371. Sum of Two Integers

June 9, 2020 | 3.3K views

Calculate the sum of two integers a and b, but you are **not allowed** to use the operator `+` and `-`.

Example 1:

```
Input: a = 1, b = 2
Output: 3
```

Example 2:

```
Input: a = -2, b = 3
Output: 1
```

Solution

Overview

Approach 1 is a detailed explanation of bit manipulation basics. Approach 2 is a language-specific discussion of a possible follow-up and mainly written for fun.

Approach 1: Bit Manipulation: Easy and Language-Independent

That's an extremely popular Facebook problem designed to check your knowledge of bitwise operators:

$$x \oplus y \quad \text{that means} \quad \text{bitwise XOR}$$
$$x \& y \quad \text{that means} \quad \text{bitwise AND}$$
$$\sim x \quad \text{that means} \quad \text{bitwise NOT}$$

Reduce the Number of Use Cases

First of all, there are too many use cases here: both a and b could be positive or negative, `abs(a)` could be greater or less than `abs(b)`. In total, that results in $2 \times 2 \times 2 = 8$ use cases.

Let's start by reducing the problem down to two simple cases:

- Sum of two positive integers: $x + y$, where $x > y$.
- Difference of two positive integers: $x - y$, where $x > y$.

Java | Python

```python
class Solution:
    def getSum(self, a: int, b: int) -> int:
        x, y = abs(a), abs(b)
        # ensure that abs(a) >= abs(b)
        if x < y:
            return self.getSum(b, a)

        # abs(a) >= abs(b) -->
        # a determines the sign
        sign = 1 if a > 0 else -1

        if a * b >= 0:
            # sum of two positive integers x + y
            # where x > y

            # TODO
        else:
            # difference of two integers x - y
            # where x > y

            # TODO

        return x * sign
```

Interview Tip for Bit Manipulation Problems: Use XOR

How to start? There is an interview tip for bit manipulation problems: if you don't know how to start, start from computing XOR for your input data. Strangely, that helps out for quite a lot of problems, Single Number II, Single Number III, Maximum XOR of Two Numbers in an Array, Repeated DNA Sequences, Maximum Product of Word Lengths, etc.

> What is XOR?

XOR of zero and a bit results in that bit.

$0 \oplus x = x$

XOR of two equal bits (even if they are zeros) results in a zero.

$x \oplus x = 0$

Sum of Two Positive Integers

Now let's use this tip for the first use case: the sum of two positive integers. Here XOR is a key as well because it's a sum of two integers in the binary form without taking carry into account. In other words, XOR is a sum of bits of x and y where at least one of the bits is not set.

x = 15	0	0	0	0	1	1	1	1
y = 2	0	0	0	0	0	0	1	0

x^y = answer without carry	0	0	0	0	1	1	0	1

The next step is to find the carry. It contains the common set bits of x and y, shifted one bit to the left. *I.e.* it's logical AND of two input numbers, shifted one bit to the left: $\text{carry} = (x \& y) << 1$.

x = 15	0	0	0	0	1	1	1	1
y = 2	0	0	0	0	0	0	1	0

(x & y) << 1 = carry	0	0	0	0	0	1	0	0

The problem is reduced down to find the sum of the answer without carry and the carry.

Technically, it's the same problem: to sum two numbers, and hence one could solve it in a loop with the condition statement "while carry is not equal to zero".

x = 15	0	0	0	0	1	1	1	1
y = 2	0	0	0	0	0	0	1	0

x = x^y = answer without carry = 13	0	0	0	0	1	1	0	1
y = (x & y) << 1 = carry = 4	0	0	0	0	0	1	0	0

x = x^y = answer without carry = 9	0	0	0	0	1	0	0	1
y = (x & y) << 1 = carry = 8	0	0	0	0	1	0	0	0

x = x^y = answer without carry = 1	0	0	0	0	0	0	0	1
y = (x & y) << 1 = carry = 16	0	0	0	1	0	0	0	0

x = x^y = answer without carry = 17	0	0	0	1	0	0	0	1
y = (x & y) << 1 = carry = 0	0	0	0	0	0	0	0	0

Difference of Two Positive Integers

As for addition, XOR is a difference of two integers without taking borrow into account.

x = 15	0	0	0	0	1	1	1	1
y = 2	0	0	0	0	0	0	1	0

x^y = answer x - y without borrow	0	0	0	0	1	1	0	1

The next step is to find the borrow. It contains common set bits of y and unset bits of x, *i.e.* $\text{borrow} = ((\sim x)\&y) << 1$.

x = 15	0	0	0	0	1	1	1	1
~x = -16	0	0	0	1	0	0	0	0
y = 2	0	0	0	0	0	0	1	0

((~x) & y) << 1 = borrow = 0	0	0	0	0	0	0	0	0

The problem is reduced down to the subtraction of the borrow from the answer without borrow. As for the sum, it could be solved recursively or in a loop with the condition statement "while borrow is not equal to zero".

x - y

x = 15	0	0	0	0	1	1	1	1
~x	0	0	0	1	0	0	0	0
y = 2	0	0	0	0	0	0	1	0

x = x^y = answer without borrow = 13	0	0	0	0	1	1	0	1
y = ((~x) & y) << 1 = borrow = 0	0	0	0	0	0	0	0	0

Algorithm

- Simplify problem down to two cases: sum or subtraction of two positive integers: $x \pm y$, where $x > y$. Save down the sign of the result.
- If one has to compute the sum:
  - While carry is nonzero: `y != 0`:
    - Current answer without carry is XOR of x and y: `answer = x^y`.
    - Current carry is left-shifted AND of x and y: `carry = (x & y) << 1`.
    - Job is done, prepare the next loop: `x = answer`, `y = carry`.
  - Return `x * sign`.
- If one has to compute the difference:
  - While borrow is nonzero: `y != 0`:
    - Current answer without borrow is XOR of x and y: `answer = x^y`.
    - Current borrow is left-shifted AND of NOT x and y: `borrow = ((~x) & y) << 1`.
    - Job is done, prepare the next loop: `x = answer`, `y = borrow`.
  - Return `x * sign`.

