# 10. 순환

# 10.1 순환 함수(1)

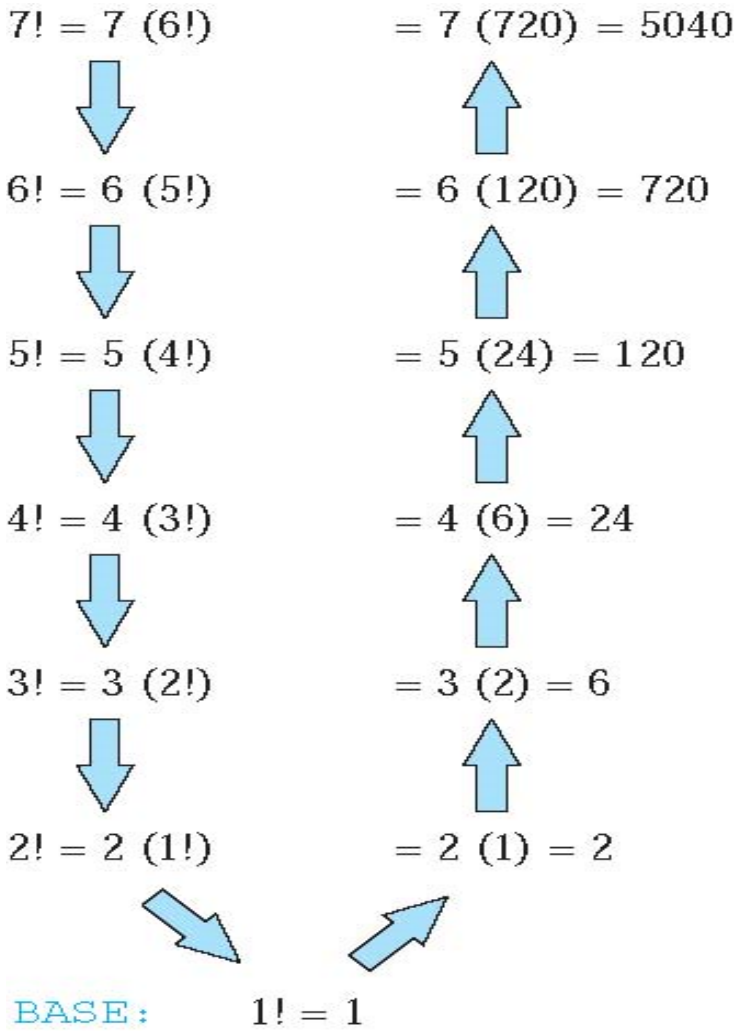
- 순환 함수(recursive function)
  - 자기 자신을 호출하는 함수
  - 가상의 루프를 사용하여 자동적으로 반복

- 예 : 계승(factorial) 함수

$$n! = \begin{cases} 1, \text{if } n = 0 \text{ or } 1 \\ n(n-1)!, \text{if } n > 1 \end{cases}$$

  - 한 숫자에 대한 계승을 이보다 작은 숫자에 대한 계승으로 정의하고 있음
  - 순환부는 나머지 수식이 비순환부에 의해 계산되는 경우를 만날 때까지 계속 적용됨
  - 이렇게 구해진 값은 최종적으로 원래 수식의 정확한 값을 리턴할 때까지 연쇄적으로 다른 값으로 계산됨

# 순환적으로 7!을 계산


7! = 7 (6!)
= 7 (720) = 5040
6! = 6 (5!)
= 6 (120) = 720
5! = 5 (4!)
= 5 (24) = 120
4! = 4 (3!)
= 4 (6) = 24
3! = 3 (2!)
= 3 (2) = 6
2! = 2 (1!)
= 2 (1) = 2
BASE:
1! = 1

