실전 알고리즘 0x06강

재귀

BaaaaaaaaaaaaaaaarkingDog

목차

0x00 재귀(Recursion) – 정의

- 재귀는 하나의 함수에서 자기 자신을 다시 호출해 작업을 수행하는 방식으로 주어진 문제를 푸는 방법을 의미합니다.
- 충분히 익숙하지 않으면 남이 재귀로 짠 코드를 이해하는 데에도 정말 오랜 시간이 걸리고, 능숙하게 재귀를 이용할 수도 없습니다.
- 자기 자신을 다시 호출할 때에는 현재 함수에서의 입력값보다 더 작은 값을 인자로 넘겨주어야 합니다.
- 함수의 입력값이 일정 크기 이하일 때에는 더 이상 자기 자신을 호출하지 말고 값을 바로 반환해야 합니다. 이러한 하위 문제를 base condition이라고 부릅니다.

0x00 재귀(Recursion) – 예시

- N을 입력 받아 N부터 1까지 차례대로 출력하는 함수

```cpp
#include <bits/stdc++.h>
using namespace std;

void func(int n){
  cout << n << ' ';
  if(n == 1) return; // base condition
  func(n-1);
}
int main(){
  func(5);
}
```

result : 5 4 3 2 1

0x00 재귀(Recursion) – 예시

- N을 입력 받아 N부터 1까지 곱한 결과(= $N!$)를 계산하는 함수

```cpp
#include <bits/stdc++.h>
using namespace std;

int func(int n){
  if(n == 1) return 1; // base condition
  return n*func(n-1);
}
int main(){
  cout << func(5);
}
```

result : 120

0x00 재귀(Recursion) – 기타 정보

- 함수가 입력에 대해 어디까지 연산을 수행하고, 어떤 입력값을 자기 자신에게 다시 넘겨주어야 할지 잘 정해야 합니다. 이걸 제대로 정하지 않고 무작정 코딩에 들어가면 엄청 헤맵니다.
- 모든 재귀 함수는 재귀 구조 없이 반복문만으로 동일한 동작을 하는 함수를 만들어낼 수 있습니다.(역도 성립합니다.) 재귀를 사용할 경우 반복문으로 구현을 했을 때에 비해 코드를 간결하고 이해하기 쉽게 만들 수 있다는 장점이 있지만 메모리/시간에서는 손해를 봅니다.
- 그렇기 때문에 경험적으로 어떨 때 재귀를 사용하면 유리하고 어떨 때에는 굳이 재귀를 사용할 필요가 없는지를 알고 있는 것이 좋습니다.

