回溯
```
res = []
path = []

def dfs(参数) :
    if 满足递归结束 :
        res.append(list(path))
        return
    # 递归方向
    for (xxxx):
        path.append(val)
        dfs()
        path.pop()
```

记忆化搜索
```
from functools import cache
@cache #缓存，避免重复运算
def dfs(i)->int:
  if 终止: return 0 #具体返回什么值要看题目的含义
  cnt = 0
    for 递归方向:
        cnt += dfs(xxx) #如果是计数，一般是叠加，也有可能是取最大或者最小
    return cnt
```

01背包
```
n, C; #n个物品， C表示背包容量
v,  w; #v[i]表示第i个物品的价格/体积     w[i]表示第i个物品的价值
dp = [[0 for _ in range(C+1)] for _ in range(n+1)] #容器规模
#初始化 dp[0][j] j∈[0,C]
for i in range(1, n+1):
    for j in range(C+1):
        if j>=v[i-1]: dp[i][j] = max(dp[i-1][j], dp[i-1][j-v[i-1]]+W[i-1])
        else: dp[i][j] = dp[i-1][j]
return dp[n][C]

n, C; //n个物品， C表示背包容量
v,  w; //v[i]表示第i个物品的价格/体积     w[i]表示第i个物品的价值
dp = [0 for _ in range(C+1)] //容器规模
//初始化 dp[j] j∈[0,C]
for i in range(1, n+1):
    for j in range(C,v[i-1]-1,-1):
        dp[j] = max(dp[j], dp[j-v[i-1]]+w[i-1])
return dp[C]
```

完全背包
```
n, C; #n个物品， C表示背包容量
v,  w; #v[i]表示第i个物品的价格/体积     w[i]表示第i个物品的价值
dp = [[0 for _ in range(C+1)] for _ in range(n+1)] #容器规模
#初始化 dp[0][j] j∈[0,C]
for i in range(1, n+1):
    for j in range(C+1):
        if j>=v[i-1]: dp[i][j] = max(dp[i-1][j], dp[i][j-v[i-1]]+W[i-1])
        else: dp[i][j] = dp[i-1][j]
return dp[n][C]

n, C; //n个物品， C表示背包容量
v,  w; //v[i]表示第i个物品的价格/体积     w[i]表示第i个物品的价值
dp = [0 for _ in range(n+1)] //容器规模
```

```
//初始化 dp[j] j∈[0,C]
for i in range(1, n+1):
    for j in range(C+1):
        dp[j] = max(dp[j], dp[j-v[i-1]]+w[i-1])
return dp[C]
```

LIS

```
def lengthOfLIS(self, nums: List[int]) -> int:
    dp = [1 for _ in range(len(nums))]
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

def lengthOfLIS(self, nums: List[int]) -> int:
    ls = []
    for num in nums:
        x = bisect_left(ls, num)
        if x == len(ls):
            ls.append(num)
        else:
            ls[x] = num
    print(ls)
    return len(ls)
```

lcs

```
def longestCommonSubsequence(self, text1: str, text2: str) -> int:
    len1, len2 = len(text1), len(text2)
    dp = [[0] *(len2 + 1) for _ in range(len1 + 1)]
    for i in range(1, len1 + 1):
        for j in range(1, len2 + 1):
            dp[i][j] = dp[i - 1][j - 1] + 1 if text1[i - 1] == text2[j - 1] else
max(dp[i-1][j],dp[i][j-1])

    return dp[len1][len2]
```

lp

```
def longestPalindrome(self, s: str) -> str:
    n = len(s)
    dp = [[False] * n for _ in range(n)]
    maxlen = 0
    for j in range(n):
        for i in range(j + 1):
            if i == j:
                dp[i][j] = True
            elif i + 1 == j:
                dp[i][j] = s[i] == s[i + 1]
            else:
                dp[i][j] = s[i] == s[j] and dp[i + 1][j - 1]
            if dp[i][j] and j - i + 1 >= maxlen:
                maxlen = j - i + 1

    return mxlen

// 区间划分为[l,mid] 和 [mid+1,r], 选择此模板
```

```
int bsec1(int l, int r)
{
    while (l < r)
    {
        int mid = (l + r)/2;
        if (check(mid)) r = mid;
        else l = mid +  1;
    }
    return r;
}

// 区间划分为[l,mid-1] 和 [mid,r], 选择此模板
int bsec2(int l, int r)
{
    while (l < r)
    {
        int mid = ( l + r + 1 ) /2;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return r;
}
```

