ㅍ[EEE2020] 22-1학기 데이터구조

End-Term Project

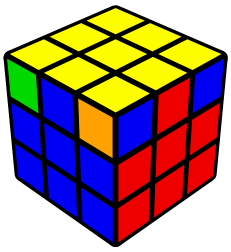
Queue를 이용한 Rubik's Cube Solving
: OP Algorithm 구현

2020132002 물리학과
전현빈

Ⅰ. Rubik's Cube Solving 기초

Ⅰ-1. Introduction

Rubik's cube는 각 3X3X3 형태의 정육면체 트위스티 퍼즐로 회전을 통해 생기는 조합은 잘 알려져 있듯이 43,252,003,274,489,856,000개이다. 이 중 모두 맞춰져 있는 경우의 수는 오직 하나뿐으로 임의의 상태에서 이 하나의 상태로 가기 위한 여러 해법이 존재한다. 컴퓨터를 이용하여 계산했을 때, 임의의 섞인 큐브는 평균 17.8회전 만에 맞출 수 있으며 최악의 경우에도 20회전으로 맞출 수 있음이 밝혀졌다. 이를 신의 수라 하며, 가장 최소 회전으로 맞추는 알고리즘을 신의 알고리즘이라고 한다.



Rubik's cube(이하 cube)를 맞추는 과정은 크게 orientation과 permutation으로 나누어 설명할 수 있다. orientation은 각 조각의 방향을 정렬하는 것을 orientation, 각 조각의 위치를 맞추는 것을 permutation이라고 한다. 예를 들어 오른쪽 그림은 노란색 면으로 orientation 되었다고 할 수 있다. 또한 제일 위 층의 모서리 조각은 permutation되었다고 할 수 있다.

신의 algorithm에 제일 가깝다고 알려진 2-phase algorithm은 먼저 모든 조각의 permutation을 완료하고, 모든 조각의 orientation을 완료하여 cube를 맞춘다. 이후 permution에 필요한 회전 수를 늘리고 orientation의 회전 수를 0으로 만드는 경우를 찾아 한 번에 permutation과 orientation을 하도록 만들어 최대한 짧은 해법을 찾아내는 알고리즘이다.

WCA(World Cube Association) 주최 공식 세계대회에서 사람이 맞춘 단일 최고기록은 3.47초, 5회 평균기록은 5.09초이다. 사람이 2-phase 알고리즘처럼 맞추는 것은 불가능하므로 사람이 맞추기 위해서 공식 기반 해법이 여러가지 존재한다. 여기서 공식이란 cube의 특정 조각만을 움직이기 위해, 기호로 표현한 회전들을 나열하여 정형화 시킨 것이다. 이 중 제일 많이 사용하는 해법은 CFOP으로 Cross-F2L-OLL-PLL의 4단계를 거쳐 맞춘다. CFOP 해법을 기본으로 다른 해법의 일

부를 더해 보완하는 식으로 많이 사용한다. 이때 꼭짓점 조각을 corner 조각, 모서리 조각을 edge 조각이라 한다.

0. Cross: 첫 번째 층의 edge 조각의 orientaion과 permutation을 완료한다. 따로 공식들은 없으며 최대한 적은 회전수로 맞출 수 있도록 한다.

1. F2L(First 2 layers): 첫 번째 층과 두 번째 층의 orientation과 permutation을 완료한다. 총 23개의 공식(좌우대칭 제외)의 공식으로 이루어져 있다.

2. OLL(Orientation last layer): 마지막(세 번째) 층의 orientation을 완료한다. 총 57개의 공식으로 이루어져 있다.

3. PLL(Permutation last layer): 마지막 (세 번째) 층의 permutation을 완료한다. 총 21개의 공식으로 이루어져 있다.

그러나 C++로 이러한 알고리즘을 구현하기 위해서는 판단해야 할 경우의 수와 상황들이 매우 복잡하고 많기 때문에, 눈을 가리고 큐브를 맞추는 눈가리기(Blindfolded) 분야에서 사용하는 해법인 OP(Old Pochmann) algorithm을 이용하여 구현하고자 한다.

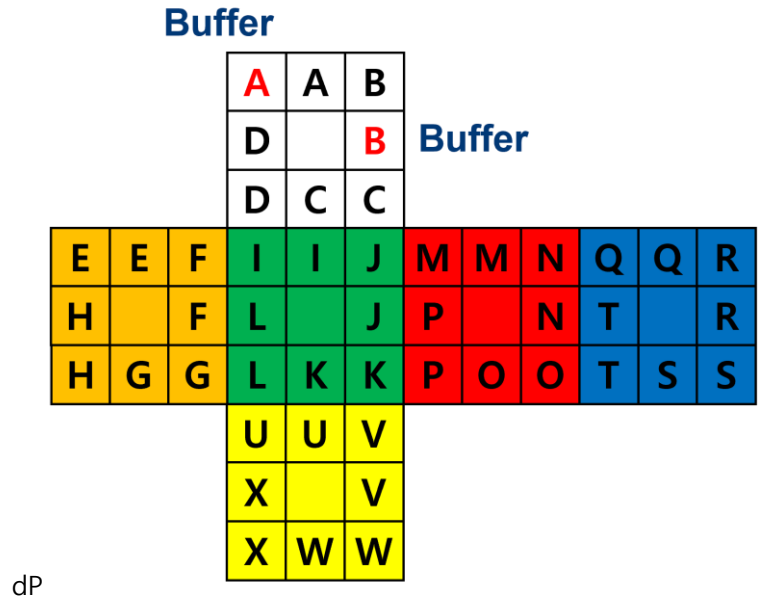
Ⅰ-2. 회전 기호와 Speffz 배치

Rubik's Cube의 공식을 표현하기 위해서 회전 기호를 이용한다. 회전하려는 면을 앞쪽 면으로 하여 시계방향으로 90도 회전시키는 것을 하나의 기호로 표현한다. 6개의 면에서 위쪽 면 회전을 U, 아래쪽 면 회전을 D, 오른쪽 면 회전을 R, 왼쪽 면 회전을 L, 앞쪽 면 회전을 F, 뒤쪽 면 회전을 B로 표현한다. 또한, 이를 확장하여 180도 회전을 회전기호에 2를 붙여 R2와 같이 표현하거나, 반시계방향 회전을 R'과 같이 표현할 수 있다. 최소회전 대회의 WCA(World Cube Association) 공식 규칙에서 R2 혹은 R'과 같은 회전은 한 회전으로 간주한다. 이러한 방식을 half turn metric이라고 한다.

자주 사용되지는 않지만 두 층 회전기호를 회전 기호에 w를 붙여 Rw 혹은, 소문자로 r과 같이 나타낸다. 또한 중간층 회전으로 Uw' U 를 합해 E, Fw F'를 합해 S, Rw R' 을 합해 M으로 나타낸다. 그러나 WCA(World Cube Association)에서 중간층 회전은 두 회전으로 간주하며, 두 층 회전은 마주보는 면 회전과 동일하고 OP Algorithm 공식에서 Lw, Dw만이 사용된다. 따라서 Lw, Dw의 두 개와 관련한 회전기호를 제외한 두 층 회전이나 중간층 회전 기호는 구현하지 않을 것이다.

Speffz 배치는 Ville Seppanen과 Rob Holf가 제안한 배치로, 각 면에다 알파벳으로 이름을 붙여, OP Algorithm에서 맞출 조각의 순서를 쉽게 외울 수 있도록 해준다. 초록색 면을 앞면으로, 흰색

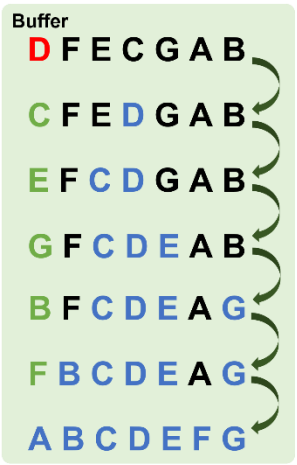
면을 윗면으로 볼 때(WCA 공식), 다음과 같이 corner 면 24개에 A ~ X, edge 면 24개에 A ~ X를 붙여 사용한다.



dP

이때 corner A와 edge B를 buffer라고 하며, 이후 OP Algorithm을 설명할 때 사용될 기준 조각이다.

I - 3. OP Algorithm



OP Algorithm의 기본 원리는 맞춰야 할 조각을 Buffer 조각부터 시작해서 순서대로 외우고, 이 순서대로 조각을 바꿔가며 맞추는 것이다. "D F E C G A B"라는 문자열을 예시로 OP Algorithm의 원리를 이용해서 순서를 정렬해보자.

1. A가 들어갈 위치를 buffer로 가정한다. 여기에는 D가 들어있다.
2. D가 들어갈 위치는 C이므로, C와 D를 교환한다.
3. C가 들어갈 위치는 E이므로, E와 C를 교환한다.
4. E가 들어갈 위치는 G이므로, G와 E를 교환한다.
5. G가 들어갈 위치는 B이므로, B와 G를 교환한다.
6. B는 들어갈 위치는 F이므로, F와 B를 교환한다.
7. F가 들어갈 위치는 A이므로, B와 A를 교환한다.
8. A가 들어갈 위치는 buffer이므로 종료한다.

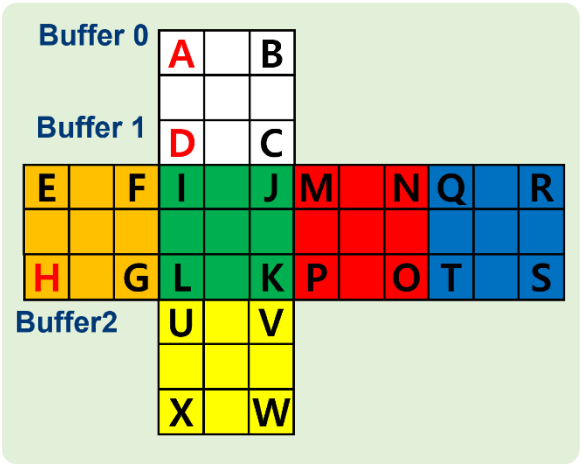
굴러온 돌이 박힌 돌 빼낸다는 속담처럼, buffer에서 시작하여 맞추려는 조각을 원래 있던 위치로 보내고, 이 과정에서 밀려난 조각을 다시 원래 위치로 보내는 것을 연쇄적으로 진행하며 이 순서를 외운다. 이후 순서대로 buffer가 포함된 조각과 나머지 하나의 조각을 교환하는 cube 공식을 사용하여 모든 cube의 조각을 맞출 수 있을 것이다.

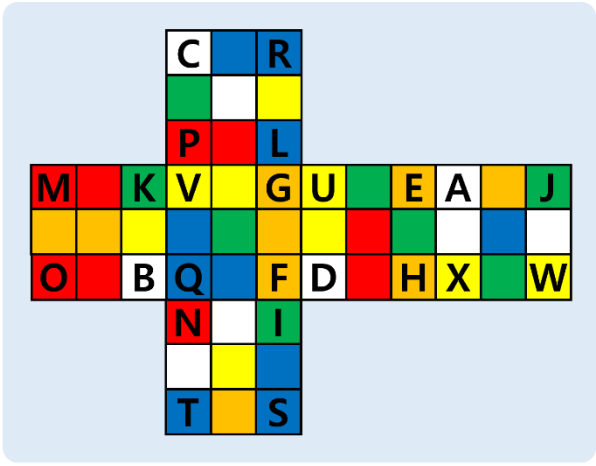
OP algorithm은 예외가 없다면, 모든 corner의 orientation과 permutation → 모든 Corner의 orientation과 permutation의 두 단계로 구성되어 있다. 위에서 문자열을 정렬하는 예시처럼, 조각을 맞추는 순서를 요약하면 다음과 같다.

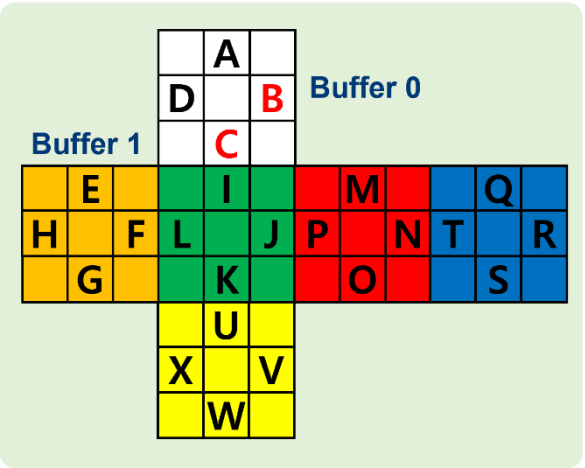
1. 조각의 한 면을 buffer로 설정(Ex: corner의 A와 edge의 B)
2. Buffer에 있는 조각이 원래 있어야할 위치의 speffz 배치 찾기
3. 찾은 조각이 원래 있어야할 위치의 speffz 배치 찾기
4. 3번을 반복하여 buffer가 포함된 조각으로 돌아올 때까지 반복
5. Buffer 막힘이 있다면, 새로운 buffer를 설정하여 사용되지 않은 조각에 대해 2~4 반복
6. 2~5번 까지의 speffz 배치의 나열을 Queue에 저장하여 공식 적용

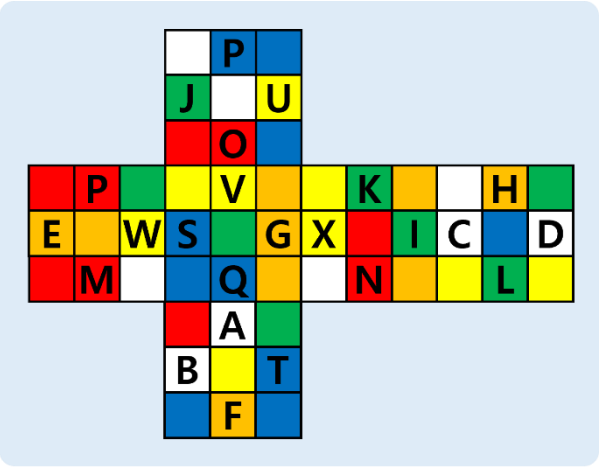
buffer에서 시작하여 buffer가 포함된 조각으로 돌아오면 이를 1 cycle이라 하며, 모든 corner 조각을 돌지 않았음에도 buffer로 돌아와 1 cycle이 형성될 수 있는데, 이를 buffer 막힘이라고 부른다. 이럴 경우 맞춰지지 않은 조각에서 임의로 한 면을 새로운 buffer로 만들어 새로운 cycle을 만든다. 여기서 새로운 buffer를 설정했기 때문에 첫 cycle에서는 buffer가 포함된 조각을 만나면 그 전에 마치지만, 두 번째 cycle 이후에서는 새로운 buffer가 포함된 조각이 원래 있어야할 위치까지 가야만 새로운 buffer를 임의로 설정했던 것을 돌려놓을 수 있다.

예시를 통해 알아보도록 하자. Cube를 섞기 위한 회전 기호의 나열을 scramble이라고 하는데, "R' L B L2 F' L D F' L B2 D R2 B2 U2 B2 L2 U' R2 D' R2"의 scramble로 큐브를 섞으면 다음과 같이 섞인다.

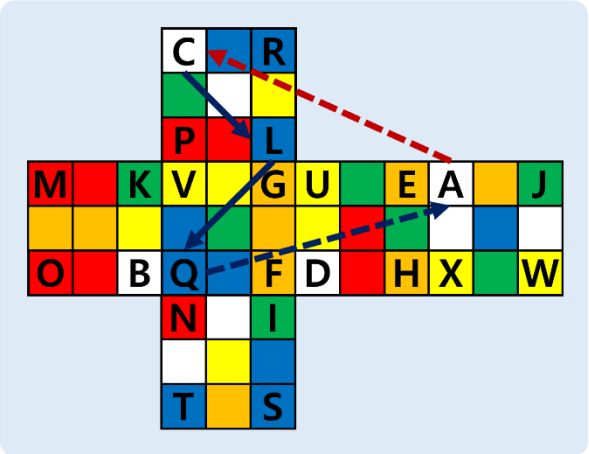




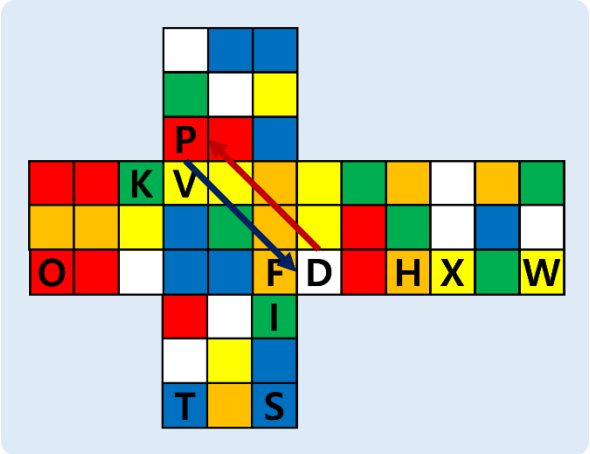




먼저 corner 조각부터 맞추면 아래와 같은 과정을 따라 맞출 순서를 결정할 수 있다.



