栈

波兰表达式求值（前缀）

```
def PN(expr: list[str]) -> str:
    stack = []
    for item in expr[::-1]:
        if item in "+-*/":
            stack.append(str(eval(stack.pop() + item + stack.pop())))
        else:
            stack.append(item)
    return stack[-1]
```

逆波兰表达式求值（后缀）

```
def RPN(expr: list[str]) -> str:
    stack = []
    for item in expr:
        if item in "+-*/":
            a, b = stack.pop(), stack.pop()
            stack.append(str(eval(b + item + a)))
        else:
            stack.append(item)
    return stack[-1]
```

Shunting yard （中缀转后缀）

```
def shunting_yard(expr: str) -> list[str]:
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    stack = []
    postfix = []
    num = ""

    for char in expr:
        if char.isnumeric() or char == '.':
            num += char

        else:
            if num:
                stack.append(num)
                num = ""

            if char == '(':
                postfix.append(char)

            elif char == ')':
                while postfix[-1] != '(':
                    stack.append(postfix.pop())
                postfix.pop()

            elif char in "+-*/":
```

```
                while postfix and postfix[-1] != '(' and precedence[postfix[-1]]
>= precedence[char]:
                    stack.append(postfix.pop())
                postfix.append(char)

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

    return stack
```

后缀转前缀（逆波兰转波兰）

```
def RPN_to_PN(expr: list) -> list:
    stack = []

    for item in expr:
        if item in "+-*/":
            stack.append(stack.pop() + stack.pop() + [item])

        else:
            stack.append([item])

    return stack[0][::-1]
```

合法出栈序列

```
def is_valid_pop_seq(push_seq: list, pop_seq: list) -> bool:
    if len(push_seq) != len(pop_seq):
        return False

    stack = []
    push_stack = push_seq[::-1]

    for item in pop_seq:
        while push_stack and (not stack or stack[-1] != item):
            stack.append(push_stack.pop())

        if stack and stack[-1] == item:
            stack.pop()
        else:
            return False

    return True
```

单调栈

```
def find_right_lteq(arr: list) -> list:
    mono_stack = []
    result = []

    for idx, val in reversed(list(enumerate(arr))):  # 找右边的，所以reverse
```

```
        while mono_stack and mono_stack[-1][1] > val:  # 找第一个小于等于的，所以大于则弹出
            mono_stack.pop()

        if mono_stack:
            result.append(mono_stack[-1][0])
        else:
            result.append(len(arr))  # 右边没有小于等于的，则返回None，或为方便返回len(arr)

        mono_stack.append((idx, val))

    result.reverse()  # reverse

    return result


def find_left_gteq(arr: list) -> list:
    mono_stack = []
    result = []

    for idx, val in enumerate(arr):  # 找左边的，不用reverse

        while mono_stack and mono_stack[-1][1] < val:  # 找第一个大于等于的，所以小于则弹出
            mono_stack.pop()

        if mono_stack:
            result.append(mono_stack[-1][0])
        else:
            result.append(-1)  # 左边没有大于等于的，则返回None，或为方便返回-1

        mono_stack.append((idx, val))

    return result


# 奶牛排队
arr = []
N = int(input())
for _ in range(N):
    arr.append(int(input()))
right_lt = find_right_lteq(arr)
left_gt = find_left_gteq(arr)

max_len = 0
for start in range(N):
    for end in range(right_lt[start] - 1, start + max_len - 1, -1):
        if left_gt[end] < start:
            max_len = end - start + 1
            break

print(max_len)
```

树

卡特兰数$(Catalan\ number)$，其公式为：

$$f(n)=\frac{C_{2n}^{n}}{n+1}$$

二叉树

```
from collections import deque

class Node:
    def __init__(self, value='', left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.value)

    def __bool__(self):
        return True

    def depth(self):
        l = self.left.depth() if self.left else 0
        r = self.right.depth() if self.right else 0
        return 1 + max(l, r)

    def height(self):
        return self.depth() - 1

    def count_leaves(self):
        if not self.left and not self.right:
            return 1
        l = self.left.count_leaves() if self.left else 0
        r = self.right.count_leaves() if self.right else 0
        return l + r

    def preorder(self) -> list:
        return [self.value] + (self.left.preorder() if self.left else []) +
(self.right.preorder() if self.right else [])

    def inorder(self) -> list:
        return (self.left.inorder() if self.left else []) + [self.value] +
(self.right.inorder() if self.right else [])

    def postorder(self) -> list:
        return (self.left.postorder() if self.left else []) +
(self.right.postorder() if self.right else []) + [self.value]

    def levelorder(self) -> list:
        queue = deque([self])
        result = []
```

```
        while queue:
            node = queue.popleft()
            result.append(node.value)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        return result
```

