디지털 논리회로 HW2
Quine-McCluskey algorithm 보고서

2021202085 컴퓨터정보공학부 전한아솔

A. Problem statement

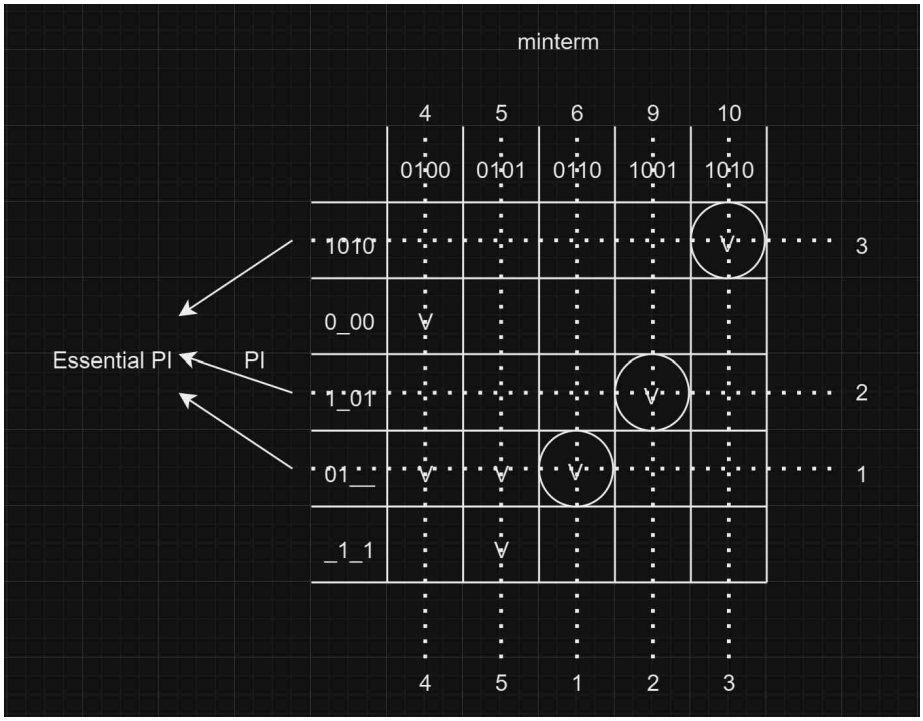
Quine-McCluskey 알고리즘을 직접 구현하는 문제로, 입력으로는 bit의 수와 minterm이 있다. 해당 minterm들을 가지고 QM 알고리즘을 적용하여 output으로 Y를 구성하는 Essential PI를 출력한다. QM 알고리즘의 핵심인 PI와 Essential PI를 찾는데 중점을 두고 과제를 간략화하기 위해서 don't care 입력은 없다고 가정하며, 모든 입력은 항상 Essential PI를 가진다고 생각하고 QM 알고리즘을 c++를 이용하여 구현한다.

A- i . QM-algorithm

퀸맥(Quine-McCluskey, QM) 알고리즘이란, 카르노 맵의 단점인 변수의 제약을 극복하고 5개 이상의 변수를 가지는 부울식을 최소화할 수 있는 알고리즘이다. QM 알고리즘을 적용하면 SOP의 형태로 최소화된 output의 부울식을 찾을 수 있다.
QM 알고리즘의 주요 단계는 4가지로 구성된다.
1. 입력된 minterm과 don't care를 2진수로 변환하여 1의 개수가 같은 항끼리 그룹화하여 첫 행을 구성한다.
2. 맨 위 그룹부터 시작하여 1의 개수가 하나 많은 그룹과 모든 항을 비교하여 Hamming distance가 1인 항끼리 묶어 다음 행을 구성한다. 이때 값이 하나만 다른 자리를 _로 표기하여 다음 행의 항으로 넣는다.
3. 이렇게 인접한 모든 항을 비교하며 결합한 항들은 표시한다. 더 이상 결합되는 항이 없을 때까지 반복한 후, 표시되지 않은 항들을 따로 뽑아내어 주항차트를 그린다.
4. 주항차트를 통해서 Essential PI를 찾는다.

A- ii . 주항차트(Prime implicant chart)

주항은 PI를 의미한다. 주항차트는 최소항을 포함하는 PI를 찾고, 그중에서 Essential PI를 찾는 데에 사용되는 차트이다.



행은 PI, 열은 minterm들로 구성한다. 한 PI가 표현할 수 있는 최소항을 체크하여 주항차트를 채운 후, 한 minterm을 보면서 해당 minterm을 커버하는 PI가 하나라면(세로로 비교하며 체크의 수가 하나라면) 그 PI는 Essential PI로 채택한다. 그림을 보게되면 0110, 1001, 1010은 각각 커버하는 PI가 하나이므로 Essential PI로 채택했다. Essential로 채택한 PI가 포함하는 다른 minterm들도 있다면, 해당 minterm도 제외한다. 그림에서 1번은 0110의 체크가 1개이므로 01__을 Essential로 채택하고 6은 이미 커버했으므로 제외한다. 2번, 3번을 반복한 후, 1010, 0_00, 01__이 커버하는 minterm들을 찾아서 제외한다(4, 5번 과정). 예시에서는 3개의 Essential PI가 모든 minterm을 커버하므로 끝이다. 하지만 아직 커버되지 않은 minterm들이 남아 있다면 Essential로 채택하지 않은 PI를 비교하여 가장 많은 minterm을 포함하는 PI를 채택하는 방식을 반복하여 모든 minterm이 커버되도록한다.

B. Your algorithm with pseudo code and flow chart

B- i . pseudo code

main:

입·출력 파일 open
bit수 저장
minterm들 string 벡터에 저장
2차원 벡터의 행을 bit+1로 할당
1의 개수로 minterm들을 그룹화하여 2차원 벡터에 저장
2차원 벡터를 리스트(qm)에 push
qm의 첫 cloumn으로 모든 PI table 구성(모든 column 생성)& PI 체크
start for(qm의 시작부터 끝까지)
    PI만 따로 저장

```
        end for
    PI의 중복 제거
    Essential PI만 저장
    출력파일에 출력
    입·출력 파일 colse
```

2차원 벡터의 행은 1의 개수, 열은 해당 개수를 가진 항들로 구성 되어있다.

PI table구성:

```
    2차원 벡터 next_column의 행을 bit+1의 크기로 미리 할당
    for(2차원 벡터의 마지막 행 전까지 반복)
        for(2차원 벡터의 열을 반복)
            for(다음 행의 크기만큼 반복)
                한 행의 minterm과 다음 행의 minterm을 모두 비교
                if(결합이 가능하다면)
                    결합한 항을 다음 열에 저장
                    end if
            end for
        end for
    end for
    if(다음 열이 비어있다면)
        return;
    else
        qm에 다음 열을 push
        다음 열을 전달하여 함수 반복
```

각 column에서 PI들만 추출:

```
    for(2차원 벡터의 마지막 행 전까지 반복)
        for(2차원 벡터의 마지막 열 반복)
            for(다음 행의 크기만큼 반복)
                한 행의 minterm과 다음 행의 minterm을 모두 비교
                if(결합이 가능하다면)
                    해당 행의 minterm에 *을 붙여서 결합을 체크
                    다음 행의 minterm에 *을 붙여서 결합을 체크
                end if
            end for
        end for
    end for
```

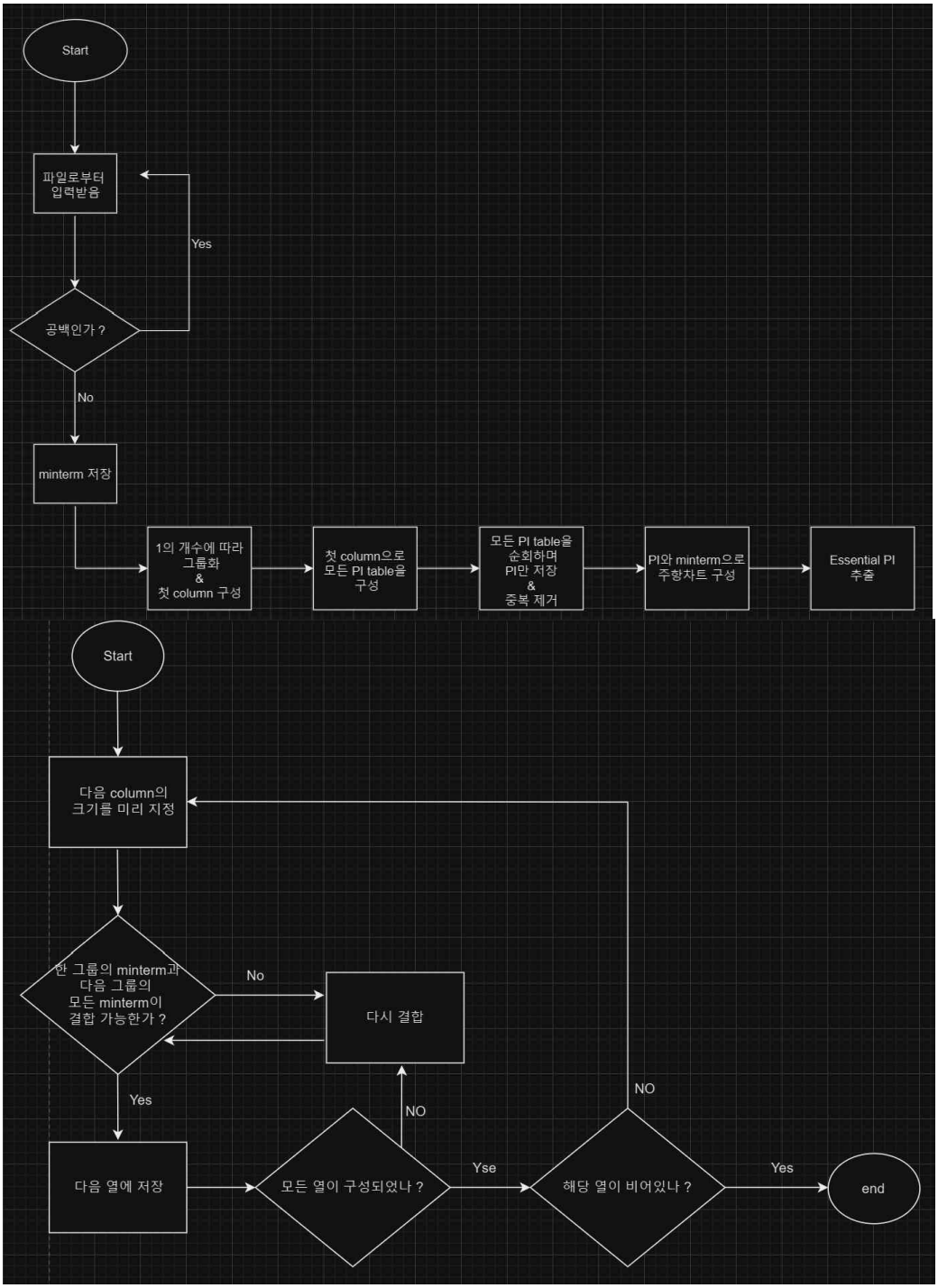
Essential PI 추출:

