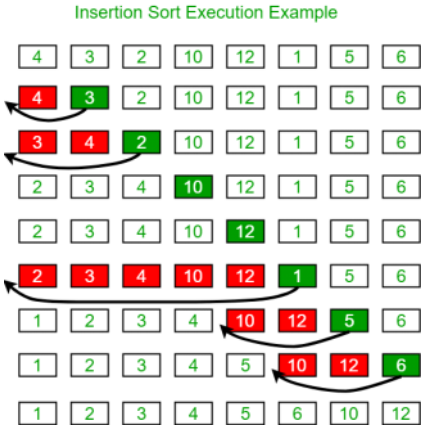
1.time complexity

Thursday, March 7, 2024 9:41

0.several notations:

1.theta-notation: above and below.
2.O-notation: only above. Used for worst-case analysis.
3.omega-notation: only below. Used for best-case.

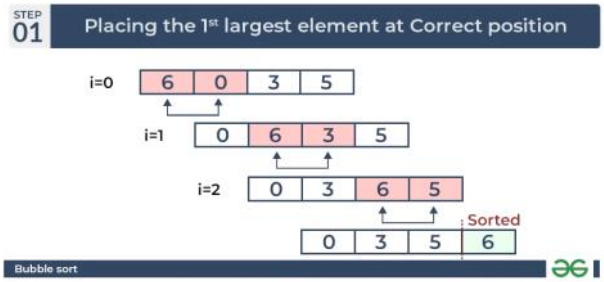
1.insertion sort: 把新元素插入到已排序的旧元素中。代码实现上，insertion是通过一步一步与前面一位互换位置实现的。O(n^2)

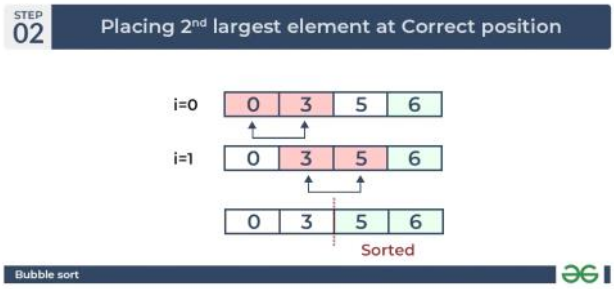


1.1.shell sort: insertion sort的改良版。从相邻的才换变成隔n个也可以换，省去了相隔很远的元素若采用相邻交换所需的巨量步骤。先确定一个较大的gap size，相隔这个size的比较并交换。然后缩小size并重复直至1. 容易忽略的细节是最深层的while是为了彻底交换到位，231-213-123. 指针j用于在表层确定开始交换的位置，指针i用于确定j后交换彻底。

```
def shellSort(arr, n):
    # code here
    gap = n // 2

    while gap > 0:
        j = gap
        # Check the array in from left to right
        # Till the last possible index of j
        while j < n:
            i = j - gap  # This will keep help in maintain gap value

            while i >= 0:
                # If value on right side is already greater than left side value
                # We don't do swap else we swap
                if arr[i + gap] > arr[i]:
                    break
                else:
                    arr[i + gap], arr[i] = arr[i], arr[i + gap]

                i = i - gap  # To check left side also
            # If the element present is greater than current element
            j += 1
        gap = gap // 2
```

2.bubble sort: 原理是每次把未排序的数中最大的放在当前的最右边。实现方式依然是每次比较相邻的两数。O(n^2)



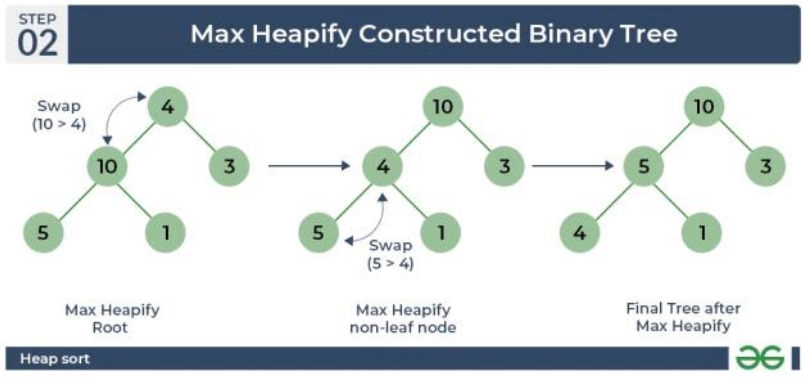


上图中实际无需再运行第三遍循环。反映在代码实现上，为了节省时间，设置一个变量为swapped，如果某一遍循环中没有任何一次交换，则排序确定。

```
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already in place
        for j in range(0, n - i - 1):

            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if (swapped == False):
            break
```

3.selection sort: 每次将最小的放在最左边，但实现方式不是相邻交换，而是遍历记录最小值的索引后，将最小值与当时最左边的交换。也就是比较次数不变，但交换次数减少到n-1次。O(n^2)

3.1 heapsort: selection sort变种，只是变成二叉树后每次将最大值换到node.