Cycle 1: C L Q (A)



Cycle 2: P D P

첫 buffer A에 들어있는 조각의 면이 원래 있어야할 위치는 C이다. (특정 위치) → (특정 위치에 들어있는 조각의 면이 원래 있어야할 위치)의 형식으로 간단하게 나열하면 다음과 같다.

(A) → C / C → L / L → Q / Q → (A)

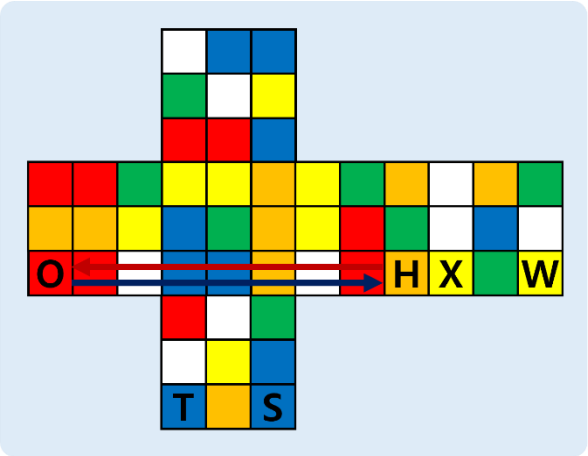
따라서 첫 cycle은 C L Q이다.

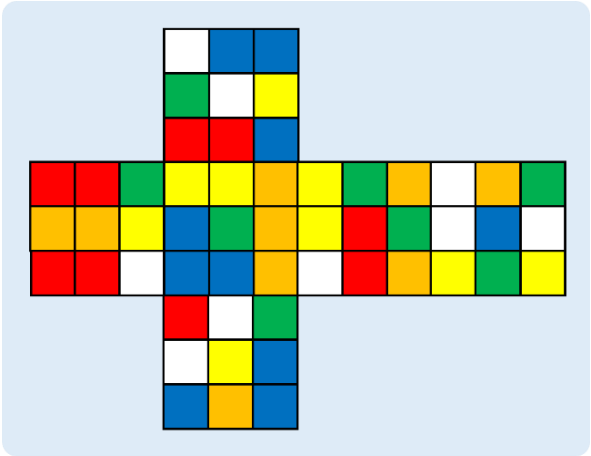
Buffer 막힘이 발생했으므로, 맞춰지지 않은 조각 중 새로운 buffer 위치를 설정한다. D 위치에 있는 조각이 맞춰지지 않았으므로, D를 새로운 buffer로 위의 과정을 반복하면 다음과 같다.

(C) → P / P → D / D → P

따라서 두 번째 cycle은 P D P 이다.

Buffer D를 만났지만 종료하지 않고, D 위치로 이동하여 P까지 추가해 주어야 새로운 buffer도 원래 자리에 돌려놓고 마칠 수 있다.





Cycle 3: O H O **C L Q P D P O H O**

다시 한 번 buffer 막힘이 발생하여 맞춰지지 않은 조각 중 새로운 buffer 위치를 설정한다. H 위치에 있는 조각이 맞춰지지 않았으므로, H를 새로운 buffer로 위의 과정을 반복하면 다음과 같다.

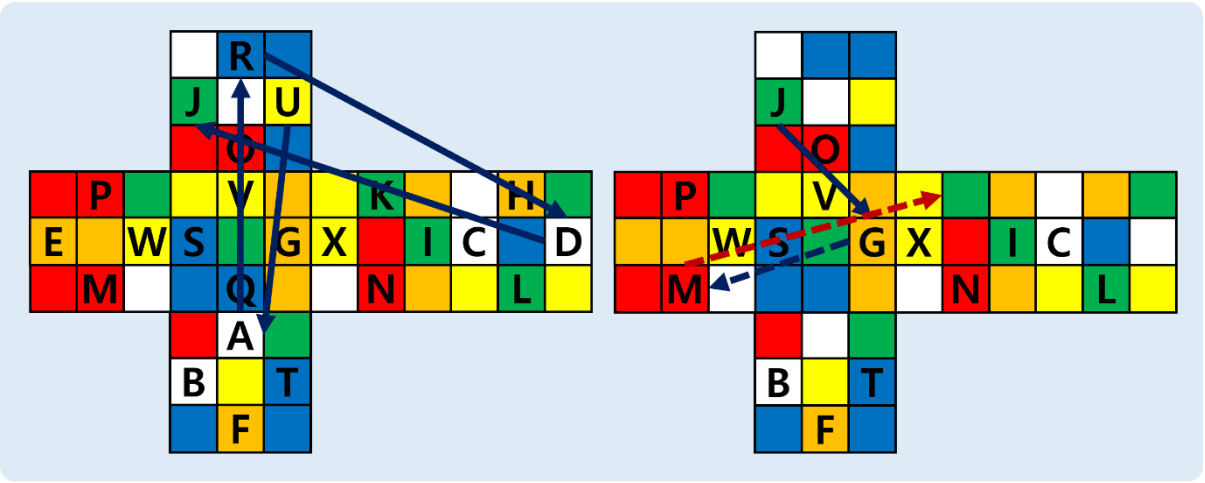
(H) → O / O → H / H → O

따라서 두 번째 cycle은 O H O 이다.

마찬가지로 buffer H를 만났지만 종료하지 않고, H 위치로 이동하여 O까지 추가해 주어야 새로운 buffer도 원래 자리에 돌려놓고 마칠 수 있다. 이제 사용되지 않은 조각이 없으므 로 종료한다.

최종적으로 corner를 맞추는 순서는 C L Q P D P O H O 임을 알 수 있다.

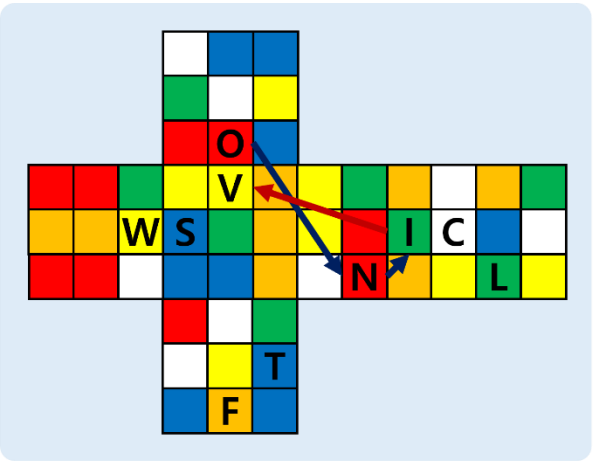
마찬가지로 edge 조각 또한 동일한 과정으로 맞추는 순서를 구하면 아래와 같다.



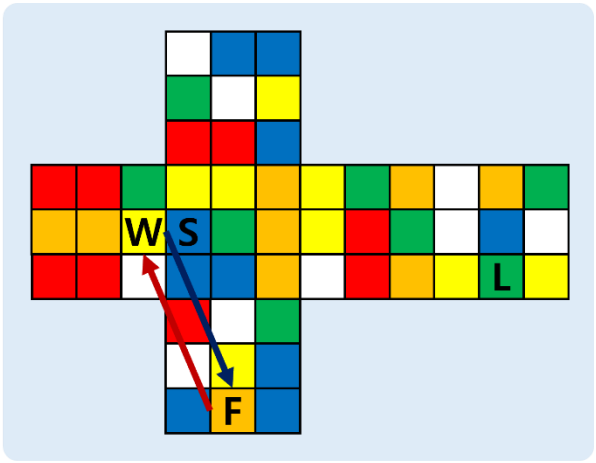
Cycle 1: U A R D J G (M)

(B) → U / U → A / A → R / R → D / D → J / J → G / G → (M)

B가 아닌 M에서 종료하는 이유는 B가 포함된 edge 조각의 다른 면이 M이므로, 위에서 언급했듯이 buffer가 '포함된' 조각을 만나면 종료하기 때문이다. 따라서 첫 번째 cycle은 U A R D J G 이다.



Cycle 2: O N I V



Cycle 3: W F W

Buffer 막힘이 발생하여 C 위치를 새로운 buffer로 설정하여 위 과정을 반복하면 다음과 같다.

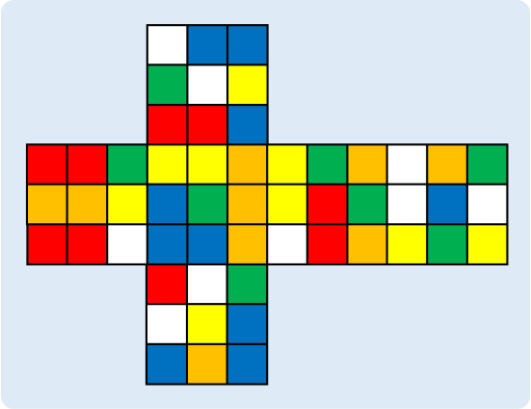
(C) → O / O → N / N → I / I → V

마찬가지로 C면이 포함된 조각의 다른 면은 I이므로 I를 만나면 I가 들어갈 위치를 하나 마지막에 추가하여 변경된 buffer도 원래 자리에 돌려놓고 마친다. 따라서 두 번째 cycle은 O N I V 이다.

다시 한 번 buffer 막힘이 발생하여 F 위치를 새로운 buffer로 설정하여 위 과정을 반복하면 다음과 같다.

(F) → W / W → F / F → W

따라서 세 번째 cycle은 W F W 이다. 더 이상 사용되지 않은 조각이 없으므로 여기서 종료한다.



U A R D J G O N I V W F W

최종적으로 edge 조각을 맞추는 순서는 U A R D J G O N I V W F W 이다.

I - 4. OP Algorithm - Formula

모든 corner 조각과 edge 조각의 맞추는 순서를 결정했다. 이제 공식을 사용하여 모든 조각을 순서에 따라 맞추는 과정만이 남았다.

기본적으로 corner 두개를 교환하는 Modified Y-Perm(이하 Y-Perm)과 edge 두 개를 교환하는 J-Perm 및 T-Perm을 사용한다. 각각의 공식은 아래와 같다.

"Modified Y-perm"
R U' R' U' R U R' F' R U R' U' R' F R

"J-perm"
R U R' F' R U R' U' R' F R2 U' R' U'

"T-perm"
R U R' U' R' F R2 U' R' U' R U R' F'

먼저 corner 조각 두 개를 교환하는 Y-Perm은, 오른쪽 그림과 같이 buffer(주황색)와 target(보라색) 두 개를 바꾸는 공식이다. 따라서 맞추기 위해서 원하는 면을 적절하게 보라색 위치로 이동시킨 뒤(Setup move), 공식을 사용한 후 다시 이동시킨 조각을 원래대로 되돌려 놓는(Reverse setup move) 과정이 필요하다. 이때 setup move와 reverse setup move는 buffer를 건드리면 안 되므로 R, F, D 회전만을 이용한다. 각 corner의 면 24개 별로 Setup move – Modified Y-Perm – Reverse setup move 를 효율적으로 정리한 공식표는 오른쪽 사진과 같다.

코너 위치	U 한	영		F		공식
UFL	가	D				F R' Y R F'
~~UBL~~	~~나~~	~~A~~				~~버퍼 조각~~
UBR	다	B				R D' Y D R'
UFR	라	C				F Y F'
DFL	마	U				F' Y F
DBL	바	X				D F' Y F D'
DBR	사	W				R2 F Y F' R2
DFR	아	V				F' R' Y R F
LFU	고	F				F2 Y F2
~~LBU~~	~~노~~	~~E~~				~~버퍼 조각~~
RBU	도	N				R2 Y R2
RFU	로	M				R' Y R
LFD	모	G				D2 R Y R' D2
LBD	보	H				D2 Y D2
RBD	소	O				R Y R'
RFD	오	P				Y
FLU	강	I				F' D Y D' F
~~BLU~~	~~냥~~	~~R~~				~~버퍼 조각~~
BRU	당	Q				R' F Y F' R
FRU	랑	J				R2 D' Y D R2
FLD	망	L				D Y D'
BLD	방	S				D' R Y R' D
BRD	상	T				D' Y D
FRD	앙	K				R F Y F' R'

[공식 사진 출처]

https://cafe.naver.com/cubemania/1063789

https://cafe.naver.com/cubemania/1063780

https://cafe.naver.com/cubemania/1063776

마찬가지로 edge 조각 역시 corner 조각과 유사하게 공식을 사용할 수 있다.

T-Perm과 J-Perm 역시 오른쪽 그림과 같이 buffer(주황색)와 target(분홍색) 두 개를 바꾸는 공식이므로 원하는 면을 적절하게 분홍색 면에 이동시키는 과정이 필요하다. 따라서 edge 역시 Setup move – Modified Y-Perm – Reverse setup move 꼴의 회전을 통해 두 조각을 교환할 수 있다.

공식 T의 타겟

공식 J의 타겟

마찬가지로 setup move와 reverse setup move는 buffer를 건드리면 안 되므로 U, F, B, R 회전을 사용할 수 없다. 따라서 L, D 회전만을 이용해야 하므로 corner에 비해서는 조금 더 복잡해진다.

각 edge의 면 24개 별로 Setup move – T-Perm / J-Perm – Reverse setup move 를 효율적으로 정리한 공식 표는 오른쪽 사진과 같다.

에지 위치	U 한	영	F		공식
UL	가	D			T
UB	나	A			Lw2 D' L2 T L2 D Lw2
~~UR~~	~~다~~	~~B~~			~~버퍼 조각~~
UF	라	C			J
FL	마	L			L' T L
BL	바	R			L T L'
BR	사	T			Dw2 L' T L Dw2
FR	아	J			Dw2 L T L' Dw2
DL	자	X			L2 T L2
DB	차	W			D L2 T L2 D'
DR	카	V			D2 L2 T L2 D2
DF	타	U			D' L2 T L2 D
LU	강	E			L' Dw L' T L Dw' L
BU	낭	Q			Lw J Lw'
~~RU~~	~~당~~	~~M~~			~~버퍼 조각~~
FU	랑	I			Lw D' L2 T L2 D Lw'
LF	망	F			Dw' L T L' Dw
LB	방	H			Dw L' T L Dw'
RB	상	N			Dw L T L' Dw'
RF	앙	P			Dw' L' T L Dw
LD	장	G			D Lw' J Lw D'
BD	창	S			Lw' D' L2 T L2 D Lw
RD	캉	O			D' Lw' J Lw D
FD	탕	K			Lw' J Lw

[사진 출처]
https://cafe.naver.com/cubemania/1063778

I - 4. OP Algorithm - Parity

이 과정이 끝이 아니다. 위에서 살짝 예외에 대한 언급을 했었는데, 예외를 parity 라고 부른다.

위의 Y, J, T 공식의 영향을 받는 조각을 보면 파란색으로 색칠되어 있다. 즉, corner 조각을 바꿀 때 관련 없는 edge 조각 2개를 바꿨고, edge 조각을 바꿀 때 관련 없는 edge 조각 2개를 바꿨다.

따라서 공식 사용 순서의 길이가 짝수이면, 관련 없는 조각을 교환하고, 한 번 더 교환하면 원래대로 돌아와 최종적으로는 영향을 받지 않은 것과 같다. 그러나 홀수이면, 영향을 받은 채로 남아있게 되어 이를 원래대로 돌리는 과정이 필요하다. 그러나 cube 특징상 원하는 두 개의 조각을 교환하는 방법은 없지만, 다행이도 corner 공식 사용 순서의 길이가 짝수면, edge 공식 사용 순서의 길이가 짝수이고, 홀수일 경우에도 마찬가지이다. 따라서 총 4개의 조각을 교환하면 되는데, 위에서 파란색으로 표시된 4개의 조각을 2개씩 서로 교환하는 공식이 존재하고 이를 Ra-Perm이라 한다.

"Ra-perm"
R U R' F' R U2 R' U2 R' F R U R U2 R' U'

Corner 조각 공식 사용 – Ra-Perm 사용 – Edge 조각 공식 사용의 순서를 거치면, 이제 최종적으로 cube가 모두 맞춰지게 될 것이다.

[사진 출처]

https://cafe.naver.com/cubemania/1063776

위에서 보았던 예시인 "R' L B L2 F' L D F' L B2 D R2 B2 U2 B2 L2 U' R2 D' R2"는 따라서 다음과 같은 과정을 거치면 cube가 모두 맞춰진다(단, 공식표에서 Setup move, Y/T/J-Perm, Reverse setup move 사이에 존재하는 중복 기호, 회전-역회전 기호들은 모두 합치거나 제거했다).

