동적계획법

Dynamic Programming

앞서 공부한 합병정렬, 퀵정렬, 이분검색과 같은 문제는 풀이 대상을 적절히 분할하여 규모를 축소함으로써 문제를 효율적으로 풀 수 있었다. 이러한 문제풀이 기법을 (하향식) 분할정복divide-and-conquer법이라고 한다. 그런데 분할정복법이 잘 통하지 않는 문제도 있다. 이 장에서는 분할정복법은 잘 통하지 않지만 (상향식) 동적계획법이라는 기법을 적용하면 효율적으로 풀리는 문제인 피보나치 수열과 조합계산 문제 사례를 차례로 공부한다.

1. 피보나치 수열

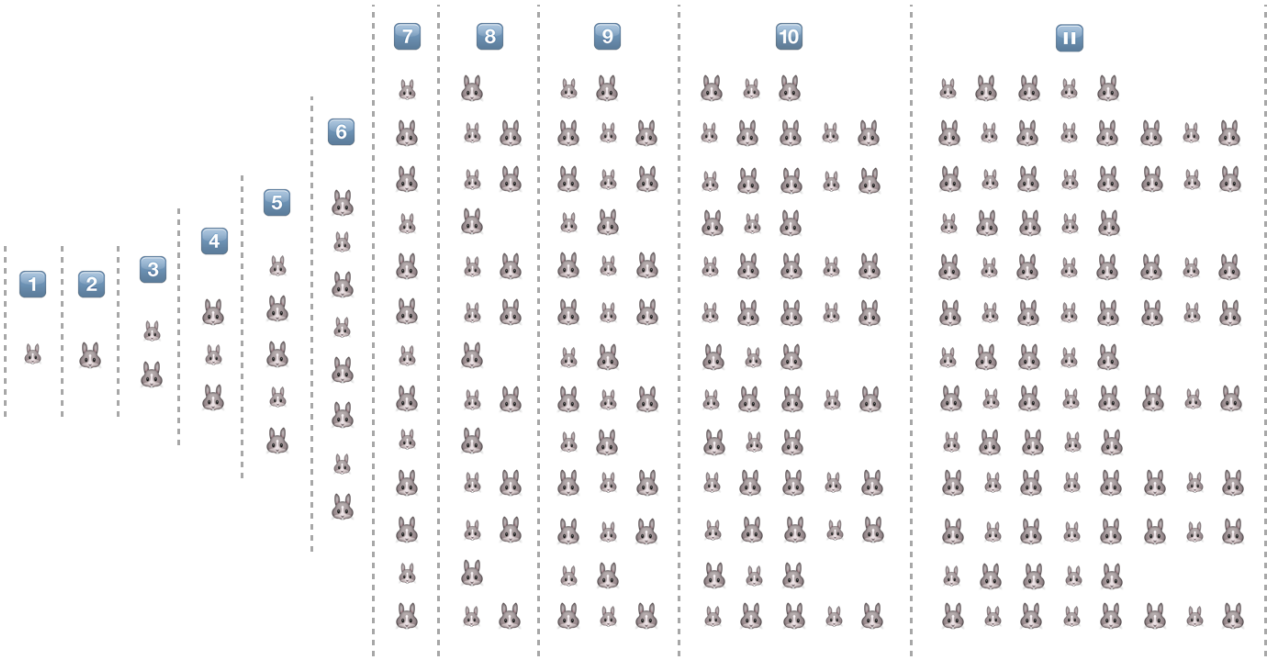
피보나치 수

1202년 이탈리아의 수학자 레오나르도 피보나치(Leonardo Fibonacci, 1170년~1250년)가 제시한 재미있는 수열을 하나 공부해보자. 달나라(!)에서 토끼가 다음과 같은 규칙으로 번식한다.

- 어른 토끼 1쌍은 매달 아기 토끼 1쌍을 낳는다.
- 아기 토끼는 태어난지 1달이 지나면 어른 토끼가 되어 번식 가능해진다.
- 토끼는 죽지 않는다.

1월에 태어난 아기 토끼 1쌍이 12월에 토끼 몇 쌍이 되어 있을까?

아기 토끼 1쌍으로 출발한 토끼 가족의 규모는 매달 아래 그림과 같이 불어난다. 이 그림에서 작은 🐰는 갓 태어난 아기 토끼 1쌍을 나타내고, 큰 🐰는 번식 가능한 어른 토끼 1쌍을 나타낸다. (12월은 생략)



토끼 가족의 개체수 역사를 월별로 따져보면 다음 표와 같다. (단위: 쌍)

월	0	1	2	3	4	5	6	7	8	9	10	11	12
아기 토끼	0	1	0	1	1	2	3	5	8	13	21	34	55
어른 토끼	0	0	1	1	2	3	5	8	13	21	34	55	89
총	0	1	1	2	3	5	8	13	21	34	55	89	144

1월에 갓 태어난 아기 토끼 1쌍이 시조가 되었다. 2월이 되면 번식이 가능한 어른 토끼 1쌍이 된다. 3월에는 아기 토끼 1쌍을 낳아 토끼 가족이 2쌍이 된다. 4월에는 어른 1쌍이 아기 1쌍을 낳고 아기 1쌍이 어른이 되어 가족이 총 3쌍이 된다. 5월에는 어른 2쌍이 아기 2쌍을 낳고 아기 1쌍이 어른이 되어, 가족이 총 5쌍이 된다. 6월에는 어른 3쌍이 아기 3쌍을 낳고 아기 2쌍이 어른이 되어 가족이 총 8쌍이 된다. 반년이 지나는 동안 토끼 가족의 번식률은 그리 높아 보이지 않는다. 그런데 이후에는 다르다. 번식률이 점점 가파르게 올라가 연말에는 토끼 가족이 144쌍이 된다. 10년 후에는 몇 쌍이 될까? 함수를 만들어 계산해보자.

