K Map Simplification

In previous chapters, we have simplified the Boolean functions using Boolean postulates and theorems. It is a time consuming process and we have to re-write the simplified expressions after each step.

To overcome this difficulty, **Karnaugh** introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2n cells for ‘n’ variables. The adjacent cells are differed only in single bit position.

### A Typical K-Map

The K-map method of solving the logical expressions is referred to as the graphical technique of simplifying Boolean expressions. K-maps are also referred to as 2D truth tables as each K-map is nothing but a different format of representing the values present in a one-dimensional truth table.

K-maps basically deal with the technique of inserting the values of the output variable in cells within a rectangle or square grid according to a definite pattern. The number of cells in the K-map is determined by the number of input variables and is mathematically expressed as two raised to the power of the number of input variables, i.e., 2n, where the number of input variables is n.

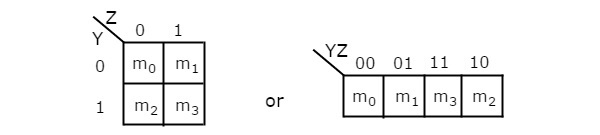
Thus, to simplify a logical expression with two inputs, we require a K-map with 4 (=22) cells. A four-input logical expression would lead to a 16 (=24) celled-K-map, and so on.

## **K-Maps for 2 to 5 Variables**

K-Map method is most suitable for minimizing Boolean functions of 2 variables to 5 variables. Now, let us discuss about the K-Maps for 2 to 5 variables one by one.

### 2 Variable K-Map

The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.



Z

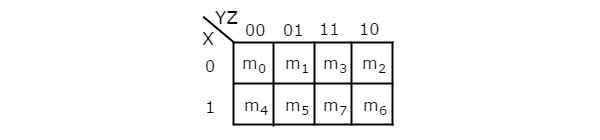
Y Z’ Z

|  |  |
| --- | --- |
| Y’ |  |
| Y |  |

* There is only one possibility of grouping 4 adjacent min terms.
* The possible combinations of grouping 2 adjacent min terms are {(m0, m1), (m2, m3), (m0, m2) and (m1, m3)}.

### 3 Variable K-Map

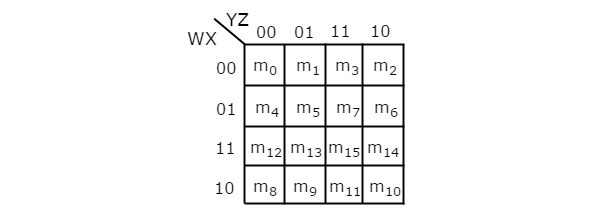
The number of cells in 3 variable K-map is eight, since the number of variables is three. The following figure shows **3 variable K-Map**.



* There is only one possibility of grouping 8 adjacent min terms.
* The possible combinations of grouping 4 adjacent min terms are {(m0, m1, m3, m2), (m4, m5, m7, m6), (m0, m1, m4, m5), (m1, m3, m5, m7), (m3, m2, m7, m6) and (m2, m0, m6, m4)}.
* The possible combinations of grouping 2 adjacent min terms are {(m0, m1), (m1, m3), (m3, m2), (m2, m0), (m4, m5), (m5, m7), (m7, m6), (m6, m4), (m0, m4), (m1, m5), (m3, m7) and (m2, m6)}.
* If x=0, then 3 variable K-map becomes 2 variable K-map.

### 4 Variable K-Map

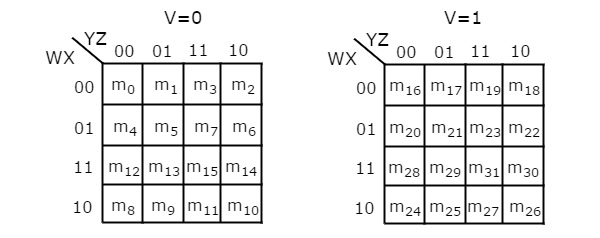
The number of cells in 4 variable K-map is sixteen, since the number of variables is four. The following figure shows **4 variable K-Map**.



