[The upper side of PAGE 6 IS NOT ARCHIVED]

B Simpler Questions (19 marks)

B.1 Sorting Question (9 marks)

In tutorial+lab 02, we discuss the performance of various sorting algorithms on ‘nearly sorted’ inputs. We also discussed that ‘nearly sorted’ can mean different things to different people. In this question, we will try to formalize this a little bit.

We say two elements in a list/array A, e.g., $A[i]$ and $A[j]$, form one inversion if $A[i] > A[j]$ and $i < j$. We say that list A that is already sorted in non-decreasing order has 0 inversion but list B that contains n distinct elements that is sorted in decreasing order has $n \times (n-1)/2$ inversions.

A nearly sorted list with n elements is closer to 0 inversion than $n \times (n-1)/2$ inversions. In this problem, we want to count the number of inversions of any input list A with n elements.

Propose an algorithm (and the associated data structure(s)) that is/are needed to solve this problem and analyze its time complexity in terms of n. To score up to 6 marks, your algorithm should be correct and runs in $O(n^2)$. To score full (9) marks in this section, your algorithm should be correct and runs in $O(n \log n)$. Hint: Modification of sorting algorithms that we have discussed in class.

B.2 Linked List Question (10 marks)

In class, we learn about Singly Linked List (SLL) where each vertex has 2 elements: the item/data/key and the pointer to the next vertex. We also remember two special vertices: the head/tail (the first/last element of the SLL, respectively). This way we can Insert at Head, Remove at Head, and Insert After Tail all in $O(1)$. For this SLL, Remove at Tail is $O(n)$ as we need to update the tail pointer to be the one *before* the current tail, yet we do not immediately have this information in an SLL.

In this question, we will *modify* the specification of an SLL implementation and ask what are the implications of implementing SLL that way. There are four modifications:

1. We introduce a new sentinel (dummy) vertex that contains no item/data/key,
2. We set the next of this sentinel vertex to be the actual first element of the SLL,
3. We drop the head pointer attribute, and
4. We set the next of tail element to be the sentinel vertex, effectively making the list circular.

Figure 1 shows one example of these modifications.

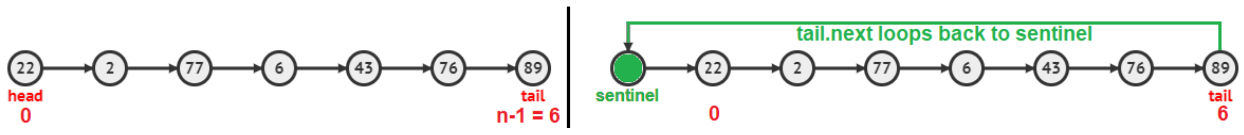

Figure 1: Left: SLL as discussed in class vs Right: Circular SLL with a Sentinel Vertex

Your task is to mention/list down as many implications/effects/changes (can be positive or negative) of this alternative SLL implication. One answer has been shown as an example. Marking scheme: 2 marks per additional correct implications, max at 10 marks.

0 To identify the head element, we need two steps, i.e., go the sentinel and advancing one time from the sentinel pointer, e.g., sentinel.next is the head element if there are more than 1 actual element in the list, whereas this is just one step if we have the head pointer.

1-5 Write your answers 1, 2, 3, 4, 5 in the answer sheet

1. The adding node is become unified, where adding a node whether at head or tail or middle need to change previous node's next attribute
2. The linked list becomes a cycle, we need a check statement to stop the loop when the iterator hit the dummy node while iterating the linked list.
3. Check whether a list is empty is more complicated, now we need to check the dummy node's next is pointing to itself, since the head pointer is removed.
4. Find tail node is more complicated, since there is no tail pointer, now need to go through the whole list.
5. Add a dummy head will increase the memory cost.

C Applications (14+16+15 = 45 marks)

C.1 Cryptic Naming (14 marks)

You code in a peculiar way as your variable names are always cryptic v_1, v_2, v_3, etc. Sometimes you could not declare a new variable called v_x because you have used that variable name v_x earlier. When that happens you tries the names $v_{2\times x}$, $v_{3\times x}$, $v_{4\times x}$, etc one by one, until you find a variable name that is not already taken.

In this question, let's simulate this peculiar variable naming strategy. First, you are given an integer n that denotes the number of variables that you will create one by one. Then, there will be n integers in the next n lines. Each of this integer x ($1 \leq x \leq 10^4$) means that you want to create variable v_x. Determine what variable names that actually declared.

For example, $n = 7$ (that is, 7 variables to be declared) and the variable names to be declared are $\{v_1, v_3, v_1, v_2, v_1, v_5, v_5\}$, one after another. Then these are the variables that are actually created using the peculiar variable naming strategy above.

