# 17

# Solutions to Hard

**17.1 Add Without Plus:** Write a function that adds two numbers. You should not use + or any arithmetic operators.

*pg 186*

## SOLUTION

Our first instinct in problems like these should be that we're going to have to work with bits. Why? Because when you take away the + sign, what other choice do we have? Plus, that's how computers do it!

Our next thought should be to deeply understand how addition works. We can walk through an addition problem to see if we can understand something new—some pattern—and then see if we can replicate that with code.

So let's do just that—let's walk through an addition problem. We'll work in base 10 so that it's easier to see.

To add 759 + 674, I would usually add `digit[0]` from each number, carry the one, add `digit[1]` from each number, carry the one, and so on. You could take the same approach in binary: add each digit, and carry the one as necessary.

Can we make this a little easier? Yes! Imagine I decided to split apart the "addition" and "carry" steps. That is, I do the following:

1. Add 759 + 674, but "forget" to carry. I then get 323.
2. Add 759 + 674 but only do the carrying, rather than the addition of each digit. I then get 1110.
3. Add the result of the first two operations (recursively, using the same process described in step 1 and 2): 1110 + 323 = 1433.

Now, how would we do this in binary?

1. If I add two binary numbers together, but forget to carry, the `i`th bit in the sum will be `0` only if a and b have the same `i`th bit (both `0` or both `1`). This is essentially an XOR.
2. If I add two numbers together but *only* carry, I will have a 1 in the `i`th bit of the sum only if bits `i - 1` of a and b are both `1`s. This is an AND, shifted.
3. Now, recurse until there's nothing to carry.

The following code implements this algorithm.

```
1  int add(int a, int b) {
2     if (b == 0) return a;
3     int sum = a ^ b; // add without carrying
4     int carry = (a & b) << 1; // carry, but don't add
```

```
5     return add(sum, carry); // recurse with sum + carry
6   }
```

Alternatively, you can implement this iteratively.

```
1   int add(int a, int b) {
2     while (b != 0) {
3       int sum = a ^ b; // add without carrying
4       int carry = (a & b) << 1; // carry, but don't add
5       a = sum;
6       b = carry;
7     }
8     return a;
9   }
```

Problems requiring us to implement core operations like addition and subtraction are relatively common. The key in all of these problems is to dig into how these operations are usually implemented, so that we can re-implement them with the constraints of the given problem.

**17.2 Shuffle:** Write a method to shuffle a deck of cards. It must be a perfect shuffle—in other words, each of the 52! permutations of the deck has to be equally likely. Assume that you are given a random number generator which is perfect.

*pg 186*

## SOLUTION

This is a very well known interview question, and a well known algorithm. If you aren't one of the lucky few to already know this algorithm, read on.

Let's imagine our n-element array. Suppose it looks like this:

```
[1] [2] [3] [4] [5]
```

Using our Base Case and Build approach, we can ask this question: suppose we had a method `shuffle(...)` that worked on n - 1 elements. Could we use this to shuffle n elements?

Sure. In fact, that's quite easy. We would first shuffle the first n - 1 elements. Then, we would take the nth element and randomly swap it with an element in the array. That's it!

Recursively, that algorithm looks like this:

```
1   /* Random number between lower and higher, inclusive */
2   int rand(int lower, int higher) {
3     return lower + (int)(Math.random() * (higher - lower + 1));
4   }
5
6   int[] shuffleArrayRecursively(int[] cards, int i) {
7     if (i == 0) return cards;
8
9     shuffleArrayRecursively(cards, i - 1); // Shuffle earlier part
10    int k = rand(0, i); // Pick random index to swap with
11
12    /* Swap element k and i */
13    int temp = cards[k];
14    cards[k] = cards[i];
15    cards[i] = temp;
16
17    /* Return shuffled array */
18    return cards;
```

```
19 }
```

What would this algorithm look like iteratively? Let's think about it. All it does is moving through the array and, for each element `i`, swapping `array[i]` with a random element between `0` and `i`, inclusive.

This is actually a very clean algorithm to implement iteratively:

```
1  void shuffleArrayIteratively(int[] cards) {
2    for (int i = 0; i < cards.length; i++) {
3      int k = rand(0, i);
4      int temp = cards[k];
5      cards[k] = cards[i];
6      cards[i] = temp;
7    }
8  }
```

The iterative approach is usually how we see this algorithm written.

**17.3 Random Set:** Write a method to randomly generate a set of `m` integers from an array of size n. Each element must have equal probability of being chosen.

