Cheating sheet

2300011505 钟明衡

动态规划（dp）

理解dp数组每一项的含义，然后写出递推公式。

以01背包为例，$dp[i][j]$表示从编号0到i中的物品里面选，放进容量为j的背包中，最大的总价值。此时有两种选择：是否加入i物品，不加入则和$dp[i-1][j]$一致，加入则为$dp[i-1][j-weight[i]]+value[i]\ \ (j\geq weight[i])$，取最大的，得到递推公式：

$$dp[i][j]=max(dp[i-1][j],\ dp[i-1][j-weight[i]]+value[i])\ \ (j\geq weight[i])$$

```
dp = [[0]*(m+1) for _ in range(n)]
for i in range(n):
    for j in range(0, m+1):
        if j >= weight[i]:
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]]+value[i])
        else:
            dp[i][j] = dp[i-1][j]
print(dp[-1][-1])
```

更简洁地，可以用一维，$dp[i]$表示背包容量为i时，最大的总价值。反过来遍历是为了保证一件物品最多选一次。

```
dp = [0]*(m+1)
for i in range(n):
    for j in range(m, weight[i]-1, -1):
        dp[j] = max(dp[j], dp[j-weight[i]]+value[i])
print(dp[-1])
```

多维也类似，要找到一个方法来表示当前状态，并且根据状态之间的转换来不断更新。

类

以“分数”类为例，说明类的写法。双下划线是python特有的魔术写法。

```
from math import gcd


class fraction:
    def __init__(self, a, b):
        g = gcd(a, b)
        if a*b >= 0:
            self.top = abs(a)//g
            self.bottom = abs(b)//g
        else:
```

```python
            self.top = -abs(a)//g
            self.bottom = abs(b)//g

    def __str__(self):  # 转化为字符串
        return '%d/%d' % (self.top, self.bottom)

    def __add__(self, other):  # 加法
        e = self.top*other.bottom+self.bottom*other.top
        f = self.bottom*other.bottom
        return fraction(e, f)


a, b, c, d = map(int, input().split())
print(fraction(a, b)+fraction(c, d))
```

另一个例子是文件夹

```python
class File:
    def __init__(self):
        self.name = ''
        self.depth = 0
        self.nameset = set()
        self.dic = {}

    def __str__(self):  # 输出格式，递归思想
        output = ''
        if self.name:
            output += ' '*self.depth+self.name+'\n'
        for file in sorted(list(self.nameset)):
            output += str(self.dic.get(file))
        return output


def build(parent, idx, s):  # 构建文件列表
    if s[idx] not in parent.nameset:
        new = File()
        new.name = s[idx]
        new.depth = idx
        parent.nameset.add(s[idx])
        parent.dic[s[idx]] = new
    if idx < len(s)-1:
        build(parent.dic[s[idx]], idx+1, s)


main = File()
for _ in range(int(input())):
    s = input().split('\\')
    build(main, 0, s)
print(str(main), end='')
```

栈

只能在尾部加减元素，可以用来辅助解决很多问题，比如括号匹配、表达式运算等。一般用一个列表来实现。

先入栈的先出栈。

单调栈

$next[i]$代表数列中元素i之后第一个大于$l[i]$的元素的下标，若不存在，则$next[i] = 0$。用一个栈$stack$来存储当前未找到更大元素的值和下标，每次在l中查找一个新的元素，就从栈顶开始比较，将更小的元素都弹出，最后再将这个新元素压入。很明显，$stack$里的元素一定是递减的，所以被叫做单调栈。

```python
n = int(input())
stack = []
next = [0]*n
for i, a in enumerate(map(int, input().split())):
    while stack and stack[-1][0] < a:
        next[stack.pop()[1]] = i+1
    stack.append((a, i))
print(*stack)
```