피보나치 수의 재귀 해법 (하향식)

위 표의 맨 아래 행을 보면 토끼 개체수(쌍)는 다음과 같이 증가함을 알 수 있다.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …

이를 피보나치 수열Fibonacci sequence이라고 하는데, 이 수열을 잘 관찰해보면 이전 두 개의 수를 더하여 다음 수를 정하는 수열임을 알 수 있다. n 째 피보나치 수는 자연수의 귀납구조를 이용하여 다음과 같이 재귀로 정의할 수 있다.

```
fib(n) = fib(n-1) + fib(n-2), n > 1
fib(1) = 1
fib(0) = 0
```

즉, k 째 피보나치 수는 k-1 째와 k-2 째 피보나치 수를 더해서 구한다. 이 재귀 정의를 Python 함수로 작성하면 다음과 같다. (음수 인수는 없다고 가정)

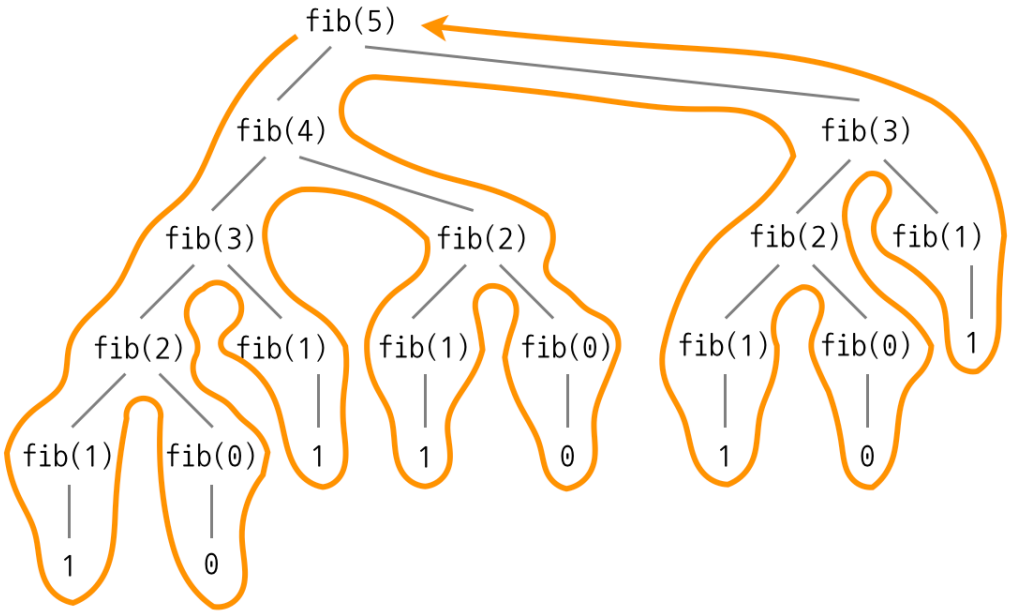
```
1 def fib(n):
2     if n > 1:
3         return fib(n - 1) + fib(n - 2)
4     else:
5         return n
```

fib(5) 호출을 실행추적해보자.

```
fib(5)
=> fib(4) + fib(3)
=> (fib(3) + fib(2)) + fib(3)
=> ((fib(2) + fib(1)) + fib(2)) + fib(3)
=> (((fib(1) + fib(0)) + fib(1)) + fib(2)) + fib(3)
=> (((1 + fib(0)) + fib(1)) + fib(2)) + fib(3)
=> (((1 + 0) + fib(1)) + fib(2)) + fib(3)
=> ((1 + fib(1)) + fib(2)) + fib(3)
```

```
=> ((1 + 1) + fib(2)) + fib(3)
=> (2 + fib(2)) + fib(3)
=> (2 + (fib(1) + fib(0))) + fib(3)
=> (2 + (1 + fib(0))) + fib(3)
=> (2 + (1 + 0)) + fib(3)
=> (2 + 1) + fib(3)
=> 3 + fib(3)
=> 3 + (fib(2) + fib(1))
=> 3 + ((fib(1) + fib(0)) + fib(1))
=> 3 + ((1 + fib(0)) + fib(1))
=> 3 + ((1 + 0) + fib(1))
=> 3 + (1 + fib(1))
=> 3 + (1 + 1)
=> 3 + 2
=> 5
```

이 함수는 몸체에서 재귀 호출을 두 번 한다. fib(5) 호출의 실행추적에서 호출관계를 그림으로 그려보면 아래와 같다. 이 그림의 상하를 뒤집어 보면 모양이 나무와 비슷하다. 그래서 컴퓨터과학에서 이런 모양의 데이터 구조를 트리tree 구조라고 한다. 호출의 실행추적 순서를 관찰해보면 아래 그림의 화살표와 같은 순서로 재귀호출이 진행되었음을 알 수 있다. (이를 깊이우선 나무가지 훑기depth-first tree traversal라고 하는데 나중에 상위 과목에서 자세히 다룰 것이다.)



이제 실행기에서 fib(12), fib(24), fib(30), fib(36), fib(42), fib(48)을 차례로 실행해보자. 다음과 같이 계산 결과가 나온다.

```
>>> fib(12)
144
>>> fib(24)
46368
>>> fib(30)
```

```
832040
>>> fib(36)
14930352
>>> fib(42)
267914296
>>> fib(48)
4807526976
```

(헉! 4년만에 토끼 개체수가 48억 쌍!) 여기서 각 호출 결과를 얻는데 걸린 시간을 관찰해보면 인수의 크기에 따라서 차이를 확연하게 느낄 수 있다[1]. 첫 두 호출 `fib(12)`과 `fib(24)`의 계산 결과는 눈 깜빡할 사이에 나온다. `fib(30)`은 즉시는 아니지만 1초도 걸리지 않아 바로 답이 나온다. `fib(36)`은 6초 정도 기다리니 답이 나온다. 그런데 `fib(42)`는 좀 기다려도 반응이 없다. 100초 정도 기다리면 나온다. 인수의 크기는 별 차이 없는데 계산시간의 차이는 꽤 커졌다. `fib(48)`은 얼마나 더 걸릴까? 아메리카노 한 잔 마시고 나면 나올까? 한잠 자고 나면 나올까? 인내를 갖고 참고 기다릴 수 있는 시간 안에 답이 나오지 않는다. 앞에서 공부한 표를 채우는 방식으로 종이에 손으로 계산해도 이보단 더 빨리 계산할 수 있을 것 같다. 컴퓨터가 나보다 느리게 계산한다? 왜 이런 현상이 발생할까?

