The Power Sum

locked

by PRASHANTB1984

Editorial by kevinsogo

Statistics

Difficulty: **Medium**
Required Knowledge: **Backtracking, recursion**
Publish Date: **Mar 10 2017**

The obvious brute-force solution for this problem is simply to *enumerate* all possible ways to express X as a sum of distinct Nth powers.

A good approach here would be to use *recursive backtracking*, a good general-purpose technique for enumerating things.

By the way, one must be careful not to double count equivalent expressions. For instance, this problem considers $9^3 + 10^3$ and $10^3 + 9^3$ to be the same sum. (The explanation section clarifies this.) We should make sure we generate each expression only once.

How can we check if two expressions are the same? Well, one way is simply to sort their summands and compare term by term. This works because for every expression, there is a unique equivalent expression whose terms are in sorted order. This suggests that we should only look at *sorted* expressions. Thus, it's sufficient to just ensure that the list of numbers we're generating is always in sorted order, i.e., increasing.

An example done by hand

Let's see how one would enumerate all possible expressions by hand. Let's use the example $X = 29$ and $N = 2$. We will try to construct the expressions *systematically* by checking all possibilities in increasing order.

- Try 1 first. Our expression currently looks like $1^2 = 1$. The next number must be ≥ 2.
  - Try 2. Now it looks like $1^2 + 2^2 = 5$. The next number must be ≥ 3.
    - Try 3. Now, we have $1^2 + 2^2 + 3^2 = 14$.
      - Try 4. We have $1^2 + 2^2 + 3^2 + 4^2 = 30$. *Backtrack*, since it exceeds 29.
    - Try 4. We have $1^2 + 2^2 + 4^2 = 21$.
      - Try 5. We have $1^2 + 2^2 + 4^2 + 5^2 = 46$. *Backtrack*.
    - Try 5. We have $1^2 + 2^2 + 5^2 = 30$. *Backtrack*.
  - Try 3. We have $1^2 + 3^2 = 10$.
    - Try 4. We have $1^2 + 3^2 + 4^2 = 26$.
      - Try 5. We have $1^2 + 3^2 + 4^2 + 5^2 = 51$. *Backtrack*.
    - Try 5. We have $1^2 + 3^2 + 5^2 = 35$. *Backtrack*.
  - Try 4. We have $1^2 + 4^2 = 17$.
    - Try 5. We have $1^2 + 4^2 + 5^2 = 42$. *Backtrack*.
  - Try 5. We have $1^2 + 5^2 = 26$.
    - Try 6. We have $1^2 + 5^2 + 6^2 = 62$. *Backtrack*.
  - Try 6. We have $1^2 + 6^2 = 37$. *Backtrack*.
- Try 2. We have $2^2 = 4$.
  - Try 3. We have $2^2 + 3^2 = 13$.
    - Try 4. We have $2^2 + 3^2 + 4^2 = 29$. **We've found a solution!**
    - Try 5. We have $2^2 + 3^2 + 5^2 = 38$. *Backtrack*.
  - Try 4. We have $2^2 + 4^2 = 20$.

        - Try 5. We have $2^2 + 4^2 + 5^2 = 45$. *Backtrack*.
    - Try 5. We have $2^2 + 5^2 = 29$. **We've found a solution!**
    - Try 6. We have $2^2 + 6^2 = 40$. *Backtrack*.
- Try 3. We have $3^2 = 9$.
    - Try 4. We have $3^2 + 4^2 = 25$.
        - Try 5. We have $3^2 + 4^2 + 5^2 = 50$. *Backtrack*.
    - Try 5. We have $3^2 + 5^2 = 34$. *Backtrack*.
- Try 4. We have $4^2 = 16$.
    - Try 5. We have $4^2 + 5^2 = 41$. *Backtrack*.
- Try 5. We have $5^2 = 25$.
    - Try 6. We have $5^2 + 6^2 = 61$. *Backtrack*.
- Try 6. We have $6^2 = 36$. *Backtrack*.

After the last backtrack, the enumeration ends.

Notice a couple of things.

- Each term has to be strictly greater than the previous term, so we always start checking from the previous number plus one. (At the very beginning, we start at 1.)
- We *backtrack* if the sum of the current expression exceeds X, since adding more terms will just increase the sum even further.
- We only *go deeper* if the current sum so far is *strictly smaller* than X.
- On the (rare) occasion that we get a sum of exactly X, it means we've found a solution! This also means that we should backtrack (since adding more terms will just increase the sum).

A crucial observation here is that *the algorithm that we're performing at every level is basically the same: enumerate the possibilities for the next number, and go deeper if necessary.* This nudges us to implement a *recursive backtracking* solution.