单调栈

```
stack = []
for i in range(len(nums)):
    while stack and nums[stack[-1]] < nums[i]:
        p = stack.pop()
        # 此时说明 nums[top]的下一个更大的元素为nums[i]
    stack.append(i)
```

单调队列

```
res = [0] * (len(nums) - k + 1)
queue = deque()
for i in range(len(nums)):
  if queue and i - k + 1 > queue[0]:
    queue.popleft()
  while queue and nums[queue[-1]] < nums[i]:
    queue.pop()
  queue.append(i)
  if i >= k - 1:
    res[i - k + 1] = nums[queue[0]]
return res
```

graph

```
graph = [[0 for _ in range(n)] for _ in range(n)]
for i in range(n):
  a,b,w = map(int, input().split())
  graph[a][b] = graph[b][a] = w  # 如果是有向图则不需要建立双向边

graph = defaultdict(list)
for i in range(m):
  a,b,w = map(int, input().split())
    graph[a].append([b,w])
    graph[b].append([a,w])
```

```
graph
vst = [False for _ in range(n)]
def dfs(node):
    for next,weight in graph[node]:
        if not vst[next]:
            vst[next] = True
            dfs(next)
            # 如果需要回溯的话 , vst[next] = false;
```

```
graph
vst = [False for _ in range(n)]
def bfs():
    q = deque()
    q.append(start)
    vst[start] = True
    while q:
        node = q.popleft()
        for next,weight in graph[node]:
            if not vst[next]:
                vst[next] = True
                q.append(next)
```

拓扑排序

```
graph = [[] for _ in range(n)]
indegre = [0] * n#存储每个节点的入度
q = deque()
for i in range(n):
    if indegre[i]==0: q.append(i)

while q:
    node = q.popleft()
    for next in graph[node]:
        indegre[next]-=1
        if indegre[next] == 0: q.append(next)
```

```
dsu

fa = [i for i in range(n)]

#找到x的根节点
def find(x):
    if x == fa[x]: return x
    fa[x] = find(fa[x])
    return fa[x]

#合并两个节点
def union(x,y):
    fa[find(x)] = find(y)
```

最小生成树

```
def kruskal(edges:List[List[int]], n:int,m:int) -> int:
    edges.sort(key=lambda x : x[2])
  fa = [i for i in range(n)]
```

```python
    #找到x的根节点
    def find(x):
        if x == fa[x]: return x
        fa[x] = find(fa[x])
        return fa[x]

    #合并两个节点
    def union(x,y):
        fa[find(x)] = find(y)

    ans = 0
    for a,b,w in edges:
        if find(a) != find(b):
            union(a,b)
            ans += w
    return ans

def prim(graph: List[List[int]], n: int) -> int:
  dis = [inf for _ in range(n)]
  vst = [False for _ in range(n)]
    res = 0
    for i in range(n):
        min_index = -1
        for j in range(n):
            if not vst[j] and (min_index == -1 or dis[min_index] > dis[j]) min_index
= j
        if i != 0: res += dis[min_index]
        vst[min_index] = True
        for j in range(n): dis[j] = min(dis[j], graph[min_index][j])
    return res
```

最短路 权为1

```python
def bfs(st: int, target: int, n: int, graph: dict) -> int:
    q = deque()
    vst = [False for _ in range(n)]
    q.append(st)
    vst[st] = True
    cnt = 0
    while q:
        size = len(q)
        for _ in range(size):
            node = q.popleft()
            if node == target: return cnt
            for next in graph[node]:
                if vst[next]: continue
                vst[next] = True
                q.append(next)
        cnt+=1
    return -1

#dij
def dijkstra(st: int, n: int, graph: List[List[int]]):
    dis = [inf for _ in range(n)]
    vst = [False for _ in range(n)]
    dis[st] = 0
    for i in range(n):
```

```python
        x = -1
        for y in range(n):
            if not vst[y] and (x==-1 or dis[y] < dis[x]): x = y
        vst[x] = True
        for y in range(n):
            dis[y] = min(dis[y], dis[x] + graph[x][y])

def dijkstra(st: int, n: int, graph: dict):
    dis = [inf for _ in range(n)]
    vst = [False for _ in range(n)]
    dis[st] = 0
    h = []
    heapq.heappush(h, [0, st])
    while h:
        d,u = heapq.heappop(h)
        if vst[u]: continue
        vst[u] = True
        for v,w in graph[u]:
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                heapq.heappush(h, [dis[v], v])
```

