Method Name	Use	Explanation
`append`	`alist.append(item)`	Adds a new item to the end of a list
`insert`	`alist.insert(i,item)`	Inserts an item at the ith position in a list
`pop`	`alist.pop()`	Removes and returns the last item in a list
`pop`	`alist.pop(i)`	Removes and returns the ith item in a list
`sort`	`alist.sort()`	Modifies a list to be sorted
`reverse`	`alist.reverse()`	Modifies a list to be in reverse order
`del`	`del alist[i]`	Deletes the item in the ith position
`index`	`alist.index(item)`	Returns the index of the first occurrence of `item`
`count`	`alist.count(item)`	Returns the number of occurrences of `item`
`remove`	`alist.remove(item)`	Removes the first occurrence of `item`

python列表的方法

Method Name	Use	Explanation
`center`	`astring.center(w)`	Returns a string centered in a field of size `w`
`count`	`astring.count(item)`	Returns the number of occurrences of `item` in the string
`ljust`	`astring.ljust(w)`	Returns a string left-justified in a field of size `w`
`lower`	`astring.lower()`	Returns a string in all lowercase
`rjust`	`astring.rjust(w)`	Returns a string right-justified in a field of size `w`
`find`	`astring.find(item)`	Returns the index of the first occurrence of `item`
`split`	`astring.split(schar)`	Splits a string into substrings at `schar`

python字符串的方法

表1-7 Python字典支持的运算

Operator	Use	Explanation
`[]`	`myDict[k]`	Returns the value associated with `k`, otherwise its an error
`in`	`key in adict`	Returns `True` if key is in the dictionary, `False` otherwise

Operator	Use	Explanation
`del`	del `adict[key]`	Removes the entry from the dictionary

表1-8 Python字典提供的方法

Method Name	Use	Explanation
`keys`	`adict.keys()`	Returns the keys of the dictionary in a dict_keys object
`values`	`adict.values()`	Returns the values of the dictionary in a dict_values object
`items`	`adict.items()`	Returns the key-value pairs in a dict_items object
`get`	`adict.get(k)`	Returns the value associated with `k`, `None` otherwise
`get`	`adict.get(k,alt)`	Returns the value associated with `k`, `alt` otherwise

语法工具

math

gcd包，计算最大公因式

```
from math import gcd
x = gcd(15,20,25)
print(x)
## 5
```

math.pow(m,n)　计算`m`的`n`次幂。

math.log(m,n)　计算以`n`为底的`m`的对数。

类的打印方法

```
def __str__(self):
    return str(self.num)+"/"+str(self.den)
>>> myf = Fraction(3,5)
>>> print(myf)
3/5
>>> print("I ate", myf, "of the pizza")
I ate 3/5 of the pizza
>>> myf.__str__()
'3/5'
>>> str(myf)
'3/5'
>>>
```

中序转后序-调度场算法

```
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''

    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <=
precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()

    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)

    while stack:
        postfix.append(stack.pop())

    return ' '.join(str(x) for x in postfix)

n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))
```

八皇后问题回溯实现（dfs）

```
def solve_n_queens(n):
    solutions = []  # 存储所有解决方案的列表
    queens = [-1] * n  # 存储每一行皇后所在的列数

    def backtrack(row):
        if row == n:  # 找到一个合法解决方案
            solutions.append(queens.copy())
        else:
            for col in range(n):
                if is_valid(row, col):  # 检查当前位置是否合法
                    queens[row] = col  # 在当前行放置皇后
```

```
                    backtrack(row + 1)  # 递归处理下一行
                    queens[row] = -1  # 回溯，撤销当前行的选择

    def is_valid(row, col):
        for r in range(row):
            if queens[r] == col or abs(row - r) == abs(col - queens[r]):
                return False
        return True

    backtrack(0)  # 从第一行开始回溯

    return solutions


# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = solve_n_queens(8)
    if b > len(solutions):
        return None
    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string


test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

二叉树的深度（递归，输入）