*pg 186*

**SOLUTION**

Like the prior problem which was similar, (problem 17.2 on page 531), we can look at this problem recursively using the Base Case and Build approach.

Suppose we have an algorithm that can pull a random set of `m` elements from an array of size `n - 1`. How can we use this algorithm to pull a random set of `m` elements from an array of size n?

We can first pull a random set of size `m` from the first `n - 1` elements. Then, we just need to decide if `array[n]` should be inserted into our subset (which would require pulling out a random element from it). An easy way to do this is to pick a random number k from `0` through n. If `k < m`, then insert `array[n]` into `subset[k]`. This will both "fairly" (i.e., with proportional probability) insert `array[n]` into the subset and "fairly" remove a random element from the subset.

The pseudocode for this recursive algorithm would look like this:

```
1  int[] pickMRecursively(int[] original, int m, int i) {
2    if (i + 1 == m) { // Base case
3      /* return first m elements of original */
4    } else if (i + 1 > m) {
5      int[] subset = pickMRecursively(original, m, i - 1);
6      int k =  random value between 0 and i, inclusive
7      if (k < m) {
8        subset[k] = original[i];
9      }
10     return subset;
11   }
12   return null;
13 }
```

This is even cleaner to write iteratively. In this approach, we initialize an array `subset` to be the first `m` elements in `original`. Then, we iterate through the array, starting at element `m`, inserting `array[i]` into the subset at (random) position k whenever `k < m`.

```
1  int[] pickMIteratively(int[] original, int m) {
2    int[] subset = new int[m];
3
```

```
4     /* Fill in subset array with first part of original array */
5     for (int i = 0; i < m ; i++) {
6        subset[i] = original[i];
7     }
8
9     /* Go through rest of original array. */
10    for (int i = m; i < original.length; i++) {
11       int k = rand(0, i); // Random # between 0 and i, inclusive
12       if (k < m) {
13          subset[k] = original[i];
14       }
15    }
16
17    return subset;
18 }
```

Both solutions are, not surprisingly, very similar to the algorithm to shuffle an array.

**17.4 Missing Number:** An array A contains all the integers from 0 to n, except for one number which is missing. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of A[i]," which takes constant time. Write code to find the missing integer. Can you do it in O(n) time?

*pg 186*

**SOLUTION**

You may have seen a very similar sounding problem: Given a list of numbers from 0 to n, with exactly one number removed, find the missing number. This problem can be solved by simply adding the list of numbers and comparing it to the actual sum of 0 through n, which is $^{n(n+1)}/_{2}$. The difference will be the missing number.

We could solve this by computing the value of each number, based on its binary representation, and calculating the sum.

The runtime of this solution is `n * length(n)`, when `length` is the number of bits in n. Note that `length(n)` = $\log_2(n)$. So, the runtime is actually `O(n log(n))`. Not quite good enough!

So how else can we approach it?

We can actually use a similar approach, but leverage the bit values more directly.

Picture a list of binary numbers (the `-----` indicates the value that was removed):

```
00000       00100       01000       01100
00001       00101       01001       01101
00010       00110       01010
-----       00111       01011
```

Removing the number above creates an imbalance of 1s and 0s in the least significant bit, which we'll call $LSB_1$. In a list of numbers from 0 to n, we would expect there to be the same number of 0s as 1s (if n is odd), or an additional 0 if n is even. That is:

```
if n % 2 == 1 then count(0s) = count(1s)
if n % 2 == 0 then count(0s) = 1 + count(1s)
```

Note that this means that `count(0s)` is always greater than or equal to `count(1s)`.

When we remove a value v from the list, we'll know immediately if v is even or odd just by looking at the least significant bits of all the other values in the list.

| | `n % 2 == 0`<br>`count(0s) = 1 + count(1s)` | `n % 2 == 1`<br>`count(0s) = count(1s)` |
|---|---|---|
| `v % 2 == 0`<br>$LSB_1(v) = 0$ | a 0 is removed.<br>`count(0s) = count(1s)` | a 0 is removed.<br>`count(0s) < count(1s)` |
| `v % 2 == 1`<br>$LSB_1(v) = 1$ | a 1 is removed.<br>`count(0s) > count(1s)` | a 1 is removed.<br>`count(0s) > count(1s)` |

So, if `count(0s) <= count(1s)`, then v is even. If `count(0s) > count(1s)`, then v is odd.

We can now remove all the evens and focus on the odds, or remove all the odds and focus on the evens.

