

디지털논리로1 Project Report

Project 제목: Quine-McCluskey

제출기한:

2023년 04월 07일 (금)

~

2023년 04월 22일 (토)

학 과: 컴퓨터정보공학부

담당교수: 유지현 교수님

실습분반: 화,목 1,2교시

학 번: 2018202065

성 명: 박 철 준

● Problem statement

■ 제목

Quine-McCluskey

■ 사전 지식

콰인-맥클러스키 알고리즘(Quine-McCluskey algorithm)은 논리식을 최소화하는 알고리즘이다. 내부적으로는 카노 맵과 동일하지만, 그림을 그려서 맞추는 카노 맵과 달리 표를 사용하기 때문에 컴퓨터에서 쉽게 돌릴 수 있다. 또한 논리함수의 최소 형태를 결정론적으로 구할 수 있다.

알고리즘은 다음 두 단계로 구성된다.

주어진 함수의 후보항(Prime Implicants)들을 모두 구한다.

후보항들을 이용해서 후보항 표에서 필수항(Essential Prime Implicant)을 구한다.

■ 프로젝트 목적

퀸 맥클러스키 방법을 활용하여 prime implicant와 essential prime implicant를 찾고 논리식을 간소화하여 SOP를 찾고 그 SOP의 최소 cost를 찾는 프로그램을 만든다.

■ 프로젝트 기능

1. QM알고리즘을 사용하여 주항(Prime Implicants)을 찾는다. 방법은 테이블을 만들어 row는 true minterm과 don't care minterm을 나열하고 column은 단계별로 나누어 minterm들 중 hamming distance가 1인 minterm들을 찾아 다른 부분을 '-'로 대체하여 다음 단계로 넘긴다. 조합이 된 minterm들은 pls 집합에서 제외하고 조합이 되지 못해 체크가 되지 못한 minterm은 pls(prime implicants)가 된다. 해당 과정을 column1, column2 등 단계별로 수행하고 조합이 되지 않는 column 단계를 발견하였을 시 이를 끝낸다.
2. 주항(pls)들을 가지고 필수 주항(essential prime implicant 줄여서 epls)를 찾는다. 방법은 테이블을 구성하는데 row는 pls column은 true minterm들로 구성하여 pls가 커버하는 true minterm을 찾는다. true minterm 중 단 한개의 pl과 만족하는 pl은 필수 주항이 되며 여러개의 true minterm을 만족하는 pl의 경우 그 조합을 잘 맞추어 가장 적은 pls를 이루는 경우가 필수 주항이 된다.
3. 이후 찾은 필수 주항을 SOP로 구성하고 그에 따른 사용한 transistor 수를 구하는데 이때 가장 적은 cost를 구한다.

■ 프로그램 예외처리

1. input_minterm.txt 파일을 열지 못한 경우. 오류 메시지를 출력하고 프로그램을 종료한다.
2. input_minterm.txt에서 input의 개수를 입력하지 않은 경우. 오류 메시지를 출력하고 프로그램을 종료한다.
3. result.txt 파일을 열지 못한 경우. 오류 메시지를 출력하고 프로그램을 종료한다.

4. 예를 들어 input의 개수가 4개이면 d XXXX혹은 m XXXX의 형식을 만족해야 하는데 만족하지 못한 경우 오류 메시지를 출력하고 프로그램을 종료한다.
5. Ture minterm을 받지 않은 경우 오류 메시지를 출력하고 프로그램을 종료한다.

■ 프로그램 사용방법

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

input_minterm.txt를 생성하여 첫줄에는 bit_num의 개수 이후 true minterm 과 don't care minterm을 그림과 같이 입력한다. 이후 실행하면 result.txt를 열어보면 간소화된 SOP와 Cost를 알 수 있다.

● Pseudo code and Flow chart

■ Pseudo code

▶ main 함수

```
int main()
{
    char *InputLine[1000]; //파일에서 인풋 민텀을 입력받을 문자열을 저장할수있는 char형 포인터 배열을 선언합니다.
    for (int i = 0; i < 1000; i++)
    {
        InputLine[i] = new char[10000]; //문자열을 받기위해 동적할당합니다.
    }
    char *Tureminterm[1000]; //돈케어 민텀을 제외한 트루 민텀을받기위한 char형 포인터 배열을 선언합니다.
    for (int i = 0; i < 1000; i++)
    {
        Tureminterm[i] = new char[10000]; //문자열을 받기위해 동적할당합니다.
    }
    char count[1000];
    ifstream fin; //파일입력 클래스인 fin을 선언합니다.
    fin.open("input_minterm.txt"); //open input_minterm.txt file
    if (!fin.is_open())
    {
        cout << "파일을 열지 못하였습니다." << endl; //파일을 여는 것을 실패한경우 실행되는 예외처리 입니다.
        for (int i = 0; i < 1000; i++)
        {
            delete[] InputLine[i]; //해제
            delete[] Tureminterm[i]; //해제
        }
        return 0;
    }
    int i = 0;
    int input = 0;
    while (!fin.eof()) //파일에서 파일의 끝까지 한행씩 입력받아 char 형 포인터 배열 InputLine에 저장합니다.
    {
        fin.getLine(InputLine[i], 10000);
        *InputLine[i + 1] = 0;
        i++;
    }

    Input = atoi(InputLine[0]); //char 형 포인터 배열 InputLine의 첫번째 행은 인풋의 개수 입니다.

    if (Input == -48)
    {
        cout << "input의 갯수를 입력하지 않으셨습니다." << endl; //첫번째행이 비어있어 인풋의 개수를 알 수 없을 때 실행되는 예외처리입니다.
        for (int i = 0; i < 1000; i++)
        {
            delete[] InputLine[i]; //해제
            delete[] Tureminterm[i]; //해제
        }
        return 0;
    }

    if (InputLine[i][input + 2] != '\0')
    {
        cout << "입력파일의 형식이 올바르지 않습니다. 수정하십시오" << endl;
        for (int i = 0; i < 1000; i++)
        {
            delete[] InputLine[i]; //해제
            delete[] Tureminterm[i]; //해제
        }
        return 0;
    }

    int v = 0;
    for (int i = 0; *InputLine[i] != 0; i++) // InputLine의 [0]의 다음행들은 돈케어나 트루 민텀이고 입력받은 문자열은 예를 들면 d 0000의 형식을 가지는 데 계산의 편의를 위해 d혹은a과 띄어쓰기 공백하나를 지웁니다.*/
    {
        if (InputLine[i][0] == 'a')
        {
            for (int j = 0; j <= input + 1; j++)
            {
                Tureminterm[v][j] = InputLine[i][j + 2];
                *Tureminterm[v + 1] = 0;
            }
            v++;
        }
    }
}
```

```

int j = 1;
int termnumber = 0;
while (InputLine[j][0] == 'd' || InputLine[j][0] == 'n')/* 마찬가지로 true인항을 저장한 TrueMinterm에서 영어 d와 먹여쓰기 공백하나를 지웁니다.
{
    for (int i = 0; i <= Input + 1; i++)
    {
        InputLine[j][i] = InputLine[j][i + 2];
    }
    termnumber = j;
    j++;
}

char * InputMinterm[1000];/*InputLine 배열의 첫번째항은 인풋의 개수입니다. 그래서 계산상 편의를 위해 인풋의 갯수를 제외하여 나머지 문자열을 저장할 배열을 다시선언합니다.
for (int i = 0; i < 1000; i++)
{
    InputMinterm[i] = new char[Input + 1];/*동적할당
}
for (int i = 0; i < termnumber; i++)
{
    for (int j = 0; j < Input + 1; j++)
    {
        /*InputLine의 첫번째 항(인풋의 개수)을 제외한 모든 트루미텀 혹은 돈케어 인텀을 InputMinterm에 저장합니다.*/
        InputMinterm[i][j] = InputLine[i + 1][j];
        *InputMinterm[i + 1] = 0;
    }
}
if (v == 0)
{
    cout << "TrueMinterm이 존재하지 않습니다." << endl;
    for (int i = 0; i < 1000; i++)
    {
        delete[] InputLine[i];/*해제
        delete[] TrueMinterm[i];/*해제
        delete[] InputMinterm[i];/*해제
    }
    return 0;
}
}

