }

  /* Insert key and value into hash table. */
  public void put(K key, V value) {
    LinkedListNode<K, V> node = getNodeForKey(key);
    if (node != null) { // Already there
      node.value = value; // just update the value.
      return;
    }

    node = new LinkedListNode<K, V>(key, value);
    int index = getIndexForKey(key);
    if (arr.get(index) != null) {
      node.next = arr.get(index);
      node.next.prev = node;
    }
    arr.set(index, node);
  }

  /* Remove node for key. */
  public void remove(K key) {
    LinkedListNode<K, V> node = getNodeForKey(key);
    if (node.prev != null) {
      node.prev.next = node.next;
    } else {
      /* Removing head - update. */
      int hashKey = getIndexForKey(key);
      arr.set(hashKey, node.next);
    }

    if (node.next != null) {
      node.next.prev = node.prev;
    }
  }

  /* Get value for key. */
  public V get(K key) {
    LinkedListNode<K, V> node = getNodeForKey(key);
    return node == null ? null : node.value;
  }

  /* Get linked list node associated with a given key. */
  private LinkedListNode<K, V> getNodeForKey(K key) {
    int index = getIndexForKey(key);
    LinkedListNode<K, V> current = arr.get(index);
    while (current != null) {
      if (current.key == key) {
```

```
                return current;
            }
            current = current.next;
        }
        return null;
    }

    /* Really naive function to map a key to an index. */
    public int getIndexForKey(K key) {
        return Math.abs(key.hashCode() % arr.size());
    }
}
```

Alternatively, we could implement a similar data structure (a key->value lookup) with a binary search tree as the underlying data structure. Retrieving an element will no longer be `O(1)` (although, technically, this implementation is not `O(1)` if there are many collisions), but it prevents us from creating an unnecessarily large array to hold items.

8

Solutions to Recursion and Dynamic Programming

8.1 **Triple Step:** A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

pg 134

SOLUTION

Let's think about this with the following question: What is the very last step that is done?

The very last hop the child makes—the one that lands her on the nth step—was either a 3-step hop, a 2-step hop, or a 1-step hop.

How many ways then are there to get up to the nth step? We don't know yet, but we can relate it to some subproblems.

If we thought about all of the paths to the nth step, we could just build them off the paths to the three previous steps. We can get up to the nth step by any of the following:

- Going to the (n-1)st step and hopping 1 step.
- Going to the (n-2)nd step and hopping 2 steps.
- Going to the (n-3)rd step and hopping 3 steps.

Therefore, we just need to add the number of these paths together.

Be very careful here. A lot of people want to multiply them. Multiplying one path with another would signify taking one path and then taking the other. That's not what's happening here.

Brute Force Solution

This is a fairly straightforward algorithm to implement recursively. We just need to follow logic like this:

```
countWays(n-1) + countWays(n-2) + countWays(n-3)
```

The one tricky bit is defining the base case. If we have 0 steps to go (we're currently standing on the step), are there zero paths to that step or one path?

That is, what is `countWays(0)`? Is it 1 or 0?

You could define it either way. There is no "right" answer here.

However, it's a lot easier to define it as 1. If you defined it as 0, then you would need some additional base cases (or else you'd just wind up with a series of 0s getting added).

A simple implementation of this code is below.

```
int countWays(int n) {
  if (n < 0) {
    return 0;
  } else if (n == 0) {
    return 1;
  } else {
    return countWays(n-1) + countWays(n-2) + countWays(n-3);
  }
}
```

Like the Fibonacci problem, the runtime of this algorithm is exponential (roughly $O(3^n)$), since each call branches out to three more calls.

Memoization Solution

The previous solution for `countWays` is called many times for the same values, which is unnecessary. We can fix this through memoization.

Essentially, if we've seen this value of n before, return the cached value. Each time we compute a fresh value, add it to the cache.

Typically we use a `HashMap<Integer, Integer>` for a cache. In this case, the keys will be exactly 1 through n. It's more compact to use an integer array.

