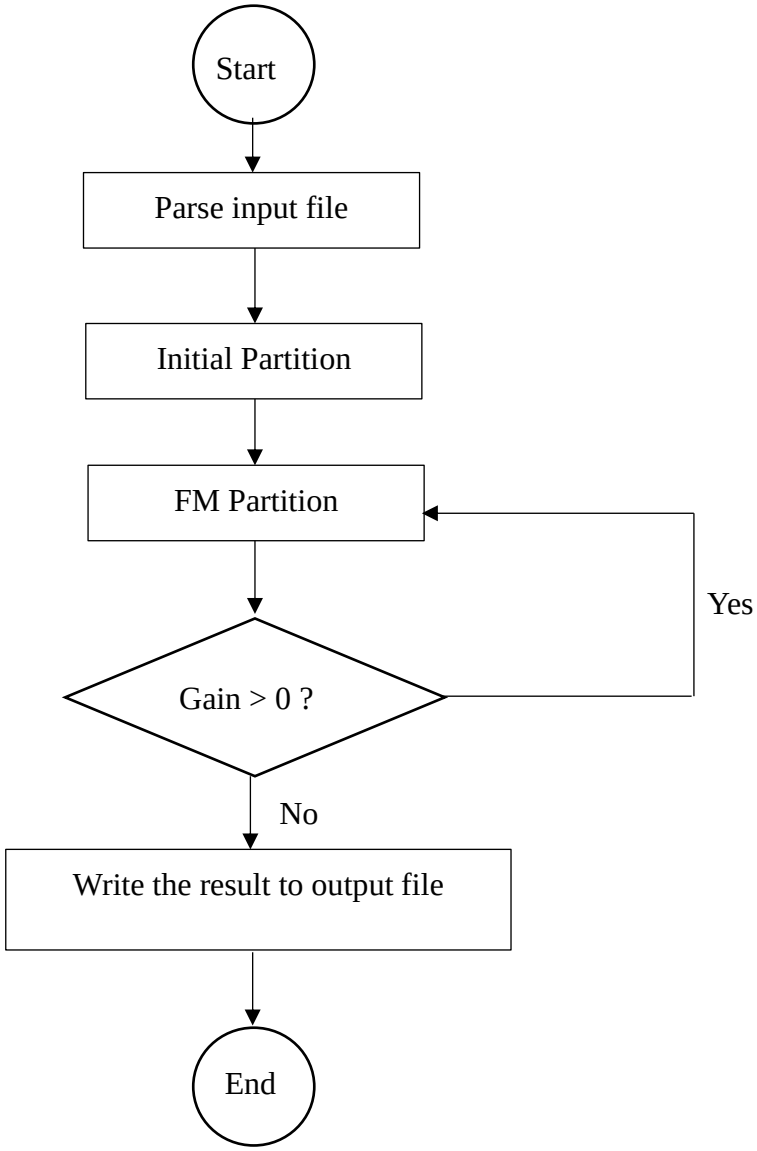
- **Implementation**

1. Algorithm Flow



According to the flow chart above

(1) After parsing the input file, all cells will be divided into two partitions (i.e. A and B) 。

(2) After initial partition, the program will get into a while loop and do FM partition algorithm until the overall gain (Gain) is **less or equal to zero**.

(3) Finally, the result will be written to the output file in correct format.

2. Initial Partition

The issue of initial partition is important since once the initial partition is decided, the result is determined as well. Therefore, instead of randomly partition, my program deal with this problem in the following steps.

(1) Sort the cells by **the number of nets connected**.
(2) Divide the cells into two partitions, A and B. Make sure that every cell in A has more or equal number of nets connected than every cell in B.

- **Data Structure**

In my program, there are five classes:

I. class Cell
II. class Net
III. class BucketList
IV. class CellV
V. class NetManager

1. Cells access

I use two vectors to manage the cells and nets. It is much more important to access a particular cell than a particular net. In order to quickly access a specific cell, I attempt to access it by **index**. However, the cells' name does not contain any index information. Thus, I apply **HashSet** to do so by means of a wrapper class CellV in the process of parsing. The key is the name of a cell, and the value is the assigned index of the cell. By this key-value pair, we can quickly get the index of the cell corresponding to its name.

```
class NetManager
{
private:
     vector<Cell*>     _cellList;
     vector<Net*>      _netList;
     HashSet<CellV>    _hash;

     int           _Gain;   // overall gain
     unsigned   _NumA; // # of partition A
     unsigned   _NumB; // # of partition B
};
```