```

```

FindPrimeImplicant(InputMinterm, Input, TrueMinterm);/*칸터클러스터 알고리즘에서 prime implicant를 찾는 함수이며 인자로는 InputMinterm 배열과, Input의 개수, TrueMinterm 배열을 받는다.
for (int i = 0; i < 1000; i++)
{
    delete[] InputLine[i];/*해제
}

return 0;
}

```

► FindPrimeImplicant 함수

void FindPrimeImplicant(char **InputMinterms, int Input, char **TrueMinterms) // 중복을 없애서 prime implicant을 찾는 함수이며, 인자로써 InputMinterms 배열과, Input의 개수, TrueMinterms 배열을 받는다.

```
int teranumber = 0; //함수를 연속적으로 실행시킬 것인데 그때마다 implicant의 합인 개수를 세어주는 변수이다.
for (int i = 0; *InputMinterms[i] != '\0'; i++)
{
    /*InputMinterms[i] != '\0' 일때까지 teranumber의 수를 증가시켜준다. */
    teranumber = i + 1;
}

char * prr[1000]; // InputMinterms를 저장하기 위한 char 포인터 배열 prr을 선언한다.
for (int i = 0; i < 1000; i++)
{
    prr[i] = new char[Input + 1]; //동적할당

    for (int i = 0; *InputMinterms[i] != '\0'; i++) //배열 prr에 InputMinterms를 저장한다.
    {
        strcpy(prr[i], InputMinterms[i]);
        *prr[i + 1] = 0;
    }

    char * crr[1000]; //해밀디스턴스가 1차이하는 문자열을 비교하여 디스턴스가 1차이하는 부분을 지워서 저장하기위한 배열을 선언한다.
    for (int i = 0; i < 1000; i++)
    {
        crr[i] = new char[Input + 1]; //동적할당
    }

    char * rest[1000]; //해밀디스턴스가 1차이하는 문자열이 없을 때 나오는 트라이 임플리칸트와 해밀디스턴스가 1차이하는 부분을 지운 문자열을 저장하기위한 배열을 선언한다.
    for (int i = 0; i < 1000; i++)
    {
        rest[i] = new char[Input + 1];
    }

    int c = 0;
    int L = 0;
    int s = 0;
}
```

```
for (int i = 0; i < teranumber; i++)
{
    int x = 0;
    for (int j = 0; j < teranumber; j++)
    {
        int a = 0;
        int u = 0;
        for (int k = 0; k < Input; k++)
        {
            if (InputMinterms[i][k] != InputMinterms[j][k])
            {
                a++; //해밀디스턴스차이를 확인하고 위한 for문을 선언합니다.
                //해밀디스턴스차이를 저장합니다.
                u = k; //바이너리 넘버가 다른 위치를 파악하기위한 변수입니다.
            }

            if (a == 1) //해밀디스턴스가 1인경우에 아래를 실행합니다.
            {
                int y = 1;
                s++;
                for (int i = 0; i < Input; i++)
                {
                    if (i == u) //바이너리 넘버가 서로 다른 위치에 있을때,
                    {
                        /*"1"을 채워넣습니다.
                        crr[c][i] = '1';
                        crr[c][i + 1] = '\0';
                        *crr[c + 1] = NULL;
                    }
                }
            }
        }
    }
}
```

```
else
{
    //바이너리 넘버가 서로 같은 경우 같은 수를 집어넣습니다.
    crr[c][i] = InputMinterms[i][i];
    crr[c][i + 1] = '\0';
    *crr[c + 1] = NULL;
}

}

c++;
x++;
}

if (x == 0)
{
    strcpy(rest[L], InputMinterms[i]); //해밀디스턴스가 1인 문자열을 찾지못한경우 대상이 되는 문자열을 rest배열에 넣습니다.
    L++;
}

if (s == 0) //아무것도 해밀디스턴스차이가 1인 경우가 없는경우 그배열에 있는 문자열들은 prime implicant입니다. 고로 FindPrimeImplicant 함수를 종료시킵니다.
//종료시키면서 SumOfProduct함수를 실행시킵니다.
{
    int TrueMinterms_number = 0;
    for (int i = 0; *TrueMinterms[i] != '\0'; i++)
    {
        /*InputMinterms[i] != '\0' 일때까지 teranumber의 수를 증가시켜준다. */
        TrueMinterms_number = i + 1;
    }
}
```

```
for (int i = 0; i < 1000; i++)
{
    delete[] crr[i]; //동적할당해제
    delete[] rest[i]; //동적할당해제
    delete[] InputMinterms[i]; //동적할당해제
}

FindEssentialPrimeImplicant(prr, Input, teranumber, TrueMinterms, TrueMinterms_number); //필수 주항찾기 시작
return ;

for (int i = 0; *crr[i] != NULL; i++) //crr 배열에 있는 문자열 중에서 중복되는 것이 있을 수도 있기 때문에 지워주는 과정입니다.
{
    for (int j = 0; *crr[j] != NULL; j++)
    {
        if ((!strcmp(crr[i], crr[j])) && i != j && crr[i] != "000" && crr[j] != "000")
        {
            /*when we found duplicate words we change that word to "000" */
            strcpy(crr[j], "000");
        }
    }
}

int id = L;
int oa = L;
```

```
/*이전의 패장은 등록을 제거한 crr의 모든 문자열들을 rest 배열에 합쳐 rest의 모든 prime implicant들과 같이 나머지는 패장입니다.*/
while (*crr[pa - 1] != NULL)
{
    if ((strcmp(crr[pa - 1], "000")))
    {
        rest[ip] = crr[pa - 1];
        *rest[ip + 1] = 0;
        ip++;
        pa++;
    }
    else
    {
        pa++;
    }
}
if (s != 0)
{
    /*해당 디스토크스기 1인 경우가있는 경우 (프라이임 임플리칸트들만 있는 문자열을 못 찾은 경우) or prime implicant를 못찾은 경우 반복해서 FindPrimeImplicant를 실행시킵니다.*/
    FindPrimeImplicant(rest, input, lreallntera);
}
}
```