```
int countWays(int n) {
  int[] memo = new int[n + 1];
  Arrays.fill(memo, -1);
  return countWays(n, memo);
}

int countWays(int n, int[] memo) {
  if (n < 0) {
    return 0;
  } else if (n == 0) {
    return 1;
  } else if (memo[n] > -1) {
    return memo[n];
  } else {
    memo[n] = countWays(n - 1, memo) + countWays(n - 2, memo) +
              countWays(n - 3, memo);
    return memo[n];
  }
}
```

Regardless of whether or not you use memoization, note that the number of ways will quickly overflow the bounds of an integer. By the time you get to just $n = 37$, the result has already overflowed. Using a `long` will delay, but not completely solve, this issue.

It is great to communicate this issue to your interviewer. He probably won't ask you to work around it (although you could, with a `BigInteger` class), but it's nice to demonstrate that you think about these issues.

8.2 Robot in a Grid: Imagine a robot sitting on the upper left corner of grid with r rows and c columns. The robot can only move in two directions, right and down, but certain cells are "off limits" such that the robot cannot step on them. Design an algorithm to find a path for the robot from the top left to the bottom right.

pg 135

SOLUTION

If we picture this grid, the only way to move to spot (r,c) is by moving to one of the adjacent spots: (r-1,c) or (r,c-1). So, we need to find a path to either (r-1,c) or (r,c-1).

How do we find a path to those spots? To find a path to (r-1,c) or (r,c-1), we need to move to one of its adjacent cells. So, we need to find a path to a spot adjacent to (r-1,c), which are coordinates (r-2,c) and (r-1,c-1), or a spot adjacent to (r,c-1), which are spots (r-1,c-1) and (r,c-2). Observe that we list the point (r-1,c-1) twice; we'll discuss that issue later.

> Tip: A lot of people use the variable names x and y when dealing with two-dimensional arrays. This can actually cause some bugs. People tend to think about x as the first coordinate in the matrix and y as the second coordinate (e.g., matrix[x][y]). But, this isn't really correct. The first coordinate is usually thought of as the row number, which is in fact the y value (it goes vertically!). You should write matrix[y][x]. Or, just make your life easier by using r (row) and c (column) instead.

So then, to find a path from the origin, we just work backwards like this. Starting from the last cell, we try to find a path to each of its adjacent cells. The recursive code below implements this algorithm.

```java
ArrayList<Point> getPath(boolean[][] maze) {
  if (maze == null || maze.length == 0) return null;
  ArrayList<Point> path = new ArrayList<Point>();
  if (getPath(maze, maze.length - 1, maze[0].length - 1, path)) {
    return path;
  }
  return null;
}

boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path) {
  /* If out of bounds or not available, return.*/
  if (col < 0 || row < 0 || !maze[row][col]) {
    return false;
  }

  boolean isAtOrigin = (row == 0) && (col == 0);

  /* If there's a path from the start to here, add my location. */
  if (isAtOrigin || getPath(maze, row, col - 1, path) ||
      getPath(maze, row - 1, col, path)) {
    Point p = new Point(row, col);
    path.add(p);
    return true;
  }

  return false;
}
```

This solution is $O(2^{r+c})$, since each path has r+c steps and there are two choices we can make at each step.

We should look for a faster way.

Often, we can optimize exponential algorithms by finding duplicate work. What work are we repeating?

If we walk through the algorithm, we'll see that we are visiting squares multiple times. In fact, we visit each square many, many times. After all, we have `rc` squares but we're doing $O(2^{r+c})$ work. If we were only visiting each square once, we would probably have an algorithm that was `O(rc)` (unless we were somehow doing a lot of work during each visit).

How does our current algorithm work? To find a path to `(r,c)`, we look for a path to an adjacent coordinate: `(r-1,c)` or `(r,c-1)`. Of course, if one of those squares is off limits, we ignore it. Then, we look at their adjacent coordinates: `(r-2,c)`, `(r-1,c-1)`, `(r-1,c-1)`, and `(r,c-2)`. The spot `(r-1,c-1)` appears twice, which means that we're duplicating effort. Ideally, we should remember that we already visited `(r-1,c-1)` so that we don't waste our time.

This is what the dynamic programming algorithm below does.

