CS206A: Data Structures

2017-08-30

카이스트 전산학부 정지원(Otfried Cheong) 著
부경대학교 컴퓨터공학과 김태원 譯

2022년 6월 23일

재귀

재귀(recursion)는 알고리즘 및 자료구조 설계에서 가장 중요한 기법이다. 일반적으로 재귀는 어떤 것을 자기 자신에 대해 정의하는 것을 뜻한다. 가령 우리는 (루트를 지니는) **트리**를 다음처럼 정의할 수 있다. 하나의 트리는 0개 이상의 트리의 루트와 연결되어 있는 루트 노드 하나를 포함한다. 여기서 재귀의 위력은 유한 명제(finite statement)로 대상들의 무한 집합(infinite set of objects)을 정의할 수 있다는 가능성에 있다.

컴퓨터 과학에서 재귀는 자기 자신을 (직접, 또는 간접적으로) 호출하는 하나의 함수를 뜻한다. 당신이 재귀와 처음 마주할 적에 재귀는 마치 일종의 마술처럼 보일지도 모른다. 그러나 한 번 익숙해지기만 하면 재귀는 문제 해결을 위한 강력하고도 꽤 자연스러운 기법이 된다.

인간은 어려운 문제를 다음처럼 해결한다. 우선 어려운 문제를 쉬운 보조 문제들로 나누고 이들 보조 문제를 해결하는 것이다. 가령 좋은 관리자라면 과제를 취해 이를 작은 과제들로 쪼개 자신의 팀에게 이들 과제를 다루라고 맡길 것이다. 프로그래밍에서 이는 우리가 문제를 취하는 메인 함수(main function)를 지닌다는 것을 뜻한다. 또 이를 작은 문제로 나눠 각 보조 문제(subproblems)에 대해 서브루틴(subroutine)을 호출한다(call)는 것을 뜻한다.

이따금 "쉬운 보조 문제"란 실상 우리가 애초에 시작한 바와 같은 문제이되 더 작거나 쉬운 입력(input)에 대한 것일 따름이라는 사실이 밝혀진다. 이 경우에는 보조 문제에 대한 함수가 따로 존재하지 않는다. 대신 우리의 메인 함수가 보조 문제를 해결하고자 자기 자신을 **재귀적으로** 호출한다.

우리의 관리자 유비(類比, analogy)에서 관리자는 작은 보조 문제들을 **그 자신에게** 위임한다. 혹시 이와 같은 자기 지시(self-reference)가 혼란스럽게 다가온다면, 다른 사람, 즉 어느 팀 구성원이 작은 보조 문제들을 해결하려는 비재귀적인 광경을 상상하여 대별하면 도움이 될 테다.

이와 같은 "다른 사람"을 위해 제프 에릭슨(Jeff Erickson)은 **재귀 요정**이라는 이름을 발명한 바 있다. 인용하면 아래와 같다.

> "당신의 유일한 임무는 원래 문제를 단순화하는 것이다. 또 단순화가 더는 불필요하거나 불가능하다면 직접 해결하는 것이다. 재귀 요정은 당신을 위해 마술적으로 모든 단순한 보조 문제를 처리해줄 것이다. 바로 네 알 바 아니니 손 떼 메소드(Methods That Are

None Of Your Business So Butt Out)를 통해. 수학적인 교양이 풍부한 독자라면 재귀 요정이란 더 형식적인 이름으로는 바로 귀납 가설이라는 사실을 알아챘을 것이다."

우리는 재귀를 다음처럼 느슨하게 정의한다.

- 문제가 충분히 작거나 단순하면 직접 해결하라. (**기초 단계**)
- 그게 아니라면 문제를 같은 문제의 더 단순한 인스턴스(instance)로 나눠 이들을 재귀적으로 푼 다음 그 해들을 결합하라.

하나의 수를 임의의 진법으로 출력하기

간단한 예제로 시작하자. 83790이라는 수를 어떻게 8진법으로 표현할 것인가? 더 일반적으로 묻는다면 주어진 n이라는 수를 어떻게 b진법으로 표현할 것인가?

이 표현의 **마지막 자릿수**를 찾는 일은 쉽다. $n \bmod b$(파이썬으로는 `n % b`)다. 나머지 자릿수는 $\lfloor n/b \rfloor$의 b진법 표기(파이썬으로는 `n // b`)다.