# 순환 함수(2)

- 베이스(base)
  - 순환이 제대로 작동하기 위해서는 반드시 직접 계산에 의해 순환 사다리를 다시 올라갈 수 있도록 베이스를 가지고 있어야 함
  - 베이스를 명시하지 않으면 무한 순환에 빠질 위험이 있음

$$n! = 1 \text{ if } n = 0 \text{ or } 1 \text{[베이스]}$$

$$n! = n(n-1)! \text{[순환]}$$

# 순환 계승 함수

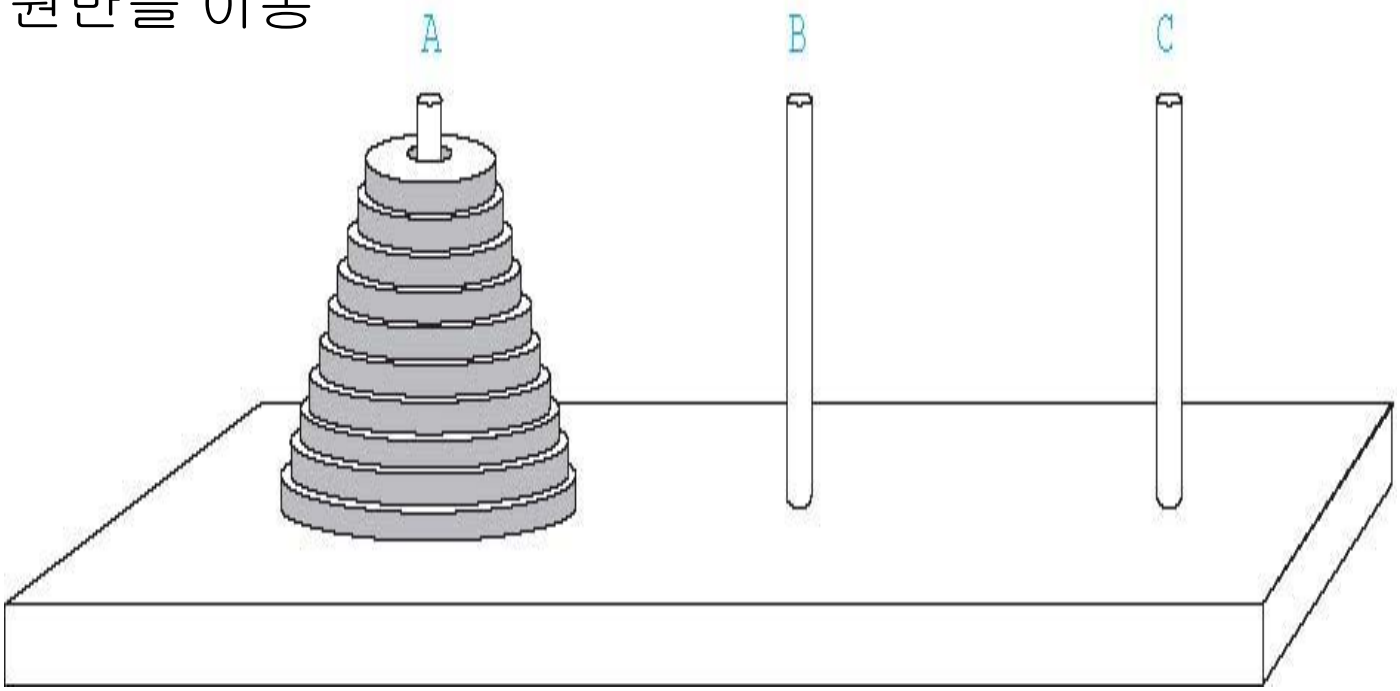
```
1  class TestRecursiveFactorial {
2    public static void main(String[] args) {
3      for (int i = 0; i < 10; i++)
4        System.out.println(i + "₩t" + factorial(i));
5    }
6
7    static long factorial(int n) {
8      if (n < 2) return 1; // base
9      return n*factorial(n-1); // recursion
10   }
11 }
```

반복 버젼

```
static long factorial(int n) {
    long m=1;
    for (int f=2; f<=n; f++)
        m *= f;
    return m;
}
```