```
class TreeNode:
    def __init__(self):
        self.left = None
        self.right = None

def tree_depth(node):
    if node is None:
        return 0
    left_depth = tree_depth(node.left)
    right_depth = tree_depth(node.right)
    return max(left_depth, right_depth) + 1

n = int(input())  # 读取节点数量
nodes = [TreeNode() for _ in range(n)]

for i in range(n):
    left_index, right_index = map(int, input().split())
    if left_index != -1:
        nodes[i].left = nodes[left_index-1]
    if right_index != -1:
        nodes[i].right = nodes[right_index-1]

root = nodes[0]
depth = tree_depth(root)
```

```
print(depth)
```

二叉树的高度与叶子数目（输入，递归，找根节点）

```
class TreeNode:
    def __init__(self):
        self.left = None
        self.right = None

def tree_height(node):
    if node is None:
        return -1  # 根据定义，空树高度为-1
    return max(tree_height(node.left), tree_height(node.right)) + 1

def count_leaves(node):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return 1
    return count_leaves(node.left) + count_leaves(node.right)

n = int(input())  # 读取节点数量
nodes = [TreeNode() for _ in range(n)]
has_parent = [False] * n  # 用来标记节点是否有父节点

for i in range(n):
    left_index, right_index = map(int, input().split())
    if left_index != -1:
        nodes[i].left = nodes[left_index]
        has_parent[left_index] = True
    if right_index != -1:
        #print(right_index)
        nodes[i].right = nodes[right_index]
        has_parent[right_index] = True

# 寻找根节点，也就是没有父节点的节点
root_index = has_parent.index(False)
root = nodes[root_index]

# 计算高度和叶子节点数
height = tree_height(root)
leaves = count_leaves(root)

print(f"{height} {leaves}")
```

括号嵌套树+前序和后序遍历

```
class TreeNode:
    def __init__(self, value): #类似字典
        self.value = value
        self.children = []

def parse_tree(s):
    stack = []
```

```
    node = None
    for char in s:
        if char.isalpha():  # 如果是字母，创建新节点
            node = TreeNode(char)
            if stack:  # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中
                stack[-1].children.append(node)
        elif char == '(':  # 遇到左括号，当前节点可能会有子节点
            if node:
                stack.append(node)  # 把当前节点推入栈中
                node = None
        elif char == ')':  # 遇到右括号，子节点列表结束
            if stack:
                node = stack.pop()  # 弹出当前节点
    return node  # 根节点


def preorder(node):
    output = [node.value]
    for child in node.children:
        output.extend(preorder(child))
    return ''.join(output)

def postorder(node):
    output = []
    for child in node.children:
        output.extend(postorder(child))
    output.append(node.value)
    return ''.join(output)

# 主程序
def main():
    s = input().strip()
    s = ''.join(s.split())  # 去掉所有空白字符
    root = parse_tree(s)  # 解析整棵树
    if root:
        print(preorder(root))  # 输出前序遍历序列
        print(postorder(root))  # 输出后序遍历序列
    else:
        print("input tree string error!")

if __name__ == "__main__":
    main()
```

哈夫曼树

```
import heapq

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
```

```
        return self.freq < other.freq

def huffman_encoding(char_freq):
    heap = [Node(char, freq) for char, freq in char_freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq) # note: 合并之后 char 字典是空
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

def external_path_length(node, depth=0):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return depth * node.freq
    return (external_path_length(node.left, depth + 1) +
            external_path_length(node.right, depth + 1))

def main():
    char_freq = {'a': 3, 'b': 4, 'c': 5, 'd': 6, 'e': 8, 'f': 9, 'g': 11, 'h':
12}
    huffman_tree = huffman_encoding(char_freq)
    external_length = external_path_length(huffman_tree)
    print("The weighted external path length of the Huffman tree is:",
external_length)

if __name__ == "__main__":
    main()
```

AVL平衡二叉树

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

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

    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self.root = self._insert(value, self.root)

    def _insert(self, value, node):
```

```
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(value, node.left)
        else:
            node.right = self._insert(value, node.right)

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        balance = self._get_balance(node)

        if balance > 1:
            if value < node.left.value: # 树形是 LL
                return self._rotate_right(node)
            else:   # 树形是 LR
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if value > node.right.value:    # 树形是 RR
                return self._rotate_left(node)
            else:   # 树形是 RL
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def _rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        x.height = 1 + max(self._get_height(x.left), self._get_height(x.right))
        return x
```

```
    def preorder(self):
        return self._preorder(self.root)

    def _preorder(self, node):
        if not node:
            return []
        return [node.value] + self._preorder(node.left) +
self._preorder(node.right)

n = int(input().strip())
sequence = list(map(int, input().strip().split()))

avl = AVL()
for value in sequence:
    avl.insert(value)

print(' '.join(map(str, avl.preorder())))
```

