```
def dominance(pi_list):
    global mintermls, origin, counting, minterm
    counting = {}
    mintermls = {}
    minterm = set()
    for s in pi_list:
        origin = s
        mintermls[origin] = []
        rec(s)
    print(mintermls)
    epi = []
    for s in mintermls:
        for m in mintermls[s]:
            if(counting[m] == 1):
                pi_list.remove(s)
                epi.append(s)
                break
    for p in epi:
        for m in mintermls[p]:
            if m in minterm:
                minterm.remove(m)
        del mintermls[p]
    print(rowDominance(minterm))
    print(colDominance(minterm))

dominance(['00-', '0-0', '11-', '1-1', '-01', '-10'])
def rec(string):
    if(string.count("-") == 0):
        s = "0b" + string
        s = int(s, 2)
        mintermls[origin].append(s)
        minterm.add(s)
        if(s in counting):
            counting[s] += 1
        else:
            counting[s] = 1
        return
    for i in range(len(string)):
        if (string[i] == "-"):
            string = string[:i] + "1" + string[i+1:]
            rec(string)
            string = string[:i] + "0" + string[i+1:]
            rec(string)
            break
```

우선 dominance라는 큰 함수가 존재합니다.
다른 함수에서도 사용이 필요한 변수들을 전역변수로 언하고, epi를 먼저 체크하는데,
재귀함수로 구현된 rec함수를 통해 각 pi에 대응되는 minterm이 몇 번 사용됐는지 체크하고,

만약 1번 체크됐다면 그 PI는 EPI기 때문에 PI리스트에서 EPI를 제거하고, EPI 리스트에 추가합니다. 이 EPI리스트에 있는 값들은 필요없으니 mintermls에서 삭제하고 rd, cd함수를 실행하게 됩니다.

```
def rowDominance(minterm):
    global useful
    useful = {}
    for s in mintermls.keys():
        useful[s] = []
        for k in mintermls[s]:
            if k in minterm:
                useful[s].append(k)
    ls = list(useful.keys())
    dom = set()
    for i in range(len(ls)-1):
        if len(useful[ls[i]]) == 0:
            dom.add(ls[i])
            continue
        for j in range(i+1, len(ls)):
            a = set(useful[ls[i]])
            b = set(useful[ls[j]])
            if len(useful[ls[j]]) == 0:
                dom.add(ls[j])
                continue
            if len(a) == len(b):
                continue
            elif len(a) > len(b):
                if b.issubset(a):
                    dom.add(ls[j])
            else:
                if a.issubset(b):
                    dom.add(ls[i])
    for d in dom:
        del useful[d]
    return rowCheck(list(useful.keys()))
```

우선 useful이라는 딕셔너리를 전역변수로 선언하는데, 이는 check를 위해 선언하였고 먼저 mintermls라는 딕셔너리 {“00-0” : [1, 2]) 이런 구조로 되어있는 딕셔너리를 통해 useful에 값을 넣는데, 만약 아까 epi에 대응되는 minterm을 제거한 minterm이라는 변수에 있는 리스트에 값이 없다면 useful에 값을 넣지 않습니다. 즉 14가 필요 없으면 “111-” : [14.15]여도 “111-” :[15] 로 들어가게 됩니다. dom이라는 리스트는 지배당하는 값을 넣는 set이고, 이중 for문을 통해 모든 pi를 2개씩 비교해보았습니다. 지배당하는 관계를 찾기 위해 부분집합을 이용했는데, 만약 a가 b의 부분집합이면 a는 b에 지배당하는 것이고, 이때 dom에 해당 minterm을 추가하였고, 다 탐색한 뒤 지배당하는 pi는 useful에서 제거합니다. 마지막으로 결과를 return할 때 check함수를 구현했습니다.

```python
def rowCheck(minterm):
    for i in range(len(minterm) - 1):
        for j in range(i+1, len(minterm)):
            a = set(useful[minterm[i]])
            b = set(useful[minterm[j]])
            if len(a) == len(b):
                continue
            elif len(a) > len(b):
                if b.issubset(a):
                    return(rowDominance(minterm))
            else:
                if a.issubset(b):
                    return(rowDominance(minterm))
    return minterm
```

만약 남아있는 pi에 대응되는 minterm들이 {1}, {1,2} {1, 2, 3} 이런 식으로 남아있을 수 있습니다. 따라서 서로 지배되는 관계가 하나라도 발견된다면 rowdominance 함수를 통해 지배당하는 관계를 처리하고, for문을 무사히 마치면 재귀의 종료로 결과값을 반환하며 마무리합니다. 재귀적으로 구현한 이유가 또 지배관계가 생길수 있기 때문엔 확실하게 rowdominance 함수로 넘기는 것입니다.

```python
def colDominance(minterm):
    global coldic
    coldic = {}
    for s in mintermls.keys():
        for k in mintermls[s]:
            if k not in minterm:
                continue
            if k in coldic:
                coldic[k].add(s)
            else:
                coldic[k] = set()
                coldic[k].add(s)
    ls = list(coldic.keys())
    dom = set()
    for i in range(len(ls)-1):
        if len(coldic[ls[i]]) == 0:
            dom.add(ls[i])
            continue
        for j in range(i+1, len(ls)):
            a = set(coldic[ls[i]])
            b = set(coldic[ls[j]])
            if len(coldic[ls[j]]) == 0:
                dom.add(ls[j])
                continue
            if len(a) == len(b):
                continue
            elif len(a) > len(b):
                if b.issubset(a):
                    dom.add(ls[j])
            else:
                if a.issubset(b):
                    dom.add(ls[i])
    for d in dom:
        del coldic[d]
    return colCheck(list(coldic.keys()))
```

column dominance도 비슷한 매커니즘인데, 다른 점은 아깐 useful dictionary가 있었다면 이젠 coldic dictionary가 있습니다. 즉 { 2 :[00-0, 00-1 } 이런 구조로 되어있고 부분집합을 이용해 지배관계를 해결합니다.