계산 복잡도

호출 횟수를 따져보자. `fib(n)`을 호출하면 재귀호출을 2번하고, 그 호출은 다시 각각 재귀호출을 2번씩 하여 총 2^2번 호출하고, 각 호출이 다시 재귀호출을 2번씩 하여 총 2^3번 호출하고, 각 호출이 다시 재귀호출을 2번씩 하여 총 2^4번 호출하고, 이런 식으로 2배씩 호출 수가 계속 증가하면, 48째 재귀호출의 경우 총 호출 횟수가 2^{48}번이 되어 280조가 넘게 된다. 즉, `n`이 증가함에 따라 호출횟수가 2^n에 비례하여 기하급수적으로 증가한다.

문제는 위 재귀 함수가 계산을 수행하면서 동일한 재귀호출을 중복 호출한다는 것이다. 이 중에 얼마나 많은 호출이 중복일까? 중복도 `n`이 증가함에 따라 기하급수적으로 증가한다. 중복계산으로 시간을 엄청나게 많이 낭비하고 있는 것이다. 이러니 나의 최신 고성능 컴퓨터도 답을 구하는데 시간이 걸릴 수 밖에 없다.

결론적으로 피보나치 수열 계산하는 문제는 하향식top-down 풀이 방식이 실용적 해법이 아니다. 그런데 다행히도 이 문제는 매우 효율적인 상향식bottom-up 해법이 있다. 이를 전통적으로 **동적계획법** dynamic programming 이라고 하는데 어떤 풀이 방식인지 탐구해보자.

[1] 사용하는 컴퓨터 하드웨어의 성능에 따라서 계산 시간은 다소 차이가 있을 수 있다. 여기서 언급한 시간은 iMac 3.5GHz Inter Core 컴퓨터에서 Python 3.4.3 실행기로 잰 것이다.

동적계획법

상향식으로 문제를 풀면 중복계산을 피할 수 있다. 즉, fib(2)를 먼저 계산하고, 다음에 fib(3)을 계산하고, 다음에 fib(4)를 계산하고, 이런 식으로 계속 계산해나가면 된다. 즉, 아래 표를 왼쪽 끝에서 시작하여 오른쪽으로 진행하면서 아기 토끼와 어른 토끼의 수를 계산하여 채워나가면 된다.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
아기 토끼 bunny	0	1	0	1	1	2	3	5	8	13	21	34	55
어른 토끼 rabby	0	0	1	1	2	3	5	8	13	21	34	55	89
fib(i)	0	1	1	2	3	5	8	13	21	34	55	89	144

이 표를 어떻게 채워나갈지 구체적으로 기술해보자. 토끼의 번식 규칙을 관찰해보면 다음과 같은 특징을 발견할 수 있다.

- 어른 토끼 1쌍은 아기 토끼를 한달에 1쌍씩 낳으니까, 이달의 아기 토끼 쌍의 수는 지난달의 어른 토끼 쌍의 수와 같다.
- 아기 토끼는 한달이 지나면 어른 토끼가 되므로, 이달의 어른 토끼 쌍의 수는 지난달의 어른 토끼 쌍의 수에 지난달의 아기 토끼 쌍의 수를 더한 수와 같다.

이 특징을 이용하면 바로 지난달의 아기 토끼 쌍의 수와 어른 토끼 쌍의 수를 가지고 이달의 토끼 쌍의 수를 알 수 있다. 이를 식으로 나타내면 다음과 같다.

이달을 i 째 달이라고 하고, 지난달을 i-1 째 달이라고 하고

i 째 달의 전체 토끼 쌍의 수는 fib_i이고

i 째 달의 아기 토끼 쌍의 수는 $bunny_i$이고

i 째 달의 어른 토끼 쌍의 수는 $rabby_i$이면,

i 째 의 어른 토끼와 아기 토끼의 쌍의 수를 각각 다음 식으로 구할 수 있다.

$$bunny_i = rabby_{i-1}$$

$$rabby_i = rabby_{i-1} + bunny_{i-1}$$

전체 토끼 쌍의 수는 아기 토끼와 어른 토끼 쌍의 수를 더하면 되므로, i 째 달의 전체 토끼 쌍의 수는 다음 식과 같다.

$$fib_i = bunny_i + rabby_i$$

즉, 위 표의 8~9 열에 화살표로 표시한대로 $bunny_9$은 $rabby_8$과 같고, $rabby_9$은 $bunny_8 + rabby_8$ 이 된다.

이 점에 착안하여 위 표를 왼쪽부터 채우는 반복문을 작성해보자. i = 1, bunny = 1, rabby = 0 에서 시작하여, 위에서 세운 식을 반복 적용하여 새로운 bunny와 rabby의 값을 구하고, 매번 i의 값을 1씩 증가시킨다. 즉, 세 변수는 다음과 같이 변화해야 한다.

i = 1, bunny = 1, rabby = 0

i = 2, bunny = 0, rabby = 1

i = 3, bunny = 1, rabby = 1

i = 4, bunny = 1, rabby = 2

i = 5, bunny = 2, rabby = 3

i = 6, bunny = 3, rabby = 5

i = 7, bunny = 5, rabby = 8

...

fib(n)을 구한다면, i == n 일때의 bunny + rabby 값이 답이 된다. 따라서 i < n 이 반복 조건이고, i == n 이 종료조건이다. 완성한 코드는 다음과 같다.

```
1  def fib(n):
2      i = 1
3      bunny, rabby = 1, 0
4      while i < n:
5          i += 1
6          bunny, rabby = rabby, bunny + rabby
7      return bunny + rabby
```

계산 복잡도

- 계산 시간 : fib(n)을 계산하면서 while 문의 몸체는 정확하게 n-1번 실행된다. 계산 시간은 입력 값 n에 정비례한다.

실행기에서 fib(12), fib(24), fib(30), fib(36), fib(42), fib(48)을 각각 실행해보자. 하향식 방식과는 달리 결과가 모두 눈 깜빡할 사이에 나온다.

잠깐!

