



# 게임 트리 개요와 확률적 탐색

게임 트리를 구성하고 승리 전략을 도출하는 기본 방법 이해

01. 퀴즈 풀이 & 예습 내용 복습

02. 게임 **트리 구성** & annotation 방법

03. 구성한 게임 트리로부터 **승리전략을 도출**하는 방법

게임의 여러 특성 분석을  
위해 트리를 활용하는 예

04. 게임 트리 관련 다양한 문제 풀이 (게임 규칙의 문제점 파악)

05. **더 빠르게** 게임 트리를 구성하고 승리 전략을 도출하는 방법

06. 게임 트리의 구현과 **확률적** 탐색

07. 실습문제 풀이

탐색 범위를 줄여가되  
**확률적으로** 줄임



## 이번 장에서 말하는 ‘게임’이란 무엇인가?

2

- 한 명 이상의 player가 지정된 규칙에 따라 play
- 각 player의 목표는 게임에서 이기는 것 (혹은 지정된 목표 달성)
- 아래 예제 참조



한 번에 동전 하나 혹은 두 개 가져가기: 승리할 수 있는 전략이 있을까?

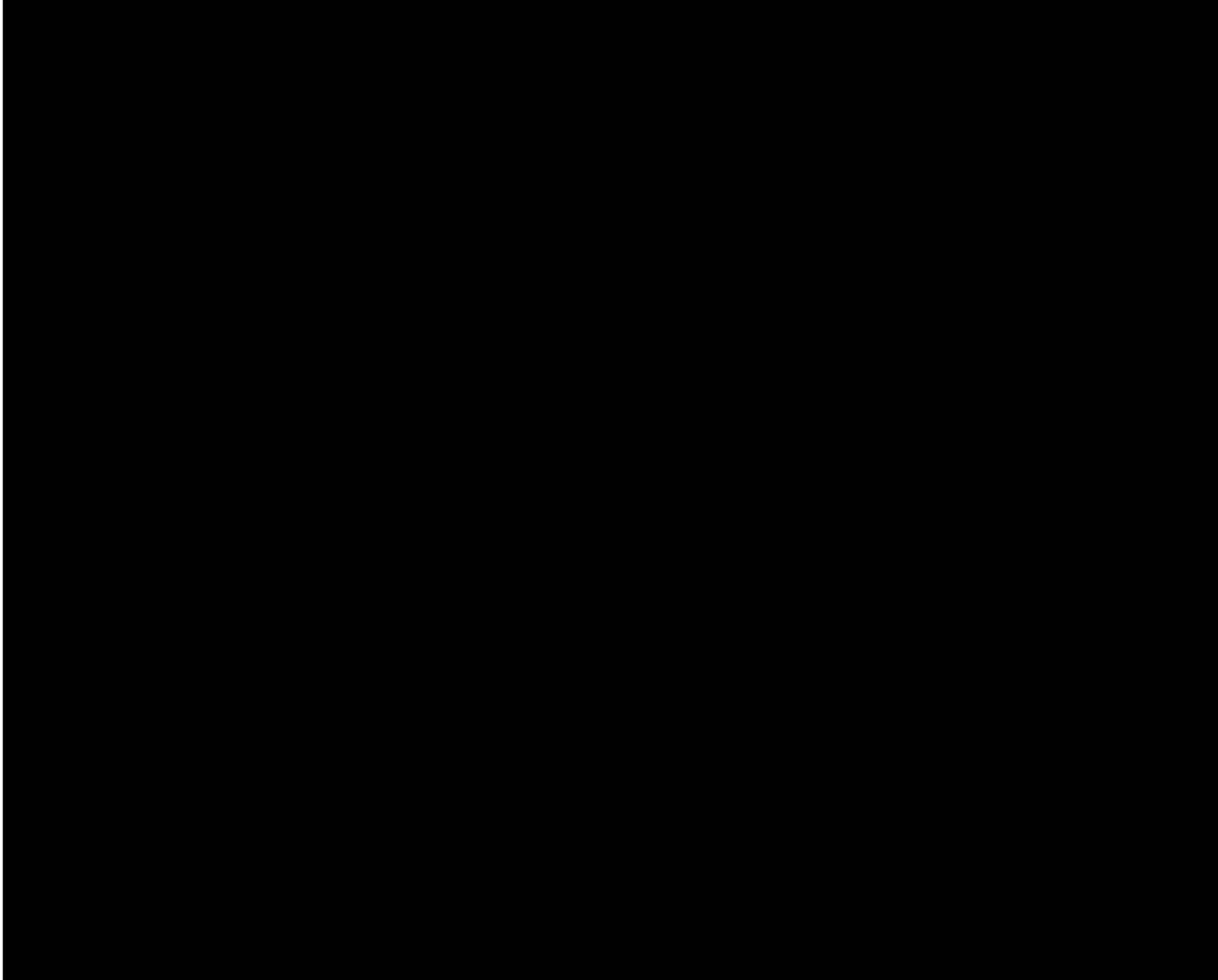
20개의 동일한 동전이 있다. 두 player A와 B가 순서를 정해 번갈아 가며 이 중 하나 혹은 두 개의 동전을 가져간다. 마지막 동전을 가져간 사람이 진다. A가 먼저 시작한다면, A가 이길 수 있는 전략이 존재하는가?





# Chess Player - Kasparov in Russia

3





# Chess Player - Kasparov in Russia

4

Born in 1963

Youngest World Champion in 1984



World champion for ~20 years until 2005, when he retired

**But he once lost a series of games in 1997,  
by WHAT?**



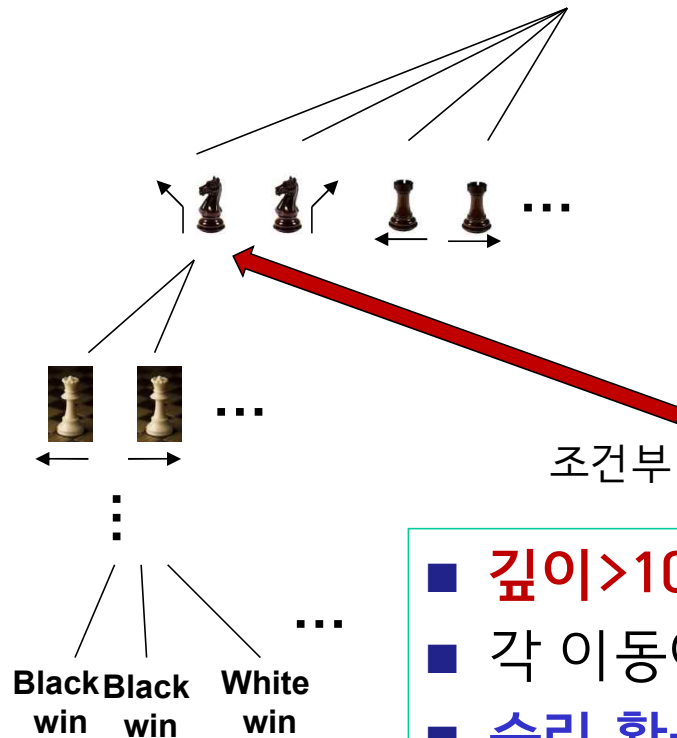
# *Deep Blue* by IBM - Chess Game Machine

5





## 게임 트리 + 승리할 수 있는 (조건부) 확률



- 깊이 > 100 인 트리 구성하고
- 각 이동에 따른 승리 확률을 계산한 후
- 승리 확률 가장 높은 이동 선택



- 많은 게임에서 승리 전략을 찾기 위해 **트리** 활용
- 또한 트리는 게임 에서와 유사하게
- 목표 주어졌을 때 최적의 해결 방법을 찾기 위해서도 활용됨

Let us have a look at the cases  
where **TREEs** are used for **GAMEs**

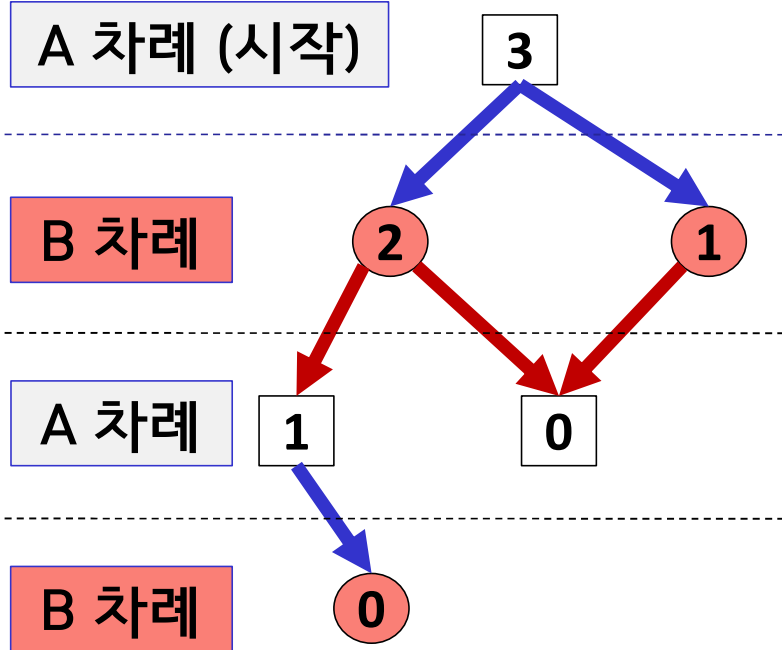






## 게임 트리를 구성하고 승리 전략을 도출하는 과정

9



① (작은 크기의 게임에서 시작)  
모든 **가능성** 나타내는  
**트리** 구성



## 게임 트리를 구성하고 승리 전략을 도출하는 과정

10

A 차례 (시작)

3

B 차례

2

1

A 차례

1

0

B 차례

0

① (작은 크기의 게임에서 시작)  
모든 **가능성** 나타내는  
**트리** 구성

A 차례 (시작)

3

B 차례

2

1

A 차례

1

0

B 차례

0

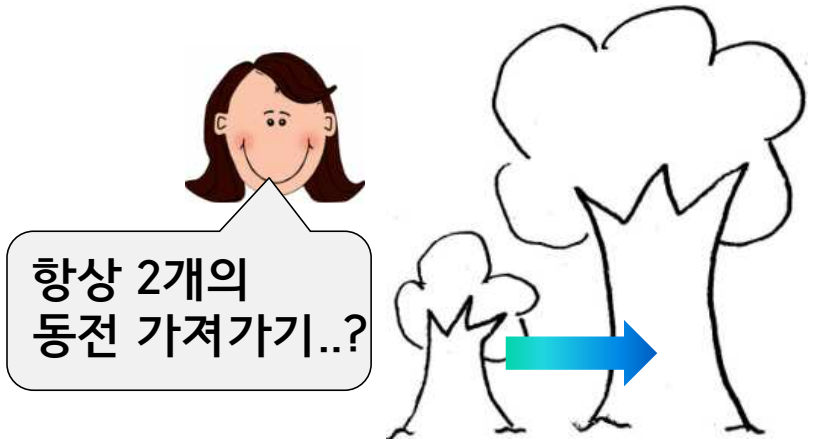
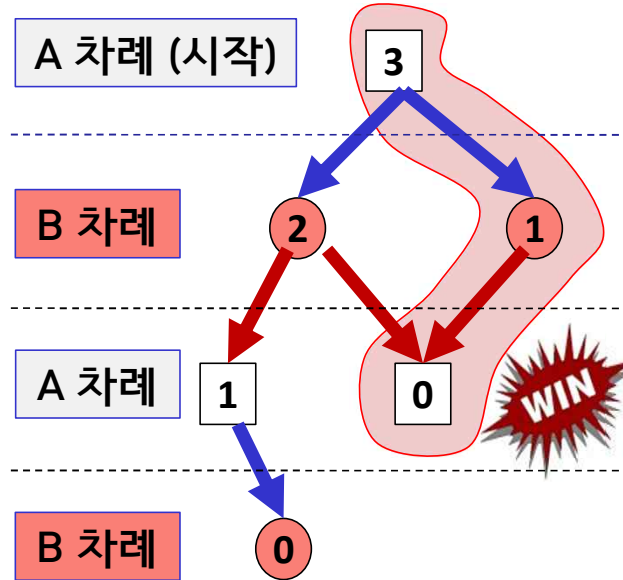
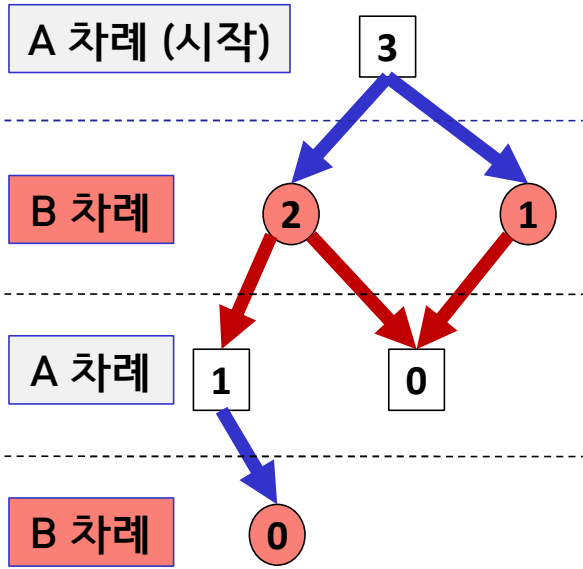


② **이기는 상태로 가는 경로**  
**찾기**



# 게임 트리를 구성하고 승리 전략을 도출하는 과정

11



① (작은 크기의 게임에서 시작)  
모든 **가능성** 나타내는  
**트리** 구성

② **이기는 상태로 가는 경로**  
찾기

③ **임의 크기의 게임에 대한**  
**일반적인 승리 전략 도출**

①~②는 컴퓨터의 도움으로  
빠르게 진행 가능하며, ③은  
사람이 개입하는 경우 많음

served.



## ■ 예제 게임을 통해 트리 활용 과정을 익혀보자.

- ① 게임 트리 구성
- ② 승리 전략 도출
- ③ 작은  $N$ 에서 시작해 승리 전략을 도출한 후 이를 임의의  $N$ 에 대해 일반화

먼저 작은  $N$  값에 대해 1~2회 직접 게임을 해보며 규칙에 익숙해져 보겠습니다.

한 번에 **동전 하나 혹은 두 개 가져가기**: 승리할 수 있는 전략이 있을까?



$N$ 개의 동일한 동전이 있다. 두 player A와 B가 순서를 정해 번갈아 가며 이 중 하나 혹은 두 개의 동전을 가져간다. **마지막 동전을 가져간 사람이 진다.** A가 먼저 시작한다면, A가 이길 수 있는 전략이 존재하는가?



**N**개의 동일한 동전이 있다. 두 player A와 B가 순서를 정해 번갈아 가며 이 중 **하나 혹은 두 개의 동전을 가져간다**. 마지막 동전을 가져간 사람이 진다. A가 먼저 시작한다면, A가 이길 수 있는 전략이 존재하는가?

13

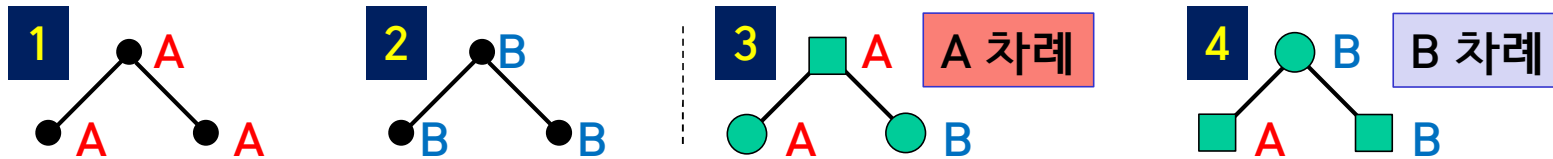
먼저 작은 N 값에 대해  
1~2회 직접 게임을 해보며  
규칙에 익숙해져  
보겠습니다.



① 모든 가능한 경우 나타내는 트리 만들기

② 잎(leaf)에서 시작해 위쪽으로 가며 각 노드에 승자 표기

● P : player P가 이 경로를 따라 내려가면 승리함을 의미



③ 승리 전략 찾기: root에서 leaf까지 모든 노드가 P로 표기된 경로가 있다면, P가 이 경로를 따르도록 게임을 운영하면 승리함

④ ③에서 구한 전략을 일반화하여 임의의 N 값에 대한 승리 전략 도출



## ① 모든 가능한 경우 나타내는 트리 만들기 (N=5)

15

같은 player 차례이며  
남은 동전 수 같다면  
같은 상태로 볼 수 있음  
(이 후 진행 가능성 같으므로)

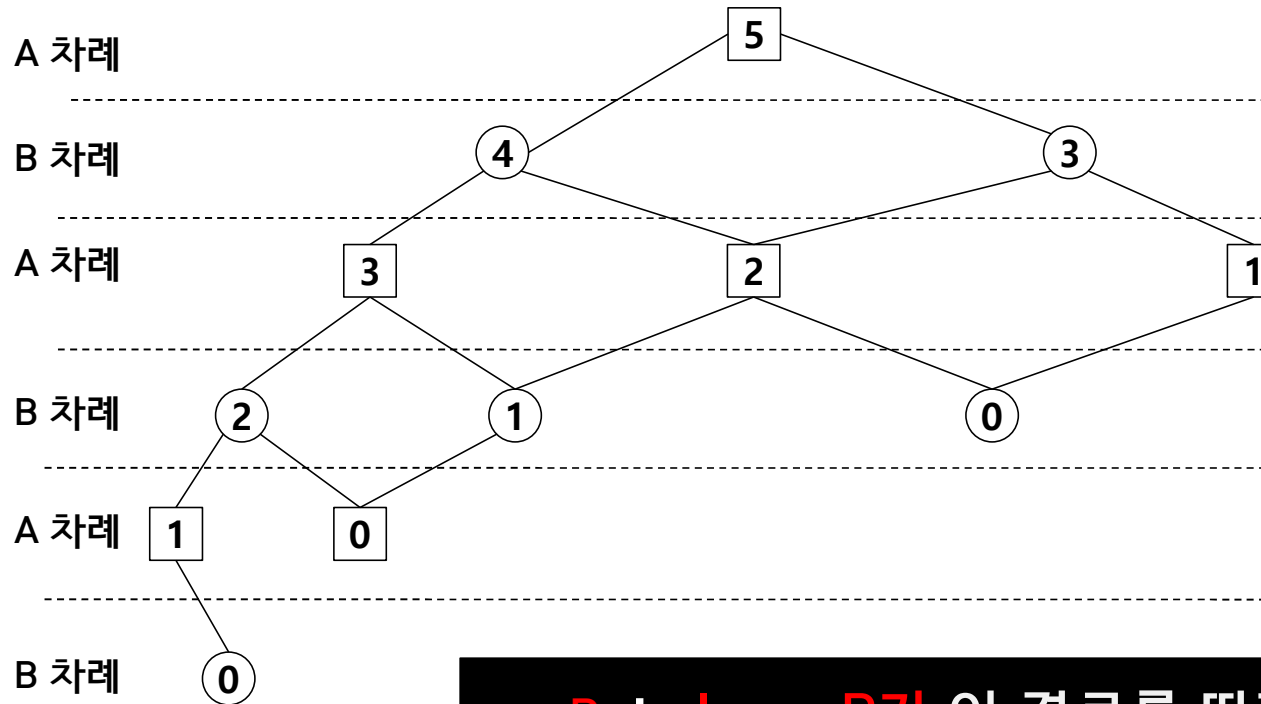
먼저 작은 N=5 값에 대해 승리  
전략을 도출한 후 이를 임의의 N에  
대해 일반화 해 보겠습니다.