* There is only one possibility of grouping 16 adjacent min terms.
* Let R1, R2, R3 and R4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C1, C2, C3 and C4 represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are {(R1, R2), (R2, R3), (R3, R4), (R4, R1), (C1, C2), (C2, C3), (C3, C4), (C4, C1)}.
* If w=0, then 4 variable K-map becomes 3 variable K-map.

### 5 Variable K-Map

The number of cells in 5 variable K-map is thirty-two, since the number of variables is 5. The following figure shows **5 variable K-Map**.



* There is only one possibility of grouping 32 adjacent min terms.
* There are two possibilities of grouping 16 adjacent min terms. i.e., grouping of min terms from m0 to m15 and m16 to m31.
* If v=0, then 5 variable K-map becomes 4 variable K-map.

In the above all K-maps, we used exclusively the min terms notation. Similarly, you can use exclusively the Max terms notation.

## **Minimization of Boolean Functions using K-Maps**

If we consider the combination of inputs for which the Boolean function is ‘1’, then we will get the Boolean function, which is in **standard sum of products** form after simplifying the K-map.

Similarly, if we consider the combination of inputs for which the Boolean function is ‘0’, then we will get the Boolean function, which is in **standard product of sums** form after simplifying the K-map.

Follow these **rules for simplifying K-maps** in order to get standard sum of products form.

* Select the respective K-map based on the number of variables present in the Boolean function.
* If the Boolean function is given as sum of min terms form, then place the ones at respective min term cells in the K-map. If the Boolean function is given as sum of products form, then place the ones in all possible cells of K-map for which the given product terms are valid.
* Check for the possibilities of grouping maximum number of adjacent ones. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
* Each grouping will give either a literal or one product term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single ‘1’ is not covered with any other groupings but only that grouping covers.
* Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

**Note 1** − If outputs are not defined for some combination of inputs, then those output values will be represented with **don’t care symbol ‘x’**. That means, we can consider them as either ‘0’ or ‘1’.

**Note 2** − If don’t care terms also present, then place don’t cares ‘x’ in the respective cells of K-map. Consider only the don’t cares ‘x’ that are helpful for grouping maximum number of adjacent ones. In those cases, treat the don’t care value as ‘1’.

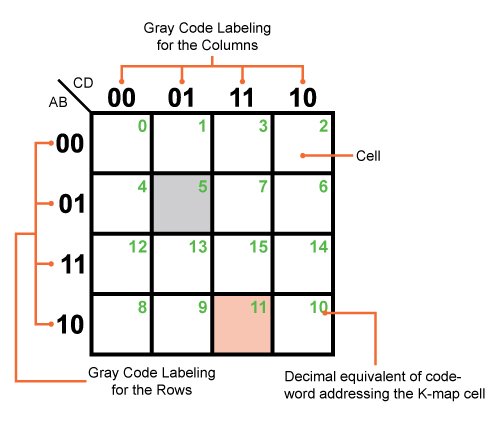
### Gray Coding

Further, each cell within a K-map has a definite place-value which is obtained by using an encoding technique known as [Gray code](https://www.allaboutcircuits.com/technical-articles/gray-code-basics/" \t "_blank).

The specialty of this code is the fact that the adjacent code values differ only by a single bit. That is, if the given code-word is 01, then the previous and the next code-words can be 11 or 00, in any order, but cannot be 10 in any case.

In K-maps, the rows and the columns of the table use Gray code-labeling which in turn represent the values of the corresponding input variables. This means that each K-map cell can be addressed using a unique Gray Code-Word.

These concepts are further emphasized by a typical 16-celled K-map shown in Figure 1, which can be used to simplify a logical expression comprising of 4-variables (A, B, C and D mentioned at its top-left corner).