多源

```python
dp = [[graph[i][j] for i in range(n)] for j in range(n)]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dp[i][j]  = min(dp[i][j], dp[i][k] + dp[k][j])
```

scc

```python
def tarjan(n, adj):
    dfn = [0] * n  # 访问节点时的时间戳
    low = [0] * n  # 节点可达的最低时间戳
    in_stack = [False] * n  # 布尔数组，用于检查节点是否在栈中
    stack = []  # 用于存储节点的栈
    dfncnt = 1  # 用于给访问的节点分配唯一编号的计数器
    scc = [0] * n  # 存储每个节点所属的强连通分量(SCC)编号的数组
    sc = 0  # 找到的SCC数量的计数器
    sz = [0] * n  # 每个SCC的大小

    def dfs(u):
        nonlocal dfncnt, sc
        dfn[u] = low[u] = dfncnt  # 给节点分配时间戳
        dfncnt += 1
        stack.append(u)  # 将当前节点加入栈
        in_stack[u] = True  # 标记节点为在栈中

        for v in adj[u]:  # 遍历每个相邻节点
            if dfn[v] == 0:  # 如果节点未被访问
                dfs(v)
                low[u] = min(low[u], low[v])  # 更新当前节点的最低可达时间戳
            elif in_stack[v]:  # 如果相邻节点在栈中
                low[u] = min(low[u], dfn[v])  # 更新当前节点的最低可达时间戳，仅包括在栈中的
节点

        # 如果当前节点是SCC的根节点
```

```
        if dfn[u] == low[u]:
            sc += 1
            while True:
                v = stack.pop()  # 弹出节点
                in_stack[v] = False  # 标记节点不在栈中
                scc[v] = sc  # 分配SCC编号
                sz[sc - 1] += 1  # 增加SCC大小
                if v == u:  # 如果回到根节点，结束循环
                    break

    # 从每个未访问的节点运行DFS
    for i in range(n):
        if dfn[i] == 0:
            dfs(i)

    return scc, sz  # 返回SCC编号和每个SCC的大小

#python语法糖可以求前缀和
pres = list(accumulate(a,initial=0))
# 二维
matrix # 原二维矩阵
m, n = len(matrix), len(matrix[0])
pre = [[0] * (n + 1) for _ in range(m + 1)]
for i in range(1, m + 1):
    for j in range(1, n + 1):
        pre[i][j] = pre[i - 1][j] + pre[i][j - 1] - pre[i - 1][j - 1] + matrix[i - 1]
[j - 1]

# 查询子矩阵的和 [x1,y1] [x2,y2]表示子矩阵的左上和右下两个顶点
sum_ = pre[x2 + 1][y2 + 1] - pre[x1][y2 + 1] - pre[x2 + 1][y1] + pre[x1][y1];
```

差分

```
nums = [1,3,2,4,5]
n = len(nums)
diff = [1 for _ in range(n)]
for i in range(1, n):
  diff[i] = nums[i] - nums[i-1]

#将区间[l,r]的元素都加上v
def update(l, r, v):
    diff[l] += v
    if r+1 < n:
      diff[r+1] -= v


'''多次调用update后，对diff数组求前缀和可以得出 多次修改后的数组'''
res = [0 for _ in range(n)]
res[0] = diff[0]
for i in range(1,n):
  res[i] += res[i-1] + diff[i]
```

