**Cheatsheet**

**Mould**

一 最大单调上升子序列：

import bisect

n = int(input())

\*lis, = map(int, input().split())

dp = [1e9]\*n

for i in lis:

dp[bisect.bisect\_left(dp, i)] = i

print(bisect.bisect\_left(dp, 1e8))

二 kadane算法

三 欧拉筛

四 背包dp

# 压缩矩阵/滚动数组 方法

N,B = map(int, input().split())

\*p, = map(int, input().split())

\*w, = map(int, input().split())

dp=[0]\*(B+1)

for i in range(N):

for j in range(B, w[i] - 1, -1):

dp[j] = max(dp[j], dp[j-w[i]]+p[i])

print(dp[-1])

五 Dijkstra

重要提示：n=int(input())  
ma=[list(map(int,input())) for \_ in range(n)]  
m,dire=len(ma[0]),[(1,0),(-1,0),(0,1),(0,-1)]  
for i in range(n):  
 for j in range(m):  
 if ma[i][j]==1:  
 x1,y1=i,j  
import heapq  
def dijkstra(x1,y1):  
 q,visited=[],[[False]\*m for \_ in range(n)]  
 heapq.heappush(q,(0,x1,y1))  
 while q:  
 step,x,y=heapq.heappop(q)  
 if visited[x][y]:  
 continue  
 visited[x][y] = True  
 if ma[x][y]==1 and step!=0:  
 return step  
 for dx,dy in dire:  
 if 0<=x+dx<n and 0<=y+dy<m and not visited[x+dx][y+dy]:  
 heapq.heappush(q,(step+1-ma[x+dx][y+dy],x+dx,y+dy))  
print(dijkstra(x1,y1))

1.heappop之后要判断一次是否visited，搜索的时候同样也要判断一次是否visited（即所谓剪枝），因为同一位置可能由不同位置以相同步数达到

2.指标要放在元组的最前面

六 后悔型greedy：

可以Import heapq之类的数据结构记下选择，如果不是最优就反悔，heappop

七 区间问题

位运算

分治

八 构建多个dp数组：

使用两个 DP 数组分别处理上述两种情况：

1. dp1[i] 表示 **以第 i 个商品结尾的连续子数组最大和**，不考虑放回商品。
   * 状态转移公式：dp1[i] = max(dp1[i - 1] + a[i], a[i])
   * 解释：当前商品 a[i] 要么加入之前的子数组，要么单独成为一个新的子数组。
2. dp2[i] 表示 **以第 i 个商品结尾的连续子数组最大和**，允许放回其中一个商品。
   * 状态转移公式：dp2[i] = max(dp1[i - 1], dp2[i - 1] + a[i], a[i])
   * 解释：
     + dp1[i - 1] 表示选择前面的子数组，但不加当前商品。
     + dp2[i - 1] + a[i] 表示当前商品加入到之前可能已经放回一个商品的子数组。
     + a[i] 表示单独选择当前商品。

最终答案是 dp2 数组中的最大值。因为 dp2 考虑了放回一个商品的情况，所以它能提供最大的可能价值。

**时间复杂度**

* **时间复杂度**: O(n)，因为只需要线性遍历数组一次。
* **空间复杂度**: O(n)（可以进一步优化到 O(1)）。

**关键点总结**

* dp1 用于计算不放回商品时的最大和。
* dp2 在 dp1 的基础上扩展，允许放回一个商品。
* 土豪不能空手而归，因此需要注意初始条件设置和最终答案选择。

好巧妙的解法，构造两个数组，一个都不放会，一个允许放回一次。这个思路可以多学习一下。在答案的基础上把空间复杂度优化到O(n)

# 俞天麒 24物理学院

a=list(map(int,input().split(",")))

n=len(a)

dp1,dp2=a[0],a[0]

ans=0

for i in range(1,n):

dp1,dp2=max(a[i],dp1+a[i]),max(dp1,dp2+a[i])

ans=max(ans,dp2)

print(ans)

需要考虑两种情况：

1. 不放回商品时的最大连续子数组和（Kadane算法）。
2. 放回一个商品时的最大连续子数组和。

由于我们可以选择放回任何一个商品，因此需要考虑放回每一个商品对最大连续子数组和的影响。我们可以通过两次遍历数组来解决这个问题：

* 第一次遍历从左到右计算以每个元素结尾的最大子数组和。
* 第二次遍历从右到左计算以每个元素开始的最大子数组和。

然后，我们遍历数组，对于每个位置，我们尝试放回该位置的商品，并检查如果放回这个商品后，左边子序列的最大和加上右边子序列的最大和是否会比当前的最大值还要大。

**Useful functions**

Zip

zip 是 Python 内置的一个函数，它用于将多个可迭代对象（如列表、元组等）中的元素一一配对，返回一个由这些配对组成的迭代器。通常情况下，zip 用于并行地遍历两个或更多的序列。zip 函数会根据最短的输入序列来截断结果，即当最短的输入序列用尽时，zip 将停止生成配对。

**zip 的基本使用**

python

深色版本

names = ["Alice", "Bob", "Charlie"]

ages = [25, 30, 35]

for name, age in zip(names, ages):

print(f"{name} is {age} years old.")