##### Figure 1: A typical but empty Karnaugh map with 16 cells

Here the rows and the columns of the K-map are labelled using 2-bit Gray code, shown in the figure, which assigns a definite address for each of its cells.

For example, the grey coloured cell of the K-map shown can be addressed using the code-word "0101" which is equivalent to 5 in decimal (shown as the green number in the figure) and corresponds to the input variable combination A̅BC̅D or A+B̅+C+D̅, depending on whether the input–output relationship is expressed in SOP (sum of products) form or POS (product of sums) form, respectively.

Similarly, AB̅CD or A̅+B+C̅+D̅ refers to the Gray code-word of "1011", equivalent to 11 in decimal (again, shown in green in the figure), which in turn means that we are addressing the pink-colored K-map cell in the figure.

Moreover, the column and row headings must be in Gray code order, or the map will not work as a Karnaugh map. Cells sharing common Boolean variables would no longer be adjacent, nor show visual patterns.

Adjacent cells vary by only one bit because a Gray code sequence varies by only one bit.

**Steps to solve expression using K-map-** 

1. Select K-map according to the number of variables.
2. Identify minterms or maxterms as given in problem.
3. For SOP put 1’s in blocks of K-map respective to the minterms (0’s elsewhere).
4. For POS put 0’s in blocks of K-map respective to the maxterms(1’s elsewhere).
5. Make rectangular groups containing total terms in power of two like1, 2,4,8,16 ..(except 1) and try to cover as many elements as you can in one group.
6. From the groups made in step 5 find the product terms and sum them up for SOP form.

Example:

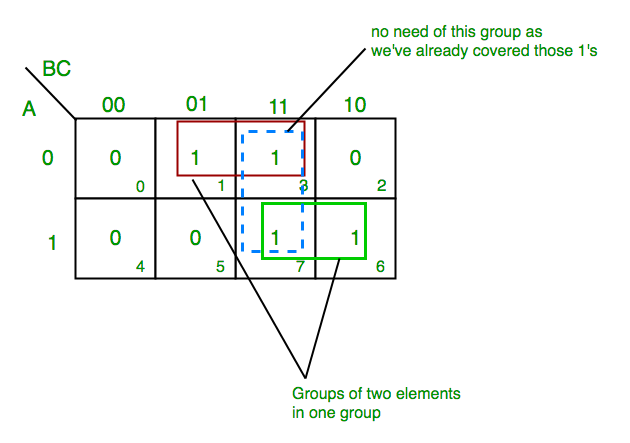
## **SOP FORM**

1. **K-map of 3 variables-**

Z= ∑A,B,C(1,3,6,7)

OR

1. A’B’C+A’BC+ABC’+ABC  
     A’c



From **red** group we get product term—

A’C

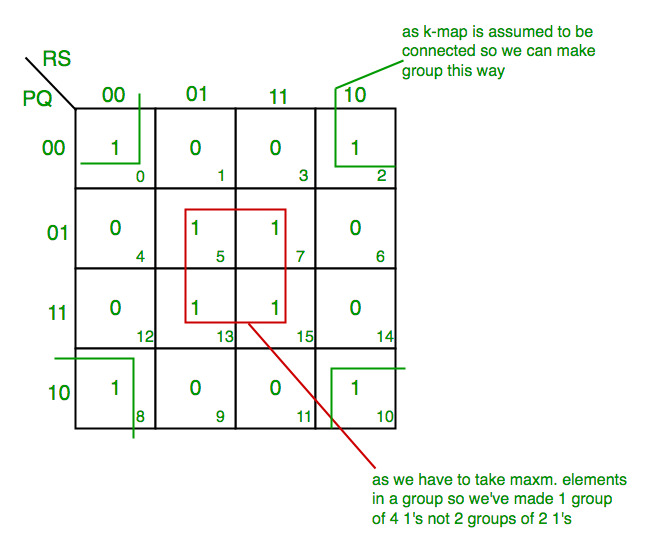
From**green** group we get product term—

