Special Problems

Big number

Exercise 2.2.4.1: Compute the last non zero digit of 25!; can we use built-in data types?
Exercise 2.2.4.2: Check if 25! is divisible by 9317; can we use built-in data types?

Exercise 2.2.4.1:

方法：

1. **阶乘计算**：计算阶乘时，跳过所有 5 的倍数（因为它们与 2 的倍数相乘会产生尾数 0）。
2. **处理 5 的倍数**：对于 25!，我们知道其中包含了 5、10、15、20、25 这些 5 的倍数。每个这样的数都至少贡献一个额外的 5。特别地，25 是 5 的平方，因此它贡献了两个 5。
3. **调整结果**：由于我们跳过了所有的 5 及其倍数，在计算过程中，我们需要相应地调整每个被跳过的数字。由于每个 5 通常与 2 相乘以产生 10，我们可以通过除以 2 来平衡因跳过 5 而失去的因子。
4. **计算最后的非零数字**：在调整后的乘积中计算最后的非零数字。

```python
def last_non_zero_digit_of_factorial(n):
    result = 1
    for i in range(1, n + 1):
        if i % 5 != 0:
            result *= i
        result %= 10  # 仅保留个位数字

    # 考虑 5 的因子
    count_of_five = n // 5 + n // 25  # 包括 25 的额外 5
    while count_of_five:
        result *= 4  # 2 的平方，平衡跳过的每个 5
        result %= 10
        count_of_five -= 1

    return result

# 计算 25!
print(last_non_zero_digit_of_factorial(25))
```

Exercise 2.2.4.2

```python
def count_factors_in_factorial(n, p):
    count = 0
    i = p
    while n // i >= 1:
        count += n // i
        i *= p
    return count

def prime_factors(n):
    factors = []
    while n % 2 == 0:
        factors.append(2)
        n //= 2
    for i in range(3, int(n**0.5) + 1, 2):
        while n % i == 0:
            factors.append(i)
            n //= i
    if n > 2:
        factors.append(n)
    return factors

# 找出 9317 的质因数
factors = prime_factors(9317)
```

```
# 用于检查 25! 是否可被 9317 整除
n = 25
can_divide = True
for factor in set(factors):
    required_count = factors.count(factor)
    count_in_factorial = count_factors_in_factorial(n, factor)
    if count_in_factorial < required_count:
        can_divide = False
        break

if can_divide:
    print("Possible")
else:
    print("Impossible")
```

Exercise 2.2.4.1: Possible, keep the intermediate computations **modulo** 10^6. Keep chipping away the trailing zeroes (either none or a few zeroes are added after a multiplication from $n!$ to $(n+1)!$).

Exercise 2.2.4.2: Possible. $9317 = 7 \times 11^3$. We also list 25! as its prime factors. Then, we check if there are one factor 7 (yes) and three factors 11 (unfortunately no). So 25! is not divisible by 9317. Alternative: use modular arithmetic (see the details in Book 2).

Sort

a. Inversion Index

Inversion index problem is defined as follows: Given a list of numbers, count the minimum number of 'bubble sort' swaps (swap between pair of consecutive items) that are needed to make the list sorted in (usually ascending) order.

For example, if the content of the list is {3, 2, 1, 4}, we need 3 'bubble sort' swaps to make this list sorted in ascending order, i.e., swap (3, 2) to get {2, 3, 1, 4}, swap (3, 1) to get {2, 1, 3, 4}, and finally swap (2, 1) to get {1, 2, 3, 4}.

$O(n^2)$ solution

The most obvious solution is to count how many swaps are needed during the actual running of the $O(n^2)$ bubble sort algorithm, but this is clearly too slow.

$O(n \log n)$ solution

One better $O(n \log n)$ Divide and Conquer solution for this inversion index problem is to modify merge sort. During the merge process of merge sort, if the front of the right (sorted) sublist is taken first rather than the front of the left (sorted) sublist, we say that 'inversion occurs' and add inversion index counter by the size of the current left sublist (as *all* of the current left sublist have to be swapped with the front of the right sublist). When merge sort is completed, we report the value of this counter. As we only add $O(1)$ steps to merge sort, this solution has the same time complexity as merge sort, i.e., $O(n \log n)$.

Stack and Queue

bracket matching

```
stack = []
n = int(input())
s = input()
idx = 0
flag = True
d = {"}":"{", ")":"(", "]":"["}
while n:
        n-=1
        if s[idx] in set("{(["):
                stack.append(s[idx])
                idx += 1
                continue
        if s[idx] in set(")}]"):
                if len(stack) == 0 or stack[-1] != d[s[idx]]:
                        flag = False
                        break
                else:
                        stack.pop()
                idx += 1
print("Valid") if flag and not len(stack) else print("Invalid")
```

Graph

Exercise 4.2.6.1: The topological sort code shown above can only generate *one* valid topological ordering of the vertices of a DAG. What should we do if we want to output *all* (or count the number of) valid topological orderings of the vertices of a DAG?

