6003 project report

Problem: Page 5, the prime number problem

Main points for the solution

The problem is asking a minimum value of ***k*** with given a number of the possibility of its prime combination multinomial. As we know the prime number is [2,3,5,7…] which are monotonous increasing. So we can model this problem as ***k*** = $2^{e1} + 3^{e2} + 5^{e3} + \ldots.$ that $e_1 \geq e_2 \geq e_3$ and $1 \leq k \leq 2^{63}$. To find the minimum ***k***, we can greedy by pick more from minimum prime number.

Regarding to the programming running efficiency concern, because the given condition is that $n \leq 2^{63}$, we can get all numbers ***k*** that in this scope. Then we can build a table with all the possible n for k. When enter ***n*** from keyboard, the algorithm will look to the table to find minimum number of the number of primes combinations.

Build a combination possibility tableau

We have two tableaus in the algorithm. One is a tableau to store the combination possibilities, named c[63][63], each cell declares as long type. The combination possibilities of the primes obey the multinomial $\binom{e1+e2+e3+..+en}{e1,e2,e3..en}$ $= \frac{(e1+e2+\cdots+en)!}{e1!e2!\ldots en!}$. In our case $n \leq 15$.

For example:

$30 = 2*3*5 \rightarrow \frac{(1+1+1)!}{1!1!1!} = 6$

$20 = 2*2*5 \rightarrow \frac{(2+1)!}{2!1!} = 3$

$100 = 2*2*5*5 \rightarrow \frac{(2+2)!}{2!2!} = 6$

The possibility of combination is a reference for ***n***. Essentially, it means to pick ***j*** items out from ***i*** items. Each picked ***j*** could be considered as one object, and the others are another one object. When add a new prime to the multinomial, it equals the size increase one and So we can recursive each one to C(i, j) = C(i – 1, j – 1) + C(i – 1, j)

A base for compute multinomial number tableau looks like:

1										
2	1									
3	3	1								
4	6	4	1							
5	10	10	5	1						
6	15	20	15	6	1					
7	21	35	35	21	7	1				
8	28	56	70	56	28	8	1			
…	…	…	…	…	…	…	…	…		
…	…	…	…	…	…	…	…	…	…	
62	1891	37820	557845	6471002	61474519	491796152	3381098545	…	62	1

Some of the numbers of combination are not existed because in ***k*** exceed 2^{63}. For example regarding to n = 64, in

this case for minimum k, we have 63 times to pick 2 and 1 time pick 3, k value is $2^{63} * 3$ which is exceed.
For example, “4,6,4,1” mean if we have four exponents, if one is different from another three, we have four possibilities; if we have two exponents different from another two there are 6 possibilities, etc.
In this case, we built this table for next step of go through all the combinations within limit with a maximum of ***k***.

Build the tableau for minimum k

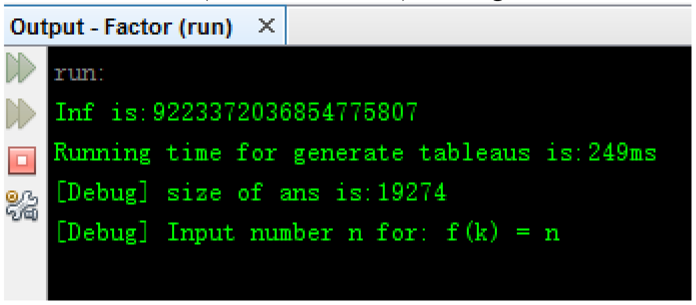
The second tableau is a hash-table named ***ans***. It has key and value. Key column save ***n*** and value save ***k.*** During programming, after entered ***n***, it will search the hash-table by key then print value.
As we can see given ***n***, the ***k*** value could be different. To find the minimum ***k***, we need to arrange the exponents from the minimum prime then to bigger ones. For example, while entering ***n*** = 4, looking from the tableau 1, it shows four primes with one different to the other three will satisfy it. The combination is 2*2*2*3 (because we pick from minimum prime 2, then put the different one to be the second smallest one which is 3), the result is 24.
The key method to build the multinomial is to recursive the exponents on prime number list. Because the problem indicates ***n*** < 2^{63}, to ensure the speed of computation, for each input number, the program computes ***f(k)*** and make it less than 2^{63}.
During the iteration, the algorithm will always put smaller ***k*** to the hash-table. So the algorithm will compute the combination of $e_1, e_2, \ldots e_{16}$ but not exceed limit, then catch all the minimum ***k*** with his n value, discard those bigger values.
To make this hash-table, first we make exponent increasing for each prime. Let the exponents of $e_1, e_2, \ldots e_{16}$ increase 1 each time, respectively. For each time of iteration, first check if it can put next prime in the multinomial, if ***k*** does not exceed, bring current ***n*** and ***k*** value as input value to next iteration, and make ***i*** plus 1 which it will point to next prime in the list, and increase counter to point next cell in the possibility table because a new prime will involve.
Regarding to how to compute ***n*** in the process, at the beginning, there is only one prime “2” with exponent 1. Each time involve a new prime, we can treat previous combination as instance of ***n*** and ***k***, and treat the new prime as different. So it increases the size of multinomial ***t*** plus 1, so the possibilities will also increase based on previous n times new possibilities by increasing ***t***.