No	Initially	Actually	Explanation
1	v_1	v_1	v_1 is available
2	v_3	v_3	v_3 is available
3	v_1	v_2	v_1 is not available, creating $v_{2\times 1} = v_2$ instead
4	v_2	v_4	v_2 is not available, creating $v_{2\times 2} = v_4$ instead
5	v_1	v_5	v_1, v_2, v_3 and v_4 are not available, creating $v_{5\times 1} = v_5$ instead
6	v_5	v_{10}	v_5 is not available, creating $v_{2\times 5} = v_{10}$ instead
7	v_5	v_{15}	v_5 and v_{10} are not available, creating $v_{3\times 5} = v_{15}$ instead

V8: v8 is available
V4: v4 is available
V4: v4 is not available, create v5*1=v5 instead
V2: v2 is available
V2: v2 is not available, create v3*1=v3 instead

C.1.1 One Manual Test Case (2 marks)

If you have understood this question, what is the answer if you are given $n = 5$ and you want to declare these 5 variables $\{v_8, v_4, v_4, v_2, v_2\}$ one after another. Follow the format shown above.

C.1.2 Solve This Problem (12 marks)

Propose an algorithm (and the associated data structure(s)) that is/are needed to solve this problem and analyze its time complexity. To score up to 6 marks, your algorithm should be correct and fast enough for $n \leq 10^3$. To score full (12) marks in this section, your algorithm should be correct for $n \leq 10^5$ — Notice the major gap between the two and hence the major gap of marks.

C.2 Grid Expansion (16 marks)

This problem is on a 2D grid. You are given the initial state of an alien bacteria on an *initially small* grid (of not more than 20×20, see the 2 '#'s on the leftmost diagram below). On each day, if any bacteria part occupies cell (r, c), that bacteria part will expand to *all eight* neighboring cells $(r \pm 1, c \pm 1)$ and still occupy cell (r, c). For example, see the middle/right diagram below to see what the bacteria looks like after one day/two days, respectively.

```
Initial 6x9             After one day           After two days
(2 # cells)             (14 cells)              (34 cells)
.........               .........               .22222...
.........               ..111....               .211122..
...#.....               ..1#11...               .21#112..
....#....               ..11#1...               .211#12..
.........               ...111...               .221112..
.........               .........               ..22222..
```

Important note: By now you should realize that the bacteria will grow quite fast. Although the input is given as a small finite grid (of not more than 20×20) initially, the bacteria can grow *beyond* these finite boundaries. For example, if the bacteria is left for additional two more days, we will have:

```
After two days          After three days        After four days
(34 cells)              (62 cells)              (98 cells)
(as above)              (bigger than 6x9)      444444444. (bigger than 6x9)
                        3333333..              4333333344
.22222...               32222233.              4322222334
.211122..               32111223.              4321112234
.21#112..               321#1123.              4321#11234
.211#12..               3211#123.              43211#1234
.221112..               32211123.              4322111234
..22222..               33222223.              4332222234
                        .3333333.              4433333334
                                               .444444444
```

C.2.1 One Manual Test Case (2 marks)

If you have understood this question, what is the state of the grid if you are initially given this 4×11 grid where a '#' denotes the initial cells occupied by the bacteria parts (there are 5 cells initially occupied) and leave the grid for 3 days. To convince the grader that the answer is not a random guess, simulate the daily growth of the bacteria as shown in the example earlier (use characters 1, 2, 3) and then mention the number of cells occupied by the bacteria parts after 3 days.

```
...#...#...
...........
....#.#....
.....#.....
```

略

C.2.2 What Is This Problem? (2 marks)

The small case of this problem (2 or more '#'s — up to 20×20 — and only $k \leq 20$ days) is a *variation* to one of the problem that we have discussed in class. What is the name of this problem?

Amoeba problem in Kattis. Can use BFS to solve

C.2.3 Special Cases (2 marks)

There are two simple mathematical formulas to count the answer if there are only 0 '#' or 1 '#' in the initial grid and you leave this grid for k days. What are they?

For 0 '#' and k days, the count is 0
For 1'#' and k days, the count is (2k+1)^2

C.2.4 Solve This Problem (10 marks)

Propose an algorithm (and the associated data structure(s)) that is/are needed to solve this problem (you just need to count the number of cells occupied by bacteria parts) and analyze its time complexity. To score up to 5 marks in this subsection, your algorithm should be correct for just $k \leq 20$ days of simulation. To score full (10) marks in this section, your algorithm should be correct for $k \leq 10^6$ days — Notice the major gap between the two and hence the major gap of marks.

Below 20 days, use BFS to count, above 20 days - 10^6 days,
需要计算一个可以包含所有点的最小矩形，减去未占领的点

C.3 Jumping Game (15 marks)

Given an $n \times n$ matrix where each cell contains an integer between 1 to k (duplicate integers may exist), you want to play a jumping game on it. First, you start on *any* cell in the matrix with integer 1, then jump to *any* cell with integer 2, then 3 and so on until you reach any cell with integer k. You can only visit exactly one cell for each integer 1 to k. You can assume that you can go from any cell (x_1, y_1) to any other cell (x_2, y_2) in the matrix (with Manhattan distance of $|x_1 - x_2| + |y_1 - y_2|$), as long as the consecutive numbering rule is satisfied, i.e., the integer on (x_1, y_1) is one less than the integer on (x_2, y_2).

Your task is to compute the shortest possible total distance that you can make to complete this game. However, if it is not possible because the matrix is missing one or more integers between 1 to k, you have to output -1 instead.