바로 위 코드의 6번 줄에서 두 변수의 새 값을 동시에 지정하였다. 만일 다음의 6~7번 줄과 같이 순차적으로 지정을 하도록 하면 어떨까?

```
1  def fib(n):
2      i = 1
3      bunny, rabby = 1, 0
4      while i < n:
5          i += 1
6          bunny = rabby
7          rabby = bunny + rabby
8      return bunny + rabby
```

그런데 이 경우 동시지정과 달리, 틀린 계산 결과가 나온다. 줄 7의 새 rabby 값을 계산하는데 변경된 bunny 값을 사용하기 때문이다. 줄 6과 7를 바꾸어도 문제가 해결되지 않는다. 이번엔 새 bunny 값을 계산하는데 변경된 rabby 값을 사용하게 되어 역시 틀린 계산 결과가 나온다. 따라서 변경전 값을 사용하려면, 그 전 값을 다음과 같이 temp 변수에 따로 보관해두고 가져다 쓸 수 밖에 없다.

```
1 def fib(n):
2     i = 1
3     bunny, rabby = 1, 0
4     while i < n:
5         i += 1
6         temp = bunny
7         bunny = rabby
8         rabby = temp + rabby
9     return bunny + rabby
```

위의 동시지정 버전과 비교해보면 코드 이해하기 더 힘들다. 그래서 Python과 같은 현대 프로그래밍언어는 대부분 동시지정을 허용한다. 작성하기도 간편할 뿐 아니라 프로그램의 가독성도 높아지기 때문이다. 하지만 그렇지 않은 언어도 있으니 임시변수를 사용하는 기술도 익혀두어야 한다.

for 문 버전

피보나치 수와 같이 반복 횟수가 고정되어 있는 경우, for 문과 **정수범위**range를 쓰면 더 쉽고 간단하게 프로그램을 짤 수 있다. 정수범위 순서열을 사용하여 for 문으로 작성한 피보나치 수 함수는 다음과 같다.

```
1 def fib(n):
2     bunny, rabby = 1, 0
3     for i in range(2, n + 1):
4         bunny, rabby = rabby, bunny + rabby
5     return bunny + rabby
```

range(2,n+1)는 2부터 n까지의 정수 순서열을 나타내므로 for 문의 몸체는 정확히 n-1번 반복한다.

그리고 변수 i는 for 문 몸체에서 쓰지 않는다. 쓰지 않을 값을 구태여 이름지을 필요가 없으므로 다음과 같이 와일드카드(밑줄)를 대신 쓰는게 좋다.

```
1 def fib(n):
2     bunny, rabby = 1, 0
3     for _ in range(2, n + 1):
4         bunny, rabby = rabby, bunny + rabby
5     return bunny + rabby
```

피보나치 수열

지금까지는 n번째 피보나치 수를 찾는데 초점이 맞추어져 있어서 지난달 아기 토끼와 지난달 어른 토끼의 수만 기억하고 나머지는 잊어버려도 이달 토끼의 수를 구하는데 문제가 없었다. 이번에는 n까지 피보나치 수열 전체를 리스트로 내주는 함수 fibseq을 만들어보자. 위의 표를 잘 관찰해보면, 이달의 아기 토끼의 수는 지지난달 전체 토끼 수와 같고, 이달의 어른 토끼의 수는 지난달 전체 토끼의 수와 같다. (사실, 표를 잘 관찰해보면 세 행을 모두 유지할 필요가 없다. 각 행이 한 열씩 밀려 있을 뿐, 행은 모두 동일한 패턴을 보인다.) 따라서 이달의 전체 토끼수는 지난달 전체 토끼 수와 지지난달 전체 토끼 수를 더해

서 구할 수 있다. 초기 수열을 리스트 [0,1]로 놓고 시작하여, 끝의 두 수를 더해서 뒤에 하나씩 계속 붙여 나가는 식으로 함수를 작성하면 다음과 같다.

```
1  def fibseq(n):
2      fibs = [0, 1]
3      for k in range(2, n + 1):
4          fibs.append(fibs[k - 1] + fibs[k - 2])
5      return fibs
```

fibseq(10) 호출을 하면 어떤 식으로 계산을 수행하는지 실행의미를 추적해보자.

이제 fib(n)은 fibseq(n)를 활용하고 pop() 메소드를 써서 다음과 같이 작성하면 된다.

```
1  def fib(n):
2      return fibseq(n).pop()
```

표준라이브러리의 리스트 메소드인 pop()의 의미는 다음과 같다.

연산	의미	비고
s.pop()	s의 마지막 원소를 내주고, 그 원소를 s에서 제거한다.	s가 빈리스트이면 실행오류 IndexErrorr가 발생한다.
s.pop(i)	s의 위치번호 i에 위치한 원소를 내주고, 그 원소를 s에서 제거한다.	i가 s의 위치번호 범위 바깥인 경우 실행오류 IndexError가 발생한다.

2. 파스칼 삼각형

조합계산

수학에서 조합combination은 n개에서 순서에 상관없이 r개를 뽑는 가지수를 말한다. 이를 계산하는 공식은 다음과 같이 재귀로 정의한다.

$$ {}_nC_r = {}_{n-1}C_{r-1} + {}_{n-1}C_r,\ r \neq 0 \ \text{ and } \ r \neq n \qquad {}_nC_0 = 1 \qquad {}_nC_n = 1 $$

이 공식을 이용하면 조합을 계산하는 함수는 다음과 같이 Python 재귀함수로 작성할 수 있다.

```
1  def comb(n, r):
2      if not (r == 0 or r == n):
3          return comb(n - 1, r - 1) + comb(n - 1, r)
4      else:
5          return 1
```

이 함수를 호출하여 실행해보자.

- comb(30, 3)과 comb(30, 27)을 실행하면 바로 답이 나온다.
- comb(30, 7)과 comb(30, 23)을 실행하면 시간이 조금 걸린다. 그런데 참을만 하다.
- comb(30, 10)과 comb(30, 20)을 실행하면 시간이 훨씬 더 많이 걸린다. 언제 답이 나오나 궁금해하면서 인내를 가지고 기다려면 결국 나타난다.
- comb(30, 15)를 실행해보자. 대단한 인내심을 요구한다. 언젠가 답을 내주긴 하니 잠시 커피 한 잔 마시며 기다려보자.

