Book my Spacecraft

2025 fall, Complied by 陈子良 物理学院

简介

这份资料总结了笔者在学习计算概论课程时的一些独到见解和经验，具有鲜明的个人特色，供读者学习借鉴。

本资料属于进阶型资料，**难度非常高**，要求读者对课程中各种算法有基本的了解。

本资料最重要的部分不是例题和模版，而是**思想**，读者要注意不能生搬硬套，而要融汇贯通，形成自己的知识框架。

> 几点说明：
>
> 1、资料编写期间，本人尚未学习数算课程，因此资料中只包含计概内容。
>
> 2、本资料完全忽略Python底层设计带来的时、空复杂度差异，只考虑算法方面的差异。例如，本资料忽略 `list` 和 `dict` 的存储空间差别，忽略手动二分和调用 `bisect` 库的时间差别，忽略局部变量与全局变量的访问时间差别，等等。正因如此，读者可以在资料中看到大量的等价性描述。
>
> 3、为了行文的连贯性，资料中的大部分示例题目，笔者只放了一个链接，或者简单描述了一下题目。建议读者打开链接，结合原题服用效果更佳。
>
> 4、本资料不可避免地会存在不合理、不完美之处，望读者批评指正。

基本数据结构

人类需要数据结构。——佚名

这里讨论的基本数据结构包括列表、字典、元组、集合。

首先我们来从数学的角度思考数据结构。**数据结构在数学上是一种映射，输入已知值，输出目标值。**

对列表来说，输入值是列表的索引，输出值是索引对应的列表元素。对字典来说，输入值是键，输出值是键对应的值。两者都满足映射的定义，即对于给定的输入，只有一个输出与之对应。

我们知道，列表的索引只能是整数，而字典的键可以取所有不可变类型（整数，浮点数，字符串，元组）。从这个意义上说，字典完全覆盖了列表的功能。而元组又是不可变的列表，因此列表又完全覆盖元组的功能。由于字典的键和集合都满足无重复的要求，因此字典又完全覆盖集合的功能。

```
# 下面两个数据结构在数学上完全相同
list1=['a','b','c','d']
dict1={0:'a',1:'b',2:'c',3:'d'}
# 下面两个数据结构在数学上完全相同
set1={1,2,3,4}
dict1={1:True,2:True,3:True,4:True}
```

这就是本笔记的第一个暴论：

原则上来讲，python中所有的基本数据结构都可以用字典代替。

其实在计算概论的学习中，可以看到很多“原则上”格式的观点，如：

- 原则上来讲，所有可计算的问题都可以用图灵机解决
- 原则上来讲，所有递归都可以用栈来实现

但没有人会真的这么干。抛开python的底层设计问题，从一个编程者的角度思考，我们为什么还要使用列表？为什么还要使用元组？

笔者认为，除了代码编写上的方便外，更重要的一点是**心理作用。**

“字典”这个词很好地描述了这个数据结构的**无序性。**当你使用字典的时候，你心里想到的大概率是一个个离散的点，每个点是一个键值对。

而当你使用列表的时候，你心里就知道这是一个有序排列的结构，是一个数轴上的**连续性**的结构，这为思考问题、理清思路提供了很大的方面。例如双指针的思想，就是基于列表“像一条线”

的特征，于是自然地想到从线段的两端向中间靠的处理办法。如果你是用字典存储的，估计就很难想到双指针了。

那元组呢？元组对人心理上的作用是**整体性**。元组一旦创建就不可改变，因此它包含的所有元素就描述了同一个整体对象的特征，比如用`(x,y)`表示点的坐标。

我们要描述一个对象，如果它只需要一个参数，可以用整数或浮点数；如果需要多个参数，就可以包装成一个元组。一个典型的应用是在矩阵中作bfs，可以把点的坐标`x`，`y`连同到达该点的时间`t`存入一个元组`(x,y,t)`，然后推入队列`queue`中。这里当然可以用列表代替，但用元组更能体现考察对象的整体性特征。

至于集合就简单了。**去重**是我们使用集合的唯一理由。凡是需要去重的场景，集合都是不二之选。

我们平时对数据结构的操作，本质上只有两种，一种是**修改**，一种是**查找**。数据结构的根本目的是让我们**尽可能快地找到我们需要的信息**。

我们在构建自己的数据结构时，要注意以下几点：

- **心里要时刻清楚你的数据结构存储的是什么。**

这是最基本的要求。不仅要关注存储的内容、格式，还要关注边界条件以及可能的出错情况。当代码比较长的时候，很容易因为粗心而错误地使用数据结构，比如把二维列表当成一维列表来索引，这就会出现代码错误。

- **针对不同的功能选择不同的数据结构。**

没有适用于所有情况的数据结构，要发挥各个数据结构的优势，如列表适合修改与遍历，字典适合查找。必要时甚至可以自己现创一个数据结构。

- **数据结构不要嫌多。每出现一个映射关系，就建一个数据结构。**

编程序的时候多花一点时间，程序运行的时候就能节约很多时间。当数据结构多起来的时候，要记得及时更新所有应该更新的数据结构。

- **数据结构不需要物理意义。**

我们处理问题时不关心原来的信息表示什么，它们之间有什么因果关系。我们只关心我们想要的结果，以及为了得到这个结果需要什么信息。比如输入商品的数量和总价，要按性价比排序，我们建立的元组就可以以性价比为第一个参数，数量和总价要靠后放。特别地，对于字典，我们还关心输入信息是否存在重复，这会涉及到键的唯一性问题。

- **只保留需要的信息，抛弃不需要的信息。**

这有助于我们节省空间，同时将精力集中在核心问题上。

- **尽可能避免遍历操作。**

原则上来讲，所有问题都可以通过遍历解决。 给一只猴子一台计算机，它能打出比《红楼梦》更好的文学作品，但等到那个时候宇宙都灭亡好几次了。

每次遇到大规模的遍历操作，都要考虑能不能进行优化。比如查找列表中的某个元素，能否通过预先建立一个字典来帮助查找。

- **尽可能避免删除操作。**

在常见的数据结构中，没有一个适合全局上的删除操作，列表和字典的删除操作时间复杂度都是`O(n)`。而事实上大部分的问题不使用删除操作也能解决，或者存在比删除更好的解决方案（如标记和懒删除）。

> **选读内容 删除操作**
>
> 在常见的数据结构中，没有一个适合全局上的删除操作。但是局域上的删除操作可以非常高效，如列表和栈的`pop()`操作对末尾元素的删除，队列的`pop()`操作和`popleft()`操作，堆的`heapq.heappop()`操作，都是时间复杂度`O(1)`的操作。
>
> 其实有一个数据结构非常适合删除操作，那就是`collections`库中的`OrderedDict`。`OrderedDict`是一种特殊的字典类型，它在普通字典的基础上，保留了键值对的插入顺序，特别适合进行关于顺序的操作。除此之外，它对各种删除操作的时间复杂度都是`O(1)`。

```
>>> from collections import OrderedDict
>>> od=OrderedDict({'a':1,'b':2,'c':3,'d':4,'e':5,'f':6,'g':7})
>>> # 移动到末尾，时间复杂度O(1)
>>> od.move_to_end('a')
>>> od
OrderedDict({'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'a': 1})
>>> # 移动到开头，时间复杂度O(1)
>>> od.move_to_end('a',last=False)
>>> od
OrderedDict({'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7})
>>> # del od[key] 删除指定元素，时间复杂度O(1)
>>> del od['d']
>>> od
OrderedDict({'a': 1, 'b': 2, 'c': 3, 'e': 5, 'f': 6, 'g': 7})
>>> # od.pop(key) 删除并返回指定键的元素，key不可省略。时间复杂度O(1)
>>> od.pop('c')
3
>>> # od.popitem() 删除并返回最后一个键值对，时间复杂度O(1)
>>> od.popitem()
('g', 7)
```

```
>>> # od.popitem(last=False) 删除并返回第一个键值对，时间复杂度O(1)
>>> od.popitem(last=False)
('a', 1)
>>> od
OrderedDict({'b': 2, 'e': 5, 'f': 6})
```

以上几点要求中，有一点值得拿出来提一下，那就是**每出现一个映射关系，就新建一个数据结构。**

下面这个小问题是我在OpenJudge - T27384:候选人追踪中想到的，虽然原题不适合用这种方法做，但正好可以讲解数据结构的妙用：

> 超大型偶像团体HIHO314159总选举刚刚结束了。制作人小Hi正在复盘分析投票过程。
>
> 小Hi获得了`N`条投票记录，每条记录都包含一个时间戳`Ti`以及候选人编号`Ci`，代表有一位粉丝在`Ti`时刻投了`Ci`一票。
>
> 请实时返回任意时刻候选人的排名情况。已知HIHO314159这个团体有`n`名团员，编号是`1~n`

我们慢慢搭建起适合本题的描述方式。

关键的几个参量包括：候选人编号，候选人的票数，候选人的排名。

首先我们要知道所有候选人的排名情况，自然想到建立一个列表`rank`，`rank`的索引`i`表示排名（从1开始），对应的值表示排在第`i`名的候选人编号。这是第一个映射关系。

接下来，我们要给候选人投票，需要知道每个候选人当前的票数。我们可以考虑用字典来存储候选人与票数的对应关系，但注意到排名是按票数递减顺序排列的，将票数与排名联系起来也是个不错的选择。

因此我们改进`rank`的结构，原来`rank[i]`只存储候选人编号，我们可以把它变成二元列表，同时存储候选人编号和其票数。这样`rank`就变成了二维列表，`rank[i][0]`表示排在第`i`位的候选人编号，`rank[i][1]`表示其票数。

```
rank=[[i,0] for i in range(n+1)]
# 故意加一个0号候选人占一个位置
```

最后，也是最关键的一步：已知某次获得投票的候选人，将其票数加一，并更新所有人的排名。

我们发现光凭一个`rank`很难由候选人编号找到其排名。一种选择是遍历`rank`列表找到目标候选人，但这绝对不是我们想要的。注意看，这里又出现了一个映射关系，意味着我们又要建一个数据结构！

我们再建立一个字典 `dict1`，键是候选人的编号，值是该候选人的排名。这其实完全是 `rank` 映射关系的逆过程。

每当投一次票，我们就利用 `dict1` 找到获票候选人 `c` 的排名 `i`，这也是该候选人在 `rank` 中的索引值。然后将 `rank[i][1]` 加一。这时将排名与票数联系起来的好处就来了，为了更新候选人 `c` 的排名，我们不断比较 `c` 与排在他前面的那个人的票数，即比较 `rank[i][1]` 与 `rank[i-1][1]`，如果 `c` 的票数较高，就将两人的位置互换，同时更新 `dict1` 中的数据。

由此，我们就高效地实现了候选人排名。

最终代码：

```
rank=[[i,0] for i in range(n+1)]
dict1={i:i for i in range(1,n+1)}
for _ in range(N):
    t,c=map(int(input().split()))
    i=dict1[c]
    rank[i][1]+=1
    while i>1 and rank[i-1][1]<rank[i][1]:
        dict1[rank[i-1][0]]+=1
        dict1[rank[i][0]]-=1
        rank[i-1],rank[i]=rank[i],rank[i-1]
        i-=1
```

> OpenJudge - T04093:倒排索引查询
>
> 现在已经对一些文档求出了倒排索引，对于一些词得出了这些词在哪些文档中出现的列表。
>
> 要求对于倒排索引实现一些简单的查询，即查询某些词同时出现，或者有些词出现有些词不出现的文档有哪些。

这题的题目比较复杂，我们好好梳理一下。

题目给出的是对于某一个单词，它出现在哪些文章中。现在我们想要知道，对于某一篇文章，它是否含有/不含有某些单词，然后问满足条件的文章有哪些。

为了记录信息，我们既可以选择建立从单词到文章的字典，也可以建立从文章到单词的字典。无论是哪一种，都是极为复杂的多向关系。那么我们需要哪一个呢？

初学者对数据结构的理解不够深入，既然题目给出的是单词到文章的映射，那么我建立一个单词到文章的字典不就好了？这样虽然方便，但后续的处理就很困难了。

记住，数据结构永远是为解决问题而服务的。既然我们需要判断某一篇文章是否包含某个单词，那就应该建立从文章到单词的映射。我们要把题目给出的单词到文章的映射，转换为文章到单词的映射。

这个字典应该以文章为键；那么它的值应该是什么？这个值应该记录它包含的单词，而且要方便我们后期查找。因此就想到了，这个值应该是单词的集合。

```
from collections import defaultdict
N=int(input())
dict1=defaultdict(set)
for word in range(N):
    list1=list(map(int,input().split()))
    for article in list1[1:]:
        dict1[article].add(word)
```

接下来我们可以建立一个函数，根据某个单词的目标状态判断某篇文章是否满足条件。

```
def check(article,word,condition):
    if condition==1 and word in dict1[article]:
        return True
    elif condition==-1 and word not in dict1[article]:
        return True
    elif condition==0:
        return True
    else:
        return False
```

如果某篇文章对所有单词均满足目标条件，就把它加入结果列表中。

最终代码：

```
from collections import defaultdict
N=int(input())
dict1=defaultdict(set)
for word in range(N):
    list1=list(map(int,input().split()))
    for article in list1[1:]:
        dict1[article].add(word)
def check(article,word,condition):
    if condition==1 and word in dict1[article]:
        return True
    elif condition==-1 and word not in dict1[article]:
        return True
    elif condition==0:
        return True
    else:
        return False
M=int(input())
```

```
for _ in range(M):
    condition=list(map(int,input().split()))
    res=[]
    for article in dict1.keys():
        if all(check(article,i,condition[i]) for i in range(N)):
            res.append(article)
    res.sort()
    if res:
        print(*res)
    else:
        print('NOT FOUND')
```

高级数据结构

这里讨论的高级数据结构包括栈、队列、堆。

> 在使用高级数据结构时，首先要明确的一点是，**遵循各个数据结构自身的结构要求，不要做违规的事情**。比如说，随意修改队列或堆中的某个值，对栈进行排序。除非你之后不想再用这个数据结构了。

首先回顾一下这三种数据结构。在Python语言中，栈与列表没有区别，栈就是一种只允许在末端操作的列表。栈能进行的操作有：

```
>>> stack=[1,2,3,4,5]
>>> # 读取最后一个数
>>> stack[-1]
5
>>> # 入栈
>>> stack.append(6)
>>> stack
[1, 2, 3, 4, 5, 6]
>>> # 出栈
>>> stack.pop()
6
>>> stack
[1, 2, 3, 4, 5]
```

队列相比于列表，只多了一个操作，就是`popleft()`操作。

```
>>> from collections import deque
>>> queue=deque([1,2,3,4,5])
>>> # 读取首位元素
>>> queue[0]
1
>>> # 读取末位元素
>>> queue[-1]
5
>>> # 入队
>>> queue.append(6)
>>> queue
deque([1, 2, 3, 4, 5, 6])
>>> # 队尾出队
>>> queue.pop()
6
>>> queue
deque([1, 2, 3, 4, 5])
```

```
>>> # 队首出队
>>> queue.popleft()
1
>>> queue
deque([2, 3, 4, 5])
```

现阶段你不需要知道堆的工作原理。你只需要知道，把一堆数扔到堆里面，它能实时返回其中的最小值。

```
>>> import heapq
>>> heap=[8,7,6,5,4,3,2,1]
>>> # 初始化一个堆
>>> heapq.heapify(heap)
>>> heap
[1, 4, 2, 5, 8, 3, 6, 7]
>>> # 将元素加入堆中
>>> heapq.heappush(heap,-1)
>>> heap
[-1, 1, 2, 4, 8, 3, 6, 7, 5]
>>> # 弹出堆顶元素
>>> heapq.heappop(heap)
-1
>>> heap
[1, 4, 2, 5, 8, 3, 6, 7]
>>> # 严正警告：heapq.heapify操作返回None，绝对不要把它赋值给堆！
>>> heap=heapq.heapify([8,7,6,5,4,3,2,1])
>>> print(heap)
None
```

