Implementation of Quine-McCluskey Algorithm

9조

2018706075 김동욱
2021202078 최경정
2021202079 김민경
2021202087 장현웅

A. Problem Statement

퀸-맥클러스키 알고리즘(Quine-McCluskey algorithm)은 부울 대수(Boolean algebra)로 표현된 논리식을 최소화하는 알고리즘이다. 논리식을 최소화하는 개념 자체는 카르노 맵(Karnaugh map)과 동일하지만, 모든 항의 조합을 행렬의 형태로 표현하여 찾아내는 대신 이진수로 표현된 최소항(Minterm)만을 이용해 표로 나타내기 때문에, 컴퓨터를 이용해 자동화하기가 더 쉽다. 또한, 컴퓨터 자동화를 배제하더라도, 카르노 맵은 함수의 출력이 0이 되는 최소항을 포함하여 모든 항을 표에 나타내야 하기 때문에, 변수의 개수가 4개 이상이 될 경우, 수동으로 최소화 하더라도 매우 복잡해진다는 문제가 존재하는 반면, 퀸-맥클러스키 알고리즘을 이용하면 함수의 출력을 1로 만들수 있는 최소항과 Don't care인 최소항만을 포함하기 때문에 상대적으로 단순하다는 이점이 있다.

이번 프로젝트의 목표는 제한 없는 양수 개수의 Input에 대한 간소화된 Boolean function을 찾기 위한 Quine-McCluskey 알고리즘을 C/ C++로 작성하는 것이다. 다음 사진과 같이, input_minterm.txt 파일의 첫 줄에는 input term의 개수가 정수로 주어진다. 이후 각 줄마다 true minterm인지 don't care인지를 표시하는 영문자 하나와 minterm이 주어진다. Output으로는 주어진 Input을 나타낼 수 있는 PI 조합과, 이를 2 level logic의 SOP로 구현하기 위한 transistor 개수를 result.txt 파일에 출력한다.

Input format	Output format
File name: input_minterm.txt	File name: result.txt
4 // input bit length d 0000 // don't care value m 0100 // input having the result with true m 0101 m 0110 m 1001 m 1010 d 0111 d 1101 d 1111	01-- 1-01 1010 Cost (# of transistors): 40

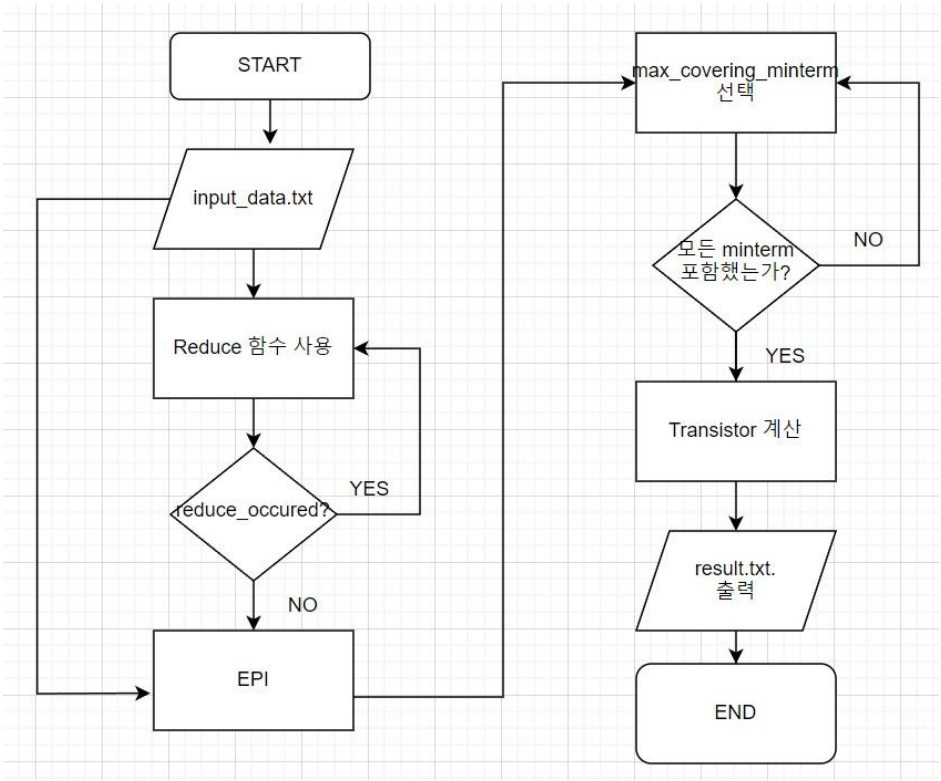
이번 프로젝트에서 구현하는 퀸-맥클러스키 알고리즘의 동작 단계는 다음과 같이 구성된다.

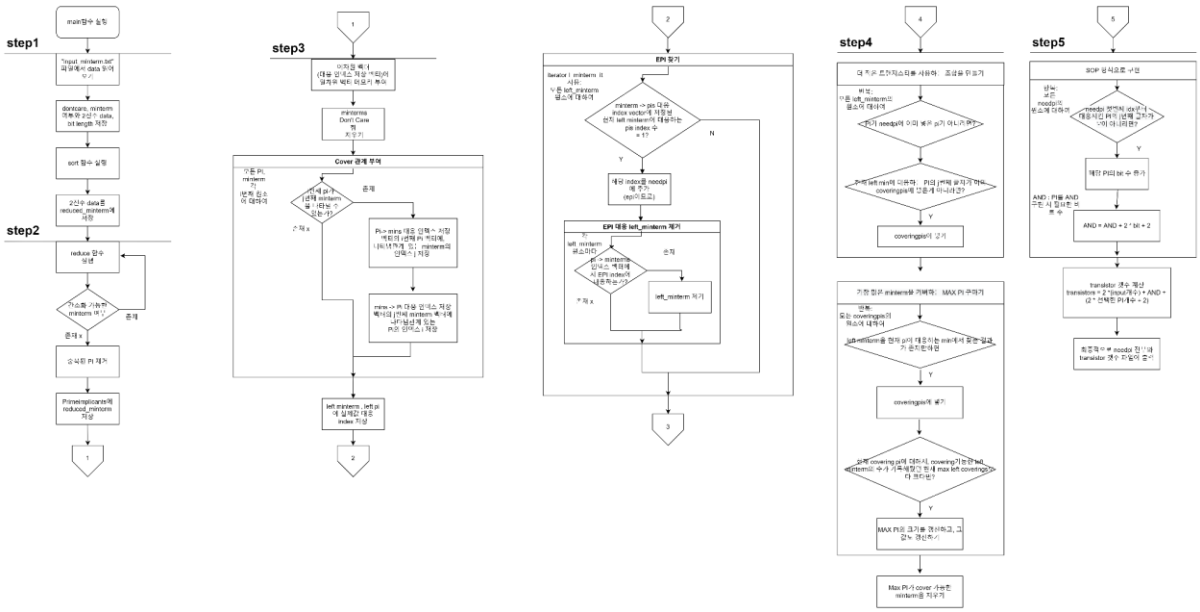
1) 텍스트 파일을 통해 간소화할 논리 식을 입력받는 단계
2) 주어진 논리 식의 후보 항(Prime Implicants)를 구하는 단계
3) 후보 항들을 이용하여 필수 항(Essential Prime Implicant)를 구하는 단계
4) 가장 많은 minterm을 나타낼 수 있는 후보 항부터 선택하여, 모든 minterm을 나타낼 수 있는 후보 항 조합 중 적은 트랜지스터를 사용하는 조합을 만드는 단계
5) 최종적으로 고른 PI 조합과 조합을 SOP form으로 나타내기 위한 transistor 개수 출력 단계