Corner Solving[Modified Y-Perm]: C L Q P D P O H O
→ C: F R U' R' U' R U R' F' R U R' U' R' F R F'
→ L: D R U' R' U' R U R' F' R U R' U' R' F R D'
→ Q: R' F R U' R' U' R U R' F' R U R' U' R' F R F' R
→ P: R U' R' U' R U R' F' R U R' U' R' F R
→ D: F U' R' U' R U R' F' R U R' U' R' F R2 F'
→ P: R U' R' U' R U R' F' R U R' U' R' F R
→ O: R2 U' R' U' R U R' F' R U R' U' R' F
→ H: D2 R U' R' U' R U R' F' R U R' U' R' F R D2
→ O: R2 U' R' U' R U R' F' R U R' U' R' F

[Parity Exists] Ra-Perm(R U R' F' R U2 R' U2 R' F R U R U2 R' U') Used

Edge Solving[T-Perm & J-Perm]: U A R D J G O N I V W F W
→ U: D' L2 R U R' U' R' F R2 U' R' U' R U R' F' L2 D
→ A: Lw2 D' L2 R U R' U' R' F R2 U' R' U' R U R' F' L2 D Lw2
→ R: L R U R' U' R' F R2 U' R' U' R U R' F' L'
→ D: R U R' U' R' F R2 U' R' U' R U R' F'
→ J: Dw2 L R U R' U' R' F R2 U' R' U' R U R' F' L' Dw2
→ G: D Lw' R U R' F' R U R' U' R' F R2 U' R' U' Lw D'
→ O: D' Lw' R U R' F' R U R' U' R' F R2 U' R' U' Lw D

→ N: Dw L R U R' U' R' F R2 U' R' U' R U R' F' L' Dw'

→ I: Lw D' L2 R U R' U' R' F R2 U' R' U' R U R' F' L2 D Lw'

→ V: D2 L2 R U R' U' R' F R2 U' R' U' R U R' F' L2 D2

→ W: D L2 R U R' U' R' F R2 U' R' U' R U R' F' L2 D'

→ F: Dw' L R U R' U' R' F R2 U' R' U' R U R' F' L' Dw

→ W: D L2 R U R' U' R' F R2 U' R' U' R U R' F' L2 D'

Ⅱ. Queue 구현

Ⅱ-1. queue.h

조각 정렬 순서를 담고, 앞에서부터 순차적으로 적용하기 적절한 자료 구조로써 Queue가 있다. queue.h 에서는 Queue를 만들기 위한 Node 클래스, Queue 클래스를 선언해 두었다.

```
class Node
private:
    std::string symbol;
    Node* next;
    Node* prev;
public:
    Node(std::string);
    Node* getNext();
    Node* getPrev();
    std::string getSymbol();

    void setNext(Node*);
    void setPrev(Node*);
```

[Node 클래스]

private: 외부에서 접근이 불가능하도록, 노드의 데이터를 담는 변수들은 private로 선언했다.

std::string symbol 회전 기호를 저장할 문자열이다.

Node* next : 다음 노드 주소를 저장할 포인터이다.

Node* prev : 이전 노드 주소를 저장할 포인터이다.

public: 각 데이터에 접근하기 위한 메소드는 외부에서 이용할 수 있도록 public 영역에 두었다.

Node(std::string) : Node의 생성자로, 인자로 문자열을 받아 symbol을 초기화한다. 또한 next와 prev를 NULL로 초기화 한다.

Node* getNext() : next를 return한다.

Node* getPrev() : prev를 return한다.

std::string getSymbol() : symbol을 return한다.

void setNext(Node*) : 인자로 받은 주소로 next를 초기화한다.

void setPrev(Node*) : 인자로 받안 주소로 prev를 초기화한다.

```
class Queue
private:
    Node* head = NULL;
    Node* tail = NULL;
public:
    Queue();
    ~Queue();
    void insertBack(std::string);
    bool popFront();
    Node* gethead();
    void printQueue();
    void inputtoqueue(std::string);
```

[Queue 클래스]

private: 외부에서 접근이 불가능하도록 Queue의 head와 tail은 private로 선언했다.

Node* head : Queue에 삽입된 노드 중 제일 첫 번째 노드를 가리키는 포인터이다.

Node* tail : Queue에 삽입된 노드 중 제일 마지막 노드를 가리키는 포인터이다.

public: 외부에서 접근이 가능하도록 Queue를 활용하기 위한 메소드는 public으로 선언했다.

Queue() : Queue의 생성자로 head와 tail을 NULL로 초기화 한다.

~Queue() : Queue의 파괴자로 모든 Node에 할당된 동적 메모리를 해제한다.

void insertback(std::string) : 회전 기호를 인자로 받아 새로운 Node를 Queue 뒤에 삽입하는 메소드이다.

bool popFront() : Queue의 맨 뒤의 노드를 삭제하는 메소드이다.

Node* gethead() : 이후 Queue에서 순서대로 노드를 뽑아가며 Cube에 적용하기 위해, head의 주소가 필요하여 이를 반환하는 메소드이다.

void printQueue() : 디버깅 용으로 사용할 Queue를 출력하는 메소드이다.

void inputtoqueue(std::string) : Scramble(여러 회전 기호를 ' '로 구분한 문자열)을 입력받아 ' '를 기준으로 끊어서 Queue에 하나씩 삽입하기 위한 메소드이다.

Ⅱ-2. queue.cpp

```
Node::Node(std::string symbol) {
        this->symbol = symbol;
        this->next = NULL;
        this->prev = NULL;
}

Node* Node::getNext() {
        return this->next;
}

Node* Node::getPrev() {
        return this->prev;
}

std::string Node::getSymbol() {
        return this->symbol;
}

void Node::setNext(Node* next) {
        this->next = next;
}

void Node::setPrev(Node* prev) {
        this->prev = prev;
}
```

저장할 회전 기호, 다음 노드와 이전 노드의 주소를 데이터로 갖는 Node 클래스의 구현부이다.

```
Queue::Queue() {
        head = NULL;
        tail = NULL;
}

Queue::~Queue() {
        Node* deleteNode = head;
        while (deleteNode != NULL) {
                head = head->getNext();
                delete deleteNode;
                deleteNode = head;
        }
```

```
}

void Queue::insertBack(std::string symbol) {
        Node* newNode = new Node(symbol);
        Node* temp = tail;
        if (head == NULL) {
                head = newNode;
                tail = newNode;;
        } else {
                tail->setNext(newNode);
                tail = newNode;
                tail->setPrev(temp);
        }
}

bool Queue::popFront() {
        Node* deleteNode = head;
        if (head == NULL) {
                std::cout << "The Queue is empty." << std::endl;
                return false;
        }
        if (head->getNext() == NULL) {
                head = NULL;
        }
        else {
                head = head->getNext();
                head->setPrev(NULL);
        }
        delete deleteNode;
        return true;
}

void Queue::printQueue() {
        Node* currentNode = head;
        if (head == NULL) {
                std::cout << "The Queue is empty." << std::endl;
        } else {
                while (currentNode != NULL) {
                        std::cout << currentNode->getSymbol();
                        if (currentNode->getNext() != NULL) {
                                std::cout << " > ";
                        }
                        currentNode = currentNode->getNext();
                }
                std::cout << std::endl;
        }

}

Node* Queue::gethead() {
        return this->head;
}
```

Queue의 구현부이다.

[~Queue] 삭제할 노드의 주소를 담을 deleteNode 변수를 선언하여 head(첫 노드의 주소)로 초기화하고, getNext 메소드를 이용하여 deleteNode != NULL일 때까지 다음 노드로 이동하면서 모든 Node에 동적 할당된 메모리를 해제한다.

[insertBack] 새로운 Node를 만들어 인자로 받은 symbol 문자열(회전 기호)로 초기화한다. 이후 Queue가 비어 있다면(head == NULL), 첫 노드의 주소(head)와 마지막 노드의 주소(tail)에 새 노

드의 주소를 대입하고 마친다. 하나 이상의 노드가 있다면, 마지막 노드(tail)의 다음 노드 주소를 새로운 노드의 주소로 하고, tail에는 새 노드의 주소를 대입하여 마지막 노드를 새 노드로 바꾼 뒤, 새 노드(tail)의 이전 노드 주소를 temp에 임시로 저장해둔 기존의 마지막 노드 주소로 설정한다.

[popFront] 삭제할 노드의 주소를 담을 deleteNode 변수를 선언하여 head(첫 노드의 주소)로 초기화한다. 이후 Queue가 비어 있다면(head == NULL), 에러 메시지를 출력하고 false를 반환하여 함수를 종료한다. 하나 이상의 노드만 있다면(head->getNext() == NULL), 첫 노드의 주소(head)를 NULL로 설정하고, 첫 번째 노드(deleteNode)에 할당된 메모리를 해제한 뒤에 true를 반환하여 함수를 종료한다. 두 개 이상의 노드가 있다면, 첫 노드의 주소를(head) 두 번째 노드로 설정하고, 두 번째 노드(head)의 이전 노드 주소를 NULL로 설정한 뒤에 첫 노드(deleteNode)에 할당된 메모리를 해제하고 true를 반환하여 함수를 종료한다.

[printQueue] 출력할 노드의 주소를 담을 currentNode를 선언하여 첫 노드의 주소(head)로 초기화한다. 이후 노드가 없다면 에러 메시지를 출력하고, 하나 이상의 노드가 있다면 currentNode가 NULL이 아닐 때까지 getNext를 통해 다음 노드로 이동해가며 getSymbol을 이용하여 symbol을 출력한다.

```
void Queue::inputtoqueue(std::string symbols) { // 문자열을 ' '를 기준으로 끊어서 각각
Queue에 한 Node로 저장하는 함수
        int index = -1;
        std::string symbol = "";
        char tempsymbol;

        for (int i = 0; i < symbols.length(); i++) {
                tempsymbol = symbols[i];
                if (tempsymbol == ' ') {
                        symbol = symbols.substr(index + 1, i - index - 1);
                        index = i;
                        this->insertBack(symbol);
                }
        }
        symbol = symbols.substr(index + 1, symbols.length());
        this->insertBack(symbol);
}
```

[inputtoqueue] 회전 기호를 ' '로 구분하여 배열한 문자열을 인자(symbols)로 받는다. 문자열 앞에서부터 ' '의 index를 저장할 index 변수, 문자열에서 뽑은 하나의 회전 기호를 저장할 symbol, 앞에서부터 차례대로 하나의 문자를 임시로 대입할 tempsymbol 변수를 선언한다. for문으로 문자 하나 하나씩 끊어가며 문자열의 끝까지 반복하는데, ' ' 문자열을 만날 때마다 index에 담아둔 이전 문자열 위치의 다음부터 현재의 ' ' 문자열 전까지 substr로 추출하고, 현재의 ' ' 위치를 다시 index에 갱신한 후, 뽑아낸 문자열(회전 기호 하나)을Queue에 삽입한다.

Ⅲ. Rubik's Cube 및 Cube 회전 구현

`Ⅲ-1. cube.h – Cube Class

Cube 본체와 회전을 구현한 클래스이다.

```
class Cube

private:
    char** up;
    char** down;
    char** front;
    char** back;
    char** left;
    char** right;

public:
    Cube();
    ~Cube();
    void printcube();
    void printSpeffz();
    void cubeReset();
    void rotatingCube(bool, int);
    void inputtoQueue(std::string);

protected:
    char** up_speffz;
    char** down_speffz;
    char** front_speffz;
    char** back_speffz;
    char** left_speffz;
    char** right_speffz;
    Queue scramble;

    void U();
    void U2();
    void Ur();
    void D();
    void D2();
    void Dr();
    void Dw();
    void Dw2();
    void Dwr();
    void R();
    void R2();
    void Rr();
    void L();
    void L2();
    void Lr();
    void Lw();
    void Lw2();
    void Lwr();
    void F();
    void F2();
    void Fr();
    void B();
    void B2();
    void Br();
```

[Cube 클래스]

public: 외부에서 접근을 막기 위해 Cube의 각 면의 3*3 포인터 배열을 private로 선언한다.

char** up ~ right : 이름과 같이 위쪽 면, 아래쪽 면, 앞쪽 면, 뒤쪽 면, 왼쪽 면, 오른쪽 면에 해당하는 3*3 포인터 배열이다.

protected: Cube를 맞추기 위한 cubeSolver 클래스를 이후에 상속시키면, 여기서 cube의 회전이나 speffz 기호를 직접적으로 사용해야 하기 때문에, 캡슐화를 위해 Cube Solving이나 회전과 관련된 변수 및 메소드는 protected로 선언한다.

char** up_speffz ~ right_speffz : 각 면의 speffz 배치를 저장할 3*3 포인터 배열이다.

Queue scramble : 이후 Cube를 섞거나 맞출 Solution을 저장하고 앞에서부터 차례대로 Cube를 회전시키기 위한 Queue

void U() ~ Br() : 각 회전 기호에 해당하는 Cube 회전 메소드

public: 큐브를 출력하거나 초기화, 입력한 회전 기호를 적용하고 회전 기호 대로 회전시키는 메소드는 상속을 시키더라도 실제 사용자가 쓸 수 있도록 public으로 선언한다. (즉, main에서 사용될 함수들)

Cube() : Cube 클래스의 생성자로, 각 면과 그 Speffz 배치 포인터에 동적 메모리를 할당시키고, 색깔이나 Speffz 배치를 초기화한다.

~Cube() : Cube 클래스의 파괴자로, 생성자에서 할당한 동적 메모리를 해제한다.

void printCube() : 색을 적용하여 콘솔에 전개도 모양으로 Cube를 출력하는 메소드이다.

void printSpeffz() : 디버깅 용도로 만든 Speffz 배치를 출력하는 메소드이다.

void rotatingCube(bool, int) : 느리게 회전 여부 및 회전 수를 인자로 받아 Cube를 실제로 회전시키는 메소드이다.

inputtoQueue : Cube를 맞추는 경우에는 Cube에 상속된 cubeSolver 클래스에서 직접 scramble에 접근할 수 있기 때문에 Queue 클래스의 inputoqueue를 사용할 수 있지만, Cube를 섞는 경우는 직접 main에서 입력 받은 문자열을 넣기 위한 메소드이다.

Ⅲ-2. cube.cpp – Cube Class 구현

```
char** setcubeface(char color) {
        char** cube = new char*[4];
        for (int i = 0; i < 3; i++) {
                cube[i] = new char[4];
                for (int j = 0; j < 3; j++) {
                        cube[i][j] = color;
                }
        }
        return cube;
}

Cube::Cube() {
        up = setcubeface('W');
        down = setcubeface('Y');
        front = setcubeface('G');
        back = setcubeface('B');
        left = setcubeface('O');
        right = setcubeface('R');

        up_speffz = new char* [3];
        for (int i = 0; i < 3; i++) up_speffz[i] = new char[3];
        down_speffz = new char* [3];
        for (int i = 0; i < 3; i++) down_speffz[i] = new char[3];
        left_speffz = new char* [3];
        for (int i = 0; i < 3; i++) left_speffz[i] = new char[3];
        right_speffz = new char* [3];
        for (int i = 0; i < 3; i++) right_speffz[i] = new char[3];
        front_speffz = new char* [3];
        for (int i = 0; i < 3; i++) front_speffz[i] = new char[3];
        back_speffz = new char* [3];
        for (int i = 0; i < 3; i++) back_speffz[i] = new char[3];

        up_speffz[0][0] = up_speffz[0][1] = 'A';
        up_speffz[0][2] = up_speffz[1][2] = 'B';
        up_speffz[2][2] = up_speffz[2][1] = 'C';
        up_speffz[2][0] = up_speffz[1][0] = 'D';
        up_speffz[1][1] = ' ';
        left_speffz[0][0] = left_speffz[0][1] = 'E';
        left_speffz[0][2] = left_speffz[1][2] = 'F';
        left_speffz[2][2] = left_speffz[2][1] = 'G';
        left_speffz[2][0] = left_speffz[1][0] = 'H';
        left_speffz[1][1] = ' ';
        front_speffz[0][0] = front_speffz[0][1] = 'I';
```

```
        front_speffz[0][2] = front_speffz[1][2] = 'J';
        front_speffz[2][2] = front_speffz[2][1] = 'K';
        front_speffz[2][0] = front_speffz[1][0] = 'L';
        front_speffz[1][1] = ' ';
        right_speffz[0][0] = right_speffz[0][1] = 'M';
        right_speffz[0][2] = right_speffz[1][2] = 'N';
        right_speffz[2][2] = right_speffz[2][1] = 'O';
        right_speffz[2][0] = right_speffz[1][0] = 'P';
        right_speffz[1][1] = ' ';
        back_speffz[0][0] = back_speffz[0][1] = 'Q';
        back_speffz[0][2] = back_speffz[1][2] = 'R';
        back_speffz[2][2] = back_speffz[2][1] = 'S';
        back_speffz[2][0] = back_speffz[1][0] = 'T';
        back_speffz[1][1] = ' ';
        down_speffz[0][0] = down_speffz[0][1] = 'U';
        down_speffz[0][2] = down_speffz[1][2] = 'V';
        down_speffz[2][2] = down_speffz[2][1] = 'W';
        down_speffz[2][0] = down_speffz[1][0] = 'X';
        down_speffz[1][1] = ' ';
}
```