二叉树建树

```
def parse_RPN(expr: list, is_nums=lambda x: x.isnumeric(), is_opers=lambda x: x
in "+-*/") -> Node:  # 后缀表达式建树
    stack = []

    for item in expr:
        if is_nums(item):
            stack.append(Node(item))

        elif is_opers(item):
            r = stack.pop()
            l = stack.pop()
            stack.append(Node(item, l, r))

    return stack.pop()

def parse_in_post(inorder: str, postorder: str) -> Node:  # 中后序遍历建树
    root_val = postorder[-1]
    idx = inorder.find(root_val)

    if idx > 0:
        left = parse_in_post(inorder[:idx], postorder[:idx])
    else:
        left = None

    if idx < (l := len(inorder)) - 1:
        right = parse_in_post(inorder[idx+1:], postorder[idx:l-1])
    else:
        right = None

    return Node(root_val, left, right)
```

多叉树

```
class MultNode:
    def __init__(self, value='', children=None):  # 缺省值不要设置成可变对象空列表
        self.value = value
        self.children = children if children else []

    def __str__(self):
        return str(self.value)
```

```
    def __bool__(self):
        return True

    def add_child(self, child):
        self.children.append(child)

    def preorder(self) -> list:
        return [self.value] + ([child.preorder() for child in self.children] if
self.children else [])

    def postorder(self) -> list:
        return ([child.postorder() for child in self.children] if self.children
else '') + [self.value]
```

多叉树建树

```
def parse_brackets(expr: str) -> MultNode:  # 括号嵌套树
    stack = []
    node = MultNode(expr[0])

    for char in expr[1:]:
        if char == '(':
            child = MultNode()
            node.add_child(child)
            stack.append(node)
            node = child

        elif char == ')':
            node = stack.pop()

        elif char == ',':
            node = stack.pop()
            child = MultNode()
            node.add_child(child)
            stack.append(node)
            node = child

        else:
            node.value = char

    return node
```

多叉树与二叉树转换

```
def mult_to_bi(mult_root: MultNode) -> Node:
    root = Node(mult_root.value)

    brother = None
    for mult_child in mult_root.children[::-1]:
        child = mult_to_bi(mult_child)
        child.right = brother
        brother = child

    root.left = brother
```

```
    return root


def bi_to_mult(root: Node) -> MultNode:
    mult_root = MultNode(root.value)

    mult_root.add_child((bi_to_mult(root.left) if root.left else None))

    if root.left:
        child = root.left
        while child.right:
            mult_root.add_child(bi_to_mult(child.right))
            child = child.right

    return mult_root
```

二叉搜索树

```
class BSTNode(Node):
    def __init__(self, value: int, left=None, right=None):
        super().__init__('', left, right)
        self.value = value

    def insert(self, value):
        if value < self.value:
            if self.left:
                self.left.insert(value)
            else:
                self.left = BSTNode(value)
        elif value > self.value:
            if self.right:
                self.right.insert(value)
            else:
                self.right = BSTNode(value)
```

二叉搜索树建树

```
def parse_preorder(expr: list) -> BSTNode:  # 根据前序遍历建树
    if (l := len(expr)) == 1:
        return BSTNode(expr[0])

    root_value = expr[0]

    for i in range(1, l):
        if expr[i] > root_value:
            break

    else:
        return BSTNode(root_value, parse_preorder(expr[1:]), None)
    if i == 1:
        return BSTNode(root_value, None, parse_preorder(expr[1:]))
    else:
```

```
        return BSTNode(root_value, parse_preorder(expr[1:i]),
parse_preorder(expr[i:]))
```