# 10.2 하노이의 탑 게임

- **장대 A에 있는 원반을 장대 B로 이동시키는 게임 (C를 이용)**
  - 이동시 아래에 있는 원반이 위에 있는 원반보다 반드시 커야 함
  - 작은 원반의 위에 큰 원반을 쌓지 않으면서 한번에 하나씩 원반을 이동



# 하노이의 탑  해결 과정

- n개의 원반을 A에서 B로 이동하기 위해서는
  - 먼저 마지막 원반을 제외한 모든 n-1개의 원반을 A에서 C로 이동
  - 마지막 원반을 A에서 B로 이동
  - C에 있는 n-1개의 원반을 B로 이동

# 하노이의 탑

```
1    class HanoiTowers {
2      public static void main(String[] args) {
3        int numTowers = 3;
4        if (args.length>0) numTowers=Integer.parseInt(args[0]);
5        print(numTowers, 'A', 'B', 'C');
6      }
7
8      static void print(int n, char x, char y, char z) {
9        // move n disks from peg x to peg y using peg z:
10       if (n == 1) System.out.println(x + " --> " + y); // base
11       else {
12         print(n-1, x, z, y); // recursion
13         System.out.println(x + " --> " + y);
14         print(n-1, z, y, x); // recursion
15       }
16     }
17   }
```

# 10.3 피보나치 수

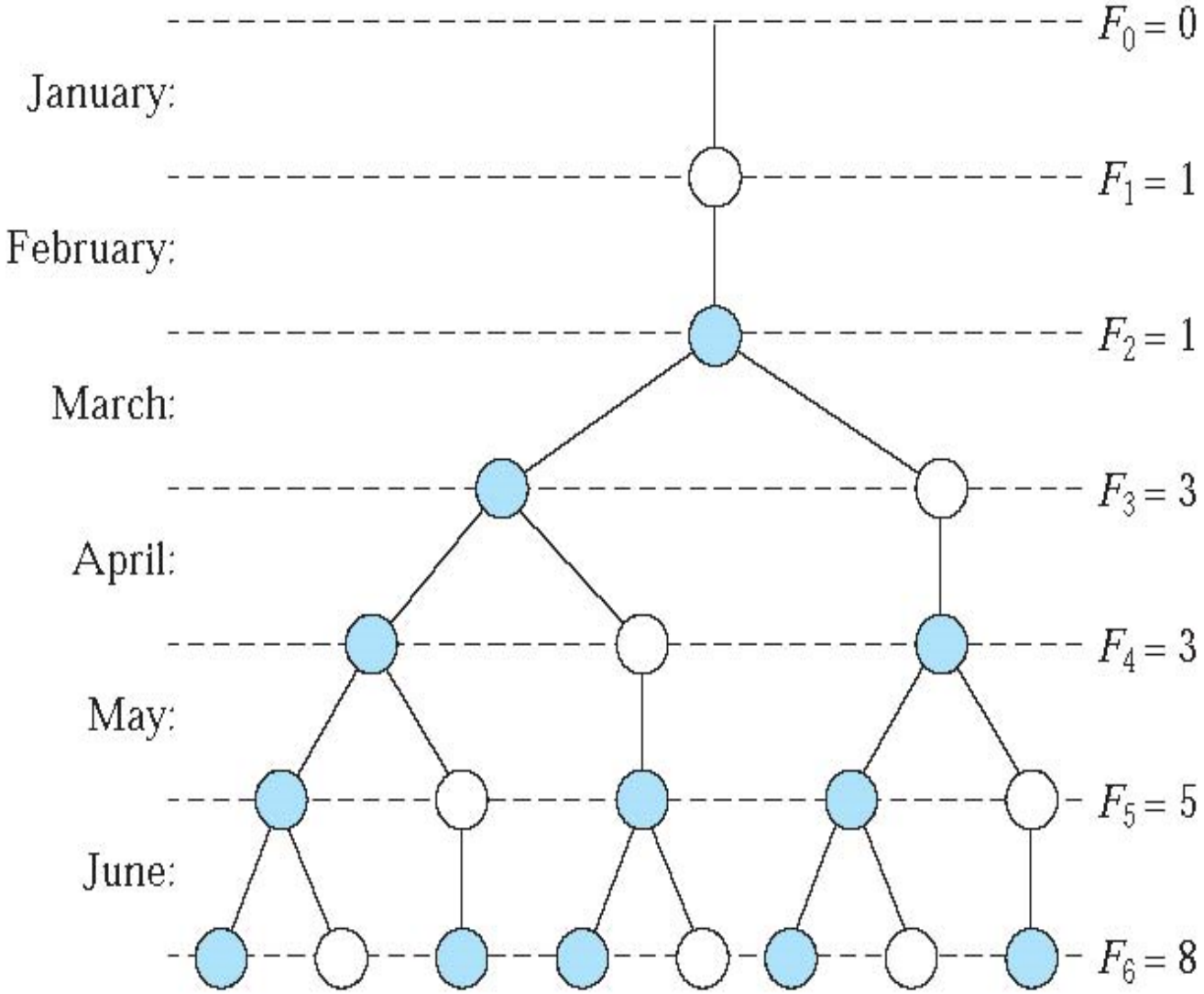
- 피보나치(Fibonacci) 수
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