> 其实 `queue` 和 `heapq` 还提供了很多其他函数，如 `queue.rotate(n)` 表示把 `queue` 轮换 `n` 次，`heapq.nsmallest(n,heap)` 查询堆中最小的 `n` 个元素。但是万变不离其宗，这些操作完全可以用简单的 `push` 和 `pop` 组合实现，因此我们不必掌握这些函数，所谓大道至简。

随着做题量的增加，你对栈和队列这两种数据结构的理解会越来越多元。

- 从数据处理的角度，栈和队列为数据处理提供了一种逻辑思路。

如OpenJudge - T29947:校门外的树又来了，我们可以利用 `stack` 对种树的区间进行合并。按左端点排序遍历区间，如果当前区间与栈尾的区间有重叠，则更新栈尾的区间，否则将当前区间压入栈中。

```
L,M=map(int,input().split())
interval=[]
for _ in range(M):
```

```
    interval.append(list(map(int,input().split())))
interval.sort()
stack=[]
for t in interval:
    if stack and t[0]<=stack[-1][1]:
        stack[-1][1]=max(stack[-1][1],t[1])
    else:
        stack.append(t)
print(L+1-sum(map(lambda x:x[1]-x[0]+1,stack)))
```

如OpenJudge - 27371:Playfair密码，其中有一步是将明文转变为字母对，我们可以利用`stack`进行处理。

```
word=input()
stack=[]
for a in word:
    if a=='j':
        a='i'
    if not stack:
        stack.append(a)
        continue
    if len(stack[-1])==2:
        stack.append(a)
        continue
    if stack[-1]!=a:
        b=stack.pop()
        stack.append(b+a)
    else:
        if stack[-1]=='x':
            stack.pop()
            stack.append('xq')
            stack.append(a)
        else:
            b=stack.pop()
            stack.append(b+'x')
            stack.append(a)
if len(stack[-1])==1:
    if stack[-1]=='x':
        stack.pop()
        stack.append('xq')
    else:
        b=stack.pop()
        stack.append(b+'x')
```

如OpenJudge - M02786:Pell数列，可以用一个长度恒为2的队列计算递推公式。

```
from collections import deque
queue=deque()
queue.append(1)
queue.append(0)
pell=[0]
for _ in range(1000000):
    p=(2*queue[-1]+queue[-2])%32767
    pell.append(p)
    queue.append(p)
    queue.popleft()
for _ in range(int(input())):
    print(pell[int(input())])
```

- 从指针的角度，栈与一个从后往前的指针是等价的。

```
# 以下两端代码完全相同
list1=[1,2,3,4,5]
n=len(list1)
for i in range(n-1,-1,-1):
    a=list1[i]
    # 后续操作

stack=[1,2,3,4,5]
while stack:
    a=stack.pop()
    # 后续操作
```

双指针语言与队列语言也是等价的。如OpenJudge - M18211:军备竞赛，以下两种写法完全相同：

```
p=int(input())
price=list(map(int,input().split()))
price.sort()
l=0
r=len(price)-1
s=0
while l<r:
    while p>=price[l]:
        p-=price[l]
        s+=1
        l+=1
    if l<r:
        if s==0:
            break
```

```
        p+=price[r]
        p-=price[l]
        r-=1
        l+=1
    if l==r and p>=price[l]:
        p-=price[l]
        s+=1
        l+=1
print(s)
```

```
from collections import deque
p=int(input())
queue=deque(sorted(list(map(int,input().split()))))
s=0
while len(queue)>1:
    while p>=queue[0]:
        p-=queue.popleft()
        s+=1
    if len(queue)>1:
        if s==0:
            break
        p+=queue.pop()-queue.popleft()
    if len(queue)==1 and p>=queue[0]:
        p-=queue.popleft()
        s+=1
print(s)
```

- 从函数的角度，栈可以看成一种递归函数，输入一个值，并等待一个与它对应的值。

如OpenJudge - M02694:波兰表达式，一个波兰表达式接受三个参数，一个运算符和两个数。当参数不足时，就暂时将其压入栈中，待参数集齐后进行计算。

```
stack1=list(input().split())
stack2=[]
while stack1:
    i=stack1.pop()
    if i in '+-*/':
        a=stack2.pop()
        b=stack2.pop()
        stack2.append(eval(f'{a}{i}{b}'))
    else:
        stack2.append(float(i))
print(f'{stack2[-1]:.6f}')
```

- 从信息的完整性角度，栈可以分为等位栈和差位栈（笔者杜撰的概念）。等位栈的长度等于原列表，差位栈的长度小于原列表。等位栈的优点是直观，逻辑清晰，差位栈的优点是节省时间和空间。

```
# 单调递减栈的两种写法
height=[73,74,75,71,62,69,64,63,61]
# 等位栈写法
stack=[2,2,2,3,5,5,6,7,8]
# 差位栈写法
stack=[2,3,5,6,7,8]
```

如经典单调栈题目[739. 每日温度 - 力扣（LeetCode）](#)，以下分别是等位栈写法和差位栈写法。

```
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        n=len(temperatures)
        stack=[]
        ans=[0]*n
        for i in range(n):
            c=1
            while stack and temperatures[i]>temperatures[stack[-1]]:
                j=stack.pop()
                if ans[j]==0:
                    ans[j]=i-j
                c+=1
            for _ in range(c):
                stack.append(i)
        return ans
```

```
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        n=len(temperatures)
        stack=[]
        ans=[0]*n
        for i in range(n):
            while stack and temperatures[i]>temperatures[stack[-1]]:
                j=stack.pop()
                ans[j]=i-j
            stack.append(i)
        return ans
```

- 从数据结构的等价性角度，队列完全覆盖了栈的功能。

有单调栈，自然就有单调队列。单调下降栈满足栈底元素为栈中最大值，如果为其添加队列的队首弹出功能，那么队列的队首元素就是队内元素的最大值，再令队列动态变化，就可以实时返回 239. 滑动窗口最大值 - 力扣（LeetCode） 。

以下分别是等位队列和差位队列的写法。由于题目数据较大，等位队列进行了一定的压缩。

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        queue=deque([])
        res=[]
        l=0
        for num in nums:
            c=1
            while queue and queue[-1][0]<=num:
                c+=queue.pop()[1]
            queue.append([num,c])
            l+=1
            if l==k:
                res.append(queue[0][0])
                l-=1
                if queue[0][1]==1:
                    queue.popleft()
                else:
                    queue[0][1]-=1
        return res
```

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        n=len(nums)
        queue=deque([])
        res=[]
        for i in range(n):
            while queue and nums[i]>nums[queue[-1]]:
                queue.pop()
            queue.append(i)
            if i>=k-1:
                res.append(nums[queue[0]])
            if i-queue[0]==k-1:
                queue.popleft()
        return res
```

堆的应用就相对比较简单了。当我们需要实时查找一个动态列表中的最小值或最大值时，就可以用堆。

关于堆的经典题目OpenJudge - M18164:剪绳子：

```
import heapq
N=int(input())
heap=list(map(int,input().split()))
heapq.heapify(heap)
s=0
while len(heap)>1:
    a=heapq.heappop(heap)
    b=heapq.heappop(heap)
    s+=a+b
    heapq.heappush(heap,a+b)
print(s)
```

堆擅长查找操作，但不擅长修改操作。一旦一个元素入堆，除非它运动到堆顶，不然我们以后就再也找不到它了。因此很多情况下，堆要与其他数据结构配合使用，而其他数据结构的主要任务就是**标记**。

Dijkstra算法就是堆的经典应用场景。在常规的bfs中，不同路径具有相同的开销。如果不同路径的开销不同，那么就不一定满足最先到达的路径是最优路径。解决方法是，维护一个到达各个点所需开销的堆，每次取出其中开销最小的点，然后对于该点的领域点更新其开销。这样可以保证每次取出的开销最小的点一定是真实的最小开销。然而bfs要求去重，我们要求已经检查过的点不能再入堆，于是用一个列表或字典对检查过的点进行标记，当再次遇到这些点时就不再加入堆中了。

如OpenJudge - T20106:走山路，数据结构 `dict1` 起到了标记的作用。

```
import heapq
m,n,p=map(int,input().split())
grid=[]
for i in range(m):
    row=list(input().split())
    for j in range(n):
        if row[j].isdigit():
            row[j]=int(row[j])
    grid.append(row)
def check(x,y):
    return 0<=x<=m-1 and 0<=y<=n-1
for _ in range(p):
    x1,y1,x2,y2=map(int,input().split())
    if grid[x1][y1]=='#' or grid[x2][y2]=='#':
        print('NO')
        continue
    heap=[(0,x1,y1)]
```

```
    heapq.heapify(heap)
    dict1={(x1,y1):0}
    while heap:
        t,x,y=heapq.heappop(heap)
        if (x,y)==(x2,y2):
            print(t)
            break
        d=[(1,0),(-1,0),(0,1),(0,-1)]
        for i in range(4):
            x1=x+d[i][0]
            y1=y+d[i][1]
            if check(x1,y1) and grid[x1][y1]!='#':
                t1=t+abs(grid[x1][y1]-grid[x][y])
                if dict1.get((x1,y1),float('inf'))>t1:
                    heapq.heappush(heap,(t1,x1,y1))
                    dict1[(x1,y1)]=t1
    else:
        print('NO')
```

再来看一个例子，OpenJudge - T27256:当前队列中位数。求动态列表的中位数是堆的另一个重要应用。基本思路是，用大小相等的两个堆分别存储列表中较小的一半和较大的一半，把最接近中间的两个数推到堆顶，于是可以利用两个堆顶元素计算中位数。

如果要求列表具有删除操作，由于堆不支持查找堆内的任一元素，因此我们无法对该元素进行物理删除。解决方法是将这个被删除的数进行标记，对于这个数所在的堆，将它包含的有效元素数减一。这时就要从控制两个堆大小相同变为控制两个堆的有效元素相同。如果被删除的元素运动到了某个堆顶，再进行物理删除。这就是所谓的**懒删除**操作。

下面的史山代码，`minheap`和`maxheap`分别表示列表较小的一半和较大的一半，它们存储数对`(a,i)`，`a`表示数的实际大小，`i`表示数在列表中的索引。`l`表示列表末位元素的索引，`dl`表示被删除的那些元素的末位索引。`dict1`记录某个索引的数是在哪个堆中，数对`c`表示两个堆中待删除的元素数。希望你能看懂。

```
import heapq,re
import sys
data=iter(sys.stdin.read().split('\n'))
minheap=[]
maxheap=[]
l=0
dl=-1
c=[0,0]
dict1={}
n=int(next(data))
def rejust():
    while minheap and minheap[0][1]<=dl:
```

```
        t=heapq.heappop(minheap)
        del dict1[t[1]]
        c[0]-=1
    while maxheap and maxheap[0][1]<=dl:
        t=heapq.heappop(maxheap)
        del dict1[t[1]]
        c[1]-=1
    while len(maxheap)-c[1]+1<len(minheap)-c[0]:
        t=heapq.heappop(minheap)
        dict1[t[1]]=1
        heapq.heappush(maxheap,(-t[0],t[1]))
    while len(maxheap)-c[1]>len(minheap)-c[0]:
        t=heapq.heappop(maxheap)
        dict1[t[1]]=0
        heapq.heappush(minheap,(-t[0],t[1]))
for _ in range(n):
    command=next(data)
    if re.match('add',command):
        num=int(command[4:])
        if not minheap or num<=-minheap[0][0]:
            heapq.heappush(minheap,(-num,l))
            dict1[l]=0
            l+=1
            if len(maxheap)-c[1]+1<len(minheap)-c[0]:
                t=heapq.heappop(minheap)
                dict1[t[1]]=1
                heapq.heappush(maxheap,(-t[0],t[1]))
        else:
            heapq.heappush(maxheap,(num,l))
            dict1[l]=1
            l+=1
            if len(maxheap)-c[1]>len(minheap)-c[0]:
                t=heapq.heappop(maxheap)
                dict1[t[1]]=0
                heapq.heappush(minheap,(-t[0],t[1]))
        rejust()
    elif re.match('del',command):
        dl+=1
        c[dict1[dl]]+=1
        rejust()
    elif re.match('query',command):
        if len(minheap)-c[0]>len(maxheap)-c[1]:
            print(-minheap[0][0])
        else:
            a=(-minheap[0][0]+maxheap[0][0])/2
            if a==int(a):
```

```
            print(int(a))
        else:
            print(f'{a:.1f}')
```

数据结构真是奇妙，仅仅依靠简单的几种操作，就有如此千变万化的功能。想要灵活运用各种数据结构，只能依靠刷题不断地积累和总结。

二分查找

我们知道，编写二分查找算法最大的问题，就是边界条件问题。左、右端点从哪里开始，循环结束条件是什么，要不要加1，很多地方要考虑边界条件，稍有不慎就可能满盘皆输。

```
# 感受边界条件的复杂性
def binary_search(nums,target):
    l=___(1)___
    r=___(2)___
    while ___(3)___:
        ___(4)___
        guess=nums[mid]
        if guess==target:
            return mid
        elif guess<target:
            ___(5)___
        else:
            ___(6)___
    return None

'''
请选择可能的选项组合。
(1) A.0                B.-1
(2) A.len(nums)-1      B.len(nums)
(3) A.r>=l              B.r>l              C.r>l+1
(4) A.mid=(l+r)//2     B.mid=(l+r+1)//2
(5) A.l=mid            B.l=mid+1
(6) A.r=mid            B.r=mid-1
'''
```

本人在刷题过程中，逐渐摸索出了一个解决二分查找边界问题的通用方法。

如何解决边界问题？答案是让你的代码更具有偏向性。

什么是“偏向性”？

二分查找的边界问题，根本上是查找的两个端点 `l` 和 `r` 的边界性。**所谓偏向性，就是让查找的区间变为半开半闭区间，使得目标值可能取在其中某个端点，但绝不可能取在另一个端点。**

那么如何实现“偏向性”呢？答案是**让代码的方方面面都体现偏向性。**

听着像废话？那我们看例题吧。

首先我们编写最简单情形的二分查找代码：

> 输入一个已从小到大排序的、不包含重复元素的列表和一个目标值，判断目标值是否在列表中，如果是，输出其所在的索引位置，否则输出"No"。

下面两个代码分别是“左倾”和“右倾”版的二分查找代码：

```
nums=list(map(int,input().split()))
t=int(input())
n=len(nums)
l=0
r=n
while r-l>1:
    mid=(l+r)//2
    if nums[mid]==t:
        print(mid)
        break
    elif nums[mid]<t:
        l=mid
    else:
        r=mid
else:
    if nums[l]==t:
        print(l)
    else:
        print('No')
```

```
nums=list(map(int,input().split()))
t=int(input())
n=len(nums)
l=-1
r=n-1
while r-l>1:
    mid=(l+r+1)//2
    if nums[mid]==t:
        print(mid)
        break
    elif nums[mid]<t:
        l=mid
    else:
        r=mid
else:
    if nums[r]==t:
        print(r)
    else:
        print('No')
```

我们以“左倾”版的代码为例讲解“偏向性”的应用。

刚刚说过，所谓“偏向性”就是让查找区间变为半开半闭区间，在这里就是左闭右开区间。我们要求在查找过程中，目标值 `t` 所在索引可能等于 `l`，但一定严格小于 `r`。

首先看代码第4、5行：

```
l=0
r=n
```

列表 `nums` 的索引范围是 `[0,n-1]`，若目标值就是列表第一个值，那么其索引可能等于0，因此 `l` 从0开始。而索引不可能大于等于 `n`，因此 `r=n` 可以保证初始状态下目标值索引小于 `r`。

相应地，“右倾”版的代码就是 `l=-1`、`r=n-1`。

代码第6行：

```
while r-l>1:
```

我们设定了目标值索引绝不等于 `r`，因此区间端点 `l` 与 `r` 是不会重合的，结束条件是两者相邻，即 `r-l=1`。

代码第7行：

```
mid=(l+r)//2
```

我们希望查找过程更加偏向于 `l` 的方向，因此取 `mid=(l+r)//2`。当 `l` 与 `r` 的差为偶数时，`(l+r)//2` 与 `(l+r+1)//2` 没有区别，但当 `l` 与 `r` 的差为奇数时，`(l+r)//2` 会使 `mid` 偏向于 `l`，`(l+r+1)//2` 会使 `mid` 偏向于 `r`。

