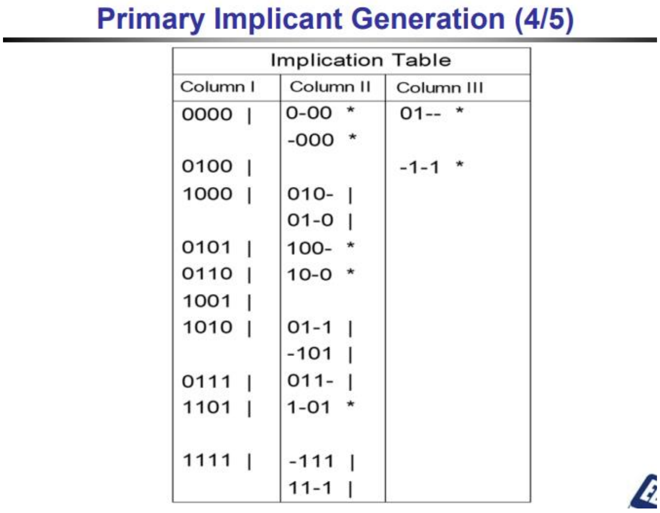
HW1: Quine-McCluskey Method report

一、演算法原理探討：

Quine-McCluskey Method 是一個化簡布林函數之方式，功能上與卡諾圖相等，但在處理**四個變數**以上之情況下比起卡諾圖實用。但因為其時間複雜度為 **NP-completed**，執行時間隨輸入大小呈**指數成長**，在很多變數之情況非常消耗時間。此方法分成以下兩個部分：

1. 生成所有 prime implicant:

   如老師講義的圖所示，首先依照 1 的數量分成好幾組，兩兩互相比對每一組的數碼，如果可以找到只相差 1 的組合，就會再下一回合新生成相差部分變成 don’t care(以-1 表示)之數碼，若無法，則以記號*表示，代表其為 prime implicant。



Primary Implicant Generation (4/5)

<table>
<tr><th colspan="3">Implication Table</th></tr>
<tr><th>Column I</th><th>Column II</th><th>Column III</th></tr>
<tr><td>0000 |</td><td>0-00 *</td><td>01-- *</td></tr>
<tr><td></td><td>-000 *</td><td></td></tr>
<tr><td>0100 |</td><td></td><td>-1-1 *</td></tr>
<tr><td>1000 |</td><td>010- |</td><td></td></tr>
<tr><td></td><td>01-0 |</td><td></td></tr>
<tr><td>0101 |</td><td>100- *</td><td></td></tr>
<tr><td>0110 |</td><td>10-0 *</td><td></td></tr>
<tr><td>1001 |</td><td></td><td></td></tr>
<tr><td>1010 |</td><td>01-1 |</td><td></td></tr>
<tr><td></td><td>-101 |</td><td></td></tr>
<tr><td>0111 |</td><td>011- |</td><td></td></tr>
<tr><td>1101 |</td><td>1-01 *</td><td></td></tr>
<tr><td>1111 |</td><td>-111 |</td><td></td></tr>
<tr><td></td><td>11-1 |</td><td></td></tr>
</table>

2. 找尋其中之 minimum cover:

   我們可以將問題表示成每個 onset 會等於某幾個 prime implicant 之 sum，又因為 onset 的結果一定要為 1，可以表示成如下圖所示之 product of sum form，我們再將 product of sum 拆開變成 sum of product，再利用各種化簡規則化到最簡，此時，統計所有 product 的項數，找尋其中最小的滿足 min literal 的解，即為本題答案。

Petrick's Method

- Solve the **satisfiability** problem of the following function

$P = (P1+P6)(P6+P7)P6(P2+P3+P4)(P3+P5)P4(P5+P7)=1$

		4	5	6	8	9	10	13
P1	0,4 (0-00)	×						
P2	0,8 (-000)				×			
P3	8,9 (100-)				×	×		
P4	8,10 (10-0)				×		×	
P5	9,13 (1-01)					×		×
P6	4,5,6,7 (01--)	×	×	×				
P7	5,7,13,15 (-1-1)		×					×

- Each term represents a corresponding column
- Each column must be chosen at least once
- All columns must be covered

二、程式碼實作:

1. 生成所有 prime implicant:

(1).讀 input file 檔案，存入變數數量、onset 跟 dcset 的數值，並將 onset 跟 dcset 十進位轉換為二進位，存入二維的 vector 中。

```
//.m .d transform to string(eg:0010)
vector<vector<int>> dec_to_binary(int var, vector<int> set)
{
    vector<vector<int>> dtob;

    for(int i=0; i < set.size(); ++i){
        vector<int> trans_set(var);

        for(int j=0;j<var;++j){ //initialize
            trans_set[j] = 0;
        }

        for(int j=var-1; j>=0 ;--j){
            trans_set[j] = set[i] % 2;
            set[i] = set[i] /2 ;
        }
        dtob.push_back(trans_set);
    }
    return dtob;
}
```