B. Algorithm with Pseudo code and flow chart

* 전체적인 알고리즘의 흐름

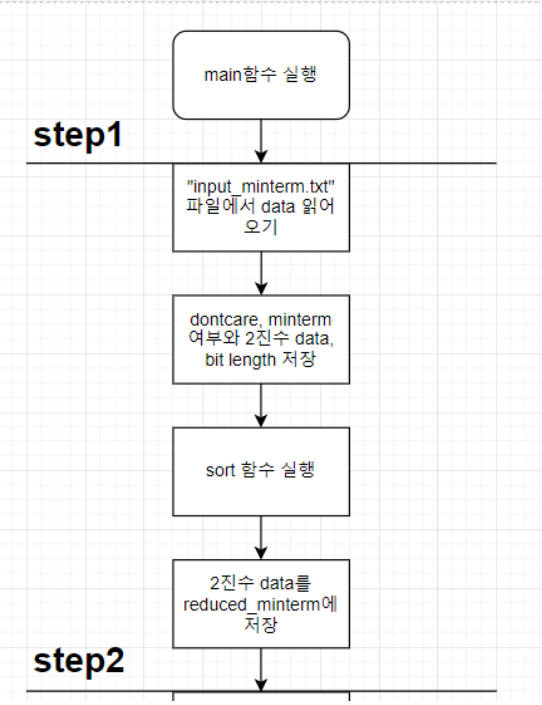
- flowchart:





1) 텍스트 파일을 통해 간소화할 논리 식을 입력받는 단계

- flowchart:



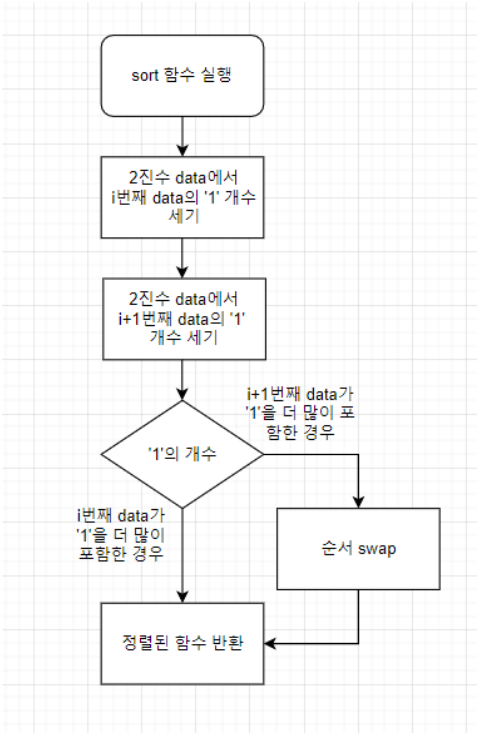
- pseudo-code:

```
vector<pair<string, string>> minterms; //dm, minterm를 저장하는 vector

    ...

char filename[30] = "입력받을 파일 이름";
    string s, ch, t; //
    int i;
    inf x;

    파일 읽기 전용으로 열기
    input file에서 input bit length 읽기

    //파일에서 한 줄씩 읽기
    while (파일의 끝에 도달할 때까지)
    {
```

파일에서 한 줄씩 읽어 s에 저장
istringstream f(s); //문자열 파싱 함수
파일에서 d또는m 과 minterm값 입력받아서 각각 ch, t에 저장
minterms.push_back(make_pair(ch, t)); //d또는m, minterm 문자열을 pair로 만들어 minterms에 push back
}
파일 스트림 닫기
minterms = x.sortinfo(minterms); //minterms을 sort()로 정렬해 minterm에 저장

이 Pseudo code는 input file로부터 입력값을 입력받는 코드이다. 파일 읽기 전용으로 input file을 열어 getline()함수를 이용하여 don't care인지 minterm인지와 minterm을 한 줄로 입력받는다. 그렇게 입력받은 각 줄을 문자열 파싱 함수로 전달하여 각각의 변수에 저장한뒤, 두 개의 string pair를 요소로 가지는 minterms vector에 저장한다.

- sortinfo 함수
- flowchart:



- pseudo-code:

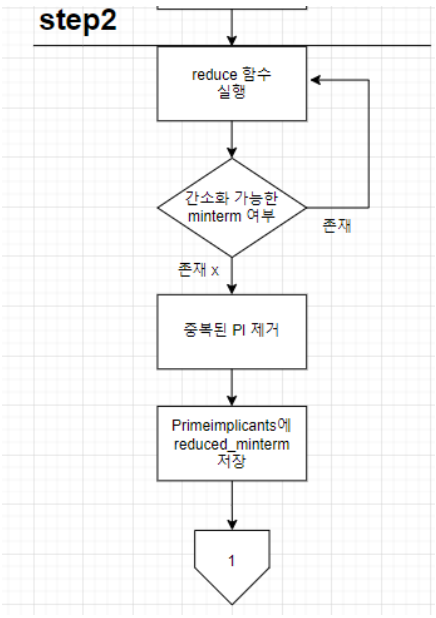
```
vector<pair<string, string>> sortinfo(vector<pair<string, string>> min)
{
```

```
        int i, j, x, y;
        for (i = 0; size of vector min - 1)
        {
                for(j = 0, size of vector min - 1 - i)
                {
                x = count(minterm of min [j], '1')
                y = count(minterm of min [j + 1], '1')

                if x > y ? swap(min[j], min[j + 1])
                }
        }
        return min
}
```

이 Pseudo code는 입력받은 minterm들을 class에 정의된 함수를 이용하여 1의 갯수로 정렬하는 code이다. class inf는 public 멤버함수인 vector <pair<string, string> >을 반환하는 함수를 가진다. sortinfo 함수는 main에서 입력받은 minterm이 저장된 vector를 매개변수로 받는다. 이중 for문을 사용하여 각 minterm에서 1이 몇 개인지 count함수로 알아내어 비교한뒤, 1의 갯수로 오름차순 정렬해준다. 그렇게 정렬된 vector를 리턴해준다.