► FindEssentialPrimeImplicant 함수

```
void FindEssentialPrimeImplicant(char** prr, int input, int prime_implicant_number, char** TureInters, int TureInters_number)
{
    //cout << "prime_implicant_number : " << prime_implicant_number << endl;
    //cout << "TureInters_number : " << TureInters_number << endl;

    //필수주항 조합을 찾는 함수
    Best_Essential_Prime_Implicant_And_Best_Cost_Best_Boolean_Expressions:
    for (int i = 0; i < 1000; i++)
    {
        Best_Boolean_Expressions.Best_Essential_Prime_Implicant[i] = new char[input + 1]; //동적할당
    }
    Best_Boolean_Expressions.Best_Cost = 0;

    int** e_pls_arr = new int*[prime_implicant_number]; //행의 크기가 prime_implicant_number인 이차원 배열
    for (int i = 0; i < prime_implicant_number; i++) //각각의 행에 길이가 TureInters_number인 열을 할당
    {
        e_pls_arr[i] = new int[TureInters_number];
        memset(e_pls_arr[i], 0, sizeof(int)*TureInters_number);
    }
}
```

//퀵맥클러스터 감소화 방법에서 주항들과 트루민임들을 테이물화하는 과정

```
for (int i = 0; xpr[i] != '\0'; i++)
{
    for (int k = 0; <TureInters[k] != '\0'; k++)
    {
        int checker = 0;

        for (int j = 0; j < input; j++){
            if (prr[i][j] == TureInters[k][j] || prr[i][j] == '_')
            {
                checker++;
            }
        }
        if (checker == input)
        {
            e_pls_arr[i][k] = 1;
        }
    }
}
```

// 주항들의 조합을 통해 필수 주항을 찾는 과정
for (int cnt = 1; cnt <= prime_implicant_number; cnt++)
{
 //n개의 주항을 k개를 1부터 n까지 조합하는 nCk의 방법을 통해 필수 주항을 찾는 과정
 //k개의 조합이 트루민임들을 전부 커버하는 지 확인 후 커버한다면 코스트를 구한다.
 //코스트가 낮은 다른 경우도 있을 수 있으므로 이를 고려해서 코딩

```
int exit_count = 0;
vector<int> v(prime_implicant_number);
for (int i = 0; i < prime_implicant_number; i++) {
    v[i] = i;
}

vector<bool> visited(prime_implicant_number, true);

int* table_checker = new int[TureInters_number];
int* table_saver = new int[cnt];
for (int i = 0; i < v.size() - cnt; i++)
{
    visited[i] = false;
}
```

```
do {

    memset(table_checker, 0, sizeof(int) * TureInters_number);
    memset(table_saver, 0, sizeof(int) * cnt);
    int table_saver_count = 0;

    for (int i = 0; i < v.size(); i++) {
        if (visited[i])
        {
            //cout << v[i] << " ";
            table_saver[table_saver_count] = v[i];
            table_saver_count++;
        }
    }
}
```

```
for (int j = 0; j < TureInters_number; j++)
{
    if (table_checker[j] != 1)
    {
        table_checker[j] = e_pls_arr[v[i][j]];
    }
}
}
```

```
int table_checker_count = 0;
for (int i = 0; i < TureInters_number; i++)
{
    table_checker_count += table_checker[i];
}
```

```
if (table_checker_count == TureInters_number) // 조합한 주항이 트루민임들을 전부 커버하는지 확인 만약 맞다면 필수 주항 후보가 됨
{
    exit_count = 1;
    char* essential_prime_implicant[1000]; // inputInters를 저장하기 위한 char형 포인터 배열 essential_prime_implicant을 선언한다.
    for (int i = 0; i < 1000; i++)
    {
        essential_prime_implicant[i] = new char[input + 1]; //동적할당
    }
    for (int i = 0; i < cnt; i++) //배열 essential_prime_implicant에 inputInters를 저장한다.
    {
        strcpy(essential_prime_implicant[i], prr[table_saver[i]]);
        essential_prime_implicant[i + 1] = 0;
    }
    int SumOfProduct_Checker = 0;

    SumOfProduct_Checker = SumOfProduct(essential_prime_implicant, input, cnt, TureInters); //코스트를 구하는 함수 호출
}
```



```

if (Best_Boolean_Expressions.Best_Cost == 0) // 초기에는 코스트가 0이기 때문에 저장
{
    Best_Boolean_Expressions.Best_Cost = SumOfProduct_Checker;

    for (int i = 0; i < cnt; i++) // 배열 prr에 InputMinterm을 저장한다 .
    {
        strcpy(Best_Boolean_Expressions.Best_Essential_Prime_Implicant[i], essential_prime_implicant[i]);
        *Best_Boolean_Expressions.Best_Essential_Prime_Implicant[i + 1] = 0;
    }

}

else if (Best_Boolean_Expressions.Best_Cost >= SumOfProduct_Checker) // 조합의 코스트가 기존에 가지고 있던 베스트 코스트 보다 작은 경우 베스트 코스트 및 베스트 필수 추항 변경
{
    Best_Boolean_Expressions.Best_Cost = SumOfProduct_Checker;

    for (int i = 0; i < cnt; i++) // 배열 prr에 InputMinterm을 저장한다 .
    {
        strcpy(Best_Boolean_Expressions.Best_Essential_Prime_Implicant[i], essential_prime_implicant[i]);
        *Best_Boolean_Expressions.Best_Essential_Prime_Implicant[i + 1] = 0;
    }

}

for (int i = 0; i < 1000; i++)
{
    delete[] essential_prime_implicant[i]; // 동적할당 해제
}

// puts("");
} while (next_permutation(visited.begin(), visited.end()));

delete[] table_checker; // 동적할당 해제
delete[] table_saver; // 동적할당 해제