二分查找

只要能明确知道，现在找到的数是大了还是小了，就可以用二分查找法，能获得$O(\log n)$的复杂度。

有自带的二分查找，bisect_left和bisect_right区别如下：对于一系列相同值，left输出最左边索引，right输出最右边索引。如果不存在查找的相同值，则二者的结果相同，均为将这个查找值插入表中，使得顺序不被打乱，此时查找值的位置。

```python
from bisect import bisect_left, bisect_right

l = [1, 1, 2, 2, 2, 3, 4, 6, 7]
#    0  1  2  3  4  5  6  7  8

print(bisect_left(l, 2))   # 2
print(bisect_right(l, 2))  # 5
print(bisect_left(l, 5))   # 7
print(bisect_right(l, 5))  # 7
print(bisect_left(l, 0))   # 0
print(bisect_right(l, 0))  # 0
print(bisect_left(l, 8))   # 9
print(bisect_right(l, 8))  # 9
```

用insort_left/insort_right可以把元素插入到查找到的位置。

也可以手写，一定要注意判断条件。

贪心综述

贪心策略可以概括为“稳赚不亏”，“稳赚”指保证总是找到当前的最优情况，“不亏”指保留的可能情况不会比之前更糟。

比如，之前的dp就是一种“稳赚”，总是找到当前收益最大化的情况，这样直到最终情况也是最好的。

在后面的bfs中，可能会用到一种“不亏”，即如果路程和花费有一项比之前低，就认为是合法路径。

递归、分治和归并

递归，说白了就是在函数里面调用自己，是一种很好的想法。递归往往要确定一个初状态，以及各状态之间如何转换。以“汉诺塔”为例：

```python
def H(n, a, b, c):
    if n:
        H(n-1, a, c, b)
        print('%d:%s->%s' % (n, a, c))
        H(n-1, b, a, c)


n, a, b, c = input().split()
H(int(n), a, b, c)
```

分治是按一定规则，将问题划分成若干个互不相干的子问题，分别递归地求解。一般来说，复杂度为$O(\log n)$。

归并则将分治的结果按照一定规则组合起来，直到最终结果。

快速排序

类似二分的思想，选取一个数放在中间，比它大的放右边，比它小的放左边。复杂度一般为$O(n\log n)$，但最坏情况可能是$O(n^2)$

```python
def quick_sort(l):
    if len(l) <= 1:
        return l
    left, middle, right = [], [], []
    y = l[0]  # 以一定规则选一个都可以
    for x in l:
        if x < l[0]:
            left.append(x)
        elif x > l[0]:
            right.append(x)
        else:
            middle.append(x)
    return quick_sort(left)+middle+quick_sort(right)  # 左、右各自也要排序
```

归并排序和逆序对数

归并排序，以从小到大的排序为例，反过来只需要修改归并规则即可。先将数组一分为二，左右分别递归地继续进行排序（分治），然后再每次选取两个数组头上较小的那个（相等优先左边），依次合并成整个排好的数组（归并）。合并的过程，可以计算逆序对数，因为分治过程中，左边数组中的数始终在右边数组的左边，如果归并时从右边取出了数，就在逆序对数中加上此时左边剩余的元素数量。

```python
def merge_sort(l):
    if len(l) <= 1:
        return l, 0
    mid = len(l) // 2
    left, left_count = merge_sort(l[:mid])
    right, right_count = merge_sort(l[mid:])  # 分别处理左、右
    l, merge_count = merge(left, right)  # 归并
    return l, left_count + right_count + merge_count


def merge(left, right):
    merged = []
    left_index, right_index = 0, 0
    count = 0
    while left_index < len(left) and right_index < len(right):  # 左、右都不为空
        if left[left_index] <= right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1
            count += len(left) - left_index  # 如果从右边取出元素，则逆序对数加上左边剩余元素数
    merged += left[left_index:]+right[right_index:]
    return merged, count
```

堆和堆排序

python有自带的堆heapq，默认是最小堆，可以以$O(n\log n)$的复杂度将列表转换为堆（空列表不用转换），以$O(\log n)$的复杂度加入元素，以$O(1)$的复杂度查询堆顶（最小）元素，以$O(\log n)$的复杂度弹出堆顶元素。想要最大堆，可以存负值。

```python
import heapq
l = [2, 4, 5, 3]
heapq.heapify(l)  # 将l转化成堆
heapq.heappush(l, 1)  # 加入元素
print(l[0])  # 输出堆顶（最小值）
heapq.heappop(l)  # 弹出堆顶
print(heapq.heappop(l))  # 可以有返回值
print(heapq.nsmallest(3, l))  # 找到n个最小
```