- 왜 인수의 값에 따라서 계산 시간이 천차만별 일까?
- `comb(100, 50)`을 실행해보자. 얼마나 기다려야 할까? 잠자리 들기 전에 실행시켜놓고 아침에 확인하면 답이 나와 있을까?

인수의 값에 따라서 실행하는데 걸리는 시간이 천차만별이고 기다릴 수 없을 만큼 오래 걸리는 경우도 있으므로 이 함수는 실용적으로 사용할 수 없다.

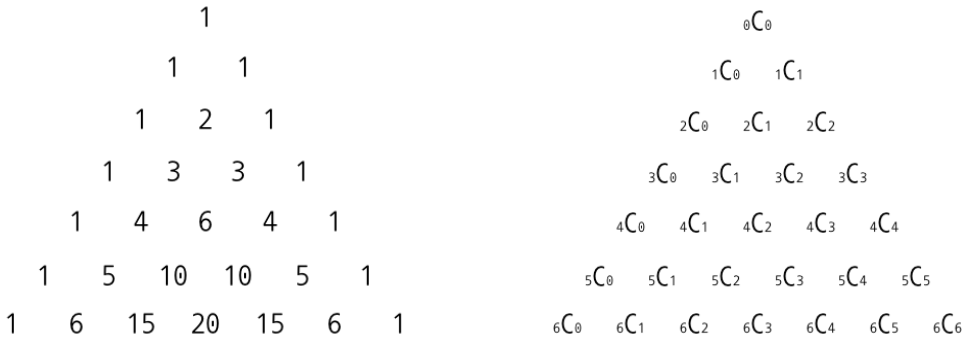
계산 복잡도

이 재귀 함수도 피보나치 재귀함수와 마찬가지로 하향식으로 문제를 풀면 재귀 호출 횟수가 지수에 비례하여 증가하고, 동시에 동일한 재귀호출을 엄청나게 많이 중복호출하여 시간과 공간을 사용하기 때문에 아무리 고성능 컴퓨터를 사용해도 한계가 있다. 그런데 다행히 이 문제도 상향식 동적계획 풀이법이 있다. 프랑스 수학자 블레즈 파스칼(Blaise Pascal, 1623~1662)이 고안해낸 삼각형을 그리면 이 문제의 해답을 빨리 (효율적으로) 구할 수 있다.

파스칼 삼각형

아래 왼쪽 그림은 파스칼 삼각형의 윗 꼭지 부분을 보여준 것이다. 이 삼각형의 양쪽 빗면은 모두 1이다. 그리고 내부의 수는 바로 위의 두 수를 더하여 결정한다. 위에서부터 시작하여 아래 방향으로 이런 식으로 차례로 수를 채울 수 있다.

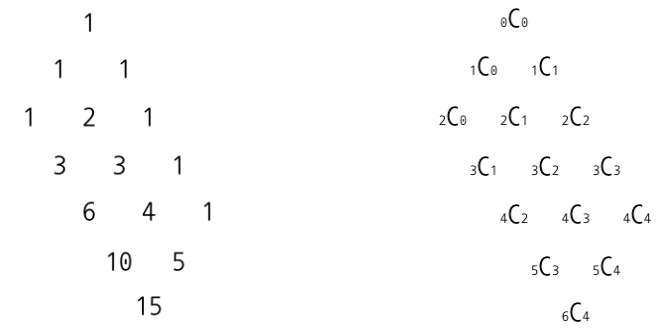
파스칼 삼각형을 잘 살펴보면 조합의 계산 결과가 삼각형으로 쌓여 있음을 알 수 있다. 아래 왼쪽 삼각형과 오른쪽 삼각형이 대응된다.



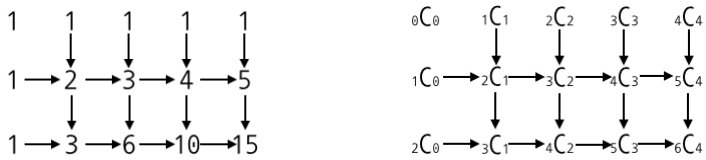
상향식 동적계획 알고리즘

하향식 재귀로 조합을 계산하는 대신, 상향식으로 파스칼 삼각형을 맨 위꼭지점에서 시작하여 아래로 채워나가면 훨씬 빨리 조합을 계산할 수 있다.

즉, $_6C_4$는 아래와 같이 위에서 아래로 계산하면서 평행사변형만 채우면 구할 수 있다.

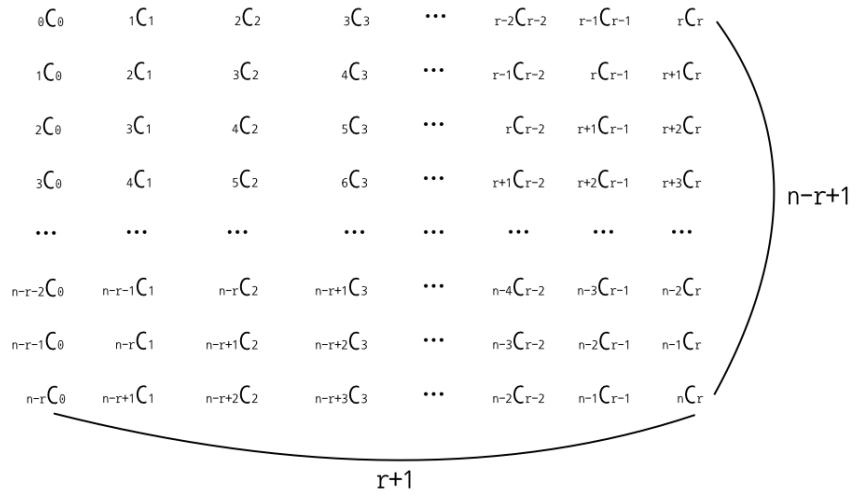


이 평행사변형을 왼쪽 아래로 비스듬히 눌러 직사각형으로 눕혀 보면, 화살표와 같은 방향인 왼쪽에서 오른쪽으로 위에서 아래로 계산을 진행하여 답을 구하는 것과 다름이 없다.