```

```

if (exit_count == 1)
{
    // 예를 들어 402에서 필수추항의 후보를 발견했다면 403, 404는 코스트가 무조건 많아 지기 때문에 볼 필요가 없다. 그러므로 nCk에서 k개를 증가시키는 for문 종료
    break;
}

Output_Result(Best_Boolean_Expressions.Best_Essential_Prime_Implicant, Best_Boolean_Expressions.Best_Cost); // 결과 파일 및 cmd 출력

for (int i = 0; i < 1000; i++)
{
    delete[] Best_Boolean_Expressions.Best_Essential_Prime_Implicant[i]; // 동적할당 해제
    delete[] prr[i]; // 동적할당 해제
    delete[] Iureminterm[i]; // 동적할당 해제
}

for (int i = 0; i < prime_implicant_number; i++) // 각각의 행에 곱이거 Iureminterm_number인 열을 할당
{
    delete[] e_pls_arr[i]; // 동적할당 해제
}

delete[] e_pls_arr; // 동적할당 해제
return;

```

▶ SumOfProduct 함수

```
int SumOfProduct(char** essential_prime_implicant , int input, int teranumber, char** Turewinters)
{
    /*아래는 SOP의 cost를 계산하는 과정이다.*/
    int* cost_table = new int [input];
    //행의 크기가 prime_implicant_number인 입차원 배열
    memset(cost_table, -1, sizeof(int)*input);
    //배열을 선언하는 이유는 input의 어떠한 위치에 not값이 들어가 있다면 그 위치를 저장하여 다른 필수주항에서 같은 위치에 not값을 사용하고자 할때 기준에 사용한 not에 대한 트랜지스터를 사용하여 코스트 값을 낮출 수 있음

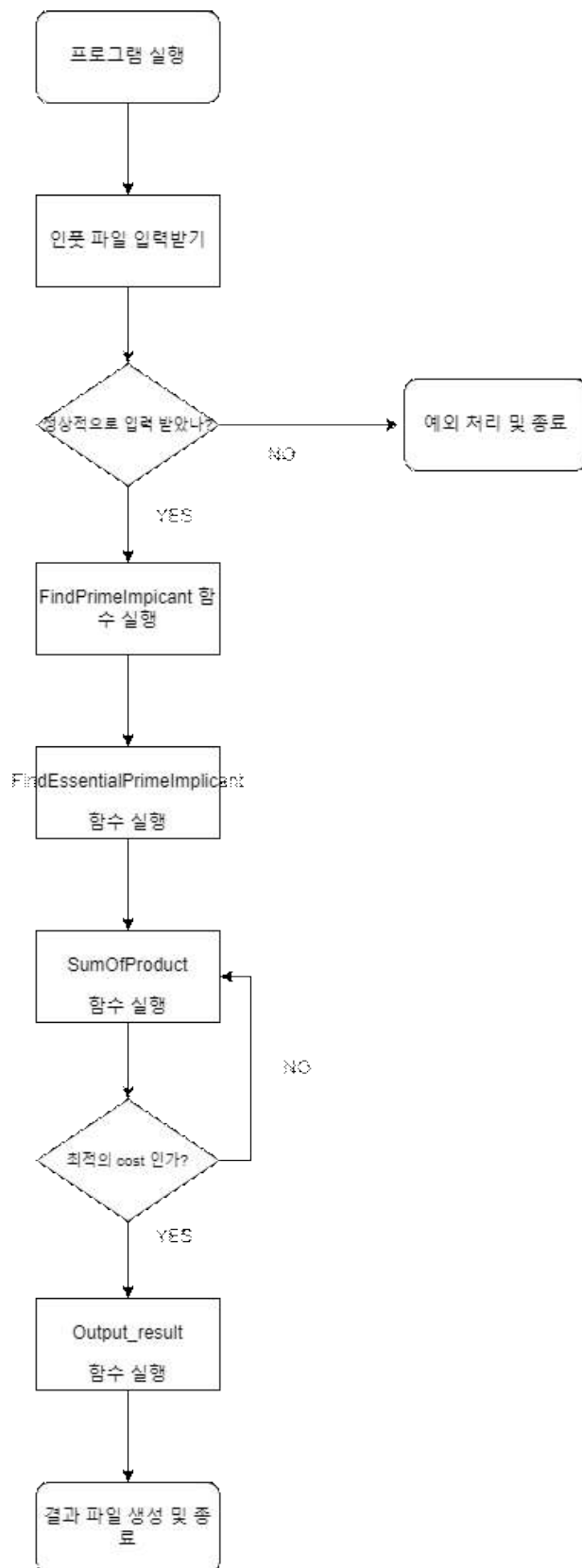
    for (int j = 0; j < input; j++)
    {
        cout << cost_table[j] << " ";
    }
    cout << endl;
    /*
    int cost = 0;
    for (int i = 0; i < teranumber; i++)
    {
        int inputnumber = 0;
        int inputposition = 0;
        for (int j = 0; j < input; j++)
        {
            if (essential_prime_implicant[i][j] != '-' && essential_prime_implicant[i][j] == '0')/*입력에 0있어 부정값이 입력으로 들어가면 인버터(트랜지스터2개)를 추가해주어야한다.*/
            {
                if (cost_table[j] == -1)
                {
                    cost += 2;
                    inputnumber++;
                    cost_table[j] = 0;
                }
            }
            else
            {
                inputnumber++;
            }
        }
        if (essential_prime_implicant[i][0] != '-' && essential_prime_implicant[i][0] == '1')
        {
            inputnumber++;
            inputposition = j;
        }
        if (inputnumber > 1)
        {
            cost += (inputnumber * 2);/*NAND게이트의 인풋의 갯수에 두배로 트랜지스터의 개수(예를들어 2인풋인 경우 트랜지스터 4개 3인풋인경우 트랜지스터 6개)가 늘어남으로 cost += (inputnumber * 2)과정을 거친다.*/
        }
        if (inputnumber > 1)
        {
            cost += 2;/*우리가 구하고자하는 것은 AND게이트임으로 NOT게이트(트랜지스터2개)를 추가하여야한다.
        }
        if (teranumber > 1)
        {
            cost += (teranumber * 2);/*프라이미 임플리칸트의 갯수가 OR게이트의 인풋의 갯수가 됨으로 인풋의 갯수에 두배로 트랜지스터의 개수(예를들어 2인풋인 경우 트랜지스터 4개 3인풋인경우 트랜지스터 6개)가 늘어남
            고로 cost += (teranumber * 2);를 계산한다.*/
        }
        if (teranumber > 1)
        {
            cost += 2;/*우리가 구하고자하는 것은 OR게이트임으로 NOT게이트(트랜지스터2개)를 추가하여야한다.
        }
        delete [] cost_table; //동적할당 해제
        return cost;
    }
}
```