0x00 재귀(Recursion) – 기타 정보

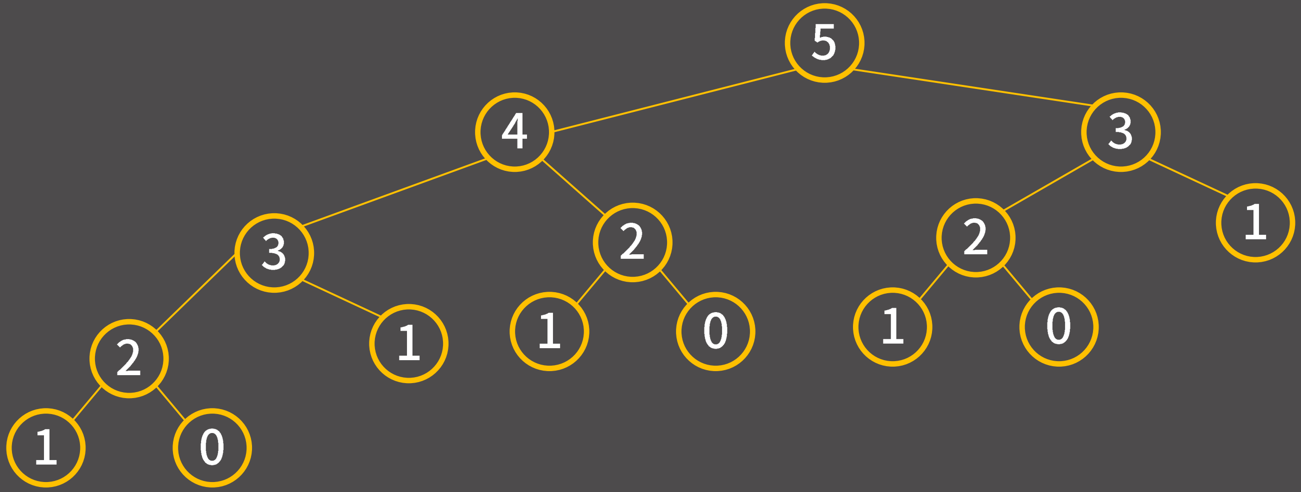
- 한 함수가 자기 자신을 여러 번 호출하게 되면 시간복잡도가 굉장히 커질 수 있습니다. 피보나치 수열은 재귀로 해결하면 안되는 대표적인 예시입니다.
- $F_0 = 1, F_1 = 1, F_i = F_{i-1} + F_{i-2}$ $(i > 1)$ 인 수열을 피보나치 수열이라고 할 때 k번째 항을 구하는 것이 목표입니다.(1 1 2 3 5 8 13…)
- 나중에 다이나믹 프로그래밍을 배우고 나면 O(k)에 구할 수 있음을 알게 되지만, 상식적으로 생각해도 앞에서부터 차례로 계산하면 k번의 덧셈으로 k번째 항을 구할 수 있을 것이라는 것을 쉽게 알 수 있습니다.
- 재귀함수로는 오른쪽과 같이 구현할 수 있습니다.

```
int f(int k){
  if(k <= 1) return 1;
  return f(k-1)+f(k-2);
}
```

0x00 재귀(Recursion) – 기타 정보

- 이 재귀함수는 이미 계산한 것을 중복해서 계산하기 때문에 비효율적으로 동작합니다.
- 시간복잡도는 $O(1.618^k)$입니다.

```
int f(int k){
  if(k <= 1) return 1;
  return f(k-1)+f(k-2);
}
```



0x00 재귀(Recursion) – 기타 정보

- 재귀함수에서 계속 깊이 들어갈 때 스택 메모리에 계속 누적이 됩니다.
- 문제 전체의 메모리 제한만 있고 스택 메모리에 다른 제한이 없다면 애초에 재귀로 스택 메모리를 다 채우고 싶어도 그 전에 시간 초과가 먼저 발생하므로 스택 메모리를 신경쓸 필요가 없습니다.
- 그런데 스택 메모리에 1MB로 제한이 있을 경우에는 분명 정상적인 코드임에도 불구하고 대략 20000-40000번 정도의 깊이를 가진 재귀 함수가 Runtime Error를 발생시킬 수 있습니다.
- 본인의 개발환경에서 오른쪽의 코드가 정상적으로 동작하지 않는다면 구글의 도움을 얻어 스택 메모리 제한을 없애세요.

```
void test(int a){
  if(a>0) test(a-1);
}

int main(void){
  test(3500000);
  cout << "DONE ^__^";
}
```

0x00 재귀(Recursion) – 기타 정보

- 이상하게 삼성이 역량테스트, SW expert academy, SCPC 등에서 스택 메모리를 1MB로 제한합니다. 실제로 앞 장의 코드를 SW expert academy에 제출해보면 Runtime Error가 발생합니다.
- 삼성과 같이 스택 메모리가 굉장히 작게 제한된 곳에서 문제를 풀 때, 본인의 풀이가 재귀 호출을 20000-40000번 이상 해야 한다면 어쩔 수 없이 재귀 호출 말고 반복문으로 풀어야 합니다.