5개의 동일한 동전이 있다. 두 player A와 B가 순서를 정해 번갈아 가며 이 중 하나 혹은 두 개의 동전을 가져간다. 마지막 동전을 가져간 사람이 진다. A가 먼저 시작한다면, A가 이길 수 있는 전략이 존재하는가?



## ② 잎(leave)에서 시작해 위쪽으로 가며 각 노드에 승자 표기 (평가 함수 결정)

16



평가 함수: 게임 트리의 각 노드에서 Player가 유리한 정도를 나타내는 값

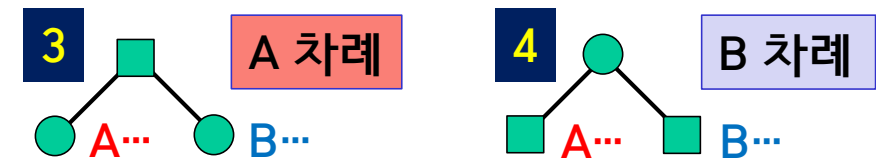
간단한 방법 중 하나는  
**승리**와 **패배**로 구분해  
나타내는 방식  
(1 vs 0 혹은 1 vs -1)

● P : player P가 이 경로를 따라 내려가면 승리함 의미

노드 0의 모든 하위 노드에  
승자 표기를 완료했을 때  
0의 승자 표기



모든 하위 노드의  
승자가 같은 경우



하위 노드의 승자가  
둘 다 가능한 경우

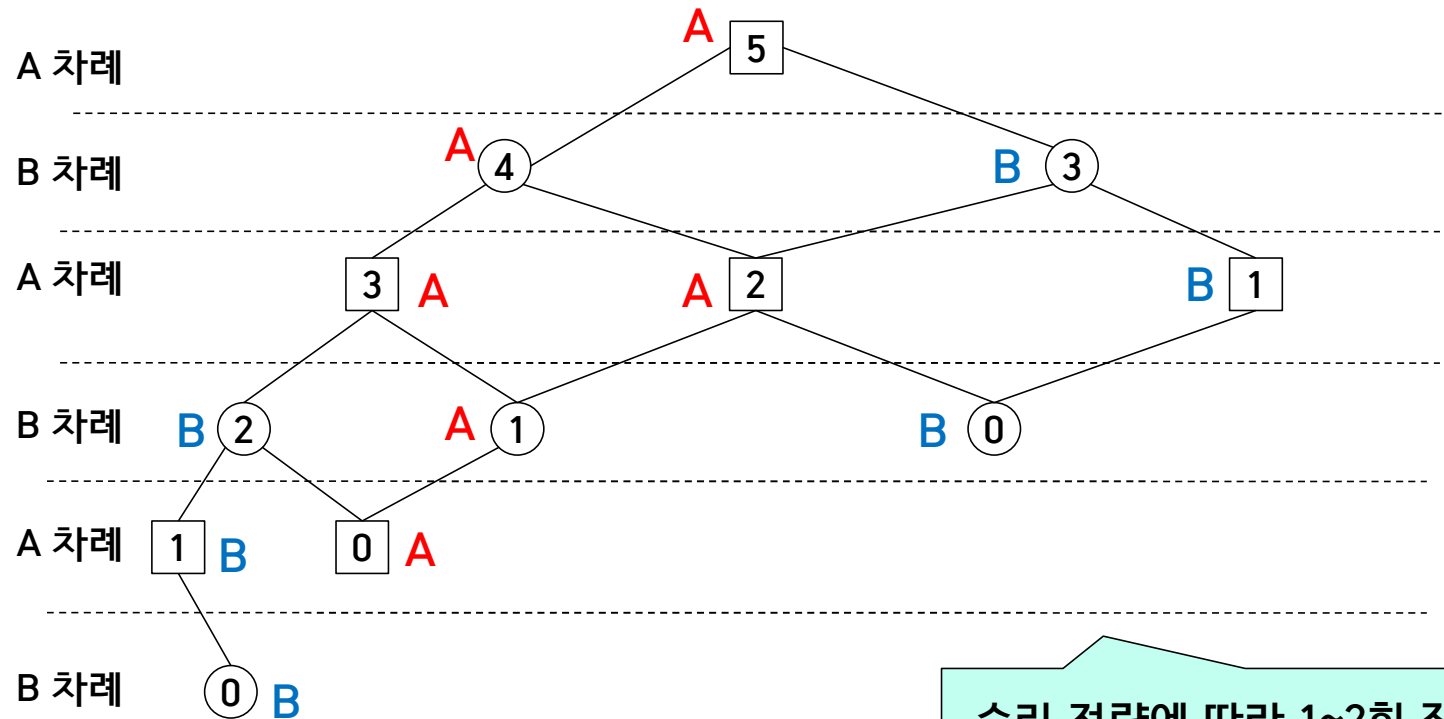
rights reserved.





③ **승리 전략 찾기:** root에서 leaf까지 모든 노드가 P로 표기된 경로가 있다면, P가 이 경로를 따르도록 게임을 운영하면 승리함

17



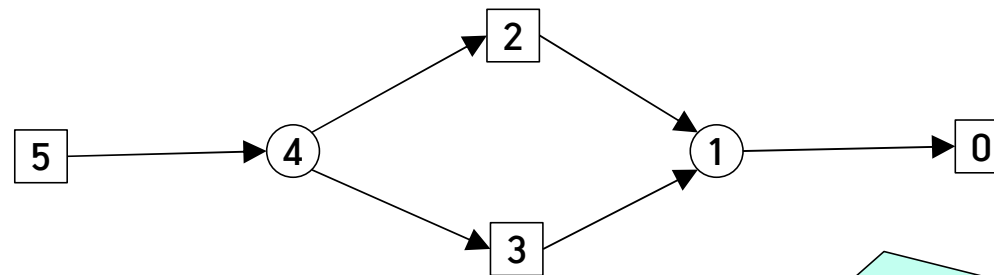
승리 전략에 따라 1~2회 직접 게임을  
해보면 더 이해하기 쉽습니다.



③ **승리 전략 찾기**: root에서 leaf까지 모든 노드가 P로 표기된 경로가 있다면, P가 이 경로를 따르도록 게임을 운영하면 승리함

18

A 차례      B 차례      A 차례      B 차례      A 차례

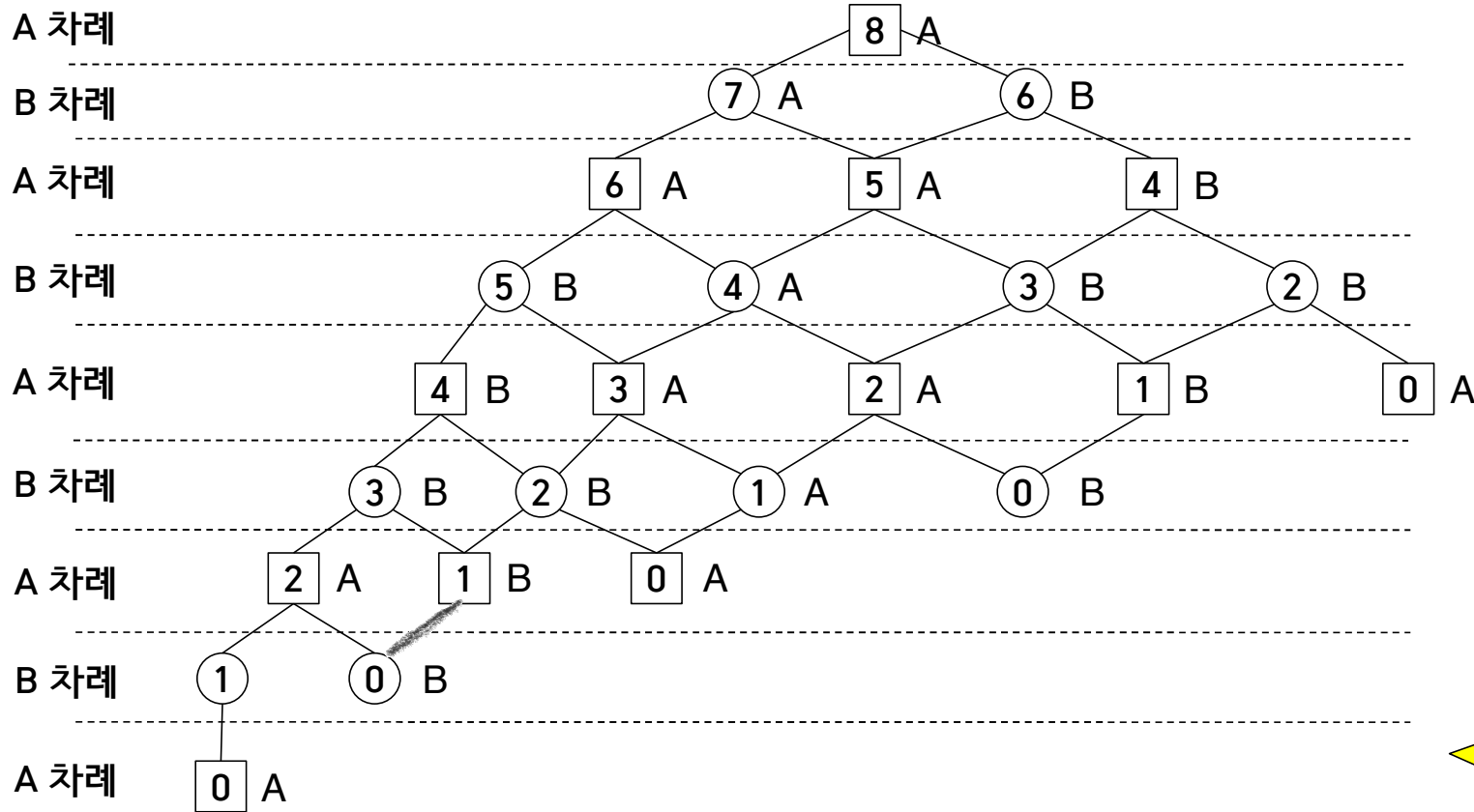


앞에서 발견한 A의 승리 전략을  
(이후에 임의의 N에 대해 일반화할 수  
있도록) 더 잘 이해해 보겠습니다.



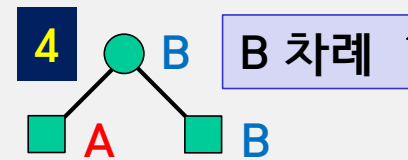
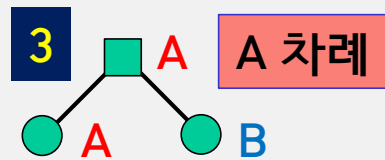
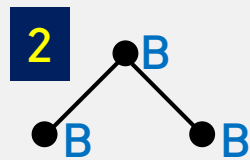
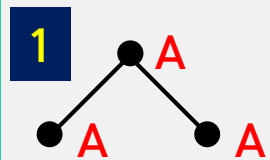
## ② 잎(leave)에서 시작해 위쪽으로 가며 각 노드에 승자 표기 (N=8)

19



종료 조건: 마지막 동전  
가져간 player가 지는 게임

● P : Player P가 이 경로를 따라 내려가면 승리함을 의미



노드 0의 모든 하위 노드에  
승자 표기를 완료했을 때  
0의 승자 표기

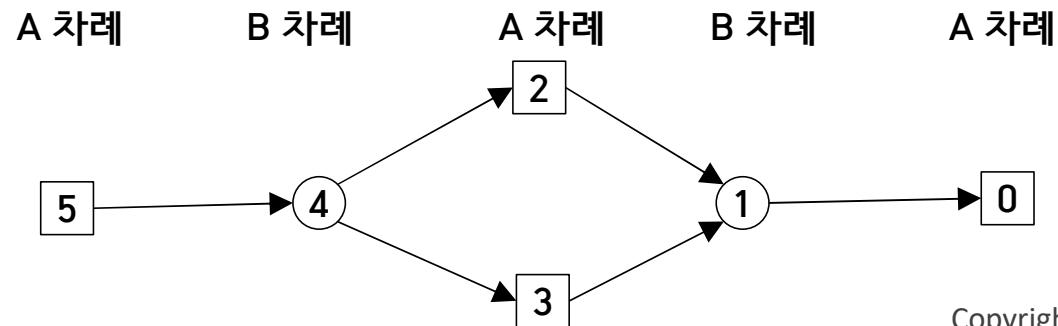
Copyright © by Sinyung Lee - All Rights Reserved.



③ **승리 전략 찾기**: root에서 leaf까지 모든 노드가 P로 표기된 경로가 있다면, P가 이 경로를 따르도록 게임을 운영하면 승리함

A 차례	B 차례	A 차례	B 차례	A 차례	B 차례	A 차례
8	7	6	4	3	1	0
		5		2		

[Q] N=8인 경우와 N=5인 경우의 승리 전략을 비교하고 유사점을 찾아보자.  
이를 임의의 N에 대해 일반화할 수 있겠는가?

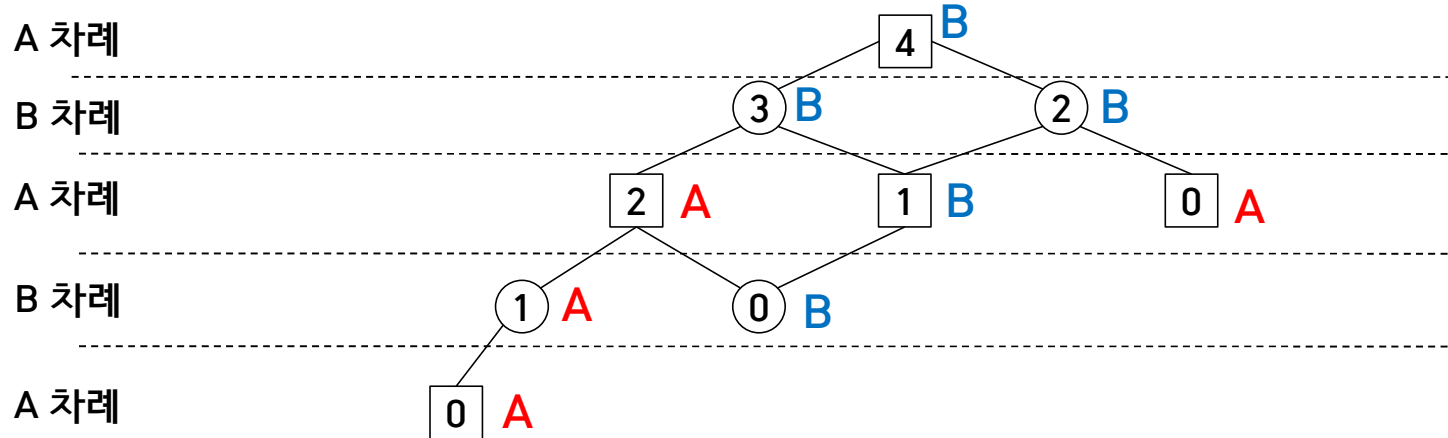




[Q] 앞에서 발견한 승리 전략은 **항상 사용 가능**한가?

예를 들어  $N=4$ 인 경우에도 A가 이길 수 있는 전략이 있는가?

(B도 (동일한 승리 전략을 사용해) 이기기 위해 최선을 다한다고 가정하자)



[Q] A(선공)의 승리 전략을 경우에 따라 분류해 보시오.

내 차례가  $3k+1$ 일 경우, 상대가 실수하고 내 차례가  $(3k, 3k+2)$ 가 올때까지 기다린다



- Push-Pop 게임의 승리 전략도
- 게임 트리 사용해
- 비슷한 과정 통해 찾을 수 있음

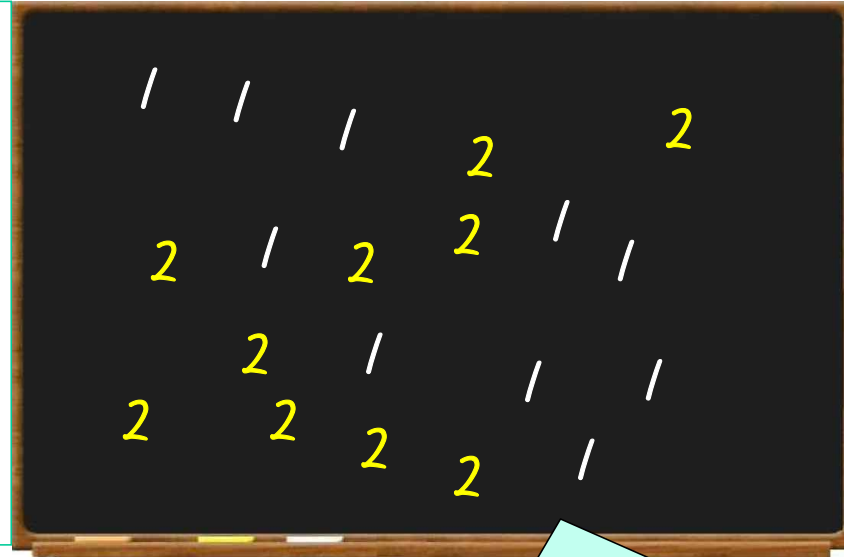




## 두 개의 숫자를 제거하고 한 숫자 추가하기 → 반드시 승리할 전략이 있는가?

23

칠판에 **같은 개수의 1과 2**가 있다. 두 player A와 B가 번갈아 가며 **두 개의 숫자를 제거한 후 아래 규칙에 따라 한 숫자를 추가**한다. 최종적으로 칠판에 남은 숫자가 1이면 A가 승리하고, 2라면 B가 승리한다. A가 먼저 시작한다면(선공) A가 승리할 전략이 있을까?



$\left. \begin{array}{l} /, / \text{ 제거} \\ 2, 2 \text{ 제거} \end{array} \right\} \longrightarrow 2 \text{ 추가}$

$/, 2 \text{ 제거} \longrightarrow / \text{ 추가}$

먼저 작은 N 값에 대해 1~2회 직접 게임을 해보며 규칙에 익숙해져 보겠습니다.



