## 970. Powerful Integers

Medium Hash Table **Math** 

# **Problem Description**

The twist here is that we are only interested in those sums that do not exceed a given limit, bound. The result should be a list of these powerful integers, and each should only appear once, regardless of the order they're presented in. Intuition

Given three integers x, y, and bound, the task is to find all unique powerful integers. A powerful integer is defined as the sum of

x raised to the power of i and y raised to the power of j, where i and j are non-negative integers (i.e., i >= 0 and j >= 0).

by a maximum value. To solve this efficiently, we think of what happens when we increment these exponents: as i or j increase, the value of  $x^i$  or  $y^j$  grows exponentially. So, even if x and y are as small as 2, by the time they are raised to the 20th power, they are already greater than  $10^6$  ( $2^2$ ) >  $10^6$ ), which exceeds the typical constraints of bound.

The key to solving this problem is recognizing that we only need a reasonable range of powers for x and y since they're bound

Based on this observation, we can intelligently limit our search for potential values of i and j. This way we avoid calculating large powers that would exceed the bound. More specifically, we need to ensure the sums x^i + y^j are within the bound, and we only need to look for values of i and j that meet this requirement.

When either x or y is 1, it doesn't make sense to keep increasing the power, because 1 raised to any power is still 1. This means that if x or y is 1, we loop through its power only once, rather than continuing to increase it, which would just give us the same result. If neither x nor y is 1, we proceed with the nested loop, ensuring each time that the sum x^i + y^j does not exceed

The solution uses a set to collect the sums to automatically handle duplicates and ensure that each powerful integer is only counted once. After looping through all potential combinations, we convert the set to a list to provide the answer that matches the problem's expected return type.

**Solution Approach** 

The implementation strategy for finding all powerful integers within the specified bound takes advantage of some basic

algorithmic principles and the properties of mathematical exponents. Let's go step by step through what the provided Python

#### Set Data Structure: The solution uses a Python set, called ans, as the primary data structure to store the powerful integers. The set is chosen because it can automatically manage duplicates, which means any sum computed more than once will only

solution does:

bound.

be stored once in the set. Loop Bounds for Efficiency: To keep the algorithm efficient, we determine the powers of x and y within a restricted loop bound, as the Reference Solution Approach suggests. If x and y are both greater than 1, i and j will never need to exceed

20 because x^i or y^j would be greater than 10^6. Hence, we can cap our loop to not exceed the bound.

and y. The outer loop is for the variable a representing  $x^i$ , and the inner loop for b representing  $y^j$ . Breaking Conditions: Within each loop iteration, we check that the current powers' sum a + b is less than or equal to bound. If it isn't, we don't need to explore higher powers of y for the current x exponent and break out of the inner loop. Similarly, if

x is 1, increasing i won't change the value of a, so we break out of the outer loop. The same logic applies for y.

within the bound. The set is cast to a list, and then the list is returned as the final result.

= 8 and  $3^3$  = 27, and any sum involving 27 would exceed our bound of 10.

Now enter the inner loop with j = 0; calculate  $b = 3^j = 3^0 = 1$ .

j = 3 would exceed our cutoff, so we do not increment j further.

Check if a + b = 1 + 3 = 4 is less than or equal to 10, it is, so add 4 to our set ans.

**Result Construction:** The values in our set ans are the powerful integers: {2, 4, 5, 10}.

integers does not matter, and the uniqueness is guaranteed by the set data structure.

**Nested While Loop and Exponentiation:** A double-nested while loop is employed to go through all the viable powers of x

after the first iteration. This is because any exponent of 1 is still 1, and so further iterations would not provide any new sums. Adding to the Set: For each valid pair of powers found within the loops, the sum a + b is added to the set ans.

**Early Termination**: If we find that x or y is 1, we can optimize further by breaking out of the respective loop immediately

Final Result: Once all potential a and b values have been tried, the set ans contains all the unique powerful integers that fall

- The above solution ensures we consider all possible sums within the constraint efficiently without redundant calculations or concerns about duplicates.
- **Step-by-step Calculation: Initial Set:** We start with an empty set ans to hold the values of all unique powerful integers.

**Identify Loop Bounds:** We assume the cutoff for i and j is 2 because if x or y is raised to the power of 3, the result is  $2^3$ 

represented as 2^i + 3^j where i and j are non-negative integers, and the result does not exceed the bound of 10.