4.quicksort: 选定一个数据做pivot，将其余数据按照是否大于pivot分成两列，并将pivot放在中间，递归。最快，但不适合小数据集！

1.双指针：常考。左边的指针筛选大于pivot的值，右边筛选小于pivot的与左边的互换。两头夹击。

```
def partition(arr, left, right):
    i = left
    j = right - 1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i
```

2.单指针：指针从左边开始遍历，将所有小于pivot的值换到最左边。螺旋前进。实际上还是双指针，只不过有一个主要筛选要调换的值，有一个记录目标位置。

3.选择枢轴：可用median of three，即选择首项、末项、中点三者的中位数值作为pivot. 中点的索引为len(list)//2，也即奇数项为中点，偶数项为中点右一项。

4.复杂度：omega-theta(NlogN), O-N^2.

To analyze the `quickSort` function, note that for a list of length *n*, if the partition always occurs in the middle of the list, there will again be $logn$ divisions. In order to find the split point, each of the *n* items needs to be checked against the pivot value. The result is $nlogn$. In addition, there is no need for additional memory as in the merge sort process.

5.易错点：由于递归，因此第n遍指的范围需要格外注意！

Q. Choose the leftmost element as pivot, given the following list of numbers [14, 17, 13, 15, 19, 10, 3, 16, 9, 12] which answer shows the contents of the list after the second partitioning according to the quicksort algorithm? (D)

A. [9, 3, 10, 13, 12]
B. [9, 3, 10, 13, 12, 14]
C. [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]
D. [9, 3, 10, 13, 12, 14, 19, 16, 15, 17]

The first partitioning works on the entire list, and the second partitioning works on the left partition not the right. It's important to remember that quicksort works on the entire list and sorts it in place.

5.merge sort: 直接均分，各自merge sort后再merge. 同样用了递归，只不过worst-case也是nlogn，另外auxiliary space = o(n). 其稳定性决定了适用于大数据集。

```python
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2

        L = arr[:mid]   # Dividing the array elements
        R = arr[mid:] # Into 2 halves

        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half

        i = j = k = 0
        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

*divide and conquer: 将母题分成几个小问题，逐个解决后再合并到一起。Quicksort和merge sort都属于此类。Binary search不属于，因为二分后并没有再合并，而是直接选定一半接着二分，它是decrease and conquer. 微妙的差别。

6.总结：

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$nlog^2n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.
Heapsort	$nlogn$	$nlogn$	$nlogn$	1	No	Selection	
Merge sort	$nlogn$	$nlogn$	$nlogn$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarian's Algorithm)
Timsort	n	$nlogn$	$nlogn$	n	Yes	Insertion & Merging	Makes $n-1$ comparisons when the data is already sorted or reverse sorted.
Quicksort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.
Shellsort	$nlogn$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.

稳定性上需要注意的有：merge is but quick is not. Insertion is but shell is not. Selection is not. In short, those allowing exchanges with far-away elements are not stable.

7.笔试：

Q: 假设需要对存储开销 1GB (GigaBytes) 的数据进行排序，但主存储器（RAM）当前可用的存储空间只有 100MB (MegaBytes)。针对这种情况，（B）排序算法是最适合的。

A：堆排序　B：**归并排序**　C：快速排序　D：插入排序

2.basic ds

Monday, March 18, 2024 16:51

1.basic concepts:

1.栈 stack: addition and removal takes place at the same end, referred to as the 'top', the opposite end thus referred to as the 'base'. LIFO principle.

2.队列 queue: front vs rear. FIFO.

3.linear structure 线性表：python的list更接近数组。

线性表是一种逻辑结构，描述了元素按线性顺序排列的规则。常见的线性表存储方式有数组和链表。逻辑结构一般有四种：集合、线性结构、树形结构、图状结构。

数组是一种连续存储结构，它将线性表的元素按照一定的顺序依次存储在内存中的连续地址空间上。数组需要预先分配一定的内存空间，每个元素占用相同大小的内存空间，并可以通过索引来进行快速访问和操作元素。访问元素的时间复杂度为O(1)，因为可以直接计算元素的内存地址。然而，插入和删除元素的时间平均为O(n)，因为需要移动其他元素来保持连续存储的特性。

链表是一种存储结构，它是线性表的链式存储方式。链表通过节点的相互链接来实现元素的存储。每个节点包含元素本身以及指向下一个节点的指针。链表的插入和删除操作时间复杂度为O(1)，因为只需要调整节点的指针。然而，访问元素的时间平均为O(n)，因为必须从头节点开始遍历链表直到找到目标元素。

存储结构除了数组（顺序存储）、链表（链式存储）还有散列表（散列存储）和不重要的其它。

2.栈的应用：

1.符号检查器：成双成对的的符号是否正确配对，如{{([][])}()}。不能直接数数量，可以根据符号配对时后进先出的特点考虑用栈。

```
if symbol in "([{":
    s.append(symbol) # push(symbol)
else:
    top = s.pop()
    if not matches(top, symbol):
        balanced = False
```

2.中缀转后缀：shunting yard 调度场算法。主要是符号的处理。加减乘除括号在此没有任何数学意义，只有优先级的区别。括号优先级最高，因此一碰到右括号就开始出栈；其余时候输入新的运算符就开始判定上一个运算符是否优先级更高，如果更高则立即结算并再往前判定，否则上一个不动。

1. Create an empty stack called `opstack` for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method `split`.
3. Scan the token list from left to right.
    - If the token is an operand, append it to the end of the output list.
    - If the token is a left parenthesis, push it on the `opstack`.
    - If the token is a right parenthesis, pop the `opstack` until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
    - If the token is an operator, *, /, +, or -, push it on the `opstack`. However, first remove any operators already on the `opstack` that have higher or equal precedence and append them to the output list.
4. When the input expression has been completely processed, check the `opstack`. Any operators still on the stack can be removed and appended to the end of the output list.