并查集

```
class DisjointSet:
    def __init__(self, size: int):
        self.size = size
        self.parent_list = [i for i in range(self.size)]

    def find(self, x):
        if self.parent_list[x] != x:
            self.parent_list[x] = self.find(self.parent_list[x])
        return self.parent_list[x]

    def is_linked(self, x, y):
        return self.find(x) == self.find(y)

    def union(self, x, y):
        if not self.is_linked(x, y):
            self.parent_list[self.find(y)] = self.find(x)

    def variety(self):
        count = 0
        for x in range(self.size):
            count += int(self.parent_list[x] == x)
        return count
```

图

```

class Vertex:
    def __init__(self, key=None):
        self.key = key
        self.neighbors: dict[Vertex] = {}
        self.prev = None
        self.dist = float('inf')
        self.in_deg = 0

    def __str__(self):
        return str(self.key)

    def __bool__(self):
        return True

    def add_neighbor(self, nbr: 'Vertex', weight):
        self.neighbors[nbr] = weight

    def get_weight(self, nbr: 'Vertex'):
        return self.neighbors[nbr]
```

```python
    def is_connected(self, nbr: 'Vertex', weight):
        return nbr in self.neighbors

    def degree(self):  # 有向图中为入边
        return len(self.neighbors)


class Graph:
    def __init__(self):
        self.vert_dict: dict[str, Vertex] = {}

    def __str__(self):
        return str(self.vert_dict.keys())

    def __iter__(self):
        return iter(self.vert_dict.values())

    def add_vertex(self, key):
        self.vert_dict[key] = Vertex(key)
        return self.vert_dict[key]

    def get_vertex(self, key):
        return self.vert_dict[key]

    def add_edge(self, key1, key2, weight):  # 此处未考虑输入数据有重边
        if key1 in self.vert_dict:
            vert1 = self.get_vertex(key1)
        else:
            vert1 = self.add_vertex(key1)

        if key2 in self.vert_dict:
            vert2 = self.get_vertex(key2)
        else:
            vert2 = self.add_vertex(key2)

        vert1.add_neighbor(vert2, weight)
        vert2.add_neighbor(vert1, weight)

    def add_directed_edge(self, key1, key2, weight):  # 此处未考虑输入数据有重边
        if key1 in self.vert_dict:
            vert1 = self.get_vertex(key1)
        else:
            vert1 = self.add_vertex(key1)

        if key2 in self.vert_dict:
            vert2 = self.get_vertex(key2)
        else:
            vert2 = self.add_vertex(key2)

        vert1.add_neighbor(vert2, weight)
        vert2.in_deg += 1

    def size(self):
        return len(self.vert_dict)
```

```
    def flush(self):
        for vert in self.vert_dict.values():
            vert.dist = float('inf')
            vert.prev = None

    def transpose(self):
        t_graph = Graph()
        for key in self.vert_dict:
            t_graph.add_vertex(key)
        for vert in self:
            for nbr in vert.neighbors:
                t_graph.add_directed_edge(nbr.key, vert.key, vert.neighbors[nbr])

        return t_graph


def prim(graph: Graph, start):
    graph.flush()  # 若dist, prev未更新

    from heapq import heappop, heappush, heapify

    key = start  # 注意start应必须在图内，否则会崩溃
    hp = [(0, key)]  # 注意heap插入Vertex有可能无法比较
    visited = set()
    heapify(hp)
    min_weight_sum = 0

    while hp:
        weight, key = heappop(hp)
        if key in visited:
            continue

        visited.add(key)
