23. Memoization (메모이제이션)

- 컴퓨터 프로그램이 동일한 계산을 반복해야 할 때, 이전에 계산한 값을 메모리에 저장함으로써 동일한 계산의 반복 수행을 제거하여 프로그램 실행 속도를 빠르게 하는 기술. (caching 이라고도 부른다.)
- Memoization 이 필요한 가장 좋은 예는 피보나치 수열의 점화식(漸化式, Recurrence relation) 이다.

  1, 1, 2, 3, 5, 8, 13, 21, 34 .....

$$F(n) := \begin{cases} 0 & \text{if n} = 0; \\ 1 & \text{if n} = 1; \\ F(n-1) + F(n-2) & \text{if n} > 1. \end{cases}$$

In [1]:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

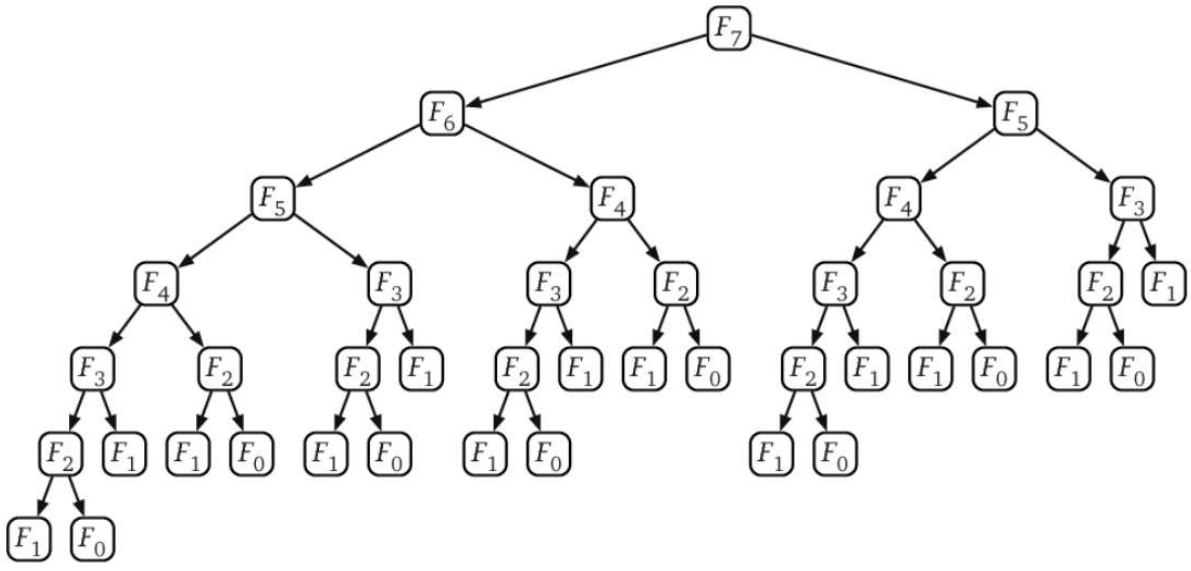
In [2]:

```
fib(9)
```

Out[2]:

34

fib(64)는 재귀 호출을 18,446,744,073,709,551,615 ($2^{64}-1$) 회 실행해야 한다. 이 알고리듬은 매우 간결하고 깔끔한 코드로 보이지만, 불필요한 중복 계산을 너무 많이 한다. 예를 들어 64번째 항을 구하기 위해서는 63번째와 62번째 항을 더해야 하므로 결국 62번째는 2번 계산된다. 즉 1에 가까워질수록 중복 계산횟수가 기하급수적으로 늘어나는 것이다.



In [3]:

```
1 fib(32)
```

Out[3]:

2178309

memoization 기법

다음 코드는 불필요한 계산을 피하기 위해서 한 번 계산한 결과를 사전에 저장했다가, 계산 결과가 있으면 바로 꺼내서 보여주는 식이다. 이러한 성능 개선 방식을 캐싱 또는 메모이제이션이라 한다.