Example 1: If you are given the following 4×4 matrix and $k = 16$, then the answer is 15 by spiraling through the 16 integers.

```
 1  2  3  4    the path is 1->2->...->15->16 that spirals through the matrix
12 13 14  5
11 16 15  6
10  9  8  7
```

Example 2: If you are given the following 4×4 matrix and $k = 7$, then the answer is 8.

```
*1 *2 *3  7     the path is 1->2->3->4->5->6->7
 4 *4 *5  5                  1  1  2  1  2  1
 3  6  5 *6     there can be other valid path with cost 8
 4  5  7 *7
```

Example 3: Same 4×4 matrix as above, but $k = 8$, then the answer is -1.

```
 1  2  3  7     there is no integer 8 in the matrix
 4  4  5  5
 3  6  5  6
 4  5  7  7
```

C.3.1 One Manual Test Case (2 marks)

If you have understood this question, what is the answer if you are given this test case: 7×7 matrix, $k = 5$. To convince the grader that the answer is not a random guess, highlight the jumping path (using * or just draw the path).

```
1 3 4 2 4 2 1
5 3 4 1 5 3 1
2 4 1 5 4 5 2
2 1 5 5 3 5 2
5 2 3 2 3 1 5
4 2 4 2 2 4 4
5 1 1 2 5 4 1
```

C.3.2 What Is This Problem? (2 marks)

This problem is a *variation* to one of the problem that we have discussed in class. What is the name of this problem?

C.3.3 Solve This Problem (11 marks)

Propose an algorithm (and the associated data structure(s)) that is/are needed to solve this problem and analyze its time complexity. To score up to 9 marks in this subsection, your algorithm should be correct and runs in $O(n^4 \log n)$. To score full (11) marks in this section, your algorithm should be correct and runs in $O(n^4)$. The $O(\log n)$ gap is not that much hence the minor difference in marks.

```python
# Question B.1
def countInversions(A, left, right):
    # 如果分到最后，数组不可分割时，返回0
    if left >= right:
        return 0
    mid = (left + right) // 2
    inv_count = 0
    # 左半部分继续分割
    inv_count += countInversions(A, left, mid)
    # 右半部分继续分割
    inv_count += countInversions(A, mid+1, right)
    #
    inv_count += mergeCount(A, left, mid, right)
    return inv_count

def mergeCount(A, left, mid, right):
    temp = []
    inv_count = 0
    i, j = left, mid+1
    while i<=mid and j<=right:
        if A[i] <= A[j]:
            temp.append(A[i])
            i += 1
        else: # invasion occur
            temp.append(A[i])
            # 计算逆序对，数量为从i到mid的每个索引都与j形成一个逆序对
            inv_count += mid-i+1
            j += 1
    # 复制剩余的元素到temp
    while i<=mid:
        temp.append(A[i])
        i += 1
    while j<=right:
        temp.append(A[j])
        j += 1
    for k in range(left, right+1):
        A[k] = temp[k-left]
    return inv_count
```

```python
# C1.2
def solution(n, names):
    declared_variables = {}  # Dictionary to keep track of variable counts
    actual_names = []  # List to store the actual declared variable names

    for x in names:
        # Check if this base name has been used before
        if x in declared_variables:
            # 读取乘数
            multiplier = declared_variables[x]
            # 每次x没有找到对应的空闲插槽，乘数+1
            while x*multiplier in declared_variables:
                multiplier += 1
            declared_variables[x*multiplier] = 1
            declared_variables[x] = multiplier
            actual_names.append(x*multiplier)
        else:
            declared_variables[x] = 1
            actual_names.append(x)

    return actual_names

# Example case
n = 7
names = [1, 3, 1, 2, 1, 5, 5]
solution(n, names)
```

```python
# C3
import sys
def solution():
    def manhattan_distance(loc1, loc2):
        return abs(loc1[0] - loc2[0]) + abs(loc1[1] - loc2[1])

    # n for matrix size, k for dest number
    n, k = map(int, input().split())
    matrix = [list(map(int, sys.stdin.readline().split())) for _ in range(n)]

    # Each number's location
    dict = {i: [] for i in range(1, k + 1)}
    for i in range(n):
        for j in range(n):
            # If the number in matrix is in the range we possibly hop to
            if matrix[i][j] in dict:
                # Append to dict
                dict[matrix[i][j]].append((i, j))
    # If missing a required number, return -1
    for i in range(1, k + 1):
        if not dict[i]:
            print(-1)
            return

    dp = [[float('inf')] * (n*n) for _ in range(k+1)]
    for (x, y) in dict[1]:
        dp[1][x*n+y] = 0

    # Dynamic Programming
    for number in range(2, k+1):
        for current in dict[number]:
            for previous in dict[number-1]:
                distance = manhattan_distance(previous, current)
                current_i = current[0]*n + current[1]
                previous_i = previous[0]*n + previous[1]
                dp[number][current_i] = min(dp[number][current_i], dp[number-1][previous_i]+distance)
    minimum = min(dp[k])

    print(minimum)
    return
solution()
```