Simulate Calculation

For the problem which are math related, I am not going to discuss the pure math theory, but focus on computer useful math theory. In this chapter we simulate some math calculation without using any math library function.

## 29. Divide Two Integers

Medium

Given two integers dividend and divisor, divide two integers without using multiplication, division and mod operator.

Return the quotient after dividing dividend by divisor.

The integer division should truncate toward zero.

**Example 1:**

**Input:** dividend = 10, divisor = 3

**Output:** 3

**Example 2:**

**Input:** dividend = 7, divisor = -3

**Output:** -2

**Note:**

* Both dividend and divisor will be 32-bit signed integers.
* The divisor will never be 0.
* Assume we are dealing with an environment which could only store integers within the 32-bit signed integer range: [−231,  231 − 1]. For the purpose of this problem, assume that your function returns 231 − 1 when the division result overflows.

### Analysis:

You cannot division, but you can use bit wise deduct, first align the divisor at least same or larger than divisor, and then keep on shifting the divisor right, if the dividend is same or more than divisor, the bit is 1 and you deduct the divisor from dividend, otherwise the bit is 0.

/// <summary>

/// Leet code #29. Divide Two Integers

///

/// Given two integers dividend and divisor, divide two integers without

/// using multiplication, division and mod operator.

///

/// Return the quotient after dividing dividend by divisor.

///

/// The integer division should truncate toward zero.

///

/// Example 1:

///

/// Input: dividend = 10, divisor = 3

/// Output: 3

///

/// Example 2:

///

/// Input: dividend = 7, divisor = -3

/// Output: -2

/// Note:

/// Both dividend and divisor will be 32-bit signed integers.

/// The divisor will never be 0.

/// Assume we are dealing with an environment which could only store

/// integers within the 32-bit signed integer range: [−2^31, 2^31 − 1].

/// For the purpose of this problem, assume that your function

/// returns 2^31 − 1 when the division result overflows.

/// </summary>

int LeetCodeMath::divide(int dividend, int divisor)

{

if ((divisor == 0) || (dividend == INT\_MIN && divisor == -1))

{

return INT\_MAX;

}

int sign = (dividend < 0) ^ (divisor < 0) ? -1 : 1;

long long long\_dividend = abs((long long)dividend);

long long long\_divisor = abs((long long)divisor);

long long sum = 0;

long long bit = 1;

while ((long\_dividend >> 1) >= long\_divisor)

{

bit <<= 1;

long\_divisor <<= 1;

}

while (bit != 0)

{

if (long\_dividend >= long\_divisor)

{

sum = sum | bit;

long\_dividend -= long\_divisor;

}

long\_divisor >>= 1;

bit >>= 1;

}

if (sign < 0)

{

return (int)(0 - sum);

}

else

{

return (int)sum;

}

}

## 50. Pow(x, n)

Medium

Implement [pow(x, n)](http://www.cplusplus.com/reference/valarray/pow/), which calculates x raised to the power n (xn).

**Example 1:**

**Input:** 2.00000, 10

**Output:** 1024.00000

**Example 2:**

**Input:** 2.10000, 3

**Output:** 9.26100

**Example 3:**

**Input:** 2.00000, -2

**Output:** 0.25000

**Explanation:** 2-2 = 1/22 = 1/4 = 0.25

**Note:**

* -100.0 < x < 100.0
* n is a 32-bit signed integer, within the range [−231, 231− 1]

### Analysis:

The exponent is integer, so the math operation is simplified as multiplication or division. However the number is too big, so you need to do it efficiently, the idea is that if the a^2n = a^n \* a^n; so if the exponent is even, we divide by 2 if odd, we deduct one, and until the n is 1, then we reverse the sequence generated above is the top of stack is same as current sum of exponent, we do a self-multiply, if it is 1 we simply multiply x.

Just watch the overflow.

/// <summary>

/// Leet code #50. Pow(x, n)

///

/// Implement pow(x, n), which calculates x raised to the power n (x^n).

///

/// Example 1:

///

/// Input: 2.00000, 10

/// Output: 1024.00000

///

/// Example 2:

///

/// Input: 2.10000, 3

/// Output: 9.26100

///

/// Example 3:

///

/// Input: 2.00000, -2

/// Output: 0.25000

/// Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25

///

/// Note:

///

/// 1. -100.0 < x < 100.0

/// 2. n is a 32-bit signed integer, within the range [−2^31, 2^31 − 1]

/// </summary>

double LeetCodeMath::myPow(double x, int n)

{

stack<int> stack;

if (n == 0)

{

return 1;

}

while (n != 0)

{

if (n < 0)

{

stack.push(-1);

if (n == INT\_MIN)

{

stack.push(1);

n = INT\_MAX;

}

else

{

n = 0 - n;

}

}

else if (n % 2 == 1)

{

stack.push(1);

n = n - 1;

}

else

{

stack.push(n / 2);

n = n / 2;

}

}

double result = 1;

int sum = 0;

while (!stack.empty())

{

int exp = stack.top();

stack.pop();

if (exp == 1)

{

result = result \* x;

sum += 1;

}

else if (exp == sum)

{

result = result \* result;

sum += exp;

}

else if (exp == -1)

{

result = 1 / result;

sum = 0 - sum;

}

}

return result;

}

## 43. Multiply Strings

Medium

Given two non-negative integers num1 and num2 represented as strings, return the product of num1 and num2, also represented as a string.