setcubeface 함수는 Cube 각 면의 색깔을 의미하는 약어 문자로 포인터에 3*3 배열을 동적 할당하고, 각 index에 초기화 시키는 함수이다. 이를 이용해 up ~ right 에 메모리를 할당하고 초기화한다.

```
Cube::~Cube() { // 동적 할당된 메모리 해제
        for (int i = 0; i < 3; i++) delete[] up[i];
        delete[] up;
        for (int i = 0; i < 3; i++) delete[] down[i];
        delete[] down;
        for (int i = 0; i < 3; i++) delete[] left[i];
        delete[] left;
        for (int i = 0; i < 3; i++) delete[] right[i];
        delete[] right;
        for (int i = 0; i < 3; i++) delete[] front[i];
        delete[] front;
        for (int i = 0; i < 3; i++) delete[] back[i];
        delete[] back;

        for (int i = 0; i < 3; i++) delete[] up_speffz[i];
        delete[] up_speffz;
        for (int i = 0; i < 3; i++) delete[] down_speffz[i];
        delete[] down_speffz;
        for (int i = 0; i < 3; i++) delete[] left_speffz[i];
        delete[] left_speffz;
        for (int i = 0; i < 3; i++) delete[] right_speffz[i];
        delete[] right_speffz;
        for (int i = 0; i < 3; i++) delete[] front_speffz[i];
        delete[] front_speffz;
        for (int i = 0; i < 3; i++) delete[] back_speffz[i];
        delete[] back_speffz;
}
```

```
void Cube::cubeReset() {
        for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++)  up[i][j] = 'W';
        for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++)  down[i][j] = 'Y';
        for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++)  front[i][j] = 'G';
        for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++)  back[i][j] = 'B';
        for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++)  left[i][j] = 'O';
```

```
        for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++)  right[i][j] = 'R';

        up_speffz[0][0] = up_speffz[0][1] = 'A';
        up_speffz[0][2] = up_speffz[1][2] = 'B';
        up_speffz[2][2] = up_speffz[2][1] = 'C';
        up_speffz[2][0] = up_speffz[1][0] = 'D';
        up_speffz[1][1] = ' ';
        left_speffz[0][0] = left_speffz[0][1] = 'E';
        left_speffz[0][2] = left_speffz[1][2] = 'F';
        left_speffz[2][2] = left_speffz[2][1] = 'G';
        left_speffz[2][0] = left_speffz[1][0] = 'H';
        left_speffz[1][1] = ' ';
        front_speffz[0][0] = front_speffz[0][1] = 'I';
        front_speffz[0][2] = front_speffz[1][2] = 'J';
        front_speffz[2][2] = front_speffz[2][1] = 'K';
        front_speffz[2][0] = front_speffz[1][0] = 'L';
        front_speffz[1][1] = ' ';
        right_speffz[0][0] = right_speffz[0][1] = 'M';
        right_speffz[0][2] = right_speffz[1][2] = 'N';
        right_speffz[2][2] = right_speffz[2][1] = 'O';
        right_speffz[2][0] = right_speffz[1][0] = 'P';
        right_speffz[1][1] = ' ';
        back_speffz[0][0] = back_speffz[0][1] = 'Q';
        back_speffz[0][2] = back_speffz[1][2] = 'R';
        back_speffz[2][2] = back_speffz[2][1] = 'S';
        back_speffz[2][0] = back_speffz[1][0] = 'T';
        back_speffz[1][1] = ' ';
        down_speffz[0][0] = down_speffz[0][1] = 'U';
        down_speffz[0][2] = down_speffz[1][2] = 'V';
        down_speffz[2][2] = down_speffz[2][1] = 'W';
        down_speffz[2][0] = down_speffz[1][0] = 'X';
        down_speffz[1][1] = ' ';
}
```

[cubeReset] 배열을 원래의 Cube로 초기화시킨다.

```
void Cube::printcube() {
        // 속도 향상을 위한 부분
        std::ios::sync_with_stdio(false);
        std::cin.tie(NULL);
        std::cout.tie(NULL);

        cur(0, 0); // System("cls");는 매우 속도가 느리므로 Cube 출력에는 사용 X

        std::cout << std::endl;
        for (int i = 0; i < 3; i++) {
                scolor(15, 0);
                std::cout << "        ";
                for (int j = 0; j < 3; j++) {
                        if (up[i][j] == 'W') scolor(0, 15);
                        else if (up[i][j] == 'Y') scolor(0, 14);
                        else if (up[i][j] == 'G') scolor(0, 10);
                        else if (up[i][j] == 'B') scolor(0, 9);
                        else if (up[i][j] == 'R') scolor(0, 12);
                        else if (up[i][j] == 'O') scolor(0, 13);
                        std::cout << "  ";
                }
                std::cout << std::endl;
        }
        std::cout << std::endl;
        for (int i = 0; i < 3; i++) {
                scolor(15, 0);
```

```
                std::cout << "  ";
                for (int j = 0; j < 3; j++) {
                        if (left[i][j] == 'W') scolor(0, 15);
                        else if (left[i][j] == 'Y') scolor(0, 14);
                        else if (left[i][j] == 'G') scolor(0, 10);
                        else if (left[i][j] == 'B') scolor(0, 9);
                        else if (left[i][j] == 'R') scolor(0, 12);
                        else if (left[i][j] == 'O') scolor(0, 13);
                        std::cout << "  ";
                }
                scolor(15, 0);
                std::cout << " ";
                for (int j = 0; j < 3; j++) {
                        if (front[i][j] == 'W') scolor(0, 15);
                        else if (front[i][j] == 'Y') scolor(0, 14);
                        else if (front[i][j] == 'G') scolor(0, 10);
                        else if (front[i][j] == 'B') scolor(0, 9);
                        else if (front[i][j] == 'R') scolor(0, 12);
                        else if (front[i][j] == 'O') scolor(0, 13);
                        std::cout << "  ";
                }
                scolor(15, 0);
                std::cout << " ";
                for (int j = 0; j < 3; j++) {
                        if (right[i][j] == 'W') scolor(0, 15);
                        else if (right[i][j] == 'Y') scolor(0, 14);
                        else if (right[i][j] == 'G') scolor(0, 10);
                        else if (right[i][j] == 'B') scolor(0, 9);
                        else if (right[i][j] == 'R') scolor(0, 12);
                        else if (right[i][j] == 'O') scolor(0, 13);
                        std::cout << "  ";
                }
                scolor(15, 0);
                std::cout << " ";
                for (int j = 0; j < 3; j++) {
                        if (back[i][j] == 'W') scolor(0, 15);
                        else if (back[i][j] == 'Y') scolor(0, 14);
                        else if (back[i][j] == 'G') scolor(0, 10);
                        else if (back[i][j] == 'B') scolor(0, 9);
                        else if (back[i][j] == 'R') scolor(0, 12);
                        else if (back[i][j] == 'O') scolor(0, 13);
                        std::cout << "  ";
                }
                std::cout << std::endl;
        }
        std::cout << std::endl;
        for (int i = 0; i < 3; i++) {
                scolor(15, 0);
                std::cout << "        ";
                for (int j = 0; j < 3; j++) {
                        if (down[i][j] == 'W') scolor(0, 15);
                        else if (down[i][j] == 'Y') scolor(0, 14);
                        else if (down[i][j] == 'G') scolor(0, 10);
                        else if (down[i][j] == 'B') scolor(0, 9);
                        else if (down[i][j] == 'R') scolor(0, 12);
                        else if (down[i][j] == 'O') scolor(0, 13);
                        std::cout << "  ";
                }
                std::cout << std::endl;
        }
        scolor(15, 0);
        std::cout << std::endl;
}
```

[printCube] scolor 함수를 만들어(뒤에 코드 첨부), 각 배열의 요소에 담긴 문자열의 종류에 따라

색을 콘솔에 출력한다. 출력 모양은 정육면체의 전개도 모양으로 공백의 길이를 조정한다. cur(뒤에 코드 첨부) 함수는 Cube를 한 번 출력할 때마다 콘솔을 비우기 위해 출력하는 커서를 강제로 왼쪽 위에 위치시켜 덮어 씌운다. System("cls")는 매우 속도가 느려서 이를 사용한다. 맨 위 세 줄은 printf보다 cout이 느린 단점을 조금이나마 해소하기 위한 부분이다.

```
void Cube::printSpeffz() {
	for (int i = 0; i < 3; i++) {
		std::cout << "      ";
		for (int j = 0; j < 3; j++) std::cout << up_speffz[i][j] << " ";
		std::cout << std::endl;
	}
	for (int i = 0; i < 3; i++) {
		for (int j = 0; j < 3; j++) std::cout << left_speffz[i][j] << " ";
		for (int j = 0; j < 3; j++) std::cout << front_speffz[i][j] << " ";
		for (int j = 0; j < 3; j++) std::cout << right_speffz[i][j] << " ";
		for (int j = 0; j < 3; j++) std::cout << back_speffz[i][j] << " ";
		std::cout << std::endl;
	}
	for (int i = 0; i < 3; i++) {
		std::cout << "      ";
		for (int j = 0; j < 3; j++) std::cout << down_speffz[i][j] << " ";
		std::cout << std::endl;
	}
}
```

```
void RotateCornerClockwise(char** face) {
	char temp_corner = face[0][0];
	face[0][0] = face[2][0];
	face[2][0] = face[2][2];
	face[2][2] = face[0][2];
	face[0][2] = temp_corner;
}
void RotateEdgeClockwise(char** face) {
	char temp_edge = face[0][1];
	face[0][1] = face[1][0];
	face[1][0] = face[2][1];
	face[2][1] = face[1][2];
	face[1][2] = temp_edge;
}
```

[RotateCornerClockwise], [RotateEdgeClockwise] Cube를 회전시킬 때, 회전시키려는 면 전체 회전과, 그 면에 연결된 옆면 4개의 층 한 개를 회전 시켜야 한다. 전자를 회전시킬 때는 모든 회전 기호 메소드가 동일하므로, 효율성을 위해 이를 함수로 따로 구현한 것이다.