AB

Summing these product terms, we get- **Final expression (A’C+AB)** 

1. **K-map for 4 variables**

F(P,Q,R,S)=∑(0,2,5,7,8,10,13,15) 

[](https://media.geeksforgeeks.org/wp-content/uploads/K-Map-Karnaugh-Map-2-1.png)

From **red** group we get product term—

QS

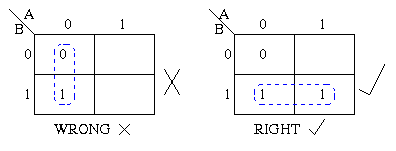
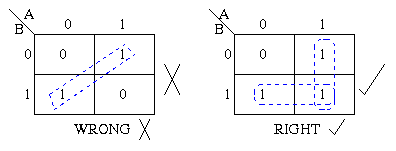
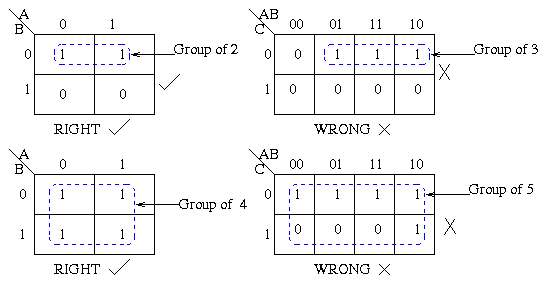
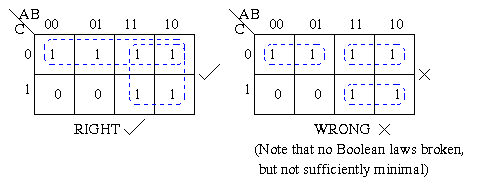
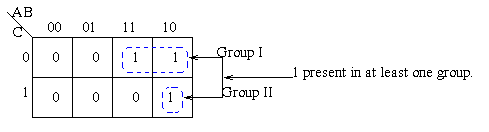
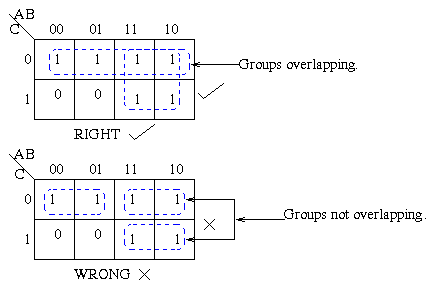
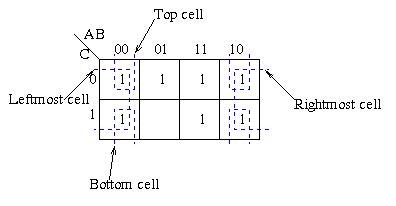
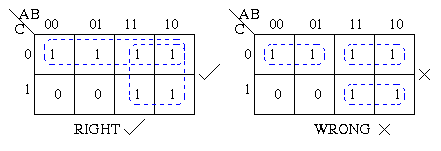
From **green** group we get product term—

Q’S’

Summing  these product terms  we get- **Final expression (QS+Q’S’)**

# **Karnaugh Maps - Rules of Simplification**

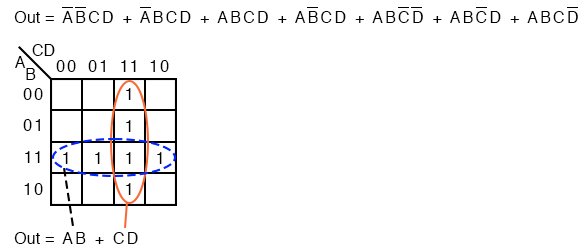
The Karnaugh map uses the following rules for the simplification of expressions by *grouping* together [adjacent](http://www.ee.surrey.ac.uk/Projects/Labview/common/glossary.html#Adj) cells containing *ones*