        min_weight_sum += weight

        for nbr, weight in graph.get_vertex(key).neighbors.items():
            if nbr.key not in visited and weight < nbr.dist:
                heappush(hp, (weight, nbr.key))
                nbr.prev = graph.get_vertex(key)  # 记录路径

    if len(visited) == graph.size():  # 判断是否完全遍历
        return min_weight_sum
    else:
        return -1


def dijkstra(graph: Graph, start, end):
    if start not in graph.vert_dict:  # 若start未在图内
        return -1

    graph.flush()  # 若dist，prev未更新
    from heapq import heappop, heappush, heapify

    hp = [(0, start)]
    visited = set()
    graph.get_vertex(start).dist = 0
```

```
    while hp:
        dist, key = heappop(hp)
        if key in visited:
            continue
        visited.add(key)

        if key == end:
            break

        vert = graph.get_vertex(key)
        for nbr, weight in vert.neighbors.items():
            if nbr.key not in visited and dist + weight < nbr.dist:
                heappush(hp, (dist + weight, nbr.key))
                nbr.dist = dist + weight
                nbr.prev = vert  # 记录路径

    else:
        return -1
    return graph.get_vertex(end).dist


def is_connected_has_loop(graph: Graph):  # 无向图是否连通，是否有环。请注意若输入数据有重边，则一定有环。
    visited = set()
    looped = False

    def dfs(vert: Vertex, prev):
        nonlocal visited, looped

        visited.add(vert.key)
        for nbr in vert.neighbors:
            if nbr.key == prev:  # 无向图，注意判断是否是父节点
                pass
            elif nbr.key in visited:
                looped = True
            else:
                dfs(nbr, vert.key)

    for vert in graph:
        visited = set()

        dfs(vert, None)

        if len(visited) == graph.size() or looped:
            break

    return (len(visited) == graph.size(), looped)


def topo_sort(graph: Graph):  # 有向图优先拓扑排序，可以对有向图是否有环判断，无论是否连通
    from heapq import heappop, heappush, heapify

    in_dic = {}  # 请注意输入数据是否会出现重边，否则in_deg会多次+1而导致错误
    for vert in graph:
        in_dic[vert.key] = vert.in_deg
```

```
    heap = []
    for vert in graph:
        if in_dic[vert.key] == 0:
            heappush(heap, vert.key)
    heapify(heap)

    result = []
    while heap:
        key = heappop(heap)
        result.append(key)

        for nbr in graph.get_vertex(key).neighbors:
            in_dic[nbr.key] -= 1
            if in_dic[nbr.key] == 0:
                heappush(heap, nbr.key)

    if len(result) == graph.size():
        return result
    else:
        return False


def kosaraju(graph: Graph):

    def dfs1(vert: Vertex):
        nonlocal visited, stack

        visited.add(vert.key)

        for nbr in vert.neighbors:
            if nbr.key not in visited:
                dfs1(nbr)

        stack.append(vert.key)

    def dfs2(vert: Vertex):
        nonlocal visited, ecc

        visited.add(vert.key)
        ecc.append(vert.key)

        for nbr in vert.neighbors:
            if nbr.key not in visited:
                dfs2(nbr)

    visited = set()
    stack = []
    for vert in graph:
        if vert.key not in visited:
            dfs1(vert)

    t_graph = graph.transpose()

    visited = set()
    result = []
```

```
    for key in stack[::-1]:
        if key not in visited:
            ecc = []
            dfs2(t_graph.get_vertex(key))
            result.append(ecc)

    return result
```