Okay, but how do we figure out what the next bit in v is? If v were contained in our (now smaller) list, then we should expect to find the following (where $count_2$ indicates the number of 0s or 1s in the second least significant bit):

$count_2(0s) = count_2(1s)$ OR $count_2(0s) = 1 + count_2(1s)$

As in the earlier example, we can deduce the value of the second least significant bit ($LSB_2$) of v.

| | $count_2(0s) = 1 + count_2(1s)$ | $count_2(0s) = count_2(1s)$ |
|---|---|---|
| $LSB_2(v) == 0$ | a 0 is removed.<br>$count_2(0s) = count_2(1s)$ | a 0 is removed.<br>$count_2(0s) < count_2(1s)$ |
| $LSB_2(v) == 1$ | a 1 is removed.<br>$count_2(0s) > count_2(1s)$ | a 1 is removed.<br>$count_2(0s) > count_2(1s)$ |

Again, we have the same conclusion:

- If $count_2(0s) <= count_2(1s)$, then $LSB_2(v) = 0$.
- If $count_2(0s) > count_2(1s)$, then $LSB_2(v) = 1$.

We can repeat this process for each bit. On each iteration, we count the number of 0s and 1s in bit `i` to check if $LSB_i(v)$ is 0 or 1. Then, we discard the numbers where $LSB_i(x) != LSB_i(v)$. That is, if v is even, we discard the odd numbers, and so on.

By the end of this process, we will have computed all bits in v. In each successive iteration, we look at n, then `n / 2`, then `n / 4`, and so on, bits. This results in a runtime of `O(N)`.

If it helps, we can also move through this more visually. In the first iteration, we start with all the numbers:

```
00000       00100       01000       01100
00001       00101       01001       01101
00010       00110       01010
-----       00111       01011
```

Since $count_1(0s) > count_1(1s)$, we know that $LSB_1(v) = 1$. Now, discard all numbers x where $LSB_1(x) != LSB_1(v)$.

```
~~00000~~   ~~00100~~   ~~01000~~   ~~01100~~
00001       00101       01001       01101
~~00010~~   ~~00110~~   ~~01010~~
-----       00111       01011
```

Now, $count_2(0s) > count_2(1s)$, so we know that $LSB_2(v) = 1$. Now, discard all numbers x where $LSB_2(x) != LSB_2(v)$.

| | | | |
|---|---|---|---|
| ~~00000~~ | ~~00100~~ | ~~01000~~ | ~~01100~~ |
| ~~00001~~ | ~~00101~~ | ~~01001~~ | ~~01101~~ |
| ~~00010~~ | ~~00110~~ | 01010 | |
| ----- | 00111 | 01011 | |

This time, $count_3(0s) <= count_3(1s)$, we know that $LSB_3(v) = 0$. Now, discard all numbers x where $LSB_3(x) != LSB_3(v)$.

| | | | |
|---|---|---|---|
| ~~00000~~ | ~~00100~~ | ~~01000~~ | ~~01100~~ |
| ~~00001~~ | ~~00101~~ | ~~01001~~ | ~~01101~~ |
| ~~00010~~ | ~~00110~~ | ~~01010~~ | |
| ----- | ~~00111~~ | 01011 | |

We're down to just one number. In this case, $count_4(0s) <= count_4(1s)$, so $LSB_4(v) = 0$.

When we discard all numbers where $LSB_4(x) != 0$, we'll wind up with an empty list. Once the list is empty, then $count_1(0s) <= count_1(1s)$, so $LSB_1(v) = 0$. In other words, once we have an empty list, we can fill in the rest of the bits of v with 0.

This process will compute that, for the example above, $v = 00011$.

The code below implements this algorithm. We've implemented the discarding aspect by partitioning the array by bit value as we go.

```
1  int findMissing(ArrayList<BitInteger> array) {
2    /* Start from the least significant bit, and work our way up */
3    return findMissing(array, 0);
4  }
5
6  int findMissing(ArrayList<BitInteger> input, int column) {
7    if (column >= BitInteger.INTEGER_SIZE) { // We're done!
8      return 0;
9    }
10   ArrayList<BitInteger> oneBits = new ArrayList<BitInteger>(input.size()/2);
11   ArrayList<BitInteger> zeroBits = new ArrayList<BitInteger>(input.size()/2);
12
13   for (BitInteger t : input) {
14     if (t.fetch(column) == 0) {
15       zeroBits.add(t);
16     } else {
17       oneBits.add(t);
18     }
19   }
20   if (zeroBits.size() <= oneBits.size()) {
21     int v = findMissing(zeroBits, column + 1);
22     return (v << 1) | 0;
23   } else {
24     int v = findMissing(oneBits, column + 1);
25     return (v << 1) | 1;
26   }
27 }
```

In lines 24 and 27, we recursively calculate the other bits of v. Then, we insert either a 0 or 1, depending on whether or not $count_1(0s) <= count_1(1s)$.