아래 메소드는 재귀적인 전략을 구현한다.

```
DIGITS = "0123456789abcdef"

def print_rec(n, base):
    if n >= base:
        print_rec(n // base, base)
    digit = n % base
    print(DIGITS[digit], end="")
```

기초 단계가 있다는 점에 유의하라. 수가 b보다 작다면 재귀 호출 없이 수 n을 직접 출력한다.

재귀 함수를 설계할 적에 학생들이 저지르는 일반적인 실수가 하나 있다. 프로그램이 어떻게 실행되는지 정신적으로 시뮬레이션하고자 하는 것이다. 대신 당신은 **재귀 요정을 믿어야 한다**. 당신은 재귀 호출이 제대로 작동한다는 가정하에 당신의 메소드에 관해 추론해야 한다. 그런 다음 메소드 자체가 옳은 일을 하리라고 논증해야 한다.

달리 말해 당신의 **유일한** 임무는 원래 문제를 **단순화**하거나 단순화가 더는 불필요하거나 불가능할 적에는 직접 해결하는 것이다. **재귀 요정**은 단순한 보조 문제를 마술적으로 처리할 테다.

수학에서는 재귀가 앞선 트리 정의를 비롯해 언제든 나타난다. (가령 자연수 $\mathbb{N}$ 정의에서도 나타난다. **0은 자연수다; n이 자연수면 $n+1$도 자연수다.**)

수학적 귀납법은 재귀의 한 형식이다. 그리고 물론 재귀적 메소드 하나가 제대로 작동한다고 증명하고자 할 때는 귀납 증명을 사용한다. 가령 `print_rec`이 옳다고 증명한다고 해 보자. n을 b진법으로 작성했을 적에 n의 자릿수의 수를 k라고 부른다고 했을 때, 우리는 바로 이 k에 대해 귀납법을 사용할 것이다. (달리 말해 $k = \lceil \log_b(n+1) \rceil$)

- 기초 단계 $k = 1$: n이 하나의 자릿수만 지닌다면 $0 \leq n < b$이다. 따라서 우리는 `print_rec`의 기초 단계에 있다. 그리고 `print_rec`은 하나의 자릿수를 정확하게 출력한다.

- 귀납 단계 $k > 1$: 우리는 `print_rec`이 k 자릿수보다 작은 수에 대해 정확하게 작동한다고 가정한다. 이제 b진법에 k 자릿수의 n이라는 수를 고려해 보자. 그렇다면 $\lfloor n/b \rfloor$는 $k-1$ 자릿수를 지닌다. 그리고 귀납 가설에 의해 이는 재귀 호출 `print_rec(n // b)`가 $\lfloor n/b \rfloor$를 정확하게 출력한다는 것을 의미한다. 메소드는 그런 다음 마지막 자릿수를 출력하고 따라서 수 n이 정확하게 출력된다.

재귀 알고리즘에는 더 단순하고 더 단순한 보조 문제를 향한 환원의 무한수열(infinite sequence of reductions)이 없어야 한다. 종국에 환원은 다른 메소드로 해결될 수 있는 기본적인 기초 단계와 함께 멈춰야 한다. 그러지 않으면 재귀 알고리즘은 영원히 끝나지 않을 것이다.

이것이 의미하는 바는 이렇다. 첫째, 재귀 호출이 요구되지 않는 기초 단계를 포함해야 한다는 당위는 기억되어야 한다. 둘째, 재귀 호출은 어느 의미에서든 원래 호출보다 "쉬워야" 한다. 그래야 진전이 이루어지며 종국에 기초 단계에 이를 수 있다. 달리 말해 재귀 요정은 당신의 요술 도우미이지만 그가 그 자신의 힘만으로 모든 일을 할 수는 없는 법이다. 당신이 문제를 축소화하거나 단순화하지 않은 채 전체 문제를 덜컥 건네면 그는 마법 부리기를 거부할 테다.

하노이의 탑

다시 재귀에 관한 제프 에릭슨의 강의록[1]을 인용하겠다. 하노이의 탑 퍼즐은 1883년 수학자 프랑수아 에두아르 아나톨 뤼카(François Édouard Anatole Lucas)가 ("Lucas dAmien"의 어구전철(語句轉綴, anagram)인) "N. 클라우스 (드 시암)"(N. Claus (de Siam))이라는 필명으로 처음 공개한 것이다. 이듬해, 앙리 드 파르빌(Henri de Parville)은 그 퍼즐을 아래와 같은 놀라운 이야기로 서술한다.