▶ Output_Result 함수

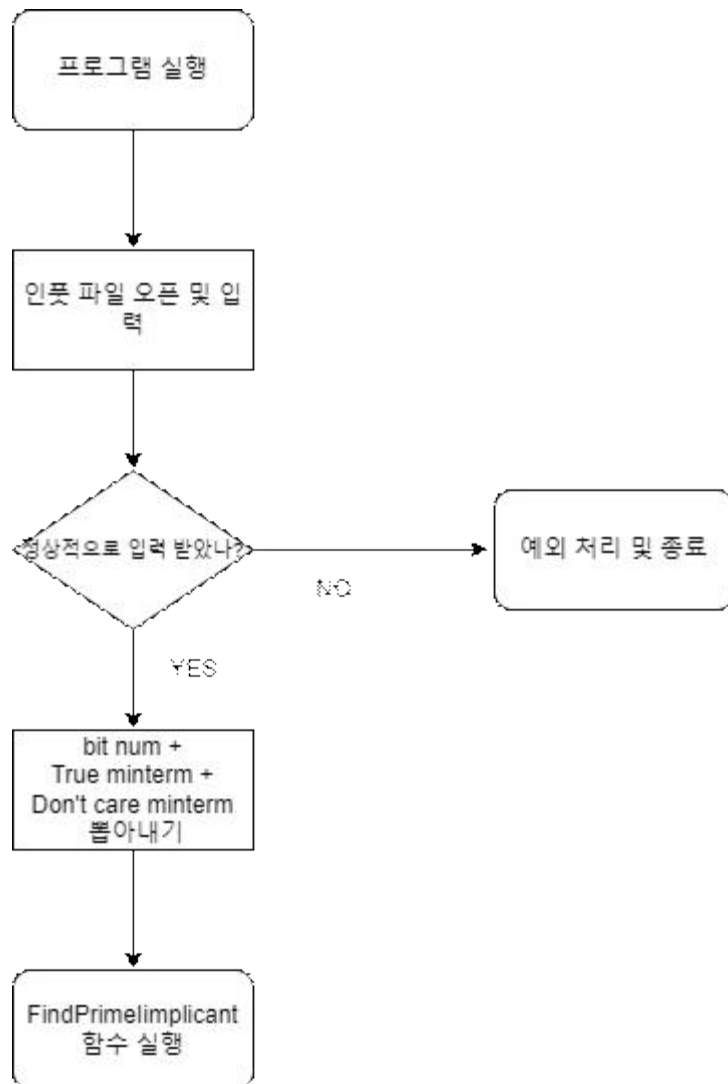
```
void Output_Result(char** Best_Essential_Prime_Implicant, int Best_Cost)
{
    /*가장 베스트 필수 주항과 베스트 코스트를 받아 파일 및 cmd에 출력하는 함수*/
    ofstream fout; //아웃풋스트림 클래스 변수 fout을 선언한다.
    fout.open("result.txt"); //result.txt파일을 연다.
    if (!fout.is_open())
    {
        cout << "파일을 열지 못하였습니다." << endl; //파일을 여는것에 실패한경우를 위한 예외처리
        return;
    }
    cout << "result: ";
    cout << endl;
    for (int i = 0; *Best_Essential_Prime_Implicant[i] != '\0'; i++)
    {
        cout << Best_Essential_Prime_Implicant[i] << endl;
    }
    cout << endl << endl;
    for (int i = 0; *Best_Essential_Prime_Implicant[i] != '\0'; i++)
    {
        fout << Best_Essential_Prime_Implicant[i] << endl;
    }
    cout << "Cost (# of transistors): ";
    cout << Best_Cost << endl;
    fout << endl << endl;
    fout << "Cost (# of transistors): " << Best_Cost;
    cout << "프로그램 성공적으로 작동하였습니다. 결과파일을 열어보십시오." << endl;
    return;
}
```