Let's assume our input values are x = 2, y = 3, and bound = 10. Our goal is to find all unique powerful integers that can be

### **Nested Loops:** Start with i = 0; calculate $a = 2^i = 2^0 = 1$ .

**Example Walkthrough** 

Check if a + b = 1 + 1 = 2 is less than or equal to 10, it is, so add 2 to our set ans. Stay in the inner loop with j = 1; calculate  $b = 3^j = 3^1 = 3$ .

## Check if a + b = 1 + 9 = 10 is less than or equal to 10, it is, so add 10 to our set ans.

it's not added.

Solution Implementation

 $base_x = 1$ 

while base x <= bound:</pre>

if y == 1:

break

 $base_y = 1$ 

**Python** 

**Next Outer Loop Iteration:** 

formed without exceeding the bound.

# Initialize the base value for x to 1

while base x + base y <= bound:</pre>

# Initialize the base value for y to 1

• With i = 1; calculate  $a = 2^i = 2^1 = 2$ .

Increment j to 2; calculate  $b = 3^j = 3^2 = 9$ .

**Last Outer Loop Iteration:**  $\circ$  With i = 2; calculate  $a = 2^i = 2^2 = 4$ .

• Repeat the inner loop with j = 0 and j = 1, adding 4 + 1 and 4 + 3 to our set ans, but the sum with j = 2 would exceed the bound, so

**Loop Completion:** Once all combinations of i and j have been tested, we find that no further valid powerful integers can be

After converting this set to a list, the final answer we return is [2, 4, 5, 10]. It's important to note that the order of these

Repeat the inner loop with j = 0 through 2, adding 2 + 1, 2 + 3, and 2 + 9 to our set ans if not exceeding the bound.

- class Solution: def powerfulIntegers(self, x: int, v: int, bound: int) -> List[int]: # Initialize an empty set to hold the unique powerful integers powerful\_ints = set()
- powerful\_ints.add(base\_x + base\_y) # Multiply base\_y by y to get the next power of y base\_y \*= y

# If v is 1, it will always be 1, so break to avoid infinite loop

# Add the sum of the powers to the set to ensure uniqueness

# If x is 1, it will always be 1, so break to avoid infinite loop

# Multiply base\_x by x to get the next power of x

// Function to find powerful integers that can be expressed as x^i + y^i

public List<Integer> powerfulIntegers(int x, int v, int bound) {

// Iterate over all possible values of  $x^i$  where  $i \ge 0$ 

// Iterate over all possible values of y^; where i >= 0

for (int powY = 1; powX + powY <= bound; powY \*= y) {

// and the sum of  $x^i$  and  $y^i$  is within the given bound.

// Convert the set of powerful integers into a vector and return it.

vector<int> result(powerfulInts.begin(), powerfulInts.end());

\* Returns an array of unique integers that are the sum of powers of

\* @param  $\{number\} \times - The base for the first term of the sum.$ 

\* @param  $\{number\}$  v - The base for the second term of the sum.

\* the given `x` and `y` that are less than or equal to the given `bound`.

powerfulInts.insert(powX + powY); // Insert the sum into the set.

// If v is 1, then  $y^j$  will always be 1 and we won't get new sums, break out of the loop.

// If x is 1, then  $x^i$  will always be 1 and we won't get new sums, break out of the loop.

// and the result is within the given bound.

if (y == 1) {

if (x == 1) {

return result;

break;

break;

for (int powX = 1; powX <= bound; powX \*= x) {

# Loop through increasing powers of y and add to base\_x while the sum is within the bound

// where i and i are integers, x and y are given values, and the result is less than or equal to the bound.

# Loop through increasing powers of x while it's less than or equal to the bound

```
base_x *= x
# Convert the set to a list to return the powerful integers
return list(powerful_ints)
```

Java

class Solution {

if x == 1:

break

```
// A set to store the unique powerful integers
        Set<Integer> powerfulInts = new HashSet<>();
        // Iterate over the powers of x
        // a will be x^i
        for (int exponent0fX = 1; exponent0fX <= bound; exponent0fX *= x) {
            // Iterate over the powers of y
            // b will be y^i
            for (int exponentOfY = 1; exponentOfX + exponentOfY <= bound; exponentOfY *= y) {</pre>
                // Add the sum of the powers of x and y to the set
                powerfulInts.add(exponent0fX + exponent0fY);
                // If y is 1, then y^j will always be 1, so we can break the loop to prevent infinite loop
                if (v == 1) {
                    break;
            // If x is 1, then x^i will always be 1, so we can break the loop to prevent infinite loop
            if (x == 1) {
                break;
        // Convert the set of integers to a list and return it
        return new ArrayList<>(powerfulInts);
C++
#include <vector>
#include <unordered set>
using namespace std;
class Solution {
public:
    // This method finds all powerful integers within a given bound.
    // An integer is powerful if it can be expressed as x^i + y^i
    // where x and y are positive integers and i and i are non-negative integers.
    vector<int> powerfulIntegers(int x, int v, int bound) {
        unordered_set<int> powerfulInts; // Use a set to store unique powerful integers.
```

```
* @param {number} bound - The upper limit for the sum of powers.
* @returns {number[]} - A unique set of integers which are the sum of powers.
```