代码第11—14行：

```
elif nums[mid]<t:
    l=mid
else:
    r=mid
```

这一段涉及二分查找代码的另一个重要思想，即“**绝对性**”。**我们要保证，如果目标值存在，那么它一定时刻在 `[l,r)` 区间以内。**取 `l=mid+1` 或 `r=mid-1` 都是不合适的。我们考察下面这种特殊情况：

```
t=5
nums[mid]=4
nums[mid+1]=6
```

这时候 `nums[mid]<t`，但如果我们取 `l=mid+1` 的话，区间 `[l,r)` 就不再包含 `t` 了，这是不允许的。同理，`r=mid-1` 也是不行的。

代码第15—19行：

```
else:
    if nums[l]==t:
        print(l)
    else:
        print('No')
```

如果执行最后的 `else` 模块，说明之前没有找到 `t`，但区间已经完全合并，`r=l+1`。不过我们还没有判断 `l` 或 `r` 是否满足条件。这时候“偏向性“的效果就出来了，如果 `t` 确实在列表中，那么它一定是 `l` 所在的位置，最后判断 `nums[l]` 与 `r` 是否相同即可。

> **选读内容 关于循环条件的进一步讨论**
>
> 可能会有人觉得，我们都退出循环体了，还要做最后一次检查，感觉有点臃肿。能不能在循环体中就完成全部的检查呢？
>
> 不行。这样的话，我们需要在区间已经合并，即 `r=l+1` 的情况下，对 `l` 再做一次判断，这就需要把循环条件改为：

```
while r>l:
```

> 如果最后 `t` 刚好位于 `nums[l]` 和 `nums[r]` 之间且不等于 `nums[l]`，那么执行流程就是：

```
r-l=1
-> 循环条件满足
-> mid=(l+r)//2 # 等于l
-> nums[l]<t
-> l=mid # 等于l
-> r-l=1
-> 循环条件满足
-> mid=(l+r)//2 # 等于l
......
```

> 陷入死循环了。

> 我调查了一下题解里的所有二分查找代码，发现了下面的现象：
>
> - 如果循环条件为 `r>=l`，那么迭代条件就同时为 `l=mid+1` 、 `r=mid-1`
> - 如果循环条件为 `r>l`，那么迭代条件中有一个为 `l=mid+1` 或 `r=mid-1`，另一个是 `l=mid` 或 `r=mid`
> - 如果循环条件为 `r-l>1`，那么迭代条件就同时为 `l=mid` 、 `r=mid`
>
> 首先，在上面三种写法中，只有最后一种能确保满足绝对性要求，理由在前面已经说过了。在某些题目条件下，前两种写法也能满足绝对性要求，如待查找列表为连续的自然数列表，在 `mid` 和 `mid+1` 之间没有其他数，因此即使取 `l=mid+1` 或 `r-mid-1` 也不会产生遗漏。当待查找列表为非连续列表时，前两种写法就无法满足绝对性要求。
>
> 另外还要指出的是，绝对性和偏向性并不是必需的，有些二分查找代码即使不满足绝对性和偏向性，也能得到正确的结果。但如果代码能够满足这两个要求，将为我们带来极大的便利，我们后面会感受到这一点。

由此可见，我们确实做到了在代码的方方面面都体现出偏向性。“偏向性”使得查找过程有了明确的方向，从而为边界条件的处理提供了逻辑支撑。我们不用盲目调节边界条件的状态，只要使其满足偏向性要求，就能妥善处理好所有边界条件。

接下来我们看一下“偏向性”的进阶应用。事实上，在很多题目中，偏向的方向是不能随意规定的。比如下面这道题：

> 34. 在排序数组中查找元素的第一个和最后一个位置 - 力扣（LeetCode）
> 给你一个按照非递减顺序排列的整数数组 `nums`，和一个目标值 `target`。请你找出给定目标值在数组中的开始位置和结束位置。
> 如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

我们要进行两次二分查找，分别找出 `target` 的开始位置 `a` 和结束位置 `b`。

以开始位置为例。现在我们需要让查找偏向哪边呢？

由绝对性的要求，`target` 的开始位置 `a` 位于 `l` 和 `r` 之间，因此 `l` 不可能在 `a` 、 `b` 之间，而 `r` 有可能。当 `r` 位于 `[a,b]` 范围内时，恒有 `nums[r]==target`，但因为要找的是开始位置，我们依旧要减小范围继续寻找！说不定哪次查找以后，刚好 `r==a`！因此我们无法让 `r` 绝不取到 `a`。但通过使 `nums[l]<target`，我们可以保证 `l` 绝对不会取到 `a`。这也就限定了本次查找只能偏向右方。

查找开始位置的代码如下：

```
l=-1
r=n-1
```

```
while r-l>1:
    mid=(l+r+1)//2
    num=nums[mid]
    if target>num:
        l=mid
    elif target<=num:
        r=mid
a=r
```

同理，查找结束位置时只能偏向左边。

```
l=0
r=n
while r-l>1:
    mid=(l+r)//2
    num=nums[mid]
    if target>=num:
        l=mid
    elif target<num:
        r=mid
 b=l
```

综合两端代码，再考虑 `target` 不在列表中的情况，就得到了最终代码。

```
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        n=len(nums)
        if n==0:
            return [-1,-1]
        l=-1
        r=n-1
        while r-l>1:
            mid=(l+r+1)//2
            num=nums[mid]
            if target>num:
                l=mid
            elif target<=num:
                r=mid
        a=r
        if nums[a]!=target:
            return [-1,-1]
        l=0
        r=n
        while r-l>1:
            mid=(l+r)//2
```

```
            num=nums[mid]
            if target>=num:
                l=mid
            elif target<num:
                r=mid
        b=l
        return [a,b]
```

我们再看一道例题：

> 74. 搜索二维矩阵 - 力扣（LeetCode）
> 给你一个满足下述两条属性的 `m*n` 整数矩阵 `matrix`：
> (1)每行中的整数从左到右按非严格递增顺序排列
> (2)每行的第一个整数大于前一行的最后一个整数
> 给你一个整数 `target`，如果 `target` 在矩阵中，返回 `True`，否则返回 `False`。

整体思路比较容易，我们进行两次二分查找，首先在每行的第一个数中查找，确定 `target` 所在的行，再在该行中查找 `target`。

我们设每行第一个数组成的列表为 `head`。

```
head=[row[0] for row in matrix]
```

现在的问题是，我们在 `head` 中二分查找时，应该偏向哪边呢?

假设我们在某时刻确定 `target` 位于第 `l` 行与第 `r` 行之间，由于每行元素从左到右是递增的，当 `target>head[l]` 时， `target` 仍有可能位于第 `l` 行中；但当 `target<head[r]` 时，由于第 `r` 行中其他元素都大于 `head[r]`，因此 `target` 绝不可能在第 `r` 行中。由此确定了我们的二分查找偏向左方。当 `l` 与 `r` 合并时，便可以确定 `target` 一定位于第 `l` 行中。

```
head=[row[0] for row in matrix]
l=0
r=m
while r-l>1:
    mid=(l+r)//2
    num=head[mid]
    if num==target:
        return True
    elif num<target:
        l=mid
    else:
        r=mid
row=matrix[l]
```

接下来在第 `l` 行中查找。这次应该偏向哪个方向呢？与之前不同，这次要偏向右方。因为第 `l` 行的首个元素一定不是 `target`。如果首个元素是 `target` 的话，上一步查找 `head` 的时候就应该已经找到了，就不会进行第二次查找了。

```
l=-1
r=n-1
while r-l>1:
    mid=(l+r+1)//2
    num=row[mid]
    if num==target:
        return True
    elif num<target:
        l=mid
    else:
        r=mid
return True if row[r]==target else False
```

综合两部分，就得到了最终代码。

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        m=len(matrix)
        n=len(matrix[0])
        head=[row[0] for row in matrix]
        l=0
        r=m
        while r-l>1:
            mid=(l+r)//2
            num=head[mid]
            if num==target:
                return True
            elif num<target:
                l=mid
            else:
                r=mid
        row=matrix[l]
        l=-1
        r=n-1
        while r-l>1:
            mid=(l+r+1)//2
            num=row[mid]
            if num==target:
                return True
            elif num<target:
                l=mid
```

```
            else:
                r=mid
        return True if row[r]==target else False
```

这道题是关于绝对性的例子：

> 153. 寻找旋转排序数组中的最小值 - 力扣（LeetCode）
> 已知一个长度为 `n` 的数组，预先按照升序排列，经由 `1` 到 `n` 次 **旋转** 后，得到输入数组。例如，原数组 `nums = [0,1,2,4,5,6,7]` 在变化后可能得到：
>
> - 若旋转 `4` 次，则可以得到 `[4,5,6,7,0,1,2]`
> - 若旋转 `7` 次，则可以得到 `[0,1,2,4,5,6,7]`
>
> 注意，数组 `[a[0], a[1], a[2], ..., a[n-1]]` **旋转一次** 的结果为数组 `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]` 。
>
> 给你一个元素值 **互不相同** 的数组 `nums` ，它原来是一个升序排列的数组，并按上述情形进行了多次旋转。请你找出并返回数组中的 **最小元素** 。

为什么我们要求二分查找过程有绝对性？这个问题可以从两个方面考虑。

- 必要性。如果我们在查找时不小心让 `target` 掉出了 `l` 和 `r` 所限定的范围，那么就失去了判断 `target` 是否存在的能力，这样的二分查找是没有意义的。
- 充分性。如果我们保证 `target` 始终在 `l` 和 `r` 之间，再结合偏向性，当最终区间完全合并，即 `r==l+1` 时， `target` 一定就是我们偏向的那个端点值。

本题正是利用了绝对性与偏向性结合带来的好处。

首先分析一下思路。乍一看，这题的数组不满足升序条件，无法使用二分查找。事实上，本题是一种推广的二分查找，我们通过每次把待搜索的数列减半来找到目标值。

由于我们无法提前知道，对于数列中的某个数，是否在列表中有比它更小的数，因此我们无法用类似于 `nums[mid]==target` 这样的方式找到目标。因此我们换一种思路，干脆让二分查找一直进行下去，直到区间完全合并，即 `r==l+1` ，这样两个端点值中我们偏向的那个就一定是目标。

至于具体的查找过程，本次的代码显然要偏向右边。因为对于大多数情况，数组分为左半序列和右半序列，左半序列的最小元素大于右半序列的最大元素，因此整个数组最小值一定在右半序列而非左半序列。唯一一种例外是输入数组根本没有旋转，还是正常的升序数组，此时直接输出首位即可。

```
if nums[0]<nums[n-1]:
    return nums[0]
```

对于中间值`mid`，它可能在左半序列，也可能在右半序列。为了满足绝对性条件，我们要求数列的最小值，即右半序列的首位值始终在`l`和`r`之间。这样就很清楚了，当`mid`在左半序列时，令`l=mid`，当`mid`在右半序列时，令`r=mid`，这样就满足绝对性条件了。

最终代码：

```
class Solution:
    def findMin(self, nums: List[int]) -> int:
        n=len(nums)
        if n==1:
            return nums[0]
        if nums[0]<nums[n-1]:
            return nums[0]
        l=0
        r=n-1
        while r-l>1:
            mid=(l+r+1)//2
            if nums[mid]>nums[l]:
                l=mid
            elif nums[mid]<nums[r]:
                r=mid
        return nums[r]
```

下面这道题我愿称之为全OJ最强二分查找题：

> OpenJudge - M08210:河中跳房子
>
> 每年奶牛们都要举办各种特殊版本的跳房子比赛，包括在河里从一个岩石跳到另一个岩石。这项激动人心的活动在一条长长的笔直河道中进行，在起点和离起点L远 (1 ≤ L≤ 1,000,000,000) 的终点处均有一个岩石。在起点和终点之间，有N(0 ≤ N ≤ 50,000) 个岩石，每个岩石与起点的距离分别为Di (0 < Di < L)。
>
> 在比赛过程中，奶牛轮流从起点出发，尝试到达终点，每一步只能从一个岩石跳到另一个岩石。当然，实力不济的奶牛是没有办法完成目标的。
>
> 农夫约翰为他的奶牛们感到自豪并且年年都观看了这项比赛。但随着时间的推移，看着其他农夫的胆小奶牛们在相距很近的岩石之间缓慢前行，他感到非常厌烦。他计划移走一些岩石，使得从起点到终点的过程中，最短的跳跃距离最长。他可以移走除起点和终点外的至多M (0 ≤ M ≤ N) 个岩石。
>
> 请帮助约翰确定移走这些岩石后，最长可能的最短跳跃距离是多少？

这题是二分查找与贪心的结合。

首先是逆向思维，不直接寻找最长的最短跳跃距离，而是在一定范围内枚举最短跳跃距离 `d`。对每个最短跳跃距离 `d`，回答以下问题：为了使每次跳跃距离均大于等于 `d`，我们至少需要拿走多少块石头？这是本题的贪心部分。

我们遍历石头坐标列表 `stones`，贪心策略是：

- 如果当前石头坐标 `stones[i]` 与上一个我们保留的石头坐标 `cd` 的间距小于 `d`，那么拿走这块石头，石头数 `m+=1`
- 如果当前石头坐标 `stones[i]` 与 `cd` 的间距大于等于 `d`，那么保留这一块石头，将 `cd` 更新为当前坐标 `stones[i]`
- 如果当前石头坐标 `stones[i]` 与终点 `L` 的间距小于 `d`，那么将这块石头连同后面所有石头全部拿走，石头数 `m+=N-i`

这样得出至少需要拿走的石头数就是 `m`。

然后就是二分查找部分了。我们枚举的最短跳跃距离 `d`，其范围是从 `1` 到 `L` 的自然数列，它是顺序增加的，因此为了提高查找效率，我们将枚举改为二分查找。

如果按照传统二分查找的观点，那么二分查找流程就是这样：

- 如果石头数 `m>M`，说明当前距离 `mid` 偏大，将二分查找的右端点 `r` 更新为 `mid` 值
- 如果石头数 `m<M`，说明当前距离 `mid` 偏小，将二分查找的左端点 `l` 更新为 `mid` 值

但是仔细推敲后发现，本题的判断过程并没有那么简单。

- 首先，如果石头数 `m==M`，不能直接输出当前 `mid` 值，因为显然可能有多个 `d` 值使 `m==M`，而我们要找其中的最大值
- 其次，可能不存在 `d` 使得 `m==M`，也就是说我们待求的目标值 `d_target`，在贪心策略下拿走的石头数小于 `M`，但 `d_target+1` 对应的石头数就大于 `M` 了。比如

```
L=9
N=2
M=1
stones=[3,6]
```

此时在起点 `x=0` 和终点 `x=9` 中间有两个石头 `x=3` 和 `x=6`。显然待求值 `d_target=3`。当 `d==3` 时，由于每次跳跃距离大于等于3即可，故贪心策略驱使我们不拿走一个石头，`m==0`。而当 `d==4` 时，我们不得不把两个石头都拿走，`m==2`。不存在 `d` 使得 `m==M`。

我们可以这样理解：当 `m<M` 时，说明 `d` 比较小，我们手头上的石头比较“宽裕”。如果我们多拿几个石头，最短的跳跃距离只会增大，仍然大于等于 `d`，也满足条件。这样的 `d` 可能是也可能不是我们要找的目标值。

这使得传统的二分查找不起作用了。但我们发现，在本题的模型下，有一点是保证满足的。

- **如果石头数 `m>M`，说明当前距离 `d` 偏大，`d` 绝对不可能是目标值**

而事实上，我们二分查找只要有这一点就够了。因为它为我们提供了一个偏向性条件：`d` 可能取查找区间中较小的一端，但绝不可能取查找区间中较大的一端。因此我们的二分查找偏向左方。

初始条件：取 `l` 为可能的最小值 `1`，`r` 为一个绝不可能的值 `L+1`。

中心值 `mid=(l+r)//2`

- 如果 `mid` 对应的石头数 `m>M`，则 `r=mid`
- 对于一切其他情况，`l=mid`

一直进行下去，直到区间完全合并，即 `r-l=1`。此时的 `l` 值就是我们待求的，最长的最短跳跃距离。

最终代码：