■ Flow Chart

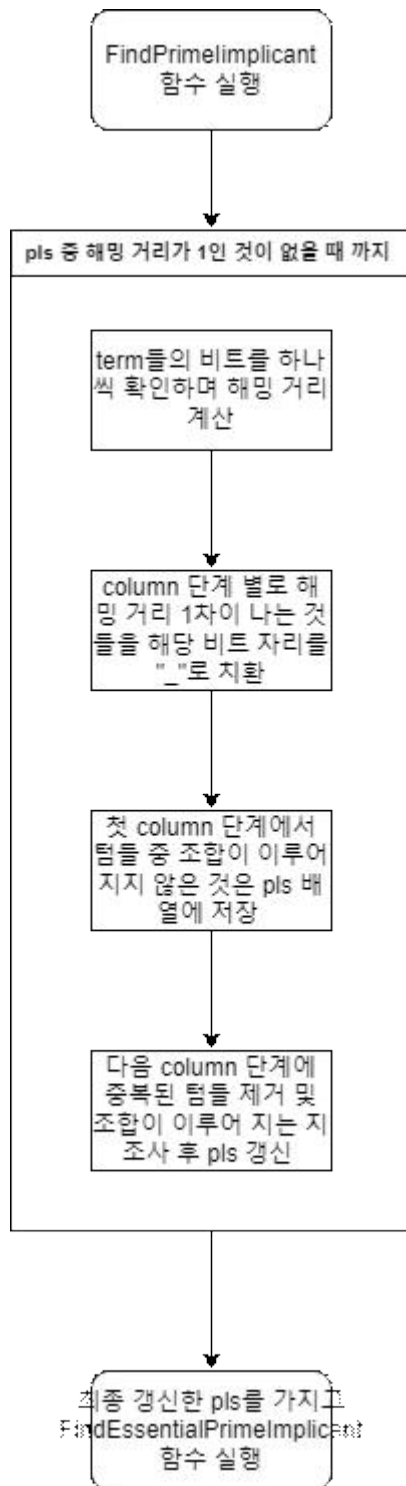
▶ 전체 프로그램



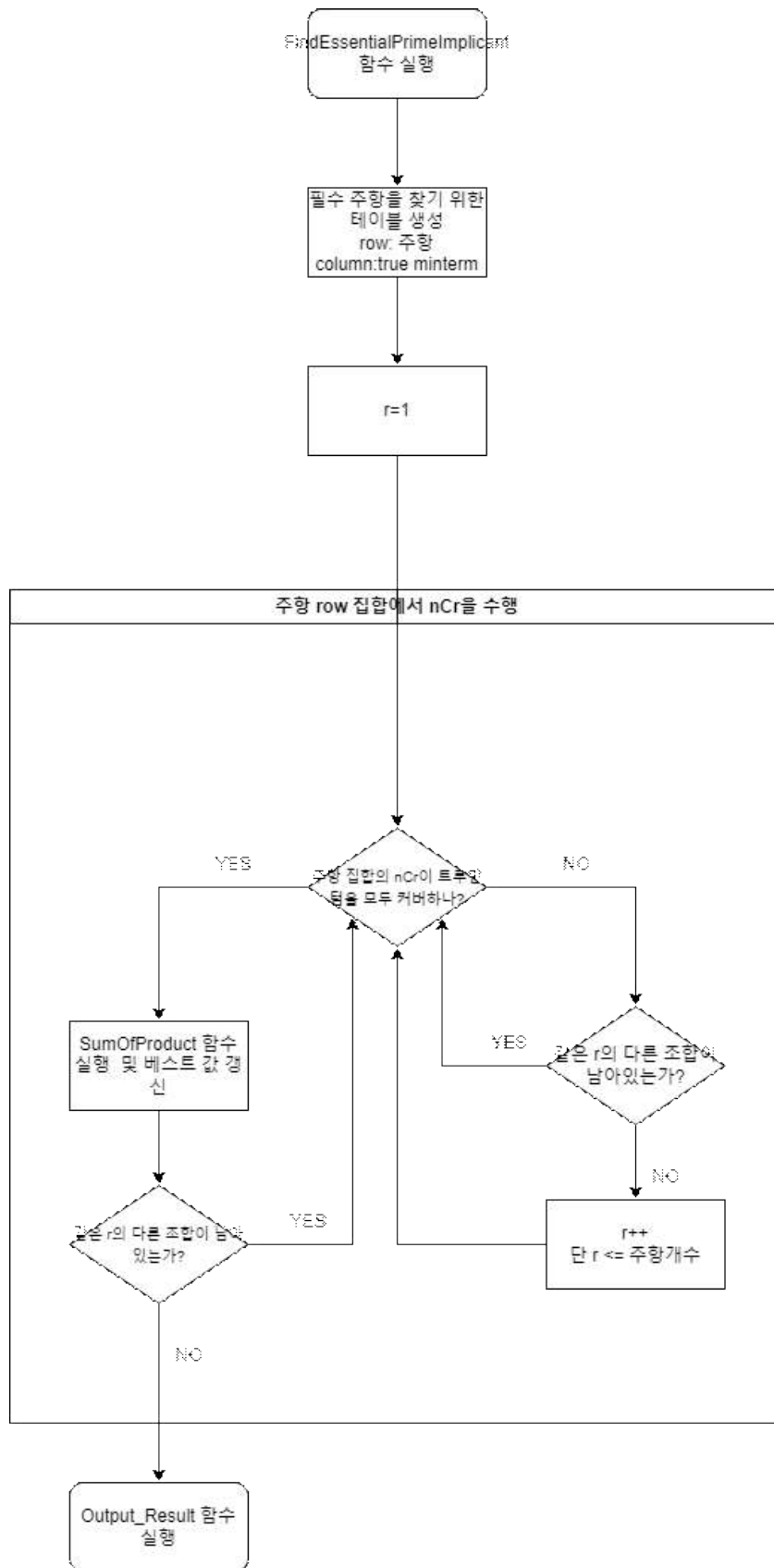
▶ main 함수



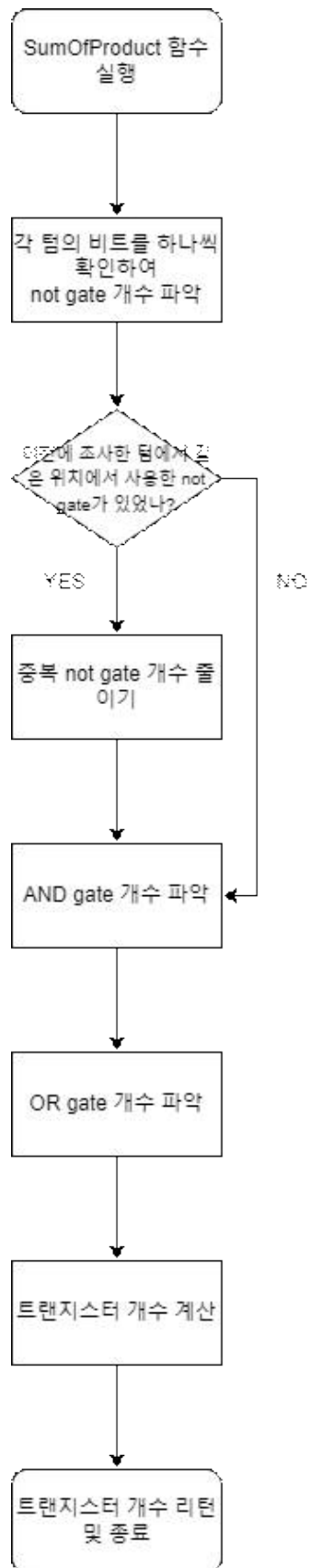
► FindPrimeImplicant 함수



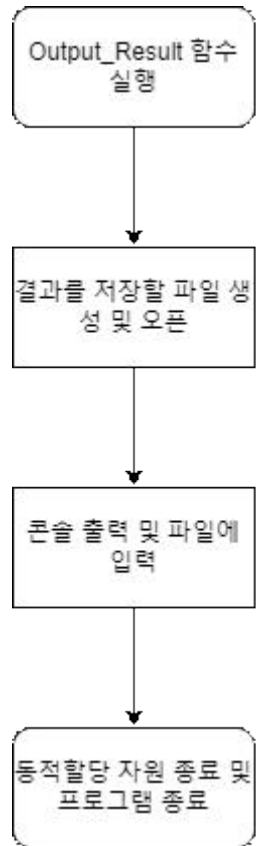
► FindEssentialPrimeImplicant 함수



► SumOfProduct 함수



▶ Output_result 함수



● Verification strategy & corresponding examples with explanation

■ 주항 찾기 알고리즘 검증 전략 및 상응하는 예시에 대한 설명

인풋 파일에 있는 minterm을 배열 포인터에 저장한다. 제안서에 있는 인풋으로 예를 들어 설명하면 다음과 같은 형태의 배열이 형성될 것이다.

0	0	0	0
0	1	0	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
0	1	1	1
1	1	0	1
1	1	1	1

해당 배열을 구성하여 해밍 거리를 계산하기 편하게 되었다. 이후 minterm의 개수 만큼 for문을 돌며 각각의 비트 자리를 비교해 가며 해밍 거리를 계산한다. 해밍 거리가 1차 이 나는 텀들이 있음이 확인 하고 해당 부분을 "_"로 치환한 후 다음 따로 위의 배열처럼 저장한다. 또한 조합이 이루어지지 않은 부분은 다음단계에서도 조합이 이루어지지 않기 때문에 배열에 저장한다. 이후 재귀적 방식으로 해당 배열을 위의 단계를 다시한번 거친다. 이후 생성된 배열의 경우 아래와 같다.

1	0	1	0
0	-	0	0
0	1	0	-
0	1	-	0
0	1	-	1
-	1	0	1
-	1	1	1
1	1	-	1

해당 배열에서 해밍 거리 1인 부분을 찾고 조합을 이루 지못한 부분을 찾아 다시 배열에 저장하여 재귀적 방식으로 함수를 다시 실행하면 최종적으로

1	0	1	0
0	-	0	0
-	1	1	1
0	1	-	-
-	1	-	1

더는 해밍 거리가 1인 부분이 존재하지 않기 때문에 해당 배열이 주항들에 대한 배열이다.

■ 필수 주항 찾기 알고리즘 검증 전략 및 상응하는 예시에 대한 설명

위의 예시에서 찾은 주항을 가지고 필수 주항을 찾는 알고리즘에 대해 말하면 다음과 같다. 먼저 row : 주항, column : true minterm에 대한 테이블을 생성한다. 이는 아래와 같다.