图的类实现

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in
self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
```

```
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,weight=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], weight)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
```

BFS算法

```
rom collections import defaultdict, deque

# Class to represent a graph using adjacency list
class Graph:
    def __init__(self):
        self.adjList = defaultdict(list)

    # Function to add an edge to the graph
    def addEdge(self, u, v):
        self.adjList[u].append(v)

    # Function to perform Breadth First Search on a graph represented using
adjacency list
    def bfs(self, startNode):
        # Create a queue for BFS
        queue = deque()
        visited = set()

        # Mark the current node as visited and enqueue it
        visited.add(startNode)
        queue.append(startNode)

        # Iterate over the queue
        while queue:
            # Dequeue a vertex from queue and print it
            currentNode = queue.popleft()
            print(currentNode, end=" ")

            # Get all adjacent vertices of the dequeued vertex currentNode
            # If an adjacent has not been visited, then mark it visited and
enqueue it
            for neighbor in self.adjList[currentNode]:
                if neighbor not in visited:
```

```
                    visited.add(neighbor)
                    queue.append(neighbor)
```

DFS算法

```
from collections import defaultdict


class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFS(self, v, visited=None):
        if visited is None:
            visited = set()
        visited.add(v)
        print(v, end=' ')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFS(neighbour, visited)


# Driver's code
if __name__ == "__main__":
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    print("Following is Depth First Traversal (starting from vertex 2)")

    # Function call
    g.DFS(2)
"""
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

判断无向图是否连通

```
def dfs(node, visited, adjacency_list):
    visited[node] = True
    for neighbor in adjacency_list[node]:
        if not visited[neighbor]:
            dfs(neighbor, visited, adjacency_list)

n, m = map(int, input().split())
adjacency_list = [[] for _ in range(n)]
for _ in range(m):
```

```
    u, v = map(int, input().split())
    adjacency_list[u].append(v)
    adjacency_list[v].append(u)

visited = [False] * n
dfs(0, visited, adjacency_list)

if all(visited):
    print("Yes")
else:
    print("No")
```

判断有向图是否有环

```
def has_cycle(n, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)

    color = [0] * n

    def dfs(node):
        if color[node] == 1:
            return True
        if color[node] == 2:
            return False

        color[node] = 1
        for neighbor in graph[node]:
            if dfs(neighbor):
                return True
        color[node] = 2
        return False

    for i in range(n):
        if dfs(i):
            return "Yes"
    return "No"

# 接收数据
n, m = map(int, input().split())
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
print(has_cycle(n, edges))
```

无向图连通块最大权值

```
def max_weight(n, m, weights, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
```

```
        graph[u].append(v)
        graph[v].append(u)

    visited = [False] * n
    max_weight = 0

    def dfs(node):
        visited[node] = True
        total_weight = weights[node]
        for neighbor in graph[node]:
            if not visited[neighbor]:
                total_weight += dfs(neighbor)
        return total_weight

    for i in range(n):
        if not visited[i]:
            max_weight = max(max_weight, dfs(i))

    return max_weight

# 接收数据
n, m = map(int, input().split())
weights = list(map(int, input().split()))
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
print(max_weight(n, m, weights, edges))
```

4.dp

最大上升子序列

```
input()
b = [int(x) for x in input().split()]

n = len(b)
dp = [0]*n

for i in range(n):
   dp[i] = b[i]
   for j in range(i):
      if b[j]<b[i]:
         dp[i] = max(dp[j]+b[i], dp[i])


print(max(dp))
```

2. Bellman-Ford算法

Bellman-Ford算法：Bellman-Ford算法用于解决单源最短路径问题，与Dijkstra算法不同，它可以处理带有负权边的图。算法的基本思想是通过松弛操作逐步更新节点的最短路径估计值，直到收敛到最终结果。具体步骤如下：