결국 $_6C_4$를 구하려면 $_1C_1$, $_2C_2$,$_3C_3$, $_4C_4$ 열의 값을 모두 1로 놓고, $_1C_0$, $_2C_0$ 행도 모두 1로 놓고, $_2C_1$, $_3C_2$, $_4C_3$, $_5C_4$ 행을 왼쪽에서 오른쪽으로 순서대로 계산하고, 다음에 $_3C_1$, $_4C_2$, $_5C_3$, $_6C_4$ 행을 왼쪽에서 오른쪽으로 순서대로 계산한다. 즉, $_6C_4$를 구하려면 각 행 마다 4회의 덧셈을 하므로 도합 8회의 덧셈을 수행한다.

이제 $_nC_r$을 구하는 방법으로 일반화해보자. $_nC_r$를 구하려면 n-r개의 행을 계산해야 하고, 각 행 마다 r번 덧셈 계산을 해야 한다. 이를 곱하여 총 (n-r) × r 회의 덧셈을 수행한다. 즉, 다음과 같은 (n-r+1) × (r+1) 크기의 행렬을 맨 위쪽의 행($_0C_0$, $_1C_1$, $_2C_2$, …, $_rC_r$)과 맨 왼쪽의 열($_1C_0$, $_2C_0$, $_3C_0$, …, $_{n-r}C_0$)은 모두 미리 1로 정해두고, 왼쪽에서 오른쪽으로, 위에서 아래로 행렬을 채워 나가도록 프로그램을 만든다.



행렬은 어떻게 표현하면 좋을까? 예를 들어, $_6C_4$를 구하는 다음 행렬은

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15

다음과 같이 중첩리스트로 표현한다.

```
[[1, 1, 1, 1, 1], [1, 2, 3, 4, 5], [1, 3, 6, 10, 15]]
```

중첩리스트로 표현하면 행렬의 행과 열 번호가 중첩리스트의 위치번호와 일치하기 때문에 아주 편리하다. 중첩리스트는 다음과 같이 내부 원소를 위치번호로 참조하고 수정할 수 있다.

```
>>> matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> matrix[1][2]
7
```

```
>>> matrix[1][2] = -7
>>> matrix
[[1, 2, 3, 4], [5, 6, -7, 8], [9, 10, 11, 12]]
>>> matrix[1] = [55, 66, 77]
>>> matrix
[[1, 2, 3, 4], [55, 66, 77], [9, 10, 11, 12]]
>>> matrix[0] = 1234
>>> matrix
[1234, [55, 66, 77], [9, 10, 11, 12]]
```

상향식으로 이 행렬을 계산하는 조합계산 함수 comb_pascal은 다음과 같이 작성할 수 있다.

```
1  def comb_pascal(n, r):
2      matrix = [[]] * (n - r + 1)
3      matrix[0] = [1] * (r + 1)
4      for i in range(1, n - r + 1):
5          matrix[i] = [1]
6      for i in range(1, n - r + 1):
7          for j in range(1, r + 1):
8              newvalue = matrix[i][j - 1] + matrix[i - 1][j]
9              matrix[i].append(newvalue)
10     return matrix[n - r][r]
```

이 함수를 차례로 이해해보자.

- 줄 2는 matrix를 빈리스트가 n - r + 1개 들어있는 중첩리스트 [[], [], …, []]로 초기화한다. 즉, n - r + 1개의 빈 행을 만든다.
- 줄 3은 행렬 맨위의 행을 모두 1로 초기화한다. 이제 matrix는 [[1, 1,…, 1], [], …, []]이다.
- 줄 4-5는 맨 왼쪽열을 모두 1로 초기화한다. 이제 matrix는 [[1, 1,…, 1], [1], …, [1]]이다.
- 줄 6-9는 for 문이 이중으로 중첩되어 있다. 안쪽 for 문은 행렬의 행 하나를 계산하여 채우는 작업을 수행하고, 바깥쪽 for 문은 행렬 전체를 계산하여 채우는 작업을 수행한다.
- 줄 10에서 답으로 내주는 값은 행렬의 맨 아래 오른쪽 끝 원소이다.

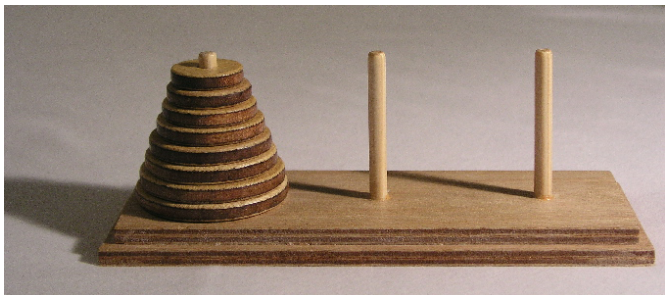
이제 comb_pascal(100,50)을 실행하여 동적계획법의 위력을 확인해보자.

3. 하노이 탑

앞서 공부한 두 사례는 효율적으로 풀 수 있는 해법이 있는 문제이다. 그런데 세상에는 (감내할 수 없을 만큼 비효율적인) 하향식 풀이법이 있지만, 효율적으로 풀 수 있는 방법이 없는 (아마도 아직 찾지 못한) 문제가 수두룩하다. 이 절에서는 이런 유형의 문제를 하나 공부해보자.

문제

크기가 모두 다른 원판이 n개 있고, 말뚝이 세 개 나란이 서 있다. 원판 중간에 구멍이 있어서 원판을 말뚝에 꽂아서 쌓을 수 있는데, 작은 원판이 큰 원판 아래에 놓이지 않도록 쌓아야 한다. 원판이 모두 한 쌓여있고, 원판을 하나씩만 움직여 원판을 모두 다른 한 말뚝으로 옮기자.



재귀 알고리즘

이 문제는 재귀로 사고 하면 쉽게 풀 수 있다. **출발말뚝**과 **도착말뚝**이 지정되면 나머지 말뚝은 **임시말뚝**으로 쓴다. n-1개의 원판을 한 말뚝에서 다른 말뚝으로 옮기는 방법을 안다고 가정하면, n개 원판을 **출발말뚝**에서 **도착말뚝**으로 옮기는 방법은 다음과 같다.

