算法

算法分析

Asymptotic notation 渐进记号

def 1. 如果存在正常数 n0 和 c，使得当 N >= n0 时 T(N) <= cf(N)，则记为 T(N) = O(f(N)) 大O是增长率上限

def 2. 如果存在正常数 n0 和 c，使得当 N >= n0 时 T(N) >= cg(N)，则记为 T(N) = Ω(f(N)) 大Ω是增长率下限

def 3. T(N) = θ(h(N)) 当且仅当 T(N) = O(h(N)) 且 T(N) = Ω(h(N))

def 4. 如果 T(N) = O(p(N)) 且 T(N) ≠ θ(p(N))，则 T(N) = o(p(N)) 排除了 大O 增长率中相等的可能性

常见增长率：O(2^N) > O(N^2) > O(NlogN) > O(N) > O(logN) > O(1)

典型的O(logN)型的算法：

1. 二分查找
2. 欧几里得算法（求最大公因数）
3. 取幂运算（Pow(X ,N) = Pow(X*X, N/2)

基本数据类型

列表

index：O(n)

search：O(n)

insert：O(1) 每次都在表头 head 处进行插入

delete：O(n) 需要查找删除的元素

双列表：

- 可以实现以倒序扫描列表
- 在数据结构中加上一个previous域即可

循环列表：让最后的节点反过来指向第一个节点。

栈

栈也是一个表，因此任何实现表的方法都能实现栈。

链表实现：

- 记录表头 head
- push 向表头插入，top 获取表头元素，pop 删除并返回表头元素
- 上述操作均花费常数时间
- 缺点是对于 malloc 和 free 的调用的开销是昂贵的，可以维持一个 freelist，删除的节点放到 freelist 中供下次使用 ?

数组实现：

- 限制是需要提前声明数组的大小
- 维持一个 topOfStack 游标，指向当前的栈顶元素，对于空栈是 -1
- 操作可以以常数时间进行

应用：

1. 平衡符号，检查括号开关正确
    1. 从空栈开始读到文件尾。如果读入开括号则push
    2. 如果读到闭括号，空栈时报错，否则出栈，出栈的符号不能配对时报错
    3. 文件尾时堆栈不为空保持哦
2. 计算后缀（逆波兰）表达式，e.g. 6 5 2 3 + 8 * + 3 + *
    1. 读到数字时push
    2. 读到运算符时pop出两个操作数，进行该运算符的计算，将计算结果push
    3. 栈中留下的最后一个数是结果
3. 中缀 -> 后缀转换
    1. 读到一个数的时候，立刻输出
    2. 遇到操作符 *，+ 和左括号时，push入栈
    3. 遇到右括号，pop直到弹出左括号，左括号不输出
    4. 如果遇到其它符号，则pop直到遇到优先级相同或更高的符号

队列

数组实现：

- 使用一个数组和两个游标：front, rear，记录队列size
- 采用循环数组，front 或 rear 到达数组尾端就绕回
    - 注意检测空队列
    - 入队的时候采用 read % size，出队的时候采用 front % size

应用：

- 不考虑任务删除的话，单任务打印机的处理过程是一个队列

树

基本知识

递归的定义：一棵树是一些节点的集合。这个集合可以是空集；若非空，则一棵树由称作根（root）的节点 r 以及 0 个或多个非空的（子树）T1，T2，…，Tk 组成，这些子树中每一棵的根都被来自 r 的一条有向边所连接。

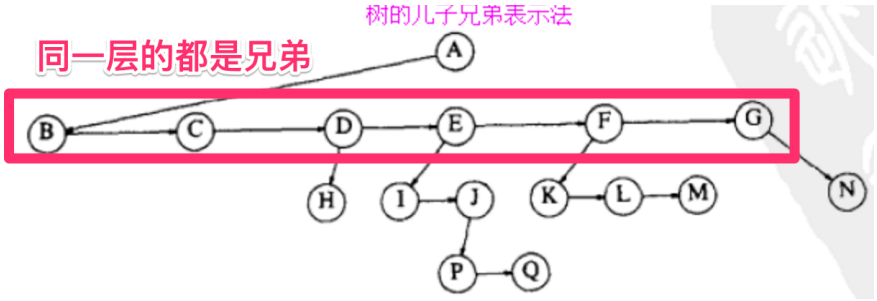
一棵树是 N 个节点和 N－1 条边的集合（root 没有向上的边）。

一条路径的长（length）是这条路径上的边的条数。

某个节点的深度是从根到该节点的唯一路径的长。
某个节点的高度是从该节点到一片叶节点的最长路径的长。一棵树的高等于它根节点的高。

对于每个节点有任意儿子数的树的实现：
first-child-next-sibling representation，将每个节点存储它的第一个儿子，和下一个兄弟。

```
class Node:
     def __init__(self, elem, firstChild, nextSibling):
     self.elem = elem
     self.firstChild = firstChild
     self.nextSibling = nextSibling
```



树的遍历：

- 先序遍历：先处理根节点再处理子节点
- 后序遍历：先处理子节点再处理根节点
- 中序遍历：（仅针对二叉树）左-中-右

二叉树
每个节点都不能有多于两个儿子

二叉树节点：

```
class TreeNode:
     def __init__(self, elem, left, right):
          self.elem = elem
          self.left = left
          self.right = right
```

应用：表达式树（expression tree）：
所有的操作数（Operand）都在叶节点上。中间节点全部是操作符（operator），之后的操作可以是：

- 中序遍历（带上左右括号）：得到中缀表达式
- 后序遍历：得到后缀表达式

二叉查找树
查找树的额外性质：

1. 左子树中的所有值均小于根节点X
2. 右子树中的所有值均大于根节点X

二叉查找树的平均深度是 O(logN)

```
class BST:
     class Node:
          def __inti__(self, elem, left, right):
               self.elem = elem
               self.left = left
               self.right = right
```