二维差分

```
'''二维矩阵依然可以进行差分运算'''

n, m = 0,0 #行和列
a = [[0 for _ in range(m+1)] for _ in range(n+1)] #原数组
```

```python
diff = [[0 for _ in range(m+1)] for _ in range(n+1)]
def insert(x1, y1,x2,y2,d):
     diff[x1][y1] += d
     diff[x2+1][y1] -= d
     diff[x1][y2+1] -= d
     diff[x2+1][y2+1] += d

#差分数组初始化
for i in range(1, n+1):
    for j in range(1,m+1):
        insert(i,j,i,j,a[i][j])

q = 0 #修改次数

while q:
    q-=1
    x1,y1,x2,y2,d = 0,0,0,0,0 #对于矩阵的值增加d
    insert(x1,y1,x2,y2,d)

#还原数组
for i in range(1,n+1):
    for j in range(1,m+1):
        a[i][j] = a[i-1][j] + a[i][j-1] -a[i-1][j-1]+ diff[i][j]

print(a)
```

树状数组

```python
class BIT:
    def __init__(self, n):
        self.MXN = n+1
        self.tree = [0 for _ in range(self.MXN)]

    def lowbit(self,x):
        return x & (-x)

    # 下标为index的元素新增x
    def update(self,index, x):
        i = index+1 #树状数组的下标从1开始
        while i < self.MXN:
            self.tree[i] += x
            i += self.lowbit(i)

    # 查询前n项总和
    def queryPre(self,n):
        ans = 0
        while n:
            ans += self.tree[n]
            n -= self.lowbit(n)
        return ans

    # 查询区间[a,b]的和
    def query(self,a, b):
        return self.queryPre(b+1) - self.queryPre(a)
```

线段树

```python
MXN = int(1e5 + 5)
n = int(input())#数组长度
A = [0] + [int(c) for c in input().split(" ")]
tree = [0 for _ in range(MXN * 4)]
mark = [0 for _ in range(MXN * 4)]

def push_down(p, len):
    mark[p * 2] += mark[p]
    mark[p * 2 + 1] += mark[p]
    tree[p * 2] += mark[p] * (len - len // 2)
    tree[p * 2 + 1] += mark[p] * (len // 2)
    mark[p] = 0

def build(l=1, r=n,p=1):
    if l==r: tree[p] = A[l]
    else:
        mid = (l+r) // 2
        build(l,mid,p*2)
        build(mid+1,r,p*2+1)
        tree[p] = tree[p*2] + tree[p*2 + 1]

def update(l,r,d,p=1,cl=1,cr=n):
    if cl > r or cr < l: return
    elif cl >= l and cr <= r:
        tree[p] += (cr - cl + 1) * d
        if cr > cl: mark[p] += d
    else:
        mid = (cl + cr) // 2
        push_down(p, cr-cl+1)
        update(l,r,d,p*2,cl,mid)
        update(l,r,d,p*2+1,mid+1,cr)
        tree[p] = tree[p*2] + tree[p*2+1]

def query(l,r,p=1,cl=1,cr=n):
    if cl > r or cr < l: return 0
    elif cl >= l and cr <= r: return tree[p]
    else:
        mid = (cl + cr) // 2
        push_down(p, cr-cl+1)
        return query(l,r,p*2,cl,mid) + query(l,r,p*2+1,mid+1,cr)

'''
1.输入数组A，注意下标从[1,n]。
2.调用update(l,r,d)函数，这里的l和r并不是下标。
3.调用query(l,r) 这里的l和r并不是下标
'''
```

素数

```python
maxCNt
primes = [] #存储了组后的素数
st = [False for _ in range(maxCNt)]
index = 0
for i in range(2, maxCNt):
   if not st[i]:
      primes.append(i)
      for j in range(i+i, maxCNt, i): st[j] = True
```

约数
```python
def get_divisors(n: int):
    res = []
    i = 1
    while i <= n//i:
        if n%i==0:
            res.append(i)
            if i!=n//i: res.append(n//i)
        i+=1
    res.sort()
    return res
```

快速幂
```python
def fast_pow(x, y, mod):
    res = 1
    while y > 0:
        if y % 2 == 1:
            res = (res * x) % mod
        x = (x * x) % mod
        y //= 2
    return res
```
离散化
```python
a = [] #原数组
as = sorted(list(set(a)))
LS = [bisect.bisect_left(as,a[i]) for i in range(n)]
#其中,LS[i]表示的就是a[i]对应的离散后的下标。

MOD = 10**9 + 7
# 使用费马小定理快速计算逆元（只适用于m是质数的情况）
def fast_inv(a, m=MOD):
    return pow(a, m-2, m)

# 计算a/b mod MOD的值
def compute(a, b):
    # 计算b的逆元
    b_inv = fast_inv(b)
    # 计算并返回结果
    return (a * b_inv) % MOD

# 示例
a = 2
b = 3
result = compute(a, b)
print(f"The result is: {result}")
```

