피자 나눠 먹기(중)

종료일	
태그	arithmetic java propgrammers
상태	Done
작업 시간	15
# Level	0

제목 입력

> 프로그래머스 알고리즘 문제 URL
>
> https://school.programmers.co.kr/learn/courses/30/lessons/120815

문제 설명

▼ 문제

머쓱이네 피자가게는 피자를 여섯 조각으로 잘라 줍니다. 피자를 나눠먹을 사람의 수 `n`이 매개변수로 주어질 때, `n`명이 주문한 피자를 남기지 않고 모두 같은 수의 피자 조각을 먹어야 한다면 최소 몇 판을 시켜야 하는지를 return 하도록 solution 함수를 완성해보세요.

제한사항

▼ 제한사항

- $1 \leq n \leq 100$

입출력 예

▼ 입출력 예

n	result
6	1
10	5
4	2

▼ 입출력 예 설명

입출력 예 #1

- 6명이 모두 같은 양을 먹기 위해 한 판을 시켜야 피자가 6조각으로 모두 한 조각씩 먹을 수 있습니다.

입출력 예 #2

- 10명이 모두 같은 양을 먹기 위해 최소 5판을 시켜야 피자가 30조각으로 모두 세 조각씩 먹을 수 있습니다.

입출력 예 #3

- 4명이 모두 같은 양을 먹기 위해 최소 2판을 시키면 피자가 12조각으로 모두 세 조각씩 먹을 수 있습니다.

풀이

▼ 풀이과정

산수 문제 피자는 6조각으로 나눠주고 모든 인원의 최소 공배수(LCM)를 계산하여 최소 공배수에서 피자의 조각을 나누는 방식으로 문제 풀이 최소 공배수계산 과정을 단계별로 설명

1. 두 수 중에서 큰 수를 찾는다. 6, 10 (여기서 피자조각 6과 먹는 인원 n을 10으로 가정)
2. 큰 수의 배수를 차례로 증가시키면서, 작은 수로 나누어 떨어지는지 확인
    a. 10의 배수 : 10, 20, 30, 40….
    b. 6로 나누어 떨어지는지 확인
        i. 10 % 6 = 4
        ii. 20 % 6 = 2
        iii. 30 % 6 = 0
3. 나누어 떨어지는 첫 번째 수를 찾습니다. (여기서 30이 첫 번 째로 나누어 떨어지는 수)
4. 찾은 수가 최소 공배수

▼ 테스트 전체 코드

▼ 초기 코드

```
package level_0.sharingPizaTwo;

public class Main {
  public static void  main(String[]  args)throws Exception {
    Solution solution = new Solution();
    System.out.println("결과 값" + solution.solution(10));
  }

  public static class Solution{
    // TODO 피자 나눠 먹기
    // INFO : 피자는 6 조각이며, 나눠먹는 사람의 수 n 일때 피자를 남김 없이 다 먹어야한다. 이때 총 피자의 판 개수 return
    public int solution(int n) {
      int servings = 6;
      if(servings % n == 0) return 1;

      int lcm = Math.max(n,servings);

      while(lcm % n != 0 || lcm % servings != 0){
        System.out.printf("lcm : %d\tn 나머지 : %d\tservings 나머지 : %d\n",lcm, lcm % n , lcm % servings);
        lcm += Math.max(n,servings);
      }
     int answer = lcm / servings;
      return answer;
    }
```

```
  }
}
```

단점

1. `lcm` 값을 초기화할 때 `Math.max(n, servings)`를 사용하는데, `servings`와 `n`이 같을 경우에도 `n`을 선택하므로 약간의 성능 저하가 있을 수 있습니다. 이 경우에는 `lcm` 값을 초기화하는 로직을 수정할 수 있습니다.
2. 코드 실행 시간은 반복문의 반복 횟수에 따라 달라지기 때문에, 최소 공배수를 찾는 데에 시간이 오래 걸릴 수 있습니다. 특히, 입력된 `n` 값이 매우 큰 수인 경우 반복 횟수가 많아질 수 있습니다.
3. 반복문 내에서 `lcm` 값을 증가시키기 위해 `Math.max(n, servings)`를 사용하는데, 반복마다 `Math.max` 함수를 호출하는 비용이 추가될 수 있습니다. 이를 개선하기 위해 초기화 시에 `max` 값을 미리 계산하고 사용할 수 있습니다.