查找操作 find:

```
def find(node, x):
     if node == None:
          return None
     elif node.elem == x:
          return node # found
     elif node.elem < x:
          return find(node.left, x)
     else:     # node.elem > x
          return find(node.right, x)
```

返回最小元素节点 findMin:

```
def findMin(node):
     if not node:
          return None
     elif node.left:
          return findMin(node.left)
     else:
          return node
```

返回最大元素节点 findMax:

```
def findMax(node):
     if not node:
          return None
     elif node.right:
          return findMax(node.right)
     else:
          return node
```

插入 insert(x):

递归实现，理解为更新 node 节点：insert(root, x)

```
def insert(node, x):
     if not node:
          node = Node(x, None, None)
     elif x < node.elem:
          node.left = insert(node.left, x)
     else: # x > node.elem
          node.right = insert(node.right, x)
     return node
```

如果有插入重复元素的可能，可以考虑修改 Node 类的定义，添加一个 count 域来记录该元素在树中被插入过的个数，比反复插入新的节点要好（会增加树的深度）。

删除 delete(x)：
找到要删除的节点 node 后的可能：

- node 为叶节点，直接删除
- node 有一个子节点，将 node 的父节点指向 node 的子节点后，删除 node
- node 有两个子节点，将 node 的右子树中的最小值替代 node 的值，再删除具有最小值的节点，因为具有最小值的节点不可能再有左儿子，这一次删除会较简单。

```
def delete(node, x):
     if not node:
          error("cant find x")
     elif x < node.elem:
          node.left = delete(node.left, x)
     elfi x > node.elem:
          node.right = delete(node.right, x)
     else: # find x
          if not node.left and not node.right:    # leaf node
               node = None
          elif not node.left:    # only have right node
               node = node.right     # 原先的 node 对象引用计数归零，自动回收
          elif not node.right:     // only have left node
               node = node.left
          else:     # have two children
               minNode = findMin(node.right)  # 找出 node 右子树中的最小值节点
               node.elem = minNode.elem # 用右子树最小值替换 node 的值
               delete(node.right, minNode.elem) # 从右子树中删除最小值
     return node # 返回更新完毕的 node
```

注意：node 有两个子节点时使用右子树的最小值来替代 node 的值，这一做法有使得树左倾的趋势。更好的实践是随机的，或者选择左子树的最大值，或者选择右子树的最小值，来替代 node 的值。

懒惰删除 lazy deletion：
删除时仍将节点留在树中，而只是做一个删除的标记：

- 无重复元素的，可以修改 Node 类增加 deleted 布尔域，表明是否已不可用
- 有重复元素的，可以修改 Node 类增加 count 域，删除时减一
- 重新插入时，如果插入的位置是一个 lazy deleted 节点，直接复用该节点，避免了创建新节点对象的开销

平均情况分析：
当树均匀时 find, findMin, findMax, insert, delete 所有操作时间均为 O(logN)

二叉树的遍历应用：

1. 按顺序打印查找二叉树：中序遍历

```
def printTree(node):
     if node:
          printTree(node.left)
          print node.elem
          printTree(node.right)
```

2. 计算树的高度：先序遍历

```
def height(node):
     if not node:
          return -1
     else:
          return 1 + max(height(node.left), height(node.right))
```

AVL树

伸展树 splay tree

B树

---

k-d 树

用于多维范围查找，如平面上的点（x,y）。

2-d 查找树的性质是：在奇数层上的节点与子节点保持第一个关键字的相对顺序，偶数层上的节点和其子节点保持第二个关键字的相对顺序。

2-d 查找树平均高度位 O(logN)，最坏情形是 O(N)。

在 2-d 树上可以进行的几种查询：

- 两个关键字精确匹配
- 某个关键字匹配，另一个关键字在一个范围内
- 两个关键字均在某个范围内

都是属于（正交）范围查询。

从一组数据构建一棵理想平衡 2-d 树的时间是 O(NlogN)

之后实现查找的时间是 O(logN)

```
class Node:
     def __init__(self, data, left, right):
          self.data = data
          self.left = left
          self.right = right

def insert(item, root, level):
     if (root == None):
          newNode = Node()
          newNode.left = None
          newNode.right = None
```

```
            newNode.data[0] = item[0] # x
            newNode.data[1] = item[1] # y
      else:
            if (item[level] < root.data[level]):
                  insert(item, root.left, 1-level)
            else
                  insert(item, root.right, 1-level)
      return root

# find items:
# low[0] <= item[x] <= high[0]
# low[1] <= item[x] <= high[1]
def search(low, high, root, level):
      if root != None:
            if low[0] <= root.data[0] <= high[0] and
                  low[1] <= root.data[1] <= high[1]:
                  print root.data

            if low[level] <=  root.data[level]:
                  search(low, high, root.left, 1-level)

            if high[level] >= root.data[level]:
                  search(low, high, root.right, 1-level)
```

对于 k-维情况，通过每层上的关键字的循环，也可行。

哈希（散列）表

特性：以常数时间 O(1) 执行插入、删除、查找。
但是需要元素间排序信息的操作无法高效实现。如 findMin，findMax 需要线性时间 O(N)。
顺序打印难以实现，需要额外的空间和数据结构。

实现思想：

- 散列表的实现就是一个存储对象的具有固定大小的数组，记其大小为 tableSize
- 对象通过*散列函数*映射到 0~tableSize-1 中的某个数，存放在数组相应位置