```
L,N,M=map(int,input().split())
stones=[]
for _ in range(N):
    stones.append(int(input()))
def stone(d):
    m=0
    cd=0
    for i in range(N):
        if L-stones[i]<d:
            m+=N-i
            break
        if stones[i]-cd<d:
            m+=1
        else:
            cd=stones[i]
    return m
l=1
r=L+1
while r-l>1:
    mid=(l+r)//2
    if stone(mid)>M:
        r=mid
    else:
        l=mid
print(l)
```

从这道题我们学到了二分查找算法的一个核心思想：**抓住主要矛盾**。这已经上升到哲学的高度了。在二分查找的过程中，系统的情况可能非常复杂。在这样的条件下，我们要牢牢抓住系统最

核心的特征，忽略其他可能产生干扰的特征，然后利用好这个核心特征解决问题。所谓“偏向性”正是充分利用了查找过程中某个方向的“绝对不可能”条件，实现查找过程的优化。

我们再看一道例题：

> OpenJudge - 28972:海拔
>
> 一片矩形地域被横平竖直地切分成了`n*m`片方形区块。 其中位于第`i`行第`j`列的区块的平均海拔是`h`。
>
> 某人要从第`1`行第`1`列的区块移动至第`n`行第`m`列的区块。每次移动时，她只能选择一个与当前所处区块有公共边的相邻区块，并移动至该区块。
>
> 跨越处于不同海拔的区块是相当耗费体力的.。定义一次移动的体力消耗值为该次移动涉及到的两个区块的海拔之差，某人希望你能够帮助她找到一条能顺利抵达目的地的路径，使得所有移动中体力消耗值的最大值尽可能小。

这题是`bfs`与二分查找的结合，整体思路与上题非常类似。

我们首先用二分查找确定某一次的体力消耗最大值`h`，查找范围最小值是`0`，最大值是图中最大高度与最小高度之差。对于确定的`h`，我们定义`bfs(h)`函数，判断在要求各次移动高度差小于等于`h`的情况下，能否从起点到达终点。如果能则返回`True`，否则返回`False`。

至于偏向性，由于在`h`过小时绝对不可能，因此我们的代码偏向右边。初始时`l=-1`，`r=max(height)-min(height)`，只要当前的高度差`mid`能到达终点，就令`r=mid`。最后输出`r`值即可。

```
from collections import deque
n,m=map(int,input().split())
grid=[]
for _ in range(n):
    grid.append(list(map(int,input().split())))
d=[(1,0),(-1,0),(0,1),(0,-1)]
def bfs(h):
    queue=deque([(0,0)])
    condition=[[False]*m for _ in range(n)]
    condition[0][0]=True
    while queue:
        x,y=queue.popleft()
        for dx,dy in d:
            x1=x+dx
            y1=y+dy
            if 0<=x1<=n-1 and 0<=y1<=m-1 and not condition[x1][y1]:
                if abs(grid[x1][y1]-grid[x][y])<=h:
                    if (x1,y1)==(n-1,m-1):
                        return True
```

```
                    queue.append((x1,y1))
                    condition[x1][y1]=True
    else:
        return False
height=[grid[i][j] for i in range(n) for j in range(m)]
l=-1
r=max(height)-min(height)
while r-l>1:
    mid=(l+r+1)//2
    if bfs(mid):
        r=mid
    else:
        l=mid
print(r)
```

动态规划

> 动态规划有递归式写法和递推式写法两种。递归式写法并入深搜部分讲，这里主要介绍递推式写法。

动态规划是一类相当灵活的算法，它利用已知的子问题结果计算新的问题的结果。就像你计算斐波那契数列 `F(n)`，肯定是先计算出 `F(n-1)` 和 `F(n-2)`，再计算出 `F(n)`。

动态规划问题一定满足以下三个特征：

- 当前问题的结果只与有限个子问题的结果有关
- 子问题与原问题有相同的结构
- 子问题的结果均已算出

上面三个特征分别对应dp的**递推公式**，**参数**，**递推顺序**三个核心要求。

用dp解决一个问题的大致过程如下：

一、确定描述系统状态的参数

一旦参数给定，则子问题的结果就是确定的。

开一个 `dp` 数组，其大小满足

```
dp的大小=系统所含基本元素数*每个基本元素所含参数数
```

所谓“基本元素”就是我们考虑问题的单元，或者说就是子问题。如“小偷背包”问题，基本元素就是每个可以偷的物品。每个元素的参数完全描述了该元素的各种状态，如“打家劫舍”中，偷与不偷分别为两种状态。

二、确定递推关系

找出当前问题的结果与子问题结果之间的关系，并用数学公式表示之。这类递推关系往往含有 `max`，`min`，`sum` 等元素。

三、确定递推顺序

将初始子问题的结果存入 `dp` 中，递推顺序要保证计算当前问题时，子问题的结果均已算出。由此递推算出所有子问题的结果。

下面我们分析动态规划的各种题型与策略，加深对动态规划的理解。

- 当前状态只与相邻状态有关

这是最简单类型的dp问题。

最经典的题目如OpenJudge - E23556:小青蛙跳荷叶，用 `dp[i]` 表示跳到第 `i` 片荷叶的方案数。青蛙既可以选择从前一个荷叶跳上来，也可以选择从前两个荷叶跳上来，因此 `dp[i]=dp[i-1]+dp[i-2]` 。

```
n=int(input())
dp={}
dp[0]=1
dp[1]=1
for i in range(2,n+1):
    dp[i]=dp[i-1]+dp[i-2]
print(dp[n])
```

如果状态传递时存在条件限制，本质上也是一样的。如Problem - 474D - Codeforces，为了摆出 `i` 朵花，我们既可以选择在 `i-1` 朵花后放一朵红花，也可以选择在 `i-k` 朵花后放 `k` 朵白花，后者要求 `i>=k` 。

```
t,k=map(int,input().split())
list1=[]
m=0
for _ in range(t):
    a,b=map(int,input().split())
    m=max(m,a,b)
    list1.append((a,b))
dp=[1]+[0]*m
M=10**9+7
for i in range(1,m+1):
    dp[i]+=dp[i-1]
    dp[i]%=M
    if i>=k:
        dp[i]+=dp[i-k]
        dp[i]%=M
dp1=[0]
for i in range(1,m+1):
    dp1.append((dp1[-1]+dp[i])%M)
for t in list1:
    print((dp1[t[1]]-dp1[t[0]-1])%M)
```

- 当前状态与过去所有状态有关

如OpenJudge - M02757:最长上升子序列，用 `dp[i]` 表示序列的第 `0-i` 个元素能组成的最长上升子序列长度。对于第 `i` 个元素，凡是满足 `j<i` ， `nums[j]<nums[i]` 的元素 `j` 都可以成为第 `i` 个元素的前一个元素。而且我们发现无法用有限个参数表示第 `i` 个元素前所有满足条件的元素，因此递推时就遍历从 `0` 到 `i-1` 的所有元素。

```
N=int(input())
nums=list(map(int,input().split()))
dp=[0]*N
for i in range(N):
    s=1
    for j in range(i):
        if nums[j]<nums[i]:
            s=max(s,1+dp[j])
    dp[i]=s
print(max(dp))
```

OpenJudge - M03532:最大上升子序列和，与上题本质上是一致的，只是不同数的权重不同。

```
N=int(input())
nums=list(map(int,input().split()))
dp=[0]*N
for i in range(N):
    s=nums[i]
    for j in range(i):
        if nums[j]<nums[i]:
            s=max(s,nums[i]+dp[j])
    dp[i]=s
print(max(dp))
```

- 二次动态规划

这是非常重要的一种思想。如果当前问题的结果不仅与之前的数有关，还与之后的数有关，就分别从前往后、从后往前进行一次动态规划，然后对两次结果进行整合。

如OpenJudge - M02995:登山，我们要找到一个位置，满足其左侧的最长上升子序列和其右侧的最长下降子序列的长度之和最大。既然这样，我们就进行两次动态规划， `dp1` 记录以某点为结尾的最长上升子序列长度， `dp2` 记录以某点为开头的最长下降子序列长度，把两者加起来就是待求的值。

```
N=int(input())
height=list(map(int,input().split()))
dp1=[1]*N
dp2=[1]*N
```

```
for i in range(1,N):
    for j in range(i):
        if height[j]<height[i]:
            dp1[i]=max(dp1[i],dp1[j]+1)
for i in range(N-2,-1,-1):
    for j in range(i+1,N):
        if height[j]<height[i]:
            dp2[i]=max(dp2[i],dp2[j]+1)
s=0
for i in range(N):
    s=max(s,dp1[i]+dp2[i]-1)
print(s)
```

OpenJudge - M04121:股票买卖，这题是动态规划与贪心的结合。假如我们会算在股票市场中买卖一次能获得的最大收益，我们如何计算买卖两次能获得的最大收益？对于列表中的每个点，我们计算在它之前的部分买卖一次的最大收益，以及在它之后的部分买卖一次的最大收益，分别记录在两个`dp`中。然后将两者加起来。

```
for _ in range(int(input())):
    N=int(input())
    price=list(map(int,input().split()))
    dp1=[0]*N
    m=float('inf')
    s=0
    for i in range(N):
        if price[i]<m:
            m=price[i]
        elif price[i]-m>s:
            s=price[i]-m
        dp1[i]=s
    dp2=[0]*N
    M=-float('inf')
    s=0
    for j in range(N-1,-1,-1):
        if price[j]>M:
            M=price[j]
        elif M-price[j]>s:
            s=M-price[j]
        dp2[j]=s
    print(max(map(lambda x:dp1[x]+dp2[x],range(N))))
```

OpenJudge - M20744:土豪购物，这题允许两种选取商品的方式：一种是直接选取连续的几个商品；另一种是选取几个连续的商品，然后放回其中一个——我们也可以把这种情况视为，以某一个商品为界，选取其左侧最大价值的连续序列，和其右侧最大价值的连续序列。我们注意到第二

种选法可由第一种选法推导而来。因此我们进行两次动态规划，每一次都是Kadane算法，分别计算以某点为结尾的连续序列的最大价值，和以某点为开头的连续序列的最大价值，然后将两者进行整合。

```
price=list(map(int,input().strip().split(',')))
n=len(price)
if n==1:
    print(price[0])
else:
    dp1=[0]*n
    dp1[0]=price[0]
    for i in range(1,n):
        dp1[i]=max(price[i],dp1[i-1]+price[i])
    dp2=[0]*n
    dp2[-1]=price[-1]
    for j in range(n-2,-1,-1):
        dp2[j]=max(price[j],dp2[j+1]+price[j])
    s=-float('inf')
    for i in range(n):
        s=max(s,dp1[i])
    for j in range(n-2):
        s=max(s,dp1[j]+dp2[j+2])
    print(s)
```

- 多维dp

前面我们讨论的都是一维 `dp`，即系统状态只有一个参数的情况。当系统状态需要多个参数才能确定时，就要建立多维 `dp`。

最经典的多维 `dp` 莫过于OpenJudge - M23421:《算法图解》小偷背包问题。

我们想知道当我们的背包承重为 `B` 时应如何装 `N` 个物品使其价值最大，这里涉及到两个参数，一个是背包承重，一个是物品种类。因此，从原理上来讲，我们应该建立一个二维 `dp`，其大小为 `N*B`， `dp[i][j]` 表示我们考虑物品列表中的前 `i` 个物品，当背包承重为 `j` 时能装的最大价值。

但是注意到在物品维度上，系统的状态只与前一个状态有关，即 `dp[i]` 只与 `dp[i-1]` 有关，因此我们可以把二维列表变为滚动一维列表以节省空间。当你以后做动态规划题，注意到某个维度的当前值只与之前有限个值有关时，都可以把这个维度变成滚动的以节省空间。

```
N,B=map(int,input().split())
cost=list(map(int,input().split()))
weigh=list(map(int,input().split()))
dp=[0]*(B+1)
for i in range(N):
```

```
    c=cost[i]
    w=weigh[i]
    dp1=[0]*(B+1)
    for j in range(1,B+1):
        if j>=w:
            dp1[j]=max(dp[j],dp[j-w]+c)
    dp=dp1[:]
print(dp[-1])
```

对于多维dp问题，尤其要注意递推顺序。比如下面这道题：

OpenJudge - M27217:有多少种合法的出栈顺序

我们对操作数序列和栈可以进行两种操作：

- 将操作数序列的首个数推入栈中
- 将栈顶元素弹出，移到输出序列尾端

这题的状态有两个参数，一个是操作数序列和栈中的总元素数 `m`，一个是操作数序列中的元素数 `a`，满足 `a<=m`。我们用 `dp[m][a]` 表示上述状态下系统能产生的输出序列个数。

根据题意，在状态 `(m,a)` 下，我们要么将操作数序列的首个数推入栈中，系统状态变为 `(m,a-1)`，要么将栈顶元素弹出，系统状态变为 `(m-1,a)`。由此可以确定状态转移方程：

```
dp[m][a]=dp[m][a-1]+dp[m-1][a-1]
```

接下来考虑边界条件。题目存在两种边界条件：

- 如果 `a=0`，说明所有元素均已在栈中，只能将其逐个弹出，因此 `dp[m][0]=0`
- 如果 `a=m`，说明所有元素均在操作数序列中，第一步肯定是将其推入栈中，因此 `dp[m][m]=dp[m][m-1]`

至于递推顺序，我们要求在递推时子问题的状态均已求出。根据前面的递推公式，我们想要知道 `dp[m][a]`，需要先求出 `dp[m][a-1]` 和 `dp[m-1][a]`，因此我们的递推顺序是，首先从 `1` 开始递增 `m`，对每一个 `m`，从 `0` 开始递增 `a` 直到 `m`。

最终代码：

```
n=int(input())
if n==1:
    print(1)
else:
    dp=[[1],[1,1]]
    for m in range(2,n+1):
        dp.append([0]*(m+1))
```

```
        dp[m][0]=1
        for a in range(1,m):
            dp[m][a]=dp[m][a-1]+dp[m-1][a]
        dp[m][m]=dp[m][m-1]
    print(dp[n][n])
```

- 隐藏参数类问题

难度较高的 dp 题，一般都是隐藏参数类问题，这种题让你感觉无法把握系统状态，难以建立递推方程。

对于这种 dp 题，首先要对动态规划算法的核心思想具有深刻的理解与把握，其次要具有敏锐的观察力，把握描述系统状态的所有可能参数，然后利用它们建立递推方程。如果你感觉因为系统的某个状态不确定而导致很难进行递推，那么就把那个状态也变成参数，其结果是你的dp表升高了一维。

OpenJudge - M19164:移动办公

我们在第 i 个月，如果选择在北京办公，可以获得 a 的营业额；如果选择在南京办公，可以获得 b 的营业额。在两者之间往来的交通费是 M，现在需要求出最大的总营业额。

在建立递推方程时，我们遇到了障碍。如果 dp[i] 表示到第 i 个月为止的最大营业额，当要根据 dp[i-1] 求解 dp[i] 时，由于我们不知道上个月是在北京还是南京，因此难以计算。

注意到了吗？上个月的地点的不确定性成为确定系统状态的一个障碍。既然这样，那我们就把上个月的地点确定下来，把它也变成系统的一个参数。

设 dp[i] 表示一个二元数对 [a,b]， a 表示如果我们第 i 个月最后待在北京，能获得的最大收益， b 表示如果我们第 i 个月最后待在南京，能获得的最大收益。

这样状态转移方程就容易建立了。如果我们第 i 个月最后待在北京，要么我们上个月本来就在北京，这个月在北京办公获得了 a 的收益；要么我们上个月在南京，这个月先在南京办完工，获得了 b 的收益，然后为了下个月获得更多收益而坐车转移到北京。因此

