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) {
```

```
            low = mid + 1;
         } else {
            return mid;
         }
      }
      return -1;
   }
```

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.

```
int search(String[] strings, String str, int first, int last) {
   if (first > last) return -1;
   /* Move mid to the middle */
   int mid = (last + first) / 2;

   /* If mid is empty, find closest non-empty string. * /
   if (strings[mid].isEmpty()) {
      int left = mid - 1;
      int right = mid + 1;
      while (true) {
         if (left < first && right > last) {
            return -1;
         } else if (right <= last && !strings[right].isEmpty()) {
            mid = right;
            break;
         } else if (left >= first && !strings[left].isEmpty()) {
            mid = left;
            break;
         }
         right++;
         left--;
      }
   }
```

```

  /* Check for string, and recurse if necessary */
  if (str.equals(strings[mid])) { // Found it!
    return mid;
  } else if (strings[mid].compareTo(str) < 0) { // Search right
    return search(strings, str, mid + 1, last);
  } else { // Search left
    return search(strings, str, first, mid - 1);
  }
}

int search(String[] strings, String str) {
  if (strings == null || str == null || str == “”) {
    return -1;
  }
  return search(strings, str, 0, strings.length - 1);
}
```

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.

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

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

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

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

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

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

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

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

- Let `rangeSize` be the size of the ranges that each block in the first pass represents.
- Let `arraySize` represent the number of blocks in the first pass. Note that `arraySize` = $2^{31}/\text{rangeSize}$ 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:

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

Second Pass: The Bit Vector

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

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

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

The below code provides one implementation for this algorithm.

```
int findOpenNumber(String filename) throws FileNotFoundException {
  int rangeSize = (1 << 20); // 2^20 bits (2^17 bytes)

  /* Get count of number of values within each block. */
  int[] blocks = getCountPerBlock(filename, rangeSize);

  /* Find a block with a missing value. */
  int blockIndex = findBlockWithMissing(blocks, rangeSize);
  if (blockIndex < 0) return -1;

  /* Create bit vector for items within this range. */
  byte[] bitVector = getBitVectorForRange(filename, blockIndex, rangeSize);

  /* Find a zero in the bit vector */
  int offset = findZero(bitVector);
  if (offset < 0) return -1;

  /* Compute missing value. */
  return blockIndex * rangeSize + offset;
}

/* Get count of items within each range. */
int[] getCountPerBlock(String filename, int rangeSize)
    throws FileNotFoundException {
  int arraySize = Integer.MAX_VALUE / rangeSize + 1;
  int[] blocks = new int[arraySize];

  Scanner in = new Scanner (new FileReader(filename));
  while (in.hasNextInt()) {
      int value = in.nextInt();
      blocks[value / rangeSize]++;
  }
  in.close();
  return blocks;
}

/* Find a block whose count is low. */
int findBlockWithMissing(int[] blocks, int rangeSize) {
  for (int i = 0; i < blocks.length; i++) {
    if (blocks[i] < rangeSize){
      return i;
    }
  }
  return -1;
}

/* Create a bit vector for the values within a specific range. */
byte[] getBitVectorForRange(String filename, int blockIndex, int rangeSize)
    throws FileNotFoundException {
  int startRange = blockIndex * rangeSize;
  int endRange = startRange + rangeSize;
  byte[] bitVector = new byte[rangeSize/Byte.SIZE];

  Scanner in = new Scanner(new FileReader(filename));
  while (in.hasNextInt()) {
    int value = in.nextInt();
    /* If the number is inside the block that's missing numbers, we record it */
    if (startRange <= value && value < endRange) {
      int offset = value - startRange;
      int mask = (1 << (offset % Byte.SIZE));
      bitVector[offset / Byte.SIZE] |= mask;
    }
  }
  in.close();
  return bitVector;
```

```
}

/* Find bit index that is 0 within byte. */
int findZero(byte b) {
  for (int i = 0; i < Byte.SIZE; i++) {
    int mask = 1 << i;
    if ((b & mask) == 0) {
      return i;
    }
  }
  return -1;
}

/* Find a zero within the bit vector and return the index. */
int findZero(byte[] bitVector) {
  for (int i = 0; i < bitVector.length; i++) {
    if (bitVector[i] != ~0) { // If not all 1s
      int bitIndex = findZero(bitVector[i]);
      return i * Byte.SIZE + bitIndex;
    }
  }
  return -1;
}
```

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.

```
void checkDuplicates(int[] array) {
  BitSet bs = new BitSet(32000);
  for (int i = 0; i < array.length; i++) {
    int num = array[i];
    int num0 = num - 1; // bitset starts at 0, numbers start at 1
    if (bs.get(num0)) {
      System.out.println(num);
    } else {
      bs.set(num0);
    }
  }
}

class BitSet {
   int[] bitset;

   public BitSet(int size) {
      bitset = new int[(size >> 5) + 1]; // divide by 32
   }

   boolean get(int pos) {
      int wordNumber = (pos >> 5); // divide by 32
      int bitNumber = (pos & 0x1F); // mod 32
      return (bitset[wordNumber] & (1 << bitNumber)) != 0;
   }

   void set(int pos) {
      int wordNumber = (pos >> 5); // divide by 32
      int bitNumber = (pos & 0x1F); // mod 32
      bitset[wordNumber] |= 1 << bitNumber;
   }
}
```

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

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

pg 151

SOLUTION

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

Solution #1: Naive Solution

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The code below implements this elimination algorithm.

```java
boolean findElement(int[][] matrix, int elem) {
  int row = 0;
  int col = matrix[0].length - 1;
  while (row < matrix.length && col >= 0) {
     if (matrix[row][col] == elem) {
        return true;
     } else if (matrix[row][col] > elem) {
        col--;
     } else {
        row++;
     }
  }
  return false;
}
```

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

Solution #2: Binary Search

Let's again look at a simple example.

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

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

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

Or, in other words:

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

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

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

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

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

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

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

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

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

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