散列函数

理想情况下应该运算简单并且保证任何两个不同的关键字映射到不同的单元，不过实际上不可避免冲突。我们希望寻找一个能在单元间均匀地分配对象的函数。

1.存储对象是整数：
合理的选择是：hash(key) = key % tableSize，并通常保证 tableSize 是一个素数

2.存储对象是字符串：

1. 把字符串的 ASCII 码加起来。但是如果 tableSize 很大，则散列函数的取值只能在 0~127*len(S) 之间，只占据了开头较小的一部分，分配不均匀。
2. 利用 Horner 法则的尝试：叠乘 32

```
def hash(s):
     hashVal = 0
     for ch in s:
          hashVal = (hashVal << 5) + ord(ch)
     return hashVal % tableSize
```

该方法的好处是乘以 32 用移位实现非常快速。问题是如果字符串长度过长的话，左移操作最终会发生溢出。

冲突解决

当两个元素计算得到相同的 hash 值，尝试插入到同一位置的时候，会发生哈希碰撞（hash collision）。两种解决办法：

1.分离链接法 separate chaining：

将散列到同一个值的元素保留到一个链表中。此时哈希数组中本身存储的不再是元素，而是一个表头节点：

- find：通过 hash 函数来确定获取表头，确定要查找哪一个链表，之后以通常的方式来遍历搜寻链表
- insert：通过 hash 函数来确定插入到哪个表，遍历以找到合适的插入位置
- delete：类似

NB：分离链接法每个表的位置不一定要采用链表，也可以使用二叉查找树，或者别的表结构。但是理想情况下，如果 tableSize 足够大并且 hash function 分配均匀，那么每个表都应很短，使用链表并不会太耗时。如果链表很长导致性能不佳，应该先考虑优化 hash function 本身，而不是直接采取更复杂的表结构。

2.开放定址法 open addressing hashing：

分离链接法的确定是分配新的节点空间较耗时，而且涉及到了另一种数据结构的实现。
在开放定址法中，如果有冲突发生，就选择另外的单元，直到找到可用的单元为止。因为所有的数据都要直接存入哈希数组内（比较分离链接法：哈希数组只存表头，元素每次有新分配的空间），因此需要的表比分离链接散列表要来的大。一般地，最终成功选中的位置是 hi(x) = (Hash(x) + F(i)) % tableSize，其中 i 是冲突次数， F(i) 是冲突解决方法，且 F(0)=0。

NB：对于开放定址法，标准的删除操作不能进行，否则后续的 find 操作会失效。只能进行 lazy deletion。

三种常见的冲突解决：z

1. 线性探测法，F 是 i 的线性函数，典型情况 F(i) = i。缺点：插入元素会形成一次聚集（primary clustering），散列到聚集区块中的任何元素都需要多次探测才能插入成功。
2. 平方探测法，可以用来解决一次聚集，典型情况 F(i) = i**2。NB：对于平方探测法，如果表有一半空（λ<0.5）且 tableSize 为素数，总能够插入一个新的元素。
3. 双散列，F(i) = i*hash2(x)。应用第二个散列函数 hash2，并且在 hash2(x), 2*hash2(x) 等处探测。一个好的 hash2 尝试是 hash2(x) = R - (x % R)，其中

R 为小于 tableSize 的素数。

平均情况分析：
定义装填因子（load factor）λ = n / tableSize：列表中存储元素的个数与列表大小的比值。
对于分离链接法，λ 可以大于 1，等同于每个链表的平均长度，执行一次成功查找的耗时为 O(1+λ/2)
对于开放定址法，λ 应该低于 0.5

其它散列表技术：

1. 再散列：当散列表填满到一定程度时，重新分配新空间，使用大小约为 2 倍的新表，并且采用新的 hash function。将原来的元素也重新填到新表中。
2. 可扩充散列：处理数据量太大，无法装进内存的情况。

优先队列（堆）
基本操作：

1. 插入 insert
2. 删除最小 deleteMin：找出、返回并删除队列中最小的元素

简单实现：
1.使用普通链表

- 插入：始终在表头以 O(1) 插入
- 删除：每次遍历以找到最小元素 O(N) <- 不理想

2.使用二叉查找树

- 插入：O(logN)
- 删除：O(logN) <- 不理想：反复删除最小元素会使树向右倾斜，可以考虑使用平衡树，但是实现过于复杂

二叉堆实现 binary heap
一般讨论堆（heap）的时候，默认指代的是二叉堆（binary heap）。堆本身也是一种树结构，具有以下的特殊性质：

1. 结构性质：堆是完全填满的二叉树，只有底层可以有例外，底层元素从左到右填入（堆=完全二叉树 complete binary tree)。
    - 如果事先可以预估堆的大小，可以采用树的数组实现（1-indexed）。
2. 堆序性质：如果要找出最小元（minHeap），则最小元应该位于根节点上。任意节点的元素应该小于其子节点的元素。NB：与BST不同，两个兄弟节点之间的顺序可以是任意的。