```
def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = [] # Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in
"0123456789":
            postfixList.append(token)
        elif token == '(':
            #opStack.push(token)
            opStack.append(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
        else:
            #while (not opStack.is_empty()) and
(prec[opStack.peek()] >= prec[token]):
            while opStack and (prec[opStack[-1]] >= prec[token]):
                postfixList.append(opStack.pop())
            #opStack.push(token)
            opStack.append(token)

    #while not opStack.is_empty():
    while opStack:
        postfixList.append(opStack.pop())
    return " ".join(postfixList)
```

3.八皇后问题：将参数以数组形式压入栈中代替函数递归。

```python
def solve_n_queens(n):
    stack = []  # 用于保存状态的栈
    solutions = []  # 存储所有解决方案的列表

    stack.append((0, [-1] * n))  # 初始状态为第一行，所有列都未放置皇后

    while stack:
        row, queens = stack.pop()

        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, queens):  # 检查当前位置是否合法
                    new_queens = queens.copy()
                    new_queens[row] = col  # 在当前行放置皇后
                    stack.append((row + 1, new_queens))  # 推进到下一行

    return solutions
```

4.单调栈：以O(n)的效率处理序列内部单调性问题，包括找到某个数据点以前或以后大于或小于其的第一个点。下面为一递减栈：

```
for i in range(n):
  while stack and a[stack[-1]] < a[i]:
    a[stack.pop()] = i + 1
  stack.append(i)
#在原数据上改可以节省内存
while stack:
  a[stack.pop()] = 0
```

3.二分法+贪心算法：另一种思维，不是穷举可能的元素组合，而是穷举可能的最值。

农夫约翰是一个精明的会计师。他意识到自己可能没有足够的钱来维持农场的运转了。他计算出并记录下了接下来$N(1 \le N \le 100{,}000)$天里每天需要的开销。

约翰打算为连续的$M(1 \le M \le N)$个财政周期创建预算案，他把一个财政周期命名为fajo月。每个fajo月包含一天或连续的多天，每天被恰好包含在一个fajo月里。

约翰的目标是合理安排每个fajo月包含的天数，使得开销最多的fajo月的开销尽可能少。

先用二分法算出可能的最值，最大值取值范围的初始最小值是max(expenses)，初始最大值是sum(expenses)。对于二分法算出的每一个最值取值，都用贪心算法验证是否满足题意。那个left = mid + 1很关键，因为中点不满足的话最小值就不应该包括中点，否则有可能会卡在这个最小值一直循环。

```python
def min_max_fajo_expense(n, m, expenses):
    # Helper function to check if we can split the
    def can_split(max_expense):
        current_sum = 0
        num_fajo_months = 1
        for expense in expenses:
            if current_sum + expense > max_expense:
                num_fajo_months += 1
                current_sum = expense
                if num_fajo_months > m:
                    return False
            else:
                current_sum += expense
        return True

    left = max(expenses)
    right = sum(expenses)

    while left < right:
        mid = (left + right) // 2
        if can_split(mid):
            right = mid
        else:
            left = mid + 1

    return left
```

3.tree

Monday, March 25, 2024 23:16

0.basic concepts:

路径 Path：由边依次连接在一起的有序节点列表。比如，哺乳纲→食肉目→猫科→猫属→家猫就是一条路径。

子节点 Children：入边均来自于同一个节点的若干节点，称为这个节点的子节点。

父节点 Parent：一个节点是其所有出边连接节点的父节点。

兄弟节点 Sibling：具有同一父节点的节点之间为兄弟节点。

子树 Subtree：一个节点和其所有子孙节点，以及相关边的集合。

叶节点 Leaf Node：没有子节点的节点称为叶节点。

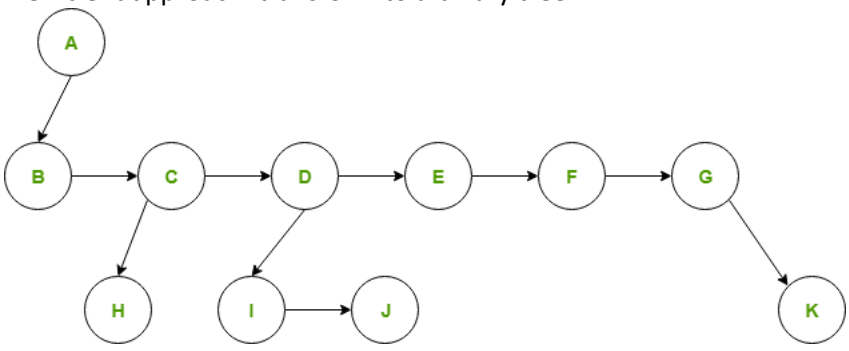
层级 Level：
从根节点开始到达一个节点的路径，所包含的边的数量，称为这个节点的层级。

1.construction:

1.dynamic arrays:

```
class Node:

    def __init__(self,data):
        self.data=data
        self.children=[]
```

2.efficient approach: transform to a binary tree.




```
class Node:
    def __init__(self, data):
        self.data = data
        self.firstChild = None
        self.nextSibling = None
```

3.condition of judgement: if node is None.

