1. 设计搜索算法求解最大团问题，输入是一个图，输出是这个图最大的全连通子图

以每个点作为出发点，寻找与该点连通的所有点和边。由于最大连通图的性质，如果一个点在之前寻找到的最大连通图里，则以两个点为出发点找到的最大连通图相同，可以不再进行遍历。

```
Input: G
Output: 该图的最大连通图

V = G.V
E = G.E
M = []
while !empty(V)
    从V中删去任意元素i
    Mi = new Map(i)
    P = E中与i相邻的边
    while !empty(P)
        if ∃(u,v)∈P s.t. u∉V && v∈V
            V.delete(v)
            E.delete(u,v)
            P.delete(u,v)
            Mi.addV(v)
            Mi.addE(u,v)
            P += E中与v相邻的边
        else
            for x in P
                Mi.addE(x)
    M += Mi
return max(M)
```

2. 设计搜索算法求解最大独立子集合问题，输入一个图，输入这个图的最大顶点集合，满足该集合中任意两个顶点之间都不存在边

由最大独立子集合的性质可知，如果用增加点的方法来搜索，必须搜索完所有的组合才能找到最大值，但是通过删减边的方法，由于删的约多剩的点越少，因此可以在找到第一个解之后进行剪枝.

```
Input: G
Output: 最大独立子集合

best = ∞
Stack = [G]

while !empty(Stack)
    (X,c) = Stack.pop()
    if c >= best // 剪枝
        continue
    if !empty(X.Edges)
        for (u,v) in X.Edges // 此处省略过滤掉相同的节点
            Stack.push(G \ u, c+1)
            Stack.push(G \ v, c+1)
    else // 说明已经找到一个独立子集合
        Result = X.Nodes
        best = c // 分支界限
return Result
```

3. 设计搜索算法求解有向图图的强连通分量问题，输入一个图，输出这个图的所有强连通分量

可以找一棵有向的生成树，搜索过程中发现的所有回边都意味着一个强连通分量。由有向图的性质可知在之前搜索到的树中的点无需再作为顶点进行搜索。

```
Input: G
Output: 所有的强连通分量

M = []
P = G.Nodes

if !empty(P)
    从P中删除任意元素i
    Root = new Tree(i)
    Stack = [Root]
    while !empty(Stack)
        T = Stack
        i = T.root
        for (i,v) in G.Edges
            if v in T的所有祖先
                形成了一个环，即找到一个强连通分量
            else
                P.delete(v) // 由于v在树中，无需再作为根节点进行搜索
                T.addChild(v)
                Stack.push(v)
```

4. 设计搜索算法求解子图匹配问题，输入:图 G 和图 P，输出:图 G 是否包含和 P 同构的子图

暴力枚举G的所有子图，并和P一一比较。点和边的数量与P不一致的子图可以跳过。

5. 设计搜索算法求解 0-1 背包问题，即给定一个容量为 C 的背包和 n 个物品，其中每个物品 i 的重量为 wi, 价格为 vi，要求物品的重量之和小于 C，且价格之和最大

采用分支界限法的DFS来进行搜索

```
Input: C, n, W, V
Output: 最佳装包向量

best = 0
bestX = null
Stack = [[],C,0]
while !empty(Stack)
    (X,w,v) = Stack.pop()
    if w <= 0
        continue
    if v > best
        best = v
        bestX = X
    if |X| < n
        Stack.push(X+1,w+W[|X|],v+V[|X|])
        Stack.push(X+0,w,v)
return bestX
```

6. 证明 KMP 算法的正确性

(1). KMP找到的子串都是匹配的

证明: 假设某次匹配时KMP跳过了π个字符，显然P[π..]是匹配的，而根据前缀表的计算规则，P[π..2π]和P[1..π]相同，而上一次比较时已经比较了P[1..π]的部分，因此P[1..π]也是匹配的，所以KMP找出的子串都是匹配的。

(2). 所有匹配的子串KMP都能找到

证明: 只需证明KMP跳过字符时，不会导致匹配子串被跳过即可。由KMP的规则可知，发生跳过π个字符时P[π+1]和α是不同的，而由前缀表的计算方法可知在P[1..n]不会和α的前n个字符匹配(n<π)，所以KMP算法不会跳过任何可能匹配的子串。

综合(1),(2)可知KMP是正确的

7. 扩展 Rabin-karp 算法，设计一个矩阵匹配算法，即输入矩阵 A 和矩阵 B，找到矩阵 A 中和矩阵 B 匹配的子阵。

使用矩阵所有元素的和与某个大素数p的模作为指纹即可，遍历方法由线性遍历改为二维矩阵的遍历。

更新指纹时如果是向下移动，则减去顶部一行并加上下方一行，再次计算模。向右移动同理。

8. 给定一棵树 T，在每条边上有一个标签，该标签包含一个或多个字符，给定一个模式 P，T 中子路径的标签定义为该路径上所有边上标签依次相连得到的字符串，问题是找到所有从根出发路径的子路径，其标签和 P 匹配。要求运行时间和树中所有标签的字符数与 P 长度之和相等。

由于要求与树上所有标签字符数是线性的，因此不能直接遍历所有子串进行匹配，那样会重复对前缀进行比较。只需要记录回溯点，在匹配失败的时候从回溯点开始继续搜索即可避免重复匹配前缀。

下面的伪代码使用递归方法遍历树，每次匹配若成功则将rest剩下的部分传给下一个节点。 为了方便起见，假设每个节点的值是它父节点到它的边上的标签

```
Input: T, P
Output: 打印所有与P匹配的路径

match(a,b):
在O(|a|+|b|)时间内匹配a的前|b|个字符是否与b相同

find(T, rest, Path):
if rest.length == 0 // rest为空表明上一次匹配已成功
    return print(Path)
if match(rest, T)
    for i in T.children
        find(i, rest[|T|..], Path + T) // 在所有子节点中查找剩下的部分是否匹配

main():
for i in T // T中的所有节点
    find(i, P, [i])
```

9. 给出一个例子，令 Robin-karp 算法的时间复杂度为 O(mn)其中 m 是模式的长度，n 是字符串长度，要说明例子并证明这个时间复杂度。

输入串任何位置都与模式串不同，但指纹相同。比如输入串为"1111111111111"，模式串为"13"，指纹算法为sum mod 2，对于输入串的任何子串，其指纹都是0，与模式串相同，因此对于每个位置，都必须要进行完整的字符串比较，复杂度为O(mn)

10. 给出一个例子，令 Robin-karp 算法的时间复杂度为 O(mn)其中 m 是模式的长度，n 是字符串长度，要说明例子并证明这个时间复杂度。

和第9题一样