- 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，源节点的距离为0，其他节点的距离为无穷大。
- 进行V-1次循环（V是图中的节点数），每次循环对所有边进行松弛操作。如果从节点u到节点v的路径经过节点u的距离加上边(u, v)的权重比当前已知的从源节点到节点v的最短路径更短，则更新最短路径。
- 检查是否存在负权回路。如果在V-1次循环后，仍然可以通过松弛操作更新最短路径，则说明存在负权回路，因此无法确定最短路径。

class Graph:
  def **init**(self, vertices):
    self.V = vertices
    self.graph = []

```
def add_edge(self, u, v, w):
    self.graph.append([u, v, w])

def bellman_ford(self, src):
    # 初始化距离数组，表示从源点到各个顶点的最短距离
    dist = [float('inf')] * self.V
    dist[src] = 0

    # 迭代 V-1 次，每次更新所有边
    for _ in range(self.V - 1):
        for u, v, w in self.graph:
            if dist[u] != float('inf') and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w

    # 检测负权环
    for u, v, w in self.graph:
        if dist[u] != float('inf') and dist[u] + w < dist[v]:
            return "Graph contains negative weight cycle"

    return dist
```

测试代码

g = Graph(5)
g.add_edge(0, 1, -1)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 3)
g.add_edge(1, 3, 2)
g.add_edge(1, 4, 2)
g.add_edge(3, 2, 5)
g.add_edge(3, 1, 1)
g.add_edge(4, 3, -3)

src = 0
distances = g.bellman_ford(src)
print("最短路径距离：")
for i in range(len(distances)):

print(f"从源点 {src} 到顶点 {i} 的最短距离为：{distances[i]}")

3 多源最短路径Floyd-Warshall算法

求解所有顶点之间的最短路径可以使用**Floyd-Warshall算法**，它是一种多源最短路径算法。Floyd-Warshall算法可以**在有向图或无向图中找到任意两个顶点之间的最短路径。**

算法的基本思想是通过一个二维数组来存储任意两个顶点之间的最短距离。初始时，这个数组包含图中各个顶点之间的直接边的权重，对于不直接相连的顶点，权重为无穷大。然后，通过迭代更新这个数组，逐步求得所有顶点之间的最短路径。

具体步骤如下：

1. 初始化一个二维数组`dist`，用于存储任意两个顶点之间的最短距离。初始时，`dist[i][j]`表示顶点i到顶点j的直接边的权重，如果i和j不直接相连，则权重为无穷大。
2. 对于每个顶点k，在更新`dist`数组时，考虑顶点k作为中间节点的情况。遍历所有的顶点对(i, j)，如果通过顶点k可以使得从顶点i到顶点j的路径变短，则更新`dist[i][j]`为更小的值。

   `dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])`
3. 重复进行上述步骤，对于每个顶点作为中间节点，进行迭代更新`dist`数组。最终，`dist`数组中存储的就是所有顶点之间的最短路径。

Floyd-Warshall算法的时间复杂度为O(V^3)，其中V是图中的顶点数。它适用于解决稠密图（边数较多）的最短路径问题，并且可以处理负权边和负权回路。

以下是一个使用Floyd-Warshall算法求解所有顶点之间最短路径的示例代码：

def floyd_warshall(graph):
    n = len(graph)
    dist = [[float('inf')] * n for _ in range(n)]

```
for i in range(n):
    for j in range(n):
        if i == j:
            dist[i][j] = 0
        elif j in graph[i]:
            dist[i][j] = graph[i][j]

for k in range(n):
    for i in range(n):
        for j in range(n):
            dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

return dist
```