手写的堆如下：

```python
class Tree:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None


class Heap:
    def __init__(self):
        self.root = None

    def add(self, val):
        if not self.root:
            self.root = Tree(val)
        else:
            self._add(self.root, val)

    def _add(self, node, val):
        if not node:
            return Tree(val)
        if val < node.val:
            node.left = self._add(node.left, val)
        else:
            node.right = self._add(node.right, val)
        return node

    def pop(self):
        if not self.root:
            return None
        node = self.root
        while node.left:
            node = node.left
        self.root = self._pop(self.root)
        return node.val
```

```python

    def _pop(self, node):
        if not node.left:
            return node.right
        node.left = self._pop(node.left)
        return node


heap = Heap()
for _ in range(int(input())):
    s = input()
    if len(s)-1:
        heap.add(int(s.split()[1]))
    else:
        print(heap.pop())
```

前缀和

$p[i]$表示$l[0]$至$l[i]$求和，这样求区间和就很方便，求$l[s]$至$l[e]$只需要求$p[e]-p[s-1]$，适用于要多次求区间和的情况

```python
n = int(input())
l = list(map(int, input().split()))
p = [0]*n
for i in range(n):
    p[i] = l[i]
    if i:
        p[i] += p[i-1]
s, e = map(int, input().split())
if s:
    ans = p[e]-p[s-1]
else:
    ans = p[e]
print(ans)
```

二维情况也可以这么求，先求出每一行（列）的前缀和，之后多次求矩形区域的和就可以节约时间。下面是一个找最大矩形区域和的例子：

```python
m, n = map(int, input().split())
M = [list(map(int, input().split())) for _ in range(m)]
for i in range(m):
    for j in range(1, n):
        M[i][j] += M[i][j-1]
ans = 1
for i in range(n):
    for j in range(max(i, 1)):
        f = [0]*m
        g = [0]*m
        for k in range(m):
```

```
                f[k] = M[k][i]
                if j:
                    f[k] -= M[k][j]
                g[k] = min(f[k], 0)
                if k:
                    f[k] += f[k-1]
                    g[k] = min(f[k], g[k-1])
            ans = max(ans, f[0])
            for k in range(1, m):
                ans = max(ans, f[k]-g[k-1])
print(ans)
```

迷宫问题综述

一定要仔细读题，看清楚题目要求，再选择用什么算法。

判断路径数用dfs，每一步时间相同用dfs，时间不同用Dijkstra。还要加上适当的判断不能走的方法，保证不超时的同时不遗漏情况。

如果要记录路径，可以直接存在每一步中，但是可能会超内存，在每一个节点中存它的上一个节点也可以。

容易踩的坑：边界，出发点和终点重合等。

深度优先搜索（dfs）

优先走新发现的能走的路，适合用来找路径数量。

```
def dfs(graph, gone, start, end):
    if start == end:
        # 走到终点
        return
    for next in graph[start]:
        if next not in gone:
            gone.add(next)
            dfs(graph, gone, next, end)   # 发现能走的路就走
            gone.remove(next)
    return
```

广度优先搜索（bfs）

优先走完下一步能走的，适合用来找最短路。

```
def bfs(graph, gone, start, end):
    l = [start]
    gone = set([start])
    s, e = 0, 1
```

```python
    cost = 0
    while s-e:
        cost += 1
        for i in range(s, e):  # 遍历下一步要走的
            for next in graph[l[i]]:
                if next == end:
                    return cost  # 走到终点
                if next not in gone:
                    gone.add(next)
                    l.append(next)
        s, e = e, len(l)
    return 'Impossible!'  # 无法走到终点
```