**}**;

**TypeScript** 

```
function powerfulIntegers(x: number, y: number, bound: number): number[] {
    // Initialize a new Set to store unique sums.
    const uniqueSums = new Set<number>();
    // Iterate over powers of `x`.
    for (let powerOfX = 1; powerOfX <= bound; powerOfX *= x) {</pre>
        // Iterate over powers of `v`.
        for (let powerOfY = 1; powerOfX + powerOfY <= bound; powerOfY *= y) {</pre>
            // Add the sum of the current powers of `x` and `y` to the Set.
            uniqueSums.add(powerOfX + powerOfY);
            // If `v` is 1, we reach a fixed point as `v` to any power is still 1.
            // No need to proceed further in the inner loop.
            if (y === 1) {
                break;
       // If `x` is 1, we reach a fixed point as `x` to any power is still 1.
       // No need to proceed further in the outer loop.
        if (x === 1) {
            break;
    // Convert the Set of unique sums into an array and return it.
    return Array.from(uniqueSums);
// Usage example:
// const result = powerfulIntegers(2, 3, 10);
// console.log(result); // Output may be something like [2, 3, 4, 5, 7, 9, 10]
class Solution:
    def powerfulIntegers(self, x: int, y: int, bound: int) -> List[int]:
       # Initialize an empty set to hold the unique powerful integers
        powerful_ints = set()
       # Initialize the base value for x to 1
       base x = 1
       \# Loop through increasing powers of x while it's less than or equal to the bound
       while base x <= bound:</pre>
           # Initialize the base value for y to 1
            base_y = 1
           # Loop through increasing powers of y and add to base_x while the sum is within the bound
           while base x + base y <= bound:</pre>
                # Add the sum of the powers to the set to ensure uniqueness
                powerful_ints.add(base_x + base_y)
                # Multiply base_y by y to get the next power of y
                base_y *= y
                # If v is 1, it will always be 1, so break to avoid infinite loop
                if v == 1:
                    break
           # If x is 1, it will always be 1, so break to avoid infinite loop
           if x == 1:
                break
           # Multiply base_x by x to get the next power of x
           base_x *= x
```

## Time and Space Complexity The time complexity of the code is determined by the nested while loops that iterate over the variables a and b. Since both a

return list(powerful\_ints)

 $O((\log bound)^2)$  for x and y greater than 1.

# Convert the set to a list to return the powerful integers

1, and only once if x == 1. Similarly, for each value of a, the inner while loop runs until b exceeds bound - a, which happens after  $0(log_y(bound))$  iterations if y > 1, and only once if y == 1. The overall time complexity is, therefore,  $0(log_x(bound)) *$  $log_y(bound)$ ), which can also be expressed as  $0((log_bound)^2)$  assuming x and y are greater than 1. If either x or y is 1, the complexity would drop to  $0(\log bound)$  or 0(1) for x and y being 1 respectively. The space complexity is determined by the size of the set ans, which contains all the different sums of a and b that do not exceed bound. In the worst case, every pair (a, b) is unique before reaching the bound. The size of the set would then be proportional to the number of pairs we can form, which is  $0(\log_x(bound) * \log_y(bound))$ , hence making the space complexity

and b are exponentially increased by multiplying by x and y, the number of iterations is proportional to the logarithm of the

bound. More specifically, the outer while loop runs until a exceeds bound, which happens after 0(log\_x(bound)) iterations if x >

In summary: • Time Complexity:  $0((\log bound)^2)$  when x > 1 and y > 1;  $0(\log bound)$  when either x or y is 1; 0(1) when both x and y are 1.

• Space Complexity: 0((log\_bound)^2) for x and y greater than 1; otherwise, it will be less depending on the number of unique sums obtainable when either x or y equals 1.