2) 주어진 논리 식의 후보 항(Prime Implicants)를 구하는 단계

- flowchart:



- pseudo-code:

vector<pair<string, string>> minterms; //dm, minterms를 저장하는 vector

pair<vector<string>, bool> reduced_minterms; //현재 단계의 축약된 minterms와, 더 reduce 가능한지의 bool을 저장하는 pair

vector<string> PrimeImplicants; //reduced_minterm에서 asterisk 항들을 받아 저장하는 vector

...

//dm데이터 제거해 minterm을 reduced_minterm에 load

for (size of vector minterm)

reduced minterm.first 에 minterms의 minterm값만 저장

//reduced_minterms를 reduce 반복하며 asterisk 항은 PrimeImplicants에 자동 저장

do {

reduced_minterms = reduce() //User defined function

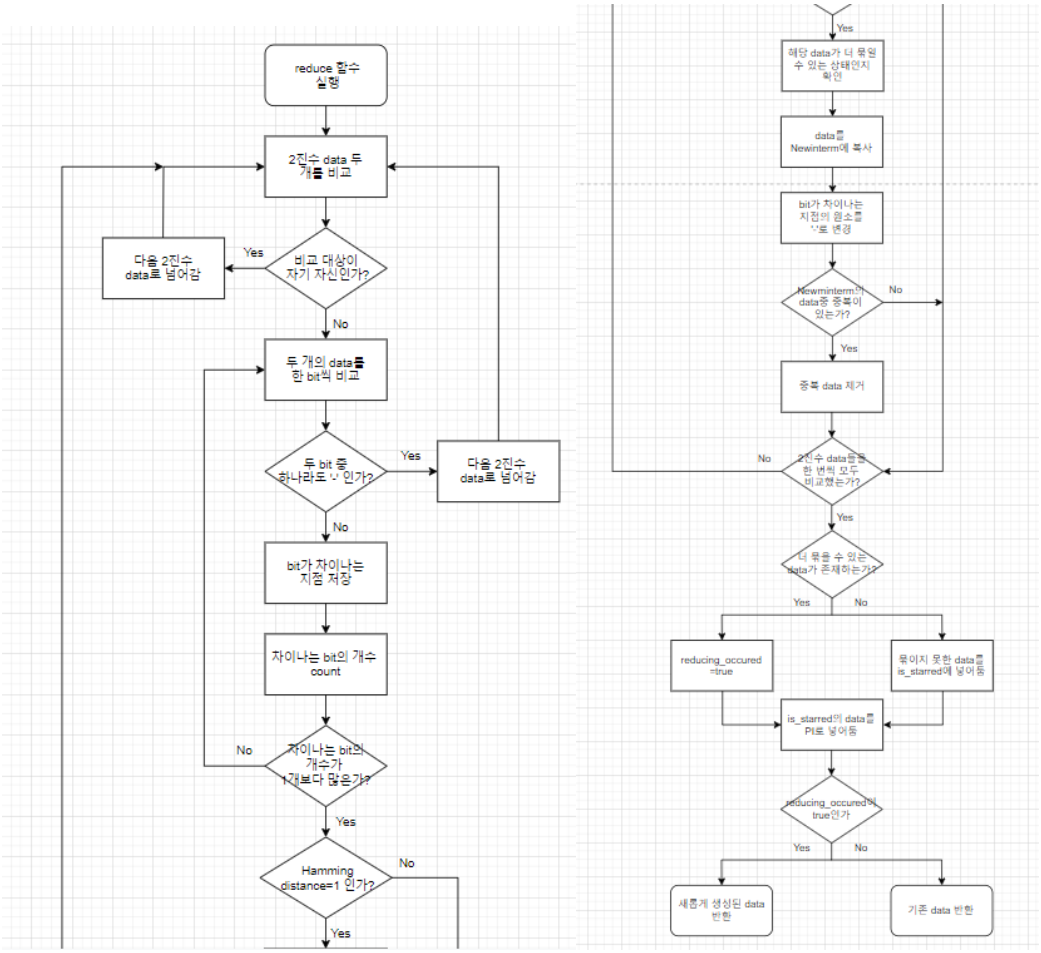
reduced_minterms.first 오름차순 정렬

reduced_minterms.first 의 중복된 PI remove

} while (reduced_minterm.second bool != false);

reduced_minterms deAllocate

- reduce 함수
- flowchart:



- pseudo-code 2:

pair<vector<string>, bool> reduce(vector<string> before_reducing, vector<string>* PI_GetStarred)

{

vector<string> reduced_minterms_now; //이번 단계의 reduced minterms
bool reducing_occured; //이번 단계에서.어떠한 minterm 쌍도 reduce되었을시, 비교하면서 true가 됨. 이후에 before_reducing과 pair로 담겨 반환
int pos_reducing; //bit차이 위치 기록
int count_reducing = 0; //bit차이 개수 기록
bool is_minterm_reducable; //각 minterm에 대해서, 다른 minterm과 reduce 가능하면 참이 됨

vector<string> is_starred; //is_minterm_reducable이 거짓이면 해당 항은 전부 여기에 포함시킴
vector<string>::iterator bef_red_it_outer; //반복문 바깥 반복자. 모든 element A로 대응
vector<string>::iterator bef_red_it_inner; //반복문 안쪽반복자 모든 element B로 대응

//쌍마다 비교하며 reduce
reducing_occured = false;
바깥쪽 반복자를 before_reducing 시작점에 두기
while (모든 before_reducing element A에 대해 검토) //바깥쪽 반복자가 before_reducing 끝부분에 도달하지않았다면
{
	if (안쪽 반복자 == 바깥쪽 반복자) { //자기 자신과의 비교 skip
	bef_red_it_inner++; //element B 다음 element 보기
	continue;
		}
	else {

	count_reducing = 0;//각 minterm쌍마다 체크할 pair개수 0으로 초기화.