$\left. \begin{array}{l} 1, 1 \text{ 제거} \\ 2, 2 \text{ 제거} \end{array} \right\} \longrightarrow 2 \text{ 추가}$   
 $1, 2 \text{ 제거} \longrightarrow 1 \text{ 추가}$

$\begin{array}{c} 1 \quad 1 \\ \quad 1 \\ 2 \quad 2 \quad 2 \end{array}$

먼저 작은 N 값에 대해  
 1~2회 직접 게임을 해보며  
 규칙에 익숙해져  
 보겠습니다.

칠판에 **같은 개수의 1과 2**가 있다. 두 player A와 B가 번갈아가며 **두 개의 숫자를 제거한 후 아래 규칙에 따라 한 숫자를 추가**한다. 최종적으로 칠판에 남은 숫자가 1이면 A가 승리하고, 2라면 B가 승리한다. A가 먼저 시작한다면 A가 승리할 전략이 있을까?

B 학생







**N=1** (숫자 1과 2 각각 하나씩 **한 쌍**이 있는 경우)


[① 트리 만들기 ② 승자 표기 ③ 승리전략 도출 ④ 일반화] 순서로 진행

25

/ 2

/, / 제거

2, 2 제거

}  2 추가

/, 2 제거

 / 추가

- 작은 N 값에서 시작해 트리를 그려본 후, 이로부터 얻은 승리 전략을 임의의 N에 대해 일반화
- 트리를 그릴 때 가능하면 **같은 의미의 노드를 중복해서 그리지 않도록** 해보자.
- 승리 전략을 임의의 N에 대한 일반화하려면 모든 N에 대해 동작함을 **증명**할 수 있어야 함



**N=2** (숫자 1과 2 각각 2개씩 **2쌍**이 있는 경우)

[① 트리 만들기 ② 승자 표기 ③ 승리전략 도출 ④ 일반화] 순서로 진행

26

/ 2  
/ 2

/, / 제거

2, 2 제거

} **→** 2 추가

/, 2 제거

**→** / 추가

- 작은 N 값에서 시작해 트리를 그려본 후, 이로부터 얻은 승리 전략을 임의의 N에 대해 일반화
- 트리를 그릴 때 가능하면 **같은 의미의 노드를 중복해서 그리지 않도록** 해보자.
- 승리 전략을 임의의 N에 대한 일반화하려면 모든 N에 대해 동작함을 **증명**할 수 있어야 함



**N=3** (숫자 1과 2 각각 3개씩 **3쌍**이 있는 경우)

[① 트리 만들기 ② 승자 표기 ③ 승리전략 도출 ④ 일반화] 순서로 진행

27

/ 2  
/ 2  
/ 2

/, / 제거

2, 2 제거

}



2 추가

/, 2 제거



/ 추가



/ 2  
/ 2  
/ 2  
/ 2

/, / 제거 }  
2, 2 제거 } → 2 추가

/, 2 제거 → / 추가

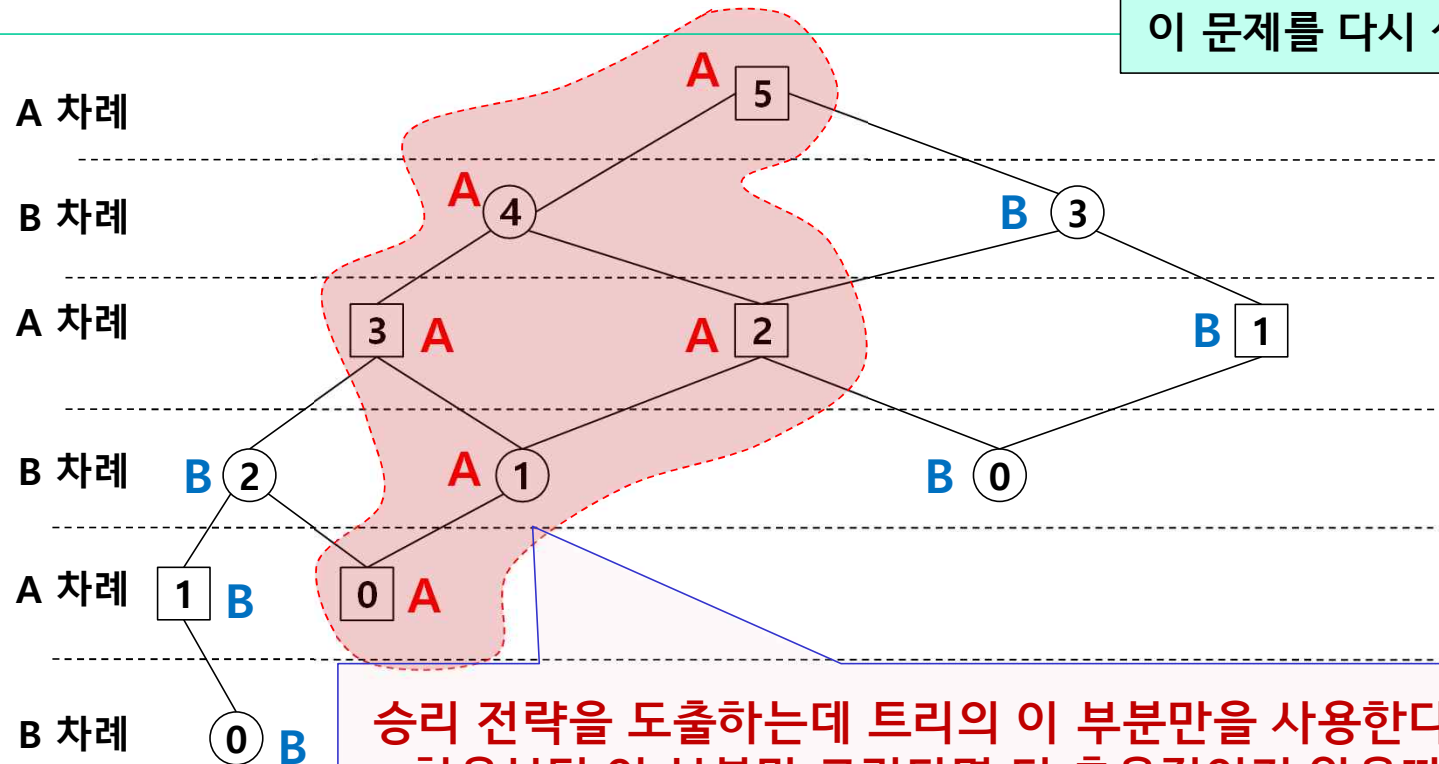


## 더 빠르게 게임 트리를 구성하고 승리 전략을 도출하는 방법

29

N개의 동일한 동전이 있다. 두 player A와 B가 순서를 정해 번갈아가며 이 중 하나 혹은 두 개의 동전을 가져간다. 마지막 동전을 가져간 사람이 진다. A가 먼저 시작한다면, A가 이길 수 있는 전략이 존재하는가?

이 문제를 다시 생각해 보자.



승리 전략을 도출하는데 트리의 이 부분만을 사용한다면, 처음부터 이 부분만 그린다면 더 효율적이지 않을까?

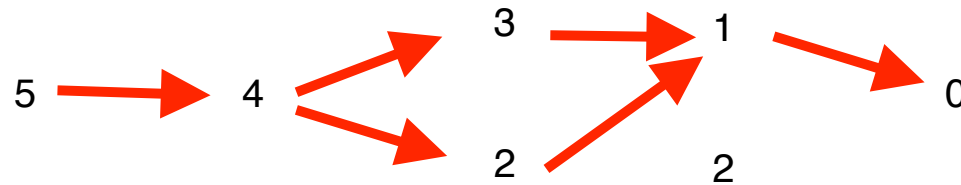


## 더 빠르게 게임 트리를 구성하고 승리 전략을 도출하는 방법

30

5개의 동일한 동전이 있다. 두 player A와 B가 순서를 정해 번갈아 가며 이 중 하나 혹은 두 개의 동전을 가져간다. 마지막 동전을 가져간 사람이 진다. A가 먼저 시작한다면, **A의 승리 전략**은?

A 차례      B 차례      A 차례      B 차례      A 차례



(A의 승리 전략을 찾는다고 가정)

- ① A가 **승리하는 상황(leaf)에서 시작** → 트리 위쪽으로 거슬러 올라가 시작 상황까지 **거꾸로 그려 가기**
- ② **B 차례: 모든 자식 노드가 A가 승리하는 노드로 가는 경우만 포함** (일부 자식이라도 B가 승리하면 포함하지 않음)  
A 차례: 마지막에 그린 B차례의 노드로 갈 수 있는 모든 노드를 포함 (일부 자식이 B가 승리해도 포함)



## 매 턴마다 $\frac{1}{2}$ 이하의 구슬 가져가기 → 승리 전략이 존재하는가?

31

즉 N개 구슬이 남아  
있다면  
 $1 \sim \max(1, \text{floor}(N/2))$   
개의 구슬 가져가야 함

상자에 100개의 구슬이 있다. 두 player A와 B가  
번갈아 가며 **현재 상자에 있는 구슬의 반 이하 ( $\leq \frac{1}{2}$ )를  
가져간다.** (예: 현재 상자에 30개의 구슬이 남았다면  
1개~15개를 가져갈 수 있음) 만약 상자에 **구슬 하나만  
남았다면 이를 반드시 가져가야 하며, 마지막 구슬을  
가져간 player가 승리**한다. A가 먼저 시작한 경우, A가  
승리할 수 있는 전략을 찾아 보시오.



먼저 작은 N 값에 대해 1~2회  
직접 게임을 해보며 규칙에  
익숙해져 보겠습니다.



먼저 작은 N 값에 대해  
1~2회 직접 게임을 해보며  
규칙에 익숙해져  
보겠습니다.

상자에 100개의 구슬이 있다. 두 player A와 B가 번갈아 가며 **현재 상자에 있는 구슬의 반 이하 ( $\leq \frac{1}{2}$ )를 가져간다**. (예: 현재 상자에 30개의 구슬이 남았다면 1개~15개를 가져갈 수 있음) 만약 상자에 **구슬 하나만 남았다면 이를 반드시 가져가야 하며, 마지막 구슬을 가져간 player가 승리**한다. A가 먼저 시작한 경우, A의 승리 전략을 찾아 보시오.





상자에 100개의 구슬이 있다. 두 player A와 B가 번갈아 가며 **현재 상자에 있는 구슬의 반 이하 ( $\leq \frac{1}{2}$ )를 가져간다.** (예: 현재 상자에 30개의 구슬이 남았다면 1개~15개를 가져갈 수 있음) 만약 상자에 **구슬 하나만 남았다면 이를 반드시 가져가야 하며, 마지막 구슬을 가져간 player가 승리**한다. A가 먼저 시작한 경우, A의 승리 전략을 찾아 보시오.

[Q] 루트에서 시작해 전체 트리를 다 그린다면 몇 개의 노드를 그려야 할까? 시간/공간 복잡도는 이 노드 수에 비례한다.



상자에 100개의 구슬이 있다. 두 player A와 B가 번갈아 가며 **현재 상자에 있는 구슬의 반 이하 ( $\leq \frac{1}{2}$ )를 가져간다.** (예: 현재 상자에 30개의 구슬이 남았다면 1개~15개를 가져갈 수 있음) 만약 상자에 **구슬 하나만 남았다면 이를 반드시 가져가야 하며, 마지막 구슬을 가져간 player가 승리**한다. A가 먼저 시작한 경우, A의 승리 전략을 찾아 보시오.

A	B	A	B	A	B	A	B	A	B	A	B	A
100		94		46		22		10		4		
	95		47		23		11		5		2	1
96		48		24		12		6		3		

$$3 \times 2^{(i-1)} - 1$$

(A의 승리 전략을 찾는다고 가정)

- ① A가 **승리하는 상황(leaf)에서 시작** → 트리 위쪽으로 거슬러 올라가 시작 상황까지 **거꾸로 그려 가기**
- ② **B 차례: 모든 자식 노드가 A가 승리하는 노드로 가는 경우만 포함** (일부 자식이라도 B가 승리하면 포함하지 않음)  
A 차례: 마지막에 그린 B차례의 노드로 갈 수 있는 모든 노드를 포함 (일부 자식이 B가 승리해도 포함)

## 지금까지 배운 내용 정리: 게임 트리의 활용도와 기본 구성 방법

- 게임 트리는 player의 선택에 따라 이기거나 질 수 있는 게임에서 **승리하기 위한 전략을 도출**하는데 도움이 된다.
- 게임 트리를 사용하여 승리 전략을 도출하다 보면 게임의 **규칙 상의 문제점을 발견할 수도 있다**. 예를 들어 규칙이 잘못 되었다면 한 player가 절대 이길 수 없거나 항상 이기는 경우도 발생하며, 게임 트리를 사용하면 이러한 문제도 파악할 수 있다.
- **더 빠르게** 게임 트리를 만들기 위해 모든 가능성을 그리는 대신 **승리 전략에 해당하는 부분만 그릴 수 있다**. 이를 위해서는 승리하는 노드(leaf)에서 root 방향으로 거꾸로 트리를 그려간다.



[Q] **종료 조건**을 아래와 같이 바꾸었을 때 승리 전략을 도출해 보자.

36

한 번에 **동전 하나 혹은 두 개 가져가기**: 승리할 수 있는 전략이 있을까?



**N**개의 동일한 동전이 있다. 두 player A와 B가 순서를 정해 번갈아 가며 이 중 하나 혹은 두 개의 동전을 가져간다. **마지막 동전을 가져간 사람이 이긴다**. A가 먼저 시작한다면, A가 이길 수 있는 전략이 존재하는가?

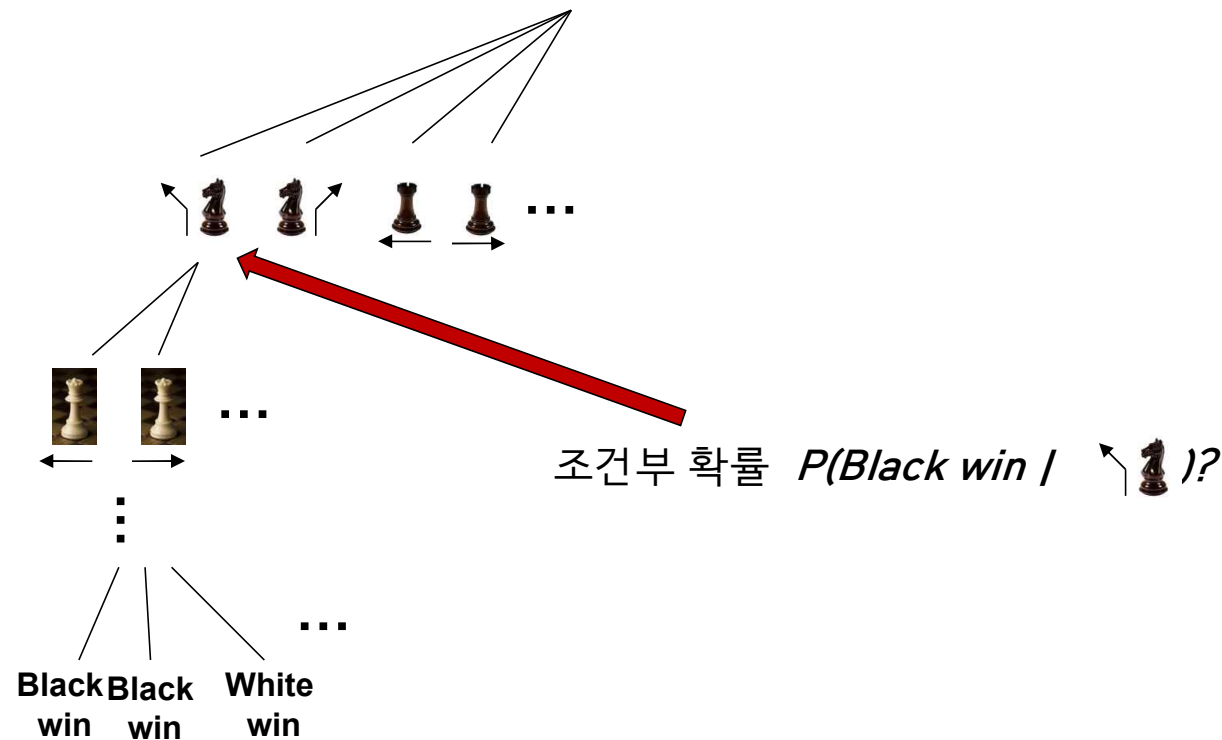
Hint: 종료 시점부터 시작 시점 방향으로 선공이 승리하는 부분만을 그려 승리 전략을 이끌어내 보세요. 특히  $N = 3k$ ,  $3k+1$ ,  $3k+2$ 인 경우 각각에 대한 승리 전략을 구분해 기술해 보세요.



[Q] 칠판에 두 숫자 20과 36이 있다. 두 player A와 B가 번갈아 가며 칠판에 있는 임의의 두 숫자의 차를 쓴다. ① 두 숫자의 차는 반드시 양수 형태로 써야 하며 (즉 큰 수 - 작은 수를 써야 함), ② 이미 칠판에 있는 수를 써서는 안 된다. 자기 차례에 더는 새로운 차를 쓸 수 없게 된 player가 진다. A가 먼저 시작했을 때 A의 승리 전략이 있는가?

Hint: 이 게임은 종료 시점부터 트리를 그려가기 쉽지 않으므로 시작 시점부터 전체를 그려 보세요. 11, 12, 22를 제거하는 게임과 유사한 결론을 얻을 수 있습니다. 이러한 결론이 나오는 '이유'를 설명해 보세요.

(Q) 체스나 바둑처럼 게임 트리 크기가 방대해 트리 전체를 그려 메모리에 저장하기 어렵고, 종료 시점부터 거꾸로 승리하는 부분만 그리기도 어렵다면 & 100% 승리하는 전략이 존재하지 않는다면 어떻게 승리 전략을 찾아야 하나?