1. **记录入度**：首先，对图中每个顶点的入度进行计数。入度是指指向该顶点的边的数量。
2. **选择顶点**：在每一步中，选择一个当前入度为0的顶点加入到当前的拓扑排序中。入度为0意味着没有其他顶点指向它，因此可以放心添加到排序中。
3. **递归**：在选择了一个顶点后，将它从图中移除，并更新其它顶点的入度（即减少与该顶点相连的顶点的入度）。然后递归地重复这个过程。
4. **回溯**：每当到达一种有效的拓扑排序（即所有顶点都被排序）时，记录下来或计数。然后进行回溯，撤销最后一步的选择，并恢复相应的入度，以探索其他可能的排序。
5. **继续搜索**：继续以上过程，直到探索所有可能的顶点组合。

```python
def all_topological_sorts(graph):
    # 计算所有顶点的入度
    in_degree = {u: 0 for u in graph}
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1
    # 用于存储所有可能的拓扑排序
    all_orders = []
    visited = {u: False for u in graph}
    # 递归函数来生成所有拓扑排序
    def dfs(order):
        # 如果找到一种排序，将其添加到结果中
        if len(order) == len(graph):
            all_orders.append(order[:])
            return
        # 尝试所有当前入度为0且未访问的顶点
        for u in in_degree:
            if in_degree[u] == 0 and not visited[u]:
                # 选择当前顶点并更新入度
                visited[u] = True
                order.append(u)
                for v in graph[u]:
                    in_degree[v] -= 1
                # 递归
                dfs(order)
                # 回溯
                visited[u] = False
                order.pop()
                for v in graph[u]:
                    in_degree[v] += 1
    dfs([])
    return all_orders

# 获取所有拓扑排序
orders = all_topological_sorts(graph)
for order in orders:
    print(order)
```

Exercise 4.2.6.1: One possible way is to modify the `toposort(u)` recursion into a *recursive backtracking* variant (see Section 3.2.2). We reset the `VISITED` flag of vertex u back to `UNVISITED` when we exit the recursion. This is an exponential (slow) algorithm.

Why exponential?

1. **重复访问**：每次递归返回时，通过将顶点状态重置为未访问，允许算法在后续的递归中重新访问这些顶点。这意味着算法可能会多次遍历同一顶点，尤其是在深度搜索中。
2. **搜索空间爆炸**：对于每个顶点，算法都需要探索所有可能的后续路径。在一个包含 V 个顶点的图中，可能的拓扑排序数量可以非常大（最坏情况下接近 $V!$，即 V 的阶乘），特别是当图中存在大量顶点且连接稀疏时。
3. **缺乏剪枝**：在普通的递归回溯算法中，通常会使用某种形式的剪枝来减少搜索空间。但在这种简单的回溯拓扑排序中，没有有效的剪枝机制，因此算法必须遍历其巨大的搜索空间中的每个分支。

- Question: A simple graph with V vertices is found out to be a Bipartite Graph. What is the maximum possible number of edges that this graph has?

在一个拥有 V 个顶点的简单图中，如果它是二分图，那么这个图最多可以有多少条边？

二分图是一种图，其顶点可以被划分为两个互不相交且独立的集合 U 和 V，使得每条边连接 U 中的一个顶点和 V 中的一个顶点。在二分图中，边的最大数量出现在完全二分图中，即 U 集合中的每个顶点都与 V 集合中的每个顶点相连。

假设这两个集合分别有 n 个和 m 个顶点，其中 $n + m = V$，V 是图中顶点的总数。那么，完全二分图中的最大边数 E 由两个集合中顶点数量的乘积给出：

$$E = n \times m$$

为了最大化 E，我们需要尽可能均匀地将顶点分配到 U 和 V 集合中，因为当 n 和 m 尽可能接近时，乘积 $n \times m$ 最大。这种分配取决于 V 是偶数还是奇数：

1. **如果 V 是偶数：** 最佳分配是每个集合中有 $V/2$ 个顶点，所以 $n = m = V/2$。因此，$E = (V/2) \times (V/2) = V^2/4$。
2. **如果 V 是奇数：** 一个集合将有 $\lfloor V/2 \rfloor$ 个顶点，另一个集合将有 $\lceil V/2 \rceil$ 个顶点。在这种情况下，$E = \lfloor V/2 \rfloor \times \lceil V/2 \rceil$。

通常来说，可以说一个拥有 V 个顶点的二分图最多可以有大约 $V^2/4$ 条边。这对于 V 的偶数和奇数值都是一个很好的估计。

Exercise 4.4.1.1*: Prove that the shortest path between two vertices u and v in a graph G that has no negative and no zero-weight weight cycle must be a *simple* path (acyclic)! What is the corollary of this proof?