cheat sheet

一、几个模版

最小生成树

```
#最小生成树prim,输入用mat存编号为0~n-1的邻接表，表内元素为(x,d)x为下一个点d为权值，输出最小总权值
import heapq
def solve(mat):
    h=[(0,0)]
    n=len(mat)
    vis=[1 for i in range(n)]
    ans=0
    while h:
        (d,x)=heapq.heappop(h)
        if vis[x]:
            vis[x]=0
            ans+=d
            for (y,t) in mat[x]:
                if vis[y]:
                    heapq.heappush(h,(t,y))
    return ans
```

```
#最小生成树kruskal,输入同上
import heapq
p=[i for i in range(n)]
def find(x):
    if p[x]==x:
        return x
    p[x]=find(p[x])
    return p[x]
def union(x,y):
    u,v=find(x),find(y)
    p[u]=v
def solve(mat):
    h=[]
    ans=0
    for i in range(n):
        for (j,d) in mat[i]:
            heapq.heappush(h,(d,i,j))
    while h:
        (d,i,j)=heapq.heappop(h)
        if find(i)!=find(j):
            union(i,j)
            ans+=d
    return ans
```

最短路径

```
#dijkstra，输入同上,求出from和to之间的最短路径，需要保证输入无负权值，输出-1表示无法到达to
#若输入to则处理点对点问题，否则返回所有点的最短距离
import heapq
def solve(mat,f,to=-1):
    h=[(0,f)]
    n=len(mat)
    vis=[-1 for i in range(n)]
    while h:
        (d,x)=heapq.heappop(h)
        if x==to:
            return d
        if vis[x]==-1:
            vis[x]=d
            for (y,s) in mat[x]:
                if vis[y]==-1:
                    heapq.heappush(h,(d+s,y))
    return vis
```

拓扑排序

```
#输入为mat存邻接表，mat[i]为i指向的点的列表，输出-1表示有环，顺便做了输出字典序最小排序方式
import heapq
def solve(mat):
    n=len(mat)
    h=[]
    ru=[len(mat[i]) for i in range(n)]
    ans=[]
    for i in range(n):
        if ru[i]==0:
            heapq.heappush(h,i)
    while h:
        x=heapq.heappop(h)
        ans.append(x)
        for y in mat[x]:
            ru[y]-=1
            if ru[y]==0:
                heapq.heappush(h,y)
    if len(ans)<n:
        return -1
    else:
        return ans
```

判断无向图是否连通、有无回路

```
#输入同上，基于并查集和图论
p=[i for i in range(n)]
e=[0 for i in range(n)]
v=[1 for i in range(n)]
def find(x):
    if p[x]==x:
        return x
    p[x]=find(p[x])
    return p[x]
def union(x,y):
    s,t=find(x),find(y)
    if s==t:
        e[t]+=1
        return
    p[s]=t
    v[t]+=v[s]
    e[t]+=e[s]+1
def connect(mat):
    for i in range(n):
        for j in mat[i]:
            union(i,j)
    root=find(0)
    if v[root]==n:
        return True
    return False
def loop(mat):
    for i in range(n):
        for j in mat[i]:
            union(i,j)
    for i in range(n):
        r=find(i)
        if e[r]>=v[r]:
            return True
    return False
```

单调栈

```
#给定一个列表，输出每个元素之前小于它的最后一个元素的下标
def solve(lis):
    n=len(lis)
    stack=[]
    ans=[]
    for i in range(n):
        x=lis[i]
        while stack:
            (y,j)=stack[-1]
            if y>=x:
                stack.pop()
                continue
            break
```

```
        if not stack:
            stack.append((x,i))
            ans.append(-1)
        else:
            ans.append(stack[-1][1])
            stack.append((x,i))
    return ans
```

一些技巧&算法

堆

堆是一种特殊的树形数据结构，其中每个节点的值都小于或等于（最小堆）或大于或等于（最大堆）其子节点的值。堆分为最小堆和最大堆两种类型，其中：

- 最小堆： 父节点的值小于或等于其子节点的值。
- 最大堆： 父节点的值大于或等于其子节点的值。
  堆常用于实现优先队列和堆排序等算法。

== 看到一直要用min，max的基本都要用堆

import heapq
x = [1,2,3,5,7]

heapq.heapify(x)
###将列表转换为堆。

heapq.heappushpop(heap, item)
##将 item 放入堆中，然后弹出并返回 heap 的最小元素。该组合操作比先调用 heappush() 再调用 heappop() 运行起来更有效率

heapq.heapreplace(heap, item)
##弹出并返回最小的元素，并且添加一个新元素item

heapq.heappop(heap,item)
heapq.heappush(heap,item)