1. **출발말뚝** 상위 n-1개의 원판을 **임시말뚝**으로 옮긴다. **(재귀)**
2. **출발말뚝**의 맨 아래에 남아있는 가장 큰 원판을 **도착말뚝**으로 옮긴다.
3. **임시말뚝** n-1개의 원판을 **도착말뚝**으로 옮긴다. **(재귀)**

아하! 이제 n-1개의 원판을 옮기는 방법만 알아내면 되니 문제가 좀 간단해졌다. 이렇게 재귀적으로 문제를 축소해나가면 언젠간 가장 쉬운 문제인 원판 1개를 옮기는 문제로 축소될 것이다. 이게 재귀 알고리즘의 비밀이다!

이 재귀 알고리즘을 적용하여 원판을 옮기는 순서를 프린트하는 재귀함수는 다음과 같다.

```
1  def hanoitower(n, source, destination, temp) :
2      if n > 1 :
3          tower(n - 1, source, temp, destination)
4          print("move from", source, "to", destination)
5          tower(n - 1, temp, destination, source)
6      else :
7          print("move from", source, "to", destination)
```

원판 4개를 1번 말뚝에서 2번 말뚝으로 3번말뚝을 임시말뚝으로 사용하여 어떻게 옮기면 되는지 알아보기 위해 `hanoitower(4, 1, 2, 3)`를 호출하여 실행하면 다음과 같은 결과가 실행창에 프린트된다.

```
move from 1 to 3
move from 1 to 2
move from 3 to 2
move from 1 to 3
```

```
move from 2 to 1
move from 2 to 3
move from 1 to 3
move from 1 to 2
move from 3 to 2
move from 3 to 1
move from 2 to 1
move from 3 to 2
move from 1 to 3
move from 1 to 2
move from 3 to 2
```

원판의 개수를 늘려서 다음와 같이 함수를 호출하여 실행 결과를 확인해보자.

```
hanoitower(6, 1, 2, 3)
hanoitower(8, 1, 2, 3)
hanoitower(16, 1, 2, 3)
```

실행 결과가 어떻게 나타나는가? 원판을 몇번 옮겨야 하는지 각각 따져보자. 원판이 n개 일때 옮기는 횟수를 식으로 표현해보자.

실습 #1 : Minimum steps to one

동적계획법으로 프로그램을 설계하고 구현하는 실습이다.

문제 :

양수 인수 n을 받아서 (가) 1을 빼거나, (나) 2로 나누어지면 2로 나누거나, (다) 3으로 나누어지면 3으로 나누는 과정을 계속하여, n이 1이 되기까지 이 과정의 최소 반복 횟수를 구하는 함수 `minsteps`를 작성하자.

이 문제를 푸는 식은 다음과 같은 재귀로 표현할 수 있다.

```
minsteps(n) = 1 + minimum {minsteps(n-1), minsteps(n/2), minsteps(n/3)}
              when n > 1
minsteps(1) = 0
```

이 식을 푸는 방법으로 가장 쉬운 방법은 가장 빨리 수가 줄어드는 과정을 우선 적용하는 것이다. 즉, 우선순위를 (다), (나), (가) 순으로 두고 우선순위가 높은 것을 먼저 적용시켜보면 직관적으로 최소 반복 횟수를 구할 수 있을 것 같다. 이와 같이 해답을 구하는 방법을 **탐욕적 풀이법**greedy approach라고 한다. 사례를 가지고 직접 해답을 구해보자.

호출	답	계산 절차
minsteps(1)	0	1
minsteps(2)	1	2 => 1
minsteps(3)	1	3 => 1 3 => 2 => 1
minsteps(4)	2	4 => 2 => 1 4 => 3 => 1
minsteps(7)	3	7 => 6 => 2 => 1 7 => 6 => 3 => 1
minsteps(10)	3	10 => 5 => 4 => 2 => 1 10 => 9 => 3 => 1
minsteps(23)	6	23 => 22 => 11 => 10 => 5 => 4 => 2 => 1 23 => 22 => 21 => 7 => 6 => 2 => 1
minsteps(237)	8	237 => 79 => 78 => 26 => 13 => 12 => 4 => 2 => 1
minsteps(317)	10	317 => 316 => 158 => 79 => 78 => 39 => 13 => 12 => 4 => 2 => 1
minsteps(514)	8	514 => 257 => 256 => 128 => 64 => 32 => 16 => 8 => 4 => 1 => 1 514 => 513 => 171 => 57 => 19 => 18 => 6 => 2 => 1

위의 사례를 보면 탐욕적 풀이법으로 구한 결과가 모두 답은 아님을 알 수 있다. 10, 23, 514가 그런 경우이다. 사실 탐욕적 풀이법으로 답을 빨리 구할 수는 있지만 항상 정답을 얻는다는 보장이 없다. 따라서 이 경우 정답을 확실히 얻는 유일한 방법은 모든 경우를 다 따져보는 수 밖에 없다.

모든 경우를 다 따져보도록 Python 함수를 작성하면 다음과 같다.

```
1  def minsteps0(n):
2      if n > 1:
3          r = 1 + minsteps0(n - 1)
4          if n % 2 == 0:
5              r = min(r, 1 + minsteps0(n // 2))
6          if n % 3 == 0:
7              r = min(r, 1 + minsteps0(n // 3))
8          return r
9      else:
10         return 0
```