Kruskal 算法

```
def kruskal(edges: list, vert_list: list):
    d_s = DisjointSet(vert_list)
    edges.sort()
    weight = 0
    for w, x, y in edges:
        if not d_s.linked(x, y):
            weight += w
            d_s.union(x, y)
    return weight
```

最大连通域面积

```
mat = [[]]
di = [-1, -1, -1, 0, 0, 1, 1, 1]; dj = [-1, 0, 1, -1, 1, -1, 0, 1]

def dfs(i, j):
    global mat
    mat[i][j] = '.'
    area = 1
    for k in range(8):
        if mat[i + di[k]][j + dj[k]] == 'W':
            area += dfs(i + di[k], j + dj[k])
    return area


T = int(input())
for test_case in range(T):
    N, M = map(int, input().split())
    mat = [['.' for j in range(M+2)]]
    for i in range(N):
        mat.append(['.']+list(input())+['.'])
    mat.append(['.' for j in range(M+2)])
    maxa = 0
    for i in range(1, N+1):
        for j in range(1, M+1):
            if mat[i][j] == 'W':
                area = dfs(i, j)
                if area > maxa:
                    maxa = area
    print(maxa)
```

马走日 dfs 路径数

```
def dfs(vert: Vertex, G: Graph, n: int, size: int):
    if n == size:
        return 1
    else:
        ans = 0
        for nbr in vert.neighbors:
            if not nbr.visited:
                nbr.visited = True
                ans += dfs(nbr, G, n+1, size)
                nbr.visited = False  # 恢复
        return ans
```

词梯

```
def build_graph(L: list):
    G = Graph()
    buckets = {}

    for word in L:
        for _ in range(4):
            bucket = word[:_] + '_' + word[_+1:]
            buckets.setdefault(bucket, []).append(word)

    for words in buckets.values():
        for i, word1 in enumerate(words):
            for word2 in words[i+1:]:
                G.add_edge(word1, word2)

    return G

def bfs(G: Graph, start: str, end: str):
    queue = deque()
    vert = G.vertlist[start]
    vert.visited = True
    queue.append(vert)

    while 1:
        if not queue:
            break
        vert = queue.popleft()
        if vert.key == end:
            return vert
        for nbr in vert.neighbors:
            if not nbr.visited:
                nbr.prev = vert
                nbr.visited = True
                queue.append(nbr)

    return False
```

其他

欧拉筛

```
def euler_sieve(maxn: int) -> list:
    primelist = []
    fltr = [True for _ in range(maxn + 1)]

    for i in range(2, maxn + 1):
        if fltr[i]:
            primelist.append(i)

            for p in primelist:
                if i * p > maxn:
                    break
                fltr[i * p] = False

        else:
            for p in primelist:
                if i * p > maxn:
                    break
                fltr[i * p] = False
                if i % p == 0:
                    break

    return primelist
```

二叉堆

```
class BinHeap(object):
    def __init__(self):
        self.arr = [0]
        self.size = 0

    def up_perc(self, i):
        while i > 1:
            if self.arr[i//2] > self.arr[i]:
                self.arr[i//2], self.arr[i] = self.arr[i], self.arr[i//2]
                i //= 2
            else:
                break

    def down_perc(self, i):
        while i <= self.size//2:
            if 2*i+1 > self.size:
                if self.arr[2*i] < self.arr[i]:
                    self.arr[2*i], self.arr[i] = self.arr[i], self.arr[2*i]
                    break
                else:
                    break
            else:
                if min(self.arr[2*i+1], self.arr[2*i]) < self.arr[i]:
                    if self.arr[2*i+1] <= self.arr[2*i]:
```

```
                        self.arr[2 * i + 1], self.arr[i] = self.arr[i],
self.arr[2 * i + 1]
                        i = 2*i+1
                    else:
                        self.arr[2 * i], self.arr[i] = self.arr[i], self.arr[2 *
i]
                        i *= 2
                else:
                    break

    def insert(self, num):
        self.arr.append(num)
        self.size += 1
        self.up_perc(self.size)

    def pop(self):
        if self.size > 1:
            popped = self.arr[1]
            self.arr[1] = self.arr.pop()
            self.size -= 1
            self.down_perc(1)
            return popped
        else:
            self.size -= 1
            return self.arr.pop()

n = int(input())
heap = BinHeap()
for _ in range(n):
    inp = input()
    if inp[0] == '1':
        heap.insert(int(list(inp.split())[-1]))
    else:
        print(heap.pop())
```

Huffman 编码

```
from heapq import heapify, heappop, heappush

class Node(object):
    def __init__(self, freq=float('inf'), left=None, right=None, value='猪'):
        self.value = value
        self.left = left
        self.right = right
        self.freq = freq

    def __str__(self):
        return self.freq

    def __lt__(self, other):
        return [self.freq, self.value] < [other.freq, other.value]

    def decode_map(self, prev=''):
```

```
        return {prev: self.value} if self.value != '猪' else
dict(list(self.left.decode_map(prev +
'0').items())+list(self.right.decode_map(prev + '1').items()))

    def encode_map(self, prev=''):
        return {self.value: prev} if self.value != '猪' else
dict(list(self.left.encode_map(prev +
'0').items())+list(self.right.encode_map(prev + '1').items()))



def build(dic: dict):
    heap = [Node(f, value=c) for c, f in dic.items()]
    heapify(heap)

    while len(heap)>1:
        node1 = heappop(heap)
        node2 = heappop(heap)
        heappush(heap, Node(node1.freq + node2.freq, node1, node2))

    return heappop(heap)

n = int(input())
dic = {char: int(freq) for char, freq in [input().split() for i in range(n)]}
root = build(dic)
encode = root.encode_map()
decode = root.decode_map()
code = set(decode.keys())

while 1:
    try:
        string = input()
        if string.isalpha():
            print(''.join([encode[char] for char in string]))
        else:
            output = ''
            temp = ''
            for char in string:
                temp += char
                if temp in code:
                    output += decode[temp]
                    temp = ''

            print(output)
    except EOFError:
        break
```