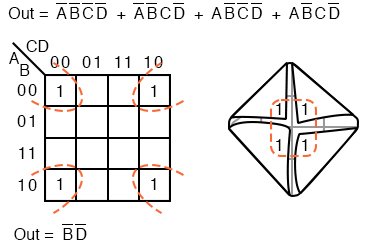
* **Groups may not include any cell containing a zero  
  **
* **Groups may be horizontal or vertical, but not diagonal.  
  **
* **Groups must contain 1, 2, 4, 8, or in general 2n cells.  
  That is if n = 1, a group will contain two 1's since 21 = 2.  
  If n = 2, a group will contain four 1's since 22 = 4.  
  **
* **Each group should be as large as possible.  
  **
* **Each cell containing a *one* must be in at least one group.  
  **
* **Groups may overlap.  
  **
* **Groups may wrap around the table. The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell.  
  **
* **There should be as few groups as possible, as long as this does not contradict any of the previous rules.  
  **

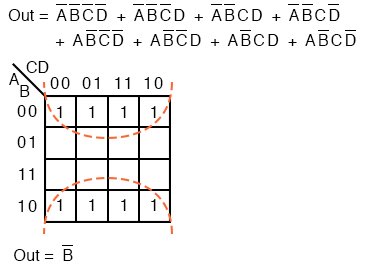
**Summmary:**

1. No zeros allowed.
2. No diagonals.
3. Only power of 2 number of cells in each group.
4. Groups should be as large as possible.
5. Everyone must be in at least one group.
6. Overlapping allowed.
7. Wrap around allowed.
8. Fewest number of groups possible.

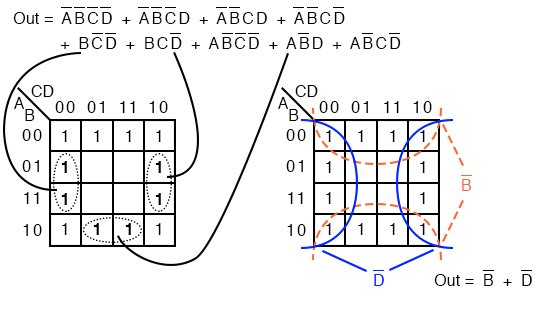
Examples:

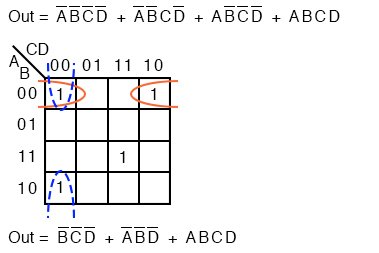


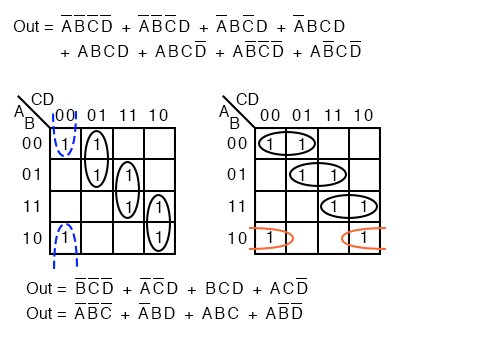


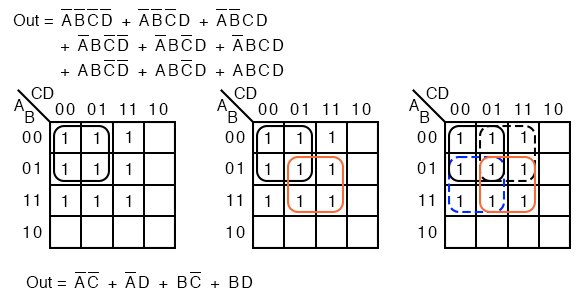


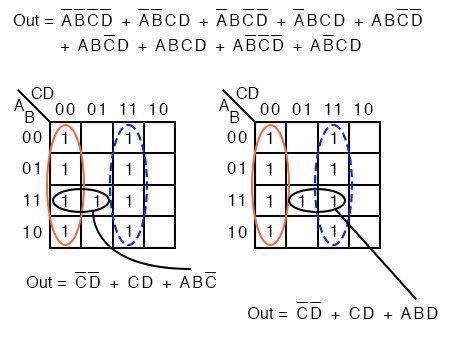
Standard from ->BC’D’=BC’D’\*1=BC’D’(A+A’)=ABC’D’+A’BC’D’