## 게임 트리 개요와 확률적 탐색

게임 트리를 구성하고 승리 전략을 도출하는 기본 방법 이해

01. 퀴즈 풀이 & 예습 내용 복습

02. 게임 트리 구성 & annotation 방법

03. 구성한 게임 트리로부터 승리전략을 도출하는 방법

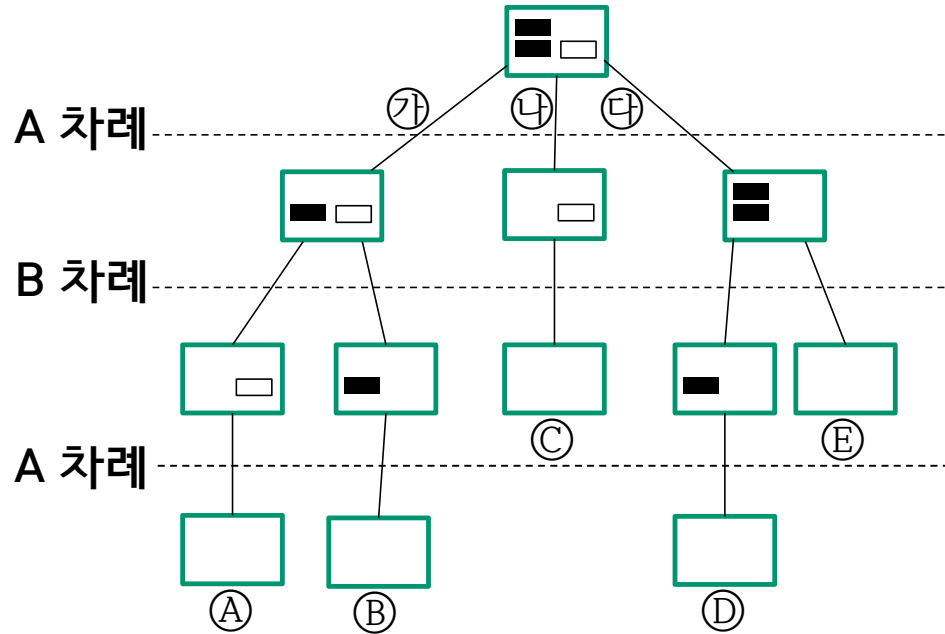
게임의 여러 특성 분석을  
위해 트리를 활용하는 예

04. 게임 트리 관련 다양한 문제 풀이 (게임 규칙의 문제점 파악)

05. 더 빠르게 게임 트리를 구성하고 승리 전략을 도출하는 방법

## 06. 게임 트리의 구현과 확률적 탐색

07. 실습문제 풀이

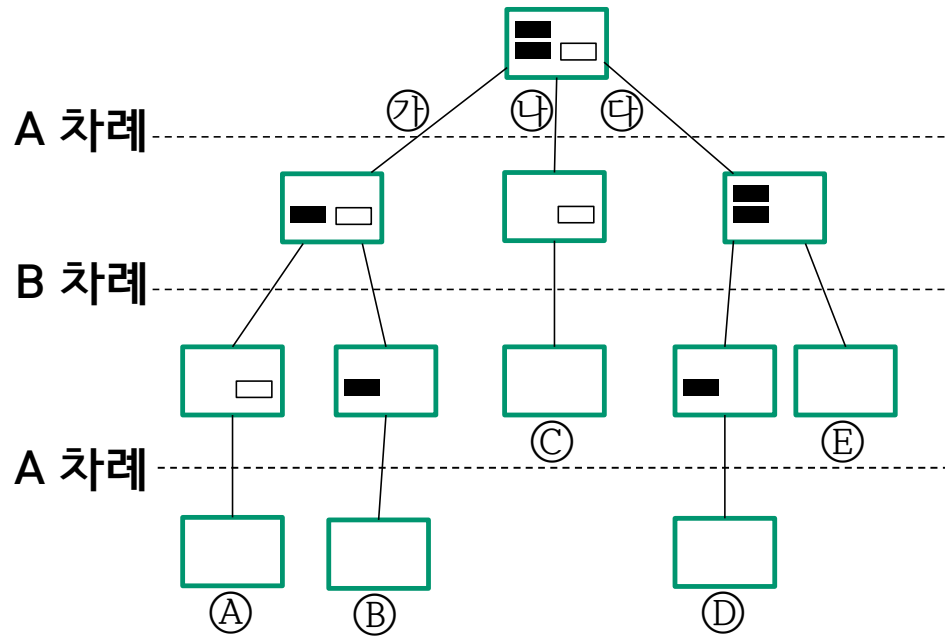


- 이 장에서 예제로 사용할 게임 규칙 이해
- 검은 동전  $b$ 개와 흰 동전  $w$ 개가 있다. 두 player A와 B가 순서를 정해 번갈아 가며 한 번에 하나 혹은 두 개의 동전을 가져간다. 단 **같은 색깔의 동전만 가져갈 수 있다**. 마지막 동전을 가져간 사람이 진다. A가 먼저 시작할 때 **선공 A가 이길 수 있는 전략**을 찾고자 한다.

[Q] 검은 동전 2개, 흰 동전 1개로 게임을 시작한 예가 위 트리에 있다. 이 트리를 보며 게임을 이해하고, 또한 모든 가능한 경우가 그려졌음을 확인하시오.

보기 쉽도록 같은 상태를 병합하지는 않았습니다.



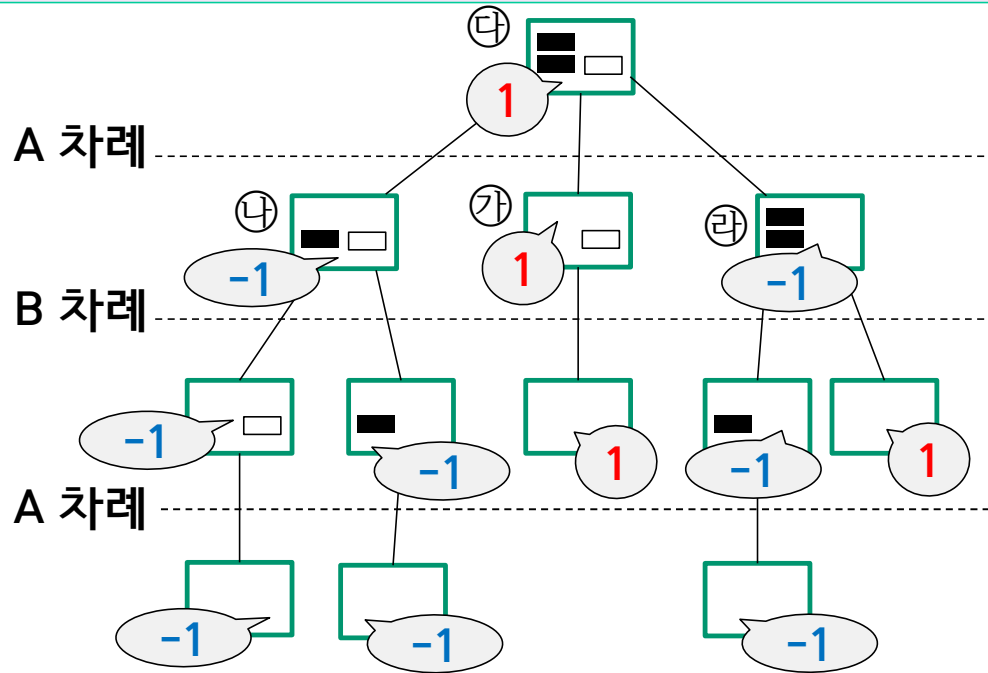


[Q] 이 게임의 player는 자기 차례가 오면 4가지의 행동 중 하나를 할 수 있다. 즉 '흰색 1개', '흰색 2개', '검은색 1개', 또는 '검은색 2개'를 가져갈 수 있다. 하지만 왼쪽 트리에서 A가 가장 먼저 한 일을 보면 ㉠㉡㉢ 3가지이다. 왜 4가지가 아니라 3가지인가?

[Q] '마지막 동전을 가져간 사람이 진다'라는 규칙에 따르면 모든 동전을 가져가서 게임이 끝난 시점인 ㉠㉡㉢㉣㉤ 각각에서 승자는 누구인가? A→B→A→B→... 차례로 게임을 진행한다.

## 평가함수 $f(n)$ : 노드 $n$ 에서 승리 전략 찾는 player가 유리한 정도

42

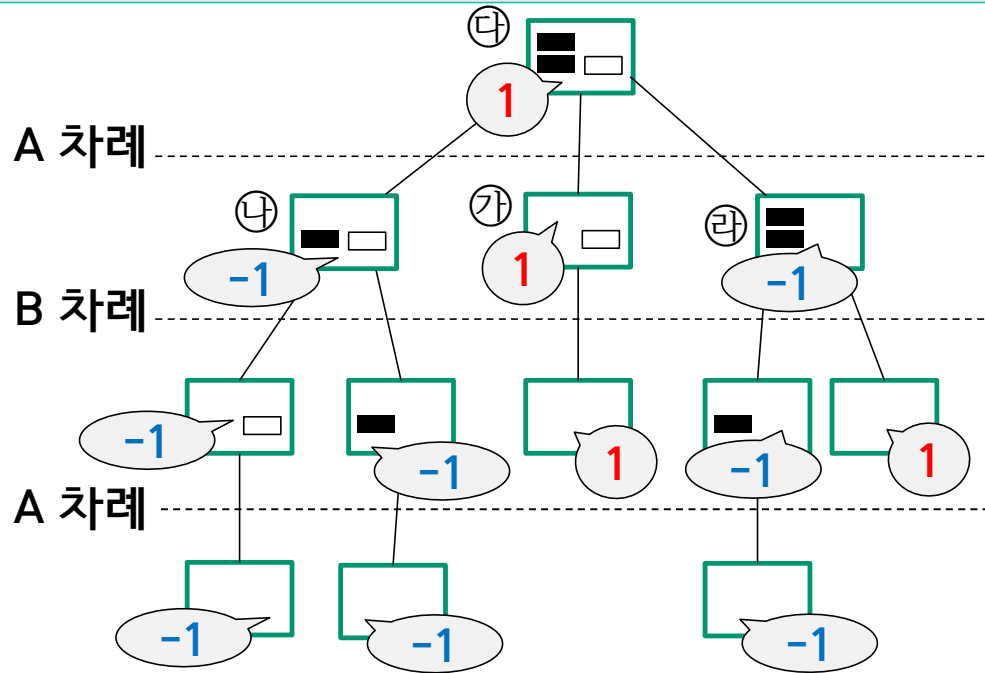


- 평가 함수 참조해 현재 상태에서 player가 승리하기 위한 가장 적절한 행동 선택하며 게임 진행

[Q] 위 트리에서는 각 노드마다 평가 함수 값으로 1 or -1의 값을 부여하였다. Root에서 게임을 시작하되, 평가 함수가 가장 높은 자식 노드 쪽으로 게임을 진행해 간다면 어떤 경로를 따라 게임을 진행하게 되는가?

# 평가함수 부여방식 1: 승자를 1, -1로 구분해 표기. min-max

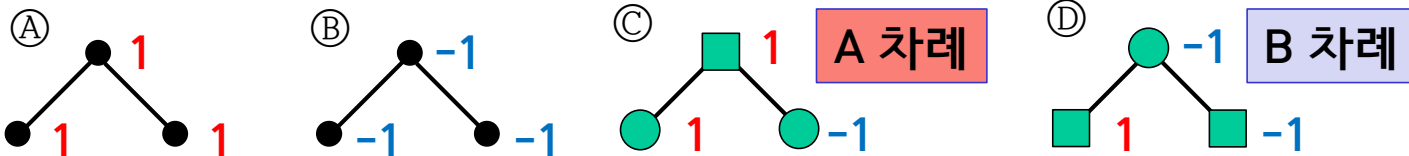
43



[Q] 왼쪽 점수 부여 방식 ㉠~㉡는 min(최솟값)과 max(최댓값) 함수를 사용해 2가지 경우로 요약해 나타낼 수 있다. 이 식의 의미를 이해해 보시오. p는 부모 노드(parent), ci는 p의 i번째 자식노드(children), k는 자식노드의 수를 의미하며, f(n)은 노드 n에 부여한 점수를 의미한다.

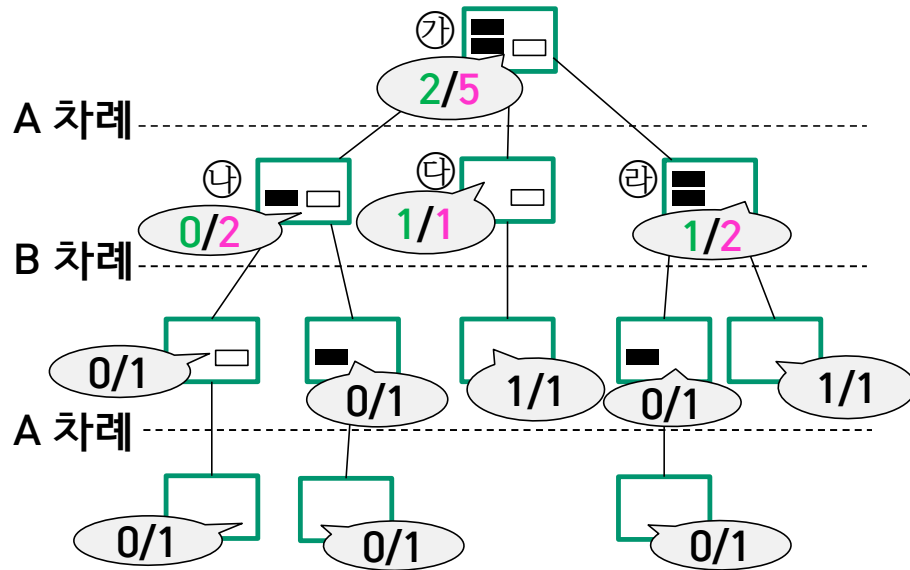
- p에서 A가 행동할 차례라면,  $f(p) = \max(f(c1), f(c2), \dots, f(ck))$

B가 행동할 차례라면  $f(p) = \min(f(c1), f(c2), \dots, f(ck))$



## 평가함수 부여방식 2: 승률 $\text{win}(n) / (\text{win}(n) + \text{draw}(n) + \text{loss}(n))$

44



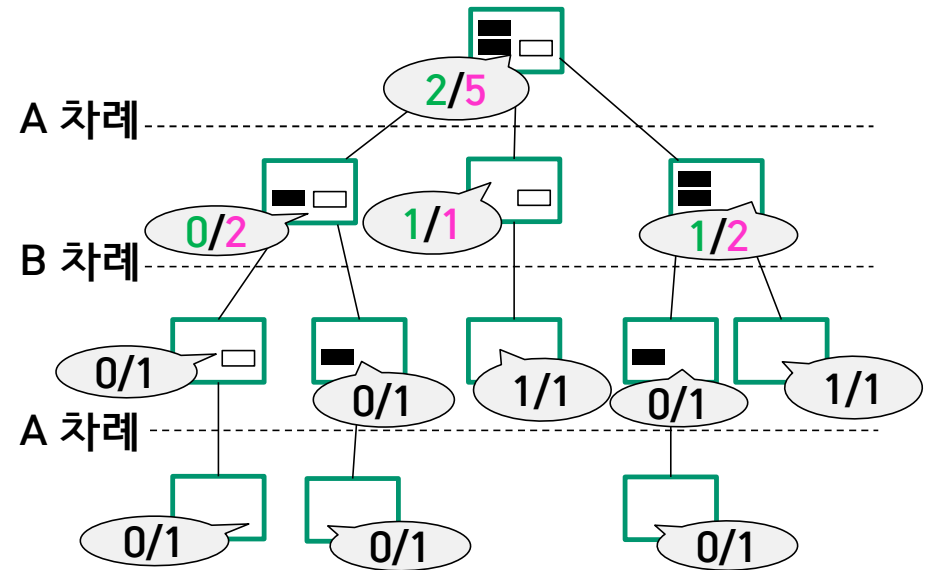
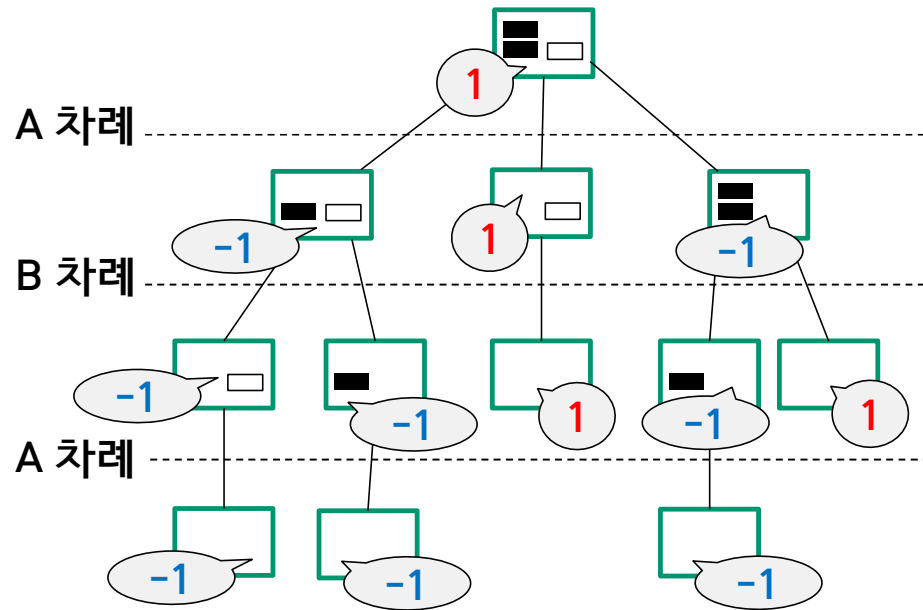
[Q] 왼쪽 예에서 각 노드에 부여된 점수가 어떤 의미인지 생각해 보시오. 특히 root인 ㉠에 부여한 점수에서 분모와 분자는 어떻게 결정된 것일까?

[Q] 왼쪽과 같이 점수 부여가 끝났다면 1/-1 방식과 마찬가지로  $f(n)$ 이 가장 큰 자식노드를 선택하는 방식으로 게임을 진행한다. 현재 ㉠에 있다면, 자식노드인 ㉡㉢㉣ 중 어디로 진행함을 의미하는가?