In [17]:

```
1  memo = dict()
2
3  def fib2(n):
4      if n in memo:
5          return memo[n]
6
7      if n in (0, 1):
8          memo[n] = n
9          return n
10
11     result = fib2(n - 1) + fib2(n - 2)
12
13     memo[n] = result
14
15     return result
```

In [18]:

```
1 print(fib2(64))
```

10610209857723

한 번 계산한 결과를 재계산 할 필요가 없고, 그것이 재귀 호출을 막는 방식이므로 64번만 계산하면 된다.

In [21]:

```
1 %timeit fib(32)
```

1.05 s ± 60 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [20]:

```
1 %timeit fib2(32)
```

173 ns ± 15.7 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)

연습문제

이항계수를 구하는 공식을 함수로 구현한다. 함수 호출 전, 후의 time check 하여 수행시간 비교

- 이항계수 : 주어진 크기의 (순서 없는) 조합의 가짓수

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n-1}{k-1} + \binom{n-1}{k}$$, if n = k or k = 0 → 1

1) 단순 재귀 표현으로 구현

2) memoization 으로 구현

Dynamic Programming

- 어려운 문제를 여러개의 하위 문제로 쪼개고, 이 하위 문제 (subproblem) 들을 먼저 푸는 방법
- 먼저 푼 subproblem 의 결과값을 table 에 저장하여 상위 문제에서 재사용

- (예제) matrix 의 최단 경로 계산 문제
  - 정수들이 저장된 nXn matrix 의 좌상단에서 우하단까지 이동한다. 단, 오른쪽이나 아래쪽으로만 이동할 수 있다.
  - 방문한 cell 에 있는 정수값들의 합이 최소화되는 경로를 구한다.
    - L[i, j] : (0,0) 에서 (i, j) 까지 이르는 최소 합
  - 점화식 도출을 위한 하위문제 도출:
    - (i, j) 에 도달하기 위해서는 반드시 (i-1, j) 혹은 (i, j-1) 을 거쳐야만 한다.
    - i = 0, j = 0 는 L[0, 0]
    - i = 0 이면 L[0, j-1] + m_ij
    - j = 0 이면 L[i-1, 0] + m_ij

matrix

[6, 7, 12, 5]
[5, 3, 11, 18]
[7, 17, 3, 3]
[8, 10, 14, 9]

minimum path

[6, 13, 25, 30]
[11, 14, 25, 43]
[18, 31, 28, 31]
[26, 36, 42, 40]

In [11]:

```
import random
mat = [[6, 7, 12, 5], [5, 3, 11, 18], [7, 17, 3, 3], [8, 10, 14, 9]]
#mat = [[random.choice(range(10)) for _  in range(18)] for i in range(18)]
m = len(mat)
for i in range(m):
    print(mat[i])
```

```
[6, 7, 12, 5]
[5, 3, 11, 18]
[7, 17, 3, 3]
[8, 10, 14, 9]
```

단순 재귀적 방법

In [12]:

```
def matrixPath(i, j):
    if i == 0 and j == 0:
        return mat[i][j]
    if j == 0:
        return matrixPath(i-1, 0) + mat[i][0]
    if i == 0:
        return matrixPath(0, j-1) + mat[0][j]
    return min(matrixPath(i-1, j), matrixPath(i, j-1)) + mat[i][j]

%timeit matrixPath(m-1, m-1)
```

20.9 µs ± 1.57 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)

Dynamic Programming (Memoization)

In [13]:

```
memo = [[] for _ in range(m)]
for i in range(m):
    for j in range(len(mat[i])):
        memo[i].append(-1)

def matrixPath_memo(i, j):
    if memo[i][j] != -1:
        return memo[i][j]

    if i == 0 and j == 0:
        memo[i][j] = mat[i][j]
    elif j == 0:
        memo[i][0] = matrixPath_memo(i-1, 0) + mat[i][0]
    elif i == 0:
        memo[0][j] = matrixPath_memo(0, j-1) + mat[0][j]
    else:
        memo[i][j] = min(matrixPath_memo(i-1, j), matrixPath_memo(i, j-1)) + mat[i][j]
    return memo[i][j]
```

In [14]:

```
%timeit matrixPath_memo(m-1, m-1)
```

266 ns ± 46.6 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

In [15]:

```
for i in range(m):
    print(memo[i])
```

```
[6, 13, 25, 30]
[11, 14, 25, 43]
[18, 31, 28, 31]
[26, 36, 42, 40]
```

In [ ]:

```
1
```