	//hamming dist = 1 인지를 count_reducing으로 확인
	//쌍의 한 bit 씩 비교하며 두 minterm의 bit 다름 체크
	for (minterm의 모든 bit k에 대해 검토)
	 {
		if (element A k비트 != element B k비트) //(*bef_red_it_outer)[k] != (*bef_red_it_inner)[k]
		{

			if (두 비트 중에 '-'가 있으면) //서로 다른 두 bit 중에 '-'bit가 다르면 그건 처리 안하니까(같은곳에 하나는 1있고 하나는 '-'있으면 무의미함)		{
				count_reducing = 0; //뒤에서 1비트
				break; //두개의 bit가 다른데, 하나라도 '-'가 있다면 비교 의미 없으므로 break
			}
			else //비트는 다른데 두개 모두 '-'가 아니면
			{
				pos_reducing = k; //bit차이지점

```
                    count_reducing++;  //bit차이 개수

                }
            }
            if (bit 차이 개수 가 1을 넘었으면)
                break; //비교 안함
        }
        //for(k)가 끝나고(이 minterm 쌍에 대해,hamming dist = 1 임을 확인), 해당 지점만 '-'치환해서 reduced_minterms_now에 추가.

if (count_reducing == 1) //hamming dist = 1이라는 의미
        {
            is_minterm_reducable = true;
            string newMinterm에 element A 복사
            newMinterm[pos_reducing] = '-'; //차이지점의 원소를 '-'로 reduce
//전체쌍검사를 하니까, 자기 자신을 포함한 적이 있는지를 생각해야됀다.
            if (reduced minterm now에 이 newMinterm을 추가한 적이 없다면)
                {
                reduced_minterms_now에 newMinterm 추가시키기
                }
            }
            안쪽 반복자++; //다음 element B보기
        }
}
//for(j)가 끝나고(element A에 대해서 나머지 element B와 묶이는지를 확인했음),
            if (is_minterm_reducable == true) {
                reducing_occured = true;
            }
            else {
                is_starred에 element A 추가;
            }
            바깥쪽 반복자++;  //다음 element A 보기
        }
        //asterisk 원소 PrImeImplicants에 추가
        for (is_starred 모든 원소에대해서)
        {
```

```
            PrimeImplicants에 is_starred원소 직접 추가
        }

        pair<vector<string>, bool> p;
        if (reducing_occured == true)       //소거 성공시

        {
            before_reducing 할당해제;
            p = make_pair(reduced_minterms_now, reducing_occured); //소거된
reduced term과 reduce 일어남 bool 변수 묶어 보내기
        }
        else
                                                //소거 실패, 기존 벡터 전달
        {
            p = make_pair(before_reducing, reducing_occured);
        }
        return p;
}
```

위의 두 Pseudo code는 각각 Prime Implicants를 찾기 위한 알고리즘과, 해당 알고리즘에서 사용한 사용자 정의 함수 reduce를 나타낸 것이다. step 1에서 전달받은 vector는 don't care, minterm의 구분과 해당 논리식을 포함하고 있는데, 이를 PI를 찾는 과정에 사용하기 편한 형태의 vector로 변환한다. 그 후, 더이상 최소화가 불가능할 때까지, 최소화 하는 과정을 담은 사용자 정의 함수인 reduce를 반복적으로 사용한다.

reduce 함수는 pair로 reduce 된 minterm과 함께, 최소화할 것이 있었음을 의미하는 bool변수를 반환하므로 이것을 체크하여 최소화가 불가능함을 확인한다. reduce 함수는 전달받은 before_reducing의 모든 element A에 대해 다른 element B와 비교하여, 문자가 1개 bit만 차이난다면, 그 차이가 doncare('-')문자가 아닌 경우에, element A를 별도의 문자열에 복사해 그 차이지점을 '-'로 바꿔서 reduced_minterms_now에 저장해둔다. 동시에 is_minterm_reducable 부울변수도 참으로 만든다.

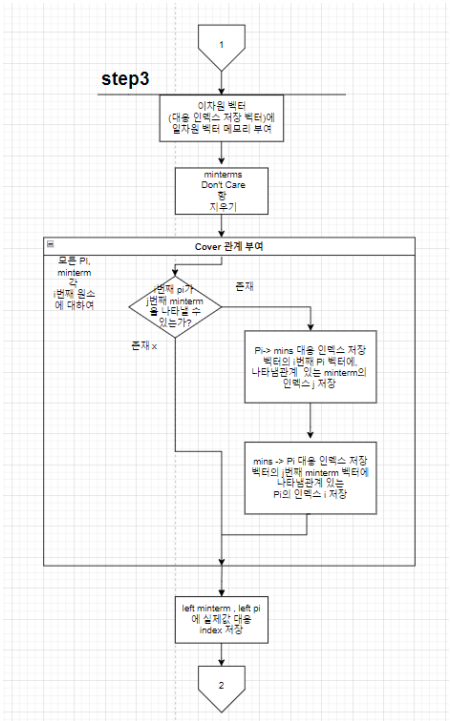
element A를 검토 종료시, is_minterm_reducable이 참이면, reducing_occured 부울 변수는 참이 되며 나중에 reduced_minterms_now와 함께 반환한다. 거짓일 경우 element A는 asterisk 항이므로, is_starred vector에 담는다. 모든 element 상호 비교 종료시, vector<string> *포인터로, 모아둔 asterisk들은 전부 PrimeImplicant에 직접 추가되게 된다.

각 element 별로 check하는 system을 만들어, 지금의 프로그램처럼 m개 원소에 대해서 순열(m*(m-1))로 이미 비교한 순서쌍을 다시 비교하지 않고, 조합 (m*(m-1)/2)으로 비교하게 할 수도 있었다. 이 system을 구현하는 데 시간이 없어, 조합으로 비교해 앞서 비교한 element와 다시 비교하지 않을 경우, 비교한 element B가 다른 element와 reduce되지 않으면 element B는 앞선 element A와 reduce됨에도, element B는 QM 알고리즘중 asterisk 항으로 처리되는 버그가 있었기 때문이다. 이것은 시스템의 작동 시간을 길어지게 하는 약점으로 남아있다.