- $\text{win}(p) = \text{sum}(\text{win}(c1), \text{win}(c2), \dots, \text{win}(ck))$  : 노드  $n$  아래로 진행했을 때 A가 승리하는 경우의 수
- $p$ 는 부모 노드(parent),  $c_i$ 는  $p$ 의  $i$ 번째 자식노드(children),  $k$ 는 자식노드의 수를 의미
- $\text{loss}(p) = \text{sum}(\text{loss}(c1), \text{loss}(c2), \dots, \text{loss}(ck))$ : 노드  $n$  아래로 진행했을 때 A가 패배하는 경우의 수
- $f(n) = \text{win}(n) / (\text{win}(n) + \text{loss}(n) + \text{draw}(n))$  #  $\text{draw}(n)$ : 비기는 경우의 수

# 평가 함수 1~2의 비교: 각 방식의 장단점은?

45

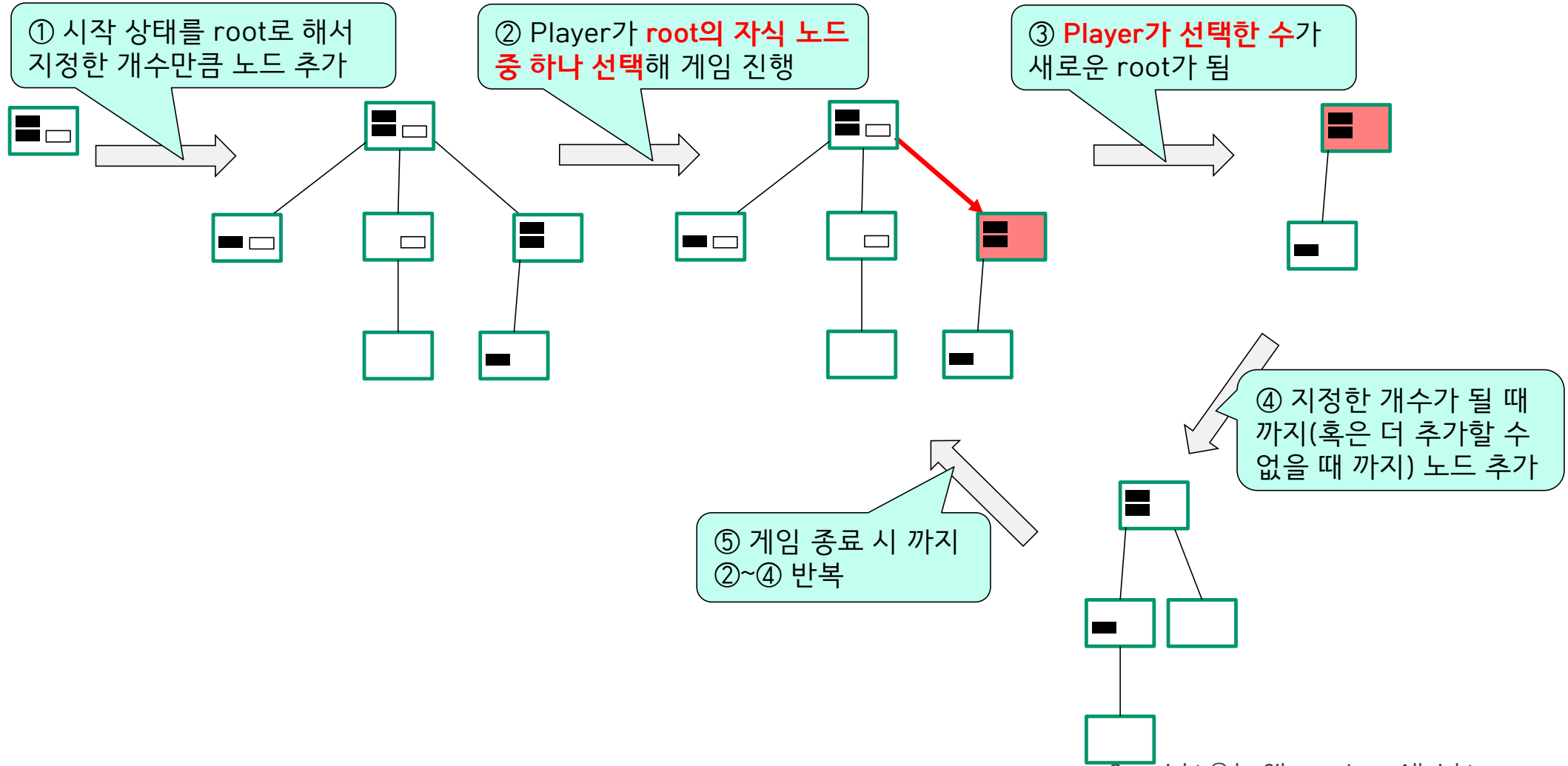




- 트리의 크기를 제한하기 위한 해결책
- 중복된 노드 병합: 가능한 경우의 수(예: 2x2 바둑판의 경우  $4 \times 3 \times 2 \times 1$ )에서 곱하는 각 수를 상수 배만큼 줄일 수 있으나, 곱하는 횟수를 줄이지는 못함
- 확률적 탐색: 트리의 크기를 지정한 수  $k$ 개 노드 이하로 제한

## 확률적 탐색: Monte-Carlo 시뮬레이션

지정한 개수  $k$ 개  $\geq$ 의 노드만 만들어보고 게임 진행 (즉, 트리 일부만 만들어 봄)



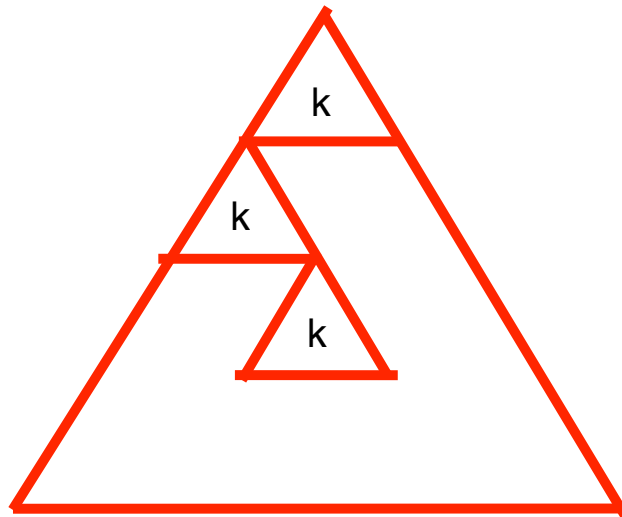
## 확률적 탐색: Monte-Carlo 시뮬레이션

지정한 개수  $k$ 개  $\geq$ 의 노드만 만들어보고 게임 진행 (즉, 트리 일부만 만들어 봄)

[Q] 지금까지 본 Monte-Carlo 방법에 따라 게임을 진행할 때, 트리에 속한 노드 수에 대한 설명으로 올바른 것은?

트리에 속한 노드 수는 항상 같은 값으로 일정하게 유지된다.

트리에 속한 노드 수는 항상 같은 값을 갖지는 않지만 지정한 최댓값 이하로 유지된다.





## k개의 노드를 어떻게 선정할 것인가?

49

- 트리 일부만 ( $\leq k$ 개)만들기 때문에 전체를 만드는 것만큼 정확한 정보를 주지는 못함
- 따라서 전체를 만들 때와 최대한 근접한 결과 줄 수 있는 **일부를 잘 선정**해야 함

**si**: 트리에서 노드  $n_i$  아래 경우를 simulation 해본 횟수

**sp**: 노드  $n_i$ 의 parent의 **si** 값

**wi**: **si**번의 simulation 중 (승리 전략을 찾고자 하는 player가) 승리한 횟수

**c**: 상수이며, 보통  $\sqrt{2}$  사용

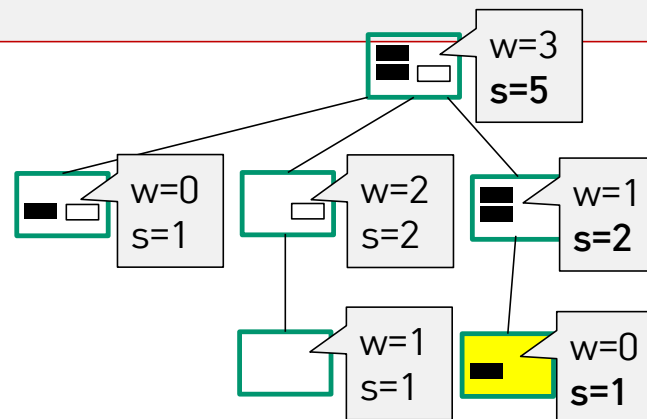
```
while( root node의 si < k ) {
```

승리 전략 찾는데 가장 도움될 것으로 예측되는 노드  $n_i$  선정해 트리에 추가

$n_i$  아래로 종료 시까지 시뮬레이션한 후  $n_i \rightarrow$  root 경로 상 노드의 **si**, **wi** 업데이트

```
}
```

시뮬레이션: 트리에서 한 노드를 선정해  
가상으로 게임 종료 시까지 진행해 보는 것



## k개의 노드 선정 방법: 승리 전략 찾는데 도움 될 가능성 큰 경우 선정

50

① 승률 높으면서 ② 탐색 많이 이루어지지 않은 노드 우선 추가

si: 트리에서 노드 ni 아래 경우를 simulation 해본 횟수

sp: 노드 ni의 parent의 si 값

wi: si번의 simulation 중 (승리 전략을 찾고자 하는 player가) 승리한 횟수

c: 상수이며, 보통  $\sqrt{2}$  사용

```
while( root node의 si < k ) {
```

① selection: 새로 추가할 노드 선택

root에서 시작해  $\frac{w_i}{s_i} + c \times \sqrt{\frac{\ln(sp)}{s_i}}$

탐색횟수가 낮고 승률이 높은 쪽으로 내려감

가장 큰 자식 선정하며 내려가다가

아직 추가되지 않은 노드를 만나면 이 노드 선택

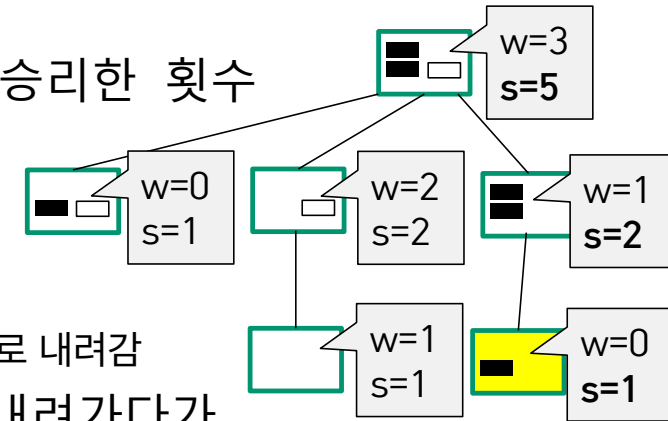
만약 게임 종료 나타내는 노드에 다다르면 더는 새 노드를 추가할 필요 없다 보고 종료

② expansion: ①에서 선정한 노드 추가하며 si, wi는 0으로 초기화

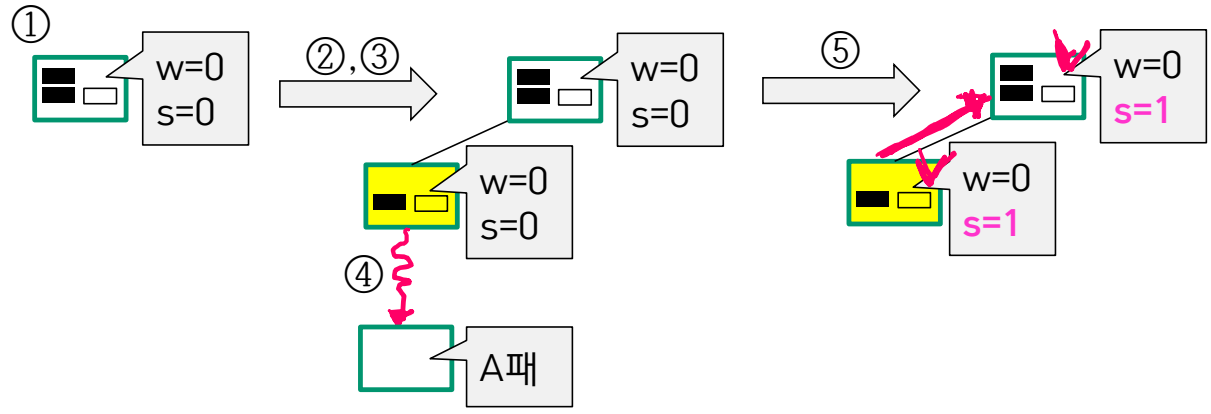
③ simulation: ②에서 추가한 새 노드에서 시작해 게임 종료 시까지 진행해 봄

④ update: ③의 simulation 결과에 따라 새 노드 → root 경로 상 노드의 si, wi 업데이트

}

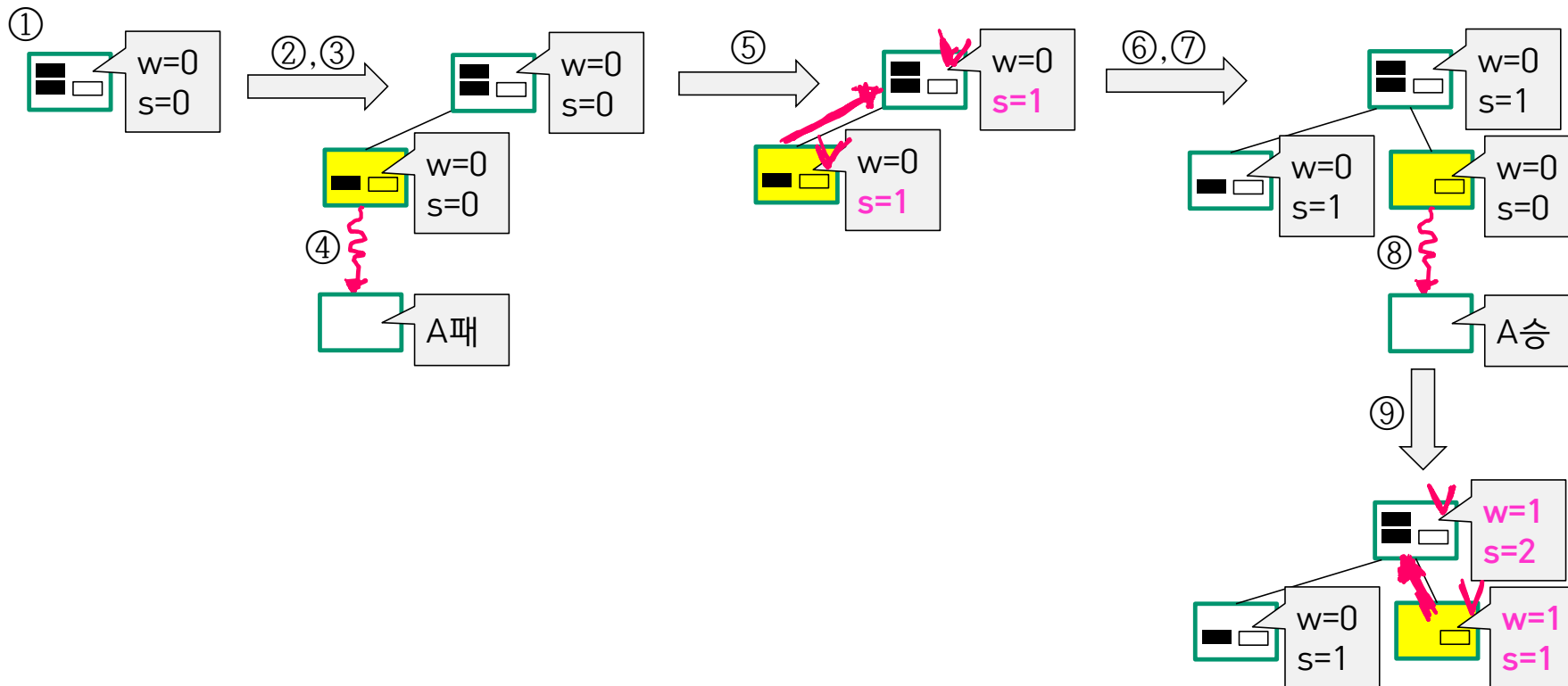


$\frac{w_i}{s_i} + c \times \sqrt{\frac{h(sp)}{s_i}}$  가장 높은 노드 따라 내려가 새 노드 추가 x k번 반복



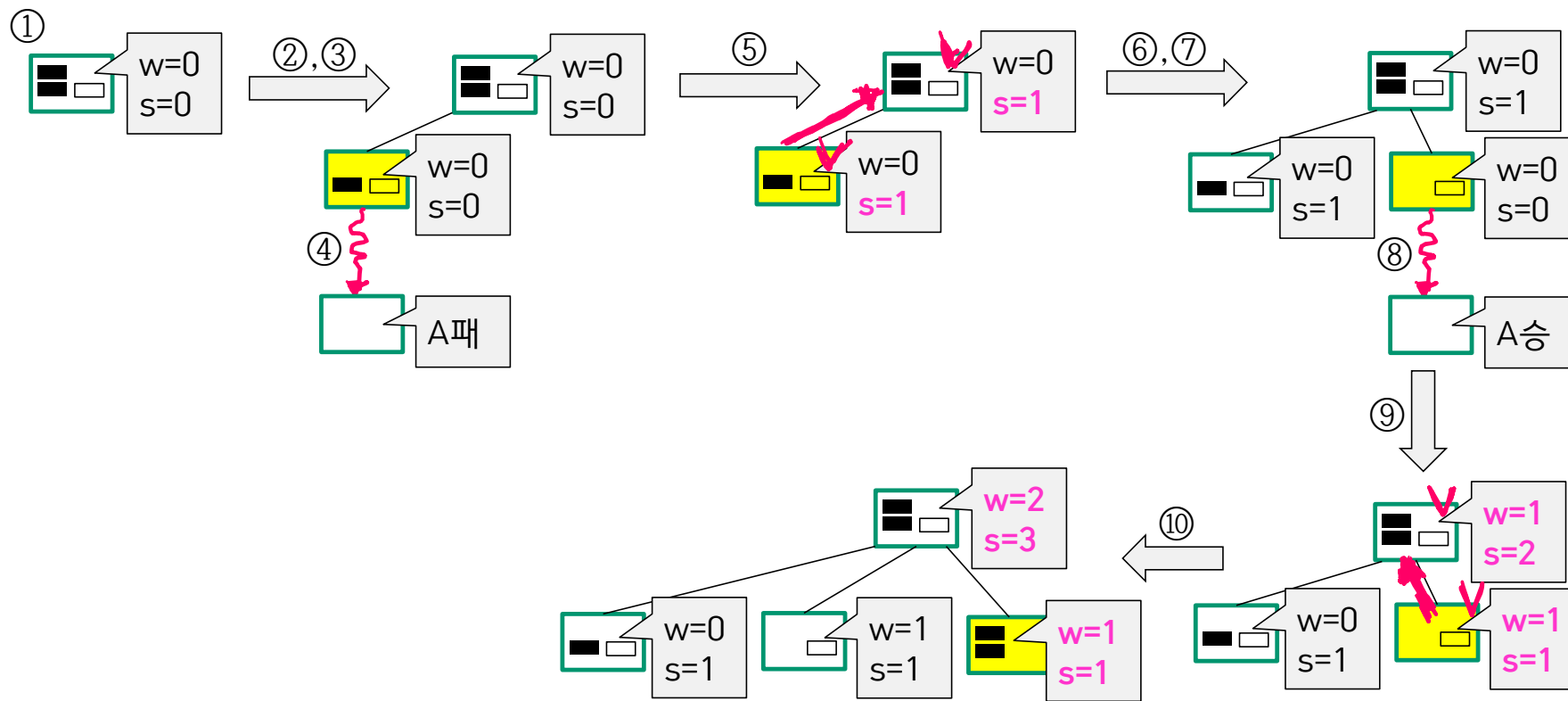
$$\frac{w_i}{s_i} + c \times \sqrt{\frac{h(sp)}{s_i}}$$