아래부터는 회전 기호대로 회전하는 메소드이다. 2가 붙은 회전 기호는 기본 회전 기호를 2번, '가 붙은 회전 기호는 기본 회전 기호를 3번 실행하여 구현한다.

```
void Cube::U() {
	RotateCornerClockwise(up);
	RotateEdgeClockwise(up);
	RotateCornerClockwise(up_speffz);
	RotateEdgeClockwise(up_speffz);
```

```
        // 옆면 회전
        char temp_side[3] = {0, 0, 0};
        for (int i = 0; i < 3; i++) temp_side[i] = front[0][i];
        for (int i = 0; i < 3; i++) front[0][i] = right[0][i];
        for (int i = 0; i < 3; i++) right[0][i] = back[0][i];
        for (int i = 0; i < 3; i++) back[0][i] = left[0][i];
        for (int i = 0; i < 3; i++) left[0][i] = temp_side[i];

        for (int i = 0; i < 3; i++) temp_side[i] = front_speffz[0][i];
        for (int i = 0; i < 3; i++) front_speffz[0][i] = right_speffz[0][i];
        for (int i = 0; i < 3; i++) right_speffz[0][i] = back_speffz[0][i];
        for (int i = 0; i < 3; i++) back_speffz[0][i] = left_speffz[0][i];
        for (int i = 0; i < 3; i++) left_speffz[0][i] = temp_side[i];
}

void Cube::U2() {
        U();
        U();
}

void Cube::Ur() {
        U();
        U();
        U();
}
```

```
void Cube::D() {
        RotateCornerClockwise(down);
        RotateEdgeClockwise(down);
        RotateCornerClockwise(down_speffz);
        RotateEdgeClockwise(down_speffz);

        // 옆면 회전
        char temp_side[3] = { 0, 0, 0 };
        for (int i = 0; i < 3; i++) temp_side[i] = front[2][i];
        for (int i = 0; i < 3; i++) front[2][i] = left[2][i];
        for (int i = 0; i < 3; i++) left[2][i] = back[2][i];
        for (int i = 0; i < 3; i++) back[2][i] = right[2][i];
        for (int i = 0; i < 3; i++) right[2][i] = temp_side[i];

        for (int i = 0; i < 3; i++) temp_side[i] = front_speffz[2][i];
        for (int i = 0; i < 3; i++) front_speffz[2][i] = left_speffz[2][i];
        for (int i = 0; i < 3; i++) left_speffz[2][i] = back_speffz[2][i];
        for (int i = 0; i < 3; i++) back_speffz[2][i] = right_speffz[2][i];
        for (int i = 0; i < 3; i++) right_speffz[2][i] = temp_side[i];
}

void Cube::D2() {
        D();
        D();
}

void Cube::Dr() {
        D();
        D();
        D();
}
```

```
void Cube::Dw() {
        RotateCornerClockwise(down);
```

```
        RotateEdgeClockwise(down);
        RotateCornerClockwise(down_speffz);
        RotateEdgeClockwise(down_speffz);

        // 옆면 회전
        char temp_side[3] = { 0, 0, 0 };
        for (int i = 0; i < 3; i++) temp_side[i] = front[2][i];
        for (int i = 0; i < 3; i++) front[2][i] = left[2][i];
        for (int i = 0; i < 3; i++) left[2][i] = back[2][i];
        for (int i = 0; i < 3; i++) back[2][i] = right[2][i];
        for (int i = 0; i < 3; i++) right[2][i] = temp_side[i];
        for (int i = 0; i < 3; i++) temp_side[i] = front[1][i];
        for (int i = 0; i < 3; i++) front[1][i] = left[1][i];
        for (int i = 0; i < 3; i++) left[1][i] = back[1][i];
        for (int i = 0; i < 3; i++) back[1][i] = right[1][i];
        for (int i = 0; i < 3; i++) right[1][i] = temp_side[i];

        for (int i = 0; i < 3; i++) temp_side[i] = front_speffz[2][i];
        for (int i = 0; i < 3; i++) front_speffz[2][i] = left_speffz[2][i];
        for (int i = 0; i < 3; i++) left_speffz[2][i] = back_speffz[2][i];
        for (int i = 0; i < 3; i++) back_speffz[2][i] = right_speffz[2][i];
        for (int i = 0; i < 3; i++) right_speffz[2][i] = temp_side[i];
        for (int i = 0; i < 3; i++) temp_side[i] = front_speffz[1][i];
        for (int i = 0; i < 3; i++) front_speffz[1][i] = left_speffz[1][i];
        for (int i = 0; i < 3; i++) left_speffz[1][i] = back_speffz[1][i];
        for (int i = 0; i < 3; i++) back_speffz[1][i] = right_speffz[1][i];
        for (int i = 0; i < 3; i++) right_speffz[1][i] = temp_side[i];
}

void Cube::Dw2() {
        Dw();
        Dw();
}

void Cube::Dwr() {
        Dw();
        Dw();
        Dw();
}
```

```
void Cube::R() {
        RotateCornerClockwise(right);
        RotateEdgeClockwise(right);
        RotateCornerClockwise(right_speffz);
        RotateEdgeClockwise(right_speffz);

        // 옆면 회전
        char temp_side[3] = { 0, 0, 0 };
        for (int i = 0; i < 3; i++) temp_side[i] = front[i][2];
        for (int i = 0; i < 3; i++) front[i][2] = down[i][2];
        for (int i = 0; i < 3; i++) down[i][2] = back[2 - i][0];
        for (int i = 0; i < 3; i++) back[2 - i][0] = up[i][2];
        for (int i = 0; i < 3; i++) up[i][2] = temp_side[i];

        for (int i = 0; i < 3; i++) temp_side[i] = front_speffz[i][2];
        for (int i = 0; i < 3; i++) front_speffz[i][2] = down_speffz[i][2];
        for (int i = 0; i < 3; i++) down_speffz[i][2] = back_speffz[2 - i][0];
        for (int i = 0; i < 3; i++) back_speffz[2 - i][0] = up_speffz[i][2];
        for (int i = 0; i < 3; i++) up_speffz[i][2] = temp_side[i];
}

void Cube::R2() {
        R();
```

```
        R();
}

void Cube::Rr() {
        R();
        R();
        R();
}
```

```
void Cube::L() {
        RotateCornerClockwise(left);
        RotateEdgeClockwise(left);
        RotateCornerClockwise(left_speffz);
        RotateEdgeClockwise(left_speffz);

        // 옆면 회전
        char temp_side[3] = {0, 0, 0};
        for (int i = 0; i < 3; i++) temp_side[i] = front[i][0];
        for (int i = 0; i < 3; i++) front[i][0] = up[i][0];
        for (int i = 0; i < 3; i++) up[i][0] = back[2 - i][2];
        for (int i = 0; i < 3; i++) back[2 - i][2] = down[i][0];
        for (int i = 0; i < 3; i++) down[i][0] = temp_side[i];

        for (int i = 0; i < 3; i++) temp_side[i] = front_speffz[i][0];
        for (int i = 0; i < 3; i++) front_speffz[i][0] = up_speffz[i][0];
        for (int i = 0; i < 3; i++) up_speffz[i][0] = back_speffz[2 - i][2];
        for (int i = 0; i < 3; i++) back_speffz[2 - i][2] = down_speffz[i][0];
        for (int i = 0; i < 3; i++) down_speffz[i][0] = temp_side[i];
}

void Cube::L2() {
        L();
        L();
}

void Cube::Lr() {
        L();
        L();
        L();
}
```

```
void Cube::Lw() {
        RotateCornerClockwise(left);
        RotateEdgeClockwise(left);
        RotateCornerClockwise(left_speffz);
        RotateEdgeClockwise(left_speffz);

        // 옆면 회전
        char temp_side[3] = {0, 0, 0};
        for (int i = 0; i < 3; i++) temp_side[i] = front[i][0];
        for (int i = 0; i < 3; i++) front[i][0] = up[i][0];
        for (int i = 0; i < 3; i++) up[i][0] = back[2 - i][2];
        for (int i = 0; i < 3; i++) back[2 - i][2] = down[i][0];
        for (int i = 0; i < 3; i++) down[i][0] = temp_side[i];
        for (int i = 0; i < 3; i++) temp_side[i] = front[i][1];
        for (int i = 0; i < 3; i++) front[i][1] = up[i][1];
        for (int i = 0; i < 3; i++) up[i][1] = back[2 - i][1];
        for (int i = 0; i < 3; i++) back[2 - i][1] = down[i][1];
        for (int i = 0; i < 3; i++) down[i][1] = temp_side[i];

        for (int i = 0; i < 3; i++) temp_side[i] = front_speffz[i][0];
```

```
        for (int i = 0; i < 3; i++) front_speffz[i][0] = up_speffz[i][0];
        for (int i = 0; i < 3; i++) up_speffz[i][0] = back_speffz[2 - i][2];
        for (int i = 0; i < 3; i++) back_speffz[2 - i][2] = down_speffz[i][0];
        for (int i = 0; i < 3; i++) down_speffz[i][0] = temp_side[i];
        for (int i = 0; i < 3; i++) temp_side[i] = front_speffz[i][1];
        for (int i = 0; i < 3; i++) front_speffz[i][1] = up_speffz[i][1];
        for (int i = 0; i < 3; i++) up_speffz[i][1] = back_speffz[2 - i][1];
        for (int i = 0; i < 3; i++) back_speffz[2 - i][1] = down_speffz[i][1];
        for (int i = 0; i < 3; i++) down_speffz[i][1] = temp_side[i];
}

void Cube::Lw2() {
        Lw();
        Lw();
}

void Cube::Lwr() {
        Lw();
        Lw();
        Lw();
}
```

```
void Cube::F() {
        RotateCornerClockwise(front);
        RotateEdgeClockwise(front);
        RotateCornerClockwise(front_speffz);
        RotateEdgeClockwise(front_speffz);

        // 옆면 회전
        char temp_side[3] = { 0, 0, 0 };
        for (int i = 0; i < 3; i++) temp_side[i] = up[2][i];
        for (int i = 0; i < 3; i++) up[2][i] = left[2 - i][2];
        for (int i = 0; i < 3; i++) left[i][2] = down[0][i];
        for (int i = 0; i < 3; i++) down[0][i] = right[2 - i][0];
        for (int i = 0; i < 3; i++) right[i][0] = temp_side[i];

        for (int i = 0; i < 3; i++) temp_side[i] = up_speffz[2][i];
        for (int i = 0; i < 3; i++) up_speffz[2][i] = left_speffz[2 - i][2];
        for (int i = 0; i < 3; i++) left_speffz[i][2] = down_speffz[0][i];
        for (int i = 0; i < 3; i++) down_speffz[0][i] = right_speffz[2 - i][0];
        for (int i = 0; i < 3; i++) right_speffz[i][0] = temp_side[i];
}

void Cube::F2() {
        F();
        F();
}

void Cube::Fr() {
        F();
        F();
        F();
}
```

```
void Cube::B() {
        RotateCornerClockwise(back);
        RotateEdgeClockwise(back);
        RotateCornerClockwise(back_speffz);
        RotateEdgeClockwise(back_speffz);

        // 옆면 회전
```

```
        char temp_side[3] = { 0, 0, 0 };
        for (int i = 0; i < 3; i++) temp_side[i] = up[0][i];
        for (int i = 0; i < 3; i++) up[0][i] = right[i][2];
        for (int i = 0; i < 3; i++) right[i][2] = down[2][2 - i];
        for (int i = 0; i < 3; i++) down[2][i] = left[i][0];
        for (int i = 0; i < 3; i++) left[i][0] = temp_side[2 - i];

        for (int i = 0; i < 3; i++) temp_side[i] = up_speffz[0][i];
        for (int i = 0; i < 3; i++) up_speffz[0][i] = right_speffz[i][2];
        for (int i = 0; i < 3; i++) right_speffz[i][2] = down_speffz[2][2 - i];
        for (int i = 0; i < 3; i++) down_speffz[2][i] = left_speffz[i][0];
        for (int i = 0; i < 3; i++) left_speffz[i][0] = temp_side[2 - i];
}

void Cube::B2() {
        B();
        B();
}

void Cube::Br() {
        B();
        B();
        B();
}
```

```
void Cube::rotatingCube(bool slow, int* count) {
        Node* currentNode = scramble.gethead();
        if(count != NULL) *count = 0; // 회전 수 Count

        if (currentNode == NULL) { // Queue가 빈 상태에서 Solve를 하면 깨지는 현상 방지
                this->printcube();
                std::cout << std::endl;
        }
        else {
                while (currentNode != NULL) {
                        if (currentNode->getSymbol() == "U") this->U();
                        else if (currentNode->getSymbol() == "U2") this->U2();
                        else if (currentNode->getSymbol() == "U'") this->Ur();
                        else if (currentNode->getSymbol() == "R") this->R();
                        else if (currentNode->getSymbol() == "R2") this->R2();
                        else if (currentNode->getSymbol() == "R'") this->Rr();
                        else if (currentNode->getSymbol() == "D") this->D();
                        else if (currentNode->getSymbol() == "D2") this->D2();
                        else if (currentNode->getSymbol() == "D'") this->Dr();
                        else if (currentNode->getSymbol() == "Dw") this->Dw();
                        else if (currentNode->getSymbol() == "Dw2") this->Dw2();
                        else if (currentNode->getSymbol() == "Dw'") this->Dwr();
                        else if (currentNode->getSymbol() == "L") this->L();
                        else if (currentNode->getSymbol() == "L2") this->L2();
                        else if (currentNode->getSymbol() == "L'") this->Lr();
                        else if (currentNode->getSymbol() == "Lw") this->Lw();
                        else if (currentNode->getSymbol() == "Lw2") this->Lw2();
                        else if (currentNode->getSymbol() == "Lw'") this->Lwr();
                        else if (currentNode->getSymbol() == "F") this->F();
                        else if (currentNode->getSymbol() == "F2") this->F2();
                        else if (currentNode->getSymbol() == "F'") this->Fr();
                        else if (currentNode->getSymbol() == "B") this->B();
                        else if (currentNode->getSymbol() == "B2") this->B2();
                        else if (currentNode->getSymbol() == "B'") this->Br();
                        this->printcube();
                        std::cout << std::endl;
                        if (count != NULL) *count += 1;
                        scramble.popFront();
```

```
                        currentNode = scramble.gethead();

                        if (slow == true) { // 느리게 회전
                                Sleep(45);
                        }
                }
        }
}
```

[rotatingCube] 적용할 회전 기호를 가리킬 currentNode를 선언하여 gethead 함수로 scramble에 첫 노드 주소로 초기화 한다. Cube를 출력할 때, 커서를 처음으로 이동시켜 덮어 쓰는 방식으로 출력하기 때문에 아무것도 안하고 Cube를 맞추면 화면이 깨지는 것을 방지하기 위해 처음부터 currentNode가 NULL인 경우를 따로 분리한다. 그렇지 않으면 while문으로 inputtoqueue로 분리시킨 회전 기호 각각에 해당하는 메소드를 실행한 뒤, Cube를 출력하고, count의 주소가 NULL이 아니면 회전 수의 값(*count)을 하나 증가시킨다. 이후 scramble Queue에서 노드를 앞에서 하나 삭제하고, currentNode에 다시 첫 노드의 주소를 대입한다.

```
void Cube::inputtoQueue(std::string input) {
        scramble.inputtoqueue(input);
}
```

mian에서 사용자가 원하는 Scramble을 Cube에 적용할 수 있어야 하지만, scramble이 protected 영역에 있기 때문에 inputtoqueue를 사용자가 이용할 수 있도록 하는 메소드이다.

```
void scolor(unsigned short text = 15, unsigned short back = 0)
{
        SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), text | (back << 4));
}

void cur(short x, short y) {
        COORD pos = { x, y };
        SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), pos);
}
```

위에서 콘솔에 색을 입히거나, Cube 출력 시 강제로 왼쪽 위에 커서를 위치시키기 위한 함수이다.

Ⅳ. OP Algorithm 구현

Ⅳ-1. cube.h – cubeSolver Class(Cube 상속)

캡슐화로 실제 사용자가 Cube 클래스에 있는 Cube에 직접 접근하지 못하도록 Cube를 맞추는

cubeSolver 클래스를 Cube 클래스로 상속시켜 사용한다. 즉, 실제 main에서는 Cube 자체 구현도 Cube 클래스가 아닌 cubeSolver 클래스로 하게 될 것이다.

```
class cubeSolver : public Cube
```

```
private:
    std::string solution;
    double solvingtime;
    Queue cornersolving, edgesolving;

    //Corner Permutation Formula
    // Basic Formulas
    std::string YPerm = "R U' R' U' R U R' F' R U R' U' R' F R";
    ... <중략> ...

    //Edge Permutation Formula
    // Basic Formulas
    std::string TPerm = "R U R' U' R' F R2 U' R' U' R U R' F'";
    std::string JPerm = "R U R' F' R U R' U' R' F R2 U' R' U'";
    ... <중략> ...

    // Parity Remove Formula
    std::string RaPerm = "R U R' F' R U2 R' U2 R' F R U R U2 R' U'";
```

```
public:
    cubeSolver();
    void findCornerPiece(char, int*);
    void findEdgePiece(char, int*);
    bool cubeCornerSolver();
    void cubeEdgeSolver();
    void cubeSolving(bool);

    std::string getSolution();
    double getSolvingTime();
```

[cubeSolver 클래스]

private: 최종 Solution이나 맞추는 시간 혹은 조각 정렬 순서를 담을 Queue와 공식들은 사용자가 직접 접근하지 못하도록 private로 선언한다.

std::string solution : Cube를 맞추면, 사용자가 최종적인 Solution을 요약해서 볼 수 있도록 저장하는 문자열

double solvingtime : Cube를 맞추는 데 걸리는 시간을 저장하기 위한 변수

Queue cornersolving, edgesolving : 코너 조각 정렬 순서, 엣지 조각 정렬 순서를 저장할 Queue

std::string YPerm ~ RaPerm : 실제 Solving에 사용할 공식을 담은 문자열

public: 사용자가 Cube를 맞추거나 Solution 및 맞추는 시간을 제공받기 위해 접근할 수 있도록 public에 선언한 메소드이다.

cubeSolver() : cubeSolver 클래스의 생성자로 solution과 solvingtime을 초기화한다.

void findCornerPiece(char, int*) : Corner 조각 정렬 순서를 구할 때, 한 조각의 면이 원래 있어야

할 위치를 찾기 위한 메소드, 찾으려는 Speffz 기호를 인자로 받아, Cube에서 찾은 위치를 배열로 반환한다.

void findEdgePiece(char, int*) : 마찬가지로 Edge 조각 정렬 순서를 구할 때, 한 조각의 면이 원래 있어야할 위치를 찾기 위한 메소드이다.

bool cubeCornerSolver() : 모든 Corner 조각을 맞추는 메소드, 예외 상황인 Parity까지 처리한다.

bool cubeEdgeSolver() : 모든 Edge 조각을 맞추는 메소드이다.

bool cubeSolving() : 느리게 회전 여부를 인자로 받아, 위의 두 메소드를 이용해 Cube를 맞추고, 맞추는 시간을 측정하는 메소드이다.

std::string getSolution : 사용자에게 최종적으로 맞추는 모든 과정을 제공하기 위해 solution 변수를 반환하는 메소드

double getSolvingTime() : 사용자에게 프로그램이 Cube를 맞춘 시간을 제공하기 위해 solvingtime 변수를 반환하는 메소드이다.

Ⅳ-2. cube.cpp – cubeSolver Class 구현

```
cubeSolver::cubeSolver() {
        solution = "";
        solvingtime = 0;
}
```

```
void cubeSolver::findCornerPiece(char Piece, int* index) {
        cubeSolver compare;
        for (int i = 0; i < 3; i += 2) {
                for (int j = 0; j < 3; j += 2) {
                        if (compare.up_speffz[i][j] == Piece) {
                                index[0] = i;
                                index[1] = j;
                                index[2] = 0;
                        }
                        else if (compare.down_speffz[i][j] == Piece) {
                                index[0] = i;
                                index[1] = j;
                                index[2] = 1;
                        }
                        else if (compare.left_speffz[i][j] == Piece) {
                                index[0] = i;
                                index[1] = j;
                                index[2] = 2;
                        }
                        else if (compare.right_speffz[i][j] == Piece) {
                                index[0] = i;
                                index[1] = j;
                                index[2] = 3;
                        }
                        else if (compare.front_speffz[i][j] == Piece) {
                                index[0] = i;
                                index[1] = j;
                                index[2] = 4;
                        }
                        else if (compare.back_speffz[i][j] == Piece) {
```

```
                                index[0] = i;
                                index[1] = j;
                                index[2] = 5;
                        }
                }
        }
}
```

[findCornerPiece] 비교할 수 있는 대상이 되도록 초기화 되어 있는 Cube가 들어있는 cubeSolver 클래스 변수를 하나 새로 만들어서, 찾고자 하는 조각의 한 면의 Speffz 배치를 인자로 받아서 초기화된 Cube와 비교하여 그 위치를 반환하는 메소드이다. 위치를 index라는 3개의 요소를 가진 1차원 배열로 반환하는데, index[0]과 index[1]은 각 면의 3*3 배열에서 찾은 면의 행과 열을 저장하고, 마지막 index[2]는 6개의 면 중 어떤 면인지를 차례대로 0에서 5까지의 숫자로 저장한다.

```
void cubeSolver::findEdgePiece(char Piece, int* index) {
        cubeSolver compare;
        for (int i = 0; i < 2; i++) {
                if (compare.up_speffz[i][i + 1] == Piece) {
                        index[0] = i;
                        index[1] = i + 1;
                        index[2] = 0;
                }
                else if (compare.down_speffz[i][i + 1] == Piece) {
                        index[0] = i;
                        index[1] = i + 1;
                        index[2] = 1;
                }
                else if (compare.left_speffz[i][i + 1] == Piece) {
                        index[0] = i;
                        index[1] = i + 1;
                        index[2] = 2;
                }
                else if (compare.right_speffz[i][i + 1] == Piece) {
                        index[0] = i;
                        index[1] = i + 1;
                        index[2] = 3;
                }
                else if (compare.front_speffz[i][i + 1] == Piece) {
                        index[0] = i;
                        index[1] = i + 1;
                        index[2] = 4;
                }
                else if (compare.back_speffz[i][i + 1] == Piece) {
                        index[0] = i;
                        index[1] = i + 1;
                        index[2] = 5;
                }

                else if (compare.up_speffz[i + 1][i] == Piece) {
                        index[0] = i + 1;
                        index[1] = i;
                        index[2] = 0;
                }
                else if (compare.down_speffz[i + 1][i] == Piece) {
                        index[0] = i + 1;
                        index[1] = i;
                        index[2] = 1;
                }
                else if (compare.left_speffz[i + 1][i] == Piece) {
```

```
                        index[0] = i + 1;
                        index[1] = i;
                        index[2] = 2;
                }
                else if (compare.right_speffz[i + 1][i] == Piece) {
                        index[0] = i + 1;
                        index[1] = i;
                        index[2] = 3;
                }
                else if (compare.front_speffz[i + 1][i] == Piece) {
                        index[0] = i + 1;
                        index[1] = i;
                        index[2] = 4;
                }
                else if (compare.back_speffz[i + 1][i] == Piece) {
                        index[0] = i + 1;
                        index[1] = i;
                        index[2] = 5;
                }
        }
}
```