이 함수를 실행해보면, n이 커질수록 답을 구하는데 걸리는 시간이 증가한다. 100 미만의 인수는 바로 답이 나오지만, 100을 넘어서면서 시간이 좀 걸리기 시작한다. 514의 경우 좀 오래 기다려야 답이 나온다. 시간이 오래 걸리는 이유는 엄청나게 많은 양의 중복계산을 하기 때문이다. 중복계산을 피하기 위한 방법은 무엇일까? 한 번 계산한 값은 기록해두고 다음에 필요할 때 가져다 쓰면 된다. 이를 **메모해두기** memoization라고 한다. minsteps(n)을 계산하려면 minsteps(1)부터 minsteps(n-1)까지의 계산결과가 잠재적으로 필요하므로 이를 저장해둘 n개의 공간을 메모 리스트로 준비해두면 된다. 메모 리스트의 이름을 memo라고 한다면, minsteps(1)의 계산 결과는 memo[1], minsteps(2)의 계산 결과는 memo[2], ..., minsteps(n)의 계산 결과는 memo[n]에 저장해둔다. (사실 n개의 공간만 필요하지만, memo의 위치번호와 그 장소에 저장되는 minsteps(n)의 n이 일치하면 프로그램이 간단해져 가독성이 좋아지므로 memo 리스트의 길이를 n+1하고 위치번호 0은 쓰지 않는 걸로 한다.) 메모행렬을 이용하도록 수정한 프로그램은 다음과 같다.

```
1  def minsteps1(n):
2      memo = [0] * (n + 1)
3      def loop(n):
4          if n > 1:
5              if memo[n] != 0:
6                  return memo[n]
7              else:
8                  memo[n] = 1 + loop(n - 1)
9                  if n % 2 == 0:
10                     memo[n] = min(memo[n], 1 + loop(n // 2))
11                 if n % 3 == 0:
12                     memo[n] = min(memo[n], 1 + loop(n // 3))
13                 return memo[n]
14         else:
15             return 0
16     return loop(n)
```

이제 이 프로그램을 실행하면 인수가 증가해도 실행시간에 큰 변화가 없이 모두 빠르게 계산한다. 그런데 998 이상의 인수에 대해서는 오류가 발생한다. Python은 재귀함수의 호출을 연속 1,000번까지 밖에 할 수 없기 때문이다. Python에서 재귀함수를 자제해야 하는 이유이다.

이 프로그램을 이해하고 아무리 큰 인수에 대해서도 답을 내줄 수 있도록 for 문을 사용하여 minsteps 함수를 다름 틀에 맞추어 완성하자.

```
1  def minsteps(n):
2      memo = [0] * (n + 1)
3      for
4
5
6
7
8
9      return memo[n]
```

실습 #2 : 구구단 출력

중첩된 for 문을 이용하여 프로그램을 작성하는 훈련을 하기 위한 실습 문제이다. 주어진 실행사례와 정확히 똑같이 실행창에 프린트하도록 중첩 for 문으로 프로그램 해야 한다.

(1) 가로전개 구구단

구구단을 가로로 다음과 같이 실행창에 프린트하는 함수 gugudan1()을 만들어보자.

```
2 x 2 =  4  2 x 3 =  6  2 x 4 =  8  2 x 5 = 10
2 x 6 = 12  2 x 7 = 14  2 x 8 = 16  2 x 9 = 18

3 x 2 =  6  3 x 3 =  9  3 x 4 = 12  3 x 5 = 15
3 x 6 = 18  3 x 7 = 21  3 x 8 = 24  3 x 9 = 27

4 x 2 =  8  4 x 3 = 12  4 x 4 = 16  4 x 5 = 20
4 x 6 = 24  4 x 7 = 28  4 x 8 = 32  4 x 9 = 36

5 x 2 = 10  5 x 3 = 15  5 x 4 = 20  5 x 5 = 25
5 x 6 = 30  5 x 7 = 35  5 x 8 = 40  5 x 9 = 45

6 x 2 = 12  6 x 3 = 18  6 x 4 = 24  6 x 5 = 30
6 x 6 = 36  6 x 7 = 42  6 x 8 = 48  6 x 9 = 54

7 x 2 = 14  7 x 3 = 21  7 x 4 = 28  7 x 5 = 35
7 x 6 = 42  7 x 7 = 49  7 x 8 = 56  7 x 9 = 63

8 x 2 = 16  8 x 3 = 24  8 x 4 = 32  8 x 5 = 40
8 x 6 = 48  8 x 7 = 56  8 x 8 = 64  8 x 9 = 72

9 x 2 = 18  9 x 3 = 27  9 x 4 = 36  9 x 5 = 45
9 x 6 = 54  9 x 7 = 63  9 x 8 = 72  9 x 9 = 81
```

(2) 세로전개 구구단

구구단을 세로로 다음과 같이 실행창에 프린트하는 함수 gugudan2()을 만들어보자.

```
2 x 2 =  4  3 x 2 =  6  4 x 2 =  8  5 x 2 = 10
2 x 3 =  6  3 x 3 =  9  4 x 3 = 12  5 x 3 = 15
2 x 4 =  8  3 x 4 = 12  4 x 4 = 16  5 x 4 = 20
2 x 5 = 10  3 x 5 = 15  4 x 5 = 20  5 x 5 = 25
2 x 6 = 12  3 x 6 = 18  4 x 6 = 24  5 x 6 = 30
2 x 7 = 14  3 x 7 = 21  4 x 7 = 28  5 x 7 = 35
2 x 8 = 16  3 x 8 = 24  4 x 8 = 32  5 x 8 = 40
2 x 9 = 18  3 x 9 = 27  4 x 9 = 36  5 x 9 = 45

6 x 2 = 12  7 x 2 = 14  8 x 2 = 16  9 x 2 = 18
6 x 3 = 18  7 x 3 = 21  8 x 3 = 24  9 x 3 = 27
6 x 4 = 24  7 x 4 = 28  8 x 4 = 32  9 x 4 = 36
6 x 5 = 30  7 x 5 = 35  8 x 5 = 40  9 x 5 = 45
6 x 6 = 36  7 x 6 = 42  8 x 6 = 48  9 x 6 = 54
6 x 7 = 42  7 x 7 = 49  8 x 7 = 56  9 x 7 = 63
6 x 8 = 48  7 x 8 = 56  8 x 8 = 64  9 x 8 = 72
6 x 9 = 54  7 x 9 = 63  8 x 9 = 72  9 x 9 = 81
```

힌트

문자열 연산	의미	사례
s.rjust(n)	n 자리를 확보한 다음 문자열을 오른쪽에 맞추어 조정하는 메소드	'h'.rjust(2) => ' h'

print 속성	의미	사례

print(…, end=<문자열>)	end 속성의 기본값은 '\n'이다.
	기본값이란 지정하지 않아도 저절로 갖고 있는 값을 말한다.
	그래서 프린트를 한 후 언급이 없으면 저절로 줄을 넘긴다.
	이를 바꾸고 싶은 경우 end 속성을 지정하면 된다.
		>>> print(3); print(4)
		3
		4
		>>> print(3, end=' '); print(4)
		3 4