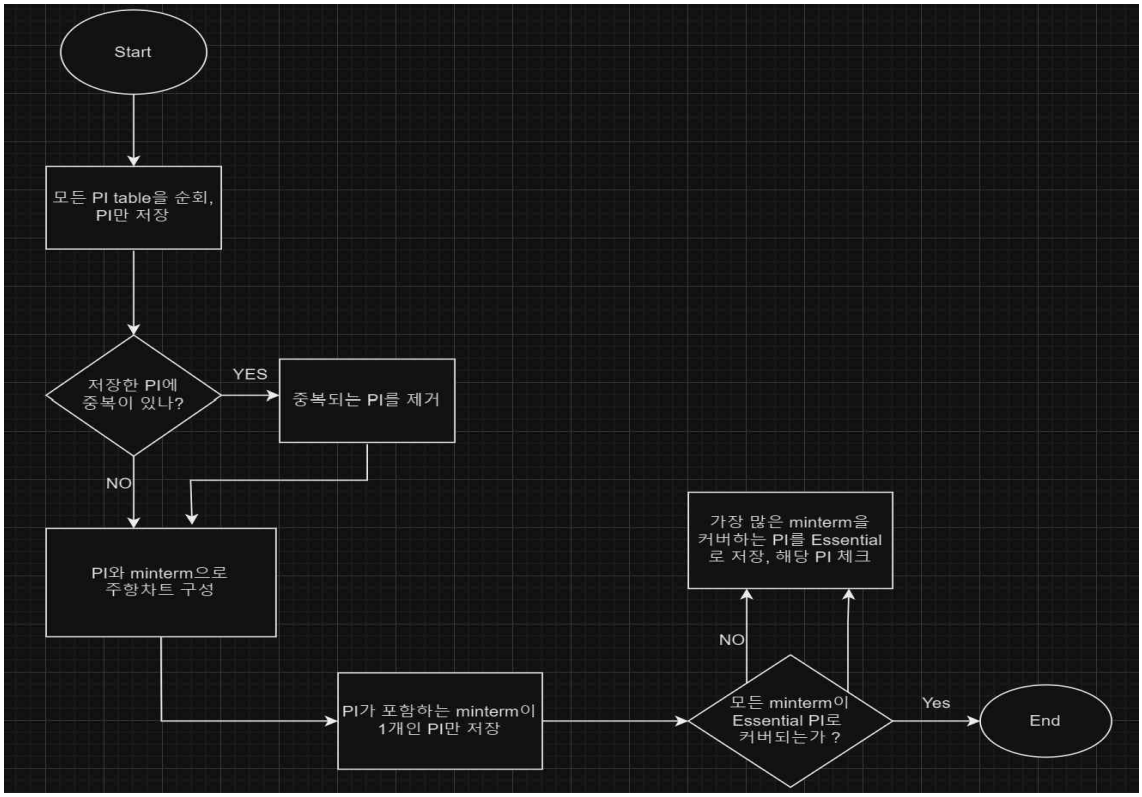
```
    주항차트를 구성하는 2차원 벡터(chart)의 크기를 PI의 개수+2, minterm의 개수
    +2로 미리 할당
```

```
0,0을 제외하고 0행에 minterm 저장
0,0을 제외하고 0열에 PI 저장
for(1부터 PI의 개수만큼 반복)
        for(1부터 minterm의 개수만큼 반복)
                for(bit의 크기만큼 반복(minterm과 PI의 인덱스를 일일이 반복))
                        if(PI의 인덱스가 _면)
                                다음 반복
                        else
                                if(문자가 다르다면)
                                        개수 증가
                                end if
                end for
                if(_를 제외하고 모든 숫자가 같다면)
                        해당 PI와 minterm이 만나는 행과 열에 *로 체크
        end for
end for
for(1부터 minterm의 개수만큼 반복)
        for(1부터 PI의 개수만큼 반복)
                if(chart[i][j]==“*”-->한 열씩 비교)
                        PI가 포함하는 minterm의 개수 증가
                        해당 행 저장
                        해당 열 저장
                end if
        end for
        if(한 PI가 포함하는 minterm의 개수가 1이라면)
                해당 PI는 Essential로 저장
                해당 PI에 *을 붙여서 체크
                해당 minterm에 *을 붙여서 체크
        end if
end for
for(1부터 PI의 개수만큼 반복)
        if(PI가 체크되어 있다면(Essential로 뽑혀있다면))
                for(1부터 minterm의 개수만큼 반복)
                        해당 PI가 커버하는 모든 minterm을 체크
                end for
        end if
end for
while(1)
        if(모든 minterm이 체크되어 있다면)
                종료
        for(1부터 PI의 개수만큼 반복)
                if(체크되지 않은 PI)
                        각 PI가 커버하는 minterm의 개수 저장
```

end for
가장 많은 minterm을 커버하는 PI를 Essential로 저장
해당 PI에 *을 붙여서 체크
해당 PI가 포함하는 아직 체크되지 않은 minterm에 *을 붙여서 체크

B-ii. flow chart





C. Verification strategy & corresponding examples with explanation

C- i . Verification strategy

PI: prime implicant

column: PI table에서의 한 열

그룹: column에서 1의 개수로 묶은 그룹

<라이브러리&함수&전역변수>

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;
int Number_one(string temp);//1의 갯수를 반환하는 함수
void next_col(vector<vector<string>>& column, vector<vector<string>>& next_column);//Hamming distance가 1인 최소항끼리 결합하여 새로운 벡터를 구성하는 함수
void combine(string& s1, string& s2, string& newstr, int& flag);//최소항끼리 결합하여 새로운 implicant를 만드는 함수
void continue_combine(list < vector<vector<string>>>& qm);//결합이 안될때까지 반복하여 모든 행을 만드는 함수
int Hamming_dis(string s1, string s2);//두 최소항의 Hamming distance를 반환하는 함수
void prime_implicant(vector<vector<string>>& column);//한 행에서 prime implicant를 뽑고 결합여부를 체크하는 함수
void Essential(vector<string> minterm, int num);//prime implicant에서 Essential만 뽑아내는 함수
int bitLen = 0; //비트의 길이
list<string> prime;//prime implicant를 저장하기위한 리스트
list<string> essential;//최종적으로 Essential만 저장하기위한 리스트
```

파일 입출력을 위해서 fstream 라이브러리를 사용했고, minterm과 PI를 string형으로 받기 위해 string 라이브러리, PI table을 구성하거나 여러 string을 저장하고 길이를 동적으로 늘리기 위해서 vector, 완성된 PI table을 담기 위해 list, min이나 max함수를 사용하기 위해서 algorithm 라이브러리를 사용했다.

<main 함수>

```
int main() {
    ifstream fin;//파일로부터 읽어오기위한 객체
    ofstream fout;//파일에 저장하기 위한 객체
    string temp;//파일로부터 읽어와 임시로 저장할 변수
    vector<string> minterm; //minterm을 저장할 벡터
    list < vector<vector<string>>> qm;//implicant table의 한 column을 2차원 벡터로 저장하여 노드의 데이터로 가지는 리스트
    int num_min = 0;//minterm의 갯수
    list < vector<vector<string>>::iterator it;//반복자
    fin.open("20input.txt");
    fout.open("result2.txt");
    getline(fin, temp);
    bitLen = stoi(temp);//bit의 수를 저장

    while (!fin.eof()) {//파일의 끝까지 반복
        getline(fin, temp);
        if (temp == "")//공백 무시
            continue;
        temp.erase(0, 2);//앞에서부터 2칸 삭제
        minterm.push_back(temp);//minterm을 저장하는 리스트에 push
        num_min++;//minterm의 갯수 증가
    }
    vector<vector<string>> column(bitLen + 1); //최소항들이 저장될 2차원 벡터의 행의 크기를 미리 할당
    int num = 0; //1의 개수를 저장하는 변수
    for (int i = 0; i < minterm.size(); i++) {
        num = Number_one(minterm[i]);
        column[num].push_back(minterm[i]);
    } //1의 개수에 해당하는 행에 minterm을 저장

    qm.push_back(column);//qm리스트에 첫 열 push
    continue_combine(qm);//모든 열을 만들어서 qm리스트에 저장함
    for (it = qm.begin(); it != qm.end(); it++) {
        vector<vector<string>> temp = *it;
        prime_implicant(temp);//리스트에 모든 요소를 순환하며 모든 prime implicant를 prime 리스트에 저장
    }
    prime.unique();//중복된 prime implicant 제거
    Essential(minterm, num_min);
    list<string>::iterator iter3;
    for (iter3 = essential.begin(); iter3 != essential.end(); iter3++)
        fout << *iter3 << endl;
    fin.close();
    fout.close();
}
```

파일로부터 읽어오기 위해서 fin, 파일에 저장하기 위해서 fout 객체를 각각 선언했다.
파일로부터 읽어온 bit의 수를 저장하고 파일의 minterm들을 리스트에 저장한다.
minterm들이 저장될 2차원 벡터의 행을 bit의 크기+1로 미리 할당한 후, 1의 개수에 해당하는 행에 minterm을 저장한다. 이 2차원 벡터를 qm에 넣은 후, continue_combine 함수를 통해서 모든 cloumn을 만들어서 qm에 저장한다. 그렇게 모든 column이 qm에 저장되면 qm의 모든 요소를 돌면서 prime_implicant 함수를 통해서 모든 PI를 prime 리스트에 저장한다.
저장된 리스트에서 중복을 제거한 후, Essential 함수를 통해서 Essential PI만 essential 리스트에 저장시키고, fout 객체를 통해서 파일에 출력한 후 열었던 파일을 모두 닫는다.

<Number_one, next_col>