Complexity

The complexity to generate the combination possibilities table is O(N*N) in our case it is 62*62. The number of iteration to generate the hash-table in our case is 2153869 times. It depends on the limitation which is 2^{63} , the number of primes which satisfy their value of multiple each other like 2*3*5*7… not exceed the limitation.
The space for the combination table is 63*63/2 while for hash-table 19274 lines.

Running efficiency

Since the solution will compute all ***n*** and ***k*** before allow user input n, the main time consumption goes for generate two tableaus. Due to my test environment, running on i7 processor, windows8, JAVA 7, the time to generate tableaus is below 1 second (249 ms in this test). See Fig 1.



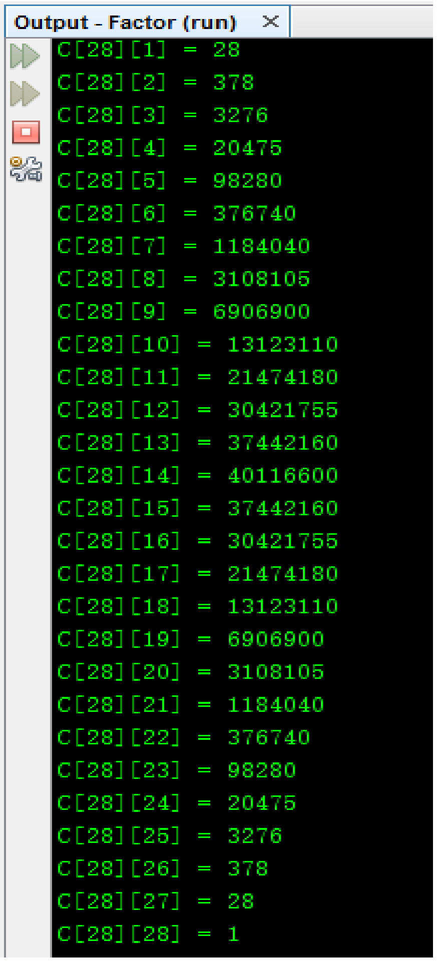
```
run:
Inf is:9223372036854775807
Running time for generate tableaus is:249ms
[Debug] size of ans is:19274
[Debug] Input number n for: f(k) = n
```

Fig 1. Running time

Since JAVA has inherited design regarding memory access comparing with C/C++, it costs a little more time in our algorithm. So we could predict a little slower than C/C++, in some test environment.