0x01 재귀(Recursion) – 예시 문제 1

- a^b mod m 을 어떻게 구할 수 있을까요?
- 제일 간편한 방법은 그냥 a를 b번 곱하는 것이겠네요. 시간복잡도 $O(b)$에 해결 가능합니다.

```
int func1(int a, int b, int m){
  int val = 1;
  while(b--) val *= a;
  return val;
}
```

- 위의 코드가 제대로 동작하지 않는 이유는 알고 계시죠?

```
int main(void){
  cout << func1(6,12,5);
}
```

result : 0

0x01 재귀(Recursion) – 예시 문제 1

- int overflow 문제를 해결해주면 정상적인 답을 얻을 수 있습니다.
- m이 2^{32} 이상일 경우에는 long long 범위에서도 해결이 안되므로 __int128을 사용하거나 Python 혹은 JAVA를 사용해야 합니다.

```
typedef long long ll;
ll func2(ll a, ll b, ll m){
  ll val = 1;
  while(b--) val = val*a%m;
  return val;
}
```

0x01 재귀(Recursion) – 예시 문제 1

- 그런데 b가 그다지 작지 않고 최대 20억이면 어떻게 해야할까요? (BOJ 1629번 : 곱셈)
- $b = 2k+1$ 일 때, $a^b = (a^k)^2 \cdot a$
- $b = 2k$ 일 때, $a^b = (a^k)^2$
- 재귀 함수의 구조가 그려지나요? 일단 직접 한 번 시도해보시고, 잘 안되면 다음 장의 코드를 참고해보세요.

0x01 재귀(Recursion) – 예시 문제 1

- 정답 코드 : http://boj.kr/2c57da2f313e40c2a2885cafc6a1963f (시간복잡도 O($lg\, b$))

```
POW(3,13,61)
  └→ POW(3,6,61)
       └→ POW(3,3,61)
            └→ POW(3,1,61)
                 └→ POW(3,0,61)
```

0x01 재귀(Recursion) – 예시 문제 1

- 아래의 함수들은 어떤 문제점을 가지고 있을까요?

```
ll POW1(ll a, ll b, ll m){
  ll val = POW(a,b/2,m);
  val = val*val%m;
  if(b%2 == 0) return val;
  return val*a%m;
}

ll POW2(ll a, ll b, ll m){
  if(b==0) return 1;
  ll val = POW(a,b/2,m);
  if(b%2 == 0) return val*val;
  return val*val*a%m;
}

ll POW3(ll a, ll b, ll m){
  if(b==0) return 1;
  ll val = POW(a,b/2,m);
  val = val*val%m;
  if(b%2 == 0) return POW(a,b/2,m)*POW(a,b/2,m)%m;
  return POW(a,b/2,m)*POW(a,b/2,m)*a%m;
}
```

0x01 재귀(Recursion) – 예시 문제 1

- 아래의 함수들은 어떤 문제점을 가지고 있을까요?

```
ll POW1(ll a, ll b, ll m){
  ll val = POW(a,b/2,m);
  val = val*val%m;
  if(b%2 == 0) return val;
  return val*a%m;
}

ll POW2(ll a, ll b, ll m){
  if(b==0) return 1;
  ll val = POW(a,b/2,m);
  if(b%2 == 0) return val*val;
  return val*val*a%m;
}

ll POW3(ll a, ll b, ll m){
  if(b==0) return 1;
  ll val = POW(a,b/2,m);
  val = val*val%m;
  if(b%2 == 0) return POW(a,b/2,m)*POW(a,b/2,m)%m;
  return POW(a,b/2,m)*POW(a,b/2,m)*a%m;
}
```

- POW1 : base condition이 없다.
- POW2 : int overflow
- POW3 : int overflow, 함수를 2번 호출함으로 인해 시간복잡도가 $O(lg\,b)$가 아닌 $O(b)$가 됨

0x02 재귀(Recursion) – 예시 문제 2

- 하노이 탑 문제는 3개의 기둥이 있을 때 작은 원판 위에 큰 원판을 놓을 수 없다는 규칙을 만족시키면서 원판을 한 번에 한 개씩 옮겨 한 기둥에 있는 n개의 원판을 다른 기둥으로 옮기는 문제입니다. n=3,4,5일 때 최소 횟수는 얼마인지 직접 시도해보세요.