```
int Number_one(string temp) { // 1의 개수를 반환하는 함수
    int cnt = 0;
    for (int i = 0; i < temp.length(); i++) {
        if (temp[i] == '1')
            cnt++;
    }
    return cnt;
}

void next_col(vector<vector<string>>& column, vector<vector<string>>& next_column) {//Hamming distance가 1인 최소항끼리 결합하여 새로운 열을 구성하는 함수
    string temp; // 최소항끼리 결합하여 만든 implicant
    for (int i = 0; i < column.size() - 1; i++) {//행
        for (int j = 0; j < column[i].size(); j++) {//열
            if (column[i][j] == "")
                continue;//공백 무시
            for (int k = 0; k < column[i + 1].size(); k++) {
                int flag = 0;
                combine(column[i][j], column[i + 1][k], temp, flag);//한 그룹에 저장된 하나의 minterm과 다음 그룹의 모든 minterm을 비교
                if (flag == 1) {
                    next_column[i].push_back(temp);//결합이 된다면 다음 열에 저장
                }
            }
        }
    }
}

```

number_one 함수는 string을 인자로 받아서 해당 string의 인덱스를 순회하며 string의 인덱스가 1이라면 cnt를 증가시켜서 cnt를 반환하여 해당 string의 1의 개수를 반환하는 함수다.

next_col 함수는 인자로 2차원 string 벡터를 2개를 받는다. 첫 인자의 행과 열을 순회하며 한 그룹의 string에 접근한다. 여기서 string은 minterm이다. 이 minterm과 다음 그룹(1의 개수 차이가 하나인 그룹)의 첫 minterm과 결합 여부를 판단한다. 결합이 가능하다면 인자로 받아온 두 번째 인자인 2차원 string 벡터에 새롭게 결합한 항을 저장한다. 이런 방식으로 한 column의 모든 minterm을 비교하고 새로운 항을 만들어 다음 열을 구성하는 함수이다.

<combine>

```
void combine(string& s1, string& s2, string& newstr, int& flag) {//최소항끼리 결합하여 새로운 implicant를 만드는 함수
    int len = min(s1.length(), s2.length());//더 작은 길이 저장
    int index = 0; // 숫자가 다른 비트의 인덱스를 저장하는 변수
    for (int i = 0; i < len; i++) {
        if (s1[i] == s2[i]) {//같으면 건너뜀
            continue;
        }
        else {
            index = i;//다르면 해당 인덱스를 저장
        }
    }
    if (Hamming_dis(s1, s2) != 1) {//Hamming distance가 1이 아니면 결합되지 않음
        flag = 0;
        return;
    }
    else {//Hamming distance가 1이면
        newstr = s1;//s1을 저장
        newstr[index] = '_';//저장한 인덱스를 _로 바꿈
        flag = 1;//flag를 1로 초기화
    }
}
```

combine 함수는 인자로 받은 두 개의 string(minterm)이 결합이 가능한지 판단하고 결합이 가능하다면 새로운 항을 만들고 결합 성공 여부를 알려주는 함수이다.

두 string중, 길이가 짧은 stirng의 크기만큼 두 string의 인덱스를 순회한다. 두 인덱스가 같으면 다음 반복으로 넘어가고 다르다면 해당 index를 저장한다. 두 string의 Hamming distance가 1이 아니라면 결합이 불가능하므로 flag에 0을 저장한 후 함수를 종료한다. 이때 flag는 참조자이므로 함수를 호출 시에 전달되는 flag도 0으로 바뀐다. 1이라면 결합이 가능하므로 newstr에 s1을 저장하고 값이 다른 인덱스를 _로 바꾼다. newstr도 마찬가지로 참조자이므로 함수를 호출 시에 전달되는 newstr도 결합된 항으로 바뀐다.

<Hamming_dis>

```
109     int Hamming_dis(string s1, string s2) {//두 최소항의 Hamming distance를 반환하는 함수
110         int len = min(s1.length(), s2.length());
111         int cnt = 0;//Hamming distance를 저장할 변수
112         for (int i = 0; i < len; i++) {
113             if (s1[i] == s2[i]) {
114                 continue;//같으면 건너뜀
115             }
116             else {
117                 cnt++;//다르면 Hamming distance 증가
118             }
119         }
120         return cnt;
121     }
```

두 string의 Hamming distance를 반환하는 함수이다.
두 string 중 짧은 string의 길이만큼 각 string의 인덱스를 순회하며 두 string이 같으면 다음 반복으로 넘어가고 다르다면 cnt를 증가시켜 해당 cnt를 반환하는 방식으로 구현했다.

<prime_implicant>

```
122     void prime_implicant(vector<vector<string>>& column) {//한 행에서 prime implicant를 뽑고 결합여부를 체크하는 함수
123         for (int i = 0; i < column.size() - 1; i++) {
124             for (int j = 0; j < column[i].size(); j++) {
125                 if (column[i][j] == "")
126                     continue;
127                 for (int k = 0; k < column[i + 1].size(); k++) {
128                     int n = 0;
129                     n = Hamming_dis(column[i][j], column[i + 1][k]);//한 그룹의 minterm과 다음 그룹의 모든 minterm을 비교
130                     if (n == 1) {//Hamming distance가 1이라면 결합이 가능하므로
131                         if (column[i][j].back() != '*')
132                             column[i][j] += '*';//*을 붙혀서 체크
133                         if (column[i + 1][k].back() != '*')
134                             column[i + 1][k] += '*';//*을 붙혀서 체크
135                     }
136                 }
137             }
138         }
139         for (int i = 0; i < column.size() - 1; i++) {
140             for (int j = 0; j < column[i].size(); j++) {
141                 if (column[i][j] != "" && column[i][j].back() != '*')//*이 붙어있지 않으면 PI이므로
142                     prime.push_back(column[i][j]);//PI를 prime 리스트에 push
143             }
144         }
145     }
146
```

인자로 전달된 2차원 string 벡터(column)에 저장된 minterm 들로부터 PI를 따로 저장하고 minterm의 결합 여부를 체크하는 함수이다.
2차원 벡터의 행과 열을 순회하며 한 그룹의 minterm과 다음 그룹의 minterm의 Hamming distance를 반환한다. 1이라면 두 minterm은 결합이 가능하므로 해당 minterm들에 *을 붙여서 표시한다. 이미 *이 붙어있다면 생략한다. 이 과정을 반복하여 한 column에서 결합 가능한 모든 minterm에 *을 붙인다. 이후에 다시 행과 열을 반복하며 back()을 사용하여 마지막에 *이 붙어있는지 확인한 후, 붙어있지 않다면 PI로 채택하여 PI만 저장하는 prime 리스트에 string을 push한다.

<continue_combine>

```
void continue_combine(list < vector<vector<string>>>& qm) {//결합이 안될때까지 반복하여 모든 열을 만드는 함수
    vector<vector<string>> cur_col = qm.back();//마지막 열을 저장
    vector<vector<string>> next_column(bitLen + 1);//다음 열의 크기를 미리 지정
    next_col(cur_col, next_column);//next_column에 다음 열을 생성
    int k = 0;
    int flag = 0;
    while (k < bitLen + 1) {
        if (!next_column[k].empty())//각 열의 그룹이 비어있지 않다면 flag를 1로 초기화
            flag = 1;
        k++;
    }
    if (flag == 1) {//flag가 1이면 다음 2차원 벡터(열)는 비어있지 않으므로 리스트에 추가 후 다음 결합 실행
        qm.push_back(next_column);//저장된 열을 qm리스트에 push
        continue_combine(qm);//다시 함수를 호출하여 반복
    }
    else {
        return;//flag가 1이 아니라면 다음열이 비어있으므로 종료
    }
}
```

인자로 2차원 string이 저장된 qm을 전달받아서 qm의 마지막 요소인 column을 통해서 다음 column을 생성한다. 다음 column이 비어있지 않다면 해당 column을 PI table을 의미하는 qm에 push하고 다시 함수를 호출하여 다음 column을 생성하는 작업을 반복한다. 다음 column이 비어있다면 결합이 끝났으므로 함수를 종료한다.

<Essential>

