**考试前先看看各题目考察的算法，看看有没有可复用的模块。**

**预估一下各题目的难度，挑软柿子捏。**

**抄代码宜慢不宜快。**

**仔细读题！审错题还得返工。**

**预估数据大小，进而调整算法的选取。**

**函数：**

ord()：a->97,A->65

any()：全假返回假，否则返回真。

all()：全真返回真，否则返回假。

**并查集：**

**class** UnionFind:

**def** \_\_init\_\_(self,size):

        self.p=list(range(size))

        self.s=[1]\*size

**def** find(self,x):

        r=x

        while self.p[r]!=r:

            r=self.p[r]

        while self.p[x]!=r:

            next\_p=self.p[x]

            self.p[x]=r

            x=next\_p

        return r

**def** union(self,x,y):

        rx=self.find(x)

        ry=self.find(y)

        if rx==ry:

            return

        if self.s[rx]>=self.s[ry]:

            self.p[ry]=rx

            self.s[rx]+=self.s[ry]

        else:

            self.p[rx]=ry

            self.s[ry]+=self.s[rx]

**def** connected(self,x,y):

        return self.find(x)==self.find(y)

**def** get\_parent(self):

        return self.p

**def** get\_size(self,x):

        return self.s[self.find(x)]

**heap：**

class MinHeap:

def \_\_init\_\_(self):

self.heap = []

def \_parent(self, i):

return (i-1) // 2

def \_left\_child(self, i):

return 2\*i + 1

def \_right\_child(self, i):

return 2\*i + 2

def \_swap(self, i, j):

self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

def \_sift\_up(self, i):

while i > 0 and self.heap[i] < self.heap[self.\_parent(i)]:

parent = self.\_parent(i)

self.\_swap(i, parent)

i = parent

def \_sift\_down(self, i):

min\_index = i

n = len(self.heap)

while True:

left = self.\_left\_child(i)

if left < n and self.heap[left] < self.heap[min\_index]:

min\_index = left

right = self.\_right\_child(i)

if right < n and self.heap[right] < self.heap[min\_index]:

min\_index = right

if i != min\_index:

self.\_swap(i, min\_index)

i = min\_index

else:

break

def push(self, value):

self.heap.append(value)

self.\_sift\_up(len(self.heap)-1)

def pop(self):

if not self.heap:

return None

min\_val = self.heap[0]

last = self.heap.pop()

if self.heap:

self.heap[0] = last

self.\_sift\_down(0)

return min\_val

def peek(self):

return self.heap[0] if self.heap else None

def size(self):

return len(self.heap)

def build\_heap(self, arr):

self.heap = arr.copy()

n = len(self.heap)

for i in range(n//2-1, -1, -1):

self.\_sift\_down(i)

**二分查找：**

def check\_func(x): return x>=x\_0 其中x\_0是取值范围为[a,b]的整数。

def binarysearch(left, right, check\_func) -> int:

while left < right:

mid = (left + right) // 2

if check\_func(mid):

right = mid

else:

left = mid + 1

return left

特别地，假如想在列表里插值，可以使用如下配件：

**def** insertindex(l,x):

**def** check\_func(i):

        if i==len(l):

            return False

        return x<=l[i]

    return binarysearch(0,len(l),check\_func)

**dfs：尝试对现有数据结构本身作变换！**

class Solution:

def exist(self, board: List[List[str]], word: str) -> bool:

# 跟岛屿数量有点像

# 枚举 i=0,1,2,…,m−1 和 j=0,1,2,…,n−1，以 (i,j) 为起点开始搜索。

# 同时，我们还需要知道当前匹配到了 word 的第几个字母，所以还需要一个参数 k。

# 定义 dfs(i,j,k) 表示当前在 board[i][j] 这个格子，要匹配 word[k]，返回在这个状态下最终能否匹配成功（搜索成功）。

# 递归过程中，为了避免重复访问同一个格子，可以用 vis 数组标记。更简单的做法是，直接修改 board[i][j]，将其置为空（或者 0），返回 false 前再恢复成原来的值（恢复现场）。注意返回 true 的时候就不用恢复现场了，因为已经成功搜到 word 了。

m, n = len(board), len(board[0])

def dfs(i: int, j: int, k: int) -> bool:

if board[i][j] != word[k]: # 当前匹配失败，就直接退出了，不会进一步递归

return False

if k == len(word) - 1: # 边界条件，k成功匹配到最后一位时，表示匹配成功！

return True

board[i][j] = '' # 标记访问过

for x, y in (i, j - 1), (i, j + 1), (i - 1, j), (i + 1, j): # 相邻格子

# 边界满足，且子递归四个方向任一满足就返回True

if 0 <= x < m and 0 <= y < n and dfs(x, y, k + 1):

return True # 搜到了！

board[i][j] = word[k] # 恢复现场

# 四个方向子问题都不满足，则当前满足也不行，也要返回 False

return False # 没搜到

# 不同起点开始搜寻，任一满足即可

return any(dfs(i, j, 0) for i in range(m) for j in range(n))

**bfs：尝试对现有数据结构本身作变换！**

class Solution:

def nearestExit(self, maze: List[List[str]], entrance: List[int]) -> int:

m, n = len(maze), len(maze[0])

# 上下左右四个相邻坐标对应的行列变化量

dx = [1, 0, -1, 0]

dy = [0, 1, 0, -1]

# 入口加入队列并修改为墙

q = deque([(entrance[0], entrance[1], 0)])

maze[entrance[0]][entrance[1]] = '+'

while q:

cx, cy, d = q.popleft()

# 遍历四个方向相邻坐标

for k in range(4):

nx = cx + dx[k]

ny = cy + dy[k]

if 0 <= nx < m and 0 <= ny < n and maze[nx][ny] == '.':

# 新坐标合法且不为墙

if nx == 0 or nx == m - 1 or ny == 0 or ny == n - 1:

# 新坐标为出口，返回距离作为答案

return d + 1

# 新坐标为空格子且不为出口，修改为墙并加入队列

maze[nx][ny] = '+'

q.append((nx, ny, d + 1))

# 不存在到出口的路径，返回 -1

return -1