在没有负权重和零权重环的图 G 中，证明两个顶点 u 和 v 之间的最短路径必须是一条简单路径（无环的)的理由如下：

1. 假设最短路径不是简单路径：如果最短路径不是简单路径，那么它至少包含一个环。我们可以进一步区分两种类型的环：正权重环和零权重环。
2. 正权重环：如果这个环是正权重的，那么去掉环中的任何部分都会减少路径的总权重，从而形成一条更短的路径。这与最初的假设（我们的路径是最短的）相矛盾。
3. 零权重环：如果环的总权重是零，那么去掉这个环同样不会增加路径的总权重，同时还会使路径变成简单路径。根据最短路径的定义，我们总是倾向于选择简单路径，因为即使权重相同，更简单的路径在实际应用中更为可取。

因此，我们可以得出结论：在没有负权重和零权重环的图中，两个顶点之间的最短路径必须是一条简单路径。

Exercise 4.4.3.5*: Dijkstra's algorithm (both variants) will run in $O(V^2 \log V)$ if run on a *complete* non-negative weighted graph where $E = O(V^2)$. Show how to modify Dijkstra's implementation so that it runs in $O(V^2)$ instead such complete graph! Hint: Avoid PQ.

在一个完全非负权重图中使用 Dijkstra 算法时，如果我们避免使用优先队列（Priority Queue, PQ），可以将算法的时间复杂度从 O(V^2 logV) 降低到 O(V^2)。这种修改主要是基于这样一个事实：在一个完全图中，每个顶点都与所有其他顶点相连，所以我们可以简化顶点选择的过程。

Dijkstra 算法的标准实现使用优先队列来每次选择当前距离起点最近的未访问顶点。在完全图中，由于每个顶点都直接连接到所有其他顶点，我们可以简单地通过遍历所有未访问的顶点来找到最近的顶点，而不需要维护一个优先队列。

修改后的 Dijkstra 算法步骤：

1. 初始化：对于图中的每个顶点，设置一个距离值，对于起始顶点设置为 0，其余所有顶点设置为无穷大。
2. 遍历顶点：遍历所有未访问的顶点，找到具有最小距离值的顶点。
3. 更新距离：对于通过当前选中的顶点可以到达的每个邻接顶点，更新其距离值。如果当前顶点到某个邻接顶点的距离加上从起始点到当前顶点的距离小于该邻接顶点的当前距离值，则更新该邻接顶点的距离值。
4. 标记已访问：将当前顶点标记为已访问。
5. 重复：重复步骤 2 至 4，直到所有顶点都被访问。

```
def dijkstra(graph, start):
    n = len(graph)  # 顶点数量
    visited = [False] * n
```

```
    distance = [float('inf')] * n
    distance[start] = 0

    for _ in range(n):
        # 找到未访问顶点中距离最小的顶点
        u = min((v for v in range(n) if not visited[v]), key=lambda v: distance[v])
        visited[u] = True

        # 更新所有邻接顶点的距离
        for v in range(n):
            if graph[u][v] and not visited[v]:
                new_distance = distance[u] + graph[u][v]
                if new_distance < distance[v]:
                    distance[v] = new_distance

    return distance
```

这个伪代码假设 `graph` 是一个邻接矩阵，其中 `graph[u][v]` 表示顶点 `u` 到顶点 `v` 的边的权重。通过避免使用优先队列，这个修改后的 Dijkstra 算法的时间复杂度为 $O(V^2)$。

Exercise 4.4.3.3: The source code for the Modified Dijkstra's algorithm shown above has this important check `if (d > dist[u]) continue;`. What if that line is removed? What will happen to the Modified Dijkstra's algorithm?

在提到的修改版 Dijkstra 算法中，检查 `if (d > dist[u]) continue;` 是非常重要的。这行代码的作用是为了确认当前从源点到达顶点 `u` 的距离 `d` 是否仍然是最短的。如果不是（即 `d > dist[u]`），那么算法就会跳过当前的迭代，继续寻找其他的路径。这是因为在这种情况下，已经找到了一条更短的路径到达顶点 `u`，所以当前的路径不再是最优的。

如果去掉这个检查 (`if (d > dist[u]) continue;`)，会有以下几个影响：

1. **效率降低**：算法会处理一些不必要的情况。即使已经找到了到达某个顶点的更短路径，算法仍然会考虑通过更长的路径到达该顶点的情况。这会增加算法的运行时间，因为它在做一些无用功。
2. **重复更新**：算法可能会不必要地重复更新某些顶点的最短路径。这不仅增加了计算量，而且在某些情况下可能导致错误的结果，尤其是在图中存在环路的情况下。
3. **潜在的错误结果**：在某些特定情况下，去掉这个检查可能导致算法无法找到真正的最短路径。这通常发生在图的结构较为复杂，尤其是存在多条路径到达同一顶点的情况下。