[findEdgePiece] Corner와 마찬가지로 찾고자하는 조각의 한 면의 Speffz 배치를 인자로 받은후, cubeSolver 클래스 변수를 만들어 초기화된 Cube와 비교하고, 그 위치를 index 배열로 반환하는 메소드이다.

```
bool cubeSolver::cubeCornerSolver() {
        std::string corner = ""; // Corner를 맞추는 순서가 저장될 string 변수
        cubeSolver compare; // 이미 맞춰진 조각인지 확인을 위한 임시 Cube
        bool cornersolved = false, paritycheck = false; // corner가 전부 맞춰졌는지 확인,
parity가 있는지 확인
        char temp = this->up_speffz[0][0]; // 조각이 원래 있어야할 위치 저장
        char temp1 = 'A', temp2 = 'E', temp3 = 'R'; // first buffer
        // buffer로 사용할 수 있는 조각 list(맞춰지지 않은 조각)
        int bufferlist[24] = { 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
1 ,1, 1, 1, 1, 1 };
        int index[3] = { -1, -1, -1 }; // 조각을 찾을 때 위치를 저장할 배열

        // 이미 맞춰진 조각은 bufferlist에서 해제
        for (int i = 0; i < 3; i += 2) {
                for (int j = 0; j < 3; j += 2) {
                        if (compare.up_speffz[i][j] == this->up_speffz[i][j])
                                bufferlist[(int)(this->up_speffz[i][j] - 65)] = 0;
                        if (compare.down_speffz[i][j] == this->down_speffz[i][j])
                                bufferlist[(int)(this->down_speffz[i][j] - 65)] = 0;
                        if (compare.left_speffz[i][j] == this->left_speffz[i][j])
                                bufferlist[(int)(this->left_speffz[i][j] - 65)] = 0;
                        if (compare.right_speffz[i][j] == this->right_speffz[i][j])
                                bufferlist[(int)(this->right_speffz[i][j] - 65)] = 0;
                        if (compare.front_speffz[i][j] == this->front_speffz[i][j])
                                bufferlist[(int)(this->front_speffz[i][j] - 65)] = 0;
                        if (compare.back_speffz[i][j] == this->back_speffz[i][j])
                                bufferlist[(int)(this->back_speffz[i][j] - 65)] = 0;
                }
        }

        // 첫 Buffer에 있는 조각이 원래 있어야할 위치 추가
        if (temp != 'A' && temp != 'E' && temp != 'R') {
                corner += temp;
                corner += " ";
```

```
        }

        while (cornersolved == false) { // Corner가 모두 맞춰질 때까지 반복
                char temptemp = temp; // Buffer의 위치에 있는 조각을 잠깐 담아두기 위한 변수
                while (1) {
                        temptemp = temp;
                        // Buffer 조각이 되면, Buffer에 있는 조각을 한번 더 넣고 종료
                        // 바뀐 버퍼를 다시 제일 처음 버퍼로 돌려놓기 위함
                        if (temp == temp1 || temp == temp2 || temp == temp3) {
                                //Buffer에서 시작하여 각 조각의 면이 원래 있어야할 조각
위치로 이동
                                this->findCornerPiece(temp, index);
                                if (index[2] == 0)
                                        temp = this->up_speffz[index[0]][index[1]];
                                else if (index[2] == 1)
                                        temp = this->down_speffz[index[0]][index[1]];
                                else if (index[2] == 2)
                                        temp = this->left_speffz[index[0]][index[1]];
                                else if (index[2] == 3)
                                        temp = this->right_speffz[index[0]][index[1]];
                                else if (index[2] == 4)
                                        temp = this->front_speffz[index[0]][index[1]];
                                else if (index[2] == 5)
                                        temp = this->back_speffz[index[0]][index[1]];
                                else
                                        std::cout << "Error" << std::endl;

                                // Buffer가 제일 처음 Buffer일 경우 한 번 더 넣을 필요 X
                                if (temptemp != 'A' && temptemp != 'E' && temptemp !=
'R') {
                                        corner += temp;
                                        corner += " ";
                                }
                                break;
                        }

                        // 버퍼로 선택할 조각을 고르기 위해 위에서 이미 사용된 조각을
bufferlist 변수에서 off(= 0) 함
                        if (temp == 'B' || temp == 'N' || temp == 'Q')
                                bufferlist[1] = bufferlist[13] = bufferlist[16] = 0;
                        else if (temp == 'C' || temp == 'J' || temp == 'M')
                                bufferlist[2] = bufferlist[9] = bufferlist[12] = 0;
                        else if (temp == 'D' || temp == 'I' || temp == 'F')
                                bufferlist[3] = bufferlist[8] = bufferlist[5] = 0;
                        else if (temp == 'U' || temp == 'L' || temp == 'G')
                                bufferlist[20] = bufferlist[11] = bufferlist[6] = 0;
                        else if (temp == 'V' || temp == 'K' || temp == 'P')
                                bufferlist[21] = bufferlist[10] = bufferlist[15] = 0;
                        else if (temp == 'W' || temp == 'O' || temp == 'T')
                                bufferlist[22] = bufferlist[14] = bufferlist[19] = 0;
                        else if (temp == 'X' || temp == 'S' || temp == 'H')
                                bufferlist[23] = bufferlist[18] = bufferlist[7] = 0;

                        //Buffer에서 시작하여 각 조각의 면이 원래 있어야할 조각 위치로 이동
                        this->findCornerPiece(temp, index);
                        if (index[2] == 0)
                                temp = this->up_speffz[index[0]][index[1]];
                        else if (index[2] == 1)
                                temp = this->down_speffz[index[0]][index[1]];
                        else if (index[2] == 2)
                                temp = this->left_speffz[index[0]][index[1]];
                        else if (index[2] == 3)
                                temp = this->right_speffz[index[0]][index[1]];
```

```
                        else if (index[2] == 4)
                                temp = this->front_speffz[index[0]][index[1]];
                        else if (index[2] == 5)
                                temp = this->back_speffz[index[0]][index[1]];
                        else std::cout << "Error" << std::endl;

                        // Corner를 맞추는 순서에 추가
                        if (temp != 'A' && temp != 'E' && temp != 'R') { // 첫 Cycle의
버퍼는 뒤에 붙일 필요 없으므로.
                                corner += temp;
                                corner += " ";
                        }
                }

                cornersolved = true; //

                // Corner 조각이 모두 맞춰지지 않았으면, buffer 재설정
                for (int i = 0; i < 24; i++) {
                        if (bufferlist[i] == 1) {
                                temp1 = (char)(65 + i);
                                cornersolved = false;
                                break;
                        }
                }

                if (cornersolved == false) {
                        // Buffer 재설정
                        this->findCornerPiece(temp1, index);
                        if (index[2] == 0)
                                temp = this->up_speffz[index[0]][index[1]];
                        else if (index[2] == 1)
                                temp = this->down_speffz[index[0]][index[1]];
                        else if (index[2] == 2)
                                temp = this->left_speffz[index[0]][index[1]];
                        else if (index[2] == 3)
                                temp = this->right_speffz[index[0]][index[1]];
                        else if (index[2] == 4)
                                temp = this->front_speffz[index[0]][index[1]];
                        else if (index[2] == 5)
                                temp = this->back_speffz[index[0]][index[1]];
                        else
                                std::cout << "Error" << std::endl;
                        corner += temp;
                        corner += " ";

                        // 선택한 Buffer를 bufferlist에서 삭제
                        if (temp1 == 'B' || temp1 == 'N' || temp1 == 'Q')
                                bufferlist[1] = bufferlist[13] = bufferlist[16] = 0;
                        else if (temp1 == 'C' || temp1 == 'J' || temp1 == 'M')
                                bufferlist[2] = bufferlist[9] = bufferlist[12] = 0;
                        else if (temp1 == 'D' || temp1 == 'I' || temp1 == 'F')
                                bufferlist[3] = bufferlist[8] = bufferlist[5] = 0;
                        else if (temp1 == 'U' || temp1 == 'L' || temp1 == 'G')
                                bufferlist[20] = bufferlist[11] = bufferlist[6] = 0;
                        else if (temp1 == 'V' || temp1 == 'K' || temp1 == 'P')
                                bufferlist[21] = bufferlist[10] = bufferlist[15] = 0;
                        else if (temp1 == 'W' || temp1 == 'O' || temp1 == 'T')
                                bufferlist[22] = bufferlist[14] = bufferlist[19] = 0;
                        else if (temp1 == 'X' || temp1 == 'S' || temp1 == 'H')
                                bufferlist[23] = bufferlist[18] = bufferlist[7] = 0;

                        // Buffer 조각으로 다시 돌아왔을 때의 종료 조건을 위한 변수 설정
                        if (temp1 == 'B' || temp1 == 'N' || temp1 == 'Q') {
                                temp1 = 'B'; temp2 = 'N'; temp3 = 'Q';
```

```
                        }
                        else if (temp1 == 'C' || temp1 == 'J' || temp1 == 'M') {
                                temp1 = 'C'; temp2 = 'J'; temp3 = 'M';
                        }
                        else if (temp1 == 'D' || temp1 == 'I' || temp1 == 'F') {
                                temp1 = 'D'; temp2 = 'I'; temp3 = 'F';
                        }
                        else if (temp1 == 'U' || temp1 == 'L' || temp1 == 'G') {
                                temp1 = 'U'; temp2 = 'L'; temp3 = 'G';
                        }
                        else if (temp1 == 'V' || temp1 == 'K' || temp1 == 'P') {
                                temp1 = 'V'; temp2 = 'K'; temp3 = 'P';
                        }
                        else if (temp1 == 'W' || temp1 == 'O' || temp1 == 'T') {
                                temp1 = 'W'; temp2 = 'O'; temp3 = 'T';
                        }
                        else if (temp1 == 'X' || temp1 == 'S' || temp1 == 'H') {
                                temp1 = 'X'; temp2 = 'S'; temp3 = 'H';
                        }
                }
        }

        if (corner == "") solution += "[Corners Already Solved]\n";
        else solution += "Corner Solving[Modified Y-Perm]: " + corner + "\n";

        // Queue에 삽입
        cornersolving.inputtoqueue(corner);

        // Queue에서 하나씩 꺼내가며 공식 실행
        Node* currentNode = cornersolving.gethead();
        while (currentNode != NULL) {
                if (currentNode->getSymbol() == "A") {
                        scramble.inputtoqueue(c_A);
                        solution += "→ A: " + c_A + "\n";
                }
                else if (currentNode->getSymbol() == "B") {
                        scramble.inputtoqueue(c_B);
                        solution += "→ B: " + c_B + "\n";
                }
                else if (currentNode->getSymbol() == "C") {
                        scramble.inputtoqueue(c_C);
                        solution += "→ C: " + c_C + "\n";
                }
                else if (currentNode->getSymbol() == "D") {
                        scramble.inputtoqueue(c_D);
                        solution += "→ D: " + c_D + "\n";
                }
                else if (currentNode->getSymbol() == "E") {
                        scramble.inputtoqueue(c_E);
                        solution += "→ E: " + c_E + "\n";
                }
                else if (currentNode->getSymbol() == "F") {
                        scramble.inputtoqueue(c_F);
                        solution += "→ F: " + c_F + "\n";
                }
                else if (currentNode->getSymbol() == "G") {
                        scramble.inputtoqueue(c_G);
                        solution += "→ G: " + c_G + "\n";
                }
                else if (currentNode->getSymbol() == "H") {
                        scramble.inputtoqueue(c_H);
                        solution += "→ H: " + c_H + "\n";
                }
                else if (currentNode->getSymbol() == "I") {
```

```
                scramble.inputtoqueue(c_I);
                solution += "→ I: " + c_I + "\n";
        }
        else if (currentNode->getSymbol() == "J") {
                scramble.inputtoqueue(c_J);
                solution += "→ J: " + c_J + "\n";
        }
        else if (currentNode->getSymbol() == "K") {
                scramble.inputtoqueue(c_K);
                solution += "→ K: " + c_K + "\n";
        }
        else if (currentNode->getSymbol() == "L") {
                scramble.inputtoqueue(c_L);
                solution += "→ L: " + c_L + "\n";
        }
        else if (currentNode->getSymbol() == "M") {
                scramble.inputtoqueue(c_M);
                solution += "→ M: " + c_M + "\n";
        }
        else if (currentNode->getSymbol() == "N") {
                scramble.inputtoqueue(c_N);
                solution += "→ N: " + c_N + "\n";
        }
        else if (currentNode->getSymbol() == "O") {
                scramble.inputtoqueue(c_O);
                solution += "→ O: " + c_O + "\n";
        }
        else if (currentNode->getSymbol() == "P") {
                scramble.inputtoqueue(c_P);
                solution += "→ P: " + c_P + "\n";
        }
        else if (currentNode->getSymbol() == "Q") {
                scramble.inputtoqueue(c_Q);
                solution += "→ Q: " + c_Q + "\n";
        }
        else if (currentNode->getSymbol() == "R") {
                scramble.inputtoqueue(c_R);
                solution += "→ R: " + c_R + "\n";
        }
        else if (currentNode->getSymbol() == "S") {
                scramble.inputtoqueue(c_S);
                solution += "→ S: " + c_S + "\n";
        }
        else if (currentNode->getSymbol() == "T") {
                scramble.inputtoqueue(c_T);
                solution += "→ T: " + c_T + "\n";
        }
        else if (currentNode->getSymbol() == "U") {
                scramble.inputtoqueue(c_U);
                solution += "→ U: " + c_U + "\n";
        }
        else if (currentNode->getSymbol() == "V") {
                scramble.inputtoqueue(c_V);
                solution += "→ V: " + c_V + "\n";
        }
        else if (currentNode->getSymbol() == "W") {
                scramble.inputtoqueue(c_W);
                solution += "→ W: " + c_W + "\n";
        }
        else if (currentNode->getSymbol() == "X") {
                scramble.inputtoqueue(c_X);
                solution += "→ X: " + c_X + "\n";
        }
        cornersolving.popFront();
```

```
                currentNode = cornersolving.gethead();
        }

        // Parity 제거
        if ((corner.length() / 2) % 2 == 1) {
                scramble.inputtoqueue(RaPerm);
                paritycheck = true;
        }

        if (paritycheck == true) solution += "\n[Parity Exists] Ra-Perm(R U R' F' R U2
R' U2 R' F R U R U2 R' U') Used\n";
        return paritycheck;
}
```

```
void cubeSolver::cubeEdgeSolver() { // Corner 조각 맞출 때 사용한 공식의 개수를 불러옴
        std::string edge = ""; // Edge를 맞추는 순서가 저장될 string 변수
        cubeSolver compare; // 이미 맞춰진 조각인지 확인을 위한 임시 Cube
        bool edgesolved = false; // edge가 전부 맞춰졌는지 확인
        char temp = this->up_speffz[1][2]; // 조각이 원래 있어야할 위치 저장
        char temp1 = 'B', temp2 = 'M'; // first buffer
        // buffer로 사용할 수 있는 조각 list(맞춰지지 않은 조각)
        int bufferlist[24] = { 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1,
1 ,1, 1, 1, 1, 1 };
        int index[3] = { -1, -1, -1 }; // 조각을 찾을 때 위치를 저장할 배열

        // 이미 맞춰진 조각은 bufferlist에서 해제
        for (int i = 0; i < 2; i++) {
                if (compare.up_speffz[i][i + 1] == this->up_speffz[i][i + 1])
                        bufferlist[(int)(this->up_speffz[i][i + 1] - 65)] = 0;
                if (compare.down_speffz[i][i + 1] == this->down_speffz[i][i + 1])
                        bufferlist[(int)(this->down_speffz[i][i + 1] - 65)] = 0;
                if (compare.left_speffz[i][i + 1] == this->left_speffz[i][i + 1])
                        bufferlist[(int)(this->left_speffz[i][i + 1] - 65)] = 0;
                if (compare.right_speffz[i][i + 1] == this->right_speffz[i][i + 1])
                        bufferlist[(int)(this->right_speffz[i][i + 1] - 65)] = 0;
                if (compare.front_speffz[i][i + 1] == this->front_speffz[i][i + 1])
                        bufferlist[(int)(this->front_speffz[i][i + 1] - 65)] = 0;
                if (compare.back_speffz[i][i + 1] == this->back_speffz[i][i + 1])
                        bufferlist[(int)(this->back_speffz[i][i + 1] - 65)] = 0;
                if (compare.up_speffz[i + 1][i] == this->up_speffz[i + 1][i])
                        bufferlist[(int)(this->up_speffz[i + 1][i] - 65)] = 0;
                if (compare.down_speffz[i + 1][i] == this->down_speffz[i + 1][i])
                        bufferlist[(int)(this->down_speffz[i + 1][i] - 65)] = 0;
                if (compare.left_speffz[i + 1][i] == this->left_speffz[i + 1][i])
                        bufferlist[(int)(this->left_speffz[i + 1][i] - 65)] = 0;
                if (compare.right_speffz[i + 1][i] == this->right_speffz[i + 1][i])
                        bufferlist[(int)(this->right_speffz[i + 1][i] - 65)] = 0;
                if (compare.front_speffz[i + 1][i] == this->front_speffz[i + 1][i])
                        bufferlist[(int)(this->front_speffz[i + 1][i] - 65)] = 0;
                if (compare.back_speffz[i + 1][i] == this->back_speffz[i + 1][i])
                        bufferlist[(int)(this->back_speffz[i + 1][i] - 65)] = 0;
        }

        // 첫 Buffer에 있는 조각이 원래 있어야할 위치 추가
        if (temp != 'B' && temp != 'M') {
                edge += temp;
                edge += " ";
        }

        while (edgesolved == false) { // Edge가 모두 맞춰질 때까지 반복
                char temptemp = temp; // Buffer의 위치에 있는 조각을 잠깐 담아두기 위한 변수
```

```
                while (1) {
                        temptemp = temp;
                        // Buffer 조각이 되면, Buffer에 담긴 조각을 한번 더 넣고 종료
                        // 바뀐 버퍼를 다시 제일 처음 버퍼로 돌려놓기 위함
                        if (temp == temp1 || temp == temp2) {
                                //Buffer에서 시작하여 각 조각의 면이 원래 있어야할 조각
위치로 이동
                                this->findEdgePiece(temp, index);
                                if (index[2] == 0)
                                        temp = this->up_speffz[index[0]][index[1]];
                                else if (index[2] == 1)
                                        temp = this->down_speffz[index[0]][index[1]];
                                else if (index[2] == 2)
                                        temp = this->left_speffz[index[0]][index[1]];
                                else if (index[2] == 3)
                                        temp = this->right_speffz[index[0]][index[1]];
                                else if (index[2] == 4)
                                        temp = this->front_speffz[index[0]][index[1]];
                                else if (index[2] == 5)
                                        temp = this->back_speffz[index[0]][index[1]];
                                else
                                        std::cout << "Error" << std::endl;
                                if (temptemp != 'B' && temptemp != 'M') { // Buffer가
제일 처음 Buffer일 경우 한 번 더 넣을 필요 X
                                        edge += temp;
                                        edge += " ";
                                }
                                break;
                        }

                        // 버퍼로 선택할 조각을 고르기 위해 위에서 이미 사용된 조각을
bufferlist 변수에서 off(= 0) 함
                        if (temp == 'A' || temp == 'Q')
                                bufferlist[0] = bufferlist[16] = 0;
                        else if (temp == 'C' || temp == 'I')
                                bufferlist[2] = bufferlist[8] = 0;
                        else if (temp == 'D' || temp == 'E')
                                bufferlist[3] = bufferlist[4] = 0;
                        else if (temp == 'H' || temp == 'R')
                                bufferlist[7] = bufferlist[17] = 0;
                        else if (temp == 'F' || temp == 'L')
                                bufferlist[5] = bufferlist[11] = 0;
                        else if (temp == 'J' || temp == 'P')
                                bufferlist[9] = bufferlist[15] = 0;
                        else if (temp == 'N' || temp == 'T')
                                bufferlist[13] = bufferlist[19] = 0;
                        else if (temp == 'U' || temp == 'K')
                                bufferlist[20] = bufferlist[10] = 0;
                        else if (temp == 'V' || temp == 'O')
                                bufferlist[21] = bufferlist[14] = 0;
                        else if (temp == 'W' || temp == 'S')
                                bufferlist[22] = bufferlist[18] = 0;
                        else if (temp == 'X' || temp == 'G')
                                bufferlist[23] = bufferlist[6] = 0;

                        //Buffer에서 시작하여 각 조각의 면이 원래 있어야할 조각 위치로 이동
                        this->findEdgePiece(temp, index);
                        if (index[2] == 0)
                                temp = this->up_speffz[index[0]][index[1]];
                        else if (index[2] == 1)
                                temp = this->down_speffz[index[0]][index[1]];
                        else if (index[2] == 2)
                                temp = this->left_speffz[index[0]][index[1]];
```