```
dp[i][0]=max(dp[i-1][0]+a,dp[i-1][1]+b-M)
```

同理，如果我们第 i 月最后在南京，可以建立类似的方程：

```
dp[i][1]=max(dp[i-1][0]+a-M,dp[i-1][1]+b)
```

递推完成后，我们取最后一个月的两种状态中收益较大的值即可。

```
T,M=map(int,input().split())
dp=[[0,0] for _ in range(T+1)]
for i in range(1,T+1):
    a,b=map(int,input().split())
    dp[i][0]=max(dp[i-1][0]+a,dp[i-1][1]+b-M)
    dp[i][1]=max(dp[i-1][0]+a-M,dp[i-1][1]+b)
print(max(dp[-1]))
```

对于有些题来说，隐藏参量不是那么容易发现的。比如

OpenJudge - T04117:简单的整数划分问题

将正整数 `n` 表示为一系列单调递减的正整数之和 `n1+n2+···+nk`， `n1>=n2>=···>=nk`， `k>=1`，问有多少种划分种类数。

乍一看可能没有什么思路。既然这样，可以拿出草稿纸比划一下，寻找规律。

比如 `n=5`，我们有以下几种划分：

```
5=5
5=4+1
5=3+2
5=3+1+1
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

刚开始，我们可能会这样建立 `dp` 表： `dp[n]` 表示对 `n` 进行划分的种类数。对于所有小于 `n` 的数 `m`，我们可以把 `m` 放在最前面，剩下的序列就是对 `n-m` 进行划分，因此 `dp[n]` 就是对所有可能的 `m` 求和。

但是经过进一步讨论，这个思路是不正确的。如上面对 `5` 的划分中，有一个 `5=2+2+1`。我们将 `2` 放在 `5` 的划分的首位，按理来讲后面就应该是 `3` 的划分的种类数，但事实不是这样，因为 `5=2+3` 是不合法的。究其原因，我们要求划分是单调递减的，对于 `3` 的划分 `3=3`，其首位 `3` 大于前面的 `2`，因此不合法；对于 `3=2+1`，其首位小于等于前面的 `2`，因此可以放在 `2` 后面，是一种划分。

注意到了吗？在前面的分析中，高频地出现一个词：首位数。也就是说，这就是描述划分的第二个参数。

我们设 `dp[n][a]` 表示对 `n` 进行划分，划分中首位数为 `a` 时的划分种类数。在此情况下，首位数 `a` 后面的划分就应该是对 `n-a` 的划分，其满足首位数小于等于 `a`。由此建立转移方程：

```
dp[n][a]=sum([dp[n-a][i] for i in range(1,a+1)])
```

对 `n` 划分的种类数就是对各种可能的首位数 `a` 求和后的总数。

最终代码：

```
dp=[[0],[0,1]]
for n in range(2,51):
    row=[0]*(n+1)
    row[-1]=1
    for a in range(n-1,0,-1):
        row[a]=sum(dp[n-a][1:a+1])
    dp.append(row)
while True:
    try:
        N=int(input())
    except EOFError:
        break
    print(sum(dp[N]))
```

接下来的题目涉及到一个哲学观点。唯物辩证法的矛盾观认为，矛盾着的双方是相互依存、相互作用的。一方面，矛盾双方互相依赖对方而存在，双方共处一个统一体中；另一方面，矛盾双方在一定条件下可以相互转化。具体到题目中，就是当系统处于某一种二元状态时，你不仅要注意到目标状态，还要注意到目标状态的反面，这同样是系统的一个参数。

OpenJudge - T25573:红蓝玫瑰

有一些红色或蓝色玫瑰排成一排，你可以进行两种操作：一是将某一株玫瑰颜色反转，一是将某一株玫瑰及其左侧的所有玫瑰颜色反转。问最少需要多少次操作，将所有玫瑰变成红玫瑰。

首先，我们注意到，我们对若干颜色相同且连续的玫瑰，执行的应该是相同的操作，因此先对输入数据进行压缩。我们将玫瑰排列表示成形如 `['R',1]`，`['B',2]` 的形式，第一位表示玫瑰颜色，第二位表示连续玫瑰的数量。这一点可以用栈来实现，可以参考前面的高级数据结构部分。

```
roses=list(input())
stack=[[roses[0],1]]
for a in roses[1:]:
    if a==stack[-1][0]:
        stack[-1][1]+=1
    else:
        stack.append([a,1])
```

我们从左到右遍历上面的压缩列表，设列表中第 `i` 位玫瑰的颜色为 `a`，数量为 `l`。如果第 `i` 位玫瑰的颜色是红色，那么很高兴，我们不用做任何操作，进入下一位；如果第 `i` 位玫瑰的颜色是蓝色，我们有以下两种操作：

要么逐株使用第一种魔法，将所有蓝色玫瑰变成红色玫瑰，对应的总操作数为蓝玫瑰的数量`l`，加上将前面的列表变红的操作数。

要么使用第二种魔法，将蓝色玫瑰连同前面的所有玫瑰反转，然后再将前面的列表的反转列表变成全红色。这样的话总操作数就是`1`加上将前面的列表的反转列表全变红的操作数。但是我们并不知道所谓"将前面的列表的反转列表全变红的操作数"，因此讨论陷入停滞。

这时我们注意到了关键词"反转列表"。一个列表与其反转列表，这难道不是一种参数吗？

我们建立两个表`dp1`和`dp2`，其中`dp1[i]`表示将原列表的前`i`项全变红所需的操作数，`dp2`表示如果我们手头的初始列表是原列表的前`i`项的反转列表，将其全部变红所需的操作数。

接下来建立递推方程。设第`i`组玫瑰的颜色为`a`，数量为`l`。

如果第`i`组玫瑰是蓝色的，那么对于`dp1`，要么用第一种魔法，操作数为`dp1[i-1]+l`，要么使用第二种魔法，操作数为`dp2[i-1]+1`。对于`dp2`，在它眼中这组玫瑰是红色的，我们无需操作。因此得到

```
if a=='B':
    dp1[i]=min(dp1[i-1]+l,dp2[i-1]+1)
    dp2[i]=dp2[i-1]
```

如果第`i`组玫瑰是红色，那么对于`dp1`，我们无需任何操作，`dp1[i]=dp1[i-1]`。对于`dp2`，在它眼中这一组玫瑰是蓝色的，因此我们有两种操作：要么用第一种魔法，操作数为`dp2[i-1]+l`，要么用第二种魔法，操作数为`dp1[i-1]+1`。因此得到

```
if a=='R':
    dp1[i]=dp1[i-1]
    dp2[i]=min(dp2[i-1]+l,dp1[i-1]+1)
```

注意，我们在上面的推导中无意识地运用了这一条件：反转列表的反转列表就是原列表。这正是矛盾观的体现：原列表与反转列表本是互相矛盾的对象，但通过魔法2的反转操作，原列表与反转列表可以相互转化。

最终代码：

```
roses=list(input())
stack=[[roses[0],1]]
for a in roses[1:]:
    if a==stack[-1][0]:
        stack[-1][1]+=1
    else:
        stack.append([a,1])
n=len(stack)
```

```
dp1=[0]*(n+1)
dp2=[0]*(n+1)
for i in range(1,n+1):
    a,l=stack[i-1]
    if a=='R':
        dp1[i]=dp1[i-1]
        dp2[i]=min(dp2[i-1]+l,dp1[i-1]+1)
    if a=='B':
        dp1[i]=min(dp1[i-1]+l,dp2[i-1]+1)
        dp2[i]=dp2[i-1]
print(dp1[-1])
```

当然，本题还有其他思路，比如我之前看到题解的做法，两个`dp`表一个表示全变红的操作数，一个表示全变蓝的操作数。

再看一道例题：

152. 乘积最大子数组 - 力扣（LeetCode）

给你一个整数数组`nums`，找出数组中乘积最大的非空连续子数组。

乍一看，本题与最大连续和的Kadane算法非常类似，因此你可能写出类似下面的代码：

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        n=len(nums)
        dp=[nums[0]]
        for i in range(1,n):
            dp.append(max(nums[i],dp[-1]*nums[i]))
        return max(dp)
```

但这是不正确的。例如对于下面的案例：

```
nums=[-2,2,3,-2]
```

最大乘积是将所有数相乘，结果为`24`，但Kadane算法得到的结果为`6`。这是因为乘积数组的第一个数是负数，Kadane算法基于局域最大值会将前面的乘积丢弃，但由于最后又有一个负数，因此前面的负数乘积会重新变为正数，并成为最大乘积。

通过乘以一个负数，原本的最大乘积会变为最小乘积，而原本的最小乘积会变为最大乘积。你看，这是不是又出现了矛盾双方的相互转化？因此我们确定这也是系统的一组参数，我们既要求最大乘积，也要求最小乘积。

设`dp[i]`为一个二元数组`[a,b]`，`a`表示以第`i`个元素结尾的连续序列中的乘积最大值，`b`表示以第`i`个元素结尾的连续序列中乘积的负数最大值。然后就可以根据`nums[i]`的正负性分类递推了。

- 如果`a`为正数，那么`dp[i][0]=a*dp[i-1][0]`，`dp[i][1]=a*dp[i-1][1]`
- 如果`a`为负数，那么`dp[i][0]=a*dp[i-1][1]`，`dp[i][1]=a*dp[i-1][0]`
- 如果`a==0`，那么`dp[i]=[0,0]`

需要注意的是，如果`nums[i]`之前的数都是正数，那么无论如何我们无法取到负数，不存在负数最大值。这可能对我们的讨论产生影响。我们可以合理运用`float('inf')`处理这类情况。

最终代码如下：

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        if len(nums)==1:
            return nums[0]
        n=len(nums)
        dp=[[-float('inf'),float('inf')]]
        for i in range(n):
            a=nums[i]
            dp1=[0,0]
            if a>0:
                dp1[0]=max(a,dp[-1][0]*a)
                dp1[1]=min(float('inf'),dp[-1][1]*a)
            elif a<0:
                dp1[0]=max(-float('inf'),a*dp[-1][1])
                dp1[1]=min(a,dp[-1][0]*a)
            else:
                dp1=[0,0]
            dp.append(dp1)
        return max([dp[i][0] for i in range(n+1)])
```

当你适应了这种矛盾辩证的思维方式后，就能很轻松地找到`dp`题的隐藏参数了。比如下面这题：

OpenJudge - 28970:预测赢家

这题是`dp`与博弈论的结合。给定一个数组`nums`，玩家1和玩家2轮流进行操作，从`nums`的两端中弹出一个数加到自己身上，最终得分高者获胜。现在玩家1先手，判断玩家1是否获胜。

首先，我们肯定不能直接判断是否获胜，而是计算玩家1能获得的最高分数，判断其与玩家2的分数的大小关系。

注意到游戏规则是对数组 `nums` 的两端进行操作，当我们试图拆分子问题时，自然想到将数组的首尾两端作为参数。设 `dp[(i,j)]` 表示对于列表 `nums[i:j+1]`，玩家1可以获得的最高分数。

从博弈的观点，当玩家面对列表 `nums[i:j+1]` 时，他既可以选择取出左端的数，这样他获得的分数就是 `nums[i]` 加上当他面对 `nums[i+1:j+1]` 且为后手时能获得的最高分数；也可以选择取出右端的数，这样获得的分数就是 `nums[j]` 加上他面对 `nums[i:j]` 且为后手时能获得的最高分数。

这时矛盾关系出现了。我们同时计算当玩家面对 `nums[i:j+1]`，且为先手、后手时能获得的最高分数。鉴于数组 `(i,j)` 离散度较高，我们可以选择用字典存储结果，即 `dp` 是一个字典，`dp[(i,j)]` 是一个二元数组 `[a,b]`， `a`， `b` 分别表示玩家面对 `nums[i:j+1]` 且为先手、后手时能获得的最高分数。

接下来建立递推关系。玩博弈的两个人都是绝顶聪明的。当玩家面对 `nums[i:j+1]` 且为先手时，他会权衡取左边和取右边能获得的最高分数，计算方式如前所述，然后选择其中较大的一方，那么当前为后手的另一个玩家就只能顺从该选择，并成为下一轮的先手。

```
l=[0,0]
a=nums[i]+dp[(i+1,j)][1] # 取左边的最高分数
b=nums[j]+dp[(i,j-1)][1] # 取右边的最高分数
if a>=b:
    l[0]=a
    l[1]=dp[(i+1,j)][0]
else:
    l[0]=b
    l[1]=dp[(i,j-1)][0]
dp[(i,j)]=l
```

注意到这个递推公式要求知道 `dp[(i+1,j)]` 和 `dp[(i,j-1)]`，这就规定了我们的递推顺序应该是以数组长度 `d=j-i` 递增的顺序。

最终代码：

```
for _ in range(int(input())):
    data=iter(map(int,input().split()))
    m=next(data)
    nums=[]
    for i in range(m):
        nums.append(next(data))
    dp={}
    for d in range(m):
        for i in range(m-d):
            if d==0:
                dp[(i,i)]=[nums[i],0]
                continue
```

```
            l=[0,0]
            a=nums[i]+dp[(i+1,i+d)][1]
            b=nums[i+d]+dp[(i,i+d-1)][1]
            if a>=b:
                l[0]=a
                l[1]=dp[(i+1,i+d)][0]
            else:
                l[0]=b
                l[1]=dp[(i,i+d-1)][0]
            dp[(i,i+d)]=l
    if dp[(0,m-1)][0]>=dp[(0,m-1)][1]:
        print('true')
    else:
        print('false')
```

深度优先搜索

> 注意！这是本笔记内容最丰富、也是最难的一章。

深度优先搜索和广度优先搜索，虽然都带有“搜索”两个字，但它们是完全不同的两种算法。深搜最鲜明的特征，就是使用`dfs`函数进行递归操作，要学习深搜，必须学习函数递归。而要学习函数递归，首先要对函数是什么、函数是如何工作的有清晰的认识。

初学者接触函数时，对“函数是什么”这个问题，大概率是这个回答：

函数就是对代码中重复出现的片段进行封装，以方便多次调用。

例如利用海伦公式计算三角形面积：

```
import math
def area(a,b,c):
    p=(a+b+c)/2
    return math.sqrt(p*(p-a)*(p-b)*(p-c))
n=int(input())
for _ in range(n):
    a,b,c=map(int,input().split())
    print(area(a,b,c))
```

这个理解有问题吗？完全没问题。但如果仅仅停留在这个理解，你是无法理解函数递归的。函数的这个定义无法解释函数递归的工作原理。

下面是笔者对“函数是什么”这个问题的回答：

每当代码遇到一个函数调用语句，它会立刻中止当前代码，转而去执行函数体中的代码，直到函数体代码运行完毕后，回到原代码继续执行。

例如函数递归的经典代码：斐波那契数列。

```
def f(n):
    if n==1:
        return 1
    else:
        return n*f(n-1)
print(f(5))
```

我们来模拟一下程序的工作流程。主程序中出现了一个函数调用语句`print(f(5))`，因此Python会立即中止执行`print()`操作，转而去执行`f(5)`中的代码。当执行`f(5)`中代码时，它又遇到了`return 5*f(4)`，再次遇到函数调用语句，于是再次中止代码，转而去执行`f(4)`。

这个过程一直循环，直到执行`f(1)`代码时，只有一句`return 1`，其中没有类似的函数调用语句，因此能够成功返回值`1`。这个返回值传递到之前中止了的`return 2*f(1)`语句中，并代替了其中的`f(1)`，因此返回值`2`。这个过程一直循环，直到最后`return 5*f(4)`，用`24`代替其中的`f(4)`，返回值`120`。于是`print(f(5))`中用`120`代替`f(5)`，输出`120`。

上面对函数的定义实际上是对Python函数工作流程的描述。只要你写递归代码时能清楚地知道函数的工作流程，就能知道你写的函数之后会干什么、怎么干，就能轻松的写出递归了。

我们知道，数学的函数有三要素：定义域、值域、映射关系。计算机的函数也有类似的三要素：输入值，输出值，函数体。

必须指出的是，计算机学中的函数与数学中的函数有很大的区别。其中一个很关键的区别，就是**Python函数不仅可以读取函数空间的值，还可以读取外部空间的值。**

狭义上来讲，Python函数的输入值就是那个圆括号中包含的值，比如前面的`f(n)`，`n`就是其输入值。但你可能已经注意到了，函数可以读取外部定义的数（不可变类型）：

```
n=5
def count():
    for i in range(n):
        print(i)
count()
```