	0100(0번째)	0101(1번째)	0110(2번째)	1001(3번째)	1010(4번째)
1010 (0번째)					
0_00 (1번째)					
1_01 (2번째)					
01__ (3번째)					
_1_1 (4번째)					

해당 테이블에서 트루 민텀을 커버하는 주항을 1로 표시하고 못하는 부분을 0으로 표시하면 아래와 같다.

	0100(0번째)	0101(1번째)	0110(2번째)	1001(3번째)	1010(4번째)
1010 (0번째)	0	0	0	0	1
0_00 (1번째)	1	0	0	0	0
1_01 (2번째)	0	0	0	1	0
01__ (3번째)	1	1	1	0	0
_1_1 (4번째)	0	1	0	0	0

여기서 이번 프로젝트에서 사용한 핵심 알고리즘이 구현되는데 이는 다음과 같다.

위의 테이블에서 n개의 row중 r개를 뽑아 각 column의 1의 위치를 파악했을 시 즉 nCr 즉 조합을 사용하여 각 column이 1을 가지고 있는 것을 확인한다. 각 컬럼이 모두1을 가지고 있음을 확인한다면 이는 필수 주항이 될 후보가 된 것이다. 이를 증명할 수 있는 이유는 위의 예시로 설명하면 4C1의 경우에서 차근차근 선택하는 r의 개수를 늘려가며 조합 각각을 확인했을 시 발견되는 조합의 경우 가장 적은 SOP의 Bool식으로 표현이 가능하기 때문이다. 여기서 논리를 조금 보충하면 만약 r을 차근히 증가해 가며 발견한 트루민텀을 커버하는 SOP를 발견한 경우 해당 r에 대한 다른 만족하는 조합이 있을 수 있다. 이러한 경우는 필수 주항을 못찾은 경우와 필수주항이 커버하는 트루민텀을 제외하고 주항 중 가장 적합한 것을 선택하는 경우인데 이러한 경우 해당 크기의 r에 대한 조합의 경우들 중 최소의 Cost를 가지는 SOP를 선택하면 된다. SOP를 찾은 이후 r을 증가시켜 더 큰 SOP의 조합을 볼 필요가 없기 때문에 break 문을 써서 조합을 찾는 for문을 종료시켜 불필요한 반복을 하지 않음으로 프로그램 실행 속도도 향상 시킬 수 있다. 예를 들어 위의 테이블에서 가장 먼저 발견되는 필수 주항 예비 후보는 5C3 조합에서 0번째, 2번째, 3번째이다 해당 SOP의 트랜지스터 사용개수 COST를 구하고 이를 저장하고 5C3의 다른 조합을 볼 것인데 해당 하는 조합이 나오지 않는다. 이후 5C4를 하지 않고 반복을 종료하여 최종 pls와 cost를 구하게 된다.