```
ArrayList<Point> getPath(boolean[][] maze) {
  if (maze = =null || maze.length == 0) return null;
  ArrayList<Point> path = new ArrayList<Point>();
  HashSet<Point> failedPoints = new HashSet<Point>();
  if (getPath(maze, maze.length - 1, maze[0].length - 1, path, failedPoints)) {
    return path;
  }
  return null;
}

boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path,
                HashSet<Point> failedPoints) {
  /* If out of bounds or not available, return.*/
  if (col < 0 || row < 0 || !maze[row][col]) {
    return false;
  }

  Point p = new Point(row, col);

  /* If we've already visited this cell, return. */
  if (failedPoints.contains(p)) {
    return false;
  }

  boolean isAtOrigin = (row == 0) && (col == 0);

  /* If there's a path from start to my current location, add my location.*/
  if (isAtOrigin || getPath(maze, row, col - 1, path, failedPoints) ||
      getPath(maze, row - 1, col, path, failedPoints)) {
    path.add(p);
    return true;
  }

  failedPoints.add(p); // Cache result
  return false;
}
```

This simple change will make our code run substantially faster. The algorithm will now take `O(XY)` time because we hit each cell just once.

8.3 **Magic Index:** A magic index in an array `A[1...n-1]` is defined to be an index such that `A[i] = i`. Given a sorted array of distinct integers, write a method to find a magic index, if one exists, in array A.

FOLLOW UP

What if the values are not distinct?

pg 135

SOLUTION

Immediately, the brute force solution should jump to mind—and there's no shame in mentioning it. We simply iterate through the array, looking for an element which matches this condition.

```
int magicSlow(int[] array) {
  for (int i = 0; i < array.length; i++) {
    if (array[i] == i) {
      return i;
    }
  }
  return -1;
}
```

Given that the array is sorted, though, it's very likely that we're supposed to use this condition.

We may recognize that this problem sounds a lot like the classic binary search problem. Leveraging the Pattern Matching approach for generating algorithms, how might we apply binary search here?

In binary search, we find an element k by comparing it to the middle element, x, and determining if k would land on the left or the right side of x.

Building off this approach, is there a way that we can look at the middle element to determine where a magic index might be? Let's look at a sample array:

-40	-20	-1	1	2	3	5	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

When we look at the middle element `A[5] = 3`, we know that the magic index must be on the right side, since `A[mid] < mid`.

Why couldn't the magic index be on the left side? Observe that when we move from `i` to `i-1`, the value at this index must decrease by at least 1, if not more (since the array is sorted and all the elements are distinct). So, if the middle element is already too small to be a magic index, then when we move to the left, subtracting k indexes and (at least) k values, all subsequent elements will also be too small.

We continue to apply this recursive algorithm, developing code that looks very much like binary search.

```
int magicFast(int[] array) {
  return magicFast(array, 0, array.length - 1);
}

int magicFast(int[] array, int start, int end) {
  if (end < start) {
    return -1;
  }
  int mid = (start + end) / 2;
  if (array[mid] == mid) {
    return mid;
  } else if (array[mid] > mid){
      return magicFast(array, start, mid - 1);
    } else {
      return magicFast(array, mid + 1, end);
    }
  }
```

Follow Up: What if the elements are not distinct?

If the elements are not distinct, then this algorithm fails. Consider the following array:

-10	-5	2	2	2	3	4	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

When we see that A[mid] < mid, we cannot conclude which side the magic index is on. It could be on the right side, as before. Or, it could be on the left side (as it, in fact, is).

Could it be *anywhere* on the left side? Not exactly. Since A[5] = 3, we know that A[4] couldn't be a magic index. A[4] would need to be 4 to be the magic index, but A[4] must be less than or equal to A[5].

In fact, when we see that A[5] = 3, we'll need to recursively search the right side as before. But, to search the left side, we can skip a bunch of elements and only recursively search elements A[0] through A[3]. A[3] is the first element that could be a magic index.

The general pattern is that we compare midIndex and midValue for equality first. Then, if they are not equal, we recursively search the left and right sides as follows:

- Left side: search indices start through Math.min(midIndex - 1, midValue).
- Right side: search indices Math.max(midIndex + 1, midValue) through end.

The code below implements this algorithm.