操作实现：

1. swim(i) 上浮操作：将 i 位置的元素上浮到正确位置，持续与父节点比较，若父节点更大，则将父节点元素替换到当前位置。
2. sink(i) 下沉操作：将 i 位置的元素下沉到正确位置，在每个位置 i 都找出两个子节点中较小的，替换到 i 位置，直到某个位置比两个子节点都小。
3. 插入：为保证结构性质，尝试在下一个空闲位置（size+1）放入，之后将插入元素上浮到正确位置。
    - 在 0 位置放入一个可能的最小整数值，作为 sentinel，可以保证插入的最小元素在 1 的位置一定成功插入。
    - 插入操作的最坏情况时间是 O(logN)，但是平均情况只需要常数次比较。
4. 删除最小元：移除第一个最小元素，将最后一个元素放入空闲位置，再通过下沉操作来使堆重新有序化

```
class MinPQ:
    def __init__(self, MAX):
        self.MAX = MAX
        self.heap = collections.deque(maxlen=MAX) # python 中的定长数组
        self.heap[0] = -sys.minint # 树的数组实现舍弃0位置不用，从1开始索引
        self.size = 0

    def insert(self, x):
        assert self.size < self.max
        self.size += 1
        self.heap[self.size] = x
        self.__swim(self.size) # 最后一个位置处的元素顺序可能不对，上浮到正确位置

    def deleteMin(self):
        assert self.size > 0
        minElement = self.heap[1]
        self.heap[1] = self.heap[self.size]
        self.size -= 1
        self.__sink(1) # 将最后一个元素放到第一个位置，顺序不对，下沉到正确位置
        return minElement

    def __swim(self, i):
        "假设除i以外的位置都是有序的，现将i上浮到正确位置"
        assert 1 <= i <= self.size, "invalid position"
        assert self.heap[i] < self.heap[i/2], "can't swim" # i 位置元素比父节点小，否则无法上浮

        elem = self.heap[i]
        while self.heap[i/2] > elem:
            self.heap[i] = self.heap[i/2] # 原来在上面的值拉到下面来
            i /= 2
            # inv: i >= 1
            self.heap[i] = x

    def __sink(self, i):
```

```
        "假设除i以外的位置都是有序的，现将i下沉到正确位置"
        assert i <= i <= self.size / 2, "invalid position"  #  i 位置至少有一个子节点，否则位于叶子节点，不需要下沉
        assert self.heap[i] > self.heap[i*2] or (i*2+1 <= self.size and self.heap[i] > self.heap[i*2+1]), "can't sink" # i 位置元素至少比一个子节点大，否则无法下沉

        elem = self.heap[i] # 记录elem，现在 i 位置有个空位
        while i * 2 <= self.size:
            # 找出 i 位置的子节点中较小的
            child = i * 2
            if child != self.size and self.heap[child+1] < self.heap[child]: # 确保位置 i 存在两个子节点
                child += 1

            # inv: 现在 child 指向较小的那个子节点
            if elem < self.heap[child]: # 比当前位置i的较小的子节点更小，则位置i是合适的
                break
            else:
                self.heap[i] = self.heap[child] # 现在 child 位置有个空位
                    i = child

                    self.heap[i] = elem

    def buildHeap(self, A):
        "从无需的数组构建 minHeap"
        self.heap = collections.deque(maxlen=len(A)+1)
        for (e,i) in enumerate(A):
            self.heap[i+1] = e

            for i in range(N/2, -1, -1):
                self.__sink(i)
```

优先队列应用

1.选择问题：输入 N 个元素和一个整数 k，找出第 k 个最小的元素。

1. 使用堆排序，读入一个数组，然后对数组应用 buildHeap，之后执行 k 次 deleteMin
2. online 算法，需要使用 maxHeap，读入 k 个元素，之后每读入一个元素都与根节点元素比较，如果比根节点元素小，则替换根节点元素，并 sink 到正确位置

---

d-堆

左式堆

斜堆

排序算法
假设数组只包含整数，整个排序工作可以在内存中完成，元素个数相对较小（少于100万个）。

选择排序
对于 i=0~N-2 趟，选择排序保证 0~i-1 位置是已排序状态，i 趟完成后，A[i] = min(A[i..N-1])

```
def selectionSort(A):
    N = len(A)
    for i in range(N-1): # [0, N-2]
        minIdx = -1, minVal = sys.maxint
        for j in range(i, N): # j=[i,N-1]
            if A[j] < minVal:
                minVal = A[j]
                    minIdx = j
                    tmp = A[i]
                    A[i] = A[minIdx]
                    A[minIdx] = tmp
```

时间复杂度总是 O(N^2)，对于已排序的数组也不会更快，因为内循环必须走完 [i,N) 才能决定最小数。
比插入排序慢。插入排序对于有序度较高的数组，内循环执行次数很少。

不稳定：5 8 5 2 9，选择最小的元素 2，交换第一个元素 5，此时两个 5 的相对顺序改变了

插入排序
插入排序由 N-1 趟排序组成。对于 i=1~N-1 趟，插入排序保证 0~i-1 位置是已排序状态。

```
def insertionSort(A):
    N = len(A)
    for i in range(1, N): # [1,N-1]
        # inv: [0,i-1] 已经有序，现在要确定 A[i] 的位置
        tmp = A[i]

        j = i # j 指向下一个可能的 A[i] 的正确位置
        while j > 0 and A[j-1] > tmp:
            A[j] = A[j-1]
                j -= 1

        # inv: j == 0 or (A[j-1] <= tmp and A[j]=A[j+1] > tmp)
        A[j] = tmp
```

时间分析：

- 平均情况 O(N^2)
- 最坏情况 O(N^2)
- 最好情况 O(N)：预先排序，内部的 while 循环立即判断为 false，不执行内循环

- 对于几乎被排序的数组，插入排序运行很快，因为内部 while 循环执行的很少

插入排序是稳定的。

希尔排序

排序思想：

- 使用一个增量序列（increment sequence）h1,h2,h3…ht，只要满足 h1=1即可
- 在使用增量 hk 的一趟排序完成后，对于每一个 i，均有 A[i] <= A[i+hk]，称数组是 hk-有序的
- 一种流行（但不好的）增量序列选择是：ht=N/2, hk=(hk+1)/2

```
def shellSort(A):
    N = len(A)
    inc = N / 2
    while (inc > 0):
        # 使用插入排序达成 inc-sorted
        for i in range(inc, N):
            tmp = A[i]
            j = i
            while j >= inc and A[j-inc] > tmp:
                A[j] = A[j-1]
                j -= 1

                A[j] = tmp
        inc /= 2
```