**Example 1:**

**Input:** num1 = "2", num2 = "3"

**Output:** "6"

**Example 2:**

**Input:** num1 = "123", num2 = "456"

**Output:** "56088"

**Note:**

1. The length of both num1 and num2 is < 110.
2. Both num1 and num2 contain only digits 0-9.
3. Both num1 and num2 do not contain any leading zero, except the number 0 itself.
4. You **must not use any built-in BigInteger library** or **convert the inputs to integer** directly.

### Analysis:

If the ith digit in num1 multiplies the jth digit in num2, you will impact the i+j th digit and i+j+1 th digit in product.

/// <summary>

/// Leet code #43. Multiply Strings

///

/// Given two non-negative integers num1 and num2 represented as strings,

/// return the product of num1 and num2, also represented as a string.

///

/// Example 1:

/// Input: num1 = "2", num2 = "3"

/// Output: "6"

///

/// Example 2:

///

/// Input: num1 = "123", num2 = "456"

/// Output: "56088"

///

/// Note:

///

/// 1. The length of both num1 and num2 is < 110.

/// 2. Both num1 and num2 contain only digits 0-9.

/// 3. Both num1 and num2 do not contain any leading zero, except the number

/// 0 itself.

/// 4. You must not use any built-in BigInteger library or convert the inputs

/// to integer directly.

/// </summary>

string LeetCodeMath::multiply(string num1, string num2)

{

string result;

vector<int> product\_digits(num1.size() + num2.size(), 0);

for (int i = num1.size() - 1; i >= 0; i--)

{

for (int j = num2.size() - 1; j >= 0; j--)

{

int digit\_pos = num1.size() - 1 - i + num2.size() - 1 - j;

int product = (num1[i] - '0') \* (num2[j] - '0');

while (product != 0)

{

product\_digits[digit\_pos] += product;

product = product\_digits[digit\_pos] / 10;

product\_digits[digit\_pos] %= 10;

digit\_pos++;

}

}

}

for (int i = product\_digits.size() - 1; i >= 0; i--)

{

if (!result.empty() || product\_digits[i] != 0)

{

result.push\_back(product\_digits[i] + '0');

}

}

if (result.empty()) result = "0";

return result;

}

## 372. Super Pow

Medium

Your task is to calculate *ab* mod 1337 where *a* is a positive integer and *b* is an extremely large positive integer given in the form of an array.

**Example 1:**

**Input:** a = 2, b = [3]

**Output:** 8

**Example 2:**

**Input:** a = 2, b = [1,0]

**Output:** 1024

### Analysis:

To calculate x^(100a + 10b +c), you will calculate x^100a, x^10b and x^c separately, to calculate x^10b, you will do by x^10^b, and to calculate x^100a, you will do x^10^10^a. This means you need to accumulate x^10.

/// <summary>

/// Leet code #372. Super Pow

///

/// Your task is to calculate a^b mod 1337 where a is a positive integer

/// and b is an extremely large

/// positive integer given in the form of an array.

///

/// Example1:

/// a = 2

/// b = [3]

/// Result: 8

///

/// Example2:

/// a = 2

/// b = [1,0]

/// Result: 1024

/// </summary>

int LeetCodeMath::superPow(int a, vector<int>& b)

{

unsigned long long product = 1;

unsigned long long single\_pow = a;

for (int i = (int)b.size() - 1; i >= 0; i--)

{

for (int j = 0; j < b[i]; j++)

{

product = product \* single\_pow;

product %= 1337;

}

unsigned long long sum\_pow = 1;

for (int j = 0; j < 10; j++)

{

sum\_pow \*= single\_pow;

sum\_pow %= 1337;

}

single\_pow = sum\_pow;

}

return (int)product;

}

## 396. Rotate Function

Medium

Given an array of integers A and let *n* to be its length.

Assume Bk to be an array obtained by rotating the array A *k* positions clock-wise, we define a "rotation function" F on A as follow:

F(k) = 0 \* Bk[0] + 1 \* Bk[1] + ... + (n-1) \* Bk[n-1].

Calculate the maximum value of F(0), F(1), ..., F(n-1).