Implementation

Java | Python3

```python
class Solution:
    def getSum(self, a: int, b: int) -> int:
        x, y = abs(a), abs(b)
        # ensure that abs(a) >= abs(b)
        if x < y:
            return self.getSum(b, a)

        # abs(a) >= abs(b) -->
        # a determines the sign
        sign = 1 if a > 0 else -1

        if a * b >= 0:
            # sum of two positive integers x + y
            # where x > y
            while y:
                answer = x ^ y
                carry = (x & y) << 1
                x, y = answer, carry
        else:
            # difference of two integers x - y
            # where x > y
            while y:
                answer = x ^ y
                borrow = ((~x) & y) << 1
                x, y = answer, borrow

        return x * sign
```

This solution could be written a bit shorter in Python:

Python3

```python
class Solution:
    def getSum(self, a: int, b: int) -> int:
        x, y = abs(a), abs(b)
        # ensure x >= y
        if x < y:
            return self.getSum(b, a)
        sign = 1 if a > 0 else -1

        if a * b >= 0:
            # sum of two positive integers
            while y:
                x, y = x ^ y, (x & y) << 1
        else:
            # difference of two positive integers
            while y:
                x, y = x ^ y, ((~x) & y) << 1

        return x * sign
```

Complexity Analysis

- Time complexity: $\mathcal{O}(1)$ because each integer contains 32 bits.
- Space complexity: $\mathcal{O}(1)$ because we don't use any additional data structures.

Approach 2: Bit Manipulation: Short Language-Specific Solution

Approach 1 is easy to attack during the follow-up:

> Please don't use multiplication to manage negative numbers and make a clean bitwise solution.

Let's be honest, it's a trap. Once you start to manage negative numbers using bit manipulation, your solution becomes *language-specific*.

Different languages represent negative numbers differently.

Java

For example, Java integer is a number of 32 bits. 31 bits are used for the value. The first bit is used for the sign: if it's equal to 1, the number is negative, if it's equal to 0, the number is positive.

And now the fun starts. Does it mean that

$1 = (\underbrace{0}_{\text{positive}} \underbrace{00000..01}_{30 \text{ times}})_2$

and

$-1 = (\underbrace{1}_{\text{negative}} \underbrace{00000..01}_{30 \text{ times}})_2?$

No!

For the representation of a negative number Java uses the so-called "two's complement":

$-1 = (\underbrace{1}_{\text{negative}} \underbrace{11111..11}_{30 \text{ times}})_2$

The idea is simple:

$$(-1 + 1)\&(\underbrace{111111..1}_{32 \text{ 1-bits}})_2 = 0$$

$$(-x + x)\&(\underbrace{111111..1}_{32 \text{ 1-bits}})_2 = 0$$

The main goal of "two's complement" is to decrease the complexity of bit manipulations. How does Java compute "two's complement" and manage 32-bits limit? Here is how:

- After each operation we have an invisible `& mask`, where `mask = 0xFFFFFFFF`, *i.e.* bitmask of 32 1-bits.
- The overflow, *i.e.* the situation of `x > 0x7FFFFFFF` (bitmask of 31 1-bits), is managed as `x --> ~(x ^ 0xFFFFFFFF)`.

At this point, we could come back to approach 1 and, surprisingly, all management of negative numbers, signs, and subtractions Java already does for us. That simplifies the solution to the computation of a sum of two positive integers. That's how the magic of "two's complement" works!

Java

```java
class Solution {
    public int getSum(int a, int b) {
        while (b != 0) {
            int answer = a ^ b;
            int carry = (a & b) << 1;
            a = answer;
            b = carry;
        }

        return a;
    }
}
```

Python

Now let's go back to real life. Python has no 32-bits limit, and hence its representation of negative integers is entirely different.

There is no Java magic by default, and if you need a magic - just do it:

- After each operation we have an invisible `& mask`, where `mask = 0xFFFFFFFF`, *i.e.* bitmask of 32 1-bits.
- The overflow, *i.e.* the situation of `x > 0x7FFFFFFF` (bitmask of 31 1-bits), is managed as `x --> ~(x ^ 0xFFFFFFFF)`.

Python3

```python
class Solution:
    def getSum(self, a: int, b: int) -> int:
        mask = 0xFFFFFFFF

        while b != 0:
            a, b = (a ^ b) & mask, ((a & b) << 1) & mask

        max_int = 0x7FFFFFFF
        return a if a < max_int else ~(a ^ mask)
```

Implementation

Each language has its beauty.

C++ | Java | Go | Python3

```python
class Solution:
    def getSum(self, a: int, b: int) -> int:
        mask = 0xFFFFFFFF

        while b != 0:
            a, b = (a ^ b) & mask, ((a & b) << 1) & mask

        max_int = 0x7FFFFFFF
        return a if a < max_int else ~(a ^ mask)
```

Complexity Analysis

- Time complexity: $\mathcal{O}(1)$.
- Space complexity: $\mathcal{O}(1)$.