```
void Essential(vector<string> minterm, int num) {//prime implicant에서 Essential만 뽑아내는 함수
    int row = prime.size() + 1;//PI의 갯수+1
    int col = num + 1;//minterm의 갯수+1
    int* arr = new int[col + 1];//Essential이 아닌 PI가 포함하는 minterm의 갯수를 저장하는 배열
    vector<vector<string>> chart;//주항차트를 구성하는 2차원 벡터
    vector<string>::iterator iter1 = minterm.begin();
    list<string>::iterator iter2 = prime.begin();
    chart.assign(row + 1, vector<string>(col + 1, ""));//row+1, col+1로 행과열의 크기를 할당
    chart[0][0] = "X";
    for (int i = 0; i < col - 1; i++) {//0행에 minterm들 저장, 0,0은 비워둠
        chart[0][i + 1] = *iter1++;
    }
    for (int i = 0; i < row - 1; i++) {//0열에 PI들 저장, 0,0은 비워둠
        chart[i + 1][0] = *iter2++;
    }
    for (int i = 1; i < row; i++) {//PI
        for (int j = 1; j < col; j++) {//minterm
            int flag = 0;
            for (int k = 0; k < bitLen; k++) {//각 인덱스
                if (chart[i][0][k] == '_') {//i행의 0열인 PI의 k번째 문자가 _이면
                    continue;
                }
                else {//0이나 1일 때
                    if (chart[i][0][k] != chart[0][j][k]) {//k번째 문자가 다르다면
                        flag++;
                    }
                }
            }
            if (flag == 0) {//PI가 minterm을 포함하면
                chart[i][j] = "*";// 해당 PI가 있는 열에 *로 체크
            }
        }
    }
```

```
199     for (int i = 1; i < col; i++) {//포함하는 minterm이 1개인 Essenetial PI만 뽑아서 리스트에 저장
200         int cnt = 0;//포함하는 minterm의 갯수
201         int idx_row;
202         int idx_col;
203         for (int j = 1; j < row; j++) {
204             if (chart[j][i] == "*") {//한 열씩 비교하므로 j,i로 비교
205                 cnt++;
206                 idx_row = j;//해당 행 저장
207                 idx_col = i;//해당 열 저장
208             }
209         }
210         if (cnt == 1&& chart[idx_row][0].back() != '*') {//한 PI가 포함하는 minterm의 갯수가 1개면 Essential PI, 이미 체크된 PI가 아닐 때
211             essential.push_back(chart[idx_row][0]);//해당 행의 0열인 PI가 Essential이 됨
212             chart[idx_row][0] += "*";//해당 행의 PI는 Essential로 뺌
213             chart[0][idx_col] += "*";//해당 열의 minterm은 더이상 고려 X
214         }
215     }
216     for (int i = 1; i < row; i++) {
217         if (chart[i][0].back() == '*') {//Essential PI로 뽑은 항들만 체크
218             for (int j = 1; j < col; j++) {
219                 if (chart[i][j] == "*" && chart[0][j].back() != '*') {//체크된 부분의 minterm이 체크되어있지 않다면 체크
220                     chart[0][j] += "*";
221                 }
222             }
223         }
224     }
225     }
226     while (1) {//모든 minterm을 포함하는 Essential PI가 나올때까지 반복
227         bool flag = false;
228         for (int i = 1; i < col; i++) {
229             if (chart[0][i].back() != '*') {//하나라도 체크가 안된 minterm이 있다면 flag를 true로 바꿈
230                 flag = true;
231             }
232         }
233         if (!flag) {//flag가 false라면 모든 minterm이 체크되어있으므로 반복종료
234             delete[] arr;
235             return;
236         }
237         for (int i = 1; i < row; i++) {
238             if (chart[i][0].back() != '*') {//체크가 안된 PI들이 포함시키는 minterm의 갯수를 저장
239                 for (int j = 1; j < col; j++) {
240                     if (chart[i][j] == "*")
241                         arr[i]++;
242                 }
243             }
244         }
245         int max = 0;
246         int idx_PI = 0;//최대로 많은 minterm을 저장하는 PI의 행(chart에서)을 저장할 변수
247         for (int i = 1; i <= col; i++) {
248             if (arr[i] == 0)
249                 continue;
250             if (max <= arr[i]) {
251                 idx_PI = i;//포함하는 minterm이 가장 많은 인덱스를 저장
252             }
253         }
254         essential.push_back(chart[idx_PI][0]);//해당 PI를 Essential에 넣어줌
255         chart[idx_PI][0] += "*";//해당 PI를 체크
256         for (int i = 1; i < col; i++) {
257             if (chart[idx_PI][i] == "*") {//Essential로 포함한 PI에 해당하는 minterm을 체크
258                 chart[0][i] += "*";
259             }
260         }
261     }
262 }
263
```

Essential 함수는 qm 리스트의 PI table을 통해서 주항차트를 구성하고, 주항차트를 토대로 Essential PI만 뽑아내는 함수이다.

1. 주항차트를 구성하는 2차원 벡터를 chart를 선언하고 행은 PI의 개수+2, 열은 minterm의 개수+2의 크기로 미리 할당 해두었다. chart에 0행의 1열부터 minterm들을 저장하고 chart에 1행의 0열부터 PI들을 저장했다. for문을 통해서 한 PI의 인덱스와 minterm의 인덱스를 비교하면서 PI의 인덱스가_라면 다음 반복, 0이나 1이라면 flag를 증가한다. 이렇게 PI와 한 minterm을 비교하고 난 후 flag가 0이라면 PI가 해당 minterm을 포함한다. 따라서

주항차트의 해당 부분에 *로 체크한다. 이런 방식을 통해서 모든 주항차트를 구성한다.
2. for(i는 1부터 열의 끝까지)
for(j= 1부터 행의 끝까지) 와 같은 방식의 2중 반복문을 사용하여 chart[j][i]로 접근하면 한 열씩 비교할 수 있다. 한 열씩 비교하며 표시의 개수를 세고 해당 행(idx_row)과 열(idx_col)을 저장한다. minterm을 커버하는 PI가 하나뿐이고(=한 열에 표시가 하나라면) PI가 Essential로 채택된 PI가 아니라면(=PI가 표시 되어있지 않다면) 해당 PI는 Essential PI로 채택하므로 essential 리스트에 PI를 push, chart[idx_row][0]+="*"을 통해서 PI에 표시, chart[0][idx_col]+="*"을 통해서 minterm에 표시한다.
이렇게 커버하는 minterm이 하나인 PI들을 Essential로 저장하고 해당 PI가 커버하는 다른 minterm들 또한 *을 붙여서 표시한다.
3. 이렇게 Essential PI를 추출한 후, chart[0][i], I는 1부터 minterm의 개수만큼 반복하며 주항차트의 0행의 minterm들을 순회하며 chart[0][i].back()을 이용하여 *이 아니라면 +="*"을 이용하여 flag를 true로 바꾼다. flag가 false라면 모든 minterm이 커버되어 있으므로 함수를 종료한다. 하나라도 커버되지 않은 minterm이 있다면 Essential로 포함되지 않은 PI (표시되지 않은 PI)들이 포함하는 minterm의 개수를 카운트한다. 가장 많이 카운트된 PI를 essential PI로 채택하고 해당 PI에 *을 붙여서 체크, for(1부터 minterm의 크기만큼 반복)문을 사용하여 해당 PI가 커버하는 minterm에 *을 붙여 체크한다. 이 부분은 while(1)에 속해있도록 구현하여 모든 minterm이 커버 되었는지 체크, step3 과정을 모든 minterm이 커버될 때 까지 반복하여 Essential PI를 추출한다.

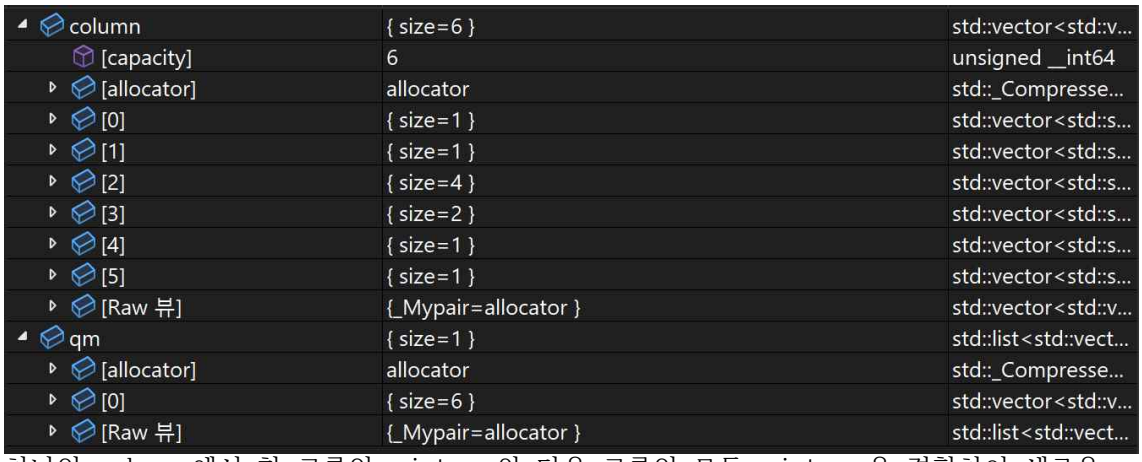
C- ii . corresponding examples with explanation