$$F_n = \begin{cases} 0, \text{if } n = 0 \\ 1, \text{if } n = 1 \\ F_{n-1} + F_{n-2}, \text{if } n > 1 \end{cases}$$

# 순환 피보나치 함수

```
1   class TestRecursiveFibonacci {
2     public static void main(String[] args) {
3       for (int i = 0; i < 13; i++)
4       System.out.println(i + "₩t" + f(i));
5     }
6     static long f(int n) {
7       if (n < 1) return 0; // base
8       if (n < 3) return 1; // base
9       return f(n-1) + f(n-2); // recursion
10    }
11 }
```

# 피보나치의 토끼 문제


$F_0 = 0$
January:
$F_1 = 1$
February:
$F_2 = 1$
March:
$F_3 = 3$
April:
$F_4 = 3$
May:
$F_5 = 5$
June:
$F_6 = 8$

# 10.4 호출 트리

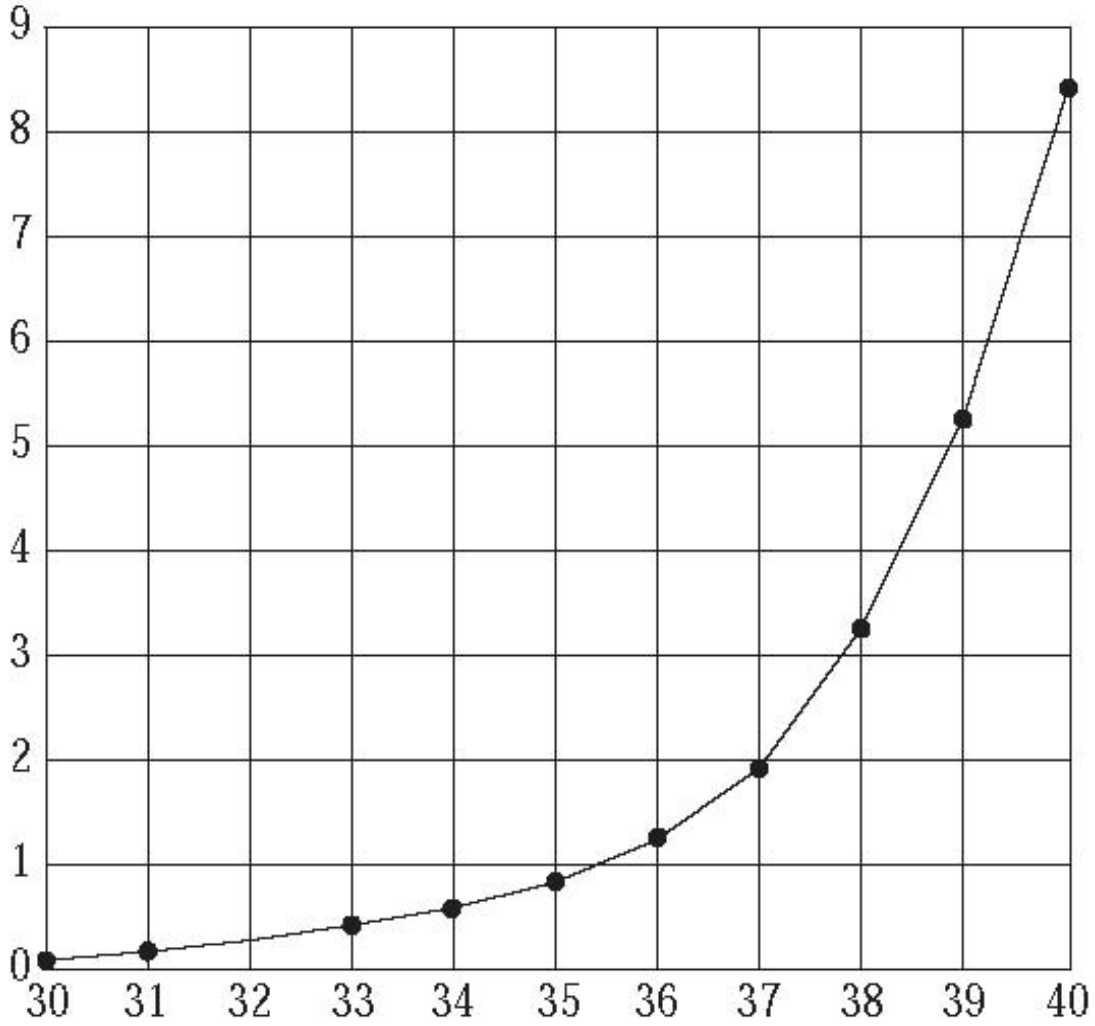
- 순환 피보나치 함수의 시간 측정

```
1   class TimeRecursiveFibonacci {
2     public static void main(String[] args) {
3       for (int n=30; n<=40; n++) {
4         long t0=System.currentTimeMillis();
5         long m=f(n);
6         long t1=System.currentTimeMillis();
7         System.out.println("f("+n+")="+m+"₩ttime: "+(t1-t0));
8       }
9     }
10
11    static long f(int n) // same as in Listing 10.3
12  }
```