函数可以读取外部的列表（可变类型）：

```
nums=[1,2,3,4,5]
def pow(n):
    return sum([x**n for x in nums])
print(pow(2))
```

加上一个`global`后，函数可以改变外部的数（不可变类型）：

```
s=0
def add(n):
    global s
    for _ in range(n):
        s+=1
add(5)
print(s)
co```

即使不加`global`，函数也可以随意改变外部的列表（可变类型）：
```

```python
condition=[False,False,False]
def check(n):
    condition[n]=True
check(0)
check(1)
check(2)
print(condition)
```

所以，不要再被数学上的函数定义所局限住了。现在对函数的输入值做一个新定义：

无论你在函数的圆括号里写什么，都不影响函数的输入值。函数的输入值是整个系统的所有值。

这里的“整个系统的所有值”，既包括圆括号内的、函数体中的值，也包括函数体外的值。既包括不可变参数类型，即整型、布尔型、元组等，也包括可变参数类型，如列表、字典等。

用同样的角度思考，我们也会得到**函数的输出值作用于整个系统的所有值。**

首先，函数自己可能返回一个值，就像前面的 `f(n)`，它返回一个数，代表 `n` 的阶乘。

除此之外，函数能改变外部的参数，比如用 `global` 改变外部整型的值，或者改变列表中的某一个值。我们可以把这种操作视为，将系统外部的值从某个状态转移到另一个状态。

由此得出一个归纳性的结论：

函数的输入值是整个系统的所有值，输出值是整个系统的所有值。函数是一个系统到另一个系统的映射，或者说是系统的一个状态到另一个状态的映射。

这样就能很好地解释初学深搜时可能出现的困惑。如用深搜生成全排列：

```python
n=int(input())
condition=[False]*(n+1)
ans=[]
def dfs(x):
    if x==n:
        print(*ans)
        return
    for i in range(1,n+1):
        if not condition[i]:
            condition[i]=True
            ans.append(i)
            dfs(x+1)
            condition[i]=False
            ans.pop()
dfs(0)
```

这里 `x` 表示当前全排列含有的数的个数。初学者可能无法理解这种写法，尤其是为什么 `dfs(x)` 函数只有一个参数。现在我们知道了，其实这个函数的输入值是整个系统状态，既包括 `x` ，也包括 `condition` ， `dfs` 必须解读为“当前全排列含有 `x` 个数，且各个数使用情况为 `condition` 时的全排列”。在 `dfs` 函数中对 `conditon` 的操作实际上就是更新了系统的状态。

在前面全排列的例子中，对于同一个 `x` ，由于外部状态 `condition` 的不同， `dfs` 函数的执行结果也不同。因此我们得到：

如果只考虑函数的标准输入，那么它不满足映射的定义，即对同一组输入，可能有多组输出与之对应。但如果考虑整个系统的状态作为函数输入，那么它就能满足映射的定义，将当前系统状态唯一地转变为另一状态。

一旦你知道了函数是从系统到系统的映射，那么系统的某一个参数，到底是在函数内部，还是在函数外部，就没那么重要了。接下来这一部分，要**充分发挥你的想象力。**

比如，如果我们把前面的全排列改成组合数，生成所有从 `n` 个数中取出 `m` 个数的组合。这时候为了去重，我们会要求组合中的各个数是单调递增的。

首先看标准写法， `x` 表示当前取出数的个数， `condition` 表示各个数的使用情况， `ans` 表示当前组合。将 `x` 作为 `dfs` 函数的参数，对 `condition` 和 `ans` 则采用回溯的写法。

```
n,m=map(int,input().split())
condition=[False]*(n+1)
ans=[]
def dfs(x):
    if x==m:
        print(*ans)
        return
    for j in range(1,n+1):
        if not condition[j] and (not ans or j>ans[-1]):
            condition[j]=True
            ans.append(j)
            dfs(x+1)
            condition[j]=False
            ans.pop()
dfs(0)
```

我们注意到生成组合时要求数列是单调递增的，既然这样，我们“无缘无故”地多加一个参数 `i` ，表示当前组合中的最大数。这样 `dfs` 函数就多了一个参数。

```
n,m=map(int,input().split())
condition=[False]*(n+1)
```

```
ans=[]
def dfs(x,i):
    if x==m:
        print(*ans)
        return
    for j in range(1,n+1):
        if not condition[j] and j>i:
            condition[j]=True
            ans.append(j)
            dfs(x+1,j)
            condition[j]=False
            ans.pop()
dfs(0,0)
```

接下来更疯狂一些，`condition`放在外面太不爽了，干脆把它也变成参数。

```
n,m=map(int,input().split())
ans=[]
def dfs(x,i,condition):
    if x==m:
        print(*ans)
        return
    for j in range(1,n+1):
        condition1=condition[:]
        if not condition1[j] and j>i:
            condition1[j]=True
            ans.append(j)
            dfs(x+1,j,condition1)
            ans.pop()
dfs(0,0,[False for _ in range(n+1)])
```

到这里，我们的所有输入值都被记录在`dfs()`的圆括号里了。

接下来考虑反向操作。标准形式中`dfs`以`x`为参数。如果我们把`x`扔到外面去会怎么样？

```
n,m=map(int,input().split())
condition=[False]*(n+1)
ans=[]
x=0
def dfs():
    global x
    if x==m:
        print(*ans)
        return
    for j in range(1,n+1):
```

```
        if not condition[j] and (not ans or j>ans[-1]):
            condition[j]=True
            ans.append(j)
            x+=1
            dfs()
            condition[j]=False
            ans.pop()
            x-=1
dfs()
```

我们甚至可以给 `x` 换一种存储形式，把它放到 `stack` 里。

```
n,m=map(int,input().split())
condition=[False]*(n+1)
ans=[]
stack=[0]
def dfs():
    if stack and stack[-1]==m:
        print(*ans)
        return
    for j in range(1,n+1):
        if not condition[j] and (not ans or j>ans[-1]):
            condition[j]=True
            ans.append(j)
            stack.append(stack[-1]+1)
            dfs()
            condition[j]=False
            ans.pop()
            stack.pop()
dfs()
```

这下我们的所有输入值都被放在 `dfs` 外面了，这和前面完全是两个极端。

再仔细观察一下，上面的代码对 `ans` 和 `stack` 的回溯操作都是标准的先 `append` 再 `pop`，唯独对 `condition` 不是，强迫症不爽。

```
n,m=map(int,input().split())
condition=[[False]*(n+1)]
ans=[]
stack=[0]
def dfs():
    if stack and stack[-1]==m:
        print(*ans)
        return
    for j in range(1,n+1):
```

```
        if not condition[-1][j] and (not ans or j>ans[-1]):
            condition1=condition[-1][:]
            condition1[j]=True
            condition.append(condition1)
            ans.append(j)
            stack.append(stack[-1]+1)
            dfs()
            condition.pop()
            ans.pop()
            stack.pop()
dfs()
```

既然这时我们对 `x`， `condition` 和 `ans` 的操作都是类似于栈的回溯操作，我们能不能把它们放在一个栈中呢?

```
n,m=map(int,input().split())
mainstack=[[[False for _ in range(n+1)],[],0]]
def dfs():
    condition,ans,x=mainstack[-1]
    if x==m:
        print(*ans)
        return
    for j in range(1,n+1):
        condition1=condition[:]
        ans1=ans[:]
        if not condition1[j] and (not ans1 or j>ans1[-1]):
            condition1[j]=True
            ans1.append(j)
            mainstack.append([condition1,ans1,x+1])
            dfs()
            mainstack.pop()
dfs()
```

这时你发现， `dfs` 函数所需要的全部输入都在 `mainstack` 中了。没错，我们刚刚还原了**函数调用栈**。前面说过，函数以整个系统作为输入，因而我们可以理解为，函数调用栈存储了函数调用瞬间的所有系统状态，这与 `mainstack` 的功能相同。

最后说明一点：我们刚刚进行的都是理论性质的探究，目的是开拓视野。所以实战的时候，你千万不要这么写，只要按最方便的写法就行。具体来说，对于可变参数类型，如果传递方式特别简单，可以放在函数体中传递。对于不可变类型，将其像前面一样复制一份既费时又费力，因此一般就用回溯的方式处理。

到目前为止，我们都是对函数的输入做改动，还没有对函数的输出做改动。接下来我们看看从函数的输出角度可以做什么。

根据经验，我们知道一个函数可以做两件事：

第一，函数本身可能返回一个值。这个值可能是不可变类型，如整形或布尔型，也可能是可变类型，如列表或字典。

第二，函数可以不返回任何值，只改变系统状态。

函数的输出类型不同，我们编写 `dfs` 代码的方式也不同，或者说代码的风格不同。下面针对各种输出类型进行介绍。

- 函数只改变系统状态，不返回任何值

这是最常见的 `dfs` 编写方式。前一小节中的 `dfs` 函数都是这种类型。它最鲜明的特点就是代码中单独出现的一行 `dfs(···)`。

下面的伪代码展示了这类深搜代码的整体思路：

```
S=0 # 泛指各类不可变类型
condition=[False for _ in range(n)] # 泛指各类可变类型
def dfs('参数'):
    if '基线条件':
        global S
        S+=1 # 泛指对不可变类型的操作
        return
    if '剪枝条件':
        return
    for i in '备选列表':
        if i '满足条件':
            condition[i]=True # 泛指对可变类型的操作
            dfs('更新后的参数')
            condition[i]=False # 泛指回溯操作
dfs('初始条件')
print(S)
```

所谓深搜，无非就是从迷宫中找到一条满足条件的路径，或者说从一组数中找到一组满足条件的数。我们采取的方式是，逐个逐个地找出符合条件的数，直到集齐所有的数。寻找过程中的某个状态，就是函数的一次递归状态。

每次寻找一个新的数时，由于条件满足与否可能与之前已选的数有关，因此它的输入就是整个系统的当前状态。当我们把这个数加入已选列表后，它又会改变系统状态，影响后续的数的选择。

在某一个状态下，满足条件的数可能有多个，就像迷宫有多条路径。因此我们首先选择其中某个数，并相应地改变系统状态，看看该状态下后续能否满足条件。在判断完之后，我们应当将系统状态还原到判断前的状态，以供其他数的判断。这就是所谓的“回溯”。

我们来看几道例题：

OpenJudge - M18155:组合乘积

给定整数集合 `S` 和一个目标数 `T`，判断是否可以从 `S` 中挑选一个非空子集，子集中的数相乘的乘积为 `T`。

假设某个时刻我们选择了一些数，为了达到乘积 `T`，剩余的乘积为 `t`。对于列表中的数 `x`，它应当满足的条件是：首先，`x` 还未被使用过；其次，`x` 为 `t` 的因数。当我们选择了一个 `x` 之后，系统的状态就改变了，剩余的乘积变为 `t//x`，且 `x` 被标记为使用过。假设当前有多个 `x` 满足条件，当我们判断完其中一个 `x` 以后，为了判断其他值，要把这个 `x` 重新标记为未使用，即进行回溯。目标状态为 `t==1`，此时将目标值 `flag` 记为 `True`。

```
T=int(input())
S=list(map(int,input().split()))
n=len(S)
condition=[False]*n
flag=False
def dfs(t):
    if t==1:
        global flag
        flag=True
        return
    if flag:
        return
    for i in range(n):
        x=S[i]
        if t%x==0 and not condition[i]:
            condition[i]=True
            dfs(t//x)
            condition[i]=False
dfs(T)
print('YES') if flag else print('NO')
```

OpenJudge - T02754:八皇后

我们将某一时刻的摆放方式存于 `stack` 中，`stack[j]` 表示棋盘第 `j` 行放置皇后的列的位置。对于下一行 `i` 可能的摆放位置 `x`，它应满足与前面的皇后不冲突，既不位于同一列，即 `stack[j]!=x`，也不位于同一对角线上，即 `abs(stack[j]-x)!=abs(j-i)`。

```
res=[0]
stack=[0]
def dfs(i):
    if i==9:
        res.append(''.join(map(str,stack[1:])))
        return
    for x in range(1,9):
        if all(stack[j]!=x and abs(stack[j]-x)!=abs(j-i) for j in range(1,i)):
            stack.append(x)
            dfs(i+1)
            stack.pop()
dfs(1)
for _ in range(int(input())):
    print(res[int(input())])
```

只要理解深搜的工作原理，这类代码在整体架构方面是没有障碍的。对于较难的 `dfs` 题，其关键都在于剪枝上，而剪枝方法要因题目而异。

OpenJudge - M28906:数的划分

将整数 `N` 分成 `K` 份，且每份不能为空，任意两个方案不相同（不考虑顺序）。求不同的分法数。

对于寻找过程中的某个状态，我们有一个列表，其中的数的和为 `n`，数的个数为 `k`。由于不同顺序的分法被视为一种，为了去重我们要求这个数列是单调递增的，因此需要记录数列末尾的数 `a`。

在选择下一个数 `i` 的时候，我们不能每次都遍历从 `a` 到 `N-n` 的所有数，否则会超时。注意到 `a` 以后的所有数都应当大于等于 `a`，因此我们存在一个剪枝：如果 `n+(K-k)*a>N`，即使后面的所有数都取最小值 `a`，总和也会大于 `N`，无法满足条件，此时应当直接 `return`。对 `i` 的遍历范围也是从 `a` 到 `int((N-n)/(K-k))`。

```
N,K=map(int,input().split())
s=0
def dfs(n,k,a):
    if k==K:
        if n==N:
            global s
            s+=1
        return
    if n+(K-k)*a>N:
        return
    for i in range(a,int((N-n)/(K-k))+1):
        dfs(n+i,k+1,i)
```

```
dfs(0,0,1)
print(s)
```

借鉴上一节的思想，把 `a` 放到外部空间也是一个不错的选择。

```
N,K=map(int,input().split())
stack=[1]
s=0
def dfs(n,k):
    if k==K:
        if n==N:
            global s
            s+=1
        return
    if n+(K-k)*stack[-1]>N:
        return
    for i in range(stack[-1],int((N-n)/(K-k))+1):
        stack.append(i)
        dfs(n+i,k+1)
        stack.pop()
dfs(0,0)
print(s)
```

- 函数输出布尔值

这类代码适合那种判断已知数据是否满足条件的题目，即输出 `Yes` 或 `No` 的题目。

下面的伪代码展示了这类 `dfs` 代码的编写框架：

```
S=0 # 泛指各类不可变类型
condition=[False for _ in range(n)] # 泛指各类可变类型
def dfs('参数'):
    if '基线条件':
        return True
    if '剪枝条件':
        return False
    for i in '备选列表':
        if i '满足条件':
            condition[i]=True # 泛指对可变类型的操作
            if dfs('更新后的参数'):
                return True
            condition[i]=False # 泛指回溯操作
    return False
```

```
if dfs('初始条件'):
    print(S)
```

简单来讲，`dfs`函数返回系统在当前状态下，最终能否满足条件。对于每一个备选数`i`，我们将其加入列表中并相应改变系统状态，判断此时的`dfs`能否成功。如果成功则返回`True`，如果不能成功就进行回溯。如果所有选项都不成功则返回`False`。

虽然这类写法并不常见，但对某些题目有奇效，读者可以找几道题目练习一下这种写法。

比如前面的题目OpenJudge - M18155:组合乘积，函数返回布尔值的写法如下：

```
T=int(input())
S=list(map(int,input().split()))
n=len(S)
condition=[False]*n
def dfs(t):
    if t==1:
        return True
    for i in range(n):
        x=S[i]
        if t%x==0 and not condition[i]:
            condition[i]=True
            if dfs(t//x):
                return True
            condition[i]=False
    else:
        return False
print('YES') if dfs(T) else print('NO')
```