```
5
m 00000
m 00100
m 00101
m 00110
m 01001
m 01010
m 00111
m 01101
m 01111
m 11111
```

파일로부터 다음과 같은 입력을 받고 bit의 수를 저장, 리스트에 각 minterm을 넣는다.

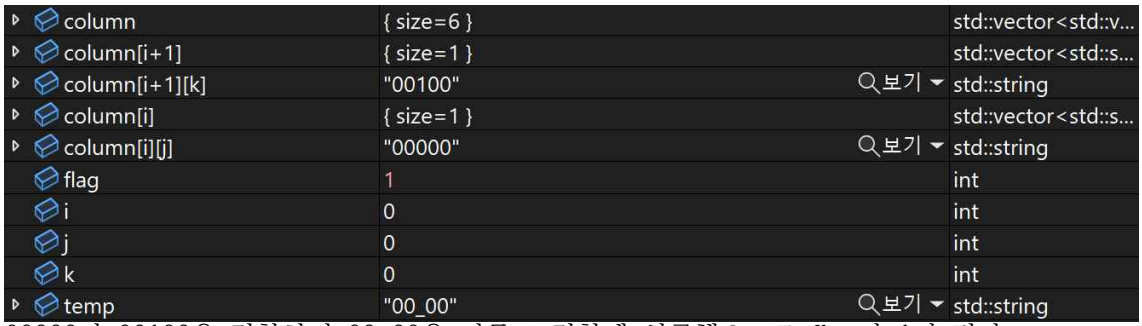
minterm	{ size=10 }	std::vector<std::s...
[capacity]	13	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	"00000" 보기	std::string
[1]	"00100" 보기	std::string
[2]	"00101" 보기	std::string
[3]	"00110" 보기	std::string
[4]	"01001" 보기	std::string
[5]	"01010" 보기	std::string
[6]	"00111" 보기	std::string
[7]	"01101" 보기	std::string
[8]	"01111" 보기	std::string
[9]	"11111" 보기	std::string

입력받은 minterm을 1의 개수대로 2차원 벡터인 cloumn에 할당하고 column을 qm 리스트에 push한다.



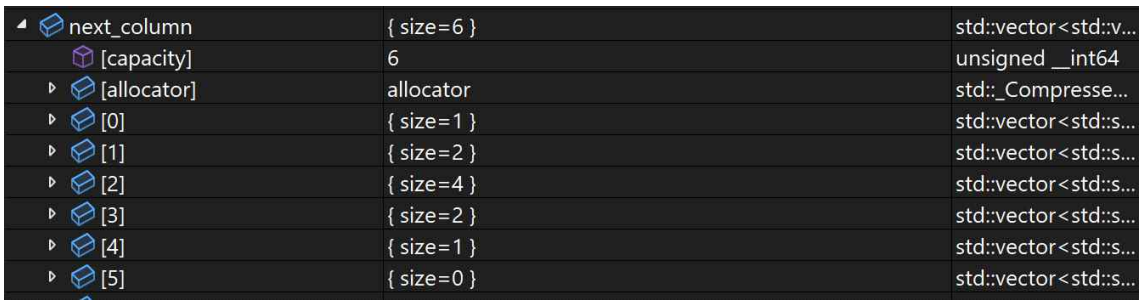
column	{ size=6 }	std::vector<std::v...
[capacity]	6	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	{ size=1 }	std::vector<std::s...
[1]	{ size=1 }	std::vector<std::s...
[2]	{ size=4 }	std::vector<std::s...
[3]	{ size=2 }	std::vector<std::s...
[4]	{ size=1 }	std::vector<std::s...
[5]	{ size=1 }	std::vector<std::s...
[Raw 뷰]	{_Mypair=allocator }	std::vector<std::v...
qm	{ size=1 }	std::list<std::vect...
[allocator]	allocator	std::_Compresse...
[0]	{ size=6 }	std::vector<std::v...
[Raw 뷰]	{_Mypair=allocator }	std::list<std::vect...

하나의 column에서 한 그룹의 minterm와 다음 그룹의 모든 minterm을 결합하여 새로운 행을 만든다.

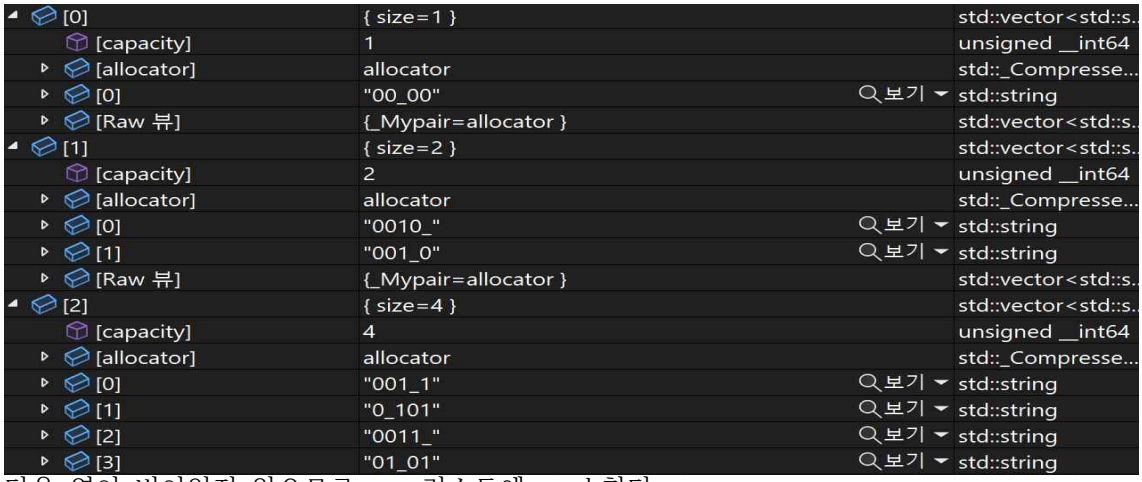


column	{ size=6 }	std::vector<std::v...
column[i+1]	{ size=1 }	std::vector<std::s...
column[i+1][k]	"00100" 보기	std::string
column[i]	{ size=1 }	std::vector<std::s...
column[i][j]	"00000" 보기	std::string
flag	1	int
i	0	int
j	0	int
k	0	int
temp	"00_00" 보기	std::string

00000과 00100을 결합하여 00_00을 만들고 결합에 성공했으므로 flag가 1이 된다.
이런 과정을 반복하여 다음 열을 구성한다.

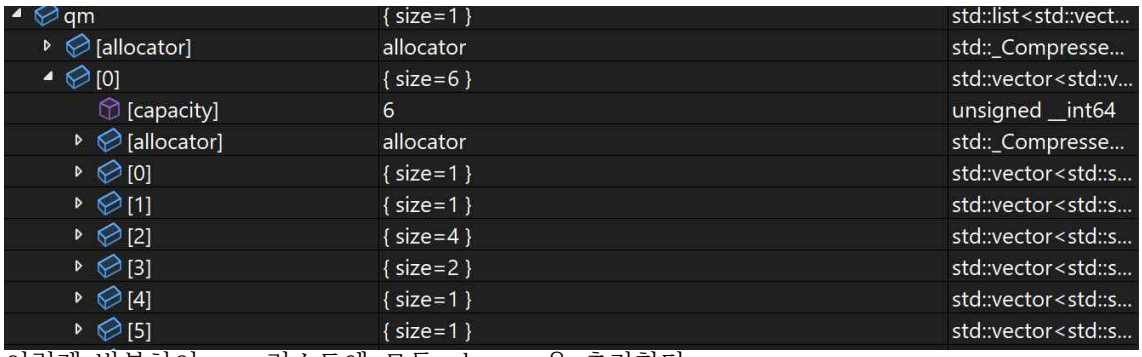


next_column	{ size=6 }	std::vector<std::v...
[capacity]	6	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	{ size=1 }	std::vector<std::s...
[1]	{ size=2 }	std::vector<std::s...
[2]	{ size=4 }	std::vector<std::s...
[3]	{ size=2 }	std::vector<std::s...
[4]	{ size=1 }	std::vector<std::s...
[5]	{ size=0 }	std::vector<std::s...



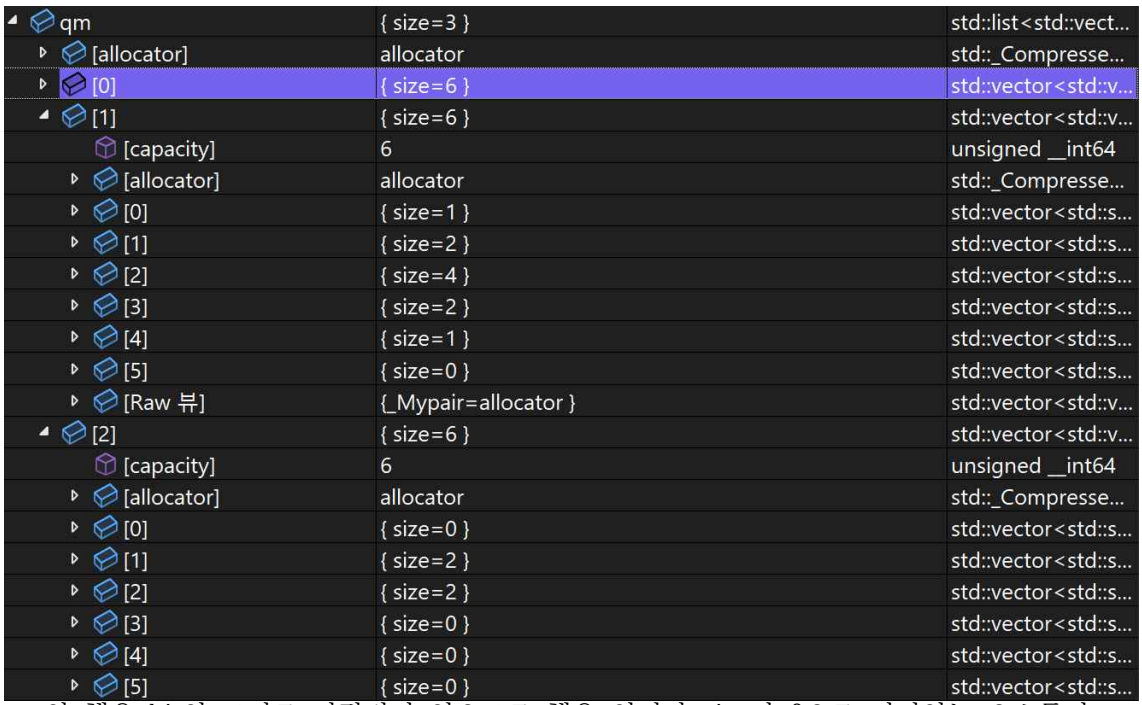
[0]	{ size=1 }	std::vector<std::s...
[capacity]	1	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	"00_00" 보기	std::string
[Raw 뷰]	{_Mypair=allocator }	std::vector<std::s...
[1]	{ size=2 }	std::vector<std::s...
[capacity]	2	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	"0010_" 보기	std::string