**17.5 Letters and Numbers:** Given an array filled with letters and numbers, find the longest subarray with an equal number of letters and numbers.

pg 186

## SOLUTION

In the introduction, we discussed the importance of creating a really good, general-purpose example. That's absolutely true. It's also important, though, to understand what matters.

In this case, we just want an equal number of letters and numbers. All letters are treated identically and all numbers are treated identically. Therefore, we can use an example with a single letter and a single number—or, for that matter, As and Bs, 0s and 1s, or `Thing1`s and `Thing2`s.

With that said, let's start with an example:

```
[A, B, A, A, A, B, B, B, A, B, A, A, B, B, A, A, A, A, A, A]
```

We're looking for the smallest subarray where `count(A, subarray) = count(B, subarray)`.

### Brute Force

Let's start with the obvious solution. Just go through all subarrays, count the number of As and Bs (or letters and numbers), and find the longest one that is equal.

We can make one small optimization to this. We can start with the longest subarray and, as soon as we find one which fits this equality condition, return it.

```
1  /* Return the largest subarray with equal number of 0s and 1s. Look at each
2   * subarray, starting from the longest. As soon as we find one that's equal, we
3   * return.
4  char[] findLongestSubarray(char[] array) {
5    for (int len = array.length; len > 1; len--) {
6      for (int i = 0; i <= array.length - len; i++) {
7        if (hasEqualLettersNumbers(array, i, i + len - 1)) {
8          return extractSubarray(array, i, i + len - 1);
9        }
10     }
11   }
12   return null;
13 }
14
15 /* Check if subarray has equal number of letters and numbers. */
16 boolean hasEqualLettersNumbers(char[] array, int start, int end) {
17   int counter = 0;
18   for (int i = start; i <= end; i++) {
19     if (Character.isLetter(array[i])) {
20       counter++;
21     } else if (Character.isDigit(array[i])) {
22       counter--;
23     }
24   }
25   return counter == 0;
26 }
27
28 /* Return subarray of array between start and end (inclusive). */
29 char[] extractSubarray(char[] array, int start, int end) {
30   char[] subarray = new char[end - start + 1];
31   for (int i = start; i <= end; i++) {
32     subarray[i - start] = array[i];
```

```
33    }
34    return subarray;
35  }
```

Despite the one optimization we made, this algorithm is still $O(N^2)$, where N is the length of the array.

### Optimal Solution

What we're trying to do is find a subarray where the count of letters equals the count of numbers. What if we just started from the beginning, counting the number of letters and numbers?

| | a | a | a | a | 1 | 1 | a | 1 | 1 | a | a | 1 | a | a | 1 | a | a | a | a | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #a | 1 | 2 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 7 | 7 | 8 | 9 | 9 | 10 | 11 | 12 | 13 | 14 |
| #1 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 |

Certainly, whenever the number of letters equals the number of numbers, we can say that from index 0 to that index is an "equal" subarray.

That will only tell us equal subarrays that start at index 0. How can we identify all equal subarrays?

Let's picture this. Suppose we inserted an equal subarray (like a11a1a) after an array like a1aaa1. How would that impact the counts?

| | a | 1 | a | a | a | 1 | a | 1 | 1 | a | 1 | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #a | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 7 |
| #1 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 4 | 4 | 5 | 5 |

Study the numbers before the subarray (4, 2) and the end (7, 5). You might notice that, while the values aren't the same, the differences are: 4 - 2 = 7 - 5. This makes sense. Since they've added the same number of letters and numbers, they should maintain the same difference.

> Observe that when the difference is the same, the subarray starts one after the initial matching index and continues through the final matching index. This explains line 10 in the code below.

Let's update the earlier array with the differences.

| | a | a | a | a | 1 | 1 | a | 1 | 1 | a | a | 1 | a | a | 1 | a | a | a | a | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| #a | 1 | 2 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 7 | 7 | 8 | 9 | 9 | 10 | 11 | 12 | 13 | 14 |
| #1 | 0 | 0 | 0 | 0 | 1 | 2 | 2 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 |
| - | 1 | 2 | 3 | 4 | 3 | 2 | 3 | 2 | 1 | 2 | 3 | 2 | 3 | 4 | 3 | 4 | 5 | 6 | 7 | 8 |

Whenever we return the same difference, then we know we have found an equal subarray. To find the biggest subarray, we just have to find the two indices farthest apart with the same value.