퀸-맥클러스키 알고리즘은 최소화된 논리식을 찾기 위한 알고리즘이기 때문에, 최소화하는 과정에서 많은 반복문들을 사용하게되어 높은 시간 복잡도를 갖게 된다. 또한, 이 코드에서는 반복 동작의 간단한 구현을 위해 사용자 정의 함수를 사용하였으므로, 반복문을 실행하는 동안 큰 오버헤드가 발생할 것으로 생각된다. 이는 동작 속도에 큰 영향을 줄 것이라고 판단되므로, 다른 방법으로 접근할 수 있을지 고민해보았다.

3) 후보 항들을 이용하여 필수 항(Essential Prime Implicant)를 구하는 단계

- flowchart:



- pseudo code:

```
//don't care 지우기
minterm_it = minterms 시작지점에 부여
while (minterm_it 가 minterms를 끝까지 검토할때까) {
        if (minterm_it->first == "d")   // minterm_it에 해당하는 minterm 요소 str1 이 "d"일 경우
                minterm_it = minterms.erase(minterm_it);        //해당 원소지우기

        else //안 그럴경우
                minterm_it++; //다음 원소로 검토
}

//PI -> minterm s, minterm -> PI s 의 index 대응 관계를 index벡터화하는단계
```

//앞으로 모든 minterm, PI는 그 실제값으로 다루지 않고, minterms, PrimeImplecants에 저장된 index로 지칭한다.

```
minsize = minterms.size(); //minterm doncare 삭제 후 크기 변경

//cover 관계 부여
//'-'이 아닌 literal을 하나씩 비교하며 PI가 minterm을 cover 하면 체크
for (각 Prime Implicant의 원소마다)

{
        for (minterms의 원소마다)
        {
                bool covers = true;
                //PI와 minterm cover를 모든 쌍에 대해 조사, 다르면 비교중지
                for (모든 bit 수에 대해서) {
                        만약 PI의 해당 bit가 '-'이 아니면
                                if (PI의 해당 bit와 minterm의 해당 bit가 다르면) {
                                        covers = false;
                                break;
                                }
                        }
```

```
                    }
                    if (covers) // doncare 제외하고 모든 bit에대해서 동일함이후
                                        {
                            pi_to_mins_idx.at(i).push_back(j);
                            min_to_pis_idx.at(j).push_back(i);
/Pi-> mins 대응 인덱스 저장 벡터의 i번째 Pi 벡터에,나타냄관계  있는 minterm의 인덱스 j 저장.
//이 벡터에 Pi 인덱스를 넣으면 대응 minterms index들이 vector로 저장되어있게됨
                    }
                }
        }
```

minterm 크기만큼, PI 크기만큼, left_minterm과 left_pi에 대응 index들을 저장해두기

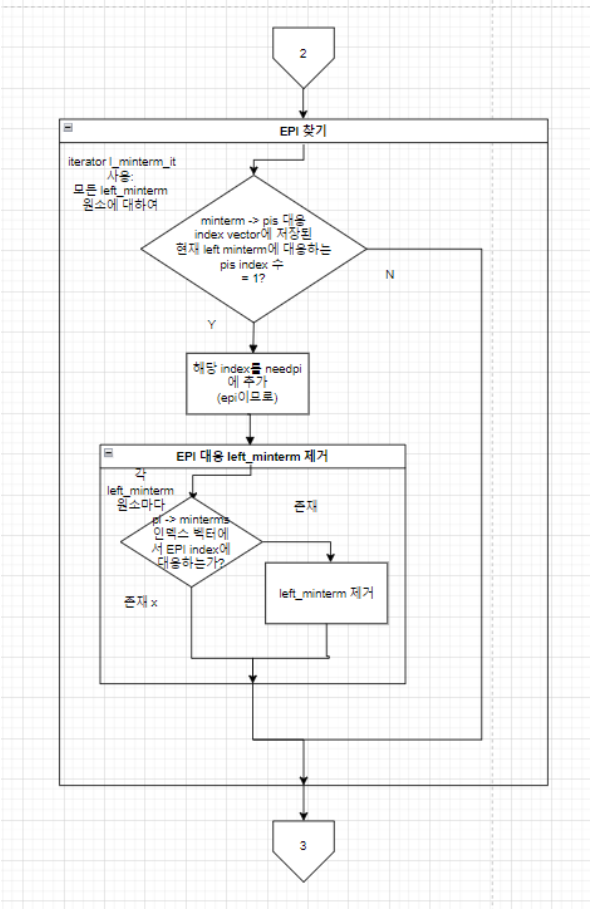
```
        }
```

QM 알고리즘에서는 PI -minterm 대응 표를 만들어야한다. 이 코드에서는 minterm을 커버할 수 있는 PI들, Pi를 커버할 수 있는 minterm들의 관계로, 대응 index를 저장하는 이차원 vector (이하 대응 index 저장 vector 또는 index vector)를 사용하여 그 표를 구현하였다.

이렇게 함으로써 앞으로 PI minterm의 실제값 대신 vector 에서 값이 저장되어있는 대응 index로 앞으로는 불러서 사용하기로 한다. PI->minterms 대응 벡터는, i번째 Pi 인덱스에 대응하는 minterm들의 index들이 묶여서 vector 원소로 저장되어있음을 의미한다. 이렇게 하면 대응의 의미를 살리면서 메모리를 아낄 수 있다.

minterm에서 doncare항을 지우고, Pi가 minterm을 커버하는 경우, Pi->minterms 대응 indexvector와 minterm->pis 대응 indexvector에 index를 집어넣는다.
Pi-> minterms 대응 vector만 있어도, 이 프로그램 구조는 아니지만, 프로그램은 메모리를 아끼면서 돌아갈 것이다. 하지만 프로그램에서 역으로 참조하는 구조를 만들어야 하므로 상당히 사용하기 불편한 코드였다.

- EPI 찾기
- flow chart:



- psedo-code:

  vector<int>::iterator l_minterm_it; //left minterm iterator

  ...

  vector<int>::iterator covered;

  l_minterm_it = left_minterm.begin(); //iterator를 left_minterm 시작 위치에 배치

  while (iterator가 끝지점에 가지 않을때) {

  if (iterator 대응 minterm index에 대해, min -> pis 대응 index vector에서, 대응되는 pi index개수가 1일때) {

  int epi_idx 에 그 대응되는 하나밖에 없는 pi index 저장;

  needpi에 epi index 추가.