```
int magicFast(int[] array) {
  return magicFast(array, 0, array.length - 1);
}

int magicFast(int[] array, int start, int end) {
  if (end < start) return -1;

  int midIndex = (start + end) / 2;
  int midValue = array[midIndex];
  if (midValue == midIndex) {
    return midIndex;
  }

  /* Search left * /
  int leftIndex = Math.min(midIndex - 1, midValue);
  int left = magicFast(array, start, leftIndex);
  if (left >= 0) {
    return left;
  }

  /* Search right * /
  int rightIndex = Math.max(midIndex + 1, midValue);
  int right = magicFast(array, rightIndex, end);

  return right;
}
```

Note that in the above code, if the elements are all distinct, the method operates almost identically to the first solution.

8.4 Power Set: Write a method to return all subsets of a set.

pg 135

SOLUTION

We should first have some reasonable expectations of our time and space complexity.

How many subsets of a set are there? When we generate a subset, each element has the "choice" of either being in there or not. That is, for the first element, there are two choices: it is either in the set, or it is not. For the second, there are two, etc. So, doing `{2 * 2 * ... }` n times gives us 2^n subsets.

Assuming that we're going to be returning a list of subsets, then our best case time is actually the total number of elements across all of those subsets. There are 2^n subsets and each of the n elements will be contained in half of the subsets (which 2^{n-1} subsets). Therefore, the total number of elements across all of those subsets is $n * 2^{n-1}$.

We will not be able to beat $O(n2^n)$ in space or time complexity.

The subsets of $\{a_1, a_2, ..., a_n\}$ are also called the powerset, $P(\{a_1, a_2, ..., a_n\})$, or just $P(n)$.

Solution #1: Recursion

This problem is a good candidate for the Base Case and Build approach. Imagine that we are trying to find all subsets of a set like $S = \{a_1, a_2, ..., a_n\}$. We can start with the Base Case.

Base Case: $n = 0$.

There is just one subset of the empty set: $\{\}$.

Case: $n = 1$.

There are two subsets of the set $\{a_1\}$: $\{\}$, $\{a_1\}$.

Case: $n = 2$.

There are four subsets of the set $\{a_1, a_2\}$: $\{\}$, $\{a_1\}$, $\{a_2\}$, $\{a_1, a_2\}$.

Case: $n = 3$.

Now here's where things get interesting. We want to find a way of generating the solution for $n = 3$ based on the prior solutions.

What is the difference between the solution for $n = 3$ and the solution for $n = 2$? Let's look at this more deeply:

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$
$$P(3) = \{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

The difference between these solutions is that $P(2)$ is missing all the subsets containing a_3.

$$P(3) - P(2) = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

How can we use $P(2)$ to create $P(3)$? We can simply clone the subsets in $P(2)$ and add a_3 to them:

$$P(2) = \{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$$
$$P(2) + a_3 = \{a_3\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}$$

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:

```java
ArrayList<ArrayList<Integer>> getSubsets(ArrayList<Integer> set, int index) {
  ArrayList<ArrayList<Integer>> allsubsets;
  if (set.size() == index) { // Base case - add empty set
    allsubsets = new ArrayList<ArrayList<Integer>>();
    allsubsets.add(new ArrayList<Integer>()); // Empty set
  } else {
    allsubsets = getSubsets(set, index + 1);
    int item = set.get(index);
    ArrayList<ArrayList<Integer>> moresubsets =
      new ArrayList<ArrayList<Integer>>();
    for (ArrayList<Integer> subset : allsubsets) {
      ArrayList<Integer> newsubset = new ArrayList<Integer>();
      newsubset.addAll(subset); //
      newsubset.add(item);
      moresubsets.add(newsubset);
    }
    allsubsets.addAll(moresubsets);
  }
  return allsubsets;
}
```

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!

```java
ArrayList<ArrayList<Integer>> getSubsets2(ArrayList<Integer> set) {
  ArrayList<ArrayList<Integer>> allsubsets = new ArrayList<ArrayList<Integer>>();
  int max = 1 << set.size(); /* Compute 2^n */
  for (int k = 0; k < max; k++) {
    ArrayList<Integer> subset = convertIntToSet(k, set);
    allsubsets.add(subset);
  }
  return allsubsets;
}
```