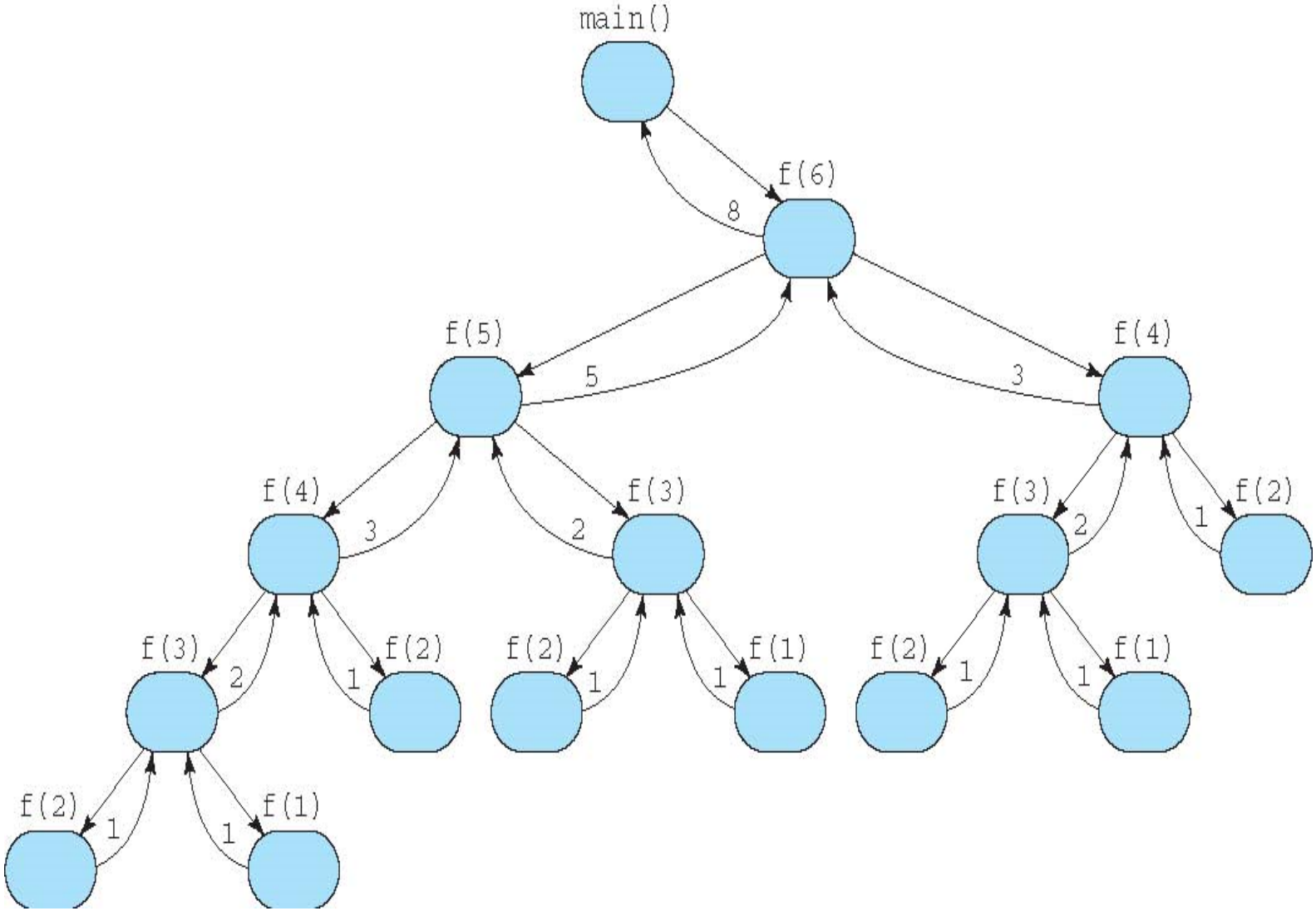
# 출력 결과

f(30) = 832040 time: 78
f(31) = 1346269 time: 125
f(32) = 2178309 time: 188
f(33) = 3524578 time: 297
f(34) = 5702887 time: 468
f(35) = 9227465 time: 782
f(36) = 14930352 time: 1265
f(37) = 24157817 time: 1953
f(38) = 39088169 time: 3219
f(39) = 63245986 time: 5250
f(40) = 102334155 time: 8531

# 실행 시간에 대한 그래프 : 지수 증가



# 순환 피보나치 메소드에 대한 호출 트리



# 실제 실행 과정

1. main() calls f(6)
2. f(6) calls f(5)
3. f(5) calls f(4)
4. f(4) calls f(3)
5. f(3) calls f(2)
6. f(2) returns 1 to f(3)
7. f(3) calls f(1)
8. f(1) returns 1 to f(3)
9. f(3) returns 1 + 1 = 2 to f(4)
10. f(4) calls f(2)
11. f(2) returns 1 to f(4)
12. f(4) returns 2 + 1 = 3 to f(5)
13. f(5) calls f(3)
14. f(3) calls f(2)
15. f(2) returns 1 to f(3)
16. **f(3) calls f(1)**
17. **f(1) returns 1 to f(3)**
18. **f(3) returns 1 + 1 = 2 to f(5)**
19. **f(5) returns 3 + 2 = 5 to f(6)**
20. **f(6) calls f(4)**
21. **f(4) calls f(3)**
22. **f(3) calls f(2)**
23. **f(2) returns 1 to f(3)**
24. **f(3) calls f(1)**
25. **f(1) returns 1 to f(3)**
26. **f(3) returns 1 + 1 = 2 to f(4)**
27. **f(4) calls f(2)**
28. **f(2) returns 1 to f(4)**
29. **f(4) returns 2 + 1 = 3 to f(6)**
30. **f(6) returns 5 + 3 = 8 to main()**