가장 높은 노드 따라 내려가 새 노드 추가 x k번 반복



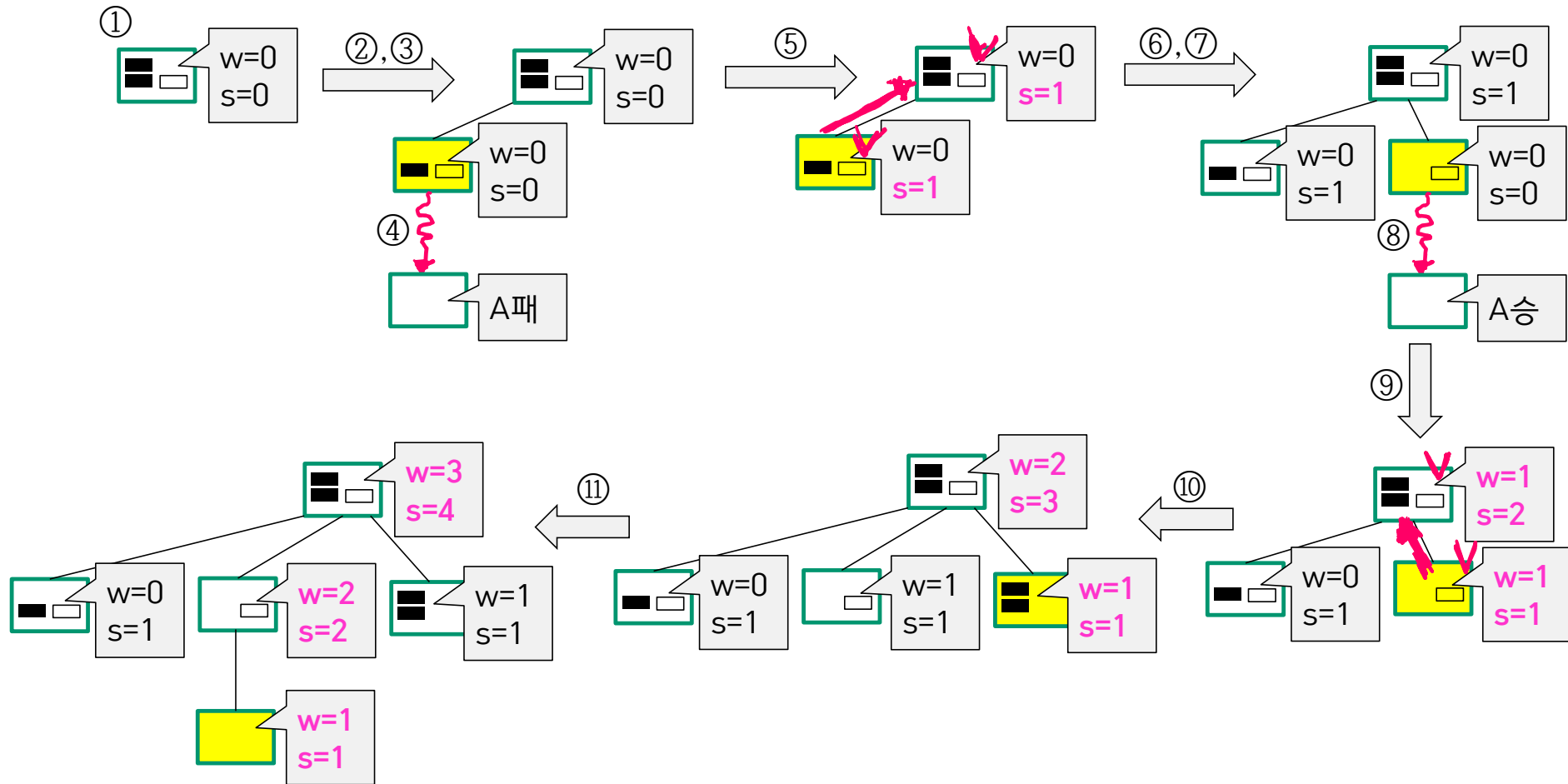
$$\frac{w_i}{s_i} + c \times \sqrt{\frac{h(sp)}{s_i}}$$

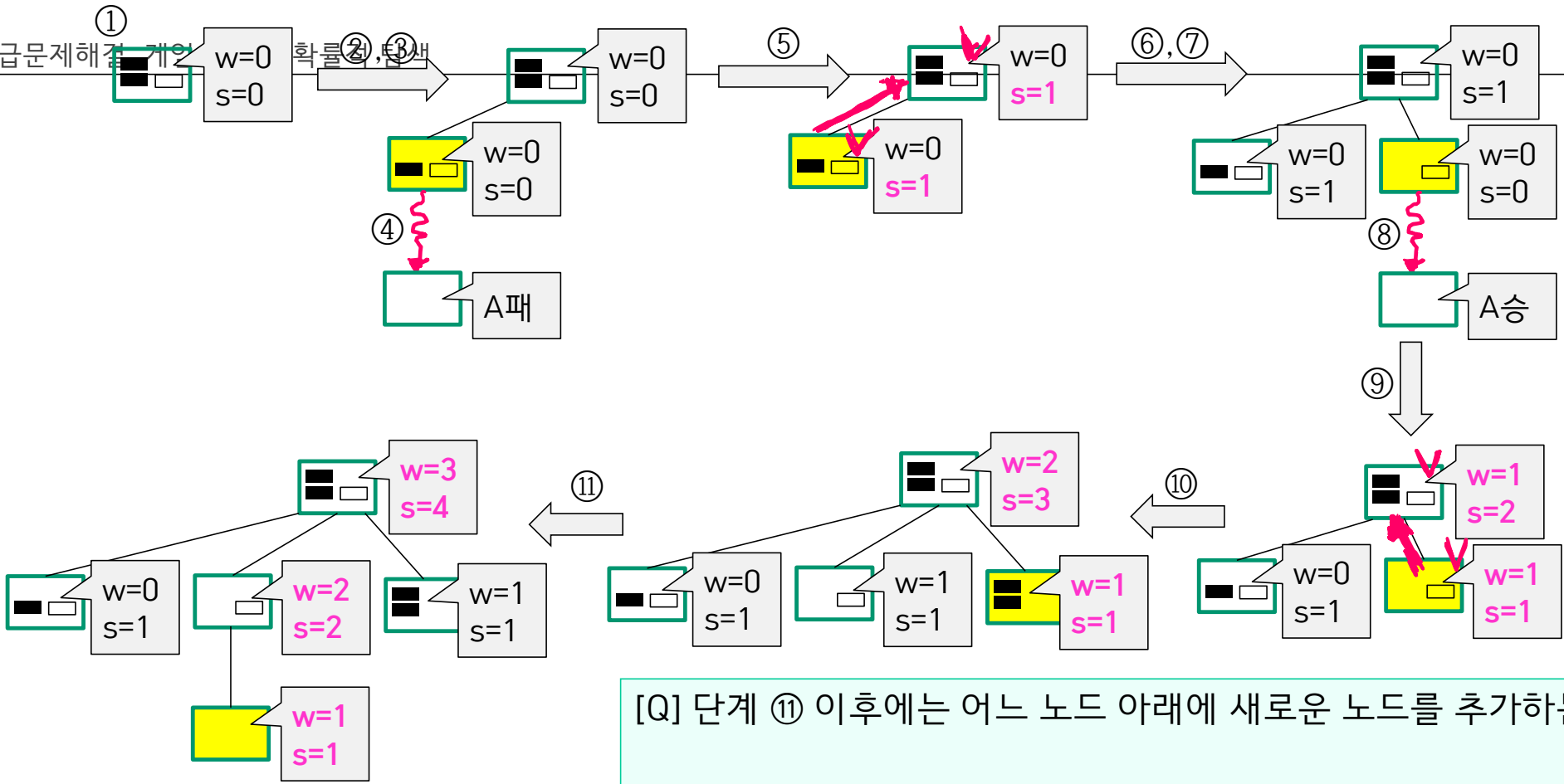
가장 높은 노드 따라 내려가 새 노드 추가 x k번 반복



$$\frac{w_i}{s_i} + c \times \sqrt{\frac{h(sp)}{s_i}}$$

가장 높은 노드 따라 내려가 새 노드 추가 x k번 반복

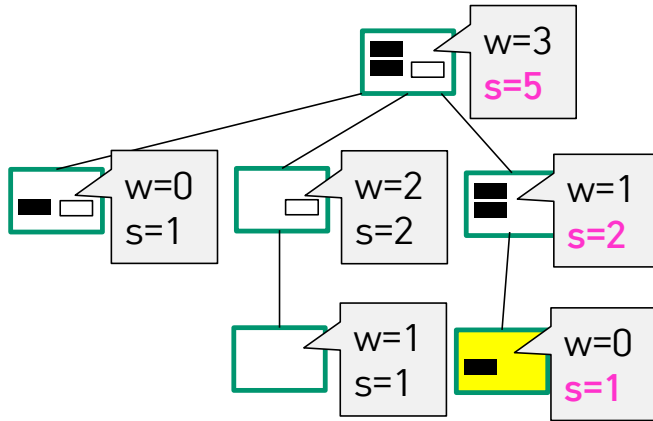




[Q] 단계 ⑪ 이후에는 어느 노드 아래에 새로운 노드를 추가하는가?

[Q] 새로 추가한 노드가 (1, 0) 이라면, 시뮬레이션 결과에 따라 어떤 노드의 값들이 어떻게 바뀌는가?

[Q] 그 이후에는 어떤 노드를 추가할까?  $k = 5$ 라고 가정하시오.



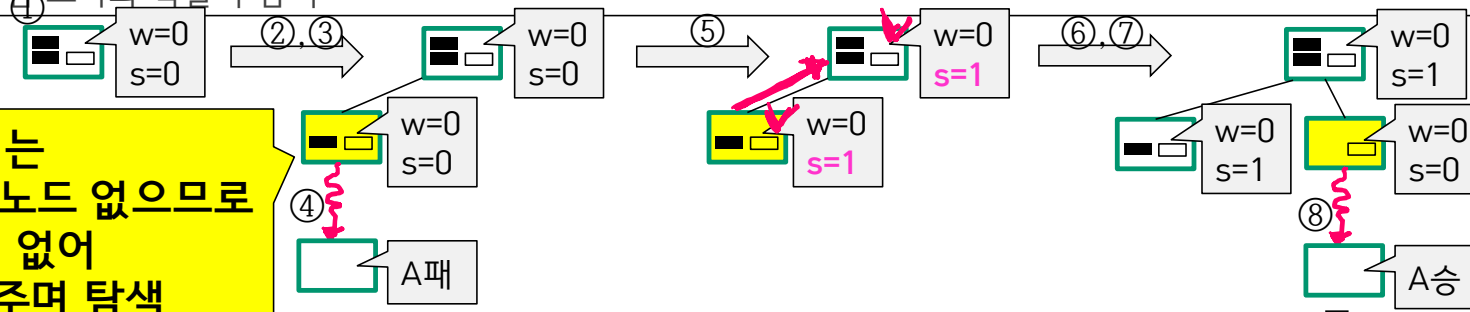
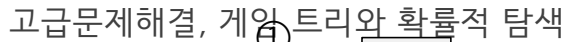
[Q] 지금까지 만들어온 왼쪽 트리와 전체를 그린 트리를 비교해 보자. 왼쪽 트리에서는 특히 어떤 노드 아래를 더 많이 & 자세히 탐색했다고 볼 수 있는가?

남은 돌의 개수가 적은 노드  
승률이 높은 노드

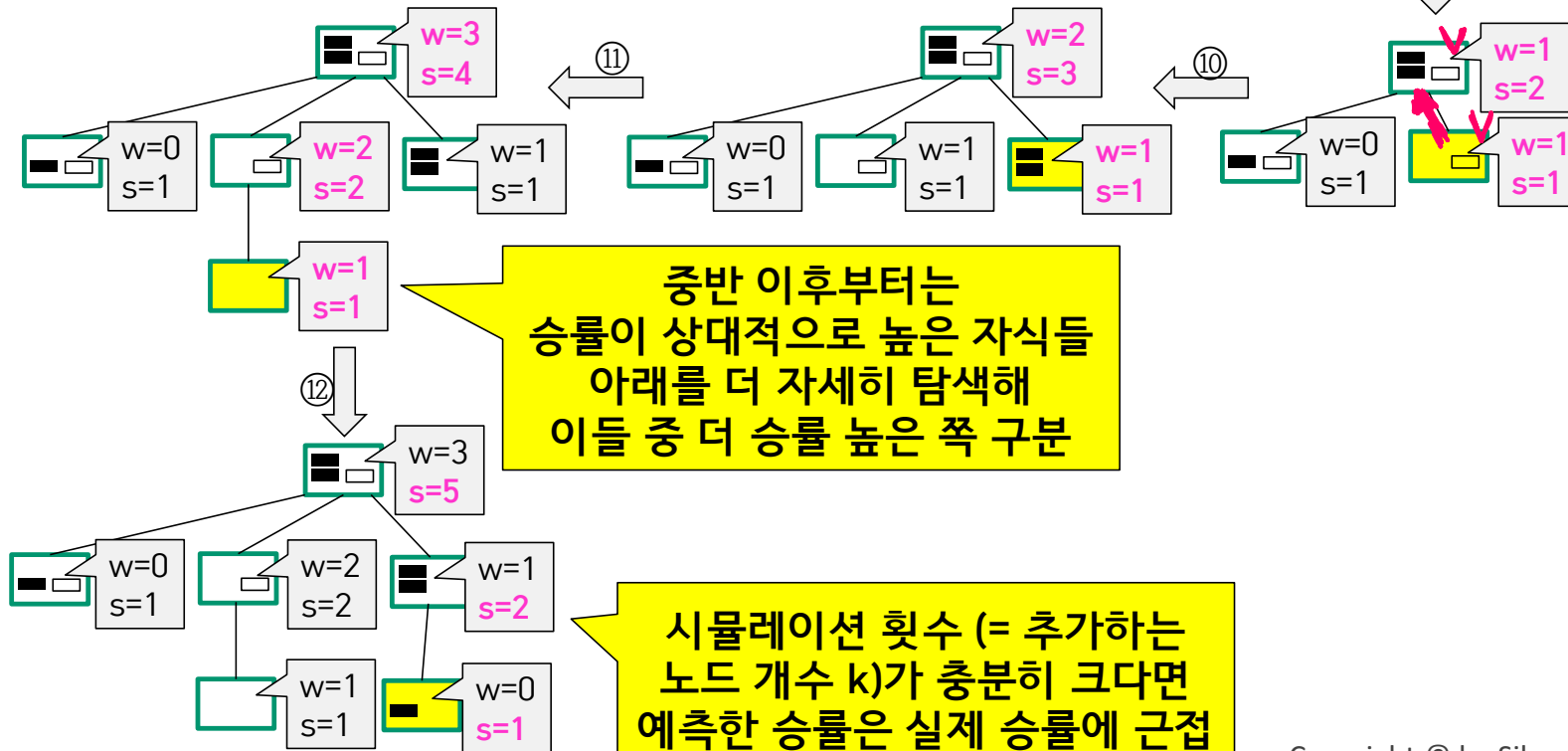
[Q] 왼쪽 트리에서 root의 자식 노드 3개의 승률을 보자. 이들의 승률은 실제 승률 (전체 트리를 그렸을 때의 승률)에 근접하는가?

[Q]  $k = 6$  이었다면 노드를 하나 더 추가하려고 시도할 것인데, 어디에 추가할까?





초반에는  
아직 탐색한 자식 노드 없으므로  
승률 차이 없어  
골고루 기회 주며 탐색



중반 이후부터는  
승률이 상대적으로 높은 자식들  
아래를 더 자세히 탐색해  
이들 중 더 승률 높은 쪽 구분

시뮬레이션 횟수 (= 추가하는 노드 개수  $k$ )가 충분히 크다면  
예측한 승률은 실제 승률에 근접

- 게임 트리 만들기 위해 탐색할 경우가 너무 많을 때
- 그 중 일부를 sampling해 일부분에 대한 트리를 만들기 위해 Monte-Carlo 방법 사용
- 단 sampling 대상으로 가능한 모든 경우 중 일부를 random하게 선정하는 것이 아니라, 승리 전략 찾는 데 더 도움 되는 경우를 선정해 sampling하며
- 이처럼 더 도움 되는 경우 찾기 위해 우선 순위 정하는 수식  $\frac{w_i}{s_i} + c \times \sqrt{\frac{h(sp)}{s_i}}$  사용

Player	RandomPlayer	TreePlayer	PTreePlayer	runBWGame()
모든 Player 클래스의 상위 클래스로, 하위 클래스가 반드시 구현해야 하는 함수를 선언하는 추상 클래스	랜덤하게 수를 결정하는 Player를 나타내는 클래스	게임 Tree 전체를 만들어 수를 결정하는 Player를 나타내는 클래스	게임 Tree 일부를 확률적으로 만들어 수를 결정하는 Player 나타내는 클래스	두 Player 클래스를 입력으로 주면 이들을 사용해 게임 진행하는 함수
구현 완료	구현 완료	구현 완료	일부 구현됨 <b>확률적으로 트리 확장하는 부분 구현하는 것이 실습 과제</b>	구현 완료

## 코드 개요: Player 클래스 - 모든 Player의 상위 클래스

```
class Player(ABC): # extends ABC(Abstract Base Class) to become an abstract class
    @abstractmethod
    def __init__(self, numBlack, numWhite, player): pass      # player: 0 선공, 1 후공

    @abstractmethod
    def doMyTurn(self): pass
    # return (c, n), meaning that we move n(1 ~ 2) coins of color c(검은색 0 or 흰색 1)

    @abstractmethod
    def doOthersTurn(self, color, number): pass
```

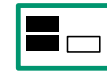
- 모든 Player 클래스가 상속해야 하는 상위(추상) 클래스
- Player 하위 클래스에게 반드시 구현해야 하는 함수와 형태 알려줌
- runBWGame() 함수에서 모든 player는 이러한 기능 구현되어 있다고 가정하고 실행하기 때문

```
class RandomPlayer(Player): # 함수 내용은 생략
    def __init__(self, numBlack, numWhite, player)
    def doMyTurn(self)
    def doOthersTurn(self, color, number)
```

```
class TreePlayer(Player): # 함수 내용은 생략
    def __init__(self, numBlack, numWhite, player)
    def doMyTurn(self)
    def doOthersTurn(self, color, number)
```