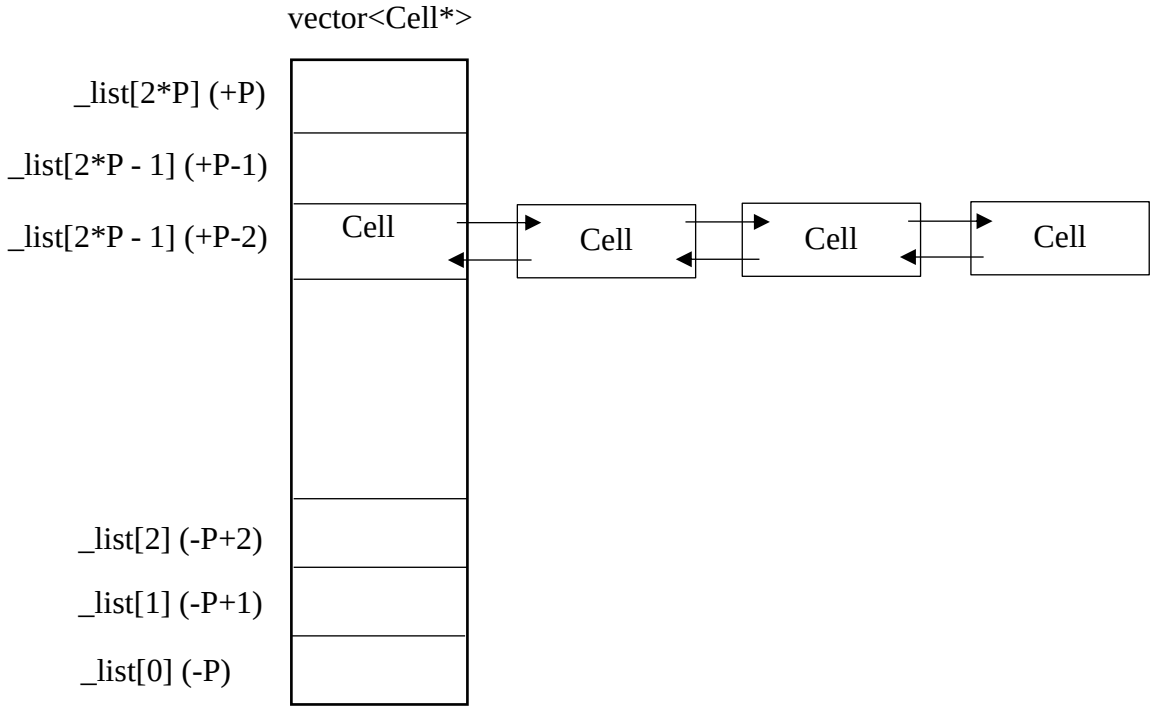
```
class CellV
{
public:
    size_t operator () () const     // hash function
    {
        size_t k = 0;
        unsigned n = (_name.length()>4 ? 4 : _name.length());
        for(int i=0; i<n; ++i)
            k ^= (size_t)_name[i] <<(i*8);
        return k;

    bool operator == (const CellV& n) {
        return (_name == n.getName());
    }

private:
    string    _name;
    int       _no
};
```

2. BucketList structure

This data structure helps FM partition algorithm has linear complexity. The figure below shows a bucket list implemented in my program. A vector<Cell*> with size (2*_maxPinNum + 1), where _maxPinNum is the maximum possible gain of a cell, is constructed, and the index i corresponds to the specific gain (i - _maxPinNum).



```
class BucketList
{
    void deleteC(Cell* c, int g);    // delete cell c whose gain is g
    void insertC(Cell*);             // insert a cell
private:
    vector<Cell*>   _list;           // bucketlist
    unsigned        _maxPinNum;      // record the max pin number
                                     // among all cells
    int             _maxGain;        // point to the max gain index
```

```
class Cell
{
private:
    vector<Net*>   _nets;        // nets connected to this cell
    string         _name;        // cell name
    int            _no;          // cell number
    int            _gain;        // gain of this cell
    BucketList*    _part;        // which part it belongs to
    Cell*          _front;       // used in double link list
    Cell*          _back;        // used in double link list
}
```

```
class Net
{
private:
    vector<Cell*>   _cells;      // cells in this net
    string          _name;       // net name
    unsigned        _numA;       // # of cells in this net in Partition A
    unsigned        _numB;       // # of cells in this net in Partition B
}
```