在增量选择理想的情况下，可以突破 O(N^2) 时间复杂度。

希尔排序不是稳定的。

堆排序

堆排序是不稳定的。

归并排序

经典的分治（divide-and-conquer）策略，将问题分成一些小的问题然后递归求解。而治的阶段则将分的阶段得到的各个答案修补到一起。

归并排序的过程是先分后治：

- 分：递归地将数组分为前半部分和后半部分，直到每个部分只有一个元素，则这一部分自然有序
- 治：将两个已经有序的数组放到第三个数组中，仍保持有序
  1. A，B是已经有序的数组，C是空的数组，采用三个游标，a，b，c
  2. A[a] 和 B[b] 中的较小者被放到 C[c] 中，相关游标前进
  3. 当两个输入表中的一个用尽时，则将另一个表中剩余部分拷贝到 C 中

```
def mergeSort(A):
```

```
tmp = [None for i in range(len(A))]
def sort(start, end):
     # sort A[start,end]
     if start < end:
          center = (start + end) / 2
          sort(start, center) # 1st part: A[start, center]
          sort(center+1, end)     # 2nd part: A[center+1, end]
          merge(start, center+1, end)
     else:
          pass # do nothing

def merge(LStart, RStart, end):
     lpos = LStart, rpos = RStart
     lend = RStart - 1, rend = end
     tpos = lpos

     while lpos <= lend and rpos <= rend:
          if A[lpos] < A[rpos]:
               tmp[tpos] = A[lpos]
               tpos += 1
               lpos += 1
          elif A[lpos] >= A[rpos]:
               tmp[tpos] = A[rpos]
               tpos += 1
               rpos += 1

     while lpos <= lend:
          tmp[tpos] = A[lpos]
          tpos += 1
          lpos += 1

     while rpos <= rend:
          tmp[tpos] = A[rpos]
          tpos += 1
          rpos += 1

     for i in range(LStart, end+1): //[LStart, end]
          A[i] = tmp[i]

sort(0, len(A)-1)
```

分析：

- 运行时间是 O(NlogN)
- 仍然难以用于主存排序，主要问题是两个排序的表需要O(N)的临时内存开销

稳定的排序算法。合并过程的时候，确保遇到两个相等元素的时候，把处在前面序列的元素保存在结果序列的前面，这就保证了稳定性。

快速排序

也是分治算法，将数组 S 排序由以下步骤完成：

1. 如果 S 中元素个数是 0 或 1，返回
2. 取 S 中的任一元素 v，作为枢纽元素（pivot）
3. 将 S 分为两个部分，所有比 v 小的元素归在一个部分 S1，所有比 v 大的元素归在另一个部分 S2
4. 返回 {quicksort(S1), v, quicisort(S2)}

枢纽元的选取：

- 理想的情况是选取 S 中所有元素的中值，即一半元素划分到 S1，另一半元素划分到 S2
- 较好的实践是 **三数中值法（Median-of-Three partitioning）**
    - 选取左端、右端、中心三个位置上的元素的中值作为枢纽元

分割策略：

1. 将枢纽元与最后的元素交换，使得枢纽元离开要被分割的数据段
2. 游标 i 指向第一个元素，j 指向倒数第二个元素（枢纽元的前一个）
3. 右移 i，直到遇到大于 v 的元素
4. 左移 j，直到遇到小于 v 的元素
5. 交换 i，j 指向的元素，重复 2-5 直到 i，j相遇

其他细节：

- 移动 i，j 时，如果遇到和 v 相同的元素，正确的做法是 i，j 都停止，并且进行交换，这有助于建立平衡的分割
- 设置小数组阀值（N <= 20），小数组使用插入排序有更好的性能，避免了过多的递归开销，也回避了一些特殊情况（如数组只有2个元素，取不出三数中值）
- 取三数中值的时候，对 left，mid，right 进行排序，这样 left 和 right 的值一开始就处于正确的分割，可以将 i 初始化到第二个元素，j 初始化到倒数第三个元素

```
def quickSort(A):
     cutoff = 20

     def median3(left, right):
          center = (left + right) / 2
          if A[left] > A[center]:
               tmp = A[left]; A[left] = A[center]; A[center] = tmp
          if A[left] > A[right]:
               tmp = A[left]; A[left] = A[right]; A[right] = tmp
          if A[center] > A[right]:
               tmp = A[center]; A[center] = A[right]; A[right] = tmp

          // inv: A[left] <= A[center] <= A[right]
          tmp = A[center]; A[center] = A[right-1]; A[right-1] = tmp
          return A[right-1]

     def sort(left, right):
          if right - left + 1 <= cutoff:
```

```
            insertionSort(A, left, right)
        else:
            pivot = median3(left, right)
            i = left; j = right - 2

            while i < j:
                while A[i] < pivot:
                    i += 1
                while A[j] > pivot:
                    j -= 1
                if i < j:
                    tmp = A[i]; A[i] = A[j]; A[j] = tmp

            // 把枢纽元移回中间
            tmp = A[right-1]; A[right-1] = A[i]; A[i] = tmp
            sort(left, i-1)
            sort(i+1, right)

    sort(0, len(A)-1)
```

快速排序的时间复杂度是 O(NlogN)

快速排序是不稳定的，不稳定性发生在中枢元素和a[j]交换的时刻。

桶排序

如果输入数据 A1,A2,…,An 只由大小小于 M 的正整数组成，可以使用一个大小为 M 的 count 数组，初始化为 0。当读取 A[i] 时，count[Ai] 加一。在所有的数据都读入之后，扫描数组 count，打印出排序后的表。

算法用时 O(M+N)，如果 M 为 O(N)，则总量为 O(N)。

桶排序可以实现为稳定的排序：

基数排序 radix sort

是桶排序的扩展，牺牲一定的运行时间来减少桶排序的空间占用。一般来说，对于 0 ~ (B^k - 1) 之间的数（基数＝B，最大数字长度为k），占用空间为O(B*n)，时间复杂度为O(k*n)，p趟每趟处理n个数。当k（数字位数）较小时，有可能优于O(nlogn)。

```
// A: the array of numbers
// B: the base
def radixSort(A, B):
    maxNum = max(A)
    k = math.ceil(math.log(maxNum, B) // 找出最大数字的位数 k
    bucket = [[] for i in range(k)]

    for i in range(1, k+1): // 每轮处理一个数位
        for num in A:
            digit = [num % (B ** (i + 1))] / (B ** i)
            bucket[digit].append(digit)
        del A[:] // 清除列表 A
        for arr in bucket:
```

```
            for num in arr:
                A.append(num)
        bucket = [[] for i in range(k)] // 清空 bucket 为下一个数位准
```