## 코드 개요: RandomPlayer 클래스

61



게임의 현재 상태만 기억하며,  
이 상태에서 가능한 선택지 중  
하나를 랜덤하게 선택하며  
진행하는 player

```
class RandomPlayer(Player):
    def __init__(self, numBlack, numWhite, player): # 생성자
        self.numCoins = [numBlack, numWhite] # 동전 수 초기화

    def doMyTurn(self): # 내 차례 진행
        # 동전 색깔 선택
        if self.numCoins[0] >= 1 and self.numCoins[1] >= 1: color = random.randint(0, 1)
        elif self.numCoins[0] >= 1: color = 0
        else: color = 1

        # 선택한 색의 동전을 가져갈 개수 선택
        number = random.randint(1, min(self.numCoins[color], 2))

        # 선택한 색/개수에 따라 남은 동전 수 줄이기
        self.numCoins[color] -= number

        return color, number # 선택한 색/개수 반환 (상대 Player도 이에 따라 상태 변화시키도록)

    def doOthersTurn(self, color, number): # 상대 차례 진행한 결과를 내 상태에 반영
        self.numCoins[color] -= number
```

## 코드 개요: TreePlayer 클래스

62

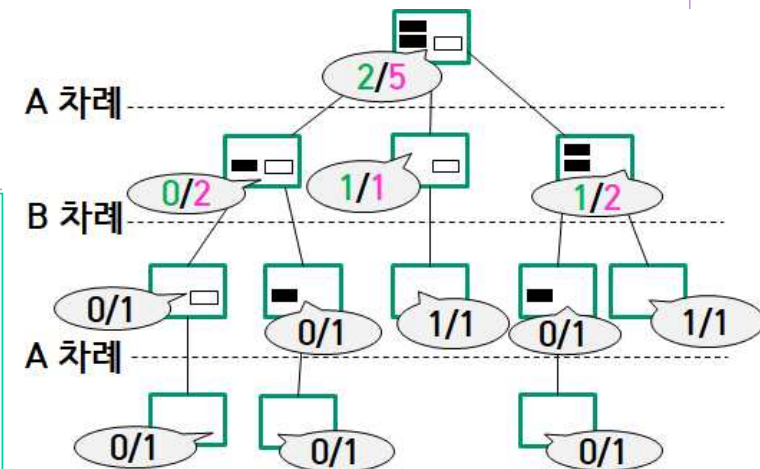
```
class TreePlayer(Player):
    def __init__(self, numBlack, numWhite, player):
        # 트리 만들기
        # self.root: 현재 루트 노드 저장. 그 외 노드는 루트에서부터 서로 reference로 연결되어 있음

    def doMyTurn(self):
        # 생성자에서 만든 트리 사용해 게임 진행
        # 특히 self.root의 children 중 승률이 가장 높은 child 선택해 진행하면서
        # self.root는 선택한 child가 됨
        # 또한 선택한 동전 종류와 개수 (color (0 or 1), number (1 or 2)) 반환

    def doOthersTurn(self, color, number):
        # 상대 player가 진행한 결과를 (color, number) 형식으로 받아
        # 이에 해당하는 child로 self.root를 이동시킴. 반환값 없음
```

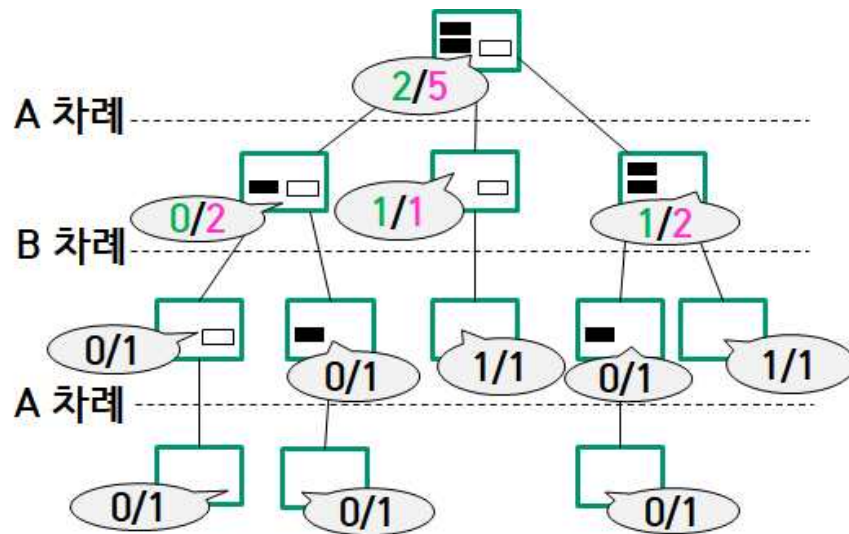
[Q] self.root가 노드 (2, 1)을 가리킨다고 가정하자. doMyTurn()을 호출하면 어떤 일을 하는가? 오른쪽 트리에서 설명해 보시오.

[Q] self.root가 노드 (2, 1)을 가리킨다고 가정하자. doOthersTurn(1, 1)을 호출하면 어떤 일을 하는가? 오른쪽 트리에서 설명해 보시오.



## 코드 개요: TreePlayer 클래스 - 각 노드를 5-tuple로 표현

63



[Q] 왼쪽 트리에서 루트 (2, 1)와 그 자식인 (2, 0)을 5-tuple 형식으로 표현해 보시오.

- 5-tuple: ([0], [1], [2], [3], [4])
- [0]: 현재 차례인 player. 0은 선공, 1은 후공
- [1]: 동전 수를 리스트에 저장 [검은 동전 수, 흰 동전 수]
- [2]: 승리 수를 리스트에 저장 [선공 승리 수, 후공 승리 수]
- [3]: 4개 자식 노드에 대한 reference를 리스트에 저장 [검은동전-1, 검은동전-2, 흰동전-1, 흰동전-2]. 해당하는 경우 없으면 None
- [4]: 부모 노드에 대한 reference. 종료 노드인 경우 부모 따라 올라가며 승리 수 update하기 위해 필요

실습 과제에서 각 함수의 코드는 함수 원형 및 기능에 맞게 동작하도록 자유롭게 작성하되 트리를 저장하는 형식은 이와 같게 해야 자동 채점할 수 있음

## GameTree.py

A total of 12 nodes  
 (2,1), win rate 0.40 (2/5)

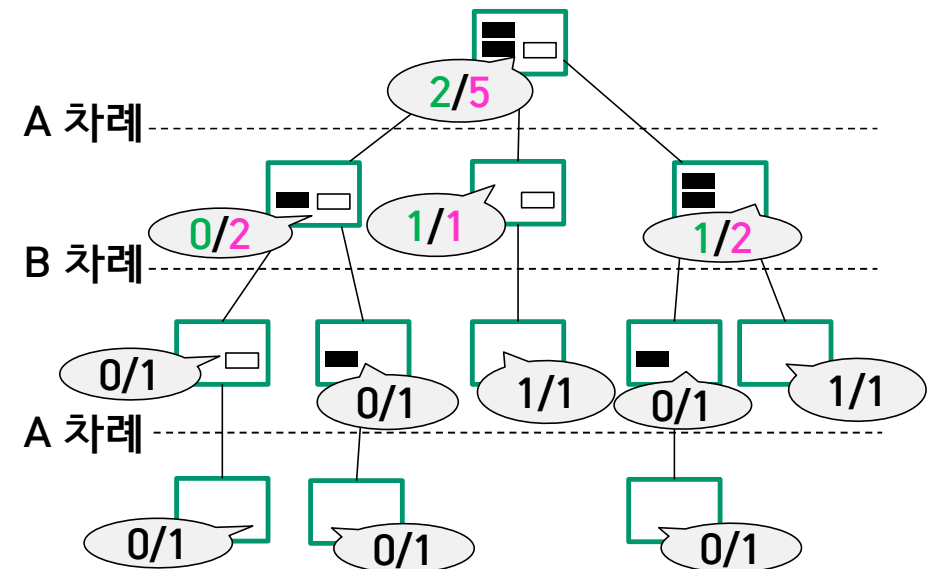
(1,1), win rate 0.00 (0/2)  
(0,1), win rate 1.00 (1/1)  
(2,0), win rate 0.50 (1/2)

(0,1), win rate 0.00 (0/1)  
 (1,0), win rate 0.00 (0/1)  
 (0,0), win rate 1.00 (1/1)  
 (1,0), win rate 0.00 (0/1)  
 (0,0), win rate 1.00 (1/1)

(0,0), win rate 0.00 (0/1)  
(0,0), win rate 0.00 (0/1)  
(0,0), win rate 0.00 (0/1)

## 게임 트리 노드를 BFS 순으로 출력

## Player의 승률





## TreePlayer: 게임 트리 활용해 진행하는 참가자 나타내는 class

```
if __name__ == "__main__":  
    # TreePlayer(검은 동전 수, 흰 동전 수, 순서(0-선공, 1-후공))  
    print(TreePlayer(2, 1, 1))
```

GameTree.py

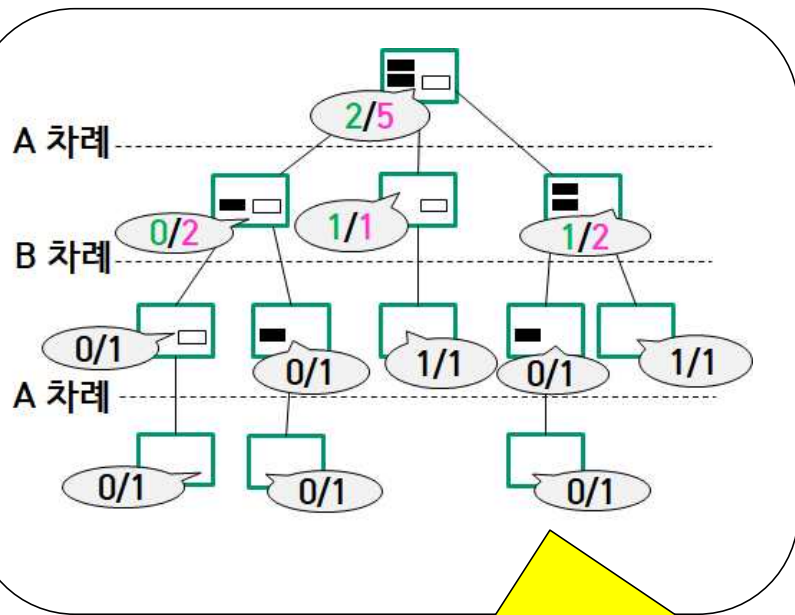
[Q] TreePlayer 객체가 후공 입장이 되도록 코드를 변경한 후 트리를 관찰해 보자. 어떻게 달라졌는가?

코드 개요: (검은동전 개수, 흰동전갯수, 선공Player, 후공Player, 결과출력여부)

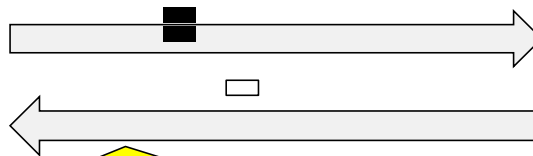
66

runBWGame(2, 1, TreePlayer, RandomPlayer, True)

TreePlayer

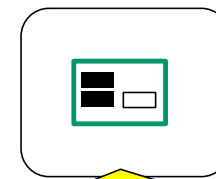


게임 트리 만들고 이에 따라 승리할 가능성이 가장 높은 선택지를 선택하며 진행



두 Player가 차례대로 각자의 선택을 주고 받으며 게임 진행

RandomPlayer



게임의 현재 상태만 기억하며, 이 상태에서 가능한 선택지 중 하나를 랜덤하게 선택하며 진행

## 코드 개요: runBWGame() - 게임 진행하는 함수

67

```
def runBWGame(numBlack, numWhite, PlayerClass0, PlayerClass1, debug):  
    # Player 객체 만들기  
    players = [PlayerClass0(numBlack, numWhite, 0), PlayerClass1(numBlack, numWhite, 1)]  
  
    # 동전 개수 및 현재 순서인 player 초기화  
    numCoins = [numBlack, numWhite]  
    currentPlayer, otherPlayer = 0, 1    # 0 선공, 1 후공  
  
    while numCoins[0] > 0 or numCoins[1] > 0:  
        color, number = players[currentPlayer].doMyTurn() # 현재 player 차례 진행  
        players[otherPlayer].doOthersTurn(color, number) # 진행한 결과를 다른 player에도 반영  
  
        numCoins[color] -= number    # 진행한 결과에 따라 남은 동전 수 줄이기  
  
        currentPlayer, otherPlayer = (currentPlayer + 1) % 2, (otherPlayer + 1) % 2  
  
    return currentPlayer # taking the last coin loses the game
```



## runBWGame(): 지정한 플레이어로 게임 진행하고 승자(0/1) 반환

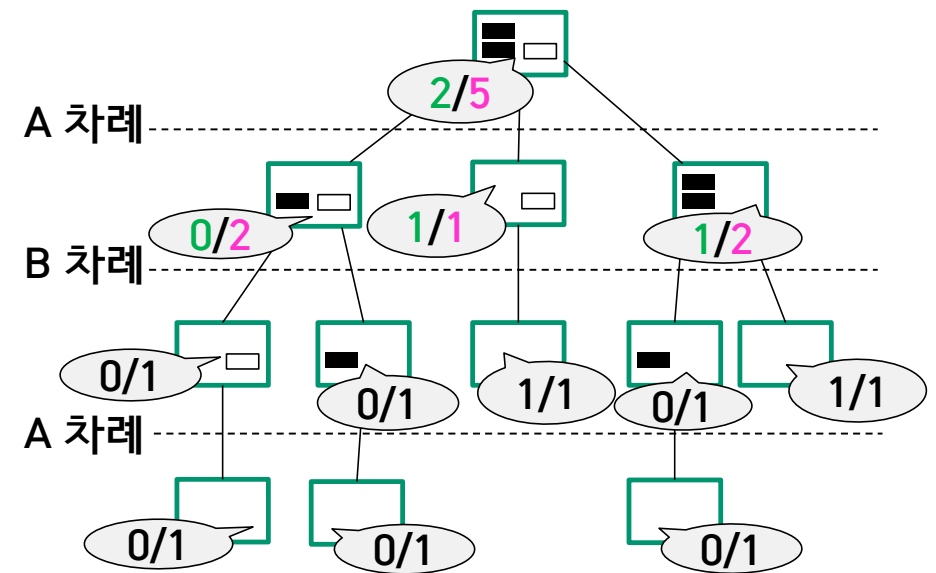
69

GameTree.py

```
if __name__ == "__main__":  
    # runBWGame(검은 동전 수, 흰 동전 수,  
    #           선공 Player 클래스, 후공 Player 클래스, 결과 출력 여부(T/F))  
    print(runBWGame(2, 1, TreePlayer, RandomPlayer, True))
```

[Q] 위 설정 그대로 게임을 여러 차례 실행해 보자. 승자가 항상 같은가 아니면 바뀌는가? 이유는 무엇인가?

[Q] 이번에는 RandomPlayer를 선공, TreePlayer를 후공으로 게임을 여러 차례 실행해 보자. 승자가 항상 같은가 아니면 바뀌는가? 이유는 무엇인가?



## 승률과 수행시간 관찰: 어느 지점부터 느려지는가?

70

```
if __name__ == "__main__":
    numBlack, numWhite = 3, 3    # 동전 수
    PlayerClass0, PlayerClass1 = TreePlayer, RandomPlayer    # 두 player의 클래스
    numGames = 10    # 게임 수
    numWins = [0, 0]

    # 승리 횟수 관찰
    for i in range(numGames):
        numWins[runBWGame(numBlack, numWhite, PlayerClass0, PlayerClass1, False)] += 1
        print(f"out of {numGames} games, player 0 and 1 win ({numWins[0]}, {numWins[1]}) times")

    # 수행 시간 관찰
    tGame = timeit.timeit(lambda: runBWGame(numBlack, numWhite, PlayerClass0, PlayerClass1, ...
    print(f"Average running time is {tGame:.10f} for inputs ({numBlack}, {numWhite}, ...
```

out of 10 games, player 0 and 1 win (10, 0) times

Average running time is 0.0008994000 for inputs (3, 3, TreePlayer, RandomPlayer)

[Q] 동전 수를 증가시키면서 수행 시간을 관찰해 보자. 어떤 값부터 1초를 넘어가는가? 왜 그럴까?