Dijkstra

每次挑选当前到达时，总花费最小的点。和bfs很像。可以用一个堆来存储当前的可能路径。如果还有别的要求，也可以按照“不亏”的原则加入。

由于使用了堆，每次走的一定是最优路径。但要注意，为了得到最优路径，不能在能走到终点时就退出，而是在下一个最优路径走向终点时才能退出。

```python
from heapq import heappush, heappop


def Dijkstra(graph, cost, start, end):
    q = []
    gone = set()
    heappush(q, (0, start))
    while q:
        c, x = heappop(q)
        if x == end:
            return c  # 走到终点，注意不是能走到，而是下一步往终点走才算结束
        gone.add(x)  # 这个点成为最小花费，才标记为走过（如果有其他限制，改为储存最大开销）
        for next in graph[x]:
            if next not in gone:
                heappush(q, (c+cost[x][next], next))
    return 'Impossible!'  # 走不到终点
```

二叉树及其遍历

用Tree类来表示二叉树，注意左右子树都是Tree类。也可以用列表或字典等，注意处理指标问题。

```python
class Tree:
    def __init__(self, val, left, right):
        self.val = val
        self.left = left
        self.right = right

    def build(self, input):
        # 以一定的方法构建树
        return
```

遍历二叉树，有前序（pre）、中序（in）、后序（suf），遍历节点顺序都是一样的，区别是输出时根节点的位置

```python
def search(node, order):
    root = node.val
    left = search(node.left, order)
    right = search(node.right, order)
    if order == 'pre':
        return root+left+right
    if order == 'in':
        return left+root+right
    if order == 'suf':
        return left+right+root
```

没出现在子节点中的那个节点就是根节点。

三者的互相转换，主要利用三者的特点：前序的第一个一定是根节点；中序左子树遍历全在根节点左边，右子树全在右边；后序最后一个是根节点。此外，无论哪种顺序，左子树一定全在右子树的前面。下面是一个中序后序转前序的例子：

```python
def pre(mid, suf):
    if len(mid) > 1:
        root = suf[-1]
        n = mid.index(root)
        left = pre(mid[:n], suf[:n])
        right = pre(mid[n+1:], suf[n:-1])
        return root+left+right
    else:
        return mid


mid = input()
suf = input()
print(pre(mid, suf))
```

二分查找树与平衡二叉树（AVL树）

二分查找树，左子节点一定比根节点小（大），右子节点一定比根节点大（小），按照这个规则进行构建和查找即可。之前的堆就是一个例子。

如果运气不好，二分查找树会变成一条，此时复杂度变为$O(n^2)$。为了避免此事，建立平衡二叉树，当左右子树高度差大于1，则旋转。

```
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1


class Tree:
    def __init__(self):
        self.root = None

    def height(self, node):
        if node is None:
            return 0
        return node.height

    def balance_factor(self, node):
        if node is None:
            return 0
        return self.height(node.left) - self.height(node.right)

    def rotate_right(self, y):
        x = y.left
        T = x.right
        x.right = y
        y.left = T
        y.height = 1 + max(self.height(y.left), self.height(y.right))
        x.height = 1 + max(self.height(x.left), self.height(x.right))
        return x

    def rotate_left(self, x):
        y = x.right
        T = y.left
        y.left = x
        x.right = T
        x.height = 1 + max(self.height(x.left), self.height(x.right))
        y.height = 1 + max(self.height(y.left), self.height(y.right))
        return y

    def insert(self, node, val):
        if node is None:
            return Node(val)
        if val < node.val:
```

```
            node.left = self.insert(node.left, val)
        else:
            node.right = self.insert(node.right, val)
        node.height = 1 + max(self.height(node.left), self.height(node.right))
        balance = self.balance_factor(node)
        if balance > 1 and val < node.left.val:
            return self.rotate_right(node)
        if balance < -1 and val > node.right.val:
            return self.rotate_left(node)
        if balance > 1 and val > node.left.val:
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)
        if balance < -1 and val < node.right.val:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)
        return node

    def pre(self, node, result):
        if node:
            result.append(node.val)
            self.pre(node.left, result)
            self.pre(node.right, result)
```