```python
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
```

2.some strategy and typical examples:

1.pluck the 'leaves' off the 'trunks': dirs and files are not of the same importance. Dirs are components of a skeleton tree while leaves are not.

```python
class dir:
    def __init__(self, dname):
        self.name = dname
        self.dirs = []
        self.files = []

    def getGraph(self):
        g = [self.name]
        for d in self.dirs:
            subg = d.getGraph()
            g.extend(["|      " + s for s in subg])
        for f in sorted(self.files):
            g.append(f)
        return g
```

2.parse a string: push (into the stack as a signal of the last kid of the former node.

3.special cases:

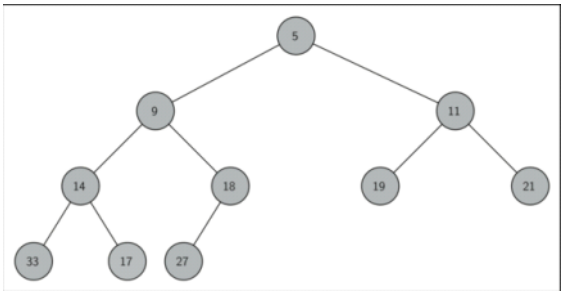
1.huffman code: to efficiently encode text into binary strings. 要构建一个最优的哈夫曼编码树，首先需要对给定的字符及其权值进行排序。然后，通过重复合并权值最小的两个节点（或子树），直到所有节点都合并为一棵树为止。为了提取出权值最小的两个节点，需要用heap实现的priority queue. heap的操作主要有三个：heapify一个列表（但是其中元素需要可比大小，因此定义类时加了__lt__这个特殊方法），heappop, heappush.

```python
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
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]
```

2.最小堆的实现：二叉堆的入队和出队只要O(logn).

1.堆的性质：有序性。以最小堆而言，任一父元素都不大于子元素。

2.完全二叉树：除了最底层，其他每一层的节点都是满的的二叉树。创建完全二叉树的目的是维持树的平衡，即尽可能让根节点的左右子树含有大致相等数量的节点。



3.完全二叉树的性质：可以用一个列表表示。在列表中处于位置 p 的节点来说，它的左子节点正好处于位置 2p；同理，右子节点处于位置 2p+1。若要找到树中任意节点的父节点，只需使用 Python 的整数除法即可。给定列表中位置 n 处的节点，其父节点的位置就是 n/2。当然，列表的0号位需要占位。

4.最小堆中插入元素：先加到列表末尾保证仍是完全二叉树，然后往上层层比较交

换放到正确位置。

5.最小堆中移除最小元素：移除索引为1的元素后，将最后一个元素补到1处保持完全，然后将其层层往下与两个子节点中较小的那个比较交换。

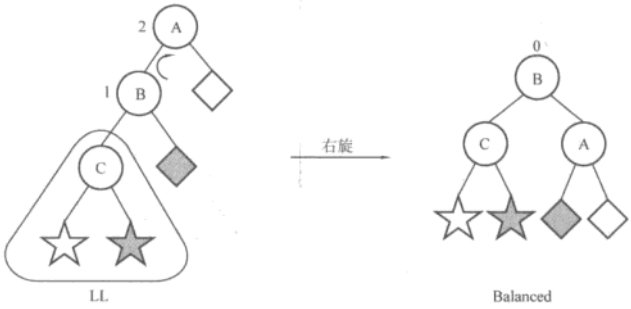
3.二叉搜索树(BST): 小于父节点的键都在左子树中，大于父节点的键则都在右子树中。可用于快速排序，O(nlogn). 但是要尽可能平衡。

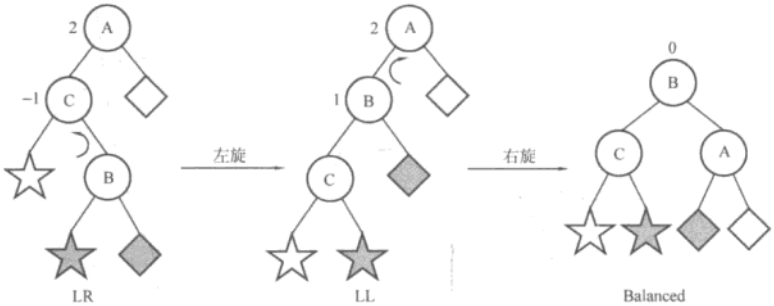
4.平衡二叉搜索树(AVL树): 通过在每个节点上维护一个平衡因子来实现平衡。平衡因子是指节点的左子树高度与右子树高度之差的绝对值。

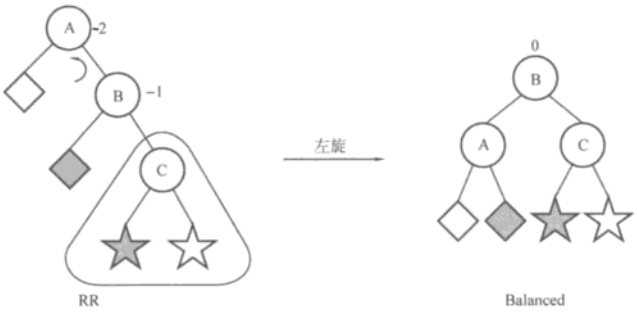
1.性质：设n层的AVL树至少有f(n)个节点，则f(n) = f(n-1) + f(n-2) + 1. 用lru_cache装饰器实现缓存。

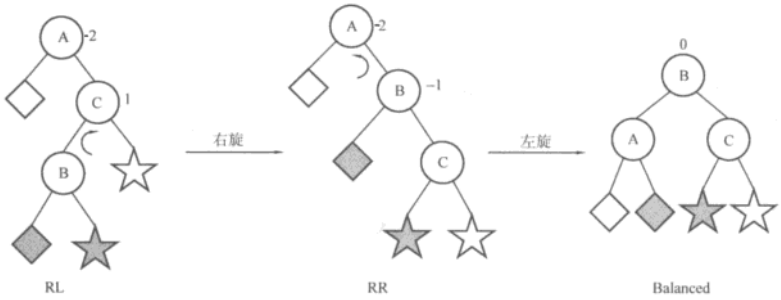
```
from functools import lru_cache