//EPI부터 leftpi에서 지운다.

  l_pi_it = left_pi에서 epi index찾은 것

  left_pi에서 l_pi_it 해당하는, 선택한 pi를 지운다.

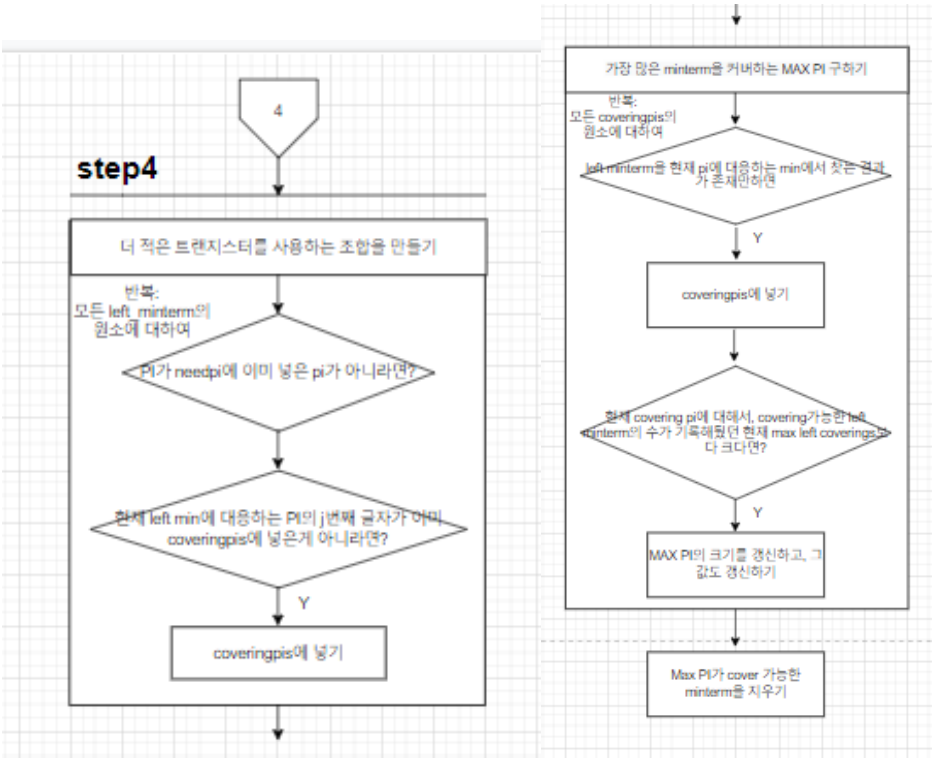
.//지운 EPI idx에 대응하는 minterm 목록에서 하나씩 넘기며, left_minterm에서 비교하여 찾는다.

```
                    for (pi-> mins 대응 index 벡터에서, epi index에 대응되는
        모든 minterm i번째에 대하여 ) {

                     now_min_crpd_pi = pi-> mins 대응 index 벡터에서, epi index에
        대응되는 minterm 중 i번째

                    covered iterator에, left_minterm 중 now_min_crpi_pi가 있는
위치를 저장

                    if (covered != left_minterm.end()) {        //만약선택한 pi 대응
        minterm들이, left minterm에서 지워지지 않고 남아잇다면

 //left minterm 중 EPI가 대응하는 minterm을 지워버림
        l_minterm_it = left_minterm.erase(covered);
                    }
            }
        }
        else  //minterm 대응 pi가 한개가 아닌경우
        l_minterm_it++;  //다음 minterm 확인하기
        }
}
```

각 left_minterm 원소들마다, min -> pis 대응 index 벡터에서, 대응하는 pI index개수가, 1인지 검사해서 그것을 EPI로 생각하기로한다.
만약 EPI일 경우, EPI를 선택함으로써 cover되는 minterm들이 여러개 있을 것이다. 그것 중 left_minterm안에 있는 것을 찾아서 지운다. pi -> mins 대응 index 벡터에서 대응 minterms를 하나씩 살펴보며, left_minterm 중 EPI가 cover하는 minterm을 역으로 찾고 erase한다.

4) 가장 많은 minterm을 나타낼 수 있는 후보 항부터 선택하여, 모든 minterm을 나타낼 수 있는 후보 항 조합 중 적은 트랜지스터를 사용하는 조합을 만드는 단계

- flowchart



남은 left_minterm 에 각각 대응하는 PI를 검색하고, 해당 PI들 중 가장 left_minterm을 많이 cover할 수 있는 PI를 골라, neepi 에 저장한 후, 그 PI를 left_pi에서 지우고. 그 PI에 대응하는 minterm들을 left_minterm에서 지워버릴 것이다.

- psedo-code:

vector<int> coveringpis;
//left_minterm을 cover 할 수 있는 PI들을 coveringpis 에 저장할 것

int max_covering_PI; // 그중에서 단 하나의 뽑아내기로 한 PI
int max_coverings;
// 이후에 coveringpis의 PI마다 탐색하며 가장 많은 left minterm을 cover 가능한 pi를 도출한다. 현재까지 탐색한 PI가, cover 가능한 left_mineterm의 개수 중 가장 컸던 값을 max_coverings 에 저장할 것이다. max_coverings의 초깃값은 0이다.

...

while (left_minterm.begin() != left_minterm.end()) { //left minterm 원소가 완전히 사라질때까지, 없으면 종료

// left_minterm을 cover 가능한 PI 들을 찾아, coveringpis에 넣을 것.

max_coverings = 0; //초기화

//left_minterm을 cover 가능한 PI 목록을 가져와 coveringpis에 쌓기

for (left_minterm 각 원소에 대해) {

//left minterm의 원소 -> 대응하는 PI들을 뽑아내는 step이다.

for (int j = 0; j < min_to_pis_idx.at(left_minterm.at(i)).size(); j++) {

//left_minterm의 i번째 원소에 대해, cover 가능한 pi 목록을 min→ pi 대응 index 저장 벡터에서 대응시켜, 목록의 j번째 원소에 대해 j++ 하며 검토한다.

int now_pi_crpd_lmin = min_to_pis_idx.at(left_minterm.at(i)).at(j);

//now_ pi_crpd _lmin = left_minterm의 I번째 minterm에 대해, cover 가능한 pi 목록의 현재 원소 (j번째 pi(의 index)) . 현재 pi that corresponded to left minterm의 줄임말이다.

if (needpi 전체에서, now_pi_crpd_lmin이 없다면) { // PI가 needpi에 이미 넣은 pi가 아니라면

// 여러 minterm들에 걸쳐, 동일한 PI가 대응되므로, 반복하며 이미 대응 PI를 coveringpis에 넣었을 수가 있다.

if (coveringpis 전체에서, now_pi_crpd_lmin이 없다면) //now_pi_crpd_lmin를 이미 coveringpi에 넣은게 아니면

coveringpis에 now_pi_crpd_lmin을 추가한다.