最小生成树

有两种方法：Prim和Kruskal，适用于不同情况。一般用Prim。

Prim算法对节点选择，每次添加到当前到达成本最小的点。每次添加了新的点，更新其未标记的邻接点的成本。点被添加后再标记。

```
from heapq import heappop, heappush
n, m = map(int, input().split())
graph = [[-1]*n for _ in range(n)]
for _ in range(m):
    u, v, w = map(int, input().split())
    graph[u][v] = w
visited = [0]*n
q = [(0, 0)]
ans = 0
while q:
    w, u = heappop(q)  # 选择成本最小的点
    if visited[u]:
        continue
    ans += w
    visited[u] = 1
    for v in range(n):
        if not visited[v] and graph[u][v] != -1:
            heappush(q, (graph[u][v], v))
print(ans)
```

Kruskal算法对边选择，每次添加权重最小的边。利用并查集来判断一条边是否有必要添加。

```python
def Find(x):  # 并查集
    if p[x] != x:
        p[x] = Find(p[x])
    return p[x]


n, m = map(int, input().split())
d = []
p = [i for i in range(n)]
for _ in range(m):
    u, v, c = map(int, input().split())
    d.append((u, v, c))
d.sort(key=lambda x: x[2])  # 按照权重排序
ans = 0
for u, v, c in d:
    pu, pv = Find(u), Find(v)  # 判断是否已经形成连接
    if pu != pv:
        p[pu] = pv
        ans += c
print(ans)
```

字典树

主要用来找一个字符串是否存在与字典中。思路就是用树，如果叫做“前缀树”其实更合适。构建字典时，每个字符下一位是它的子树，字符串末尾要做标记。查找时则一位一位找，如果找前缀那查到底就行，如果找字符串是否存在，就看最后一位是否被标记。

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False


class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):  # 加入字符串
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True

    def search(self, word):  # 查找完整字符串
        node = self.root
```

```python
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end

    def starts_with(self, prefix):  # 查找前缀
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True


# 输入字典，构建字典树
dictionary = input().split()
trie = Trie()
for word in dictionary:
    trie.insert(word)

# 判断字符串是否在字典中
search_word = input()
print(trie.search(search_word))

# 判断是否有字符串以某个前缀开头
prefix = input()
print(trie.starts_with(prefix))
```

并查集

$p[i]$是i元素的老祖宗，每个元素属于的集合用老祖宗表示。查找老祖宗的同时，会自动找到最老的祖宗。要将两个集合合并，只需要让其中的一个老祖宗成为另一个老祖宗的祖宗。

处理一些特殊问题，比如食物链问题，可以多开几倍的空间，存储更多可能情况。

```python
def Find(x):  # 查询
    if p[x] != x:
        p[x] = Find(p[x])
    return p[x]


def Union(x, y):  # 合并
    p[Find(x)] = Find(y)
    return

p = [i for i in range(n)]  # 初始化
```

拓扑排序与环判断

依次弹出入度为0的点，得到的顺序就是拓扑排序。如果出现了环，则环里的点都不会被排出来，拓扑排序结果会比节点数少，由此可以判断是否存在环。

```python
def topo_sort(graph):
    in_degree = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1  # 计算入度
    q = [u for u in in_degree if in_degree[u] == 0]
    topo_order = []
    i = 0
    while i != len(q):
        u = q[i]  # 弹出入度为0的节点
        i += 1
        topo_order.append(u)
        for v in graph[u]:
            in_degree[v] -= 1
            if in_degree[v] == 0:
                q.append(v)
    if len(topo_order) != len(graph):  # 存在环
        return 'Loop!'
    return topo_order


graph = {0: [1, 2, 3], 1: [2, 3], 2: [3], 3: []}
print(topo_sort(graph))
```

失配表和kmp算法

失配表$next$可以用来优化查询，其含义是：$next[n] = k$表示在字符串s的前$n+1$位（即末尾索引是n）这部分中，前k位和后k位是一样的。比如，对于$s = \text{'AABAACAABAA'}$，$next = [0,1,0,1,2,0,1,2,3,4,5]$。构建的代码如下：

```python
def cal_next(s):
    n = len(s)
    next = [0]*n
    for i in range(1, n):
        j = next[i-1]
        while j > 0 and s[i] != s[j]:
            j = next[j-1]
        if s[i] == s[j]:
            j += 1
        next[i] = j
    return next
```

这个表的意义在于，如果过程中出现完全匹配或者匹配失败，需要重新开始找的时候，由于能走到这一位j，在此之前的每一位都是匹配的，那么，下次开始匹配，可能成功的最近位置，就在$next[j-1]$位置处。

kmp算法用来查找某个字符串第一次出现在另一字符串中的位置：