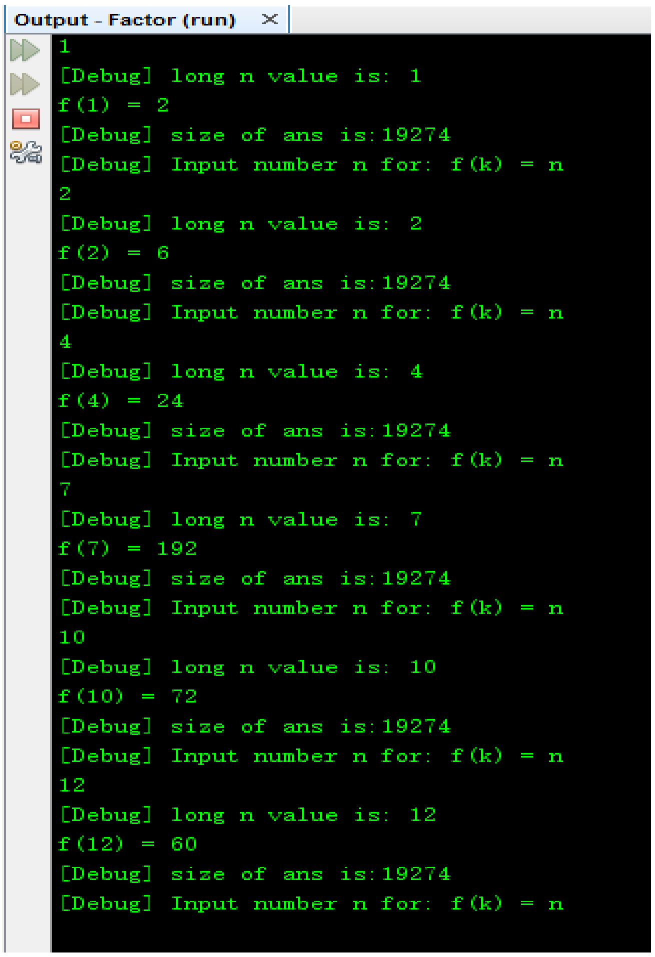
Sample output screen shot

The tableau and running screenshot shows below: (Fig.2 and Fig.3)



```
C[28][1] = 28
C[28][2] = 378
C[28][3] = 3276
C[28][4] = 20475
C[28][5] = 98280
C[28][6] = 376740
C[28][7] = 1184040
C[28][8] = 3108105
C[28][9] = 6906900
C[28][10] = 13123110
C[28][11] = 21474180
C[28][12] = 30421755
C[28][13] = 37442160
C[28][14] = 40116600
C[28][15] = 37442160
C[28][16] = 30421755
C[28][17] = 21474180
C[28][18] = 13123110
C[28][19] = 6906900
C[28][20] = 3108105
C[28][21] = 1184040
C[28][22] = 376740
C[28][23] = 98280
C[28][24] = 20475
C[28][25] = 3276
C[28][26] = 378
C[28][27] = 28
C[28][28] = 1
```

Fig 2. Generate possibility tableau



```
1
[Debug] long n value is: 1
f(1) = 2
[Debug] size of ans is:19274
[Debug] Input number n for: f(k) = n
2
[Debug] long n value is: 2
f(2) = 6
[Debug] size of ans is:19274
[Debug] Input number n for: f(k) = n
4
[Debug] long n value is: 4
f(4) = 24
[Debug] size of ans is:19274
[Debug] Input number n for: f(k) = n
7
[Debug] long n value is: 7
f(7) = 192
[Debug] size of ans is:19274
[Debug] Input number n for: f(k) = n
10
[Debug] long n value is: 10
f(10) = 72
[Debug] size of ans is:19274
[Debug] Input number n for: f(k) = n
12
[Debug] long n value is: 12
f(12) = 60
[Debug] size of ans is:19274
[Debug] Input number n for: f(k) = n
```

Fig 3. Given *n*, output minimum *k*

Conclusions

My solution to the problem has two aspects.
First, since every number can be treated as a multinomial of prime numbers, we can make all possible combinations on the exponent for each prime. Due to the limitation of 2^{63} so that we know it only needs first 16 primes.

Second, put all the result in memory, when input n, we can search for the k very fast rather than compute it respectively each time.

Appendix

Source code and comments

```
package factor;
import java.io.*;
import java.util.*;
import java.math.*;

public class Factor {

static long inf = Long.MAX_VALUE;
static int prime[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53}; // basic prime set, no need to include the 17th.
static long[][] c = new long[63][63]; //tableau for possibilities
static int counter = 0;

static HashMap ans = new HashMap(); // hashtable for n and k

static public void initarray(){ //initial array - c
    for(int i=0; i<63; i++) {
        for(int j=0; j<63; j++) {
            c[i][j] = 0;
        }
    }
}

static void search(long n, long k, int sum, int i) { //main method to build hashtable
    counter++;
    if (sum > 0) {
        if(ans.get(n) == null || k < (Long)ans.get(n)) { ////// if k is new or smaller, accept k to index n
            //System.out.println("[ans]n = "+n+", k="+k+", Sum = "+sum+", i = "+i+".");
            ans.put(n,k);
        }
    }
    long cur = k;
    int t = 1;
    while (true) {
        if (prime[i] > (inf/cur)) break; //check if can put this prime to current k, control not exceed.
        cur *= prime[i]; //multiple current prime to k
        if (c[sum + t][t] <= (inf/n)) //ensure the next n value from tableau will not exceed.
            //System.out.println("[ans building]cur = "+cur+", n = "+n+", k="+k+", Sum = "+sum+", i = "+i+".");
            search(n * c[sum + t][t], cur, sum + t, i + 1); //involve next prime to k, it could bring new possibilities to n, thus do this
iteration again with current situation
        t ++; //counter +1, to move to next line/column in combination possibility table for next iteration
    }
}
public static void main(String[] args) {
    initarray();
    c[0][0] = 1;
    for (int i = 1; i <= 62; i ++) { //build the possibility tableau
        c[i][0] = 1;
        for (int j = 1; j <= i; j ++) {
            if (c[i - 1][j] < inf && c[i - 1][j - 1] < inf) {
                c[i][j] = c[i - 1][j] + c[i - 1][j - 1];
            } else {
                c[i][j] = inf;
            }
        //System.out.println("C["+i+"]["+j+"] = "+c[i][j]);
        }
    }

    search(1, 1, 0, 0); //start iteration from n = k = 1
    System.out.println("Inf is:"+inf);
    System.out.println("counter is:"+counter);
    while (true) { //get n value from console
        long n;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("[Debug] size of ans is:"+ans.size());
        System.out.println("[Debug] Input number n for: f(k) = n");
        try{
           n = new Long(br.readLine()).longValue();
           System.out.println("[Debug] long n value is: "+n);
           System.out.println("f("+n+") = "+ans.get(n)); //search the hash-table for k
        }catch(Exception ioe){
            System.out.printf("IO error");
        }
    }
  }
}
```