@lru_cache(maxsize=None)
def avl_min_nodes(n):
if n == 0:
  return 0
elif n == 1:
  return 1
else:
  return avl_min_nodes(n-1) + avl_min_nodes(n-2) + 1
```

2.旋转的原理：









3.删除某节点的原理：如果该节点没有两个子节点时直接补上来即可；否则，需要将右子节点中的最小值补到节点上来，再递归地删除右子节点中的最小值。

4.实现：

1.辅助函数：因为要实现递归且不想返回递归中途的任何值。用递归而非循环的原因是因为要记录经过的节点并在插入后更新它们的平衡因子。

```
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)
```

```
        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
```

4.other adts

Monday, May 13, 2024 21:24

1.散列表 hash table:

(1) **散列函数和散列地址**：在记录的存储位置p和其关键字 key 之间建立一个确定的对应关系 H，使p=H(key)，称这个对应关系H为散列函数，p为散列地址。

(2) **散列表**：一个有限连续的地址空间，用以存储按散列函数计算得到相应散列地址的数据记录。通常散列表的存储空间是一个一维数组，散列地址是数组的下标。

(3) **冲突和同义词**：对不同的关键字可能得到同一散列地址,即 key1≠key2,而 H(key1) = H(key2) 这种现象称为冲突。具有相同函数值的关键字对该散列函数来说称作同义词，key1与 key2 互称为同义词。

1.散列函数的构造方法：数字分析法、平方取中法、折叠法、除留余数法等。

2.处理冲突的方法：

1.开放地址法：冲突后用某种方法再算出另一个地址。具体有双散列法、线性探测法（按顺序往下找）、二次探测法（d_i =1^2, -1^2, 2^2,-2^2,3^2...）、伪随机探测法；

2.链地址法：将冲突的同义词放在同一个链表中，数组储存该链表的头指针。

2.并查集 disjoint set: 本质上是用树形结构表示一个集合，合并集合即合并树，只是用邻接表表示树形结构，每个节点存储父节点的指针。

1.基本的find and union:

```
def find(i):

    # If i is the parent of itself
    if (parent[i] == i):

        # Then i is the representative of
        # this set
        return i
    else:

        # Else if i is not the parent of
        # itself, then i is not the
        # representative of his set. So we
        # recursively call Find on its parent
        return find(parent[i])
```

```
def union(parent, rank, i, j):
    # Find the representatives
    # (or the root nodes) for the set
    # that includes i
    irep = find(parent, i)

    # And do the same for the set
    # that includes j
    jrep = find(parent, j)

    # Make the parent of i's representative
    # be j's representative effectively
    # moving all of i's set into j's set)

    parent[irep] = jrep
```

2.优化：

1.路径压缩：

```
def find(self, x):

    # Finds the representative of the set that x is an
ent of
    if (self.parent[x] != x):

        # if x is not the parent of itself
        # Then x is not the representative of its set
        self.parent[x] = self.find(self.parent[x])

        # so we recursively call Find on its parent
        # and move i's node directly under the
        # representative of this set

    return self.parent[x]
```

2.union by rank: rank指某集合树的高度。另取一个与parent数量相同的集合rank[]，全部取1，然后比较被合并的两集合的rank，将较小的作为较大的那个的子节点。如果相等，则无所谓，但是合并后rank会加1.

3.典型写法：

```
def find(x):
    if parent[x] != x: # 如果不是根结
        parent[x] = find(parent[x])
    return parent[x]

def union(x, y):
    parent[find(x)] = find(y)
```

3.进阶题：食物链。思路是用3n+1大小的表分别装某动物、它能吃的动物和捕食它的动物，因为每一个动物自身设定了三个集合，所以若每个动物都不同则有3n个集合。也因此合并部分有三个操作。

动物王国中有三类动物A,B,C，这三类动物的食物链构成了有趣的环形。A吃B，B吃C，C吃A。
现有N个动物，以1 - N编号。每个动物都是A,B,C中的一种，但是我们并不知道它到底是哪一种。
有人用两种说法对这N个动物所构成的食物链关系进行描述：
第一种说法是"1 X Y"，表示X和Y是同类。
第二种说法是"2 X Y"，表示X吃Y。
此人对N个动物，用上述两种说法，一句接一句地说出K句话，这K句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
1） 当前的话与前面的某些真的话冲突，就是假话；
2） 当前的话中X或Y比N大，就是假话；
3） 当前的话表示X吃X，就是假话。
你的任务是根据给定的N（1 <= N <= 50,000）和K句话（0 <= K <= 100,000），输出假话的总数。