[1]	"001_0" 보기	std::string
[Raw 뷰]	{_Mypair=allocator }	std::vector<std::s...
[2]	{ size=4 }	std::vector<std::s...
[capacity]	4	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	"001_1" 보기	std::string
[1]	"0_101" 보기	std::string
[2]	"0011_" 보기	std::string
[3]	"01_01" 보기	std::string

다음 열이 비어있지 않으므로 qm 리스트에 push한다.



qm	{ size=1 }	std::list<std::vect...
[allocator]	allocator	std::_Compresse...
[0]	{ size=6 }	std::vector<std::v...
[capacity]	6	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	{ size=1 }	std::vector<std::s...
[1]	{ size=1 }	std::vector<std::s...
[2]	{ size=4 }	std::vector<std::s...
[3]	{ size=2 }	std::vector<std::s...
[4]	{ size=1 }	std::vector<std::s...
[5]	{ size=1 }	std::vector<std::s...

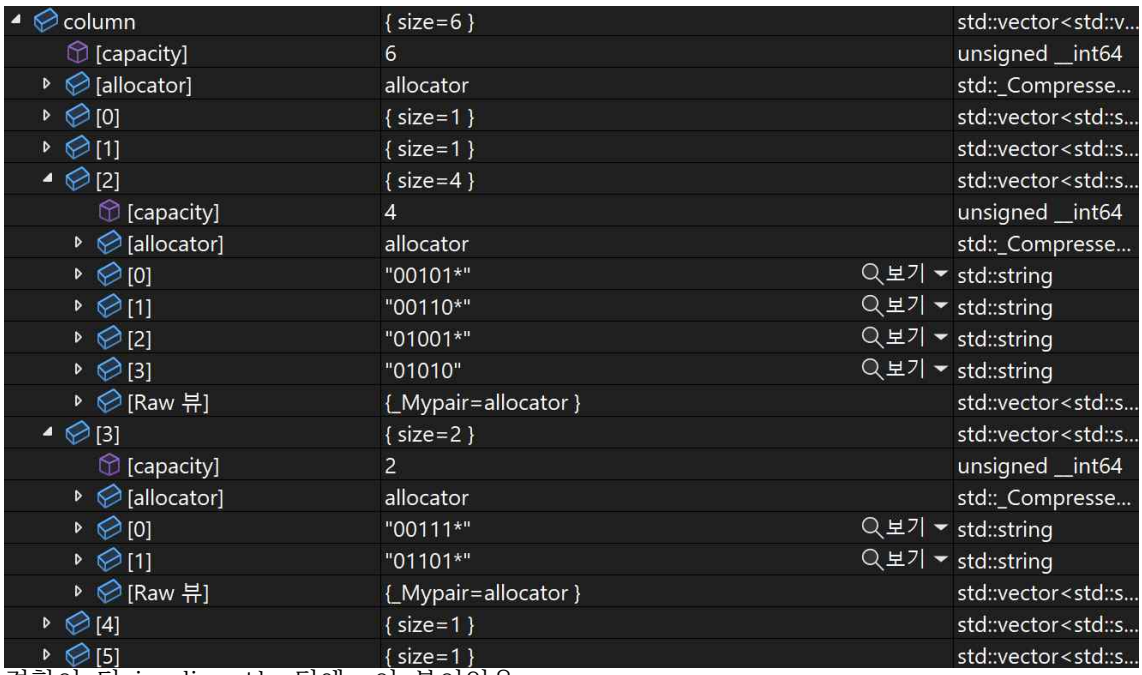
이렇게 반복하여 qm 리스트에 모든 cloumn을 추가한다.



qm	{ size=3 }	std::list<std::vect...
[allocator]	allocator	std::_Compresse...
[0]	{ size=6 }	std::vector<std::v...
[1]	{ size=6 }	std::vector<std::v...
[capacity]	6	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	{ size=1 }	std::vector<std::s...
[1]	{ size=2 }	std::vector<std::s...
[2]	{ size=4 }	std::vector<std::s...
[3]	{ size=2 }	std::vector<std::s...
[4]	{ size=1 }	std::vector<std::s...
[5]	{ size=0 }	std::vector<std::s...
[Raw 뷰]	{_Mypair=allocator }	std::vector<std::v...
[2]	{ size=6 }	std::vector<std::v...
[capacity]	6	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	{ size=0 }	std::vector<std::s...
[1]	{ size=2 }	std::vector<std::s...
[2]	{ size=2 }	std::vector<std::s...
[3]	{ size=0 }	std::vector<std::s...
[4]	{ size=0 }	std::vector<std::s...
[5]	{ size=0 }	std::vector<std::s...

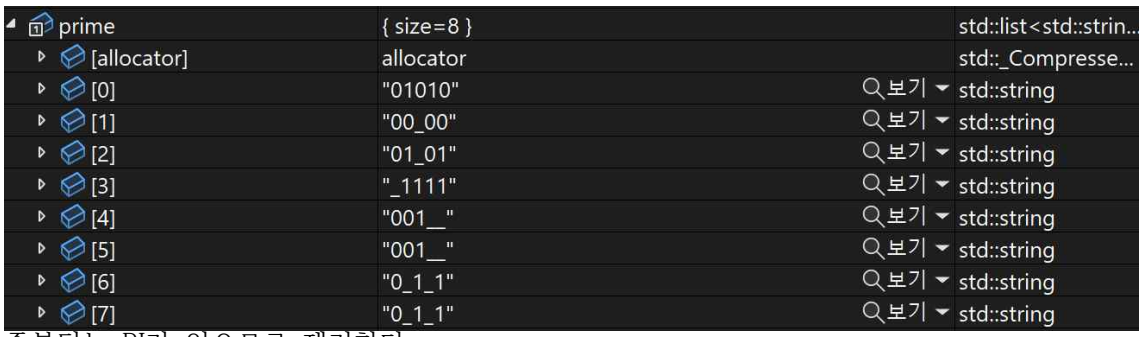
qm의 행은 bit의 크기로 지정되어 있으므로 행은 있지만 size가 0으로 비어있는 요소들이 있다. 따라서 다음 열이 비어있는지 판단할 때, 모든 행에 접근하여 empty()를 사용하여 판단해야한다.

이후 qm 리스트를 순회하면서 결합여부를 체크하고 PI를 뽑아낸다.



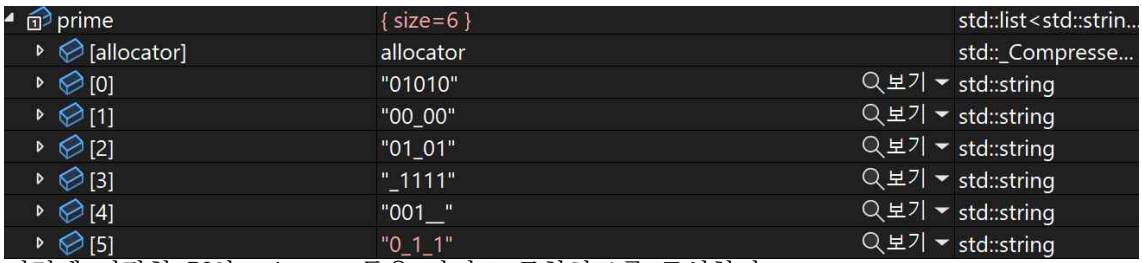
column	{ size=6 }	std::vector<std::v...
[capacity]	6	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	{ size=1 }	std::vector<std::s...
[1]	{ size=1 }	std::vector<std::s...
[2]	{ size=4 }	std::vector<std::s...
[capacity]	4	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	"00101*" 보기	std::string
[1]	"00110*" 보기	std::string
[2]	"01001*" 보기	std::string
[3]	"01010" 보기	std::string
[Raw 뷰]	{_Mypair=allocator }	std::vector<std::s...
[3]	{ size=2 }	std::vector<std::s...