To do so, we use a hash table to store the first time we see a particular difference. Then, each time we see the same difference, we see if this subarray (from first occurrence of this index to current index) is bigger than the current max. If so, we update the max.

```
1   char[] findLongestSubarray(char[] array) {
2     /* Compute deltas between count of numbers and count of letters. */
3     int[] deltas = computeDeltaArray(array);
4
5     /* Find pair in deltas with matching values and largest span. */
6     int[] match = findLongestMatch(deltas);
7
8     /* Return the subarray. Note that it starts one *after* the initial occurence of
9      * this delta. */
10    return extract(array, match[0] + 1, match[1]);
11  }
12
```

```
/* Compute the difference between the number of letters and numbers between the
 * beginning of the array and each index. */
int[] computeDeltaArray(char[] array) {
  int[] deltas = new int[array.length];
  int delta = 0;
  for (int i = 0; i < array.length; i++) {
    if (Character.isLetter(array[i])) {
      delta++;
    } else if (Character.isDigit(array[i])) {
      delta--;
    }
    deltas[i] = delta;
  }
  return deltas;
}

/* Find the matching pair of values in the deltas array with the largest
 * difference in indices. */
int[] findLongestMatch(int[] deltas) {
  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
  map.put(0,  -1);
  int[] max = new int[2];
  for (int i = 0; i < deltas.length; i++) {
    if (!map.containsKey(deltas[i])) {
      map.put(deltas[i],  i);
    } else {
      int match = map.get(deltas[i]);
      int distance = i - match;
      int longest = max[1] - max[0];
      if (distance > longest) {
        max[1] = i;
        max[0] = match;
      }
    }
  }
  return max;
}

char[] extract(char[] array, int start, int end) { /* same */ }
```

This solution takes O(N) time, where N is size of the array.

**17.6 Count of 2s:** Write a method to count the number of 2s between 0 and n.

*pg 186*

## SOLUTION

Our first approach to this problem can be—and probably should be—a brute force solution. Remember that interviewers want to see how you're approaching a problem. Offering a brute force solution is a great way to start.

```
/* Counts the number of '2' digits between 0 and n */
int numberOf2sInRange(int n) {
  int count = 0;
  for (int i = 2; i <= n; i++) { // Might as well start at 2
    count += numberOf2s(i);
```

```
    }
    return count;
}

/* Counts the number of '2' digits in a single number */
int numberOf2s(int n) {
    int count = 0;
    while (n > 0) {
        if (n % 10 == 2) {
            count++;
        }
        n = n / 10;
    }
    return count;
}
```

The only interesting part is that it's probably cleaner to separate out `numberOf2s` into a separate method. This demonstrates an eye for code cleanliness.

### Improved Solution

Rather than looking at the problem by ranges of numbers, we can look at the problem digit by digit. Picture a sequence of numbers:

```
  0   1   2   3   4   5   6   7   8   9
 10  11  12  13  14  15  16  17  18  19
 20  21  22  23  24  25  26  27  28  29
...
110 111 112 113 114 115 116 117 118 119
```

We know that roughly one tenth of the time, the last digit will be a 2 since it happens once in any sequence of ten numbers. In fact, any digit is a 2 roughly one tenth of the time.

We say "roughly" because there are (very common) boundary conditions. For example, between 1 and 100, the 10's digit is a 2 exactly $1/10^{th}$ of the time. However, between 1 and 37, the 10's digit is a 2 much more than $1/10^{th}$ of the time.

We can work out what exactly the ratio is by looking at the three cases individually: `digit < 2`, `digit = 2`, and `digit > 2`.

*Case digit < 2*

Consider the value `x = 61523` and `d = 3`, and observe that `x[d] = 1` (that is, the dth digit of x is 1). There are 2s at the 3rd digit in the ranges `2000 - 2999`, `12000 - 12999`, `22000 - 22999`, `32000 - 32999`, `42000 - 42999`, and `52000 - 52999`. We will not yet have hit the range `62000 - 62999`, so there are 6000 2s total in the 3rd digit. This is the same amount as if we were just counting all the 2s in the 3rd digit between 1 and 60000.

In other words, we can round *down* to the nearest $10^{d+1}$, and then divide by 10, to compute the number of 2s in the dth digit.

```
if x[d] < 2: count2sInRangeAtDigit(x, d) =
    let y = round down to nearest 10^(d+1)
    return y / 10
```