```
def kmp(text, pattern):
    m = len(text)
    n = len(pattern)
    next = cal_next(pattern)
    i = 0  # text指标
    j = 0  # pattern指标
    while i < m:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == n:  # 找到了
            print('在%d位置找到' % (i-j))
            j = next[j - 1]
        elif i < m and pattern[j] != text[i]:  # 无法匹配
            if j != 0:
                j = next[j - 1]
            else:
                i += 1
```

判断是否存在循环的代码如下：

```
next = cal_next(s)
for i in range(1, len(s)+1):
    if next[i-1] != 0 and i % (i-next[i-1]) == 0:
        print(i, i//(i-next[i-1]))  # 输出存在循环的位置以及最多循环节数量
```

质数筛

下面是一个欧拉筛：

```
n = int(input())
is_prime = [True] * (n + 1)  # 判断表
primes = []  # 值表
for i in range(2, n + 1):
    if is_prime[i]:
        primes.append(i)
        for j in range(i * i, n + 1, i):
            is_prime[j] = False
```

运算表达式

我们惯用的运算表达式其实是中序的，但是计算机不会算中序表达式，要将其转化为后序表达式。按照如下步骤对表达式进行处理：

1. 读取：从输入字符串中逐个读取字符，按顺序放入列表。由于一个数会有多位，要特别处理，当出现数字或小数点，将其存在当前的数字字符串中，直到读到非数字，就先将非空数字字符串存入。记得在最后也要存一次。

```
num = ''
l = []
for char in input().strip():
    if '0' <= char <= '9' or char == '.':
        num += char
    else:
        if num:
            l.append(num)
            num = ''
        l.append(char)
if num:
    l.append(num)
```

2. 中序转后序：由于数字始终是符号的左右子树，数字的顺序不改变，直接进入后序列表。用一个辅助栈来存储运算符号，当出现匹配的括号，就进行括号内的全部运算，当出现运算符号，就将之前的同级或更高级运算先进行，再将运算符号压入栈。最后，将栈清空，放入后序列表。

```
order = {'+': 0, '-': 0, '*': 1, '/': 1}  # 运算顺序
suf = []
stack = []
for char in l:
    if '0' <= char[0] <= '9':
        suf.append(char)
    else:
        if char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                suf.append(stack.pop())
            stack.pop()
        else:
            while stack and stack[-1] != '(' and order[stack[-1]] >= order[char]:
                suf.append(stack.pop())
            stack.append(char)
while stack:
    suf.append(stack.pop())
print(*suf)
```

3. 运算：依次从后序表达式读入，遇到数字就压入栈中，遇到运算符就从栈中弹出最后两个，运算结果压回栈中。最终栈里会剩下一个数，就是结果。

```python
stack = []
for char in suf:
    if '0' <= char[0] <= '9':
        stack.append(char)
    else:
        a = float(stack.pop())
        b = float(stack.pop())
        if char == '+':
            stack.append(str(b+a))
        elif char == '-':
            stack.append(str(b-a))
        elif char == '*':
            stack.append(str(b*a))
        elif char == '/':
            stack.append(str(b/a))
print(stack[0])
```

工具

输出格式

```python
x = 12.345
print('%d' % x)  # 输出x整数
print('%4d' % x)  # 以最小长度4输出x整数
print('%.3g' % x)  # 输出x保留3位有效数字
print('%f' % x)  # 输出x浮点数
print('%.1f' % x)  # 输出x保留一位小数
```

海象写法（赋值同时返回值）

```python
t = 5
while (t := t-1):  # 这个循环将执行4次
    print(t)
```

进制转换

```python
int(s, n)  # 将字符串s转换为n进制的整数
```

遍历

```python
for key, value in dict.items()  # 遍历字典的键值对
for index, value in enumerate(list)  # 枚举列表，提供元素及其索引
dict.get(key, default)  # 从字典中获取键对应的值，如果键不存在，则返回默认值default
list(zip(A, B))  # 将两个列表元素一一配对，生成元组的列表
for a, b in zip(A, B)  # 同时遍历A和B中对应元素
print(*ans)  # 将ans中的元素用空格隔开输出
```

数学 math