这段代码会输出：

深色版本

Alice is 25 years old.

Bob is 30 years old.

Charlie is 35 years old.

**zip 的特性**

* **多输入**：你可以传递任意数量的可迭代对象给 zip。
* **不同长度的输入**：如果传递给 zip 的可迭代对象长度不一，zip 会在最短的那个用尽时停止。
* **解压**：zip 可以与 \* 操作符一起使用来进行“解压”，即将之前压缩的数据重新分配回原始的序列中。

例如：

python

深色版本

pairs = [(1, 'a'), (2, 'b'), (3, 'c')]

numbers, letters = zip(\*pairs)

print(numbers) # 输出: (1, 2, 3)

print(letters) # 输出: ('a', 'b', 'c')

**使用 zip\_longest**

如果你需要处理长度不同的可迭代对象，并且希望在较短的序列用尽后继续迭代，可以使用 itertools.zip\_longest 函数。这个函数会用指定的填充值（默认是 None）来填充较短的序列，直到所有序列都同时用尽。

python

深色版本

from itertools import zip\_longest

names = ["Alice", "Bob"]

ages = [25, 30, 35]

for name, age in zip\_longest(names, ages, fillvalue="Unknown"):

print(f"{name} is {age} years old.")

这段代码会输出：

深色版本

Alice is 25 years old.

Bob is 30 years old.

Unknown is 35 years old.

**zip 的应用**

zip 经常用于各种场景，比如并行迭代、创建字典、矩阵转置等等。它的简洁性和高效性使得它成为Python编程中非常有用的工具之一。如果你有更具体的应用场景或问题，请提供更多信息，我可以进一步帮助你。

Enumerate

enumerate 是 Python 内置的一个函数，它允许你在遍历一个可迭代对象（如列表、元组、字符串等）的同时获取元素的索引。这在你需要知道当前处理的是序列中的第几个元素时特别有用，可以避免手动维护一个计数器。

**enumerate 的基本用法**

enumerate 函数返回一个枚举对象，该对象是一个迭代器，产生由索引和值组成的元组对。你可以通过解包这些元组来同时访问索引和值。

python

深色版本

for index, value in enumerate(['apple', 'banana', 'cherry']):

print(f"Index {index}: {value}")

这段代码会输出：

深色版本

Index 0: apple

Index 1: banana

Index 2: cherry

**enumerate 的参数**

enumerate 接受两个参数：

1. **iterable**：你想要枚举的可迭代对象。
2. **start**（可选）：指定索引的起始值，默认为 0。

例如，如果你想从 1 开始计数：

python

深色版本

for index, value in enumerate(['apple', 'banana', 'cherry'], start=1):

print(f"Item {index}: {value}")

这段代码会输出：

深色版本

Item 1: apple

Item 2: banana

Item 3: cherry

**使用场景**

* **并行迭代**：当你需要同时迭代一个序列及其索引时非常方便。
* **创建字典或集合**：可以轻松地将一个序列转换为字典或集合，其中键是索引，值是序列中的元素。
* **简化代码**：避免使用额外的计数变量来追踪当前位置。

**示例：创建字典**

python

深色版本

fruits = ['apple', 'banana', 'cherry']

fruit\_dict = dict(enumerate(fruits))

print(fruit\_dict) # 输出: {0: 'apple', 1: 'banana', 2: 'cherry'}

**示例：与 zip 结合使用**

enumerate 还可以与其他函数如 zip 结合使用，以实现更复杂的逻辑。

python

深色版本

names = ["Alice", "Bob", "Charlie"]

scores = [88, 95, 70]

修改递归深度：

import sys

sys.setrecursionlimit(1>>30)

一次性读入数据：

Import sys

Data=sys.stdin.read().split()

#以字符串方式读入

#本地调试可以ctrl+d提供EOF信号

Dfs缓存：

From functools import lru\_cache

@lru\_cache(maxsize = None)

浅拷贝：

=Copy.deepcopy（list）为深拷贝

List[:]为浅拷贝

判断数据类型：

Isinstance（object，int）（判断是否为整型）

判断一个字符是否为数字：

Object.isnumeric()

在 Python 中，你可以使用字符串的方法 isdigit() 来判断一个字符（或字符串）是否全由数字组成。这个方法会返回一个布尔值：如果字符串中的所有字符都是数字，并且字符串至少有一个字符，则返回 True；否则返回 False。

这里是一个简单的例子，用于判断单个字符是否为数字：

python

深色版本

def is\_digit(character):

return character.isdigit()