We can formalize the algorithm by writing it in pseudocode. To be more specific, we will define a function, called `count_expressions`, which will attempt to generate all decompositions of X as the sum of distinct Nth powers, following the general strategy we've exemplified above. Each call of this function will represent "the current step". Also, we will need to have access to the *expression generated so far*, so it should be passed as an argument to this function.

In this pseudocode, `vals` represents the list of numbers representing the terms in the expression. We also use `**` for exponentiation (since `^` is more popularly used for a different operation).

```
def count_expressions(x, n, vals):

    # compute the sum
    s = 0
    for v in vals:
        s += v**n

    if s == x:
        # we've found a solution!
        return 1
    else:
        # compute the smallest possible next value
        if vals.empty():
            v = 1
        else:
            v = (last element of vals) + 1

        answer = 0
        loop forever:
            # try v
            if s + v**n > x:
                # backtrack
                return

            # go deeper
            answer += count_expressions(x, n, vals + [v])

            # increment to get the next v
            v++
```

```
        return answer
```

To get the answer, one initially calls `count_expressions(x, n, [])` where `[]` represents the empty list. The returned value will be the answer.

Here, `vals + [v]` is the list obtained by adding `v` at the end of `vals`. This is supposed to create a new copy of `vals`, so the original `vals` list is not affected. (Since this is pseudocode, the actual implementation in your programming language will probably be a bit different.)

Here is the same code without the comments (and with slight refactoring):

```
def count_expressions(x, n, vals):

    s = sum(v**n for v in vals)

    if s == x:
        return 1
    else:
        v = (if vals.empty() then 1 else (last element of vals) + 1)

        answer = 0
        while s + v**n <= x:
            answer += count_expressions(x, n, vals + [v])
            v++
        return answer
```

Here are a few implementations.

C++:

```
#include <iostream>
#include <vector>
using namespace std;

int ipow(int b, int e) {
    if (e == 0)
        return 1;
    return e == 0 ? 1 : b * ipow(b, e - 1);
}

int count_expressions(int x, int n, vector<int>& vals) {

    int s = 0;
    for (int v : vals) {
        s += ipow(v, n);
    }

    if (s == x) {
        return 1;
    } else {
        int answer = 0;
        int v = vals.empty() ? 1 : vals.back() + 1;
        while (s + ipow(v, n) <= x) {
            vals.push_back(v);
            answer += count_expressions(x, n, vals);
            vals.pop_back();
            v++;
        }
        return answer;
    }
}

int main() {
    int x, n;
    cin >> x >> n;
    vector<int> vals;
    cout << count_expressions(x, n, vals) << endl;
}
```

Python:

```
def count_expressions(x, n, vals):

    s = sum(v**n for v in vals)

    if s == x:
        return 1
    else:
        answer = 0
        v = vals[-1] + 1 if vals else 1
        while s + v**n <= x:
            answer += count_expressions(x, n, vals + [v])
            v += 1

        return answer

print(count_expressions(int(input()), int(input()), []))
```

You can play with this code and better understand how it works by adding a bunch of print statements. For example, you could have something like the following, which prints the current expression and also tells us whenever we've found a solution.

```
from sys import stderr

def count_expressions(x, n, vals):

    s = sum(v**n for v in vals)

    print('The current expression looks like:', ' + '.join('{}^{}'.format(v
, n) for v in vals), file=stderr)

    if s == x:
        print("We've found a solution!", file=stderr)
        return 1
    else:
        answer = 0
        v = vals[-1] + 1 if vals else 1
        while s + v**n <= x:
            answer += count_expressions(x, n, vals + [v])
            v += 1

        return answer

print(count_expressions(int(input()), int(input()), []))
```

Doing things like this also helps in debugging. (You'll probably want to learn about *debuggers* at some point, though.)

There's one possible improvement. Notice that we don't really need to remember the full expression at every step; to be able to decide what to do next, we only need to know the *sum so far* and the *last element*. This gives us the following solution:

```
def count_expressions(x, n, s, v):
    if s == x:
        return 1
    else:
        v++
        answer = 0
        while s + v**n <= x:
            answer += count_expressions(x, n, s + v**n, v)
            v++
        return answer
```

Here, `s` represents the "sum so far", and `v` represents the last element. The answer is then obtained as `count_expressions(x, n, 0, 0)`.

One can see that this is faster than the previous solution since we're not creating any lists, or even manipulating lists in any way. We're also not computing `s` from scratch every time, which removes one loop.

Finally, I'd like to mention that there's a way to solve this problem even if X is slightly

larger, say in the 10^5 range, which drastically increases the number of expressions we need to check and making enumeration solutions quite slow. This technique is called *dynamic programming*. To learn it, I suggest you start by trying to solve some of the problems in the dynamic programming section.