**PTreePlayer (Probabilistic TreePlayer) 클래스 구현 조건:**  
**이번 시간에 배운 확률적 탐색 방법 사용해 BW 동전 게임 진행하는 Player 클래스 구현**

- 이번 시간에 제공한 코드 GameTree.py의 PTreePlayer 클래스 내에 코드 작성해 제출
- PTreePlayer 클래스의 아래 멤버 함수 구현  
**def expandTree(self):**  
**# 확률적 탐색 방법에 따라 트리에 노드를 추가하되**  
**# root의 simulation 횟수가 self.maxNumSimulations을 넘지 않도록 추가**
- 채점 코드에서는  $0 \leq$  검은 동전 수, 흰 동전 수  $\leq 30$  까지 입력하나, 배운 방식대로 잘 구현하였다면 이보다 훨씬 큰 입력이 들어오더라도 (예: 100, 1000, ...) 잘 동작함
- PTreePlayer 내에 이미 구현된 함수 및
- 위 함수에 대한 자세한 요구조건은 다음 페이지에 이어짐

- PTreePlayer 클래스의 멤버 변수

self.root # 트리의 root 노드를 저장하며, 나머지 노드는 이 노드에 연결됨

# root 노드는 게임 진행 중 늘 현재 상태를 나타내야 하며

# 따라서 게임을 한 차례 진행할 때마다 root는 자신의 자식 노드 중 하나로 바뀜

self.player # 이 객체가 선공을 하는 객체인지 (0) 후공을 하는 객체인지 (1) 나타냄

self.maxNumSimulations = 50 # root 아래에 추가하는 노드 수의 최댓값

# 즉 트리 전체를 만들면 시간이 너무 많이 걸리므로

# 항상 root 아래에는 최대 50개의 노드만 있도록 제한함

self.CONST\_C = math.sqrt(2) # 수식  $\frac{w_i}{s_i} + c \times \sqrt{\frac{h(sp)}{s_i}}$  에서 상수 c 값을 나타내며

# 실험적으로 잘 동작한다고 보여진 2의 제곱근을 사용함



- PTreePlayer 클래스의 멤버 함수

- ① def \_\_init\_\_(self, numBlack, numWhite, player):

- # 클래스 생성자로 멤버 변수를 초기화하며, 또한 expandTree() 호출해 지정된 크기의 트리를 구성함

- ② def doMyTurn(self) # 생성자에서 만든 트리 사용해 player 차례의 게임 진행

- ③ def doOthersTurn(self, color, number) # 상대 player가 선택한 결과에 따라 게임 진행

- ④ def \_\_str\_\_(self) # 트리 정보를 문자열 형식으로 반환하며, 트리 객체를 출력할 때 호출됨 (디버깅에 활용)

- ⑤ **def expandTree(self) # 지정된 크기의 트리 구성**

- 위 함수 각각에 대한 자세한 설명은 다음 페이지에 이어짐

## ① PTreePlayer 클래스의 생성자 함수 `__init__(self, numBlack, numWhite, player)`

```
def __init__(self, numBlack, numWhite, player):
    # numBlack, numWhite: 검은 동전 수, 흰 동전 수
    # player: 이 객체가 선공 player인지 (0) 후공 player인지 (1) 나타냄

    # 노드를 저장하는 형식 (5-tuple)에 맞게 root 노드 생성
    self.root = (0, [numBlack, numWhite], [0, 0], [None, None, None, None], None)
    self.player = player
    self.maxNumSimulations = 50 # root의 최대 simulation 횟수 ≤ 50 으로 제한하도록 트리 구성
    # 즉 root 아래에 최대 50개의 자식 노드가 있도록 함
    self.CONST_C = math.sqrt(2) # 상수 c 값을 2의 제곱근으로 초기화

    # expandTree() 호출하여 self.root 아래에 최대 self.maxNumSimulations 만큼의 노드 추가
    self.expandTree()
```

- 노드를 저장하는 형식 (5-tuple): ([0], [1], [2], [3], [4])
- [0]: 현재 차례인 player 0은 선공, 1은 후공
- [1]: 동전 수를 리스트에 저장 [검은 동전 수, 흰 동전 수]
- [2]: 승리 수를 리스트에 저장 [선공 승리 수, 후공 승리 수]. **두 값을 더한 값이 시뮬레이션 횟수 si가 됨**
- [3]: 4개 자식 노드에 대한 reference를 리스트에 저장 [검은동전-1, 검은동전-2, 흰동전-1, 흰동전-2]. 해당하는 경우 없으면 None
- [4]: 부모 노드에 대한 reference. 종료 노드인 경우 부모 따라 올라가며 승리 수 update하기 위해 필요

**self.root 및 그 아래 모든 노드는 이 형식에 맞게 저장해야 채점이 올바르게 진행됨**

## ② PTreePlayer 클래스의 doMyTurn(self) 함수: 생성한 트리 사용해 player 차례 진행

```
def doMyTurn(self):
    # self.root의 자식 중 승률이 가장 높은 자식과 그 승률을 저장할 변수 초기화
    maxWinRate, childWithMaxWinRate = None, None

    # self.root의 자식 중 승률이 가장 높은 자식 찾아 저장
    for child in self.root[3]:
        if child != None:
            if maxWinRate == None: maxWinRate, childWithMaxWinRate = child[2][self.player]/sum(child[2]), child
            else:
                winRate = child[2][self.player]/sum(child[2])
                if winRate > maxWinRate: maxWinRate, childWithMaxWinRate = winRate, child

    # 승률이 가장 높은 자식으로 진행할 때 어떤 색의 코인을 (color) 몇 개 가져가는지 (number) 확인
    if self.root[1][0] > childWithMaxWinRate[1][0]: color, number = 0, self.root[1][0] - childWithMaxWinRate[1][0]
    else: color, number = 1, self.root[1][1] - childWithMaxWinRate[1][1]

    # self.root를 승률이 가장 높은 자식으로 교체하고
    # expandTree() 함수 호출해 새로운 root의 시뮬레이션 횟수가 self.maxNumSimulation이 될 때까지 노드 추가
    self.root = childWithMaxWinRate
    self.expandTree()

    # 앞에서 선정한 코인 색깔과 (color) 개수 (number) 반환 - 상대 player도 이에 따라 게임을 진행하도록 하기 위함
    return color, number
```

### ③ PTreePlayer 클래스의 doOthersTurn(self) 함수: 상대 player의 진행 결과에 따라 상태 변경

```
def doOthersTurn(self, color, number):  
    # 상대 player가 선택한 코인 색깔과 (color) 개수 (number)를 입력으로 받음  
  
    # self.root를 color, number에 대응되는 자식 노드로 교체  
    # (color, number) = (0, 1), (0, 2), (1, 1), (1, 2)가 각각 자식 노드 index 0, 1, 2, 3에 대응됨  
    self.root = self.root[3][color * 2 + number - 1]  
  
    # expandTree() 함수 호출해  
    # 새로운 root의 시뮬레이션 횟수가 self.maxNumSimulation이 될 때까지 노드 추가  
    self.expandTree()
```

⑤ PTreePlayer 클래스의 expandTree(self) 함수 수도코드:  
self.root 아래에 최대 self.maxNumSimulations 만큼의 노드가 있도록 트리 확장

```
def expandTree(self):
    while self.root의 simulation 횟수 < self.maxNumSimulations:
        # ㉠~㉢ 탐색 & 노드 추가
        node = self.root # self.root에서 시작해 추가할 노드 탐색 시작
        while True:
            node가 게임 종료를 나타낸다면 더는 노드를 추가할 필요 없다고 보고 return
            node의 자식 4개 중  $\frac{w_i}{s_i} + c \times \sqrt{\frac{h(sp)}{s_i}}$  가 가장 큰 노드 maxChild 찾기
            maxChild가 아직 트리에 추가하지 않은 노드라면 추가하고 break
            maxChild가 기존에 이미 추가한 노드라면 node = child로 대체

        # ㉣ 시뮬레이션
        maxChild에서 게임 시작해 종료시까지 임의로 진행하여 승자 기억

        # ㉤ wi, si 업데이트
        ㉣의 시뮬레이션 결과에 따라 maxChild → self.root 경로 상 모든 노드의 wi, si 업데이트
```

## 실행 예

78

```
print(PTreePlayer(1, 1, 0))
```

A total of 5 nodes

(1,1), win rate 1.00 (4/4)

(0,1), win rate 1.00 (2/2)

(1,0), win rate 1.00 (2/2)

(0,0), win rate 1.00 (1/1)

(0,0), win rate 1.00 (1/1)

**self.maxNumSimulations = 50** 이지만  
**self.root** 아래로 추가할 수 있는 노드 수가  
50개에 미치지 않기 때문에 그보다 적은 수의  
노드가 추가되었습니다.

## 실행 예

79

```
print(PTreePlayer(2, 1, 0))
```

A total of 6 nodes

(2,1), win rate 0.60 (3/5)

(1,1), win rate 0.00 (0/1)

(0,1), win rate 1.00 (2/2)

(2,0), win rate 0.50 (1/2)

(0,0), win rate 1.00 (1/1)

(1,0), win rate 0.00 (0/1)

(2, 0) 아래로  
첫 시뮬레이션 했을 때  
승리한 경우

A total of 5 nodes

(2,1), win rate 0.50 (2/4)

(1,1), win rate 0.00 (0/1)

(0,1), win rate 1.00 (2/2)

(2,0), win rate 0.00 (0/1)

(0,0), win rate 1.00 (1/1)

(2, 0) 아래로  
첫 시뮬레이션 했을 때  
패배한 경우

확률적 탐색 방법에서는  
시뮬레이션이 랜덤하게 진행되므로  
같은 조건으로 트리를 여러 번 만들었을 때  
생성한 게임 트리 모양과 w, s값은 항상 같지는  
않습니다.

## 실행 예

80

```
print(sum(PTreePlayer(100, 100, 0).root[2]))
```

50

동전 수가 충분히 크다면  
self.maxNumSimulation = 50개를  
가득 채워 트리를 만들었음을  
확인할 수 있습니다.

self.root[2]가 5-tuple 중 두  
player의 승수를 나타내므로, 이를  
더한(sum) 값은 총 simulation  
횟수를 나타냄



## 실행 예

81

```
runBWGame(2, 1, PTreePlayer, RandomPlayer, True)
```

```
player 0: (2, 1) --> (0, 1)  
player 1: (0, 1) --> (0, 0)  
player 0 wins
```

확률적 탐색 방법에서는  
시뮬레이션이 랜덤하게 진행되므로  
생성한 게임 트리 모양과 w, s값은 항상 같지는 않으며  
따라서 같은 조건으로 게임을 여러 번 실행하더라도  
게임 진행 순서가 항상 같지는 않습니다.

```
runBWGame(5, 5, PTreePlayer, RandomPlayer, True)
```

```
player 0: (5, 1) --> (3, 1)  
player 1: (3, 1) --> (2, 1)  
player 0: (2, 1) --> (0, 1)  
player 1: (0, 1) --> (0, 0)  
player 0 wins
```

그 외 실행 예는 채점 스크립트를 참조하세요.

## 그 외 프로그램 구현 조건

- 최종 결과물로 GameTree.py 파일 하나만 제출하며, 이 파일만으로 코드가 동작해야 함
- 채점이 올바르게 되기 위해 아래 사항을 지켜주세요.
  - 트리의 각 노드는 반드시 **5-tuple 형식을 지켜 저장**해야 함
  - 같은 상태를 나타내는 노드를 **병합하지 마세요**. 병합하지 않았다고 가정하고 채점합니다. 병합도 고려하도록 하면 과제 분량이 많아져 이번 과제에서는 생략했습니다. 따라서 **'확률적 탐색' 방법을 잘 구현해 보는 것에 초점**을 맞추세요.
- import는 사용할 수 없음
- \_\_main\_\_ 아래의 코드는 작성한 함수가 올바른지 확인하는 코드로
- 코드 작성 후 스스로 채점해 보는데 활용하세요.
- 각 테스트 케이스에 대해 P(or Pass) 혹은 F(or Fail)이 출력됩니다.
- 코드를 제출할 때는 \_\_main\_\_ 아래 코드는 제거하거나 수정하지 말고 제출합니다. (채점에 사용되기 때문)
- **속도 테스트는 거의 항상 pass 해야만 통과**입니다.

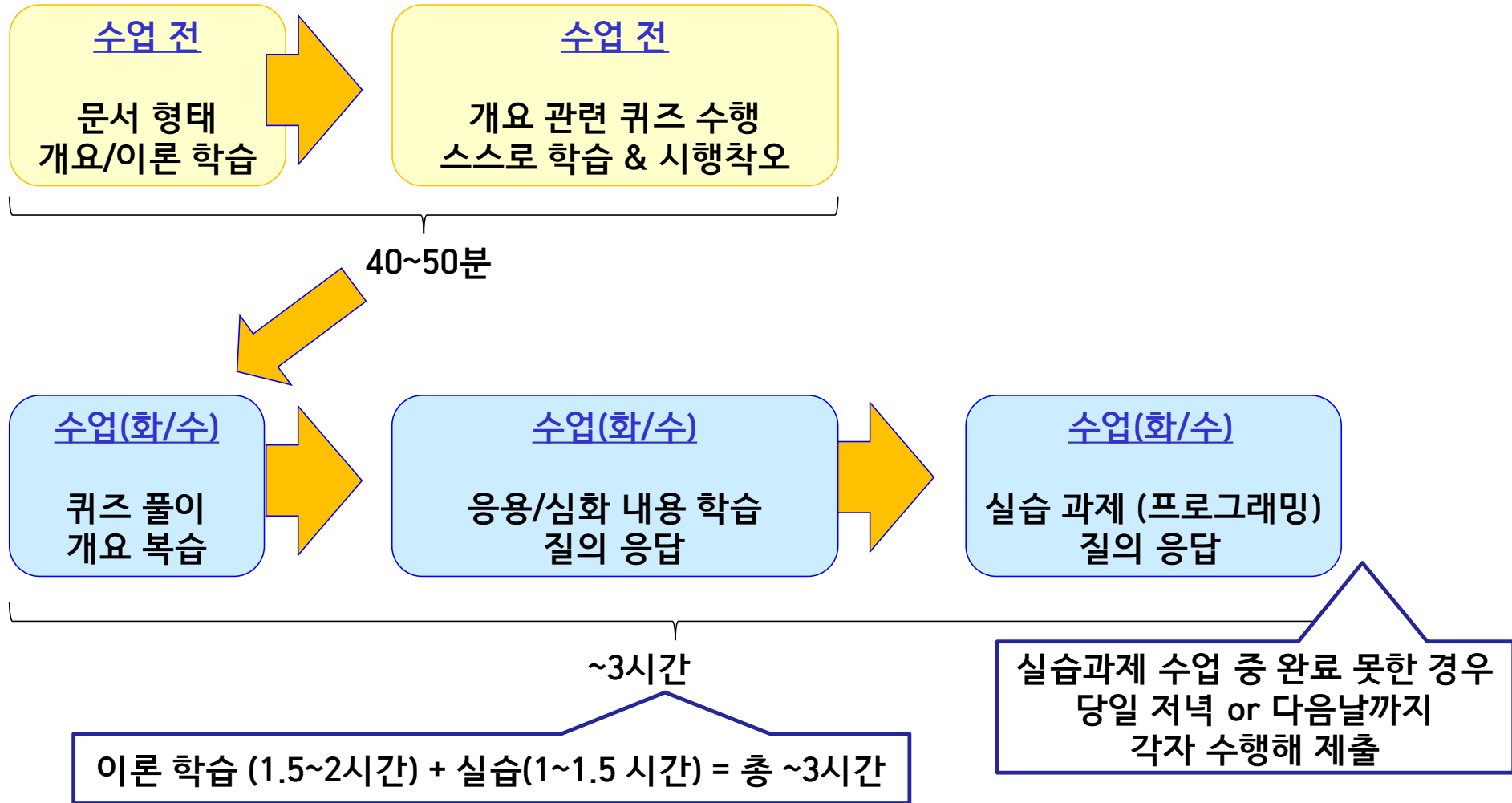


## 이번 시간 제공 코드 함께 보기

### ■ GameTree.py



# 스마트 출결





## 실습 문제 풀이 & 질의 응답

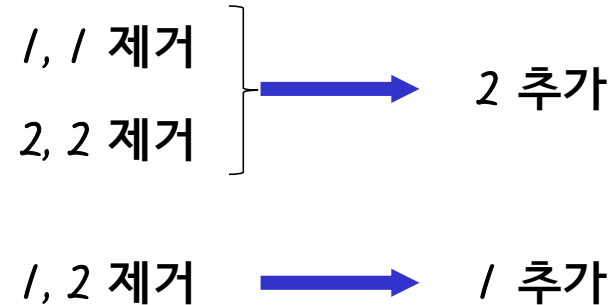
- 종료 시간(17:00) 이전에 **일찍 모든 문제를 통과**한 경우 각자 퇴실 가능
- 실습 문제에 대한 코드는 lms 과제함에 **다음 날 23:59까지 제출** 가능합니다.
- 마감 시간까지 작성을 다 못한 경우는 (제출하지 않으면 0점이므로) 그때까지 작성한 코드를 꼭 제출해 부분점수를 받으세요.
- 실습 문제는 **개별 평가**입니다. 제출한 코드에 대한 유사도 검사를 실습 문제마다 진행하며, 코드를 건네 준 사례가 발견되면 0점 처리됩니다.
- 로직에 대해 서로 의견을 나누는 것은 괜찮지만, 코드를 직접 건네 주지는 마세요.
- **채점 시 정확도/속도 비교에 사용하는 함수를 그대로 복사해 제출하거나 채점 코드에 대해서만 올바른 답을 반환하도록 작성한 경우에도 점수가 없습니다.**
- 본인 실력 향상을 위해서도 **코드는 꼭 각자 직접 작성**해 주세요.



## 두 개의 숫자를 제거하고 한 숫자 추가하기 → 반드시 승리할 전략이 있는가?

86

칠판에 **같은 개수의 1과 2**가 있다. 두 player A와 B가 번갈아 가며 **두 개의 숫자를 제거한 후 아래 규칙에 따라 한 숫자를 추가**한다. 최종적으로 칠판에 남은 숫자가 1이면 A가 승리하고, 2라면 B가 승리한다. A가 먼저 시작한다면(선공) A가 승리할 전략이 있을까?



- (힌트)
- 1, 2의 쌍이 홀수개인 경우 A가 항상 승리하며
- 짝수개의 쌍이 있다면 B가 항상 승리
- 위와 같이 승자가 결정되는 이유를 게임 규칙을 사용해 설명해 보세요.



[Q] **종료 조건**을 아래와 같이 바꾸었을 때 승리 전략을 도출해 보자.

87

한 번에 **동전 하나 혹은 두 개 가져가기**: 승리할 수 있는 전략이 있을까?

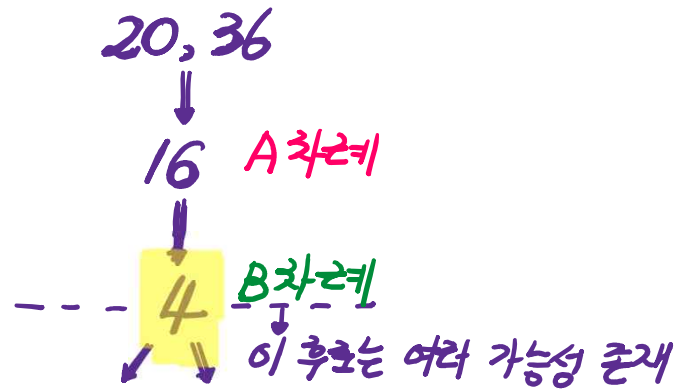


**N**개의 동일한 동전이 있다. 두 player A와 B가 순서를 정해 번갈아 가며 이 중 하나 혹은 두 개의 동전을 가져간다. **마지막 동전을 가져간 사람이 이긴다**. A가 먼저 시작한다면, A가 이길 수 있는 전략이 존재하는가?

- (힌트)
- A 차례에  $3k+1$  혹은  $3k+2$ 개에서 시작한 경우: B 차례에  $3k$ 개의 동전을 남겨주는 것을 반복하면 A가 승리할 수 있음
- A 차례에  $3k$ 개에서 시작한 경우: B가 실수해서 A 차례에  $3k+1$  혹은  $3k+2$ 개가 될 때까지 기다린 후 앞의 전략대로 진행



[Q] 칠판에 두 숫자 20과 36이 있다. 두 player A와 B가 번갈아 가며 칠판에 있는 임의의 두 숫자의 차를 쓴다. ① 두 숫자의 차는 반드시 양수 형태로 써야 하며 (즉 큰 수 - 작은 수를 써야 함), ② 이미 칠판에 있는 수를 써서는 안 된다. 자기 차례에 더는 새로운 차를 쓸 수 없게 된 player가 진다. A가 먼저 시작했을 때 A의 승리 전략이 있는가?



- (힌트)
- A가 항상 승리한다.
- 위와 같이 승자가 결정되는 이유를 설명해 보세요.