稳定的排序算法。

外部排序

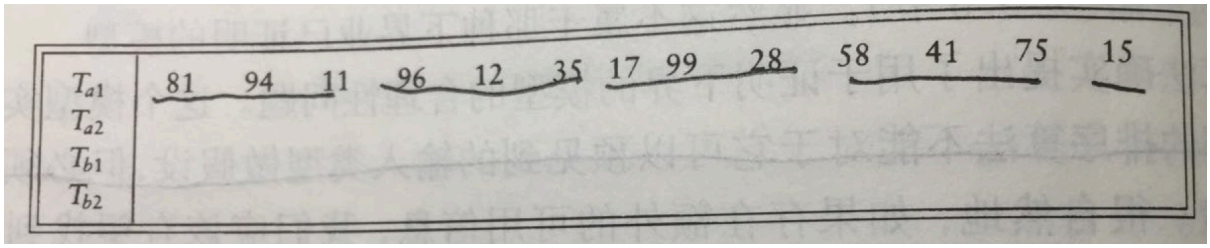
模型：

内部排序算法依赖于内存可直接寻址的事实，即有能力在常数时间内完成对任意 A[i] 和 A[j] 的比较。如果输入数据在磁带上，元素只能被顺序访问，因为需要把磁带转到正确的位置，只有连续的顺序才能有效访问数据。磁带可能是最受限制的存储媒体（编程实际中可以使用文件输入流 fileInputStream 模拟磁带）。假设需要三个磁带来排序，两个执行有效的排序，第三个继续简化的工作。如果只有一个磁带可用，则不得不说：任何算法那都需要 Ω(N^2) 次磁带访问。

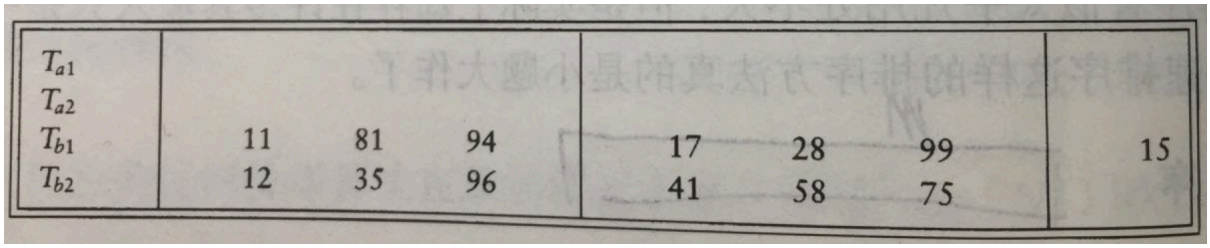
简单算法：

基本外部排序使用归并排序中 merge 阶段的思想。考虑使用 Ta1,Ta2,Tb1,Tb2 四盘磁带，两盘做输入，两盘做输出。数据最初在 Ta1 上，并且内存一次可以容纳 M 个记录。外部排序步骤：

1. 从输入磁带读入 M 个记录
2. 在内部将这些记录排序
3. 将排序过的记录交替写到 Tb1 或 Tb2 上，把每一次内存中排过序的 M 个记录叫做一个顺串（run）
4. 做完之后，倒回所有磁带



如果 M=3，顺串构造之后磁带状态：

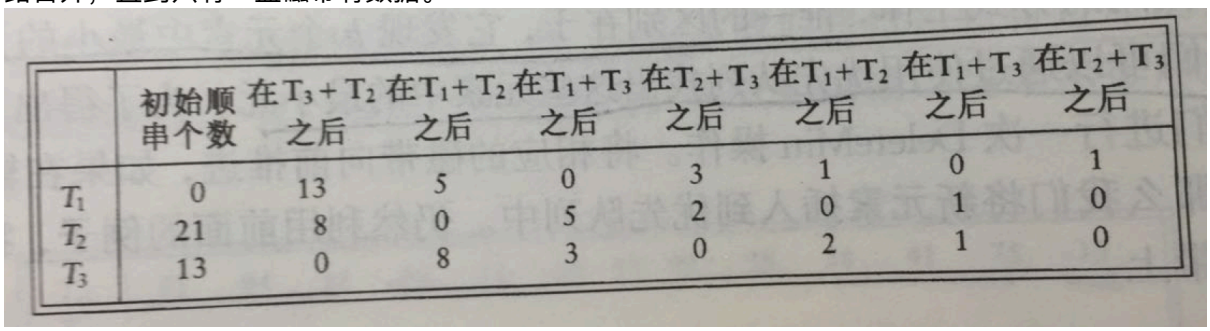


现在将 Tb1 和 Tb2 的第一个顺串取出并将两者合并，把结果写到 Ta1 上，得到一个两倍长的顺串（合并的过程：每次从两个顺串开头各读一个记录，比较大小，将小的输出，小记录的磁带前进到下个位置）。交替进行，直到这两个顺串有一个为空，此时将另一个顺串全部输出。

接下来重复过程，以 a 作为输入，b 作为输出，再次合并，直到最终完成。如果最终顺串的长度为 N 个记录，则需要的躺数是 ceiling(log(N/M))。

改进1：多路合并
如果有额外的磁带（2k盘），就可以将基本的（2-路）合并扩展为 k-路 合并。对于 k 路合并的时候，需要发现 k 个元素中的最小元素，可以通过使用优先队列来找出这些元素中的最小元素，此时需要的趟数是 ceiling(logk(N/M))。