0x02 재귀(Recursion) – 예시 문제 2

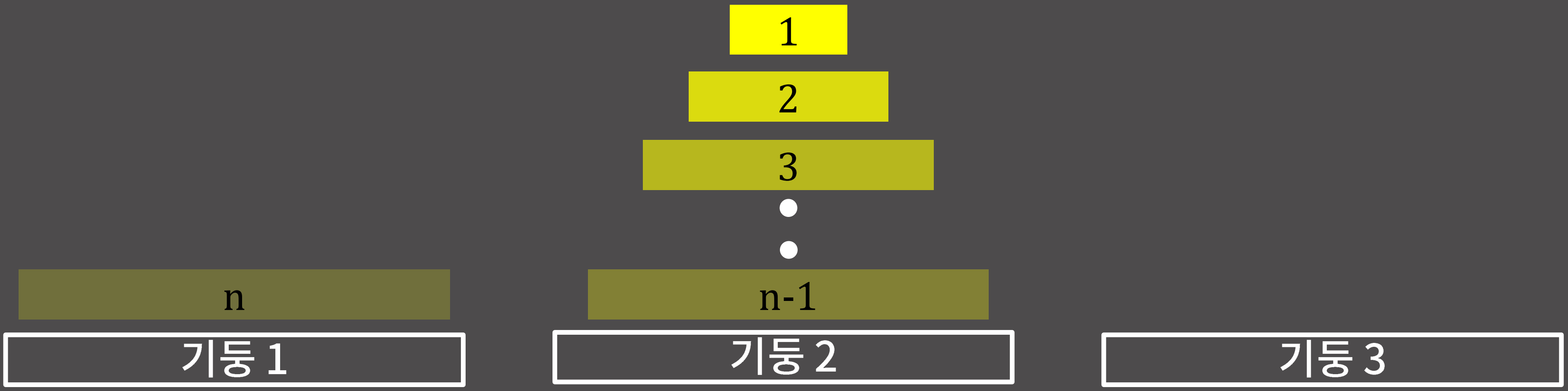
- 기둥 1에 n개의 원판이 있을 때 기둥 3으로 모두 옮기려면 적어도 몇 번이 필요하고, 또 어떻게 옮겨야할까요? (BOJ 11729번 : 하노이 탑 이동 순서)

0x02 재귀(Recursion) – 예시 문제 2

- 어려울 것 같지만 차근차근 생각해보면 쉽습니다. 기둥 1에서 기둥 3으로 모든 원판을 옮기기 위해서는 어떤 절차를 거쳐야하는지 재귀적인 관점에서 충분히 고민해보고 다음 슬라이드로 넘어와주세요.

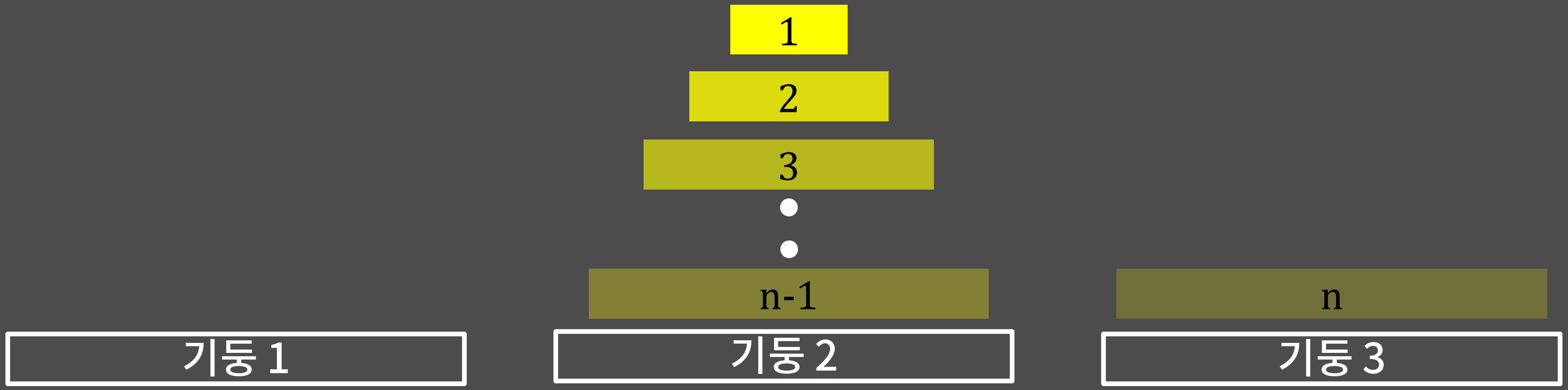
0x02 재귀(Recursion) – 예시 문제 2

- 1. 원판 1부터 n-1까지를 기둥 1에서 기둥 2로 옮깁니다. 그렇지 않으면 원판 n이 움직일 수 없기 때문입니다.