```python
def find(x):     # 并查集查询
    if p[x] == x:
        return x
    else:
        p[x] = find(p[x])    # 父节点设为根节点。目的是
        return p[x]

n,k = map(int, input().split())

p = [0]*(3*n + 1)
for i in range(3*n+1):   #并查集初始化
    p[i] = i

ans = 0
for _ in range(k):
    a,x,y = map(int, input().split())
    if x>n or y>n:
        ans += 1; continue

    if a==1:
        if find(x+n)==find(y) or find(y+n)==find(x):
            ans += 1; continue
        # 合并
        p[find(x)] = find(y)
        p[find(x+n)] = find(y+n)
        p[find(x+2*n)] = find(y+2*n)
    else:
        if find(x)==find(y) or find(y+n)==find(x):
            ans += 1; continue
        p[find(x+n)] = find(y)
        p[find(y+2*n)] = find(x)
        p[find(x+2*n)] = find(y+n)
```

3.前缀树 trie tree: 储存一系列可能有相同前缀的字符串，如通讯录或词典。比哈希表更快。每个节点有两个属性：列表kids和布尔变量is_end，其中is_end指示从头到此是否是一个完整的词。

4.线段树 segment tree: 用于快速求解一个数表中某一段区间内的和。本质上是完全二叉树的妙用，将数表作为叶子，两两一组求和作为上一层，依此类推至最高层。

5.graph

Thursday, April 18, 2024 7:27

1.基本概念：

1.度：顶点的度是指和该顶点相连的边的条数。有向图中分为出度与入度。

2.权值：顶点和边可以具有的可量化的属性。

2.实现：标准方法是用类，出于方便可用字典套列表形式的邻接表。

3.bfs and dfs: 关键是加一个变量visited用以储存已经遍历过的变量。如果是复杂到需要用到类的题目，则改为在vertex类中加一个布尔变量。两个算法的时间复杂度都是O(V+E).

1.bfs: 注意把即将被遍历的节点也要添加进visited。如果需要追踪路径，则还要变量previous记录每个节点的前一个节点。

```
graph = {'A': ['B', 'C', 'E'],
         'B': ['A','D', 'E'],
         'C': ['A', 'F', 'G'],
         'D': ['B'],
         'E': ['A', 'B','D'],
         'F': ['C'],
         'G': ['C']}


def bfs(graph, initial):
    visited = []
    queue = [initial]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)
            neighbours = graph[node]

            for neighbour in neighbours:
                queue.append(neighbour)
    return visited
```

2.dfs: 入栈在前还是标记经历过在前无所谓。往往要写一个递归的函数，不然处理visited会比较棘手。

```
visited = []
stack = []
graph = ret_graph()
path = []


stack.append(start)
visited.append(start)
while stack:
    curr = stack.pop()
    path.append(curr)
    for neigh in graph[curr]:
        if neigh not in visited:
            visited.append(neigh)
            stack.append(neigh)
            if neigh == dest :
                print("FOUND:", neigh)
                print(path)
                sys.exit(0)
print("Not found")
print(path)
```

有时可以用启发式技术改良dfs的深入方向，例如骑士周游问题中按照再下一步的走法由少到多将下一步的方向排序按次深入，先快速减少搜索空间（Warnsdorff算法）。

另外，标准写法是变色，未历经时是白色，正在经历时是灰色，已历经是黑色。这种写法在判环时可以提高效率：如果碰见灰色说明有环；如果碰见黑色说明后续已经被之前的步骤判定为死胡同，可以不用再继续。如果只用是否历经的二分变量，则黑色部分还要再经历一次。

```python
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"
```

3.通用dfs, depth first forest: 记录每一节点的开始时间和结束时间，并且需要遍历图中所有节点以致可能产生不止一棵树。主要关注discovery_time和closing_time.