改进2：多相合并
2k 盘磁带的量太大，可以只使用 k+1 盘磁带就完成任务。每次合并的时候，将 k 有数据的磁带作为输入，进行 k 路合并，输出到当前空白的磁带上，直到 k 盘输入磁带中有一盘耗尽。这时候将这盘磁带逆转，作为空白输出磁带，将其余 k 盘有记录的磁带作为输入磁带，继续进行 k 路合并，直到只有一盘磁带有数据。



	初始顺串个数	在T_3+T_2之后	在T_1+T_2之后	在T_1+T_3之后	在T_2+T_3之后	在T_1+T_2之后	在T_1+T_3之后	在T_2+T_3之后
T_1	0	13	5	0	3	1	0	1
T_2	21	8	0	5	2	0	1	0
T_3	13	0	8	3	0	2	1	0

此时顺串最初的分配很重要，如果顺串的总个数是一个斐波那契数列 Fn，则分配的最好办法是分裂成两个斐波那契数列 Fn-1 和 Fn-2，否则的话要将顺串的个数用一些哑顺串补足为斐波那契数列。

不相交集/查并集（union find）

图论算法
定义

- 图 G=(V,E)
- V 为顶点集合
- E 为边集合
- 有时候边具有权重（weight/cost）

路径

顶点序列：w1,w2,w3,…,wn 使得 (wi, wi+1) ∈ E。路径的长（length）是该路径上的边数。
简单路径是指其上所有顶点都是互异的，但第一个顶点和最后一个顶点可能相同。

有向图中的圈（cycle）是满足 w1=wn 且长至少为 1 的一条路径。如果该路径是简单路径，那么这个圈就是简单圈。对于无向图，要求边是互异的。有向无圈图也简称为 DAG。

连通性

如果无向图中从每一个顶点到每个其他顶点都存在一条路径，则称该无向图是*连通的*。
若有向图中每一个顶点都每个其他顶点都存在一条路径，则称该有向图是*强连通的*。
若有向图不是强连通，但是其基础图（underlying grpah），即去掉边上的方向所形成的图是连通的，则称该有向图是*弱联通的*。

完全图

每一对顶点间都存在一条边的图。

图的表示

1. 邻接矩阵（adjacency matrix）表示法

   使用二维数组，对于每条边 (u,v)，置 A[u][v]=1，否则数组元素为 0。
   如果边有权重，就置 A[u][v] 等于权重。而使用很大或很小的值表示边不存在。

   优点：操作简单，存取快速 O(1)
   缺点：空间需求大 θ(|V|^2)，若图是稠密的，|E|= θ(|V|^2)，则邻接矩阵法是合适的，否则代价太大。

2. 邻接表（adjacency list）表示法

   对于每一个顶点 u，用一个表存放所有邻接顶点 v，来表示边 u->v.
   空间需求是 O(|E|+|V|)。如果边有权重，可以存储在表的节点单元中。
   邻接表适合稀疏图，是表示图的标准方法。
   对于无向图，(u,v) 即出现在 u 的邻接表中，也出现在 v 的邻接表中，空间使用基本是**双倍**的。

   实现顶点名字（字符串）到内部编号的处理，用一个散列表：key=名字，value=编号

拓扑排序

对有向无圈图的一种排序，使得如果存在一条从 vi 到 vj 的路径，那么在排序中 vj 出现在 vi 的后面。

算法1：

先找出任意一个没有入边的顶点，然后输出该顶点，再将它和它的出边从图中删除。然后对图的剩余部分重复上述过程。
定义**入度**（indegree）：v 的入度是边(u,v) 的条数，即有多少条边进入 v。

以下代码可以生成 indegrees 数组，记录每个节点的入度：

```
def findIndegrees(vertices):
     indegrees = [0 for v in range(len(vertices))]
     for vList in vertices:
          for u in vList:
               # edge: (v,u)
               indegrees[u] += 1
```

```
    return indegrees
```

如果已经有了 indegrees 数组，可以用如下代码实现算法1

```
def topsort(indegrees, vertices):
    for i in range(len(vertices)):
        v = findNewVertexOfIndegreeZero()
        if v == None:
            return # 存在环
        else:
            print v # 输出 v
            for w in vertices[v]:
                # remove v
                # remove edge (v,w)
                indegrees[w] -= 1
```

findNewVertexOfIndegreeZero 需要 O(|V|)，算法总运行时间为 O(|V|^2)

算法1的改进：

在寻找入度为 0 的节点的时候，每次都进行 O(N) 的扫描是低效的，因为只有在执行 indegrees[w] -= 1 的时候会减少入度。如果每次在减少后将入度变为 0 的顶点放入队列中，每次都从队列中出队即可。

```
def topsort(vertices, indegrees):
    # use a queue to store all vertices with 0 indegree
    zeroDeg = collections.deque(maxlen=len(vertices))

    for (v,indeg) in enumerate(indegrees):
        if indeg == 0:
            zeroDeg.append(v)

    while !zeroDeg.isEmpty:
        v = zeroDeg.popLeft()
        print v # 输出 v

        for w in vertices[v]:
            # remove v, edge (v,w)
            indegrees[w] -= 1
            if indegrees[w] == 0:
                zeroDeg.append(w)

    assert sum(indegrees) == 0 # 若还有顶点有入度，则说明有环
```

改进之后的算法所用时间为 O(|E|+|V|)， while 中的 for 循环对每条边最多执行一次，耗时 O(|E|)，一开始初始化 zeroDeg 对所有顶点遍历一次，耗时 O(|V|)