**Note:**  
*n* is guaranteed to be less than 105.

**Example:**

A = [4, 3, 2, 6]

F(0) = (0 \* 4) + (1 \* 3) + (2 \* 2) + (3 \* 6) = 0 + 3 + 4 + 18 = 25

F(1) = (0 \* 6) + (1 \* 4) + (2 \* 3) + (3 \* 2) = 0 + 4 + 6 + 6 = 16

F(2) = (0 \* 2) + (1 \* 6) + (2 \* 4) + (3 \* 3) = 0 + 6 + 8 + 9 = 23

F(3) = (0 \* 3) + (1 \* 2) + (2 \* 6) + (3 \* 4) = 0 + 2 + 12 + 12 = 26

So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.

### Analysis:

For each rotation, you are make the right most element as 0 and adding the sum of the remaining number, which can be calculated as the sum of subarray, by using the total sum deduct the missing number.

/// <summary>

/// Leet code #396. Rotate Function

///

/// Given an array of integers A and let n to be its length.

/// Assume Bk to be an array obtained by rotating the array A k positions

/// clock-wise, we define a "rotation function" F on A as follow:

/// F(k) = 0 \* Bk[0] + 1 \* Bk[1] + ... + (n-1) \* Bk[n-1].

/// Calculate the maximum value of F(0), F(1), ..., F(n-1).

/// Note:

///

/// Example:

/// A = [4, 3, 2, 6]

/// F(0) = (0 \* 4) + (1 \* 3) + (2 \* 2) + (3 \* 6) = 0 + 3 + 4 + 18 = 25

/// F(1) = (0 \* 6) + (1 \* 4) + (2 \* 3) + (3 \* 2) = 0 + 4 + 6 + 6 = 16

/// F(2) = (0 \* 2) + (1 \* 6) + (2 \* 4) + (3 \* 3) = 0 + 6 + 8 + 9 = 23

/// F(3) = (0 \* 3) + (1 \* 2) + (2 \* 6) + (3 \* 4) = 0 + 2 + 12 + 12 = 26

/// So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.

/// </summary>

int LeetCodeMath::maxRotateFunction(vector<int>& A)

{

int sum = 0;

int max\_value = 0;

int function = 0;

for (size\_t i = 0; i < A.size(); i++)

{

sum += A[i];

function += A[i] \* i;

}

max\_value = function;

for (size\_t i = 1; i < A.size(); i++)

{

function += sum - A[A.size() - i] \* A.size();

max\_value = max(max\_value, function);

}

return max\_value;

}

## 780. Reaching Points

Hard

A move consists of taking a point (x, y) and transforming it to either (x, x+y) or (x+y, y).

Given a starting point (sx, sy) and a target point (tx, ty), return True if and only if a sequence of moves exists to transform the point (sx, sy) to (tx, ty). Otherwise, return False.

**Examples:**

**Input:** sx = 1, sy = 1, tx = 3, ty = 5

**Output:** True

**Explanation:**

One series of moves that transforms the starting point to the target is:

(1, 1) -> (1, 2)

(1, 2) -> (3, 2)

(3, 2) -> (3, 5)

**Input:** sx = 1, sy = 1, tx = 2, ty = 2

**Output:** False

**Input:** sx = 1, sy = 1, tx = 1, ty = 1

**Output:** True

**Note:**

* sx, sy, tx, ty will all be integers in the range [1, 10^9].

### Analysis:

Because x and y are all positive, so from tx, ty and calculate towards sx to sy will be much more easy, you only need to deduct the bigger number among tx, ty.

/// <summary>

/// Leet code #780. Reaching Points

///

/// A move consists of taking a point (x, y) and transforming it to either

/// (x, x+y) or (x+y, y).

///

/// Given a starting point (sx, sy) and a target point (tx, ty), return

/// True if and only if a sequence of moves exists to transform the point

/// (sx, sy) to (tx, ty). Otherwise, return False.

///

/// Examples:

/// Input: sx = 1, sy = 1, tx = 3, ty = 5

/// Output: True

/// Explanation:

/// One series of moves that transforms the starting point to the target is:

/// (1, 1) -> (1, 2)

/// (1, 2) -> (3, 2)

/// (3, 2) -> (3, 5)

///

/// Input: sx = 1, sy = 1, tx = 2, ty = 2

/// Output: False

///

/// Input: sx = 1, sy = 1, tx = 1, ty = 1

/// Output: True

///

/// Note:

///

/// sx, sy, tx, ty will all be integers in the range [1, 10^9].

/// </summary>

bool LeetCodeMath::reachingPoints(int sx, int sy, int tx, int ty)

{

while (tx > 0 && ty > 0)

{

if ((sx == tx) && (sy == ty)) return true;

if (tx > ty)

{

int n = (tx - sx) / ty;

tx -= n \* ty;

}

else

{

int n = (ty - sy) / tx;

ty -= n \* tx;

}

}

return false;

}