재귀

하나의 함수에서 자기 자신을 다시 호출해 작업을 수행하는 알고리즘

텍스트이(가) 표시된 사진

자동 생성된 설명

재귀는 하나의 함수에서 자기 자신을 다시 호출해 작업을 수행하는 알고리즘입니다.

재귀로 N부터 1까지 출력하는 함수와 1부터 N까지의 합을 구하는 함수를 한 번 짜보는 시간을 가져보겠습니다.

화살이(가) 표시된 사진

자동 생성된 설명

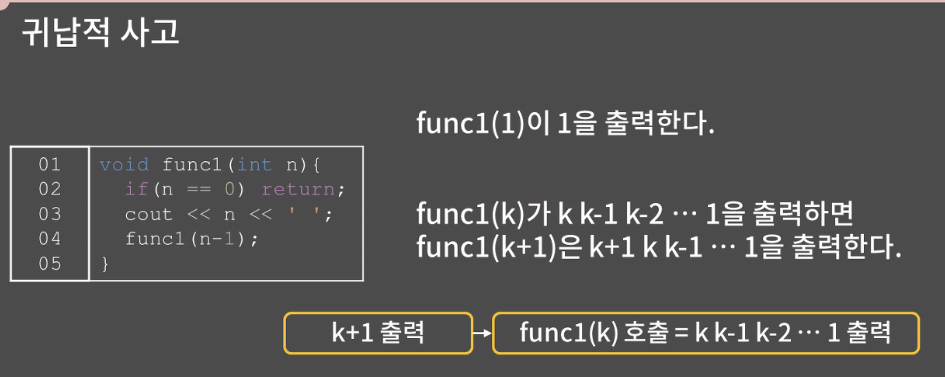
어떤 문제를 재귀로 푼다는 것은 곧 귀납적인 방식으로 문제를 해결하겠다는 것’

 두 번째 설명 방법은 수학적 귀납법을 이용한 방법인데 '1번 도미노가 쓰러진다', 'k번 도미노가 쓰러지면 k+1번 도미노도 쓰러진다'가 참이니까 모든 도미노가 쓰러진다는 설명 방법입니다.

하지만 앞으로는 '1번 도미노가 쓰러진다', 'k번 도미노가 쓰러지면 k+1번 도미노도 쓰러진다' 까지만 생각한 후에 바로 모든 도미노가 쓰러진다는 결론에 도달할 수 있어야 합니다.



일단 func1(3)가 호출되면 3을 출력하고 func1(2)를 호출합니다. func1(2)는 2를 출력한 후에 func1(1)을 호출할거고 func1(1)은 1을 출력한 후에 func1(0)을 호출합니다. 그리고 func1(0)은 02번 줄에 걸려서 종료됩니다. 이렇게 과정을 따라가고 나면 func1(3)을 실행했을 때 3 2 1이 출력된다는 것을 알 수 있습니다.



.첫 번째로 func1(1)이 1을 출력한다, 이건 굉장히 자명합니다. 그 다음이 관건인데 func1(k)가  k k-1 k-2 … 1을 출력하면, 즉 k부터 1까지 차례대로 출력하면 func1(k+1)은 k+1부터 1까지 차례로 출력한다는걸 보여야 합니다.

이걸 보이는건 func1(k+1)이 호출될 때 어떤 상황이 생기는가를 보면 되는데 k+1이 출력된 이후 func1(k)가 호출되고 func1(k)는 k부터 1까지 차례로 출력한다 가정을 했으니 func1(k+1)은 k+1부터 1까지 차례대로 출력함을 알 수 있습니다.

이 두 문장이 참이므로 귀납적으로 func1 함수가 n부터 1까지 차례로 출력하는 함수임을 알 수 있게 됩니다. 이제 귀납적으로 생각하는 것에 친숙해졌나요?

텍스트이(가) 표시된 사진

자동 생성된 설명

올바른 재귀 함수는 반드시 특정 입력에 대해서는 자기 자신을 호출하지 않고 종료되어야 합니다. 이러한 입력을 base condition 내지는 base case라고 합니다.

. 이 코드를 보면 n = 0일 때 자기 자신을 호출하지 않고 종료가 되니 이것이 base condition이고 우리는 이 함수에 자연수만 넣을테니 모든 입력은 결국엔 n = 0으로 수렴하게 됩니다.