又如OpenJudge - T01011:Sticks，也使用了这种写法。这题是剪枝大题，但这里并不准备讲解它的剪枝策略，读者只要从下面的代码中学习`dfs`函数输出布尔值的写法即可。详细解释可以看题解。2020fall_cs101.openjudge.cn_problems.md

```
import bisect
while True:
    n=int(input())
    if n==0:
        break
    sticks=list(map(int,input().split()))
    sticks.sort()
    L=sum(sticks)
    M=sticks[-1]
    for k in range(L//M,1,-1):
        if L%k==0:
```

```
            d=L//k # 剪枝1
            condition=[False]*n
            stack=[]
            def dfs(m,x,d):
                if m==n:
                    if x==0:
                        return True
                    else:
                        return False
                if x==0:
                    # 剪枝2
                    for i in range(n-1,-1,-1):
                        if condition[i]==False:
                            condition[i]=True
                            stack.append(sticks[i])
                            if dfs(m+1,sticks[i]%d,d):
                                return True
                            condition[i]=False
                            stack.pop()
                            return False
                a=bisect.bisect(sticks,min(d-x,stack[-1])) # 剪枝3
                for i in range(a-1,-1,-1):
                    if not condition[i] and sticks[i]<=min(d-x,stack[-1]):
                        if i<a-1 and not condition[i+1] and
sticks[i+1]==sticks[i]:
                            continue # 剪枝4
                        condition[i]=True
                        stack.append(sticks[i])
                        if dfs(m+1,(x+sticks[i])%d,d):
                            return True
                        condition[i]=False
                        stack.pop()
                        if sticks[i]==d-x: # 剪枝5
                            return False
                return False
            if dfs(0,0,d):
                print(d)
                break
    else:
        print(L)
```

- 函数输出目标值

这是非常重要的一类写法。我们接下来要讲的是 `dfs` 与 `dp` 的结合。在 `dfs` 领域，它叫做**记忆化搜索**。在 `dp` 领域，它叫做**递归式** `dp`。

首先从 `dfs` 的角度思考。我们知道，`dfs` 函数的输入值是整个系统的状态，输出要求的目标值。如果 `dfs` 函数的输入可以局限在若干个数中，即用几个数表示整个系统的状态，那么我们可以把这几个数都放到 `dfs()` 的圆括号里，这时候的 `dfs` 函数就变成了**纯函数**。

所谓纯函数就是真正意义上的、数学上的函数。它满足映射的定义，对于输入 `dfs` 的圆括号中的一组值，有唯一的输出值与之对应，不会因外面的某个 `condition` 而改变。比如前面的斐波那契数列，`f(n)` 的值只与 `n` 有关，这就是一种纯函数。

我们知道，在利用递归求 `f(n)` 时，我们会依次求 `f(n-1)`，`f(n-2)` 直到 `f(1)`，要使用 `n` 次递归。如果我们要求很多次 `f(n)`，那么每次都会进行 `n` 次递归，非常浪费时间。注意到 `f(n)` 的值只与 `n` 有关，我们能不能将 `f(n)` 的值存起来呢？

我们可以建一个数据结构，它可以是列表或字典，把它记为 `dp`。每次我们使用 `f(n)` 时，首先看看 `dp` 中是否存储了 `f(n)` 的值。如果存了，万事大吉，直接返回该值。如果没存，那就使用递归，计算 `n*f(n-1)`。如此递归下去，可能某次递归到了 `f(m)`，而 `f(m)` 已经存储在 `dp` 中了，那么 `f(m)` 后面的递归都可以省略了，直接返回 `f(m)` 的值。

由此可见，通过“记忆化搜索”，每当搜索到的位置之前已经算过了，就不用继续递归，直接返回过去的计算结果，这就起到了节省时间的作用。

接下来从 `dp` 的角度思考。“动态规划”章节中提到，一次动态规划有三个关键要素：参数，递推公式，递推顺序，三者缺一不可。其中对于递推顺序，我们要求在求当前问题的值时，与该问题相关的子问题均已求出。

在有些问题中，我们知道如何计算子问题可以保证递推顺序的要求，如最长上升子序列，可以按序列从左到右的顺序递推。但也有很大一部分题目，我们无法提前知道递推顺序，这时候该怎么办呢？

递归式 `dp` 采取的策略是，首先“假装”你自己知道子问题的结果，并用它来计算当前问题。但在计算之前会检查各个子问题是否已经解决，如果你发现某个子问题还没解决，就中止当前求解流程，优先解决子问题。待所有子问题均已解决，再完成当前问题。

注意到了吗？上面所述的流程与函数的工作流程几乎完全相同。因此，我们就想到了用 `dfs` 函数辅助 `dp` 问题的求解，以此解决递推顺序问题。

打一个比喻：`dp` 就像是一个精明的人办事情，他能按照顺序有条不紊地完成所有任务。而 `dfs` 就像一个相对马虎的人办事情，当他要开组会的时候，突然发现“坏了，我报告PPT还没做！”，于是赶紧去做PPT，而做PPT时突然又发现“坏了，我实验数据不够！”，又赶紧去做实验，做完实验后再回来做PPT，再去开组会。这样虽然慢，但总是能完成的。这就是递归式 `dp` 的核心思想。

具体到代码编写上，记忆化搜索又有手动实现和自动实现两种。

首先是手动实现。我们来看看伪代码：

```
dp={}
def dfs(x): # x泛指各类参数
    if x in dp: # 之前求解过了
        return dp[x]
    a=f(dfs(y) for y in '与当前问题有关的子问题') # f表示递推关系
    dp[x]=a
    return a
```

这段代码虽然短，但基本涵盖了记忆化搜索的关键结构。

数据结构 `dp` 存储了 `dfs` 函数的执行结果。每当调用 `dfs(x)` 时，首先在 `dp` 表中检查是否求解过，如果求解过则直接返回值。

如果没求解过，就使用递推公式计算 `dfs(x)` 的值。在传统的动态规划中，递推公式中子问题的值是 `dp[y]`，但现在我们都用 `dfs(y)` 代替，这意味着我们计算前会先判断 `dfs(y)` 是否已求解，并优先解决尚未解决的子问题。

我们特别关注一下最后两行。它同时令 `dp[x]=a` 和 `dfs(x)=a`，也就是说 `dp` 和 `dfs` 存储的内容是完全相同的！在第一章数据结构中我们说过，数据结构永远不要嫌多，在这里我们可以把 `dp` 和 `dfs` 看成是两种数据结构，它们存储了完全相同的信息。记忆化搜索其实是利用了这两种数据结构的不同属性。`dp` 结构具有稳定性和实时访问性，且一旦输入就不再更改；而 `dfs` 函数是动态的，它不具有存储功能，但函数递归为 `dfs` 提供了极大的灵活性。

接下来我们看看自动实现方式。它的伪代码甚至更短。

```
from functools import lru_cache
@lru_cache(maxsize=None)
def dfs(x): # x泛指各类参数
    return f(dfs(y) for y in '与当前问题有关的子问题')
```

使用 `lru_cache` 的效果与手动建 `dp` 表的效果完全相同。你可以理解为，在 `dfs` 函数前加上一个 `lru_cache` 装饰器后，系统会自动帮你记录 `dfs(x)` 的值，并存储在后台，当某个 `dfs(x)` 之前算过了，就可以直接调用结果。

我们来看例题：

OpenJudge - M01088:滑雪

对于一个确定的区域地图，当滑雪的起始位置 `(x,y)` 确定后，我们能够滑雪的最大距离也确定了，因此该题满足纯函数的条件。递推条件是，对于 `(x,y)` 领域内的四个点，如果某点 `(x1,y1)` 的高度小于 `(x,y)` 点的高度，那么就能选择向 `(x1,y1)` 方向滑，从而 `dfs(x,y)=1+dfs(x1,y1)`。我们要选四个点中滑雪距离最长的那一个，这就是代码中的 `dp[x][y]=max(dp[x][y],dfs(x1,y1)+1)`。

这里用列表存储 `dfs(x,y)` 的值，并用 `condition` 表示某点是否求解过。你也可以通过判断 `dp[x][y]` 是否等于 `1` 来判断是否求解过，但我们说数据结构不要嫌多，加一个 `condition` 更爽一些。

```
R,C=map(int,input().split())
grid=[]
for _ in range(R):
    grid.append(list(map(int,input().split())))
def check(x,y):
    return 0<=x<=R-1 and 0<=y<=C-1
dp=[[1]*C for _ in range(R)]
condition=[[False]*C for _ in range(R)]
def dfs(x,y):
    if condition[x][y]:
        return dp[x][y]
    d=[[1,0],[-1,0],[0,1],[0,-1]]
    for i in range(4):
        x1=x+d[i][0]
        y1=y+d[i][1]
        if check(x1,y1) and grid[x1][y1]<grid[x][y]:
            dp[x][y]=max(dp[x][y],dfs(x1,y1)+1)
    condition[x][y]=True
    return dp[x][y]
s=1
for x in range(R):
    for y in range(C):
        s=max(s,dfs(x,y))
print(s)
```

`lru_cache` 写法：

```
from functools import lru_cache
R,C=map(int,input().split())
grid=[]
for _ in range(R):
    grid.append(list(map(int,input().split())))
def check(x,y):
    return 0<=x<=R-1 and 0<=y<=C-1
@lru_cache(maxsize=None)
def dfs(x,y):
    d=[[1,0],[-1,0],[0,1],[0,-1]]
    a=1
    for i in range(4):
        x1=x+d[i][0]
        y1=y+d[i][1]
```

```
            if check(x1,y1) and grid[x1][y1]<grid[x][y]:
                a=max(a,dfs(x1,y1)+1)
        return a
    s=1
    for x in range(R):
        for y in range(C):
            s=max(s,dfs(x,y))
    print(s)
```

OpenJudge - T01661:帮助 Jimmy

对于空间中的某一点 `(x,y)`，从该点到达地面的最短时间是确定的，因此本题也满足纯函数的条件。我们找到点 `(x,y)` 下方的那一个平台，它的横坐标范围是 `(x1,x2)`，纵坐标是 `h`。Jimmy落到那个平台上后，要么向左走，时间是 `x-x1+dfs(x1,h)`，要么向右走，时间是 `x2-x+dfs(x2,h)`。如果落到地面，直接返回 `0`。

由于空间坐标 `(x,y)` 的范围较大，我们用字典来存储结果。

```
for _ in range(int(input())):
    N,X,Y,MAX=map(int,input().split())
    stage=[]
    for _ in range(N):
        stage.append(list(map(int,input().split())))
    stage.sort(key=lambda x:-x[2]) # 按高度降序排序
    stage.append([-1<<15,1<<15,0])
    dp={}
    def dfs(x,y):
        if (x,y) in dp:
            return dp[(x,y)]
        for i in range(N+1):
            x1,x2,h=stage[i]
            if y-MAX<=h<y and x1<=x<=x2:
                if h==0: # 到达地面
                    dp[(x,y)]=0
                    return 0
                dp[(x,y)]=min(x-x1+dfs(x1,h),x2-x+dfs(x2,h))
                return dp[(x,y)]
        else: # 下方无平台
            return float('inf')
    print(Y+dfs(X,Y))
```

`lru_cache` 写法：

```
from functools import lru_cache
for _ in range(int(input())):
```

```
    N,X,Y,MAX=map(int,input().split())
    stage=[]
    for __ in range(N):
        stage.append(list(map(int,input().split())))
    stage.sort(key=lambda x:-x[2]) # 按高度降序排序
    stage.append([-1<<15,1<<15,0]) # 代表地面
    @lru_cache(maxsize=None)
    def dfs(x,y):
        for i in range(N+1):
            x1,x2,h=stage[i]
            if y-MAX<=h<y and x1<=x<=x2:
                if h==0:
                    return 0
                return min(x-x1+dfs(x1,h),x2-x+dfs(x2,h))
        else:
            return float('inf')
    print(Y+dfs(X,Y))
```

理论上来讲，记忆化搜索完全克服了 `dp` 问题对递推顺序的限制，因此所有 `dp` 问题都可以用记忆化搜索解决。只不过时间复杂度可能会上升。

OpenJudge - T09267:核电站

设 `dfs(n,m)` 表示一共有 `n` 个坑，且最后有 `m` 个连续的坑放有核物质时的方案数。如果 `m` 不为零，那么就是在前面 `m-1` 个坑的基础上加上一个核物质，`dfs(n,m)=dfs(n-1,m-1)`；如果 `m` 为零，就是在所有 `n-1` 个坑的方案后加上一个空的坑，故 `dfs(n,0)=sum([dfs(n-1,i) for i in range(M)])`。

```
N,M=map(int,input().split())
dp={}
def dfs(n,m):
    if (n,m) in dp:
        return dp[(n,m)]
    a=0
    if m>n:
        a=0
    elif n==1:
        a=1
    elif m==0:
        a=sum([dfs(n-1,i) for i in range(M)])
    else:
        a=dfs(n-1,m-1)
    dp[(n,m)]=a
    return a
print(sum([dfs(N,m) for m in range(M)]))
```

`lru_cache`写法

```
from functools import lru_cache
N,M=map(int,input().split())
@lru_cache(maxsize=None)
def dfs(n,m):
    if m>n:
        return 0
    elif n==1:
        return 1
    elif m==0:
        return sum([dfs(n-1,i) for i in range(M)])
    else:
        return dfs(n-1,m-1)
print(sum(dfs(N,m) for m in range(M)))
```

由于不用考虑递推顺序，本题中的记忆化搜索代码显得非常简洁优雅。

最后再次强调，记忆化搜索的使用条件是`dfs`函数为纯函数，与系统的外部参数无关。如果外面有一个什么`condition`，就不能用记忆化搜索了。

广度优先搜索

虽然深搜和广搜都是搜索代码，但我们对不同题目采取的搜索策略并不相同。比如，如果问在图中从起点到终点至少需要多少步，我们肯定选择广搜。但像前面的八皇后，我们就肯定不会选择广搜了。为什么有这种差别？

从工作原理来讲，深搜以搜索的深度为核心，它自带记录搜索路径的功能，单次搜索的各个节点具有强关联性，因此可以讨论状态之间的互相影响。就像前面的八皇后，我们利用前面的皇后位置判断当前合法的皇后位置。而广搜以搜索的宽度为核心，将距离相同的若干点同步加入队列中进行检查，这些点之间关联性一般不强。因此我们看到，**广搜能解决的问题，待搜索的图一般是固定的**，不像深搜那样灵活多变。

广搜天然适合最短距离问题。广搜以从起点到地图某点的距离为核心参数，既然距离随搜索顺序是单调递增的，那么我们第一次到地图的某个位置时，所走的距离一定就是到达该点的最短距离。因此我们以后就不需要再次走过相同的点了，我们要对走过的点进行标记，以提醒自己不要再次搜索该点。

由此我们知道，广搜的基本流程就是：找出当前距离最短的点，把它的邻居加入队列，对走过的点进行标记。

广搜代码的核心成分，我们可以简单记为：`grid`，`queue`，`neighbour`，`set`。

`grid`表示待搜索的图，`queue`表示搜索队列，`neighbour`函数生成某点的邻居，`set`对走过的点进行标记。万变不离其宗，所有广搜代码本质上都是这四个成分。

我们来看例题：

OpenJudge - M19930:寻宝

一道经典的广搜入门题。

用`queue`表示搜索队列，初始时它只有起始点。每次我们从队列左侧弹出一个元素，把它的满足条件的邻居加入队列并进行标记，由此扩大搜索范围。

对于图中的某一点`(x,y)`，它的邻居`(x1,y1)`就是周围四个点，此外要满足`(x1,y1)`位于图中、不是陷阱、未搜索过。这就是代码倒数第五行最“长”的那个判断条件。

```
from collections import deque
m,n=map(int,input().split())
grid=[]
for _ in range(m):
    grid.append(list(map(int,input().split())))
def check(x,y):
    return 0<=x<=m-1 and 0<=y<=n-1
```

```
queue=deque([(0,0,0)])
set1=set([(0,0)])
while queue:
    x,y,t=queue.popleft()
    if grid[x][y]==1:
        print(t)
        break
    d=[(1,0),(-1,0),(0,1),(0,-1)]
    for i in range(4):
        x1=x+d[i][0]
        y1=y+d[i][1]
        if check(x1,y1) and grid[x1][y1] in [0,1] and (x1,y1) not in set1:
            queue.append((x1,y1,t+1))
            set1.add((x1,y1))
else:
    print('NO')
```