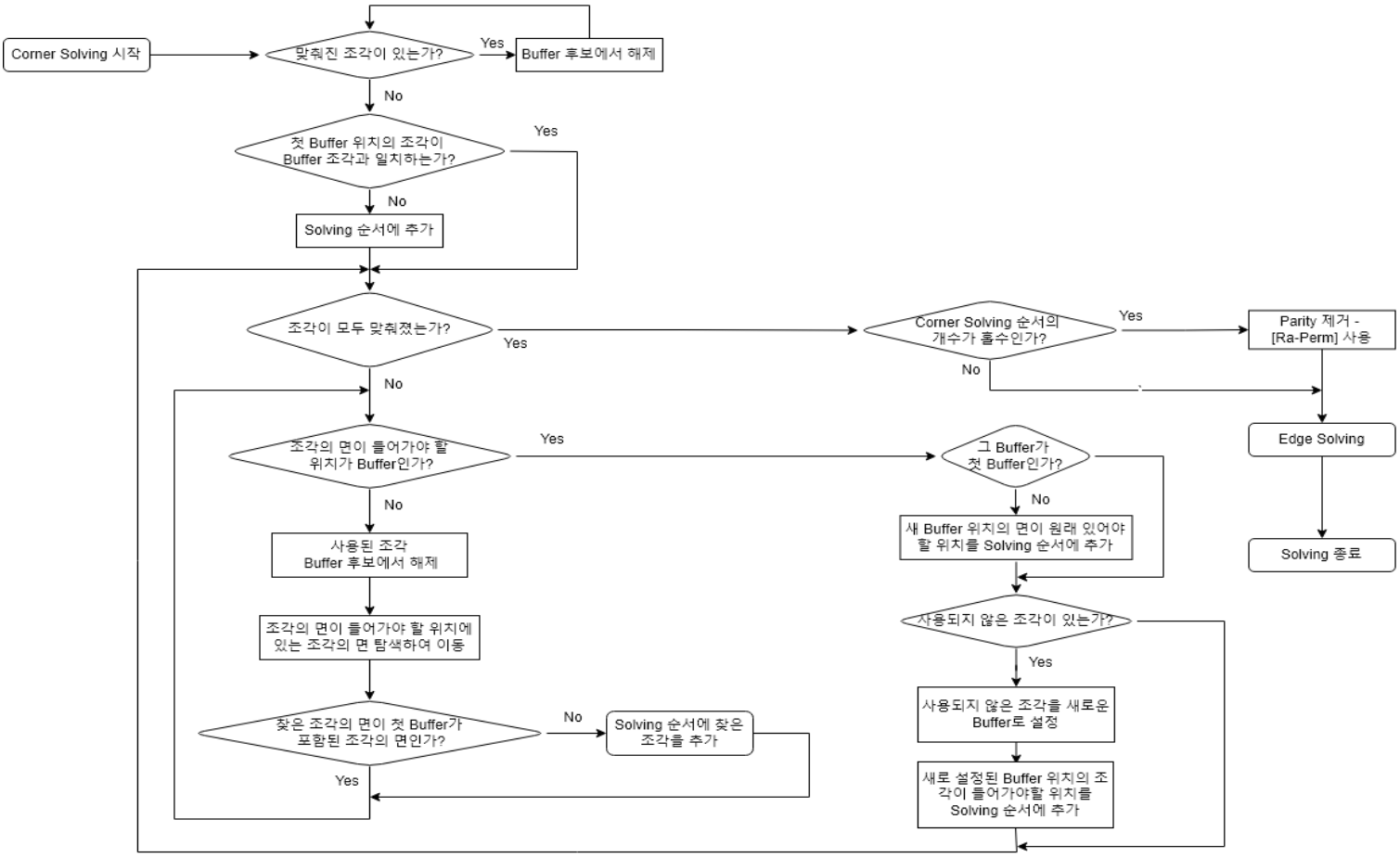
```
                        else if (index[2] == 3)
                                temp = this->right_speffz[index[0]][index[1]];
                        else if (index[2] == 4)
                                temp = this->front_speffz[index[0]][index[1]];
                        else if (index[2] == 5)
                                temp = this->back_speffz[index[0]][index[1]];
                        else std::cout << "Error" << std::endl;

                        // Edge를 맞추는 순서에 추가
                        if (temp != 'B' && temp != 'M') { // 첫 Cycle의 버퍼는 뒤에 붙일
필요 없으므로.
                                edge += temp;
                                edge += " ";
                        }
                }

                edgesolved = true;

                // Edge 조각이 모두 맞춰지지 않았으면, buffer 재설정
                for (int i = 0; i < 24; i++) {
                        if (bufferlist[i] == 1) {
                                temp1 = (char)(65 + i);
                                edgesolved = false;
                                break;
                        }
                }

                if (edgesolved == false) {
                        // Buffer 재설정
                        this->findEdgePiece(temp1, index);
                        if (index[2] == 0)
                                temp = this->up_speffz[index[0]][index[1]];
                        else if (index[2] == 1)
                                temp = this->down_speffz[index[0]][index[1]];
                        else if (index[2] == 2)
                                temp = this->left_speffz[index[0]][index[1]];
                        else if (index[2] == 3)
                                temp = this->right_speffz[index[0]][index[1]];
                        else if (index[2] == 4)
                                temp = this->front_speffz[index[0]][index[1]];
                        else if (index[2] == 5)
                                temp = this->back_speffz[index[0]][index[1]];
                        else
                                std::cout << "Error" << std::endl;
                        edge += temp;
                        edge += " ";

                        // 선택한 Buffer를 bufferlist에서 삭제
                        if (temp1 == 'A' || temp1 == 'Q')
                                bufferlist[0] = bufferlist[16] = 0;
                        else if (temp1 == 'C' || temp1 == 'I')
                                bufferlist[2] = bufferlist[8] = 0;
                        else if (temp1 == 'D' || temp1 == 'E')
                                bufferlist[3] = bufferlist[4] = 0;
                        else if (temp1 == 'H' || temp1 == 'R')
                                bufferlist[7] = bufferlist[17] = 0;
                        else if (temp1 == 'F' || temp1 == 'L')
                                bufferlist[5] = bufferlist[11] = 0;
                        else if (temp1 == 'J' || temp1 == 'P')
                                bufferlist[9] = bufferlist[15] = 0;
                        else if (temp1 == 'N' || temp1 == 'T')
                                bufferlist[13] = bufferlist[19] = 0;
                        else if (temp1 == 'U' || temp1 == 'K')
                                bufferlist[20] = bufferlist[10] = 0;
```

```
                        else if (temp1 == 'V' || temp1 == 'O')
                                bufferlist[21] = bufferlist[14] = 0;
                        else if (temp1 == 'W' || temp1 == 'S')
                                bufferlist[22] = bufferlist[18] = 0;
                        else if (temp1 == 'X' || temp1 == 'G')
                                bufferlist[23] = bufferlist[6] = 0;

                        // Buffer 조각으로 다시 돌아왔을 때의 종료 조건을 위한 변수 설정
                        if (temp1 == 'A' || temp1 == 'Q') {
                                temp1 = 'A'; temp2 = 'Q';
                        }
                        else if (temp1 == 'C' || temp1 == 'I') {
                                temp1 = 'C'; temp2 = 'I';
                        }
                        else if (temp1 == 'D' || temp1 == 'E') {
                                temp1 = 'D'; temp2 = 'E';
                        }
                        else if (temp1 == 'H' || temp1 == 'R') {
                                temp1 = 'H'; temp2 = 'R';
                        }
                        else if (temp1 == 'F' || temp1 == 'L') {
                                temp1 = 'F'; temp2 = 'L';
                        }
                        else if (temp1 == 'J' || temp1 == 'P') {
                                temp1 = 'J'; temp2 = 'P';
                        }
                        else if (temp1 == 'N' || temp1 == 'T') {
                                temp1 = 'N'; temp2 = 'T';
                        }
                        else if (temp1 == 'U' || temp1 == 'K') {
                                temp1 = 'U'; temp2 = 'K';
                        }
                        else if (temp1 == 'V' || temp1 == 'O') {
                                temp1 = 'V'; temp2 = 'O';
                        }
                        else if (temp1 == 'W' || temp1 == 'S') {
                                temp1 = 'W'; temp2 = 'S';
                        }
                        else if (temp1 == 'X' || temp1 == 'G') {
                                temp1 = 'X'; temp2 = 'G';
                        }
                }
        }

        if (edge == "") solution += "\n[Edges Already Solved]\n";
        else solution += "\nEdge Solving[T-Perm & J-Perm]: " + edge + "\n";

        // Queue에 삽입
        edgesolving.inputtoqueue(edge);

        // Queue에서 하나씩 꺼내가며 공식 실행 후 Solution 출력용 문자열에 추가
        Node* currentNode = edgesolving.gethead();
        while (currentNode != NULL) {
                if (currentNode->getSymbol() == "A") {
                        scramble.inputtoqueue(e_A);
                        solution += "→ A: " + e_A + "\n";
                }
                else if (currentNode->getSymbol() == "B") {
                        scramble.inputtoqueue(e_B);
                        solution += "→ B: " + e_B + "\n";
                }
                else if (currentNode->getSymbol() == "C") {
                        scramble.inputtoqueue(e_C);
                        solution += "→ C: " + e_C + "\n";
```

```
                }
                else if (currentNode->getSymbol() == "D") {
                        scramble.inputtoqueue(e_D);
                        solution += "→ D: " + e_D + "\n";
                }
                else if (currentNode->getSymbol() == "E") {
                        scramble.inputtoqueue(e_E);
                        solution += "→ E: " + e_E + "\n";
                }
                else if (currentNode->getSymbol() == "F") {
                        scramble.inputtoqueue(e_F);
                        solution += "→ F: " + e_F + "\n";
                }
                else if (currentNode->getSymbol() == "G") {
                        scramble.inputtoqueue(e_G);
                        solution += "→ G: " + e_G + "\n";
                }
                else if (currentNode->getSymbol() == "H") {
                        scramble.inputtoqueue(e_H);
                        solution += "→ H: " + e_H + "\n";
                }
                else if (currentNode->getSymbol() == "I") {
                        scramble.inputtoqueue(e_I);
                        solution += "→ I: " + e_I + "\n";
                }
                else if (currentNode->getSymbol() == "J") {
                        scramble.inputtoqueue(e_J);
                        solution += "→ J: " + e_J + "\n";
                }
                else if (currentNode->getSymbol() == "K") {
                        scramble.inputtoqueue(e_K);
                        solution += "→ K: " + e_K + "\n";
                }
                else if (currentNode->getSymbol() == "L") {
                        scramble.inputtoqueue(e_L);
                        solution += "→ L: " + e_L + "\n";
                }
                else if (currentNode->getSymbol() == "M") {
                        scramble.inputtoqueue(e_M);
                        solution += "→ M: " + e_M + "\n";
                }
                else if (currentNode->getSymbol() == "N") {
                        scramble.inputtoqueue(e_N);
                        solution += "→ N: " + e_N + "\n";
                }
                else if (currentNode->getSymbol() == "O") {
                        scramble.inputtoqueue(e_O);
                        solution += "→ O: " + e_O + "\n";
                }
                else if (currentNode->getSymbol() == "P") {
                        scramble.inputtoqueue(e_P);
                        solution += "→ P: " + e_P + "\n";
                }
                else if (currentNode->getSymbol() == "Q") {
                        scramble.inputtoqueue(e_Q);
                        solution += "→ Q: " + e_Q + "\n";
                }
                else if (currentNode->getSymbol() == "R") {
                        scramble.inputtoqueue(e_R);
                        solution += "→ R: " + e_R + "\n";
                }
                else if (currentNode->getSymbol() == "S") {
                        scramble.inputtoqueue(e_S);
                        solution += "→ S: " + e_S + "\n";
```

```
                }
                else if (currentNode->getSymbol() == "T") {
                        scramble.inputtoqueue(e_T);
                        solution += "→ T: " + e_T + "\n";
                }
                else if (currentNode->getSymbol() == "U") {
                        scramble.inputtoqueue(e_U);
                        solution += "→ U: " + e_U + "\n";
                }
                else if (currentNode->getSymbol() == "V") {
                        scramble.inputtoqueue(e_V);
                        solution += "→ V: " + e_V + "\n";
                }
                else if (currentNode->getSymbol() == "W") {
                        scramble.inputtoqueue(e_W);
                        solution += "→ W: " + e_W + "\n";
                }
                else if (currentNode->getSymbol() == "X") {
                        scramble.inputtoqueue(e_X);
                        solution += "→ X: " + e_X + "\n";
                }
                edgesolving.popFront();
                currentNode = edgesolving.gethead();
        }
}
```

[cubeCornerSolver], [cubeEdgeSolver]: 각각 Corner 조각과 Edge 조각의 정렬 순서를 구하는 메소드이다. 아래 첨부한 순서도에 흐름대로 조각의 정렬 순서를 구하고, 이를 cornersolving Queue와 edgesolving Queue에 넣어 저장한다. 그 후 앞에서부터 하나씩 꺼내 가면서 순서에 맞는 공식을 불러와 scramble Queue에 회전 기호를 삽입한다.




```
void cubeSolver::cubeSolving(bool slow) {
        clock_t start, end; // Solving 시간 측정을 위한 변수

        start = clock(); // 시간 측정 시작
        cubeCornerSolver(); // Corner 조각 Solving + Parity Check
        cubeEdgeSolver(); // edge 조각 Solving
        rotatingCube(slow, &count); // 풀어낸 대로 Cube 회전
        end = clock(); // 시간 측정 종료

        solvingtime = (double)(end - start) / CLOCKS_PER_SEC;
}
```

[cubeSolving] 위에서 구현한 cubeCornerSolver, cubeEdgeSolver를 이용하여 사용자가 Cube를 맞출 수 있도록 하는 메소드이다. time.h의 clock_t과 clock()을 이용하여 맞추는 시간을 계산하고, cubeCornerSolver와 cubeEdgeSolver에서 Cube를 맞추기 위한 모든 회전 기호를 scramble에 담았으므로 이를 rotatingCube를 이용하여 Cube를 회전시켜 최종적으로 맞춘다. 이때, 인자로 느리게 회전시킬 것인지 여부를 인자로 받아 넣고, 회전 수를 세기 위한 count 변수를 call by reference를 통해 인자로 넣는다. 시간 측정이 종료되면, solvingtime 변수에 측정한 시각의 차이를 구해 second 단위로 저장한다.

```
std::string cubeSolver::getSolution() {
        // solution 문자열 Return 후, solution 문자열 비우기
        std::string temp = solution;
        solution = "";
        return temp;
}

double cubeSolver::getSolvingTime() {
        return solvingtime;
}

int cubeSolver::getCount() {
        return count;
}
```

[getSolution], [getSolvingTime], [getCount] 사용자가 최종 풀이 과정, 맞추는 시간, 회전 수를 알기 위해 이를 반환하는 메소드이다. 특히 getSolution에서는 한 번 반환하고 solution을 다시 빈 문자열로 만들어 다음에 Cube를 맞췄을 때, 이전 solution이 남아있지 않도록 ""로 초기화한다.