■ SOP에 대한 최적의 Cost 찾는 알고리즘 검증 전략 및 상응하는 예시에 대한 설명

SOP 각각의 텀을 비트단위로 나누어 가장 먼저 NOT 게이트 사용 여부를 확인한다. 여기서 핵심 알고리즘은 NOT 게이트 다른 텀들도 같은 위치에서 사용이 가능하다는 것이다. 그렇기 때문에 각 텀의 비트를 확인하여 0일 경우 NOT 게이트 사용 개수를 증가 시키지 말고 해당 위치에서 NOT게이트가 사용되었는 지 확인한다. 예를들어 $A'BC'D+A'BCD$ 라는 SOP에서 A의 경우 NOT 게이트를 한번만 사용할 수 있다는 것이다. 이후 SOP의 각 비트의 개수를 통해 사용한 AND게이트를 구하기 위해 NAND 게이트로 가정하여 트랜지스터를 개산하게 되면 NAND게이트의 인풋의 갯수에 두배로 트랜지스터의 개수(예를들어 2인풋인 경우 트랜지스터 4개 3인풋인 경우 트랜지스터 6개)가 늘어나게되고 최종적으로 구한 트랜지스터개수에 NOT게이트 트랜지스터 2개를 더하여 구한다. 이후 최종 epls의 갯수가 NOR게이트의 인풋의 갯수가 됨으로 인풋의 갯수에 두배로 트랜지스터의 개수(예를 들어 2인풋인 경우 트랜지스터 4개 3인풋인 경우 트랜지스터 6개)가 늘어난다. 이후 마찬가지로 OR게이트를 위해 NOT게이트 트랜지스터 2개를 더하여 구한다.

- A testcase that I think it is very hard to solve

■ $f(A,B,C,D,E,F,G) = \sum m(64, 65, 69, 71, 74, 78) + \sum d(79)$

비트 개수가 7개이며 필수 주항이 커버하는 트루 민텀을 제외하고 주항 중 가장 적합한 것을 선택하는 경우인데 필수 주항을 찾는 알고리즘을 조합의 경우로 구했기 때문에 해당 부분이 잘 나타나는지 확인해 볼 수 있다.

Column1	Column2	Column3
1000000	100000_ *	
1000001	1000_01 *	
1000101	10001_1 *	
1001010	1001_10 *	
1000111	100_111 *	
1001110	100111_ *	
1001111		

	1000000	1000001	1000101	1001010	1000111	1001110
100000_	x	x				
1000_01		x	x			
10001_1			x		x	
1001_10				x		x
100_111					x	
100111_						x

$f = 100000_ + 10001_1 + 1001_10$, $f = AB'C'D'E'F' + AB'C'D'EG + AB'C'DFG'$

6-input and gate(14)*3 + 3-input or gate(8) + not gate(2)*6 = 42+8+12 = 62


 input_minterm.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
7
m 1000000
m 1000001
m 1000101
m 1001010
m 1000111|
m 1001110
d 1001111
```

```
Microsoft Visual Studio 디버깅 콘솔
result:
100000_
100001_1
1001_10

Cost (# of transistors): 62
프로그램 성공적으로 작동하였습니다. 결과파일을 열어보십시오.
C:\Users\User\Desktop\4학년 1학기\디지털 논리회로1\Quine_McCluskey\64\Debug\Quine_McCluskey.exe(프로세스 68056개)이(가)
종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

 result.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
|100000_  
10001_1  
1001_10
```

Cost (# of transistors): 62

결과가 잘 출력됨을 알 수 있다.

■ bit의 개수가 10개이상(15개인 경우) + 30개 이상의 minterm을 입력받은 경우

input_minterm.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

15

```
m 0000000000000000
m 0000000000000001
m 0000000000000010
m 0000000000000011
m 0000000000000100
m 0000000000000101
m 0000000000000110
m 0000000000000111
m 0000000000001000
m 0000000000001100
m 0000000000001101
m 0000000000001110
m 0000000000001111
m 0000000000010001
m 0000000000010101
d 0000000000010110
d 0000000000010111
m 0000000000011101
m 0000000000011111
m 0000000000100000
m 0000000000100001
m 0000000000100011
m 0000000000100100
m 0000000000100101
m 0000000000100110
m 111111110110111
m 111111110111000
d 111111110111001
d 111111110111010
d 111111110111011
d 111111110111100
d 111111110111101
```

C:\Windows\System32\cmd.exe

```
C:\Users\User\Desktop\QM-Memory-Check-main>QMChk.exe Quine_McCluskey.exe
Automatic Process Memory Checker
```

```
=====
Trying to create child process: Quine_McCluskey.exe
Running: Quine_McCluskey.exe
Please wait....
```

```
result:
111111110110111
000000000000__00
0000000000_000_1
0000000000_0_01
0000000000_001_0
1111111101110__
000000000000___
0000000000_00_0_
0000000000_1__
0000000000_1_1
```

Cost (# of transistors): 328

프로그램 성공적으로 작동하였습니다. 결과파일을 열어보십시오.

RUN REPORT

```
=====
Peak page size: 23 MiB (24117248 B)
Peak working set size: 26 MiB (28024832 B)
Runtime: 63 msec
```

result.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
111111110110111
000000000000__00
0000000000_000_1
0000000000_0_01
0000000000_001_0
1111111101110__
000000000000___
0000000000_00_0_
0000000000_1__
0000000000_1_1
```

Cost (# of transistors): 328

프로그램 실행 속도가 작은 비트 사이즈 예시보단 조금 더 걸렸지만 결과를 만들어 낼 수 있음을 알 수 있다. 또한 결과의 주항들이 트루 민텀을 커버함을 알 수 있다.