> "세계의 중심을 장식하는 돔 아래 베나레스의 위대한 사원에는 세 개의 다이아몬드 바늘이 고정된 놋쇠 판이 놓여 있는데 각각은 1큐빗 높이에 벌의 몸처럼 두껍다. 창조할 적에 신은 이들 바늘 가운데 하나에 64개의 순금 원반을 놓았는데 가장 큰 원반은 놋쇠 판 위에 놓고 나머지 원반은 그 위에서 점점 더 작아졌다. 이것이 브라마의 탑이다. 사제들로 하여금 한 번에 하나 이상의 원반을 옮길 수 없고 더 작은 원반이 밑에 놓여서는 안 된다는 브라마의 고정불변의 법칙에 따라 사제들은 밤이고 낮이고 하나의 다이아몬드 바늘에서 다른 다이아몬드 바늘로 원반을 옮긴다. 그리하여 64개의 원반이 신이 창조한 바늘에서 다른 바늘로 옮겨졌을 때 탑과 신전과 브라마인들은 모두 흙으로 바스러지고 우레와 함께 세상은 사라지리라."

자고로 좋은 컴퓨터 과학자가 된다는 말이란 이런 이야기를 읽자마자 단단하게 고정된 상수 64를 n으로 대체한다는 뜻이다. 어떻게 더 작은 원반 위에는 원반을 올리지 않는 동시에 세 번째 바늘을 종종 플레이스홀더(placeholder)로 사용하면서 n개 원반의 탑을 하나의 바늘에서 다른 바늘로 옮길 수 있는가? 이 퍼즐을 푸는 요령은 바로 재귀적인 생각이다. 전체 퍼즐을 한 번에 해결하려고 하지 말고 가장 큰 원반을 옮기는 일에만 주목한다. 가장 큰 원반을 처음부터 움직일 수는 없는데 왜냐하면 다른 모든 원반들이 가로막고 있기 때문이다. n번째 원반을 세 번째 바늘로 옮길 수 있기 전에 그 위에 있는 $n-1$개의 원반을 옮겨야 한다. 따라서 우리가 생각해 내야 할 것은...

[1] https://jeffe.cs.illinois.edu/teaching/algorithms/book/01-recursion.pdf

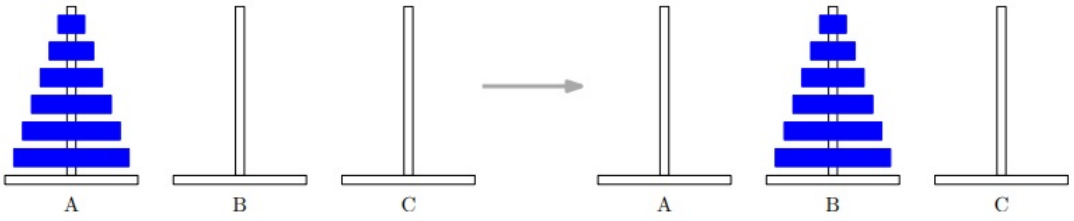

그림 1: 하노이의 탑

멈춰라!! 됐다! 끝났다! 우리는 성공적으로 n 원반 하노이 탑 문제를 해결했다. 바로 $(n-1)$ 원반 하노이 탑 문제의 두 경우를 해결하여 즐겁게 재귀 요정에게 떠넘김으로써 말이다. (아니면 원래 이야기를 이어 가자면 사원의 견습 승려에게 떠넘김으로써 말이다)

우리의 알고리즘에는 미묘하지만 중요한 가정이 하나 있다. 바로 **가장 큰 원반이 있다**는 것이다. 달리 말해 우리의 재귀 알고리즘은 임의의 $n \geq 1$에 대해 작동하지만 $n = 0$일 때는 깨진다. 이런 기초 단계는 직접 다뤄야 한다. 다행히 베나레스의 승려들은 하나의 바늘에서 다른 바늘로 0개의 원반을 옮기는 일에 꽤 능숙하고 우리는 다음 코드에 이른다.

```
1  def solveHanoi(n, source, destination, spare):
2      if n == 1:
3          print("Move disc 1 from %s to %s" % (source, destination))
4      else:
5          solveHanoi(n-1, source, spare, destination)
6          print("Move disc %d from %s to %s" % (n, source, destination))
7          solveHanoi(n-1, spare, destination, source)
8
9  solveHanoi(n, 'A', 'B', 'C')
```