Diophantine

是否有解：gcd(a,b) | C
化简：除以 gcd(a,b)
使用扩展欧几里得算法求特解
写出通解形式

```python
def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
```

```python
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y

def solve_diophantine(a, b, c):
    gcd, x0, y0 = extended_gcd(a, b)
    if c % gcd != 0:
        return None  # 无解
    else:
        # 化简方程
        a_prime = a // gcd
        b_prime = b // gcd
        c_prime = c // gcd
        # 特解
        x1 = x0 * c_prime
        y1 = y0 * c_prime
        # 通解
        return (x1, y1, b_prime, -a_prime)

# 示例
a = 56
b = 15
c = 1
solution = solve_diophantine(a, b, c)
if solution:
    x1, y1, b_prime, a_prime = solution
    print(f"特解: x = {x1}, y = {y1}")
    print(f"通解: x = {x1} + {b_prime}k, y = {y1} + {a_prime}k")
else:
    print("方程无解")
```

分组背包

```python
def group_knapsack(V, groups):
    """
    分组背包问题的 Python 实现

    :param V: 背包容量
    :param groups: 物品分组, 每组是一个列表, 列表中的每个元素是 (体积, 价值) 的元组
    :return: 最大价值
    """
    # 初始化 dp 数组, dp[j] 表示容量为 j 的背包的最大价值
    dp = [0] * (V + 1)

    # 遍历每一组物品
    for group in groups:
        # 遍历背包容量 (从大到小, 确保每组只选一个物品)
        for j in range(V, -1, -1):
            # 遍历组内的每个物品
            for v, w in group:
                if j >= v:  # 确保背包容量足够
                    dp[j] = max(dp[j], dp[j - v] + w)

    return dp[V]
```

```python
# 示例
if __name__ == "__main__":
    V = 10  # 背包容量
    groups = [
        [(2, 3), (3, 4)],  # 第一组物品
        [(4, 5), (5, 6)],  # 第二组物品
        [(1, 2), (2, 3), (3, 4)]  # 第三组物品
    ]

    result = group_knapsack(V, groups)
    print("最大价值:", result)
```

插头 dp

```python
def plug_dp(n, m):
    # 状态总数（根据具体问题确定）
    max_state = 1 << (m + 1)

    # dp[i][j] 表示第 i 行状态为 j 的方案数
    dp = [[0] * max_state for _ in range(n + 1)]

    # 初始化第一行
    dp[0][0] = 1

    for i in range(1, n + 1):
        for j in range(max_state):
            if dp[i - 1][j]:  # 如果上一行状态 j 有效
                # 枚举当前行的状态转移
                for k in range(max_state):
                    if is_valid_transition(j, k):  # 检查状态转移是否合法
                        dp[i][k] += dp[i - 1][j]

    # 统计最后一行的合法状态
    result = 0
    for j in range(max_state):
        if is_final_state(j):  # 检查状态是否满足终止条件
            result += dp[n][j]

    return result
```

状压 dp

```python
def state_compression_dp(n, graph):
    # 状态总数
    max_state = 1 << n

    # dp[mask][u] 表示在状态 mask 下，当前位于城市 u 的最短路径
    dp = [[float('inf')] * n for _ in range(max_state)]

    # 初始化：从起点 0 出发
    dp[1][0] = 0

    # 枚举所有状态
    for mask in range(max_state):
        for u in range(n):
            if dp[mask][u] == float('inf'):
                continue  # 无效状态
            # 枚举下一个城市 v
```

```
        for v in range(n):
            if not (mask & (1 << v)):  # 如果 v 未被访问
                new_mask = mask | (1 << v)  # 更新状态
                dp[new_mask][v] = min(dp[new_mask][v], dp[mask][u] + graph[u][v])

# 计算最终结果：回到起点 0
result = float('inf')
for u in range(1, n):
    result = min(result, dp[max_state - 1][u] + graph[u][0])

return result
```