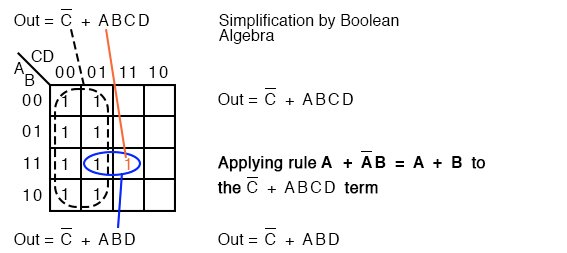








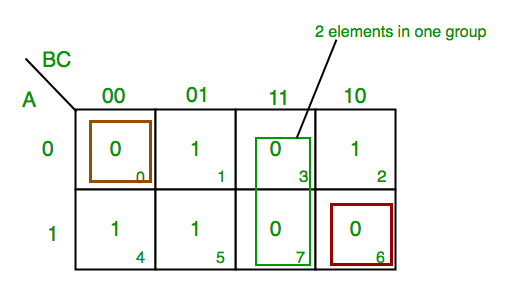




## **POS FORM**

1. **K-map of 3 variables-**

F(A,B,C)=π(0,3,6,7)

[](https://media.geeksforgeeks.org/wp-content/uploads/K-Map-Karnaugh-Map-4.png)

From **red**group we find  terms

A    B      C’

Taking complement of these two

A’     B’     C

Now **sum** up them

(A’ + B’ + C)

From**green** group we find  terms

B         C

Taking complement of these two terms

B’         C’

Now sum up them

(B’+C’)

From **brown**group we find terms

A’ B’ C’

Taking complement of these two

A B C

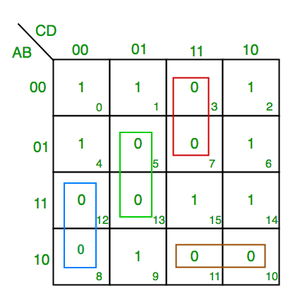
Now **sum** up them

(A + B + C)

We will take product of these three terms :**Final expression (A’ + B’ + C) (B’ + C’) (A + B + C)**

**2. K-map of  4 variables-**

F(A,B,C,D)=π(3,5,7,8,10,11,12,13)



From **green** group we find  terms

C’     D     B

Taking their complement and summing them

(C+D’+B’)

From **red** group we find terms

C     D    A’

Taking their complement and summing them

(C’+D’+A)

From **blue**  group we find  terms

A     C’     D’

Taking their complement and summing them

(A’+C+D)

From **brown** group we find  terms

A    B’    C

Taking their complement and summing them

(A’+B+C’)

Finally we express these as product –**(C+D’+B’).(C’+D’+A).(A’+C+D).(A’+B+C’)**

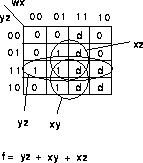
## **Designing with Don't-Care Values**

**In some situations, we don't care about the value of a logic function.  
For example, if we use wxyz to represent a number from 0 to 9, we need not worry about the function value produced for wxyz = 10...15.  
For these situations, the function can be assigned an output in order to make the resulting circuit as simple as possible.  
Suppose we wish to implement the function**

**f(wxyz)=Sum(3,5,6,7)  
and we have the don't-care condition of**

**d=Sum(10,11,12,13,14,15).**

**The sum-of-products implementation:**

****