5주차 - Recursion

이번주에는 재귀(Recursion)에 대해서 공부해 보겠습니다. Recursion은 함수가 인수만 바꿔서 자기 자신을 다시 호출하는 방식으로 동작하는데, 익숙해지면 많은 문제를 해결하기 위한 알고리즘을 보다 편하게 생각하고 구현할 수 있습니다. 주의해야할 점은 이런 재귀적인 방식에서는 중복된 호출이 많이 발생하면서 코드의 효율 또는 성능이 크게 저하될 수 있다는 부분입니다. 이 문제를 피하기 위해 동적 프로그래밍(Dynamic Programming, DP), Tail Recursion을 사용하거나 반복(Iteration) 형태로 바꾸는 방법을 사용할 수 있습니다.

피보나치 수열

정수 n이 주어졌을 때, n번째 피보나치 수를 계산하는 함수를 작성하세요. 100번째 피보나치 수를 1초 미만으로 계산할 수 있어야 합니다.

피보나치 수열은 다음의 식으로 계산이 가능합니다.

fib(n) = fib(n-2) + fib(n-1)

따라서 return fib(n-2) + fib(n-1)에 더해서 n = 1, 2인 경우 1을 반환하게 하면 아주 쉽게 재귀버전을 구현할 수 있습니다. 하지만 이 재귀버전을 이용하면 엄청난 중복호출이 발생하므로, 100번째 피보나치 수를 계산하는데 매우 많은 시간이 걸립니다. 직접 실행해서 확인해보세요.

Tail recursion은 함수 호출시 stack을 매번 다시 할당하는 것을 피하고 코드를 최적화하여 실행하기 위해 재귀 호출을 함수 실행의 마지막 명령(Tail)으로 만드는 방식입니다. 이를 위해서는 재귀호출에서 반환된 값으로 함수 내부에서 무엇인가 추가적인 연산을 해서는 안됩니다. 위의 예시에서는 fib(n-2)와 fib(n-1)이 반환될 때까지 기다려서 둘의 값을 더하도록 되어있으므로, tail recursion이 성립되지 않습니다.

이런 문제를 피하는 요령은, 계산이 필요한 경우 이를 저장하는 변수를 함수의 인자로 계속 넘겨주는 것입니다. 예를 들면, fib(n-2)와 fib(n-1)에 해당하는 값을 a, b라는 변수에 저장하기로 하고, fib(n, a, b)의 형태로 함수를 정의할 수 있습니다. 이런 요령은 다른 경우에도 적용이 되므로 간단한 피보나치 수의 예를 이용해 익숙해지는 것이 좋습니다. tail recursion을 사용한 피보나치 수 계산 함수는 쉽게 검색이 가능하므로 구현을 시도해 보고 잘 되지 않으면 찾아서 확인하시기 바랍니다.

피보나치 수는 DP를 이용해 구할 수도 있습니다. 특정 n에 해당하는 피보나치 수를 저장하는 배열을 만들고 읽어오는 방식으로 구현하면 됩니다.

코드는 좀 더 복잡해지나, 피보나치 수는 재귀 호출 대신 반복문을 사용하여 구할수도 있습니다. 사실 이론적으로 모든 재귀호출 함수는 반복 형태로 변환할 수 있습니다. 적절히 구현한다면 재귀를 사용하여 구현한 것보다 좋은 성능을 보여주지만, 문제는 대부분의 경우 반복을 사용한 코드가 더 복잡하고 직관적이지 않다는 것입니다.

피보나치 수를 구하는 함수는 매우 단순하므로 위의 4가지 버전 - Recursion, Tail Recursion, DP, Iteration - 을 모두 연습해 보기 좋은 문제입니다. 처음에는 Recursion 형태로 빠르게 코드를 작성하고, 이를 나머지 3가지 형태로 변환하는 연습을 하시기 바랍니다.