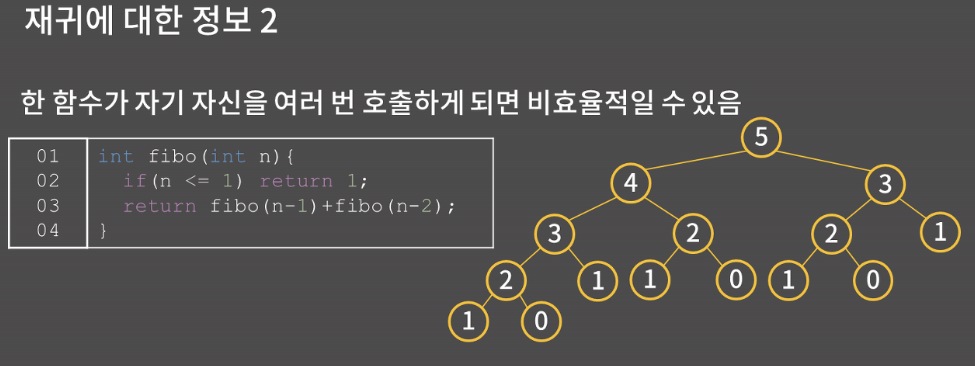
무한히 들어가다가 런타임 에러가 발생하게 될 것입니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

재귀에서는 함수를 명확하게 정의해야 합니다. 정의라는건 함수의 인자로 어떤 것을 받을지, 그리고 어디까지 계산한 후 자기 자신에게 넘겨줄지를 의미합니다. 예시 문제를 풀어볼 때 함수의 형태를 잡는 부분을 계속 강조할 것입니다.

 모든 재귀 함수는 재귀 구조 없이 반복문만으로 동일한 동작을 하는 함수를 만들 수 있습니다. 재귀는 적재적소에 사용하면 코드가 간결해지지만 함수 호출이 꽤 비용이 큰 연산이기 때문에 메모리와 시간에서는 손해를 봅니다. 그렇기 때문에 굳이 재귀를 쓰지 않아도 구현에 큰 어려움이 없으면 재귀 대신 반복문으로 코드를 짜는게 좋지만 재귀 없이 구현을 하면 코드가 너무 복잡해지는 일부 문제들은 재귀로 구현을 하는게 좋습니다.



 재귀 함수에 대해 또 한 가지 알고 있어야하는 점은, 재귀 함수가 자기 자신을 여러 번 호출하게 되면 예상과는 다르게 굉장히 비효율적일 수 있습니다.

재귀 함수가 자기 자신을 여러 번 호출하게 되면 예상과는 다르게 굉장히 비효율적일 수 있습니다

 fibo(5)를 예로 들어 설명드리겠습니다. fibo(5)는 fibo(4)와 fibo(3)을 호출하고 fibo(4)는 fibo(3)과 fibo(2)를, fibo(3)은 fibo(2)와 fibo(1)을 호출합니다. 전체적인 재귀 호출 상황을 나타내보면 슬라이드의 그림과 같습니다.

이미 계산한걸 또 계산한다는 일이 아주 빈번함을 알 수 있습니다. 당장 fibo(3)만 보더라도 왼쪽에서 fibo(3)을 계산하기 위해 fibo(2)와 fibo(1)을 부르고 fibo(2)는 fibo(1)과 fibo(0)을 부르는 일이 발생했는데 오른쪽에서 또 fibo(3)을 계산하려고 함수를 따라들어가는 짓을 합니다. 이렇게 이미 계산한 값을 다시 계산하는 일이 빈번하게 발생해서 시간복잡도가 말도 안되게 커져버렸습니다. 이와 같이 한 함수가 자기 자신을 여러 번 호출할 경우에는 시간복잡도가 이상하게 될 수 있어서 조심해야 합니다.

텍스트이(가) 표시된 사진

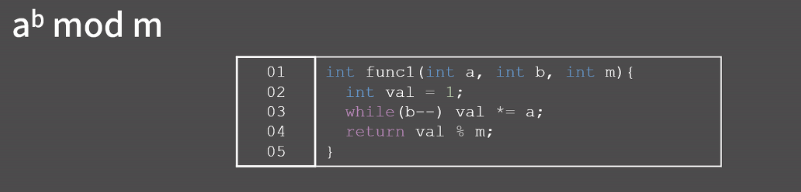
자동 생성된 설명

재귀 함수가 자기 자신을 부를 때 스택 영역에 함수에 대한 정보가 누적됩니다. 이 스택 영역이라고 하는 것은 메모리 구조에서의 스택 영역을 말하는건데 태어나서 생전 처음 들어보셨다고 하시는 분은 나중에 메모리 구조와 관련된 수업을 들으면 배우게 될 것입니다.