# 반복 피보나치 함수의 시간 측정

```
1    class TimeIterativeFibonacci {
2      public static void main(String[] args) {
3        for (int n = 30; n <= 40; n++) {
4          long t0=System.currentTimeMillis();
5          long m=f(n);
6          long t1=System.currentTimeMillis();
7          System.out.println("f(" + n + ") = " + m+"₩ttime: "+(t1-t0));
8        }
9      }
10     static long f(int n) {
11       if (n<2) return n;
12       long f0=0, f1=1, f2=1;
13       for (int i=2; i<n; i++) {
14         f0 = f1;
15         f1 = f2;
16         f2 = f1 + f0;
17       }
18       return f2;
19     }
20   }
```

# 출력

f(30) = 832040 time: 0
f(31) = 1346269 time: 0
f(32) = 2178309 time: 0
f(33) = 3524578 time: 0
f(34) = 5702887 time: 0
f(35) = 9227465 time: 0
f(36) = 14930352 time: 0
f(37) = 24157817 time: 0
f(38) = 39088169 time: 0
f(39) = 63245986 time: 0
f(40) = 102334155 time: 0

# 10.5 재계산 대신 저장

- 반복 버전과 순환 버전
  - 일반적으로 반복 버전은 빠르고
  - 순환 버전은 간단함

- 순환 버전이 느린 이유
  - 같은 값을 여러 번 재계산하기 때문
  - 처음 계산될 때 계산 결과를 저장해 놓으면 재계산을 피할 수 있음

# 피보나치 수를 계산하는데 임시 저장소를 사용

```
1    class TimeStoredFibonacci {
2      public static void main(String[] args) {
3        for (int n = 30; n <= 40; n++) {
4          long t0 = System.currentTimeMillis();
5          long m = Fibonacci.number(n);
6          long t1 = System.currentTimeMillis();
7          System.out.println("f(" + n + ") = " + m+"₩ttime: "+(t1-t0));
         }    }    }
12   class Fibonacci {
13     private static long[] fib = new long[100];
14     private static int lastFibIndex = 2;
16     static { // class initializer
17        fib[1] = fib[2] = 1;
18     }
20     public static long number(int n) {
21       for (int i = lastFibIndex+1; i <= n; i++)
22          fib[i] = fib[i-1] + fib[i-2];
23       if (n > lastFibIndex) lastFibIndex = n;
24       return fib[n];
25     }
26   }
```

# 출력 결과

```
f(30) = 832040      time: 31
f(31) = 1346269     time: 0
f(32) = 2178309     time: 0
f(33) = 3524578     time: 0
f(34) = 5702887     time: 0
f(35) = 9227465     time: 0
f(36) = 14930352    time: 0
f(37) = 24157817    time: 0
f(38) = 39088169    time: 0
f(39) = 63245986    time: 0
f(40) = 102334155   time: 0
```

# 10.7 순환 이진 탐색 알고리즘

- 순환 이진 탐색 알고리즘
  1. 만일 $q \le p$ 이면, $-p-1$(베이스) 리턴
  2. $i=(p+q)/2$ 이라 함(정수 나눗셈)
  3. 만일 $a_i=x$ 이면, $i$ 리턴
  4. 만일 $a_i < x$ 이면, 상위 시이퀀스 $\{a_{i+1}, a_{i+2}, \ldots, a_{q-1}\}$을 탐색한 결과를 리턴
  5. 만일 $a_i > x$ 이면, 하위 시이퀀스 $\{a_p, a_{p+1}, \ldots, a_{i-1}\}$을 탐색한 결과를 리턴

- 순환 이진 탐색 알고리즘은 **$\Theta(\lg n)$** 시간에 실행됨