V. Main 함수

```
int main() {
	cubeSolver rubikscube; // Rubik's Cube 생성
	std::string input = ""; // 입력한 값을 저장할 String

	while (input != "4") { // Quit이 입력되기 전까지 반복
		input = ""; // input 초기화

		// Cube 출력
		rubikscube.printcube();
		std::cout << "1: Reset Cube | 2: Solve Cube | 3: Solve Cube Slowly | 4:
			Stop Program\n\n> ";
		getline(std::cin, input, '\n'); // 회전기호를 ' '로 구분하여 입력
		if (input == "1") { // 입력이 1이면 Cube 초기화
			system("cls");
			rubikscube.cubeReset();
		}
		else if (input == "2") { // 입력이 2면 Solve면 빠르게 Cube 맞추기
			system("cls");
			rubikscube.cubeSolving(false);

			// Cube를 맞추는데 걸린 시간과 Solution 출력
			std::cout << "\n\n\nSolving Time : " <<
				rubikscube.getSolvingTime() << " seconds"  << std::endl;
			std::cout << "Rotation Count : " << rubikscube.getCount() << "
				rotates\n" << std::endl;
			std::cout << rubikscube.getSolution() << std::endl;
		}
		else if (input == "3") { // 입력이 3이면 느리게 Cube 맞추기
			system("cls");
			rubikscube.cubeSolving(true);

			std::cout << "\n\n\nSolving Time : " <<
				rubikscube.getSolvingTime() << " seconds\n" << std::endl;
			std::cout << "Rotation Count : " << rubikscube.getCount() <<
				"rotates\n" << std::endl;
			std::cout << rubikscube.getSolution() << std::endl;
		}
		else {
			//회전 기호 입력을 Cube에 적용하여 출력
			system("cls");
			rubikscube.inputtoQueue(input);
			rubikscube.rotatingCube(false, 0);
		}
	}
}
```

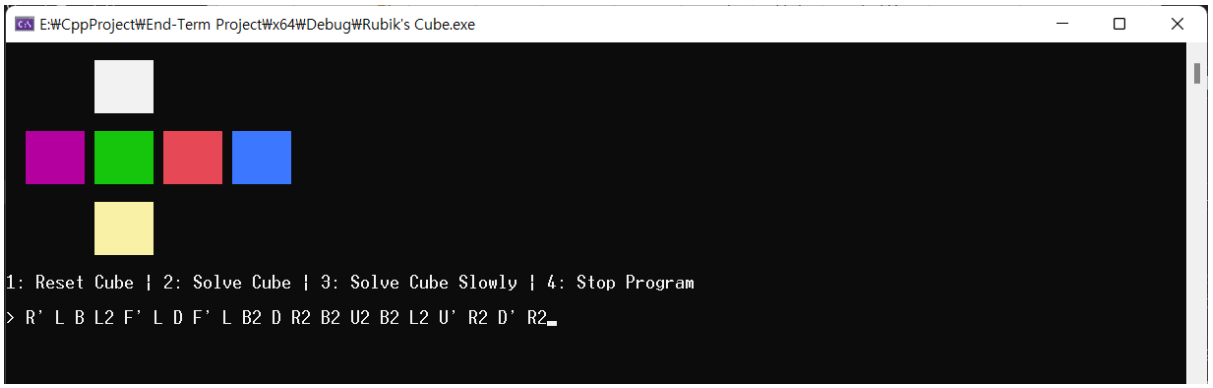
사용자는 cubeSolver 클래스 하나와, 회전 기호들을 입력 받을 input string 변수만을 선언하고, printCube, cubeReset, cubeSolving, getSolvingTime, getSolution, inputoQueue, rotatingCube의 메소드만으로 Cube를 생성, 출력, 맞추기, 섞기, 맞추는 과정 상세 출력이 가능하다.

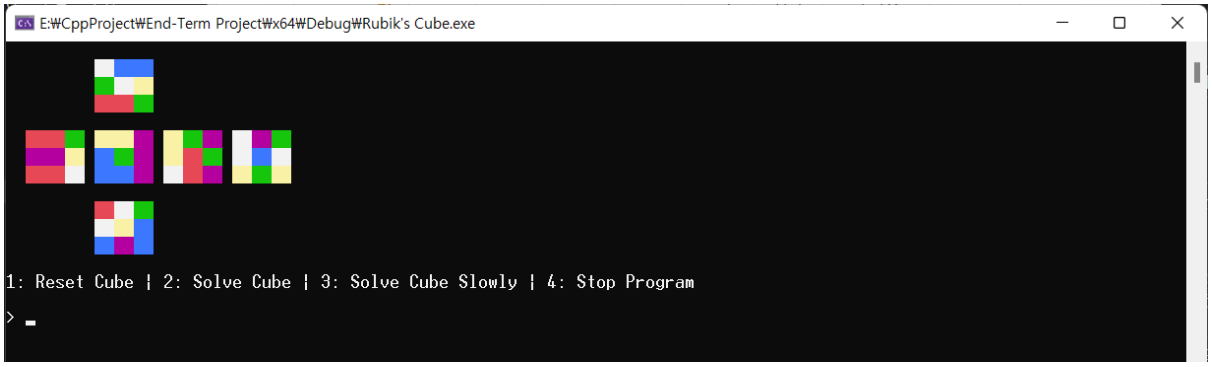
VI. 프로그램 실행 결과

위에서 구했던 "R' L B L2 F' L D F' L B2 D R2 B2 U2 B2 L2 U' R2 D' R2"의 Scramble로 Cube를 섞고

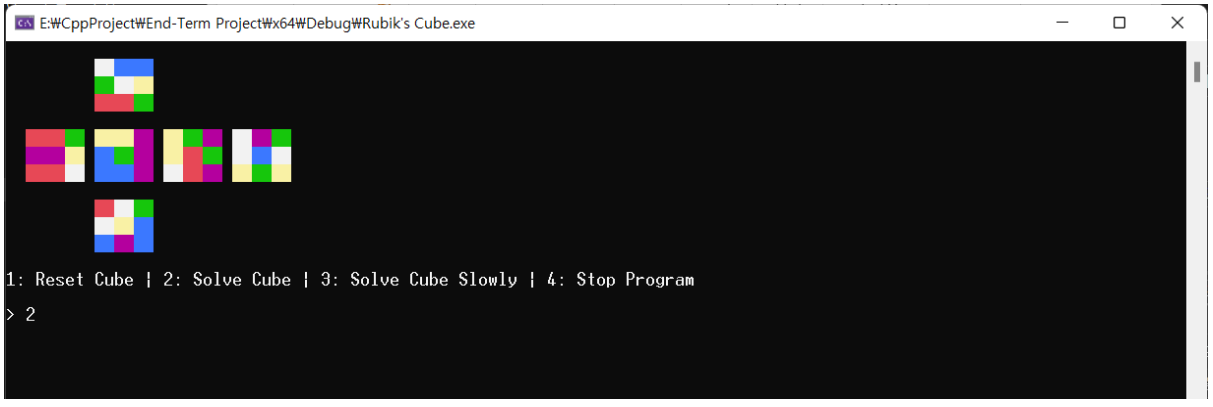
프로그램을 이용해 맞추면 아래와 같다.

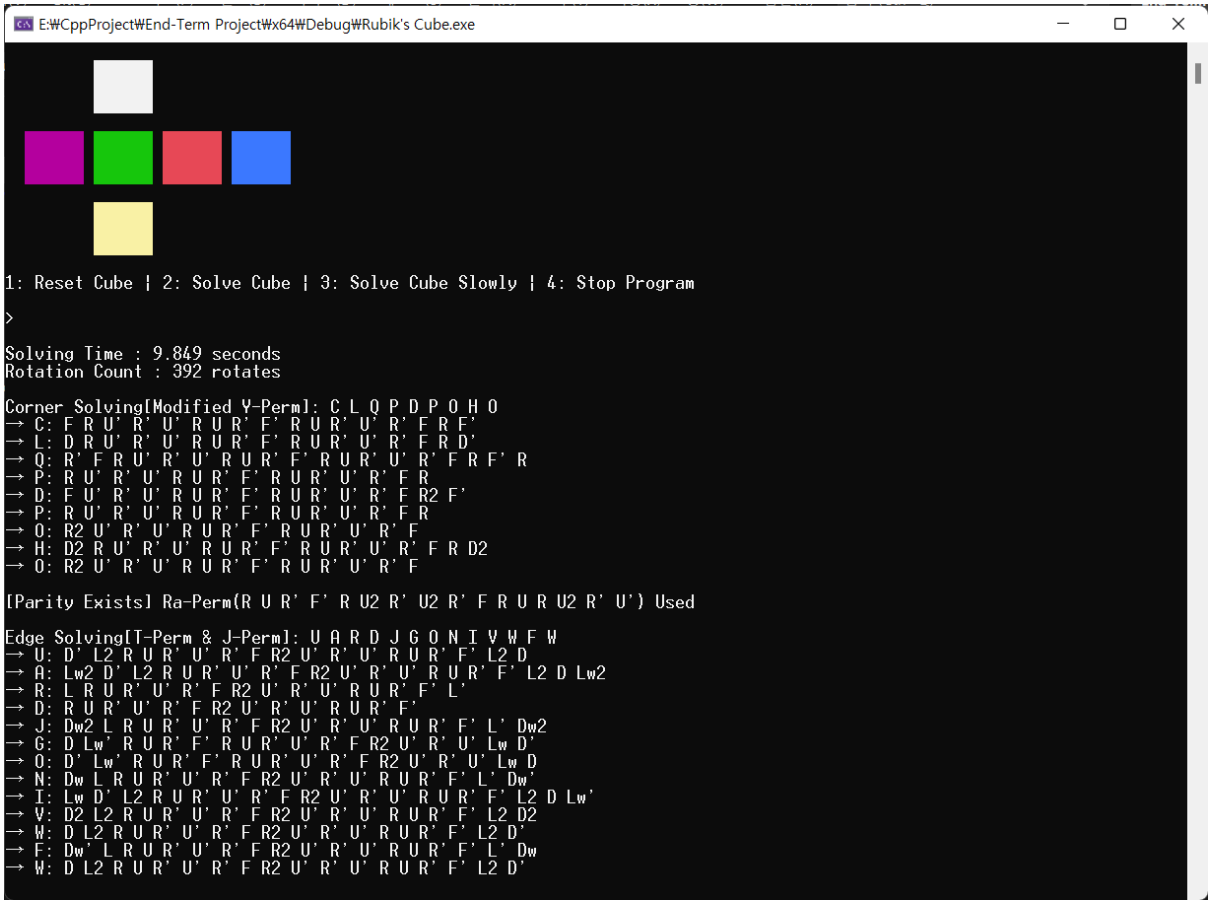
[Scramble 적용]





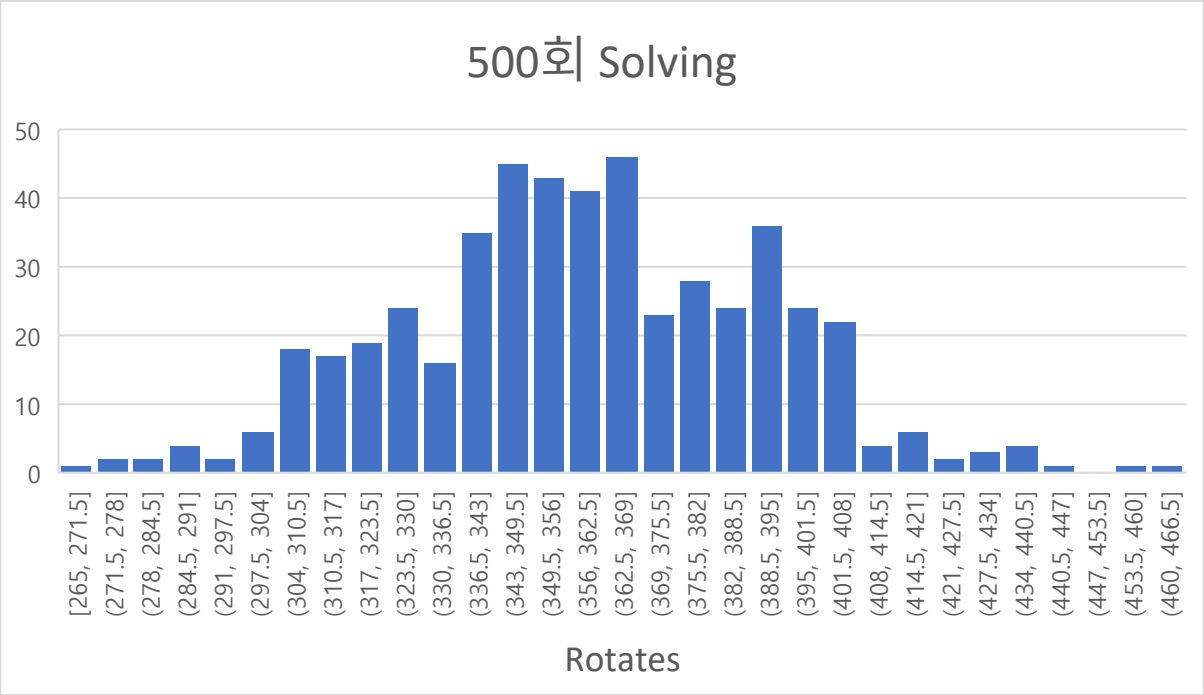
[Cube 맞추기]





Ⅶ. 결론

WCA 공식 대회 기준을 충족하는 Scramble 500개에 대해 500번의 시행을 하여 통계를 구해보았다. 500번 시행에 대해 500번 모두 Cube를 성공적으로 맞추었다.



평균 359.38 회전, 중앙값 358 회전, 표준편차 32.12 의 분포가 나타났다. 이는 공식과 공식을 이어 붙일 때 중복기호(R + R = R2), 회전-역회전 기호(R R′ = X) 등을 고려하지 않은 결과이다. 실제로 사람이 OP Algorithm으로 Cube를 맞출 때에는 이를 고려하긴 어려우므로 고려하지 않았다.

결과적으로 OP Algorithm은 평균 359.38 회전으로 임의로 섞인 모든 Cube를 맞출 수 있었다. 그러나 이는 신의 수로 알려진 평균 17.8 회전과는 매우 괴리가 있는 회전수이다. 즉, OP Algorithm은 애초에 눈을 가리고 맞추기 위해 개발된 해법으로 최적의 해법을 찾는 데에는 매우 비효율적인 해법이라 할 수 있겠다.

이후에도 2-Phase Algorithm이나 군론(commutator 등이 Cube에 적용됨) 등의 수학적 이론을 공부하여 최소 회전에 가까운 프로그램을 개발해 보는 것이 차후 목표이다.