```python
def dfs_visit(self, start_vertex):
    start_vertex.color = "gray"
    self.time = self.time + 1       #记录算法的步骤
    start_vertex.discovery_time = self.time
    for next_vertex in start_vertex.get_neighbors():
        if next_vertex.color == "white":
            next_vertex.previous = start_vertex
            self.dfs_visit(next_vertex)     #深度优先
    start_vertex.color = "black"
    self.time = self.time + 1
    start_vertex.closing_time = self.time
```

4.拓扑排序:

1.深度优先森林算法:

(1) 对图 g 调用 dfs(g) 。之所以调用深度优先搜索函数，是因为要计算每一个顶点的结束时间。
(2) 基于结束时间，将顶点按照递减顺序存储在列表中。
(3) 将有序列表作为拓扑排序的结果返回。

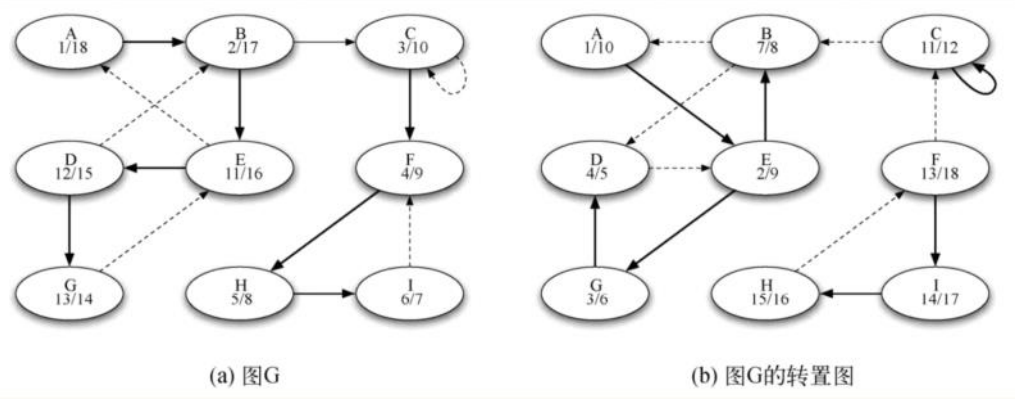
2.kahn算法：

1. 计算每个顶点的入度（Indegree），即指向该顶点的边的数量。
2. 初始化一个空的结果列表 `result` 和一个队列 `queue`。
3. 将所有入度为 0 的顶点加入队列 `queue`。
4. 当队列 `queue` 不为空时，执行以下步骤：
    - 从队列中取出一个顶点 `u`。
    - 将 `u` 添加到 `result` 列表中。
    - 对于顶点 `u` 的每个邻接顶点 `v`，减少 `v` 的入度值。
    - 如果顶点 `v` 的入度变为 0，则将 `v` 加入队列 `queue`。
5. 如果 `result` 列表的长度等于图中顶点的数量，则拓扑排序成功，返回结果列表 `result`；否则，图中存在环，无法进行拓扑排序。

```
# 计算每个顶点的入度
for u in graph:
    for v in graph[u]:
        indegree[v] += 1

# 将入度为 0 的顶点加入队列
for u in graph:
    if indegree[u] == 0:
        queue.append(u)

# 执行拓扑排序
while queue:
    u = queue.popleft()
    result.append(u)

    for v in graph[u]:
        indegree[v] -= 1
        if indegree[v] == 0:
            queue.append(v)
```

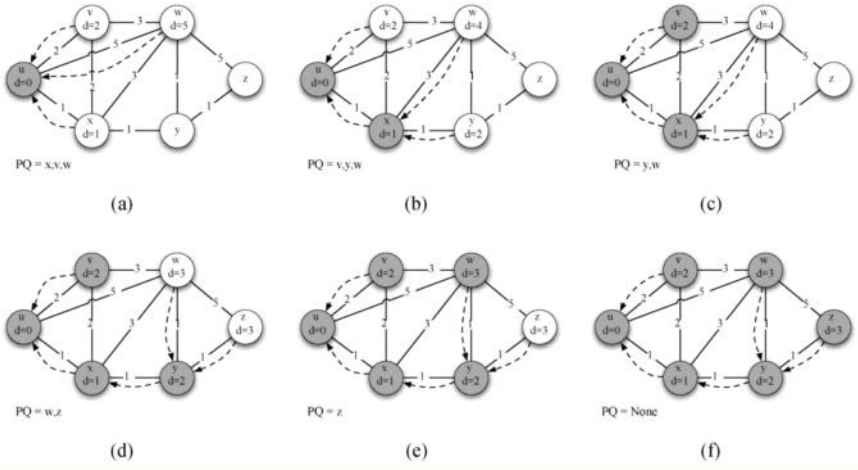
5.强连通分量：kosaraju算法：

(1) 对图G调用dfs，以计算每一个顶点的结束时间。
(2) 计算图G^T。
(3) 对图G^T调用dfs，但是在主循环中，按照结束时间的递减顺序访问顶点。
(4) 第3步得到的深度优先森林中的每一棵树都是一个强连通单元。输出每一棵树中的顶点的id。



(a) 图G　　(b) 图G的转置图

6.最短路径：dijkstra算法：只适用于所有边的权重均为正的情况。总时间复杂度为O((V+E) logV)，因为调整heap需要logV。在vertex中设立一个变量distance用以记录从起点到当前顶点的最小权重路径的总权重，其默认值应该很大；设立一个previous变量记录前驱顶点。先将起点的distance设为0，遍历其邻居，计算邻居distance = prev.distance + weight。若新的distance小于原来的，则更新邻居的distance and previous。每次选定邻居的顺序由以distance为权重的heap决定。



(a)　(b)　(c)

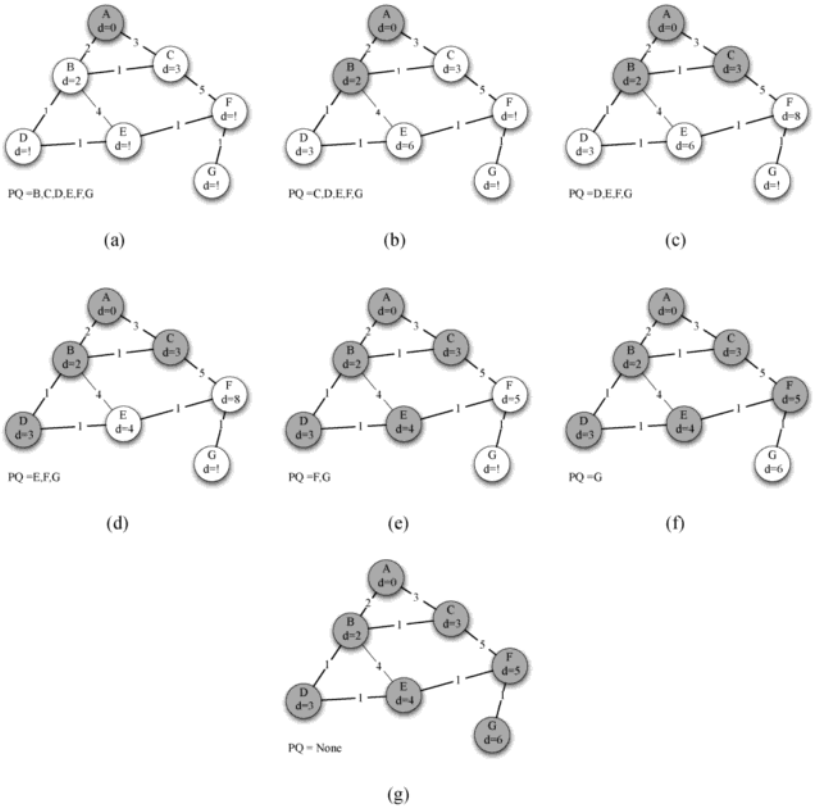
(d)　(e)　(f)