0x02 재귀(Recursion) – 예시 문제 2

- 2. 원판 n을 기둥 1에서 기둥 3로 옮깁니다.



0x02 재귀(Recursion) – 예시 문제 2

- 3. 원판 1부터 n-1까지를 기둥 2에서 기둥 3으로 옮깁니다.

0x02 재귀(Recursion) – 예시 문제 2

- 재귀적으로 생각하니 정말 간단하네요!
- 조금 더 일반화해서 n개의 원판이 놓인 기둥을 a, 목적지를 b, 원판이 놓여있지도 않고 목적지도 아닌 기둥을 c라고 합시다. 그렇다면 n개의 원판을 a에서 b로 옮기는 과정은 3개의 과정으로 나눌 수 있습니다.

- 1. n-1개의 원판을 a에서 c로 옮깁니다.
- 2. 마지막 원판을 a에서 b로 옮깁니다.
- 3. n-1개의 원판을 c에서 b로 옮깁니다.

- 예외적으로 n이 1일 경우, 2번 과정만 하면 됩니다.

0x02 재귀(Recursion) – 예시 문제 2

- **func(a,b,n)** 를 n개의 원판을 a에서 b로 옮기는 과정이라고 할 때 앞에서 살펴본 과정에 따라 재귀적으로 코드를 짤 수 있습니다.

```
void func(int a, int b, int n){
  if(n==1){ // a에 있는 원판 1개를 b로 옮기기만 하면 됨
    cout << a << ' ' << b << '\n';
    return;
  }
  int c = 6-a-b; // a, b가 아닌 나머지 기둥의 번호
  func(a,c,n-1); // a에 있던 1 to n-1번째 원판을 a에서 c로 이동
  cout << a << ' ' << b << '\n'; // a에 남아있던 n번째 원판을 b로 이동
  func(c,b,n-1); // c에 있던 1 to n-1번째 원판을 c에서 b로 이동
}
```

0x02 재귀(Recursion) – 예시 문제 2

- 더 나아가 재귀적으로 생각한 방법으로부터 n개의 원판을 옮기는 최소 횟수 = n-1개의 원판을 옮기는 최소 횟수×2+1 임을 알 수 있습니다.
- $A_1 = 1$, $A_n = 2A_{n-1}+1$이고 이 점화식의 일반항은 $A_n = 2^n-1$ 입니다. 점화식의 일반항을 구하는 방법을 알아야 해당 식을 도출할 수 있긴 하지만, 코딩 테스트에서 하노이 탑이 아니면 점화식의 일반항을 물어볼 일은 없기 때문에 어떻게 $A_n = 2^n-1$을 구할 수 있는지는 몰라도 괜찮습니다.
- 정답 코드 : http://boj.kr/e2da8c1a1c924385a81efa9a054744da

0x03 문제 소개

문제 번호	발상 난이도	구현 난이도
1074	3/10	1/10
2447	3/10	2/10
2448	3/10	3/10
1992	4/10	4/10
16684	9/10	9/10

- 이번 시간에 재귀를 확실하게 익혀둬야 다음 시간에 할 백트래킹을 무난하게 넘어갈 수 있습니다. 문제 꼭 풀어보세요!

강의 정리

- 재귀함수에 대해 이해하고 한 함수가 자기 자신을 여러 번 부를 경우 비효율적일 수 있음을 알게 되었습니다.
- 거듭제곱 계산하기, 하노이 탑 문제를 재귀함수로 풀어보았습니다.