应用：

- 课程排序（先修课程->高级课程）

最短路径算法

与边 (vi,vj) 相关联的是该边的值 $c_{i,j}$
有权路径长是一条路径上每条边的权重之和
无权图的路径长（unweighted path length）是路径上的边数
对于有负值边的图，如果存在**负值圈（negative-cost cycle）**，则最短路径问题变得不确定，因为在负值圈中可以滞留任意长。在没有负值圈的时候，从 s 到 s 的最短路径为 0。

单源最短路径问题

给定图 G=(V,E) 和一个特定顶点 s，找出从 s 到 G 中的每一个其他顶点的最短路径。

应用：

- 寻找计算机网络中的最低通信成本
- 航线图找最佳路线

考虑问题的四种形态：

1. 无权最短路径，以 O(|E|+|V|)解决 （有权最短路径的特殊情况，所有权重相同）
2. 无负边的有权最短路径，O(|E|log|V|)
3. 有负边的有权最短路径，O(|E|·|V|)
4. 特殊情形下的有权无圈图，线性时间

1.无权最短路径：
采用广度优先搜索（BFS）。使用三个薄记数组：

1. visited：搜索过程中，某节点是否已访问
2. distance：s 到该节点的最短 distance
3. previous：s 到该节点的最短路径中，当前节点的前一个节点下标

visited 数组可以去除，只要检查 previous 来看是否已访问过

```
def unweightedSingleSource(vertices, s):
     visited = [false for i in range(len(vertices))]
     distance = [None for i in range(len(vertices))]
     previous = [None for i in range(len(vertices))]

     queue = collection.deque()
     queue.append(s)
     distance[s] = 0
     previous[s] = s

     while !queue.isEmpty:
          v = queue.popLeft # dequeue
          visited[v] = true

          for w in vertices[v]: # (v,w)
               if !visited[w]:
                    queue.append(w)
                    distance[w] = distance[v] + 1
```

```
                    previous[w] = v

    return (distance, previous)
```

运行时间是 O(|E|+|V|),

2.无负边有权最短路径 Dijkstra 算法:
属于贪心算法（greedy）
记录三个数组:

1. visited: 节点是否已访问
2. distance: 使用已访问顶点作为中间顶点从 s 到 v 的最短路径长
3. previous: 引起 d_v 变化的最后的顶点

步骤

1. 选择一个顶点 v，是所有未访问的节点中具有最小的 d_v 值的。
2. 标注 visited[v] = true
3. 更新所有与 v 连接的顶点 w 的 $d_w = \min(d_w, d_v + c_{v,w})$，若采用 d 作为中间节点来到达 w 可以更近的话，就采用。
4. 1-3步每次完成后的 distance 数组，都是当前仅仅使用**已访问**节点所能走出的最短路径。

```
def Dijkstra(vertices, s):
    previous = [None for i in rnage(len(vertices))]
    distance = [sys.maxint for i in range(len(vertices))] // 利用所有已访问过的节点所能走出的最短路径
    distance[s] = 0

    unvisitedPQ = PriQueue()

    for v in range(len(vertices)):
        unvisitedPQ.insert(v, distance[v])

    # start main body of dijkstra algorithm
    while !unvisitedPQ.isEmpty:
        # 还有节点没有访问
        v = unvisitedPQ.delMin() # 从小顶堆中找出具有最小 d 的未访问节点

        for w in vertices(v): # 检查从 v 出发的每一条边（其实只要检查从 v 出发，另一个节点还没访问过的边）
            if (distance[w] > distance[v] + cost(v,w)):
                previous[w] = v
                distance[w] = distance[v] + cost(v,w)
                unvisitedPQ.decreaseKeyTo(w, distance[w]) # 下调优先队列中 w 的权重

    return (distance, previous)
```

如果边中有负值，Dijkstra 算法将会失效!

运行时间

delMin 操作共执行 |V| 次（最坏）
decreaseKey 操作共执行 |E| 次（最坏）
总时间为 $O(|V|T_{dm} + |E|T_{dk})$
如果使用优先队列（小顶堆）的话，时间是 O(|V|log|V|+|E|log|V|)

其它细节

1. 如何实现 O(logN) 的 decreaseKey 操作?

   实现的困难在于，需要 decreaseKey for a value 的时候，不知道这个 value 在优先队列的内部结构（数组）中的下标位置。解决的办法可以是在优先队列的内部维护一个 hashmap(value -> index)，在优先队列的各个操作中都维护这个 hashmap（耗费常熟时间），这样在 decreaseKey 的时候可以立即找到 value 的小标位置，之后调用 swim 上浮操作只需耗时 O(logN).

2. 如果不采用 decreaseKey 操作? 可以将每次 distance 减少成功的顶点直接用新的 key 重新插入，不管原来的旧的 key。我们通过维护一个 visited 数组，每次 delMin 之后，检查一下这个顶点是否已经访问。某个节点第一次被 delMin 之后，distance 就已经是最短的了，之后再被 delMin 出来的话就忽略。

贪婪算法

分治算法

动态规划

随机算法

回溯算法

其他

完美算法:
仅需常量空间，并以线性时间运行的 online 算法

二分查找法:

```
// A 为有序数组
// 循环实现
def binarySearch(A, x):
     lo = 0; hi = len(A) - 1
```

```
     while lo <= hi:
          mid = (lo + hi) / 2
          if A[mid] < x:
               lo = mid + 1
          elif A[mid] > x:
               hi = mid - 1
          else:
               return mid // find x
     return -1 // didn't finds

// 递归实现
def binarySearch(A, lo, hi, x):
     mid = (lo + hi) / 2
     if A[mid] < x:
          return binarySearch(A, mid+1, hi, x)
     elif A[mid] > x:
          return binarySearch(A, lo, mid-1, x)
     else:
          return mid
```

联机算法（online）：can process its input piece-by-piece in a serial fashion, the order of the input is fed to the algorithm, without having the entire input from the start.
脱机算法（offline）：given the whole problem data from the beginning then required to output an answer.