채점 사이트 중에서도 설정을 따로 바꾸지 않아 스택 메모리의 제한이 문제의 메모리 제한과 별도로 작게 걸려있는 경우가 있습니다.

BOJ는 스택 메모리의 제한이 없지만 현재(2020년 6월) 기준 swexpertacademy.com 에는 제한이 걸려 있습니다. 그래서 지금 이 코드처럼 재귀를 한 10만번 정도만 들어가도 스택 메모리에 함수에 대한 정보가 계속 쌓이다가 1MB를 넘겨서 제출해보면 런타임 에러가 발생합니다. 만약 swexpertacademy.com 과 같이 스택 메모리가 작게 제한된 곳에서 문제를 푸는데 본인의 풀이가 재귀를 깊게 들어가는 풀이라면 어쩔 수 없이 재귀 대신 반복문으로 문제를 풀어야합니다. 참고로 스택 메모리에는 지역 변수도 들어갑니다.

그리고 BOJ에 제출하면 "맞았습니다"가 뜨는 남의 코드를 로컬에서 돌렸을 때 계속 런타임에러가 나는 일을 겪어보신 분이 있을텐데 가장 의심해볼만한건 재귀가 너무 깊거나 지역 변수로 int arr[2000][2000]과 같 큰 배열을 잡았지 않았을까 하는 것입니다. int 400만개면 벌써 16MB를 잡아먹기 때문입니다. 만약 본인의 개발 환경에서 슬라이드 상의 저 코드를 실행했을 때 정상적으로 동작하지 않는다면 구글 검색을 통해 스택 메모리 제한을 해제해두시는걸 추천드립니다.



 6100 mod 5가 1이라는 것을 알 수 있을 것입니다. 하지만 결과는 놀랍게도 1이 아니라 0이 나옵니다.

그 이유는 바로 int overflow 때문입니다. 6100은 int의 범위를 까마득하게 벗어났습니다. (int의 범위 -20억~20억)이걸 해결해주려면 곱하는 중간 중간 계속 m으로 나눠서 나머지만 챙겨가면 됩니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

 왜 m으로 나눈 나머지만 챙겨가면 되는지 이해가 잘 안갈 수 있는데, 수학적으로 증명을 하려고 해도 그다지 어렵지 않게 할 수 있지만 어떤 식으로도 생각을 해볼 수 있냐면 234236116 × 268921 × 29123의 일의 자리를 구할 때 아마 대부분의 사람이 직접 저 값을 계산하는 대신 그냥 각 수의 일의 자리인 6, 1, 3만 곱한 후 답이 8이라는 것을 알아낼 것 같습니다. 왜 이렇게 계산을 하는거냐면 우리는 이미 직관적으로 a × b × c의 일의 자리, 즉 10으로 나눈 나머지는 a와 b와 c 각각의 일의 자리를 구한 후 곱하면 된다는 것을 알고 있기 때문입니다. 지금 코드도 10이 m으로 달라졌다는 것 뿐이지 우리가 상식적으로 알고있던 그 내용 그대로입니다.

그리고 type도 long long으로 바꿔주면 더 좋겠습니다. int overflow를 고려한 코드는 확인해보세요. 코드를 보면 m 미만의 수 2개를 곱하는 상황이 계속 발생하니 m이 232 보다 크다면 long long의 범위조차 넘어설 수 있습니다. 이런 상황에서는 그냥 Big Integer 기능이 있는 JAVA 혹은 Python을 사용하거나 \_\_int128과 같은  것을 써야 하지만 정상적인 코딩테스트라면 m이 232 보다 작을 것입니다. 결론적으로 저희는 ab mod m을 O(b)에 구할 수 있습니다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

우리가 12116 mod 67을 구하고 싶다고 하면 되게 막막합니다. 그런데 제가 1258 mod 67이 4라는걸 알려드린다면 116승을 구하는게 아주 쉬워진다는걸 알겠죠? 첫 번째 힌트로 드린 성질을 쓰면 그냥 4 × 4를 하면 끝이기 때문입니다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

1258 과 12116 의 관계에서 보듯 k승을 계산했으면 2k승을 계산할 수 있다는건 이해가 갈거고, 2k승을 계산하고 a를 한 번만 곱해주면 2k+1승이 구해지니 2k승과 2k+1승 모두 k승을 계산했으면 O(1)에 계산할 수 있습니다. 이 두 문장이 참이기 때문에 우리는 a의 임의의 지수승을 귀납적으로 계산할 수 있습니다.