```
def dijkstra(graph, start):
    pq = []
    start.distance = 0
    heapq.heappush(pq, (0, start))
    visited = set()

    while pq:
        currentDist, currentVert = heapq.heappop(pq)
顶点的最短路径确定后（也就是这个顶点

队列中被弹出时），它的最短路径不会再改变。
        if currentVert in visited:
            continue
        visited.add(currentVert)

        for nextVert in currentVert.getConnections():
            newDist = currentDist +
currentVert.getWeight(nextVert)
            if newDist < nextVert.distance:
                nextVert.distance = newDist
                nextVert.pred = currentVert
                heapq.heappush(pq, (newDist, nextVert))
```

7.最小生成树 minimun spanning tree mst：

1.概念：边权和最小的生成树。生成树指包含所有顶点的树。mst的边数一定为n-1.

2.prim算法：从顶点出发，某种意义上来说是简化的dijkstra，只是距离的定义不一样。首先设立一个列表和一个heap分别储存已经弹出的节点和尚未弹出的节点，heap先储存所有的节点，distance默认无穷大。随便选定一初始节点并更新distance = 0，之后循环以下步骤：弹出最小值节点储存至列表，计算其邻居distance = weight，若新distance小于原来的则更新。最后按照previous建树。下图中距离的定义是错的！C到F的distance实际上都是1.



3.kruskal算法：从边出发。常用并查集，并查集可以高效地判断两个顶点是否已经连通。

1. 将图中的所有边按照权重从小到大进行排序。
2. 初始化一个空的边集，用于存储最小生成树的边。
3. 重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：
   - 选择排序后的边集中权重最小的边。
   - 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中。
4. 返回最小生成树的边集作为结果。

8.动态规划：本身是指一种将复杂问题化简为推导式，然后调用memo储存之前算过的值来减少计算量的思想，例如求斐波那契数。现在特指用数组储存结果来解决在限制之下的最优问题的方法。如“鸣人与佐助”中用一个数组专门储存到达各点时最多的查克拉数。再如下面这道“道路”：

第一行包含一个整数K, 0 <= K <= 10000, 代表Bob能够在他路上花费的最大的金币数。第二行包含整数N， 2 <= N <= 100, 指城市的数目。第三行包含整数R, 1 <= R <= 10000, 指路的数目.
接下来的R行，每行具体指定几个整数S, D, L 和 T来说明关于道路的一些情况，这些整数之间通过空格间隔:
S is 道路起始城市, 1 <= S <= N
D is 道路终点城市, 1 <= D <= N
L is 道路长度, 1 <= L <= 100
T is 通行费 (以金币数量形式度量), 0 <= T <=100
注意不同的道路可能有相同的起点和终点。

输出

输入结果应该只包括一行，即从城市1到城市N所需要的最小的路径长度（花费不能超过K个金币）。如果这样的路径不存在，结果应该输出-1。

第二个if非常重要，不能遗漏，虽然加入heap时已经筛选过一遍，但是新加入的可能比已在堆里的老的更优化，只有再筛选才能让老的被筛选出去。

```python
def find_shortest_path(K, N, R, roads):
    # Initialize dp array with infinity
    dp = [[float('inf')] * (K + 1) for _ in range(N + 1)]
    dp[1][0] = 0  # Starting city 1, with 0 cost, has 0 distance

    # Priority queue for Dijkstra's algorithm
    pq = [(0, 1, 0)]  # (current_distance, current_city, current_cost)

    # Adjacency list to store roads
    adj = [[] for _ in range(N + 1)]
    for s, d, l, t in roads:
        adj[s].append((d, l, t))

    while pq:
        current_distance, current_city, current_cost = heapq.heappop(pq)

        # If we reached the last city with a valid cost
        if current_city == N:
            return current_distance

        # If the current distance is not optimal, continue
        if current_distance > dp[current_city][current_cost]:
            continue

        # Explore neighbors
        for neighbor, length, toll in adj[current_city]:
            new_cost = current_cost + toll
            new_distance = current_distance + length
            if new_cost <= K and new_distance < dp[neighbor][new_cost]:
                dp[neighbor][new_cost] = new_distance
                heapq.heappush(pq, (new_distance, neighbor, new_cost))
```