排序

```
def insertion_sort(arr):
    l = len(arr)
    for i in range(l):
        j = i
```

```
        while j>0 and arr[j] < arr[j-1]:
            arr[j], arr[j-1] = arr[j-1], arr[j]
            j -= 1

    return arr

def bubble_sort(arr):
    l = len(arr)

    for i in range(l-1):
        swapped = 0
        for j in range(0, l-i-1):
            if arr[j] > arr[j+1]:
                swapped = 1
                arr[j], arr[j+1] = arr[j+1], arr[j]
        if not swapped:
            break

    return arr

def selection_sort(arr):
    l = len(arr)

    for i in range(l-1):
        min_idx = i
        for j in range(i+1, l):
            if arr[j] < arr[min_idx]:
                min_idx = j

        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr

def merge_sort(arr):
    l = len(arr)
    if l > 1:
        mid = l // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i, j, k = 0, 0, 0
        while i<=mid-1 and j<l-mid:
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
                k += 1
            else:
                arr[k] = R[j]
                j += 1
                k += 1

        while i<=mid-1:
            arr[k] = L[i]
```

```
            i += 1
            k += 1
        while j <l-mid:
            arr[k] = R[j]
            j += 1
            k += 1

def quick_sort(arr, start, end):
    if start < end :
        mid = (start + end) // 2
        if arr[start] <= arr[mid] <= arr[end] or arr[end] <= arr[mid] <=
arr[start]:
            pivot = arr[mid]
            pivot_idx = mid
        elif arr[start] <= arr[end] <= arr[mid] or arr[mid] <= arr[end] <=
arr[start]:
            pivot = arr[end]
            pivot_idx = end
        else:
            pivot = arr[start]
            pivot_idx = start

        i, j = start, end
        while i < j:
            while (i < end and arr[i] < pivot) or i == pivot_idx:
                i += 1
            while (j > start and arr[j] >= pivot) or j == pivot_idx:
                j -= 1
            if i < j:
                arr[i], arr[j] = arr[j], arr[i]


        if arr[i] < pivot:
            arr[i], arr[pivot_idx] = arr[pivot_idx], arr[i]
            partition = i
        elif pivot_idx < i:
            arr[i-1], arr[pivot_idx] = arr[pivot_idx], arr[i-1]
            partition = i-1
        else:
            if arr[i] > pivot:
                arr[i], arr[pivot_idx] = arr[pivot_idx], arr[i]
            partition = i

        quick_sort(arr, start, partition-1)
        quick_sort(arr, partition+1, end)

def shell_sort(arr):
    l = len(arr)
    gap = l // 2

    while gap >= 1:
        for j in range(gap, l):
            for i in range(j, -1, -gap):
                if arr[i] < arr[i-gap] and i-gap >= 0:
                    arr[i], arr[i-gap] = arr[i-gap], arr[i]
                else:
```

```
                    break

        gap //= 2
```

Below is a table of comparison sorts. A comparison sort cannot perform better than *O*(*n* log *n*) on average.

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.

KMP

```
def kmp(s1: str, s2: str):
    l = len(s2)

    def get_next():
        next = [0 for _ in range(l + 1)]
        next[0] = -1
        next[1] = 0

        for j in range(2, l):
            k = j-1
            while k >= 0 and s2[next[k]] != s2[j-1]:
                k = next[k]
            next[j] = next[k] + 1

        return next

    next = get_next()

    i, j = 0, 0
    while i < len(s1) and j < l:
        if j == -1 or s1[i] == s2[j]:
            i += 1
            j += 1
        else:
            j = next[j]

    if j == l:
        return i - j
    else:
        return -1
```