C - Divide and Divide Editorial by en_translator

Let function $f(N)$ be "the cost required to break an integer N written on the blackboard into ones." Then $f(N)$ satisfies the following recurrence relation:

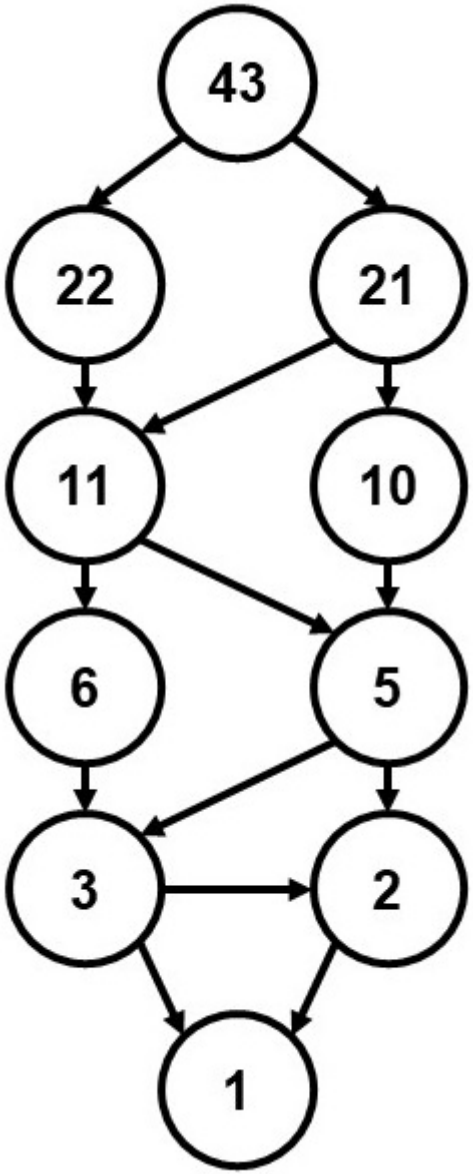
$$f(N) = \begin{cases} 0 & (N = 1) \\ f(\lfloor \frac{N}{2} \rfloor) + (\lceil \frac{N}{2} \rceil) + N & (N \neq 1) \end{cases}$$

What we want is $f(N)$. If N is sufficiently small, one fan for instance use a recursive function as in the following sample code to solve the problem. However, this implementation requires so long execution time when N is large, leading to TLE (Time Limit Exceeded) verdict.

- Naive implementation (resulting in TLE)

```
#include <iostream>
long long f(long long N) {
  return N == 1 ? 0 : f(N / 2) + f((N + 1) / 2) + N;
}
int main() {
  long long N;
  cin >> N;
  cout << f(N) << endl;
}
```

Let us consider how can we optimize it using the properties of the function.
We try writing the call graph of the function. That is, $f(43)$ calls $f(22)$ and $f(21)$, so try drawing an arrow from 43 to 22 and 21.
Then we obtain the following figure (when $N = 43$):



As you can see in the figure, it turns out that only a few states are visited. Specifically, the following two kinds of states are visited:

- $43/2^n$ rounded down: $(21, 10, 5, 2, 1)$
- $43/2^n$ rounded up: $(22, 11, 6, 3, 2, 1)$

This fact can be generalized. In the sample code above, the argument N on the function `f(N)` is classified into the following two types:

- "integers represented as $\lfloor \frac{N}{2^n} \rfloor$ for some integer n" and "integers represented as $\lceil \frac{N}{2^n} \rceil$ for some integer n."

(It can be proved by induction, but it is complicated, so we will omit it here.)

Therefore, one can optimize the implementation using the technique called *memorized recursion," which can be used to optimize recursive functions.

We explain memorized recursion. For a function $f(N)$, consider maintaining a dictionary (associated array) whose keys are N and values are the return values of $f(N)$, and storing $f(N)$ on its first evaluation. On the second and succeeding calls, return the stored value without actually evaluating the function. Such an optimization technique is called memorized recursion. (We adopted the explanation in the editorial of ABC247C.)

With memorized recursion, the function is no longer called twice for the same argument, leading to a significant improvement on the computational cost. Indeed, while the complexity without memorized recursion is $\mathrm{O}(N)$, once adopted it is reduced to about $\mathrm{O}(\log N)$ or $\mathrm{O}(\log^2 N)$.

Therefore, the problem can be solved using memorized recursion. Sample code follows below.

- Sample code (C++)
  - In C++, one can use a dictionary type like `std::map` to memorize a recursive function.

```cpp
#include <iostream>
#include <map>
using namespace std;
map<long long, long long> m;
long long f(long long N) {
  if (N == 1) return 0;
  if (m.count(N)) return m[N];
  return m[N] = f(N / 2) + f((N + 1) / 2) + N;
}
int main() {
  long long N;
  cin >> N;
  cout << f(N) << endl;
}
```

- Sample code (Python)
  - In Python, one can add a decorator called `@cache` to automatically turn a recursive function into a memorized one. (This first appeared in Python 3.9, and is more convenient than conventional `@lru_cache`.)

```
from functools import cache
@cache
def f(N):
    return 0 if N == 1 else f(N // 2) + f((N + 1) // 2) + N
print(f(int(input())))
```