从本题我们立刻看出：`bfs`代码与`dp`代码和`dfs`代码都完全不同，往往又臭又长，非常容易在细节上出错。因此一定要多练。

OpenJudge - M25572:螃蟹采蘑菇

广搜加上了体积`1*2`的限制，题目变得复杂了，但本质上没有区别。

我们采取的方式是同时记录螃蟹占据的两个点的坐标`(x1,y1,x2,y2)`。同样是生成满足条件的邻居，同样加入队列，同样进行标记。坐标点的邻居应该满足螃蟹占据的两个体积都不穿过墙。

```
from collections import deque
n=int(input())
grid=[]
position=[]
for i in range(n):
    row=list(map(int,input().split()))
    grid.append(row)
    for j in range(n):
        if row[j]==5:
            position.append(i)
            position.append(j)
position=tuple(position)
def check(x,y):
    return 0<=x<=n-1 and 0<=y<=n-1
def neighbour(x1,y1,x2,y2):
    l=[]
    d=[(1,0),(-1,0),(0,1),(0,-1)]
    for dx,dy in d:
        if check(x1+dx,y1+dy) and check(x2+dx,y2+dy):
```

```
            if grid[x1+dx][y1+dy] in [0,5,9] and grid[x2+dx][y2+dy] in
[0,5,9]:
                l.append((x1+dx,y1+dy,x2+dx,y2+dy))
    return l
queue=deque([position])
set1=set([position])
while queue:
    x1,y1,x2,y2=queue.popleft()
    if grid[x1][y1]==9 or grid[x2][y2]==9:
        print('yes')
        break
    for t in neighbour(x1,y1,x2,y2):
        if t not in set1:
            queue.append(t)
            set1.add(t)
else:
    print('no')
```

必须说明的是，对搜索过的点进行标记，这是非常必要的步骤。对于规模较大的广搜题，如果不进行标记，代码的时、空复杂度将显著上升。比如下面这道题：

OpenJudge - T27237:体育游戏跳房子

这题是在数轴上的广搜。队列 `queue` 记录当前的操作过程，用 `calculate(s)` 函数计算对应的点坐标。很明显，到达数轴上某一点的方式有很多种，如果不进行标记，空间复杂度将达到 `2**k` 量级，`k` 为跳跃次数。笔者刚开始就因为没标记而反复MLE。除此之外，本题并没有更多难点，与常规 `bfs` 没有太大区别。

```
from collections import deque
while True:
    n,m=map(int,input().split())
    if (n,m)==(0,0):
        break
    def calculate(s):
        x=n
        for a in s:
            if a=='H':
                x*=3
            if a=='O':
                x//=2
        return x
    queue=deque([''])
    set1=set([n])
    while queue:
        s=queue.popleft()
```

```
        x=calculate(s)
        if x==m:
            print(len(s))
            print(s)
            break
        if x*3 not in set1:
            queue.append(s+'H')
            set1.add(x*3)
        if x//2 not in set1:
            queue.append(s+'O')
            set1.add(x//2)
```

要注意的是，在前面的深搜部分我们了解到，系统的状态是真正重要的东西。既然广搜的图是固定的，所谓“系统状态”其实就是搜索点的坐标状态。搜索的时候同样要关注搜索点的所有状态，只有所有状态都相同的点才可以视为同一点。

OpenJudge - T04129:变换的迷宫

本题在传统空间维度上添加了一个时间维度，当时间为 `K` 的倍数时可以通过石头。系统的“所有状态”既包括空间坐标，也包括时间坐标，因此是三维的，只有空间坐标和时间坐标均相同的点才能视为同一个点。但由于决定状态的是时间对 `K` 的余数，因此时间维的状态数就是 `K` 个。这里用列表 `condition` 代替 `set` 进行标记。

```
from collections import deque
for _ in range(int(input())):
    R,C,K=map(int,input().split())
    grid=[]
    for i in range(R):
        row=list(input())
        grid.append(row)
        for j in range(C):
            if row[j]=='S':
                x0,y0=i,j
    def check(x,y):
        return 0<=x<=R-1 and 0<=y<=C-1
    queue=deque([(x0,y0,0)])
    condition=[[[False]*K for _ in range(C)] for _ in range(R)]
    condition[x0][y0][0]=True
    while queue:
        x,y,t=queue.popleft()
        if grid[x][y]=='E':
            print(t)
            break
        d=[(1,0),(-1,0),(0,1),(0,-1)]
        for i in range(4):
```

```
            x1=x+d[i][0]
            y1=y+d[i][1]
            if check(x1,y1):
                if grid[x1][y1] in ['.','S','E']:
                    if not condition[x1][y1][(t+1)%K]:
                        queue.append((x1,y1,t+1))
                        condition[x1][y1][(t+1)%K]=True
                if grid[x1][y1]=='#' and (t+1)%K==0:
                    if not condition[x1][y1][(t+1)%K]:
                        queue.append((x1,y1,t+1))
                        condition[x1][y1][(t+1)%K]=True
    else:
        print('Oop!')
```

OpenJudge - 29954:逃离紫罗兰监狱

本题在空间维度上添加了一个能量维度，穿过一次石头消耗一次能量。系统状态同时包含两个空间坐标和一个能量坐标，只有三者都相同的点才视为同一个状态点。除此之外，就与前面的题没什么区别了。

```
from collections import deque
R,C,K=map(int,input().split())
grid=[]
for i in range(R):
    row=list(input())
    grid.append(row)
    for j in range(C):
        if row[j]=='S':
            x0,y0=i,j
def check(x,y):
    return 0<=x<=R-1 and 0<=y<=C-1
queue=deque([(x0,y0,0,0)])
condition=[[[False]*(K+1) for _ in range(C)] for _ in range(R)]
condition[x0][y0][0]=True
while queue:
    x,y,t,k=queue.popleft()
    if grid[x][y]=='E':
        print(t)
        break
    d=[(1,0),(-1,0),(0,1),(0,-1)]
    for i in range(4):
        x1=x+d[i][0]
        y1=y+d[i][1]
        if check(x1,y1):
            if grid[x1][y1] in ['.','S','E']:
```

```
                if not condition[x1][y1][k]:
                    queue.append((x1,y1,t+1,k))
                    condition[x1][y1][k]=True
            if grid[x1][y1]=='#':
                if k<K and not condition[x1][y1][k+1]:
                    queue.append((x1,y1,t+1,k+1))
                    condition[x1][y1][k+1]=True
else:
    print(-1)
```

接下来我们介绍 bfs 算法的几个变种。

- 双向 bfs

我们知道，bfs 算法的时空复杂度与起始点和目标点的距离有关，设待求步数为 k，每一步有 n 种选择，那么时空复杂度为n^k。当 k 值较大时，时空复杂度为指数量级，很容易超内存。

这时我们可以选择双向 bfs，同时从起点和终点开始搜索，记录从起点能到达的点和从终点能到达的点。如果两者有某一点重合，那么我们就找到了一条路径，把从起点到该点的路径与从终点到该点的路径拼接起来就是最终路径。

在双向 bfs 下，为了找到目标值，从起点和终点所走的步数均为 k/2，因此时空复杂度变为$2 \cdot n^{k/2}$，相比于单向 bfs 显著降低了。对于搜索规模较大的题，可以尝试使用双向 bfs。

一个常见的问题是：我们如何选择当前是扩展起点队列还是扩展终点队列呢？

我们有以下几种策略：

- 平衡队列策略：判断当前起点队列与终点队列的大小，选择其中较小的进行扩展
- 交替扩展策略：一次扩展起点队列，一次扩展终点队列
- 层次交替策略：让起点队列和终点队列交替扩展一层，从而队列末端到起点、终点的距离相同

实际应用中可根据题目特点选择合适的扩展策略。

我们来看例题：

OpenJudge - 30022:坐地铁

我们要找到图中与起点和终点距离相同的点，并选择满足条件的所有点中距离最短的点。这道题很适合使用双向 bfs，并且由于要求搜索点到两端的距离相同，因此可以选择层次交替策略。具体表现在代码中，就是 `for _ in range(len(queue))`，它会一次性检查所有与起点距离相同的点，并进行更新。

```
from collections import deque
n,k,s=map(int,input().split())
grid=[]
for _ in range(n):
    grid.append(list(map(int,input().split())))
if k==s:
    print(0)
    exit()
queue1=deque([(k,0)])
dict1={k:0}
queue2=deque([(s,0)])
dict2={s:0}
while queue1 or queue2:
    for _ in range(len(queue1)):
        k1,t1=queue1.popleft()
        for k3 in range(n):
            if grid[k1][k3]==1 and k3 not in dict1:
                queue1.append((k3,t1+1))
                dict1[k3]=t1+1
                if k3 in dict2 and dict2[k3]==dict1[k3]:
                    print(dict1[k3])
                    exit()
    for _ in range(len(queue2)):
        k2,t2=queue2.popleft()
        for k3 in range(n):
            if grid[k2][k3]==1 and k3 not in dict2:
                queue2.append((k3,t2+1))
                dict2[k3]=t2+1
                if k3 in dict1 and dict1[k3]==dict2[k3]:
                    print(dict2[k3])
                    exit()
else:
    print(-1)
```

OpenJudge - 30283:骑士大游历

骑士在 `1000*1000` 的棋盘上走日字步，判断任意两点之间的最短步数。这题也是经典的双向 `bfs`，我们同时从起点和终点扩展，用 `dict1` 和 `dict2` 分别记录某点到起点、到终点的步数。由于搜索规模很大，我们采取平衡队列策略，表现在代码中就是 `if len(queue1)<len(queue2)`。这样可以最大限度地减少内存空间。

```
from collections import deque
def solve(x1,y1,x2,y2):
    if (x1,y1)==(x2,y2):
        return 0
```

```
    def check(x,y):
        return 0<=x<=999 and 0<=y<=999
    queue1=deque([(x1,y1,0)])
    dict1={(x1,y1):0}
    queue2=deque([(x2,y2,0)])
    dict2={(x2,y2):0}
    while queue1 or queue2:
        if len(queue1)<len(queue2):
            x,y,t=queue1.popleft()
            d=[(1,2),(-1,2),(1,-2),(-1,-2),(2,1),(-2,1),(2,-1),(-2,-1)]
            for i in range(8):
                x3=x+d[i][0]
                y3=y+d[i][1]
                if check(x3,y3) and (x3,y3) not in dict1:
                    queue1.append((x3,y3,t+1))
                    dict1[(x3,y3)]=t+1
                    if (x3,y3) in dict2:
                        return dict1[(x3,y3)]+dict2[(x3,y3)]
        else:
            x,y,t=queue2.popleft()
            d=[(1,2),(-1,2),(1,-2),(-1,-2),(2,1),(-2,1),(2,-1),(-2,-1)]
            for i in range(8):
                x3=x+d[i][0]
                y3=y+d[i][1]
                if check(x3,y3) and (x3,y3) not in dict2:
                    queue2.append((x3,y3,t+1))
                    dict2[(x3,y3)]=t+1
                    if (x3,y3) in dict1:
                        return dict1[(x3,y3)]+dict2[(x3,y3)]
for _ in range(int(input())):
    x1,y1,x2,y2=map(int,input().split())
    print(solve(x1,y1,x2,y2))
```

- Dijkstra算法

在传统的`bfs`题中，不同路径所对应的开销是相同的。如果为不同路径加上不同的权重，那么首次到达某点的路径就不一定是开销最短的路径。这时候就应当使用Dijkstra算法。

一般的教材中解释Dijkstra时，用的都是类似于“价格”、“开销”之类的表述，因此不容易理解其工作原理。笔者发现，如果不用空间上的开销，改用时间上的开销，就非常容易理解Dijkstra的深刻内涵。

现在我们建立这样一个模型：

在一个岛的不同位置上住着一些人，其中某些人之间用网线连接起来，他们之间传播信息所需的时间为确定值。现在其中某一个人获得了一条信息，他需要把这条信息传遍全岛，问岛上每个人

接受到该信息的时间。

我们想象一条单调递增的时间轴。在 `t=0` 时刻，初始获得信息的人把这条信息传给了与他直接相邻的所有人。想象一个辐射状的信息传播网，信息沿这些网传播给其他人。

在这些人中，必然有一个最快接受到信息的人，设他在 `t1` 时刻接受到信息。那么他会马上把这条信息传给与他直接相邻的人，于是这个网变大了，扩展到了更多的人。

下一次信息传播网扩展的时候是什么时候呢？当然是下一个最快的人接受到信息的时候。由此可见，为了确定信息传播网的状态，我们每次要从当前所有人中，找到下一个接受到信息的人，并利用他与周围的人的联系，扩展信息传播网，直到覆盖全岛。

这就是Dijkstra算法的核心思想：每次找到当前开销最小的节点，并利用它更新网络。

对于网络中的某个人，可能有多个人连接到他，因此他可能多次接受到该信息。其中他第一次接受到信息是什么时候呢？就是他首次并入到信息传播网络的时候，而此时他就是所有未接收到信息的人中，最先接受到信息的人。

由此可见，Dijkstra算法之所以有效，是因为每次我们找到的开销最小的点，它的实际最小开销一定就是我们找到他时的开销。

只要知道了Dijkstra算法的工作原理，那么就能很自然地写出代码了。

想一想：我们需要从所有未探索的节点中找到开销最小的节点，如何实现这一点？没错，我们要使用堆。因此Dijkstra算法中，原来的队列被堆所代替。除此之外，该标记的还是得标记。

我们来看例题：

OpenJudge - T20106:走山路

本题中与某一点相邻的节点，就是其周围四个点，而开销是两者之间高度差的绝对值。我们用 `dict1` 记录到达各个点的开销，每次从 `heap` 中取出当前开销最小的点，对于其周围的四个点，如果其开销小于原本的开销，就进行更新，把相邻点加入 `heap` 中。

```
import heapq
m,n,p=map(int,input().split())
grid=[]
for i in range(m):
    row=list(input().split())
    for j in range(n):
        if row[j].isdigit():
            row[j]=int(row[j])
    grid.append(row)
def check(x,y):
    return 0<=x<=m-1 and 0<=y<=n-1
for _ in range(p):
```

```
    x1,y1,x2,y2=map(int,input().split())
    if grid[x1][y1]=='#' or grid[x2][y2]=='#':
        print('NO')
        continue
    heap=[(0,x1,y1)]
    heapq.heapify(heap)
    dict1={(x1,y1):0}
    while heap:
        t,x,y=heapq.heappop(heap)
        if (x,y)==(x2,y2):
            print(t)
            break
        d=[(1,0),(-1,0),(0,1),(0,-1)]
        for i in range(4):
            x1=x+d[i][0]
            y1=y+d[i][1]
            if check(x1,y1) and grid[x1][y1]!='#':
                t1=t+abs(grid[x1][y1]-grid[x][y])
                if dict1.get((x1,y1),float('inf'))>t1:
                    heapq.heappush(heap,(t1,x1,y1))
                    dict1[(x1,y1)]=t1
    else:
        print('NO')
```

OpenJudge - 28972:海拔

本题中的开销定义为走过路径中高差的最大值，我们可以类似地写出传递公式，这样就与上题没什么区别了。

```
import heapq
n,m=map(int,input().split())
grid=[]
for _ in range(n):
    grid.append(list(map(int,input().split())))
def check(x,y):
    return 0<=x<=n-1 and 0<=y<=m-1
res=[[float('inf')]*m for _ in range(n)]
res[0][0]=0
heap=[(0,0,0)]
heapq.heapify(heap)
while heap:
    t,x,y=heapq.heappop(heap)
    if (x,y)==(n-1,m-1):
        print(t)
        break
    d=[(1,0),(-1,0),(0,1),(0,-1)]
```

```
    for i in range(4):
        x1=x+d[i][0]
        y1=y+d[i][1]
        if check(x1,y1):
            c=max(t,abs(grid[x1][y1]-grid[x][y]))
            if res[x1][y1]>c:
                heapq.heappush(heap,(c,x1,y1))
                res[x1][y1]=c
```

完结撒花，感谢陪伴！