하노이의 탑 (Tower of Hanoi)

하노이의 탑은 피보나치 수와 마찬가지로 Recursion에 대한 매우 유명한 예제입니다. 주어진 문제에서 디스크의 숫자를 기준으로 삼고, 더 적은 숫자의 디스크에 대한 해를 이용하여 현재의 디스크 숫자에 대한 해를 구하는 방식으로 문제를 해결합니다. 어떤 문제가 주어졌을 때 이를 재귀를 사용하여 해결할 수 있는지 판단하는 방법을 연습하기 위해 좋은 문제입니다. 실제 문제의 조건은 검색 등을 통해 확인해보시기 바랍니다.

하노이의 탑 문제에서 세 개의 기둥에 디스크가 4개 있는 경우 현재 디스크가 있는 기둥에서 주어진 기둥으로 디스크를 옮기는 순서를 출력하는 프로그램을 작성하세요.

1. 이 프로그램을 위에서 얘기했던 3가지 방식을 써서 더 효율적으로 만들 수 있는지 생각해 봅시다.
2. 관심이 있다면 기둥을 하나 더 추가하는 경우 보다 효율적으로 (적은 수의 이동으로) 디스크를 이동시킬 수 있는지 고민해 봅시다. 이 문제를 일반화해서 원래 기둥과 옮기려고 하는 기둥 외에 활용할 수 있는 기둥의 숫자를 인수로 받아 하노이의 탑 문제를 해결할 수 있는지도 생각해 봅니다. 이 문제에 대한 최적화된 해(또는 방법)을 구하는 것은 단순하지 않은 문제이므로, 기둥이 4개인 경우에 코드가 어떻게 달라지는지 생각해 보는 수준에서 끝내도 좋습니다.

Longest Common Subsequence, LCS

일반적인 LCS문제는 주어진 두 sequence에서 공통된 subsequence를 찾는 문제입니다. 우리가 이미 다루었던 Longest common substring문제와의 차이점은 이 subsequence는 반드시 연속될 필요가 없다는 점입니다.

A**BC**D**E**F**G**

X**BC**Y**E**Z**G**

위의 경우 LCS는 BCEG가 됩니다. 구하는 방식 자체는 우리가 이미 다룬 것과 거의 동일합니다. 이미 DP를 이용해 구하는 방식은 연습을 하였으므로, 이를 tail recursion이나 iteration을 이용하여 해결할 수 있는 코드를 작성하는 연습을 해보세요. Recursion 자체에 보다 집중하기 위해 실제 subsequence를 찾는 것이 아닌 subsequence의 길이만 구하는 코드를 작성합니다. 우선 단순히 recursion을 사용하는 코드를 작성한 뒤 이를 다른 방법을 이용하도록 변경하는 방식으로 연습하고, 작성된 코드의 수행시간을 비교해 봅니다. 이 문제는 한 번 다뤘던 것이므로 검색을 통해 찾아보기 보다는 스스로 처음부터 구현해 보시기 바랍니다.

행렬 경로 문제

n x n 행렬이 주어졌을 때, 왼쪽 위 (0,0)에서 시작하여 오른쪽 아래 (n-1, n-1)까지 이동하며 방문한 칸에 있는 수를 더한 값을 경로의 합으로 놓고, 가능한 경로 중 가장 높은 경로의 합을 구하는 문제입니다.

왼쪽 위에서 출발하여 한 번에 1칸씩 움직이는데, 오로지 오른쪽 또는 아래쪽으로만 움직일 수 있습니다. 이를 이용하여 (0, 0)에서 (i, j)까지 이르는 경로의 합의 최대값을 재귀적으로 구하는 것이 가능하지만, 중복호출로 인해 코드가 비효율적이 됩니다. 따라서 앞에서 얘기했던 다른 방식을 이용하여 효율적인 코드를 작성할 필요가 있습니다. 어떤 방식이 좋을지 생각하여 코드를 작성해 보세요.

n이 주어졌을때, 1~20사이의 값을 무작위로 선택하여 n x n 행렬을 만드는 함수를 만들고, 이렇게 생성된 n x n 행렬에 대해 작성한 행렬 경로의 합을 구하는 함수가 잘 동작하는지 확인해 보세요. 작성중의 테스트를 위해 matrix.txt파일에 최대 경로값이 27인 3x3 행렬이 저장되어 있습니다.