▼ 수정

```
package level_0.sharingPizaTwo;

public class Main {
  public static void  main(String[]  args)throws Exception {
    Solution solution = new Solution();
    System.out.println("결과 값" + solution.solution(4));
  }

  public static class Solution{
    // TODO 피자 나눠 먹기
    // INFO : 피자는 6 조각이며, 나눠먹는 사람의 수 n 일때 피자를 남김 없이 다 먹어야한다. 이때 총 피자의 판 개수 return
    public int solution(int n) {
      int servings = 6;
      if (servings % n == 0) return 1;

      int lcm = n;

      while (lcm % servings != 0) {
        lcm += n;
      }

      int answer = lcm / servings;
      return answer;
    }
  }
}
```

단점

1. 재적인 단점 중 하나는 `lcm`을 `n`만큼 증가시키면서 `n`과 `servings` 모두로 나누어 떨어질 때까지 LCM을 찾는 무차별 대입법(brute-force) 접근 방식을 사용합니다. 이 접근 방식은 `n`과 `servings`의 값이 클 경우 비효율적일 수 있습니다.

▼ 최종 수정

```
package level_0.sharingPizaTwo;

public class Main {
  public static void  main(String[]  args)throws Exception {
    Solution solution = new Solution();
    System.out.println("결과 값" + solution.solution(4));
    // System.out.println("결과 값" + solution.solutionTwo(4));
  }

  public static class Solution{
```

```
    // TODO 피자 나눠 먹기
    // INFO : 피자는 6 조각이며, 나눠먹는 사람의 수 n 일때 피자를 남김 없이 다 먹어야한다. 이때 총 피자의 판 개수 return
    public int solution(int n) {
      int servings = 6;
      if (servings % n == 0) return 1;

      int gcd = gcd(n, servings);
      int lcm = n * servings / gcd;

      int answer = lcm / servings;
      return answer;
    }

    public int gcd(int a, int b) {
      while (b != 0) {
          int temp = a % b;
          a = b;
          b = temp;
      }
      return a;
    }
  }
}
```

최대 공약수 gcd 를 구하기 위해 유클리드 호제법 사용하고, 최소 공배수 lcm 를 게산하는 데에도 최대 공약수를 활용 하여 효율적이고 정확하게 수정함

장점:

1. 최대 공약수를 구하는 `gcd` 함수를 별도로 분리하여 재사용성과 코드의 모듈화를 높였습니다.
2. 유클리드 호제법을 사용하여 최대 공약수를 효율적으로 계산하였습니다.
3. 최소 공배수를 계산할 때 최대 공약수를 활용하여 정확한 값을 구하였습니다.

단점:

1. 반복문을 사용하여 최대 공약수를 구하는 과정에서, 두 수의 크기에 따라 반복 횟수가 달라질 수 있습니다. 하지만 일반적인 상황에서는 효율적으로 동작합니다.

▼ 다른 사람 풀이

```
class Solution {
    public int GCD(int num1, int num2) {
        if (num1 % num2 == 0)
            return num2;
        return GCD(num2, num1 % num2);
    }

    public int LCM(int num1, int num2) {
        return num1 * num2 / GCD(num1, num2);
    }

    public int solution(int n) {
        return LCM(n, 6) / 6;
    }
}
```

```
class Solution {
    public int solution(int n) {
        int answer = 1;

        while(true){
            if(6 * answer % n==0) break;
            answer++;
        }
```

```
        return answer;
    }
}
```

```
class Solution {
    private int pizza_piece = 6;
    public int solution(int n) {
        int answer = 0;
        int pizzaCount = 1;
        while(true) {
            if ((pizzaCount * pizza_piece) % n == 0) {
                answer = pizzaCount;
                break;
            }
            pizzaCount++;
        }
        return answer;
    }
}
```

느낀점

1. 재귀 함수를 이용하여 최소공배수와 최대 공약수를 구하며 최종적으러 값을 구하는 방식을 활용한것을 확인 이 코드는 간결하고 가독성이 좋아 보인다.또한, 재귀 함수를 활용하면 재사용성과 유연성이 높아지는 장점을 가지고 있다. 하지만 `호출 스택을 사용하기 때문에 매우 큰 수에 대해서는 스택 오버플로우가 발생할수 있어 보인다.` 따라서 매우 큰 수에 대해서는 성능적인 측면에서 조정일 필요할 수 도 있다.
2. while 반복문과 코드가 간결하고 이해하기 쉽다. 반복문을 사용하여 최소 공배수를 찾기 때문에 일반적인 상황에서 효율적이다. 하지만 `무차별 대입법(brute-force) 접근 방식을 사용하여` answer를 1 씩 증가 시키면서 6* answer가 n으로 나누어 떨어질 때까지 반복합니다. 이 접 근 방식은 n의 값이 클 경우 비효율적일수 있다.
   코드 실행 시간은 `answer`의 증가에 비례하기 때문에, 최소 공배수를 찾는 데에 시간이 오래 걸릴 수 있습니다.