```
imort math
math.pow(m,n)  # 计算m的n次幂（m^n）
math.log(m,n)  # 计算以n为底的m的对数（log_n(m)）
math.sqrt(x)  # 计算x的平方根
```

记忆化搜索 lrucache

```
from functools import lru_cache
@lru_cache(maxsize=None)
```

设置最大递归次数

```
import sys
sys.setrecursionlimit(1000000000)
```

字符串

```
str.lstrip() / str.rstrip() / str.strip()  # 移除字符串左侧/右侧/两侧的空白字符
str.find(sub)  # 返回子字符串sub在字符串中首次出现的索引，如果未找到，则返回-1
str.replace(old, new)  # 将字符串中的old子字符串替换为new
str.startswith(prefix) / str.endswith(suffix)  # 检查字符串是否以prefix开头或以suffix结尾
str.isalpha() / str.isdigit() / str.isalnum()  # 检查字符串是否全部由字母/数字/字母和数字组成
str.title()  # 每个单词首字母大写
str.upper() / str.lower()  # 全部大写/小写
str.capitalize()  # 句子首字母大写
```

字符串转换（常见值：$'A':65, 'a':98, '0':48$）

```
int = ord(str)  # 将str转换成int
str = chr(int)  # 将int转换成str
```

初始字典 defaultdict

```
from collections import defaultdict
a = defaultdict(int)  # 初始为整型的字典
b = defaultdict(list)  # 初始为空列表的字典
c = defaultdict(lambda: [0])  # 自定义初始值
```

计数器 Counter

```python
from collections import Counter
# 创建一个Counter对象
count = Counter(['apple', 'banana', 'apple', 'orange', 'banana', 'apple'])
print(count)  # 输出: Counter({'apple': 3, 'banana': 2, 'orange': 1})
print(count['apple'])  # 输出: 3
print(count['grape'])  # 输出: 0
count.update(['grape', 'apple'])
print(count)  # 输出: Counter({'apple': 4, 'banana': 2, 'orange': 1, 'grape': 1})
```

全排列 permutations

```python
from itertools import permutations
# 创建一个可迭代对象的排列
perm = permutations([1, 2, 3])
# 打印所有排列
for p in perm:
    print(p)
# 输出: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

组合 combinations

```python
from itertools import combinations
comb = combinations([1, 2, 3], 2)
for c in comb:
    print(c)
# 输出: (1, 2), (1, 3), (2, 3)
```

累次运算 reduce

```python
from functools import reduce
product = reduce(lambda x, y: x * y, [1, 2, 3, 4])  # 使用reduce计算列表元素的乘积
print(product)  # 输出: 24
```

笛卡尔积 product

```python
from itertools import product
prod = product([1, 2], ['a', 'b'])
print(*prod)
# 输出: (1, 'a') (1, 'b') (2, 'a') (2, 'b')
```

捕捉报错

```python
try:
    s=input()
except EOFError/KeyError/IndexError/...:
    break
```

其他技巧

加个负号，就可以把最大问题转换成最小问题。

可以通过数据量来反推允许的的最大时间复杂度：

$10^9 - O(n)$

$10^5 - O(n \log n)$

$10^3 - O(n^2)$

如果一个问题正着很难，可以试着反过来。

变量名不要取重了。

类似词梯的问题，如果现场找下一步能走的，可能会超时，可以利用桶，将能连接的放在一个桶里。

骑士周游问题，优先走下一步能走的路少的位置，会大大减少回溯次数。

未通过的应对方法

CE：看报错的内容，针对性地修改。

WA：改细节，注意特殊情况的判别，有可能要改算法。

MLE/TLE：加强剪枝，不过大概率要换算法。MLE也可以尝试优化存储结构。

RE：指针越界，或者递归层数太多，前者很好改，后者可能要改递归层数或者改算法。