(2).統計各個數串 1 的數量，進行分組(此時使用到 3D int vector)，相鄰差一之 column 兩兩互相比較，利用另外一個 label vector 紀錄是否還能化簡(初始為 n)，若發現只差一個元素，把那個數字設成-1，將化簡過之數串 label 記為 |，代表此回合比較結束會被移除，若無法找到另一數與之化簡，則知數串為 prime implicant，label 記為 *。

(3).最終刪除重複產生之化簡項

(4).合併 grouping 內容進 vector <pri_implicant> con_gro 中(統計 dc 項數目並依照數目由大到小排序。

```
//"-" more smaller
struct vec{
    int dc_count;
    vector<int> pri;
};

typedef struct vec pri_implicant;

bool mycompare(const pri_implicant p1,const  pri_implicant p2){
    return  p1.dc_count > p2.dc_count;
}
```

```
    //concatenate
    vector <pri_implicant> con_gro;

    for(int i=0;i<grouping.size();++i){
        for(int j=0;j<grouping[i].size();++j){
            pri_implicant prii;
            prii.pri.assign(grouping[i][j].begin(),grouping[i][j].end());
            prii.dc_count = 0;
            for(int k=0;k<grouping[i][j].size();++k){
                if(grouping[i][j][k] == -1){
                    prii.dc_count++;
                }
            }
            con_gro.push_back(prii);
        }
    }

    sort(con_gro.begin(),con_gro.end(),mycompare);
```

2. Petrick's method 找 mincover:

(1) 尋找 onset 與 primary implicant 在圖上的交集，將 sum 存入 product of sum vector 裡面。

```
    //product of sum form
    for(int i=0;i<onset.size(); ++i){
        //vector<vector<int>> sum;
        vector<string> sum;
        for(int j=0;j<con_gro.size();++j){
            for(int k=0;k<input_variable;++k){
                if(onset[i][k] == con_gro[j].pri[k]) ++cnt;
                else if(con_gro[j].pri[k] == -1) ++cnt;
            }
            if(cnt == input_variable){
                string con;
                for(int k=0;k<input_variable;++k){
                    if(con_gro[j].pri[k]>= 0) con += to_string(con_gro[j].pri[k]);
                    else if(con_gro[j].pri[k] == -1) con += "-";
                }
                sum.push_back(con);
            }
            cnt = 0;
        }
        product_of_sum.push_back(sum);
    }
```

(2) 將 product of sum 展開成 sum of product

```
void Expansion(set<string> &product,int i_pos,int maxi)
{
    if(i_pos == maxi){
        string str = settostring(product); //expansion
        if( s_SOP.find(str) == s_SOP.end()){
             s_SOP.insert(str);
             sum_of_product.push_back(product);
        }
    }

    for(int i=0; i< product_of_sum[i_pos].size(); ++i){
        if( product.find(product_of_sum[i_pos][i]) == product.end()){
            product.insert(product_of_sum[i_pos][i]);
            Expansion(product, i_pos+1, maxi);
            product.erase(product_of_sum[i_pos][i]);
        }
        else{
            Expansion(product, i_pos+1, maxi);
        }
    }
}
```

(3) 排序 sop 之 product 變數數目由小到大

```
bool sort_by_SOP(const set<string> a, const set<string> b) {
    return ( a.size() < b.size() ) ;
}
```

(4) 找尋變數最小而 literal 數統計也最小之 product 即為 min cover

```
/*-------------------------------Petrick's method-------------------------------*/
    int sop_index = 0;
    int literal = 0;
    int min_literal = 99999,min_index;

    set <string> product;
    Expansion(product,0,product_of_sum.size() );
    sort( sum_of_product.begin(), sum_of_product.end(), sort_by_SOP);

    for(int i=0;i<sum_of_product.size();++i)
    {
        if(sum_of_product[i].size()>sum_of_product[0].size())
        {
            sop_index = i - 1;
            break;
        }
    }
    for(int i=0;i<=sop_index;++i){
        //calculate literal
        literal = 0;
        for(iter = sum_of_product[i].begin();iter != sum_of_product[i].end(); ++iter){
            for(int j=0;j<(*iter).size();++j)
            {
                if( (*iter)[j] == '0')
                {
                    literal += 1;
                }
                else if((*iter)[j] == '1')
                {
                    literal += 1;
                }
            }
        }
        if(literal< min_literal){
            min_literal = literal;
            min_index = i;
        }
    }
    /*
```

三、參考資料:

1. https://www.allaboutcircuits.com/technical-articles/prime-implicant-simplification-using-petricks-method/
2. https://github.com/topics/quine-mccluskey-algorithm
3. https://medium.com/mirkat-x-blog/implement-quine-mccluskey-algorithm-and-petricks-method-in-c-40168163474