# 测试

char = '5'

print(is\_digit(char)) # 输出: True

char = 'a'

print(is\_digit(char)) # 输出: False

需要注意的是，isdigit() 方法也会将上标的数字（如 '²'）、罗马数字（在 Unicode 中）和其他非阿拉伯数字的数值字符视为数字。如果你只想检查常规的阿拉伯数字（0-9），你可以使用字符串的 isdecimal() 或者 isnumeric() 方法，或者是直接用一个简单的比较操作：

python

深色版本

def is\_arabic\_digit(character):

return character >= '0' and character <= '9'

# 或者使用 str.isdecimal()

def is\_decimal(character):

return character.isdecimal()

# 测试

char = '5'

print(is\_arabic\_digit(char)) # 输出: True

print(is\_decimal(char)) # 输出: True

char = 'Ⅴ' # 罗马数字5

print(is\_arabic\_digit(char)) # 输出: False

print(is\_decimal(char)) # 输出: False

请根据你的具体需求选择合适的方法。

**Useful data structure**

Deque

Heapq：

heapq 是 Python 标准库中的一个模块，它实现了最小堆（min-heap），这是一种特殊的二叉树数据结构。最小堆的特点是堆顶元素总是最小的。heapq 模块提供了一系列函数来处理堆，这些函数可以在 O(log n) 的时间复杂度内插入和弹出元素，非常适合用来实现优先队列。

**heapq 常用函数**

1. **heapq.heappush(heap, item)**:
   * 将 item 添加到堆 heap 中，并保持堆的性质。
2. **heapq.heappop(heap)**:
   * 从堆 heap 中弹出并返回最小的元素。如果堆为空，则会引发 IndexError。
3. **heapq.heapify(x)**:
   * 将列表 x 转换为堆。这个操作是原地完成的，时间复杂度为 O(n)，其中 n 是列表的长度。
4. **heapq.heappushpop(heap, item)**:
   * 先将 item 推入堆中，然后弹出并返回堆中最小的元素。这个组合操作比单独调用 heappush 和 heappop 更加高效。
5. **heapq.heapreplace(heap, item)**:
   * 弹出并返回堆中最小的元素，同时将 item 放入堆中。该操作也比先 pop 再 push 更有效率。
6. **heapq.nlargest(n, iterable, key=None)**:
   * 返回 iterable 中最大的 n 个元素组成的列表。key 参数可以用于指定比较的关键字。
7. **heapq.nsmallest(n, iterable, key=None)**:
   * 返回 iterable 中最小的 n 个元素组成的列表。key 参数同样可以用于指定比较的关键字。

**使用 heapq 实现优先队列**

在很多算法中，例如 Dijkstra 最短路径算法或 A\* 搜索算法，我们需要根据节点的权重或代价来决定下一个要处理的节点。heapq 提供了高效的接口来管理这些节点，确保我们可以快速找到具有最低权重或代价的节点。

**Chores**

在 PyCharm 中设置自动补全，可以通过调整编辑器的设置来实现。以下是一些基本步骤和选项，你可以根据自己的需要进行配置：

启用代码完成：

打开 PyCharm。

进入 File > Settings（在 macOS 上是 PyCharm > Preferences）。

在左侧菜单中选择 Editor > General > Code Completion。

调整代码完成设置：

在 Code Completion 页面，你可以看到几个选项卡，如 General, AutoPopup, Parameters Info 等等。

你可以勾选或取消勾选特定的选项来控制哪些类型的建议应该显示，以及如何显示它们。

例如，你可以设置是否在你输入时自动弹出代码完成建议 (AutoPopup code completion)。

设置热键：

如果你想自定义触发代码完成的快捷键，可以在 Keymap 设置页面找到与代码完成相关的命令，并为它们分配新的快捷键。

路径是 Settings/Preferences > Keymap，然后在搜索框中输入 "code completion" 来查找相关项。

Live Templates (实时模板)：

PyCharm 提供了 Live Templates 功能，可以让你创建自定义的代码片段，这些代码片段可以在你输入预设缩写时自动展开。

你可以通过 Settings/Preferences > Editor > Live Templates 来管理现有的模板或者添加新的模板。

Python-specific settings (Python 特定设置)：

对于 Python 项目，你还可以调整特定于语言的代码完成设置。这可能包括类型提示、框架支持等。

可以通过 Languages & Frameworks > Python 下的相关子项来访问这些设置。

插件支持：

某些插件也可能提供额外的代码完成功能。确保你安装了所有必要的插件，并且它们是最新的。

重新索引项目：

如果你觉得代码完成没有像预期那样工作，可能是由于索引过期或损坏。你可以尝试重新索引项目：File > Invalidate Caches / Restart，然后点击 Invalidate and Restart。

请记住，不同的 PyCharm 版本可能会有略微不同的用户界面和选项名称，但总体流程应该是相似的。如果你使用的是社区版还是专业版，可用的功能也可能会有所不同。