}

}

}

//coveringpis들을 한개씩 살펴본다. cover 가능한 left minterm을 coverings에 넣어가면서 max_covering인 PI max_covering_PI 구하기

```
            for (coveringpis 전체에 대해서) {
                                        // left minterms 순서대로 불러오기
                    int now_cvpi = coveringpis.at(i);
//now_cvpi = coverings 중 현재 살펴보는 원소.
                    vector<int> coverings;
// PI마다, cover 가능한 left minterm (index) 목록 = coverings

                    for (left_minterm 각 원소마다) {
        //coveringpis 순서대로 불러오기
                            vector<int>::iterator findlmin_crpd_pi =
find(pi_to_mins_idx.at(now_cvpi).begin(), pi_to_mins_idx.at(now_cvpi).end(),
left_minterm.at(j));

//findlmin_crpd_pi = coveringpis i번째 원소 pi를, pi_to_mins_idx 검색해 나온 mins,
에서, 지금검토중인 j번째 left_minterm 원소가 존재하는지 find()한 결과 iterator다.

                            if (pi -> mins indexvector 에서, now_cvpi의 결과로
대응하는 minterms vector에, findlmin_crpd_pi의 위치가 존재만 한다면) { //left
minterm을 현재 pi대응mins에서 찾는 결과가 존재만하면,
                                    coverings에 left_minterm(j)를 넣는다.
                            }
                    }

                    if (한개의 covering pi에 대해서, covering가능한 left minterm의
수가 기록해뒀던 max left coverings보다 크다면) {
                            max_coverings 를 현재 coverings의 원소개수로 갱신
                            max_covering_PI 를 now_cvpi로 갱신
//최종적으로 이것을 반복하며 가장 큰 coveringpis의 index가 max_covering_PI가 됨
                    }
                    vector<int>().swap(coverings); //이제 now_cvpi, 즉
coverings.at(i)가 cover하는 left_minterms들을 비움
            }


//가장 큰 covering PI를 needpi에 넣고, 그에 대응하는 left minterm 구해서 삭제
            max_covering_PI를 needpi에 넣는다.
```

```
            for (max_coverings 개수 동안) {
        //max_coverings = max_covering_PI가 하는 left minterm 개수
                    auto p = find(left_minterm.begin(), left_minterm.end(),
pi_to_mins_idx.at(max_covering_PI).at(i));
//max_covering_PI가 cover 가능한 minterm을 차례로 left minterm에서 찾아서 지워버림
                    left_minterm.erase(p);
            }
            //최종적으로 현재의 left minterm마다 최대 coverings의 PI와 covering
하는 minterm을 뽑아내면서 left_minterm은 0이됨
        }
```

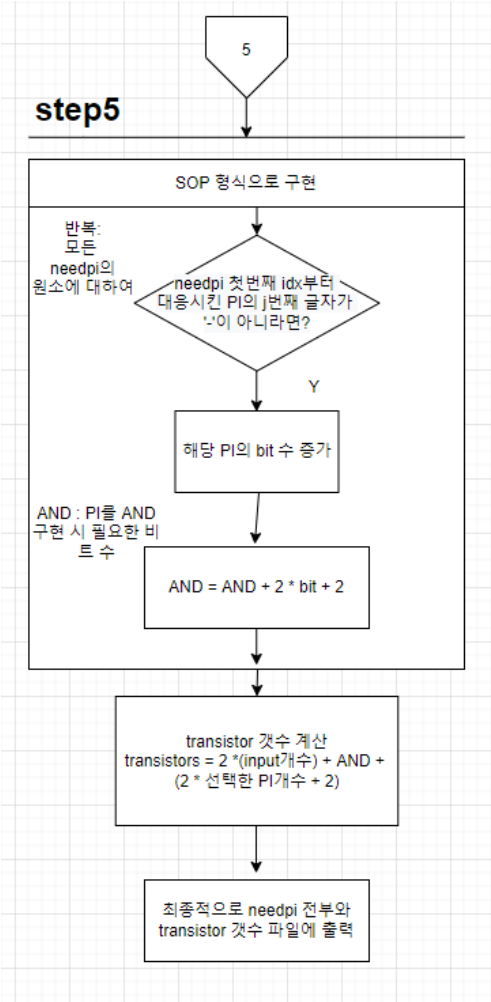
EPI를 고른 이후 left_minterm들을 전부 cover할 수 있는 PI 조합을 찾는다. 찾는기준은, 남은 left_minterm 중, pi가 커버할 수 있는 minterm이 가장 큰 PI부터 우선적으로 고르고, left_minterm들을 삭제하는 알고리즘으로 채택하였다.
transistor에 필요한 수학식 transistors = 2 *(input개수) + sigma(2* 각 PI에 필요한 term 개수 + 2) + (2 * 선택한 PI개수 + 2) 를 생각해볼 때, 더 많은 left_minterm이 커버될 수록 해당 PI를 AND게이트로 구현할 때 필요한 transistor 개수가 줄어드는 것을 볼 수 있었기 때문이다.
하지만 이것은 cover 가능한 개수가 동일할 때 등의 다른 성질등을 고려하지 않은 알고리즘이라는 한계가 존재한다. 따라서, transistor 개수는 가능한 PI 중에서 최소가 아니다.
이 알고리즘으로 cover 가능한 transistor 개수와, petrick's method를 사용하여 cover 가능한 transistor 개수의 수학적인 크기 차이를 계산해보지는 못하였다. 장기간 test하면서 실험하여야 그 결과를 이해할 수 있을 것이다.