[capacity]	2	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	"00111*" 보기	std::string
[1]	"01101*" 보기	std::string
[Raw 뷰]	{_Mypair=allocator }	std::vector<std::s...
[4]	{ size=1 }	std::vector<std::s...
[5]	{ size=1 }	std::vector<std::s...

결합이 된 implicant는 뒤에 *이 붙어있음



prime	{ size=8 }	std::list<std::strin...
[allocator]	allocator	std::_Compresse...
[0]	"01010" 보기	std::string
[1]	"00_00" 보기	std::string
[2]	"01_01" 보기	std::string
[3]	"_1111" 보기	std::string
[4]	"001__" 보기	std::string
[5]	"001__" 보기	std::string
[6]	"0_1_1" 보기	std::string
[7]	"0_1_1" 보기	std::string

중복되는 PI가 있으므로 제거한다.



prime	{ size=6 }	std::list<std::strin...
[allocator]	allocator	std::_Compresse...
[0]	"01010" 보기	std::string
[1]	"00_00" 보기	std::string
[2]	"01_01" 보기	std::string
[3]	"_1111" 보기	std::string
[4]	"001__" 보기	std::string
[5]	"0_1_1" 보기	std::string

이렇게 저장한 PI와 minterm들을 가지고 주항차트를 구성한다.

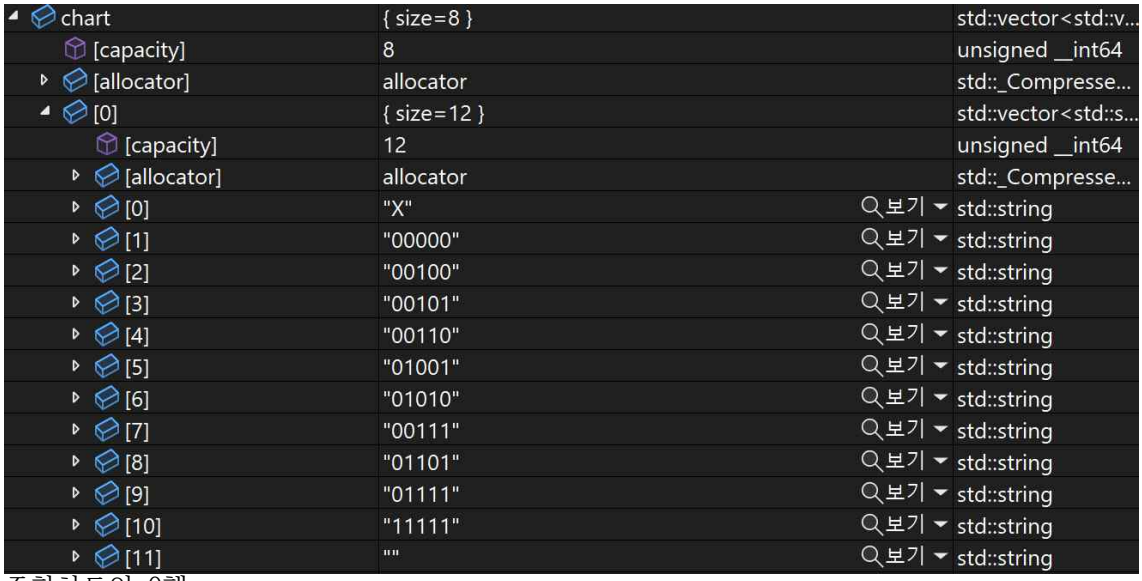
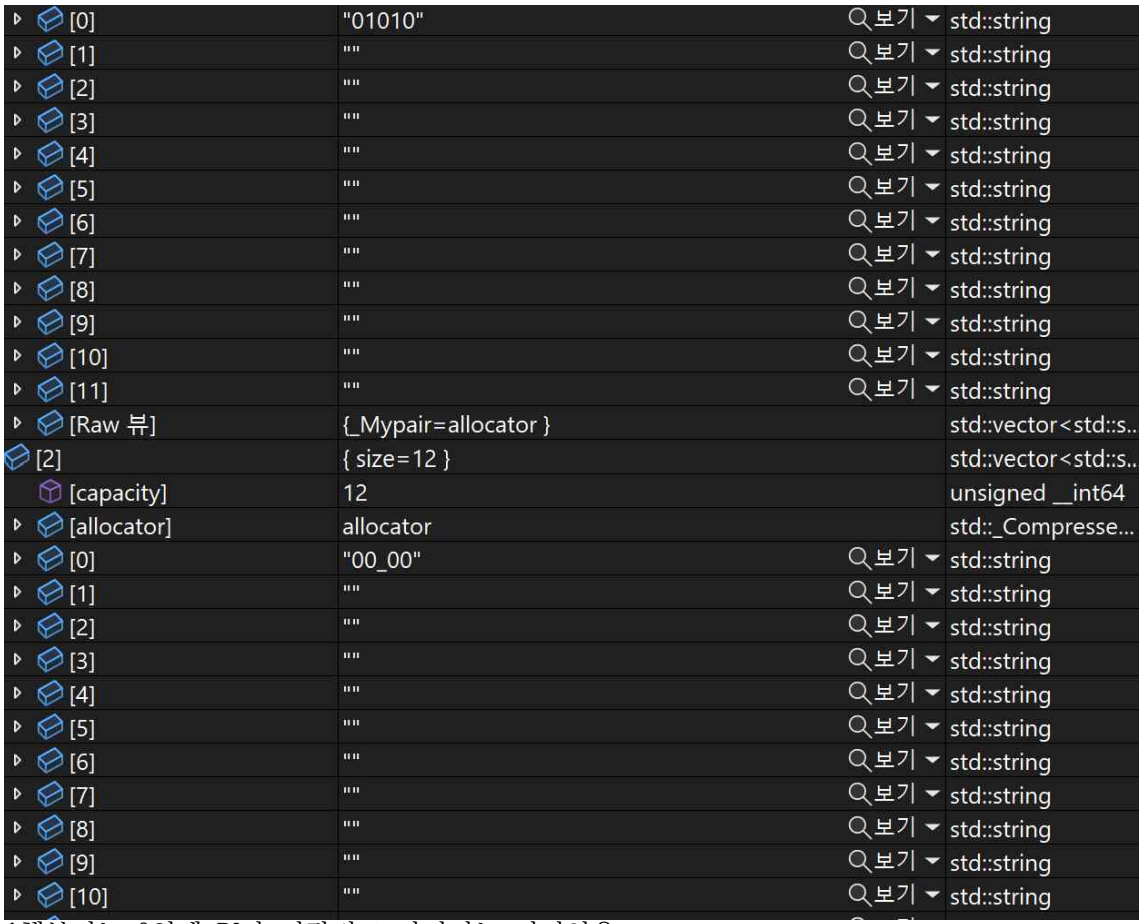


chart	{ size=8 }	std::vector<std::v...
[capacity]	8	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	{ size=12 }	std::vector<std::s...
[capacity]	12	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	"X" 보기	std::string
[1]	"00000" 보기	std::string
[2]	"00100" 보기	std::string
[3]	"00101" 보기	std::string
[4]	"00110" 보기	std::string
[5]	"01001" 보기	std::string
[6]	"01010" 보기	std::string
[7]	"00111" 보기	std::string
[8]	"01101" 보기	std::string
[9]	"01111" 보기	std::string
[10]	"11111" 보기	std::string
[11]	"" 보기	std::string

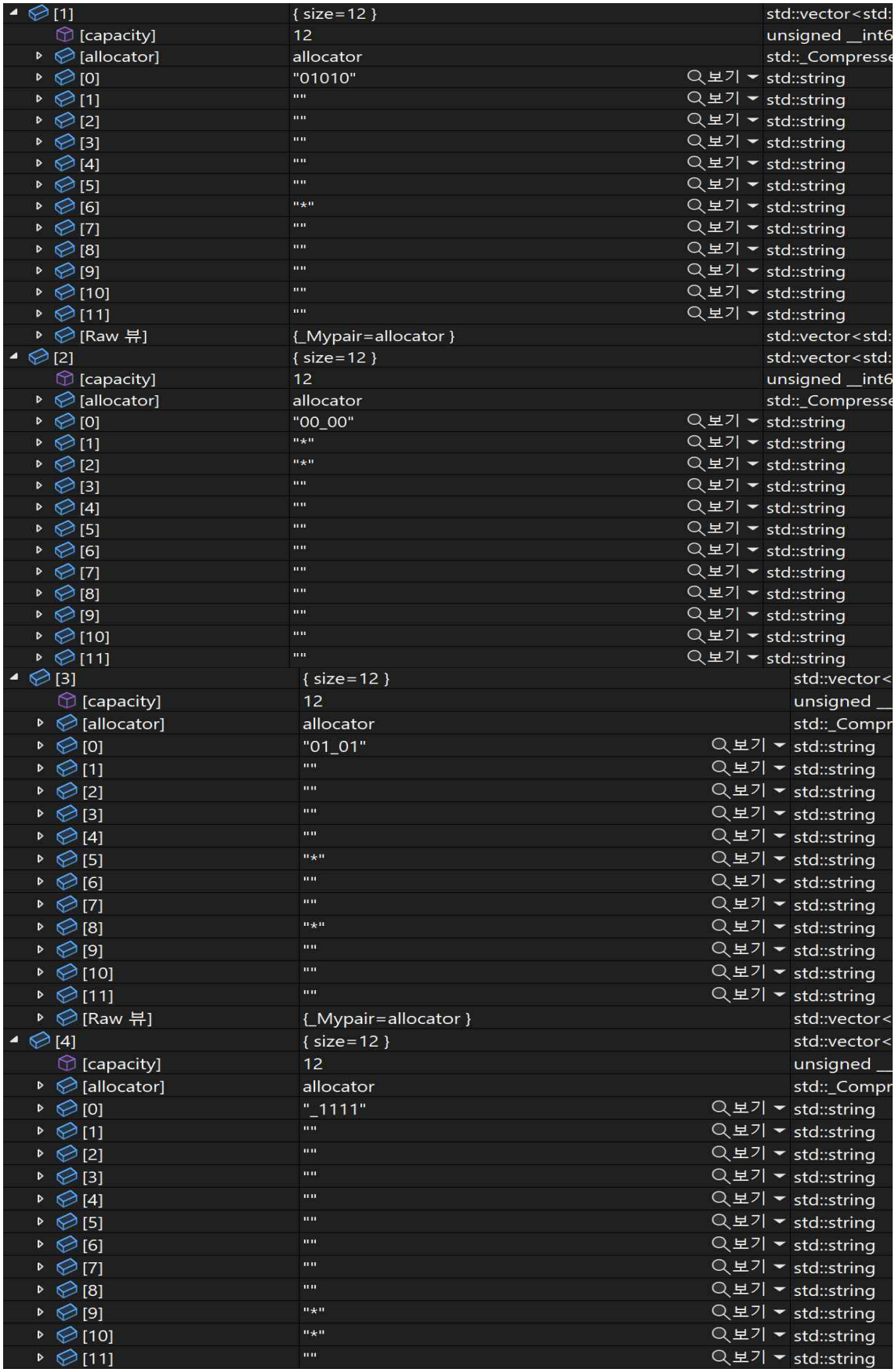
주항차트의 0행



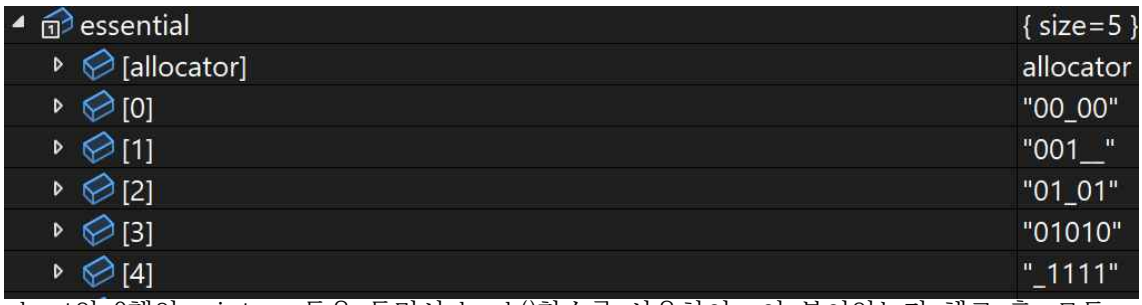
[0]	"01010" 보기	std::string
[1]	"" 보기	std::string
[2]	"" 보기	std::string
[3]	"" 보기	std::string
[4]	"" 보기	std::string
[5]	"" 보기	std::string
[6]	"" 보기	std::string
[7]	"" 보기	std::string
[8]	"" 보기	std::string
[9]	"" 보기	std::string
[10]	"" 보기	std::string
[11]	"" 보기	std::string
[Raw 뷰]	{_Mypair=allocator }	std::vector<std::s..
[2]	{ size=12 }	std::vector<std::s..
[capacity]	12	unsigned __int64
[allocator]	allocator	std::_Compresse...
[0]	"00_00" 보기	std::string
[1]	"" 보기	std::string
[2]	"" 보기	std::string
[3]	"" 보기	std::string
[4]	"" 보기	std::string
[5]	"" 보기	std::string
[6]	"" 보기	std::string
[7]	"" 보기	std::string
[8]	"" 보기	std::string
[9]	"" 보기	std::string
[10]	"" 보기	std::string

1행부터는 0열에 PI가 저장되고 나머지는 비어있음

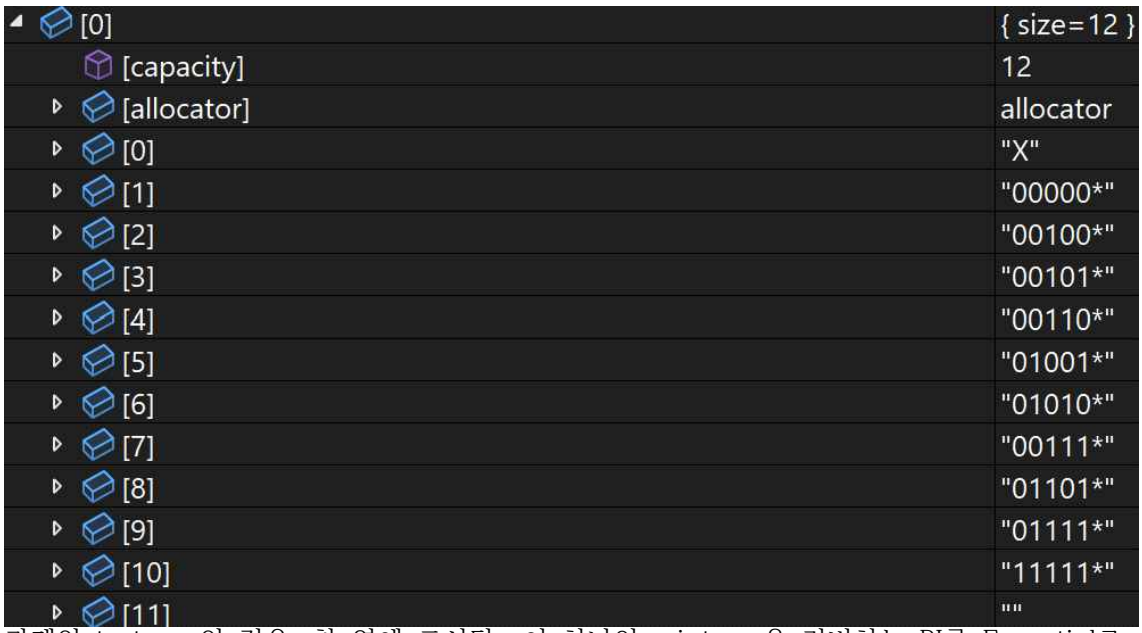
이제 PI가 커버할 수 있는 minterm이 있는 부분에 *로 표시한다.


[1] { size=12 } std::vector<std:
[capacity] 12 unsigned __int6
[allocator] allocator std::_Compresse
[0] "01010" 보기 std::string
[1] "" 보기 std::string
[2] "" 보기 std::string
[3] "" 보기 std::string
[4] "" 보기 std::string
[5] "" 보기 std::string
[6] "*" 보기 std::string
[7] "" 보기 std::string
[8] "" 보기 std::string
[9] "" 보기 std::string
[10] "" 보기 std::string
[11] "" 보기 std::string
[Raw 뷰] {_Mypair=allocator } std::vector<std:
[2] { size=12 } std::vector<std:
[capacity] 12 unsigned __int6
[allocator] allocator std::_Compresse
[0] "00_00" 보기 std::string
[1] "*" 보기 std::string
[2] "*" 보기 std::string
[3] "" 보기 std::string
[4] "" 보기 std::string
[5] "" 보기 std::string
[6] "" 보기 std::string
[7] "" 보기 std::string
[8] "" 보기 std::string
[9] "" 보기 std::string
[10] "" 보기 std::string
[11] "" 보기 std::string
[3] { size=12 } std::vector<
[capacity] 12 unsigned __
[allocator] allocator std::_Compr
[0] "01_01" 보기 std::string
[1] "" 보기 std::string
[2] "" 보기 std::string
[3] "" 보기 std::string
[4] "" 보기 std::string
[5] "*" 보기 std::string
[6] "" 보기 std::string
[7] "" 보기 std::string
[8] "*" 보기 std::string
[9] "" 보기 std::string
[10] "" 보기 std::string
[11] "" 보기 std::string
[Raw 뷰] {_Mypair=allocator } std::vector<
[4] { size=12 } std::vector<
[capacity] 12 unsigned __
[allocator] allocator std::_Compr
[0] "_1111" 보기 std::string
[1] "" 보기 std::string
[2] "" 보기 std::string
[3] "" 보기 std::string
[4] "" 보기 std::string
[5] "" 보기 std::string
[6] "" 보기 std::string
[7] "" 보기 std::string
[8] "" 보기 std::string
[9] "*" 보기 std::string
[10] "*" 보기 std::string
[11] "" 보기 std::string

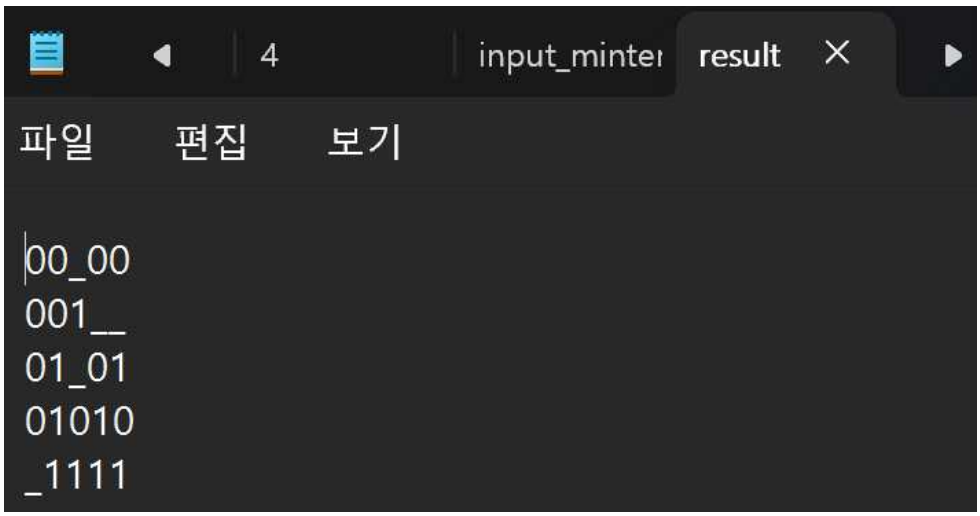
한 열에 표시된 *이 하나라면 해당 *을 가지는 PI는 Essential로 처리한다.



chart의 0행인 minterm들을 돌면서 back()함수를 사용하여 *이 붙어있는지 체크 후, 모두 붙어있다면 함수를 종료, 아니라면 가장 많은 minterm을 포함하는 PI를 Essential로 채택, PI와 해당 PI가 커버하는 minterm에 *로 표시 후 다시 minterm을 확인하는 과정을 반복하여 모두 체크될 때까지 반복하게 된다.



과제의 testcase의 경우, 한 열에 표시된 *이 하나인 minterm을 커버하는 PI를 Essential로 채택했을 때, 모든 minterm을 포함하므로 함수가 종료되고 Essential PI를 파일에 출력하게 된다.



D. A testcase that you think it is very hard to solve

```
5
m 00000
m 00001
m 00010
m 00011
m 00100
m 00101
m 00110
m 00111
m 01000
m 01001
m 01010
m 01011
m 01100
m 01101
m 01110
m 01111
m 10000
m 10001
m 10010
m 10011
m 10100
m 10101
m 10110
m 10111
m 11000
m 11001
m 11010
m 11011
m 11100
m 11101
m 11110
m 11111
```

해당 testcase는 5비트 2진수의 모든 입력을 minterm을 가진다. 따라서 모든 항이 결합되고 남은 output은 _____이 된다.

Output