이들 작은 원반이 어떻게 이동하는지 생각하는 일은 매력적이다. 달리 말하면 재귀가 펼쳐질 때는 무슨 일이 일어나냐는 것이다. 다만 필수적인 일은 아니다. 실상 더 복잡한 문제들의 경우 재귀 호출을 펼치는 일은 **산만**할 뿐이다. 우리의 유일한 임무는 문제를 하나 이상의 더 단순한 인스턴스로 환원하거나 그런 환원이 불가능할 경우 문제를 직접 해결하는 것이다. 우리의 알고리즘은 $n = 0$일 때 사소하게 옳다. 임의의 $n \geq 1$에 대해 재귀 요정은 정확하게 상위 $n-1$ 원반을 옮기므로 우리의 알고리즘은 분명하게 옳다.

n 원반에 대해 문제를 해결하는 데는 얼마나 많은 이동이 필요한가? 우리는 재빨리 $n = 0, 1, 2, 3$에 대한 답을 확인한 다음 $0, 1, 3, 7$ 이동이라고 답할 수 있다. 우리는 따라서 n 원반에 대한 이동 수가 $2^n - 1$이라고 추측하고 이 주장을 수학적 귀납법으로 증명한다.

- 기초 단계 ($n = 0$ 원반): 이동 수는 $0 = 2^0 - 1$이며, 따라서 주장은 참이다.
- 귀납 단계: $n > 0$이라고 가정하고 (**귀납 가설**) $n-1$ 원반에 대한 이동 수가 $2^{n-1} - 1$이라고 가정하자. n 원반을 옮기려면 우리는 우선 A에서 C로 $n-1$ 상위 원반을 옮겨야 한다. 귀납 가설에 따라 이 과정에는 $2^{n-1} - 1$ 이동수가 필요하다. 그런 다음 우리는 가장 큰 원반을 옮긴다. (이동수 1) 그리고 최종적으로 $n-1$ 상위 원반을 C에서 B로 옮긴다. 여기에는 다시 귀납

가설에 의해 $2^{n-1}-1$ 이동수가 필요하다. 따라서 전체 이동 수는

$$(2^{n-1}-1)+1+(2^{n-1}-1)=2^n-1$$

이고 우리는 n 원반에 대한 주장을 증명했다.

피보나치 수

피보나치 수 F_n은 다음처럼 정의된다.

$$F_0=0$$
$$F_1=1$$
$$F_n=F_{n-1}+F_{n-2}$$

이는 즉각적으로 피보나치 수를 계산하는 재귀 메소드를 제공한다.

```
1  def fib(n):
2      if n == 0:
3          return 0
4      elif n == 1:
5          return 1
6      else:
7          return fib(n-1) + fib(n-2)
```

알고 보면 이 메소드는 $n \approx 35$일 때 굉장히 느리다. 그 이유를 보기 위해 우리는 함수 `fib`에 대한 호출 및 재귀 호출을 전부 보여주는 트리를 그려 본다. 그림 2를 보라.

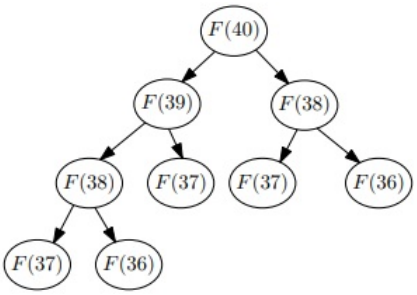

그림 2: `fib(40`을 계산하기 위한 재귀 호출들

트리에서 볼 수 있듯이 `fib(40)`을 계산한다는 말은 F_{38}을 두 번 F_{37}을 세 번 F_{36}을 다섯 번 등등 이런 식으로 계속 계산하여 F_{41-k}가 F_k번 계산된다는 뜻이다. 피보나치 수가 지수적으로 증가하기에 이 프로시저는 순식간에 너무 느려진다. 이 예제의 문제는 재귀 루틴이 **중복**(redundant) 계산을 수행한다는 점이다. 즉 이미 끝난 계산을 또 한다는 것이다. 이 예제는 맹목적으로 쓰인 재귀란 언제나 적절하지만은 않다는 점을 보여준다.

해결책은 (따라서 재귀가 불필요하도록) 이전 피바나치 수 전체를 저장하기 위한 배열을 사용하거나 F_n과 F_{n-1} 모두 함수 `fib(n)`에서 반환하는 것이다(예제 코드 `fibonacci2.py`와 `fibonacci3.py`를 보라).