별도로, PI 조합을 고르는 과정상에서, Petrick's Method 라는것이 존재한다. 모든 minterm이 cover 되었을 때 true가 되는 boolean 함수를 POS 형태로 나타내는 것이다. PI들은 선택되었을 때 true가 되는 bool input 변수로 생각하며, 각각의 minterm을 cover 가능한 PI들의 SUM으로 나타낸다. 이후 해당 SUM들을 서로 곱하면, 각각의 minterm들을 cover 할 수 있는 PI를 sum으로 전부 고를 때 전체 boolean 함수는 true 가 된다.
이 POS 형태를 다시 SOP형태로 고치게 되면, 각각의 항들은 PI들의 SOP 형태로 나타나게 되는데 즉 어느 하나의 PI곱 항을 선택해도 만든 boolean 함수는 참이 되며 모든 minterm은 cover된다는 것이다. 따라서 이 각 PI곱 항 중 PI 개수가 가장 적은 PI곱 항을 선택하면 transistor 개수는 최소가 될 것이다. 이것이 Petrick's Method이다.

가장 좋은 알고리즘은, EPI를 먼저 걸러내서 덮을 minterm수를 줄여서 petrick's method를 적용하는 것일 것이다. 하지만, 조사상의 한계 때문에 어쩔 수없이 이 알고리즘을 선택하게 되었다.

5) 최종적으로 고른 PI 조합과 조합을 SOP form으로 나타내기 위한 transistor 개수 출력 단계

- flowchart



- pseudo-code:

ofstream fout("result.txt"); //쓰기 전용으로 파일 생성 및 열기

//SOP 형식으로 구현
for (i, needpi의 size) { //needpi 전체에 대해
PrimeImplicants의 needpi의 i번째 string 파일에 출력
bit 변수 0으로 초기화
for (j, MINTERM_LENGTH만큼) {

```
                    if (PrimeImplicants의 needpi의 i번째에 있는 string의 j번째
literal이 '-'이 아니라면)                    //needpi 첫번째 idx부터 대응시킨 PI의, j번째
글자가 -이 아니면
                        bit++;
                            //해당 PI의 bit 수 추가
            }
            AND = AND + 2 * bit + 2;
                //이 PI를 AND 구현 시 필요한 비트 수
        }

        transistors = 2 * static_cast<unsigned long long>(MINTERM_LENGTH) +
AND + (2 * needpi.size() + 2)            //총 트랜지스터 개수

        transistors의 개수 출력

        파일 닫기
```

SOP form형식으로 구현하고, transistor 개수를 출력하는 코드이다. 선택한 pi (index)들인 needpi에 대응하는 PI와, SOP 형태로 구현하기 위한 transistor 개수를 출력한다.

결과값은 파일 출력을 통해 출력하는데, needpi의 크기만큼 반복하여 Prime Implicants vector에서 needpi에 담긴 index에 대응하는 PI값을 받아 출력한다. 그리고 변수 bit를 0으로 초기화하고, needpi 원소에 대응하는 PI 실제값의 '-' (don't care)가 아닌 bit 마다 bit를 추가한다. 그리고 AND에 값을 해당 PI를 AND로 구현하는데 드는 transistor만큼 증가시켜 저장한다.

transistors 개수 저장 변수와 bit수 저장 변수를 사용했다. 변수 AND는 각 PI를 PRODUCT로 구현하기 위해 AND게이트에 쓰이는 transistor 합계이다. transistors의 개수는 '2 * static_cast<unsigned long long>(MINTERM_LENGTH) + AND + (2 * needpi.size() + 2)'의 식을 통하여 구한다.

C. Verification strategy & Examples

예제 1

input

4

d 0000

m 0100

m 0101

m 0110

m 1001

m 1010

d 0111

d 1101

d 1111

output

01__

1_01

1010

Cost(#of transistors): 40

프로그램이 정상적으로 동작하는지 확인하기 위해 프로젝트의 제안서에서 제시된 테스트 케이스를 가장 먼저 수행해 본 결과, 정상적으로 동작하는 것을 확인할 수 있었다.

예제 2

input

4

m 0000

m 0100

m 0101

m 0110

m 1001

m 1010

m 0111

m 1101

m 1111

output

1010

0_00

1_01

01__

_1_1

Cost(#of transistors): 58

예제 2의 경우, 입력받은 minterm들이 전부 don't care가 아닌 경우이다. 만약 입력받은 모든 minterm이 true minterm이면, PI table의 row에 입력받은 모든 minterm들이 포함되어야 하므로, 이를 이용하여 Essential Prime Implicants를 구하는 과정에서 오류가 발생할 수도 있겠다는 생각이 들어 test case로 추가했다.

예제 3 (다 d이라면)

input

4

d 0000

d 0100

d 0101

d 0110

d 1001

d 1010

d 0111

d 1101

d 1111

output

Cost(#of transistors): 10

위의 경우에는 입력받은 minterm들이 모두 don't care일 경우이다.

예제 4

input

3

d 001

m 010

m 011

d 100

d 101

m 110

m 111

output

_1_

Cost(#of transistors): 14

예제 4의 경우, 입력 변수가 4개였던 앞의 예제들과는 다르게, 입력 변수가 3개인 경우이다. 이번 프로젝트에서 설계한 알고리즘은 입력 변수의 개수에 관계 없이 간소화된 논리식을 출력할 수 있어야 하므로, 입력 변수의 개수가 바뀌었을 경우에도 정상 동작하는지를 확인하기 위해 테스트해 보았다.

D. Testbench hard to solve

1) 입력받은 minterm들이 모두 don't care인 경우

결론적으로 논리회로를 구성하였을 때 함수 F = 1이다. 이것을 SOP의 2level logic으로 제대로 구현하였는지 체크하여야만 한다.

2) minterm 중 input이 1개인 input이 존재할 경우

예를 들어 F = B는 필요한 transistor가 사실 0개이다. 이것은 우리 조에서도 예외처리를 하지 않았는데, 이것에 대하여 다른 조들에 대해서도 검토가 필요하다.

3) inverter를 사용하지 않는 input이 존재할 경우

inverter는 쓸모가 없다. 따라서 이것을 어떻게 처리하게 될 지에 대한 예외처리가 필요하다.

E. References

- C++로 구현하는 자료구조와 알고리즘 (Michael T. Goodrich, WILEY)
- Digital Design (Morris M. Mano, Pearson Education)
- petrick's method (https://blastic.tistory.com/204)

F. 추가 자료

- flowchart: 안될 시 공지의 drive 확인

sortinfo():

https://app.diagrams.net/#G1_9cZvjhmwvwPsA72ZPf8XlOF5JDJka4G

main:

https://app.diagrams.net/#G1Gtgl5Tc0IEs1qBDR3dqb2a7mSKrFzJYl

reduce():

https://app.diagrams.net/#G1Nw2lFyV-wlzxG_43KqCGpI_Ezz9PbDsp